Technische Universität München
Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen

Cluster-Based Parallelization of Simulations on Dynamically Adaptive Grids and Dynamic Resource Management

Martin Schreiber

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des Akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Florian Matthes
Prüfer der Dissertation:	Univ.-Prof. Dr. Hans-Joachim Bungartz
	Univ.-Prof. Christian Bischof, Ph. D.

Die Dissertation wurde am 30.01.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11.05.2014 angenommen.

Contents

I Introduction 9

II Essential numerics of hyperbolic PDEs 13

1 Continuity equation and applications 15
1.1 Continuity equation . . . 15
1.2 Examples of hyperbolic systems . . . 16

2 Discontinuous Galerkin discretization 19
2.1 Grid generation . . . 19
2.2 Triangle reference and world space . . . 20
2.3 Basis functions . . . 22
2.4 Weak formulation . . . 23
2.5 Mass matrix $\mathcal{M}$ . . . 24
2.6 Stiffness matrices $\mathcal{S}$ . . . 24
2.7 Flux matrices $\mathcal{E}$ . . . 25
2.8 Source term . . . 26
2.9 Rotational invariancy and edge space . . . 26
2.10 Numerical flux $\mathcal{F}$ . . . 27
2.11 Boundary conditions . . . 28
2.12 Adaptive refining and coarsening matrices $\mathcal{R}$ and $\mathcal{C}$ . . . 29
2.13 CFL stability condition . . . 33
2.14 Time stepping schemes . . . 33

III Efficient framework for simulations on dynamically adaptive grids 35

3 Requirements and related work 37
3.1 Simulation: grid, data and communication management . . . 37
3.2 HPC requirements . . . 38
3.3 Space-filling curves . . . 39
3.4 Related work . . . 39

4 Serial implementation 43
4.1 Grid generation with refinement trees . . . 44
4.2 Stacks . . . 45
4.3 Stack-based communication . . . 45
4.4 Classification of data lifetime . . . 53
4.5 Stack- and stream-based simulation on a static grid . . . 54
4.6 Adaptivity . . . 58
4.7 Verification of stack-based edge communication . . . 61
4.8 Higher-order time stepping: Runge-Kutta . . . 65
4.9 Software design, programmability and realization . . . 66
4.10 Optimization . . . 70
4.11 Contributions . . . 76

5 Parallelization **79**
5.1 SFC-based parallelization methods for DAMR 81
5.2 Inter-partition communication and dynamic meta information 84
5.3 Parallelization with clusters 90
5.4 Base domain triangulation and initialization of meta information 95
5.5 Dynamic cluster generation 96
5.6 Shared-memory parallelization 106
5.7 Results: Shared-memory parallelization 110
5.8 Cluster-based optimization 115
5.9 Results: Long-term simulations and optimizations on shared-memory 124
5.10 Distributed-memory parallelization 124
5.11 Hybrid parallelization 129
5.12 Results: Distributed-memory parallelization 131
5.13 Summary and Outlook 133

6 Application scenarios **135**
6.1 Prerequisites 135
6.2 Analytic benchmark: solitary wave on composite beach 138
6.3 Field benchmark: Tohoku Tsunami simulation 143
6.4 Output backends 148
6.5 Simulations on the sphere 151
6.6 Multi-layer simulations 152
6.7 Summary and outlook 154

IV Invasive Computing **155**

7 Invasive Computing with invasive hard- and software **157**
7.1 Inavsive hardware architecture 157
7.2 Invasive software architecture 158
7.3 Invasive algorithms 159
7.4 Results 161

8 Invasive Computing for shared-memory HPC systems **163**
8.1 Invasion with OpenMP and TBB 163
8.2 Invasive client layer 165
8.3 Invasive resource manager 167
8.4 Scheduling decisions 170
8.5 Invasive programming patterns 173
8.6 Results 174

9 Conclusion and outlook **179**

V Summary **181**

A Appendix **183**
A.1 Hyperbolic PDEs 183
A.2 Test platforms 187

Acknowledgements

First of all, I like to thank my supervisor Hans-Joachim Bungartz for supporting and encouraging me during this work. I'm also grateful to many colleagues involved in the feedback process of this thesis and like to mention all of them: Alexander Breuer, Benjamin Peherstorfer, Christoph Riesinger, Konrad Waldherr, Philipp Neumann, Sebastian Rettenberger, *Tobias Neckel*, Tobias Weinzierl. Thank you all for your valuable feedback! My special thanks goes to my family for their ongoing support.

The software development of this thesis is based on many open-source products and I like to thank all developers who published their code as open source. For sake of reproducibility, the source code related to this thesis as well as the related publications is also available at one the following URLs:
`http://www5.in.tum.de/sierpinski`
`http://www.martin-schreiber.info/sierpinski`

Abstract

The efficient execution of numerical simulations with dynamically adaptive mesh refinement (DAMR) belongs to the major challenges in high performance computing (HPC). With simulations demanding for steadily changing grid structures, this imposes efficiency requirements on handling grid structure and managing connectivity data. Large-scale HPC systems furthermore lead to additional requirements such as load balancing and thus data migration on distributed-memory systems, which are non-trivial for simulations running with DAMR.

The first part of this thesis focuses on the optimization and parallelization of simulations with DAMR. Our dynamic grid generation approach is based on the Sierpiński space-filling curve (SFC). We developed a novel and efficient parallel management of the grid structure, simulation data and dynamically changing connectivity information, and introduced the cluster concept for grid partitioning. This cluster-based domain decomposition directly leads to efficient parallelization of DAMR on shared-, distributed- as well as hybrid-memory systems, and further yields optimization methods based on such a clustering.

The second part of this work is on optimization of HPC parallelization models currently assigning compute resources statically during the program start. This yields a perspective for dynamic resource distribution addressing the following issues: First, static resource allocation restricts starting other applications in case of insufficient resources available at program start. Second, changing efficiency of applications with different scalability behaviour is not considered. We solve both issues with a resource manager based on Invasive Computing paradigms, dynamically redistributing resources to applications aiming at higher application throughput and thus efficiency.

For several executions of simulations based on our DAMR, we are now able to redistribute the computation resources dynamically among concurrently running applications on shared-memory systems. With dynamic resource assignment, this results in improved throughput and thus higher efficiency.

PART I

INTRODUCTION

Computational fluid dynamics (CFD) play an important role in a variety of disciplines. They are essential in fields such as car manufacturing, weather forecasting and combustion-engine design where they are used to optimize the aerodynamic behavior of car designs, to predict the weather and to understand processes inside an engine, respectively.

In this work, we focus on CFD simulations that are based on models described by partial differential equations (PDE), in particular by *hyperbolic* PDEs such as the shallow water and Euler equations. These hyperbolic PDEs model wave-propagation dominated phenomena and cannot be solved analytically in general. Therefore, these equations have to be discretized, which typically leads to a grid on which the scenario is computed. However, these simulations have a high computational intensity and, hence, require high-performance computing (HPC) systems.

Implementing such simulations with a static and regularly refined grid perfectly fits to the HPC parallel architectures and well-known parallelization techniques are applicable. Despite achieving high scalability, the time-to-solution for wave-dominated CFD simulations on regularly resolved domains is non-optimal. Considering a simulation of a wave propagation, such regular grids result in a very high resolution in the entire simulation domain despite the fact that the wave is propagating only in parts of the domain. Hence, many computations are invested in grid areas without a large contribution to the numerical solution.

With grids refining and coarsening during run time, this seems to be an easy-to-accomplish approach if only considering a non-parallel implementation and using standard computer science algorithms. However, further requirements for an efficient grid traversal and parallelization on state-of-the-art HPC systems lead to additional challenges with the major ones briefly summarized here:

(a) *Memory-bandwidth and -hierarchy awareness*:
As memory access is considered to be one of the main bottlenecks for the next decades of computing architectures, the algorithms should aim at reducing the memory footprint, e.g. during grid traversals. Regarding the data exchange between grid cells, the grid traversal should focus on reaccessing data which is already stored on higher cache levels.

(b) *Parallelization models for shared- and distributed-memory systems*:
For current HPC architectures, a cache-coherency is available within each compute node, hence supporting shared-memory parallelization models. However, a cache-coherency of memory access among all compute nodes is not feasible and demands for distributed-memory parallelization models. Furthermore, for an efficient parallelization on accelerators such as XeonPhi, shared- as well as distributed-memory parallelization models are mandatory for efficiency reasons.

(c) *Load-balancing with dynamically changing grids*:
Since the grid is refined and coarsened over the simulation's runtime, load imbalances are created which lead to idle times and hence reduced efficiency. To avoid such idle times on large-scale systems, a load-balancing scheme is required to compensate these idle times.

(d) *Usability*:
Considering all the aforementioned HPC aspects (a)-(c), it is in general challenging to keep a sufficient usability which hides the complexity of the traversals of a dynamically changing grid, the utilization of both parallelization models and the load-balancing approach from the application developer.

Former work on the execution of simulations with stack- and stream-based algorithms already provides a solution for the memory-awareness issues (a). However, a scalable and efficient parallelization (b) with both parallelization models (c) is still subject of research and with a new parallelization approach also its usability (d).

We first give a basic introduction to the discontinuous Galerkin discretization of wave-propagation-dominated models in Part II. Based on this knowledge, we present our framework and the cluster-based parallelization in Part III. Simulations on a dynamically changing grid lead to a changing workload, hence also profit from dynamically changing resources on which we focus on Part IV for concurrently executed simulations.

Content overview

Part II: Essential numerics of hyperbolic PDEs
For the development of our framework, we first *determine the interface requirements* with representative standard models for wave propagations. Here, we consider the shallow water equations (which are depth-averaged Navier-Stokes equations) and the Euler equations as two representatives of the fundamental equations in fluid mechanics. For the discretization, we use the discontinuous Galerkin method which is well-researched for the considered models. Afterwards, we determine data access and data exchange patterns of this model.

Part III: Efficient framework for simulations on dynamically adaptive grids
We start with an introduction to existing work on stack- and stream-based simulations based on SFC-induced grids. These stacks and streams account for issue (a). This is followed by a verification of the correct stack communication and extensions to higher-order time-stepping methods. For usability reasons (d), we separate the grid traversals and the kernels operating on the grid data. Based on the underlying grid traversal, we present further optimizations.

Then, we introduce our new parallelization approach which is based on a partitioning of the domain into intervals of the SFC. We derive properties of the SFC-based stack communication system which shows the validity of a run-length encoded (RLE) communication scheme, yielding a cluster-based parallelization approach. This RLE communication scheme yields several advantages, e.g. the possibility of implicitly updating the meta information based on transferred adaptivity markers and block-wise communication resulting in shared- and distributed-memory support (c). In combination with the stack-based communication system, clustering directly leads to an efficient data migration for distributed-memory systems (d). We close the parallelization section with cluster-based optimizations and scalability results for simulations on dynamically adaptive grids which are parallelized with OpenMP and TBB for shared- and MPI for distributed-memory parallelization models.

The scalability benchmarks show the efficiency of the parallelization, whereas we tested the real applicability of our dynamically adaptive simulations with analytical benchmarks and a realistic Tsunami simulation. Furthermore, we present other implemented application scenarios such as interactive steering with an OpenGL back end, efficient writing of grid data to persistent memory, simulations on the sphere and multi-layer simulations.

Part IV: Invasive Computing

Once running simulations on dynamically adaptive grids, the workload can change over the simulation's run time, hence also the resource requirements vary. For state-of-the-art parallelization models in high-performance computing, such a change in resource requirements is not considered so far for concurrently running applications. Here, a dynamical redistribution of resources would have the potential of improving the overall system's efficiency.

In Part IV, we evaluated such a dynamic resource management on shared-memory HPC systems with the Invasive Computing paradigm. We realized this with an Invasive resource manager which receives application-specific information to optimize the distribution of the computational resources among concurrently running applications. Several experiments have been conducted with this resource manager based on our simulations on dynamically adaptive grids developed in the previous part of this thesis.

PART II

ESSENTIAL NUMERICS OF HYPERBOLIC PDES

This part provides the fundamental basics behind simulations of hyperbolic partial differential equations (PDEs) and their numerical discretization. These aspects are relevant for taking appropriate decisions concerning the development of the framework discussed in Part III of this thesis. Also closing the gap between a theoretical mathematical formulation of a model and its concrete implementation can only be achieved by a deep knowledge of the requirements set up by the problem to be solved.

Chapter 1: Continuity equation and applications
We start with an introduction to the governing equation to provide a better understanding of the underlying system of equations. For a concrete applicability of our framework, we introduce the shallow water and the Euler equations which are numerically solved by the discretization and solvers presented in the next chapter.

Chapter 2: Discontinuous Galerkin discretization
The basic discretization for higher-order discontinuous Galerkin (DG) methods on triangular grids is presented, followed by refinement and coarsening projections for dynamic adaptivity.

1 Continuity equation and applications

Mathematical descriptions of scenarios should conserve particular quantities. We use the continuity equation as the underlying equation, which conserves these quantities for our considered simulations and start with a short derivation of this equation. This is followed by presenting two typical applications which are used in this work and introduce a notation for the conserved quantities. These conserved quantities are then successively discretized in the upcoming chapters.

1.1 Continuity equation

As a starting point for the derivation of the continuity equation, we consider a one-dimensional simulation scenario (see Fig. 1.1) placed at an infinitesimal small interval of size dx. The quantities we are interested in are given continuously by $\hat{U}(x,t) \in \mathbb{R}^n$ for n conserved quantities at position x at time t. Furthermore a so-called flux function $F(\hat{U}(x,t)) : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is given. Its values account for the change of the conserved quantities at a particular position $x \in \mathbb{R}$ for the current state $\hat{U}(x,t)$ at time $t \in \mathbb{R}$. To give an example, we consider a single simulated quantity, such as the gas density or the water height, only. Based upon an advection model assuming a constant advection speed v_c of conserved quantity, then, the flux function is given by $F(\hat{U}(x,t)) := v_c\, \hat{U}(x,t)$.

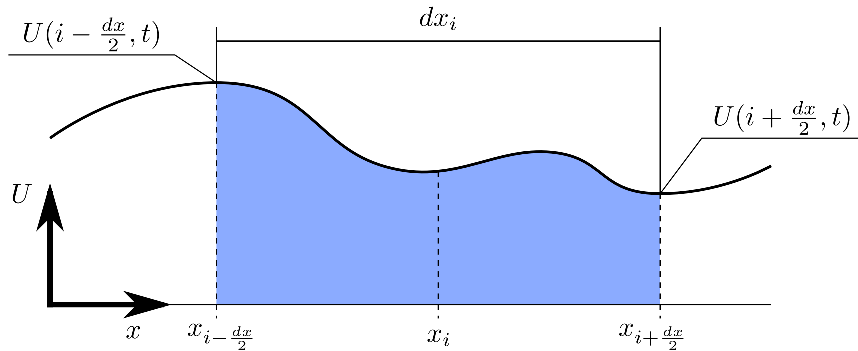


Figure 1.1: Conservation of quantities in an infinitesimal small one-dimensional shallow water simulation interval.

A sketch of a single interval $I^{(i)} := [x_i - \frac{dx}{2}; x_i + \frac{dx}{2}]$ of our model is given in Figure 1.1. In each interval, we consider the *average* quantities $\hat{U}^{(i)}(t) := \frac{1}{dx} \int_{I^{(i)}} \hat{U}(x,t)dx$. We further should track the incoming and outgoing quantities at the interval boundaries $dI^{(i)} := \{x_i - \frac{dx}{2}, x_i + \frac{dx}{2}\}$ with the change of quantities given by the flux function which is evaluated at the corresponding boundary points. Assuming a change in conserved quantities $\hat{U}^{(i)}$ in an interval with a mesh size dx in space and time-step size dt, we describe the change of the system by the change on

the boundaries and get

$$\hat{U}^{(i)}(t+dt)dx = \hat{U}^{(i)}(t)dx + dt\left(F(\hat{U}(x_i - \tfrac{dx}{2}, t)) - F(\hat{U}(x_i + \tfrac{dx}{2}, t))\right). \tag{1.1}$$

The left side of Equation (1.1) represents the average of conserved quantities in the interval $I^{(i)})$ at the next time step whereas the right side represents the old state and the change of this state over time via the flux at both interval boundaries.

These terms can be rearranged to a forward finite differencing scheme, yielding

$$\frac{\hat{U}^{(i)}(t+dt) - \hat{U}^{(i)}(t)}{dt} = \frac{F(\hat{U}(x_i - \frac{dx}{2}, t)) - F(\hat{U}(x_i + \frac{dx}{2}, t))}{dx}. \tag{1.2}$$

Assuming that the solution $\hat{U}$ is sufficiently smooth, the limits $dt \to 0$ and $dx \to 0$ of (1.2) then gives the one-dimensional continuity equation

$$\hat{U}_t(x,t) + F_x(\hat{U}(x,t)) = 0 \tag{1.3}$$

with $\hat{U}$ as the set of conserved quantities and the subscript x and t used as an abbreviation for the derivative w.r.t. to space and time.

We further add a source term $S(\hat{U}(x,t))$ at the right side of the Eq. (1.3). This source term is frequently used to account for non-homogeneous effects such as external forces, changing bathymetry, etc. An extension to two and three dimensions is directly given by including flux functions with a derivative with respect to y and z. For flux functions based on two-dimensional conserved quantities such as the momentum, we use the notation G for the flux function derived to x and H derived to y

$$\hat{U}_t(x,y,t) + G_x(\hat{U}(x,y,t)) + H_y(\hat{U}(x,y,t)) = S(\hat{U}(x,y,t)). \tag{1.4}$$

Using the gradient $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y})^T$ and $F(\hat{U},t) = G(\hat{U},t)(1,0)^T + H(\hat{U},t)(0,1)^T$, this equation can be written compactly with the two-dimensional equation of conservation

$$\hat{U}_t(x,y,t) + \nabla \cdot F(\hat{U}(x,y,t)) = S(\hat{U}(x,y,t)). \tag{1.5}$$

1.2 Examples of hyperbolic systems

Up to now, the governing equation was derived using the conserved quantities U, the flux function F and the source term S. This section concretizes these terms by presenting two different scenarios: simulations assuming a shallow water, based on the shallow water equations, and compressible gas simulations neglecting the viscosity, based on the Euler equations.

1.2.1 Shallow water equations

Simulation of water with free surfaces is mainly considered to be a three-dimensional problem. Using depth averaged Navier-Stokes equations (see e.g. [Tor01]), this three-dimensional problem can be reformulated to a two-dimensional problem under special assumptions. Among others, vertical moving fluid is assumed to be negligible. This leads to the so-called shallow water equations (SWE) giving a description of the water by its water surface height h, the two-dimensional velocity components $(u,v)^T$ and the bathymetry b.

Several ways of a mathematical representation of these quantities are used such as specifying the water surface relative to the horizon or to the bathymetry. We use the shallow water formulation storing the bathymetry b relative to the steady state of the water and the water

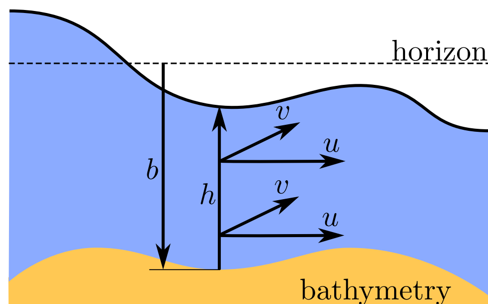


Figure 1.2: Sketch of shallow water equation with the conserved quantities at a particular spatial position. The bathymetry b is specified relative to the horizon, the quantity h represents the depth-averaged height and (hu, hv) the momentum of the fluid. Since we work with depth-averaged values, (hu, hv) is equal over the entire height at position x

surface height h relative to the bathymetry [LeV02,BSA12]. For a sea without waves and tidals, b is positive for terrain and negative for the sea floor.

By using the velocity components as conserved quantities, this is considered not to be a momentum conserving formulation. Thus, instead of storing the velocity components, we store both momentums hu and hv as the conserved quantities. This leads to the tuple of conserved quantities

$$U := (h, hu, hv, b)^T$$

for each nodal point with a sketch in Fig. 1.2. Despite that hu and hv are frequently described as momentum, we should be aware that h actually describes the height of a volume above a unit area. Therefore, this momentum may not be seen as a momentum per cell, but rather as momentum per unit area.

Using this form based on the conserved quantities and assuming a constant bathymetry without loss of generality, we formulate the flux functions G and H (see Eq. (1.4)):

$$G(U) := \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix} \qquad H(U) := \begin{pmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{pmatrix}$$

For the flux computations (see Section 2.10), we need the eigenvalues of the Jacobian $\frac{dG(U)}{dU}$. We start by substituting all v with $\frac{(hv)}{h}$ and u with $\frac{(hu)}{h}$. This yields

$$G(U) = \begin{pmatrix} hu \\ \frac{(hu)^2}{h} + \frac{1}{2}gh^2 \\ \frac{(hu)(hv)}{h} \end{pmatrix}. \tag{1.6}$$

By computing the derivative with respect to each of the conserved quantities (h, hu), this yields

$$\frac{dG(U)}{dU} = \begin{pmatrix} 0 & 1 & 0 \\ -u^2 + gh & 2u & 0 \\ -uv & v & u \end{pmatrix}. \tag{1.7}$$

Finally, the sorted eigenvalues of the Jacobian of G are given by

$$\left| \frac{dG(U)}{dU} \right| = \begin{pmatrix} u - \sqrt{gh} \\ u \\ u + \sqrt{gh} \end{pmatrix} \tag{1.8}$$

which are required for the computation of the maximum time step size.

1.2.2 Euler equations

For compressible flow with a negligible viscosity, the Navier-Stokes equations can be simplified to the so-called Euler equations which belongs to one of the standard models used for fluid simulations. For this work, we consider the two-dimensional Euler equations with the conserved quantities given by $U := (\rho, \rho u, \rho v, E)$ with ρ being the fluid density, $(u, v)^T$ the velocity components and E the energy. The similarities to the shallow water equations are clearly visible by considering the density instead of the height of a water surface. The flux terms G and H are then given with

$$G(U) := \begin{pmatrix} \rho u \\ p + \rho u^2 \\ \rho uv \\ u(E+p) \end{pmatrix} \qquad H(U) := \begin{pmatrix} \rho v \\ \rho uv \\ p + \rho v^2 \\ v(E+p) \end{pmatrix} . \tag{1.9}$$

In this terms, the pressure p depends on the type of gas being simulated. For our simulations, we only consider an isothermal gas with

$$p = \rho a^2$$

and a constant a specific to the simulated gas (see [LeV02], page 298). For the computation of the wave speed, the Jacobian G is given by

$$\frac{dG(U)}{dU} := \begin{pmatrix} 0 & 1 & 0 & 0 \\ -u^2 + a^2 & 2u & 0 & 0 \\ -uv & v & u & 0 \\ a^2 u - \frac{(E+a^2\rho)u}{r} & \frac{E+a^2\rho}{r} & 0 & u \end{pmatrix}$$

with the associated eigenvalues of its Jacobian

$$\left| \frac{dG(U)}{dU} \right| = \begin{pmatrix} u-a \\ u \\ u+a \end{pmatrix} . \tag{1.10}$$

2 Discontinuous Galerkin discretization

With the continuity equation given in its continuous form (1.5), solving this system of equations analytically has been accomplished only for special cases so far. Those special cases are e.g. one-dimensional simplifications and particular initial as well as boundary conditions [SES06, Syn91]. Therefore, we have to solve these equations numerically by discretization of the continuous form of the continuity equation.

Several approaches exist to solve such a system of equations numerically. Here, we give a short overview of the most important ones with an Eulerian [Lam32] approach in spatial domain:

- *Finite differences* belong to one of the most traditional methods. They approximate the spatial derivatives by computing derivatives based on particular points in the simulation domain. Following this approach, conserved quantities are given per point.
- *Classical finite elements methods (FEM)* discretize the equations based on an overlapping support of basis functions of disjunct cells. This leads to continuous approximated solutions at the cell boundaries. However, continuous finite element methods suffer from complex stencils with access patterns over several cells.
- With the *Discontinuous Galerkin finite elements method* (DG-FEM), a similar approach as for the classical FEM is taken. However, the basis functions are chosen in a way that their support is cell-local. On the one hand, this avoids complex access patterns, whereas on the other hand, this generates discontinuities at cell borders requiring solvers for the so called Riemann problem. Finite volume simulations can be interpreted as a special case of the DG-FEM formulation with a single basis function of 0-th order.

With our main focus on wave-propagation dominated problems and the DG-FEM being well-suited to solve such problems (see e.g. [LeV02, HW08, Coc98]), we continue studying the discretization of the continuity equation for the remainder of this thesis with the DG-FEM. To determine the framework requirements to run such DG-FEM simulations on dynamic adaptive grids, a basic introduction to the discretization and approximations of the solution in each grid cell is required and given in the next sections.

2.1 Grid generation

To store our simulation data at spatially related positions, we require a grid-generation strategy which decomposes the domain into non-overlapping cells. Relevant simulation data can then be stored on each grid cell and at hyperfaces separating the grid cells. The most commonly used cell-wise grid generation for DG simulations on two-dimensional domains are Cartesian, Voronoi and triangular ones:

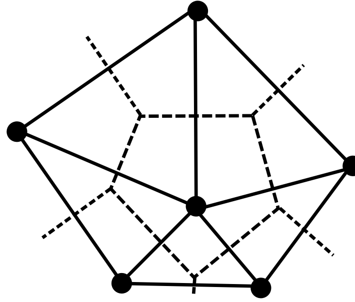


Figure 2.1: Voronoi grid-based generation (dashed line) and the corresponding triangle-based grid generation (solid lines). See e.g. [Ju07] for information about generation.

(a) A *Cartesian* two-dimensional grid generation results in cells with edges parallel to the basis vectors of a Cartesian coordinate system. Using such a structured grid, direct access of the neighboring cells can lead to more efficient access of data in adjacent cells. However, due to refinement of the grid in particular areas, hanging nodes are created (see [BL98] for AMRClaw) and have to be handled in a special manner which is typically based on explicit knowledge of the developer (cf. [BHK07, MNN11]).

(b) Simulations executed on grids generated by *Voronoi diagrams* (e.g. [Wel09, Ju07], see dashed lines in Fig. 2.1) provide an alternative meshing, but make computations with higher-order methods challenging due to the manifolds (e.g. different number of corners) of different shapes.

(c) Using *triangular* grids for simulations, this yields two major beneficial components: First, there is a duality between triangular grids generated by Delaunay triangulation and Voronoi diagrams [She02, DZM07] as shown in Fig. 2.1. Thus, using a triangular grid would also allow us to use it as a grid created by a Voronoi diagram algorithm. Second, adaptive Cartesian grid generation leads to hanging nodes. These nodes can be avoided by inserting additional edges. However, this also requires interfaces provided by the simulation developer. Using adaptive triangular grids based on bisection of the triangle, such hanging nodes can be directly avoided by inserting additional edges.

Due to the hanging nodes created by Cartesian-aligned grids and the very common usage of triangular grids for two-dimensional DG-FEM (cf. [HW08, AS98]), we decided to use a triangulation of our simulation domain.

2.2 Triangle reference and world space

The entire simulation domain can be represented by a set of triangles

$$\Omega = \cup\{C_i | 1 \leq i \leq \#\text{cells}\}$$

with triangle cell primitives C_i only overlapping at their boundaries. Dropping the subindex i, each cell area is given with

$$C := \Big\{(x_0 + \xi(x_1 - x_0) + \eta(x_2 - x_0), y_0 + \xi(y_1 - y_0) + \eta(y_2 - y_0)) | \xi \geq 0 \wedge \eta \geq 0 \wedge \xi + \eta \leq 1\Big\}$$

and (x_n, y_n) referring to one of the three vertices of the triangle C, see Fig. 2.2 for an example. We refer to the space of the coordinates (x, y) as *world space*, whereas (ξ, η) are coordinates in *reference space*.

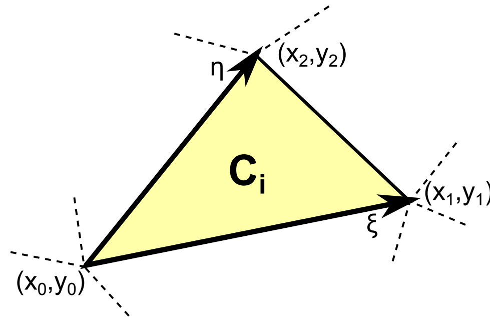


Figure 2.2: Triangular cell C_i in world space.

By applying affine transformations, each triangle C_i and the conserved quantities can be mapped from *world space* to a so-called *reference triangle*. For the sake of clarity, the formulae in the following sections are given relative to such a reference triangle.

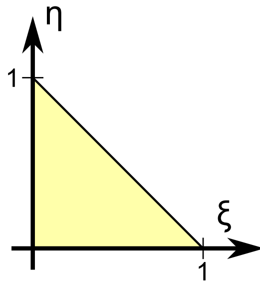


Figure 2.3: Triangle reference space with isosceles triangle

We use the triangle reference space with both triangle legs of a length of 1 and aligned at the x- and y-axes, see Fig. 2.3. The support in reference space is then given by

$$T = \left\{ (\xi, \eta) \in [0,1]^2 \mid \xi \geq 0 \wedge \eta \geq 0 \wedge \xi + \eta \leq 1 \right\}. \tag{2.1}$$

Using a homogeneous point representation, mappings of points from reference to world space are achieved with

$$\mathcal{W}_i^*(\xi, \eta) := \begin{pmatrix} x_1 - x_0 & x_2 - x_0 & x_0 \\ y_1 - y_0 & y_2 - y_0 & y_0 \end{pmatrix} \cdot \begin{pmatrix} \xi \\ \eta \\ 1 \end{pmatrix}.$$

Assuming a mathematical formulation which is independent of the spatial position of the triangle, we can drop the orientation-related components and simplify this mapping to

$$\mathcal{W}_i(\xi, \eta) := \begin{pmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{pmatrix} \cdot \begin{pmatrix} \xi \\ \eta \end{pmatrix},$$

by defining one of the triangle's vertices as the world space origin. Only considering the matrix formulation, this is more commonly known as the Jacobian

$$J_{\mathcal{W}_i}(\xi, \eta) := \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{pmatrix} \tag{2.2}$$

computing the derivatives in world space with respect to reference space coordinates (ξ, η). Affine transformations to and from world space can then be accomplished by additional projections of particular terms (see [AS98,HW08]) and for our simulations on the sphere in Section 6.5.

With the points

$$\begin{aligned} e'_1(w) &:= (1-w,w), \\ e'_2(w) &:= (0,1-w), \text{and} \\ e'_3(w) &:= (w,0) \end{aligned} \tag{2.3}$$

parameterized with w, the interval for each edge is given by $e_i := \{e'_i(w) | w \in [0,1]\}$. Note the unique anti-clockwise movements of the points $e'_i(w)$ *for all edges* and a growing parameter w which is important for the unique storage order of quantities at edges.

For the remainder of this chapter, we stick to the world-space coordinates.

2.3 Basis functions

To approximate the conserved quantities in each reference element, a representation of our state $U(\xi,\eta,t)$ at coordinate ξ and η at time stamp t has to be chosen. We use a basis which is generated by i polynomial functions $\varphi_i(\xi,\eta)$ for each reference element. A linear combination of these polynomials and their associated weights $q_i^k(t)$ for each conserved quantity k then approximates the solution

$$U^k(\xi,\eta,t) \approx q^k(\xi,\eta,t) = \sum_i q_i^k(t)\varphi_i(\xi,\eta) = \hat{U}^k(\xi,\eta,t)$$

and in vector form

$$U(\xi,\eta,t) \approx \begin{pmatrix} \sum_i q_i^1(t)\varphi_i(\xi,\eta) \\ \sum_i q_i^2(t)\varphi_i(\xi,\eta) \\ \dots \\ \sum_i q_i^k(t)\varphi_i(\xi,\eta) \end{pmatrix} = \hat{U}(\xi,\eta,t). \tag{2.4}$$

Next, we choose a set of basis functions for the approximation of our solution and introduce the nodal and modal basis functions. These different sets of basis functions have different properties such as their direct applicability to flux computations and their computational intensity, see e.g. [LNT07]. We introduce nodal and modal basis functions to show the applicability of the developed interfaces for both of them.

2.3.1 Nodal basis functions

Nodal polynomial basis functions are generated for a particular set of nodal points $p_i \in T$ with T the triangle reference space. Evaluating the corresponding basis function at each nodal point holds $\varphi_i(p_j) = \delta_{i,j}$ with δ being the Kronecker delta. Thus, the related coefficient q_i directly represents the approximated solution at a nodal point p_i. Based on a given set of nodal points, the basis functions can then e.g. be computed using a Lagrange interpolation method. For triangles, a manifold of choices for nodal points exists and the locations of nodal points are chosen very carefully. Otherwise, this leads to ill-conditioned or even singular mass matrices [SSD03]. Integrals which are evaluated during or in advance of DG simulations can be computed with numerical integration. Therefore, choosing quadrature points from numerical integration as nodal points would be an obvious choice.

Extending the well known one-dimensional Gaussian quadrature formula to two-dimensions using tensor product would lead to quadrature points outside the reference triangle and more quadrature points would have to be evaluated than necessary. Dunavant [Dun85] developed a distribution of quadrature points with symmetric layout also minimizing the number of quadrature points, however with quadrature points still outside the triangle for higher-order quadrature.

An alternative to these quadrature points is given by Gauss-Lobatto (GL) points [HW08]. In contrast to Dunavant quadrature points, Gauss-Lobatto quadrature points are also placed at the boundaries of the reference triangle and have the advantageous property of creating *sparse matrices for flux computations* (see Section 2.7). An additionally required property of our nodal basis is a unique representation of the polynomials used as basis, also denoted as unisolvency [SSD03]. This unisolvency can be assured by testing for a non-singular Vandermonde matrix e.g. by inverting it to compute the coefficients for the basis polynomials. This property is also fulfilled by the (GL) points. With the properties of creating sparse matrices for flux computations and non-singular Vandermonde matrices, we focus on using these (GL) nodal points in our simulation.

Independent of the nodal points, we further assume a given degree d of the polynomials which we use for our discretization. Then, the maximum number of basis functions is given by

$$N := \frac{(d+1)(d+2)}{2}.$$

2.3.2 Modal basis functions

Using orthogonal basis functions, this would lead to diagonal mass matrices (derived in Section 2.5). This is stated to lead to a computational efficient way to compute the mandatorily required inverted mass matrix [KDDLPI07]. Whereas one-dimensional Jacobi polynomials provide such an orthogonality on the unit interval, Dubiner presented a required extension to the projection of those polynomials to conserve the orthogonality also for integrals over our reference triangle [Dub91]. Here, n-th Jacobi polynomial $P_n^{(a,b)}(\xi)$ is parameterized with the coefficients a and b and the spatially related component ξ.

Using $PQ(n,a,b,\xi) := P_n^{(a,b)}(2\xi - 1)$, the m and n-th orthogonal Jacobi polynomial on a triangle is then given by

$$Po_{tri}(m,n,\xi,\eta) := PQ\left(m,0,0,\frac{\xi}{1-\eta}\right)(1-\eta)^m \cdot PQ\left(n,2m+1,0,\eta\right). \tag{2.5}$$

These orthogonal Jacobi polynomials can be further transformed into ortho*normal* polynomials for a triangle by normalization:

$$P_{tri}(m,n,\xi,\eta) := \frac{Po_{tri}(m,n,\xi,\eta)}{\sqrt{\int_T Po_{tri}(m,n,\xi,\eta)^2 d(\xi,\eta)}}. \tag{2.6}$$

Examples are given in Appendix A.1.2.

2.4 Weak formulation

We restrict the support of our basis functions to the reference triangle and continue by formulating the continuity equation (1.5) in the weak form [Bra07]. Using the basis functions as a particular set of test functions, this yields

$$\int_T \hat{U}_t(\xi,\eta,t)\varphi_j(\xi,\eta) + \int_T \nabla \cdot F(\hat{U}(\xi,\eta,t))\varphi_j(\xi,\eta) = \int_T S(\hat{U}(\xi,\eta,t))\varphi_j(\xi,\eta). \tag{2.7}$$

Applying the Gauss divergence theorem yields

$$\begin{array}{ll} \int_T \hat{U}_t(\xi,\eta,t)\varphi_j(\xi,\eta) & \text{Time derivative} \\ -\int_T F(\hat{U}(\xi,\eta,t)) \cdot \nabla\varphi_j(\xi,\eta) & \text{Spatial derivative} \\ +\oint_{dT} F(\hat{U}(\xi,\eta,t))\varphi_j(\xi,\eta) \cdot \vec{n}(\xi,\eta) & \text{Flux} \\ =\int_T S(\hat{U}(\xi,\eta,t))\varphi_j(\xi,\eta) & \text{Source} \end{array} \tag{2.8}$$

with $\vec{n}(\xi,\eta)$ the normal on each edge. Expanding $\hat{U}$, this yields the Ritz-Galerkin formulation

$$\begin{array}{ll} \int_T \sum_i \frac{U_i(t)}{dt}\varphi_i(\xi,\eta)\varphi_j(\xi,\eta) & \text{Time derivative} \\ -\int_T F\left(\sum_i U_i(t)\varphi_i(\xi,\eta)\right)\cdot\nabla\varphi_j(\xi,\eta) & \text{Spatial derivative} \\ +\oint_{dT} F(\sum_i U_i(t)\varphi_i(\xi,\eta))\cdot\vec{n}(\xi,\eta)\varphi_j(\xi,\eta) & \text{Flux} \\ =\int_T S(\sum_i U_i(t)\varphi_i(\xi,\eta))\varphi_j(\xi,\eta) & \text{Source.} \end{array} \tag{2.9}$$

The following sections continue with successive discretization of those terms, aiming at a matrix and vector representation of all terms, see [GW08].

2.5 Mass matrix $\mathcal{M}$

Discretizing the time step term from Eq. (2.9). We can factor out U by considering that φ are functions only depending on spatial parameters. This yields

$$\int_T \sum_i \frac{U_i(t)}{dt}\varphi_i(\xi,\eta)\varphi_j(\xi,\eta) \;=\; \sum_i \frac{U_i(t)}{dt} \underbrace{\int_T \varphi_i(\xi,\eta)\varphi_j(\xi,\eta)}_{\text{Mass matrix entry } \mathcal{M}_{j,i}}. \tag{2.10}$$

The integrals on the right-hand side of Eq. (2.10) can then be precomputed and their values stored to the mass matrix $\mathcal{M}$. Using vector notation $\vec{U}$, we get a matrix-vector formulation

$$\int_T \sum_i \frac{U_i(t)}{dt}\varphi_i(\xi,\eta)\varphi_j(\xi,\eta) = \mathcal{M}\cdot\vec{U_t} \tag{2.11}$$

with $\vec{U_t}$ also a matrix in case of several conserved quantities.

See Appendix A.1 for examples of matrices derived in this section and used in our simulations. Those matrices are computed with Maple[1], a computer algebra system for symbolic computations.
Applying the explicit Euler time-stepping method, this can then be rearranged to

$$U(t+\Delta t) = U(t) + \Delta t\cdot\mathcal{M}^{-1}(\ldots),$$

thus requiring the computation of the inverse of the mass matrix (see Section A.1.3 for examples). Higher-order integration in time is further discussed in Sec. 2.14.

2.6 Stiffness matrices $\mathcal{S}$

For a partial evaluation of the stiffness terms, we follow the direct approach of expanding the approximated solution, yielding

$$\int_T F(\hat{U}(\xi,\eta,t))\cdot\nabla\varphi_j(\xi,\eta) = \int_T F\left(\sum_i U_i(t)\varphi_i(\xi,\eta)\right)\cdot\nabla\varphi_j(\xi,\eta).$$

Since an accurate integration for flux functions with rational terms would be computationally infeasible, approximations are typically used for this evaluation [Coc98, HW08, Sch03]. Such an

[1] http://www.maplesoft.com/products/maple/

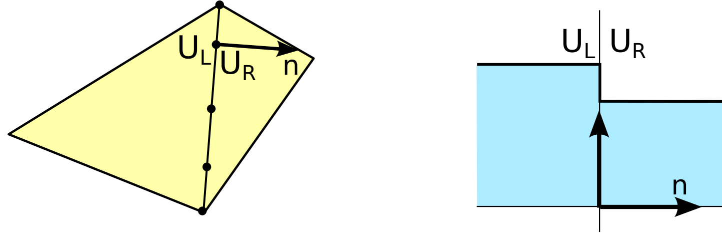


Figure 2.4: Sketch of flux evaluation. Right handed image: The conserved quantities U_L and U_R are in the triangles in world space evaluated at edge quadrature points. Left handed image: After projecting the conserved quantities to the edge space, the flux computation is then based on one-dimensional conserved quantities. These quantities are then used for the flux computation in edge space.

approximation is the nodal-wise evaluation of the flux term at nodal points and reconstruction of a continuous function with the basis functions $\varphi_i(\xi, \eta)$. This simplifies our stiffness term to

$$\begin{aligned} \int_T F\left(\hat{U}(\xi,\eta,t)\right) \cdot \nabla\varphi_j(\xi,\eta) \quad &\approx \quad \int_T \sum_i F(U_i(t))\varphi_i(\xi,\eta) \cdot \nabla\varphi_j(\xi,\eta) \\ &= \quad \sum_i F\left(U_i(t)\right) \int_T \varphi_i(\xi,\eta) \cdot \nabla\varphi_j(\xi,\eta). \end{aligned} \tag{2.12}$$

For the modal basis function, a projection to/from nodal basis is required. We continue by replacing F with two flux functions G and H (see Section 1.1). Then, we can again evaluate integrals and rearrange the equations to a matrix-matrix formulation

$$\sum_i F\left(\sum_i U_i(t)\varphi_i(\xi_i,\eta_i)\right)\varphi_j(\xi,\eta) \approx \mathcal{S}_x^j \cdot G(U_i(t)) + \mathcal{S}_y^j \cdot H(U_i(t))$$

with $\mathcal{S}^j$ selecting the j-th row of matrix S (see Appendix A.1.4).

2.7 Flux matrices $\mathcal{E}$

The flux term $\oint_{dT} (F(\sum_i U_i(t)\varphi_i(\xi,\eta))\varphi_j(\xi,\eta)) \cdot \vec{n}(\xi,\eta)$ in Eq. (2.9) represents the change of conserved quantities via a flux across an edge. With the non-overlapping support of any two adjacent triangles, also the approximated solution is unsteady at the triangle boundaries. By considering each triangle edge separately, we can evaluate the edge-flux term in three steps:

1. First, we evaluate our approximated solution at particular quadrature points on the edge, see left image in Fig. 2.4. Here, we can use a beneficial property of the GL quadrature points (see Sec. 2.3.1): one-dimensional GL quadrature points on the edge directly coincide with the GL quadrature and are thus nodal points of the reference triangle. Therefore, computing the conserved quantities at the quadrature points only involves the conserved quantities stored at these points. See Appendix A.1.5 for the sparse matrix selecting the corresponding conserved quantities.

2. Second, the flux is computed at the quadrature points based on pairs of selected coinciding quadrature points from the local and the adjacent cell. We can simplify this two-dimensional Riemann problem to a one-dimensional one (see right image in Fig. 2.4) by changing the basis to the edge normal taken as the x-basis axis and the discontinuity on the edge at $x = 0$: Only considering a single conserved quantity on an edge, $\hat{U}^L$ and $\hat{U}^R$

represent the current solution $\hat{U}$ on the left and right side of the y-axis, respectively. The change over time can then be computed by using flux solvers which is further discussed in Sec. 2.10.

3. Third, we reconstruct our approximated solution $\sum_i F(U_i(t))\varphi_i(\xi,\eta)$ for each edge based on the computed flux crossing the edge [GW08]. By splitting the surface integral $\oint_{dT}$ over the triangle reference boundaries into three separate integrals $\sum_{e\in E}\oint_{de}$ over each edge $e \in E$ (see Section 2.2) and factoring out the term with the computed flux approximation from the integral, this yields

$$\sum_{e\in E} \underbrace{F(U_i(t)) \cdot \vec{n}(\xi,\eta)}_{\text{Approximated flux}} \int_e \varphi_i(\xi,\eta)\varphi_j(\xi,\eta)$$

   with the evaluation of the Riemann problem term $F(U_i(t)) \cdot \vec{n}(\xi,\eta)$ further described in Section 2.10. Note that the "Approximated flux" term may also depend on other conserved quantities than U_i. Solutions for this integral can again be stored in matrices similarly to the previous sections. For GL points, these matrices are again sparse and influence conserved quantities only if these are stored on the corresponding triangle's edge (see Appendix A.1.5).

The presented method is a quadrature of the flux on each edge with fluxes evaluated at pairs of given quadrature points. Rather than evaluating the flux for several pairs of given points, alternative approaches use flux computations based on a single pair of given functions [AS98]. These functions represent the conserved quantities on the entire edge. Since the interfaces derived in this framework can be also used for such an implementation, we continue using the previously described method without loss of applicability.

2.8 Source term

External effects such as gravitation, the Coriolis force and change of bathymetry can be included in the equations. These effects are handled in the source term or the flux function.

Using the way of extending the source term, this term can be evaluated together with the local cell operations such as the evaluation of the stiffness term. Since we are interested in determination of framework requirements, we neglect this term and expect no impact on the applicability of the framework for a system of equations including source terms.

2.9 Rotational invariancy and edge space

Working with triangles, a straightforward evaluation of the term $F(U_i(t))\cdot\vec{n}(x,y)$ involving both flux functions $G(U)$ and $H(U)$ followed by a multiplication with the outward pointing normal can be optimized. We use the so-called rotational invariancy [Tor01], with flux functions for the hyperbolic systems considered in this thesis holding a crucial property.

We consider a two-dimensional normal vector $\vec{n}_e = (\cos(\alpha), \sin(\alpha))^T$ pointing outward the edge e. Then, for the computation of the flux update, the equation

$$F(U) \cdot \vec{n}_e = G(U) \cdot n_x + H(U) \cdot n_y = R(\alpha)^{-1} F(R(\alpha)(U)) \tag{2.13}$$

holds true for a given rotation matrix $R(\alpha)$. The matrix is setup with an n-dimensional rotation matrix with the entries stored to the $n \times n$ direction dependent components such as the velocity and momentums and 1 on the diagonal for direction independent components. This matrix

rotates, e.g., orientation-dependent components such as the velocity and momentum which are parallel to $\vec{n}_e$ to the x-axis (see Appendix A.1.7 for an example).

For later purpose, the right hand side of Equation (2.13) is further described:

1. To compute the flux across an edge, the conserved quantities depending on a direction are first *rotated from the reference space to the so-called edge space* [AS98].
2. Then, the *change of conserved quantities is evaluated in the one-dimensional edge space.*
3. Finally, the updates related to the flux are *rotated back to reference space* and applied to the conserved quantities stored in the reference space.

2.10 Numerical flux $\mathcal{F}$

With the evaluation of the flux function only involving pairs of selected conserved quantities on the edge from the local and adjacent triangle, we can interpret this as a Riemann problem, for which we consider two different ways of handling it:

- *Net updates:* For finite-volume simulations (DG-FEM with basis function of order 0), Riemann solvers computing *net updates* (See e.g. [LeV02]) are most frequently used. Such solvers compute the net-quantities of the conserved quantities crossing the considered cell boundaries.
- *Flux updates:* On the other hand, Riemann solvers computing *flux updates* are typically used for higher-order methods (e.g. [MBGW10,GW08]). Those solvers rely on flux update computations directly involving the flux term. Even if no conserved quantity is exchanged, a flux computation still leads to a flux generation. This is e.g. due to gravity terms generating a flux despite no convection.

Since both types of Riemann solvers can be implemented with the same interfaces, we only introduce the most frequently used flux solver for higher-order methods: the local Lax-Friedrichs flux, also called Rusanov flux.

2.10.1 Rusanov flux

We consider flux computations which are based on the one-dimensional representation in edge space (see Section 2.9) and a constant state representation of the solution on each side of the edge with U_i^L and U_i^R. This leads to a Riemann problem in edge space with

$$U(z) = \begin{cases} U^L, & \text{if } z \leq 0 \\ U^R, & \text{otherwise} \end{cases} \tag{2.14}$$

with the function

$$\mathcal{F}(U_i^L(t), U_i^R(t))$$

computing the change of conserved quantities for this one-dimensional Riemann problem. Note the difference between $\mathcal{F}(U_i^L(t), U_i^R(t))$ and $F(U)$ with $\mathcal{F}$ evaluating a Riemann problem on an edge whereas F evaluates the flux function inside the reference triangle.

Averaging both points by using the equation

$$\mathcal{F}\left(U_i^L(t), U_i^R(t)\right) := \frac{1}{2}\left(F(U_i^L) + F(U_i^R)\right) \tag{2.15}$$

and using this average value as the flux over the edges would result in a numerically unstable simulation. Therefore, an artificial numerical damping is frequently introduced to overcome these instabilities. This damping is often referred to as numerical diffusion or artificial viscosity [LeV02].

One example of such fluxes involving a numerical diffusion is the Rusanov flux [Rus62]. This flux function introduces a diffusive term with its magnitude depending on the speed of the information propagation. This propagation speed is given by the eigenvalues of the Jacobian matrix of the flux (see Section 1.2.1 and Section 1.2.2 for concrete examples). For the two-dimensional Lax-Friedrichs flux function for Riemann problems given by U_j^L, U_j^R and $F(U)$ we use the flux function [BGN00]

$$\hat{\mathcal{F}} = \underbrace{\frac{1}{2}\left(F(U_j^L) + F(U_j^R)\right) \cdot \vec{n}_e^+}_{\text{average}} - \underbrace{\frac{1}{2}\nu\left(U_j^R - U_j^L\right)}_{\text{numerical flux viscosity}} \tag{2.16}$$

with the numerical Lax-Friedrichs viscosity component given by

$$\nu = \max(|J_F(U^R)|, |J_F(U^L)|). \tag{2.17}$$

Here, $J_F(U)$ is the Jacobian of the function F with respect to the conserved quantities U.

2.10.2 Limiter

With higher-order spatial discretization methods, artificial oscillations can be generated in the solution. To avoid these oscillations, limiters can be used to modify the solution based on data either adjacent via *edges* or *vertices* (see [GW08, HW08]). Since the interfaces for edge-based data exchange required for flux computations and vertex-based exchange for visualization as further explained in Sec. 4.3.4 are considered in the framework, we resigned implementing limiters.

An alternative approach to avoid oscillations is to introduce additional artificial viscosity which was already applied to DG simulations for atmospheric simulations in [Mül12]. Since this part of the thesis focuses on framework requirements for simulations with higher-order DG, we used a similar concept to avoid implementation of challenging flux limiters. For DG simulations and polynomial degrees which are larger than 2, we slightly slow down the fluid's velocity while still being able to test our algorithms for running higher-order simulations on dynamically adaptive grids.

2.11 Boundary conditions

So far, we discretized our problem in space by decomposing the domain into non-overlapping cells $\mathcal{C}_i$. The simulation itself can be executed with cell-local computations and computations based on adjacent cells. However, for the domain boundary $\partial\Omega$, particular boundary conditions have to be constructed and applied. Considering our flux computations, we need DoF at the edges shared by both adjacent cells. Our approach is based on the construction of a ghost edge layer based on the data of the cell which is sharing the ghost edge.

2.11.1 Dirichlet & Inflow

We can apply Dirichlet boundary conditions directly by setting the DoF on the edge to the Dirichlet parameters instead of recomputing them based on cell information adjacent to the boundary edge. Those boundary conditions are of particular interest for simulations of analytic benchmarks with changing *inflow* conditions in each time step (see Section 6.2).

2.11.2 Outflow

One possible way of handling outflow boundary conditions is to set the conserved quantities at the ghost edges to be equal to those from the adjacent cell. However, one severe drawback exists by directly applying this outflow boundary condition. In case of a *flow into* the domain (e.g., due to a negative momentum), the non-directional dependent conserved quantity such as mass or density would be reconstructed to be of equal quantity. This might lead to possible undesired mass inflow. We circumvent this issue by a combination of Dirichlet and outflow boundary conditions: the non-direction related components are set to Dirichlet values, whereas the direction related components orthogonal to the edge are set to zero.

Combining in- and outflow for conserved quantities in edge space q^L of the cell at the boundary, the conserved quantities q^R on the right side of the Riemann problem are respectively set to

$$q_i^R = \begin{cases} q_i^L, & \text{if } q_k^L > 0 \quad \text{\# Outflow: set to left state} \\ 0, & \text{elif } i = k \quad \text{\# Inflow: Cancel momentum} \\ \overline{q}_i, & \text{otherwise} \quad \text{\# Inflow: Dirichlet} \end{cases} \tag{2.18}$$

with k the index of the direction dependent component orthogonal to the edge (e.g. the momentum) and $\overline{q}$ the Dirichlet conditions. Among others, this boundary condition is used in our Tsunami simulations (see Sec. 6.3).

2.11.3 Bounce back

With bounce back boundaries, we can simulate walls streaming back the flow with components orthogonal to the domain boundary. Let the DoF on an edge in edge space be given by u_i^L. We can then reconstruct the DoF on the ghost layer boundary with $U^R := (u_1^L, \ldots, -u_k^L, \ldots, u_n^L)$ in edge space and u_k the direction-dependent component perpendicular to the edge. Among others, this boundary condition is used in our analytic benchmark (see Sec. 6.2).

2.12 Adaptive refining and coarsening matrices $\mathcal{R}$ and $\mathcal{C}$

With a simulation on a dynamically adaptive grid, we refine and coarsen our cells during runtime by splitting triangles into two triangles and merging them respectively. Then, the conserved quantities require projection to the refined or restricted grid cells.

2.12.1 Coefficient matrix

We start by determining a way to compute the coefficients of the approximating solution on our reference triangle based on the weights of given basis functions. This also leads to computations of the weights of the basis functions from a given approximating solution.

We start by denoting the coefficients of the monomials which assemble the basis polynomials (see Section 2.3) $\varphi_i(x,y) := \sum_{(a,b)} \alpha_i^{(a,b)} x^a y^b$, with $\alpha_i^{(a,b)}$. Here, it holds that $0 \leq a+b \leq d$,

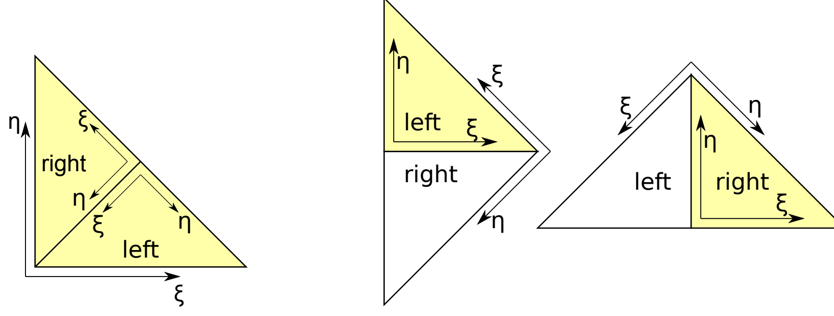


Figure 2.5: Left image: parent's reference space split for left and right child triangle. Right images: child reference spaces. Mapping to and from reference space can be expressed by scaling, rotation and translation.

$0 \leq a$ and $0 \leq b$. We construct a $d \times n$ *coefficient matrix*

$$\mathbf{B}(\varphi) := \begin{pmatrix} \alpha_1^{(0,0)} & \alpha_2^{(0,0)} & \dots & \alpha_N^{(0,0)} \\ \alpha_1^{(1,0)} & \alpha_2^{(1,0)} & \dots & \alpha_N^{(1,0)} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_1^{(0,1)} & \alpha_2^{(0,1)} & \dots & \alpha_N^{(0,1)} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_1^{(0,d)} & \alpha_2^{(0,d)} & \dots & \alpha_N^{(0,d)} \end{pmatrix}$$

with the coefficients for each basis function polynomial given in a respective column and coefficients for identical monomials of basis functions in each row. With q_i a single DoF corresponding to the basis function ϕ_i, we can then rewrite our approximated solution $\varphi(x,y) := \sum_i q_i \varphi_i(x,y)$ in the reference space to a matrix-vector product

$$\vec{\alpha} = \mathbf{B}(\varphi)\ \vec{q}.$$

Then the entries $\alpha^{(a,b)}$ in the vector $\vec{\alpha}$ represent the coefficients of the monomials of our approximated solution $\varphi(x,y) := \sum_i q_i \varphi_i(x,y)$.

In case of only one conserved quantity, we are now able to compute the coefficients of the polynomial related to our approximating solution

$$U(x,y) := \sum_{a=0}^{d} \sum_{b=0}^{d-a} \vec{\alpha}^{(a,b)} x^a y^b.$$

with a matrix-vector product. For multiple conserved quantities, a matrix-matrix product can be used.

We use this matrix formulation for computing the basis function weights $\vec{q}^k$ of the basis functions for a given approximating solution by inverting the coefficient matrix $\mathbf{B}(\varphi)$, yielding

$$\vec{q}^k := \mathbf{B}(\varphi)^{-1} \vec{\alpha}.$$

2.12.2 Affine transformations to and from the child space

Due to splits and joins, a transformation of the DoF to and from different triangles is required, see Fig. 2.5. We respectively denote the triangle space which is split as the parent triangle space and both split ones which have to be possibly joined as the children triangle spaces.

Let

$$\mathbf{R}(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.19}$$

be a two-dimensional rotation matrix computing rotations in clockwise direction for positive α,

$$\mathbf{T}(\xi,\eta) = \begin{pmatrix} 1 & 0 & \xi \\ 0 & 1 & \eta \\ 0 & 0 & 1 \end{pmatrix} \tag{2.20}$$

a translation matrix and

$$\mathbf{S}(s) = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.21}$$

a scaling matrix. A transformation matrix returning the sampling point in the child's left reference space if sampling in parent reference space is then given by

$$\mathbf{P}^M_{\text{toLeft}} := \mathbf{R}\left(\frac{3}{4}\pi\right)\mathbf{S}\left(\sqrt{2}\right)\mathbf{T}\left(-\tfrac{1}{2},-\tfrac{1}{2}\right)$$

with the corresponding function

$$\mathbf{P}_{\text{toLeft}}(\xi,\eta) := \mathbf{P}^M_{\text{toLeft}} \cdot \begin{pmatrix} \xi \\ \eta \end{pmatrix} = \begin{pmatrix} -\xi-\eta+1 \\ \xi-\eta \end{pmatrix},$$

applying the matrix at the point (ξ,η) given as parameters. E.g. this transforms the point $(1,0)$ in the parents reference space, thus at the right triangle corner, to $(0,1)$ in the parents reference space. An example is given in the left image of Fig. 2.5: First the left child's origin is translated to the origin of the parent's reference space, then the child's space is scaled up followed by a final rotation.

For the right reference triangle, this yields

$$\mathbf{P}^M_{\text{toRight}} := \mathbf{R}\left(-\frac{3}{4}\pi\right)\mathbf{S}\left(\sqrt{2}\right)\mathbf{T}\left(-\tfrac{1}{2}\right)$$

$$\mathbf{P}_{\text{toRight}}(\xi,\eta) := \mathbf{P}^M_{\text{toRight}} \cdot \begin{pmatrix} \xi \\ \eta \end{pmatrix} = \begin{pmatrix} -\xi+\eta \\ -\xi-\eta+1 \end{pmatrix}.$$

Projections from the child triangles to parent triangles are then similarly given by

$$\mathbf{P}_{\text{fromLeft}}(\xi,\eta) \;:=\; (\mathbf{P}^M)^{-1}_{\text{toLeft}} \cdot \begin{pmatrix} \xi \\ \eta \end{pmatrix} = \begin{pmatrix} -\frac{1}{2}\xi+\frac{1}{2}\eta+\frac{1}{2} \\ -\frac{1}{2}\xi-\frac{1}{2}\eta+\frac{1}{2} \end{pmatrix} \quad \text{and} \tag{2.22}$$

$$\mathbf{P}_{\text{fromRight}}(\xi,\eta) \;:=\; (\mathbf{P}^M)^{-1}_{\text{fromRight}} \cdot \begin{pmatrix} \xi \\ \eta \end{pmatrix} = \begin{pmatrix} -\frac{1}{2}\xi-\frac{1}{2}\eta+\frac{1}{2} \\ \frac{1}{2}\xi-\frac{1}{2}\eta+\frac{1}{2} \end{pmatrix}. \tag{2.23}$$

2.12.3 Prolongation to child space

Sampling of the child's approximated solution using reference coordinates from the parent reference space then gives

$$\varphi_{\text{toLeft}}(\xi,\eta) := \varphi(\mathbf{P}_{\text{toLeft}}(\xi,\eta)) \qquad \text{and} \qquad \varphi_{\text{toRight}}(\xi,\eta) := \varphi(\mathbf{P}_{\text{toRight}}(\xi,\eta)) \tag{2.24}$$

By using a representation of the child's basis functions in parent's reference space, the refinement matrix is determined by first computing the polynomial coefficients for the approximating polynomial in parent space with $\mathbf{B}(\varphi)$ and then reconstructing the weight q^k for the child's polynomials by using the representation of the child's polynomials $\varphi_{\text{toLeft}}(\xi,\eta)$ in the parent space:

$$R_{\text{left}} := \mathbf{B}(\varphi_{\text{toLeft}})^{-1}\mathbf{B}(\varphi) \qquad \text{and} \qquad R_{\text{right}} := \mathbf{B}(\varphi_{\text{toRight}})^{-1}\mathbf{B}(\varphi) \tag{2.25}$$

2.12.4 Restriction to the parent space

Restriction operations can be computed in a similar manner to the prolongation (see Eq. (2.25)). However, due to the discontinuity at the shared edges of the children, the approximated solution of both children cannot be represented by a single parent triangle.

We choose a restriction which is averaging the polynomial basis functions of both children extended to the parent's support yielding

$$\varphi_{\text{toLeft}}(\xi,\eta) := \varphi(\mathbf{P}_{\text{toLeft}}(\xi,\eta)) \qquad \varphi_{\text{toRight}}(\xi,\eta) := \varphi(\mathbf{P}_{\text{toRight}}(\xi,\eta)) \tag{2.26}$$

$$C_{\text{parent}} := \frac{1}{2}\left(\mathbf{B}(\varphi)^{-1}\mathbf{B}(\varphi_{\text{fromLeft}}) + \mathbf{B}(\varphi)^{-1}\mathbf{B}(\varphi_{\text{fromRight}})\right) \tag{2.27}$$

Such an averaging is obviously not able to reconstruct the polynomial accurately in case of discontinuities at the shared edges as this is typically the case for our DG simulations. We want to emphasize, that other coarsening computations, e.g. reconstructing the basis polynomial with minimization with respect to a norm, can yield improved results. With improved reconstruction methods being transparent to the determined interface requirements and since this section focuses on the determination of the basic requirements for the computation of DG-based simulations, we continue using this coarsening method.

The derived restriction and prolongation matrices can be applied transparently to either nodal or modal basis functions since their construction is based on the coefficient matrix, which is again based on the coefficients of the basis polynomials.

2.12.5 Adaptivity based on error indicators

To trigger refining or coarsening requests on our grid, we use error indicators to distinguish between refining, coarsening or no cell modifications.

For this work, we considered two different adaptivity indicators:

- The first one is based on the nodal points of our approximated solution. Considering in particular the shallow water equations, our error indicator for most of the benchmark studies is based on the relative deviation of the water surface from the horizon: $|h+b|$.
- The second indicator is based on net-updates which is further discussed in Section 6.1.3.

If an indicator exceeds a particular refinement threshold value, a split of the cell is requested. In case of undershooting the coarsening threshold value, the indicator allows joining this cell with an adjacent one in case of both cells agreeing in the coarsening process.

2.13 CFL stability condition

Using an explicit time-stepping scheme, the size of a time step to run a stable simulation must not exceed a particular value. We use the Courant-Friedrichs-Lewy (CFL) [CFL28] condition which relates the maximum time-step size in each cell to its shape and size as well as the wave propagation speed in the cell.

Several methods for advancing the simulation in time can then be used, each one different in the number of cells being advanced with the same time-step size: Cell-local [HW08], patch-wise [BGLM11], cluster-based [CKT09] and global time stepping methods with the cluster-based one not yet existing for dynamically adaptive grids. We decided to implement the global time stepping method with the same integration width over time for all cells.

Considering isosceles triangles, the cell size and shape dependent factor is computed with the in-circle radius

$$r_{\text{inner}} := (2 - \sqrt{2}) \cdot |cathetus|$$

with $|cathetus|$ representing the length of one of the triangle legs of cell C_i in the world space.

The propagation speed of the wave itself depends on the solution computed for the Riemann problem. For flux solvers based on linearization of the flux function (see Sec. 1.2.1, 1.2.2), this wave propagation speed s_{wave} is given by the eigenvalues of the Jacobian of the linearized flux function. For our Rusanov flux, this propagation speed is equal to the viscosity parameter ν. This yields

$$\Delta t_e := \frac{r_{\text{inner}}}{s_{\text{wave}}} \cdot \text{CFL} \tag{2.28}$$

for all time-step sizes Δt_e based on the flux computations on all cell edges e with the CFL set to a constant value depending on the maximum degree of basis functions and the used flux solver. For basis functions of degree N, we set this value to

$$CFL := 0.5 \frac{1}{2N+1},$$

see [KD06]. The global time-step size is then determined with $\Delta t = \min_e(\Delta t_e)$.

2.14 Time stepping schemes

Regarding the derivative of our conserved quantities with respect to time, we only considered the explicit Euler method which is of first order. This section is about higher-order Runge-Kutta time stepping schemes and sets up the basic requirements of the integration of our simulation in time. For sake of readability, the spatial parameters ξ and η for the conserved quantities are dropped in this section.

Typical higher-order time stepping methods such as Runge-Kutta (RK) are a commonly chosen alternative to the explicit Euler due to their higher accuracy in time. Regarding our requirement analysis, such higher-order methods should be obviously be considered in our development.

With the RK method, accuracy of higher-order is achieved combining the conserved quantity updates based on several smaller time-step computations. A *generalization of the RK method* for higher-order time integration leads to s stages $i = (1, 2, \ldots, s)$ [But64, CS01, HW08]:

$$\begin{aligned} V_0 &:= U(t); \\ V_i &:= V_0 + \Delta t \textstyle\sum_{j=1}^{s} a_{i,j} D_j; \\ D_i &:= R(V_i); \end{aligned} \tag{2.29}$$

yielding an explicit formulation for $a_{i,j} = 0$ for $i < j$. The solution is finally given by

$$\hat{U}(t + \Delta t) := V_0 + \Delta t \sum_{i=1}^{s} b_i D_i.$$

Considering the framework development in Part III, we store both update values D_i and conserved quantities V_i for each stage and finally update the conserved quantities by the formula given with $a_{i,j}$ and b_i. The coefficients $a_{i,j}$ and b_i depend on the desired order of the method and are typically given in the format of the Butcher tableau [But64] (See Appendix A.1.6).

PART III

EFFICIENT FRAMEWORK FOR SIMULATIONS ON DYNAMICALLY ADAPTIVE GRIDS

This part of the thesis introduces an efficient simulation of wave propagations based on a model with hyperbolic equations. With waves propagating over time, refining the grid in feature-rich areas and coarsening it otherwise is one major requirement. This requires efficient implementations of simulations on dynamically adaptive grids which represents one of the great challenges in nowadays HPC. Typical requirements are fast grid traversals, memory-hierarchy-aware data access and parallelization approaches for large-scale systems including fast load balancing to name just a few of them. Our software development approach aims at solving these issues and is presented in this part.

Chapter 3: Requirements and related work
Since our main goal is an efficient implementation of simulations on dynamically adaptive grids, we first specify requirements on the framework mainly driven by the underlying hardware and show the related work.

Chapter 4: Serial implementation
Based on this requirements analysis, spacetrees provide a solution. After a short introduction to these spacetrees, we provide a formal description of existing work on the grid-data management and communication system based on stacks and streams. This is followed by extensions, modifications and optimizations compared to previous developments.

Chapter 5: Parallelization
We then introduce the parallelization of the inherently serial simulation with our cluster-based domain decomposition. The resulting software design directly offers parallelization on shared- and distributed-memory systems with the results presented in the respective sections.

Chapter 6: Application scenarios
To show the applicability of our development, we extended the framework with state-of-the-art solvers for analytical convergence studies and a realistic Tsunami simulation. Furthermore, we present different output backends for the simulation data. This chapter closes with extensions for simulations on the sphere and multi-layers.

Chapter 6.7: Summary and outlook
Based on our new parallelization approach, we will give conclusions and an outlook for further extensions of the framework in this Chapter.

3 Requirements and related work

Based on the discretization to solve the hyperbolic system of equations, we continue with a requirement analysis for our development. These requirements are mainly driven by the considered simulation and the underlying hardware. This is followed by a short introduction to the SFCs and a presentation of the related work.

3.1 Simulation: grid, data and communication management

For our simulations, the *grid management* has to allow adaptivity during runtime due to demands on the resolution to be increased by refining the grid in feature-rich areas (e.g. close to the wave front) and coarsening the grid in areas which do not have a large contribution to the final result.

The shallow water equations are frequently used models to research wave-propagations and a simulation of them only requires a two-dimensional domain discretization. Therefore, we focus on efficient simulations of two-dimensional scenarios with triangles as the basic primitives. Despite its two-dimensional nature, particular domain triangulations can be created to assemble a two-dimensional surface on a sphere (see Sec. 6.5). Using multiple layers in each cell, this leads to further possible applications such as weather simulations (see Sec. 6.6).

For the simulation of our considered hyperbolic equations, we particularly focus on the following requirements:

- **Data access:** To advance the simulation in time, data in other cells has to be made accessible, e.g. to flux limiters via edge- or node-based communication.

  We developed clear interfaces (Section 4.9) yielding efficient parallel communication schemes (Sections 5.2)

- **Usability and parallelization:** For usability reasons, a parallelization approach should lead to a grid, data and communication management which is almost transparent to the application developer.

  A multi-layer framework design was chosen to provide appropriate abstraction levels and extendibility (Section 5.3).

- **Flexible simulation domain:** Under consideration of the communication schemes developed in this work, we are able to generate domains which can be assembled with triangle primitives.

  Different kinds of domain triangulations such as those required for a simulation on a sphere can then be assembled (Section 5.4).

- **Flexible cell traversals:** For several scenarios, not all cells of the simulation domain require to be traversed. A well-known example is the optimization of the execution of

iterative solvers. Additional smoother iterations on grid areas with a residual already undershooting a threshold can be avoided, yielding a local relaxation method [Rüd93]. Another optimization is given for the generation of a conforming grid without hanging nodes. Here, areas with an already conforming grid do not require further execution of traversals generating a conforming grid.

These issues require sophisticated cell traversals and a software design offering the corresponding flexibility. We provide a possible solution by introducing clustering on dynamic adaptive grids and show results for the skipping of cluster traversals for creating a conforming grid, see e.g. Section 5.8.2

3.2 HPC requirements

We highlight several mandatory aspects in the context of next-generation HPC systems. These systems demand for consideration of memory hierarchies, cache-oblivious behavior as well as data locality. For the *memory access optimizations*, we focus on the following three aspects:

(a) *Size of accessed memory:* With the memory wall assumed to be one of the main bottlenecks in the future [Xie13], the memory transfers should be reduced to a minimum.

(b) *Data migration:* For load balancing reasons, also efficient data migration has to be provided. Such a data migration should use asynchronous communication, lowering the amount of migrated data and provide an efficient method to update the adjacency information.

(c) *Energy efficiency:* With memory access on the next generation architectures expected to require increasing energy consumption compared to the computations [BC11], a reduction of memory accesses is expected to directly lead to energy optimized algorithms.

Next, we discuss the parallelization on thread and instruction level. Current hardware generations are not able to scale with Moore's law by increasing the frequency only due to physical constraints [BC11]. To reduce the computation time for simulations with a given discretization, the current way out of this frequency limitation is a parallelization on thread and instruction level. Therefore, the algorithm design further requires the capability to run efficiently on highly parallel systems with dozens and even hundreds of cores. With a dynamically changing grid, this is considered to be challenging due to the steadily changing workload, and in our case changing workload after each time step. Two different parallelization methods regarding Flynn's taxonomy [Fly66] are considered nowadays:

(a) MIMD (multiple instructions, multiple data): For *thread level parallelization*, future parallel HPC systems provide a mix of cache coherency and are considered for the framework design: on shared-memory systems, cache-coherent memory is typically available whereas non-cache-coherent memory is provided across distributed-memory systems. Considering accelerator cards such as the Xeon Phi, a hybrid parallelization is getting mandatory.

(b) SIMD (single instruction, multiple data): On *instruction level parallelism*, today's HPC computing architectures demand data to be stored and processed in vector format. This allows efficient data access and processing with vector operations executing multiple operations on each vector element in parallel. E.g. on the current XeonPhi architecture, one vector can store 16 single-precision numbers. Using such operations is mandatory for getting close to the maximum flop rate of one chip, thus should also be considered in the software development.

3.3 Space-filling curves

We give a very brief introduction to space-filling curves (SFCs) and refer the interested reader to [Sag94] for more information on SFCs. These SFCs were motivated by Cantor's discovery of an equivilant cardinality of higher-dimensional manifolds and a one-dimensional interval. However, the higher-dimensional curve generated by the one-dimensional interval was not continuous. Peano then searched for such a continuous curve parametrized with the one-dimensional form and touching every point in the higher-dimensional manifold and discovered the SFCs, yielding these properties. For a long time, SFCs were only considered from a theoretical point of view to show that there is a mapping between $[0;1]^2$ and $[0;1]$. Nowadays, their discrete form based on iterations play an increasing role in multiple areas such as databases [FR89], computer graphics [GE04], adaptive mesh refinement [BZ00] and performance optimized computations of linear algebra operations [HT12]. Hence, space-filling curves (SFC) provide beneficial properties which can provide an efficient solution to the memory hierarchy and load balancing challenges. Using scaling and translation of our simulation domain, we assume our simulation space being enclosed by a unit-hypercube $\Omega_d := [0;1]^d$. The SFCs considered in this work then provide a *surjective mapping* from a one-dimensional interval $\Omega_1 := [0;1]$ to each point in the simulation domain Ω_d.

Frequently used SFCs for higher dimensions are Hilbert and Peano curves for a 2^d- and 3^d-section of each cell on Cartesian grids. For two-dimensional grids, the Sierpiński curve on triangular grids offers bi-section of triangular cells.

Further considering cache-oblivious communication structures, stack- and stream-based communications structures [GMPZ06] yield cache-oblivious communications. With our simulation based on two-dimensional triangular grids, this leads us to the bi-secting Sierpiński curve with a stack-based communication which was initially presented for a serial traversal in [BSVB08] and is further described in Section 4.

We can use SFCs to optimize for spatial and temporal data access locality and to enumerate the cell primitives on Ω_1, yielding beneficial properties:

(a) *Spatial local data access:*
Spatial locality refers to data access which is close to the previously accessed data. Storing the simulation data consecutively ordered along the one-dimensional mapping of the multi-dimensional SFC coordinates hence improves the spatial local access.

(b) *Temporal locality:*
This refers to the same data being accessed again after a short time interval. If accessing adjacent data such as conserved quantities computed on edges, the probability of this data being already stored in cache can be increased due to the recursively structured domain.

(c) *Load balancing:*
SFCs furthermore provide an efficient way of partitioning simulation domains, see Section 3.4.1. This is due to their capability of enumerating all available grid cells, hence reducing the complexity to a one-dimensional partitioning problem.

3.4 Related work

With a wide range of already existing research for dynamic grid creation and management, we give a brief overview of the related work. We describe this work top-down, starting with the pure mesh-generation and decomposition approaches, followed by frameworks to run simulations on

dynamically changing meshes and then related work which investigated the Sierpiński SFC for running simulations.

3.4.1 Dynamic mesh partitioning

We first introduce different approaches for dynamic mesh partitioning as well as dynamic mesh rebalancing and assume the mesh to be already given. We do not aim for a complete literature survey, but only give an overview of most related work.

Graph partitioner

One way to deal with mesh generation and its partitioning is to consider the mesh as a graph connecting mesh elements via their hyperfaces and apply graph partitioning algorithms on this graph. Such HPC partitioning algorithms aim at creating partitions of meshes with very good properties for efficient parallelization. One optimization property is e.g. reducing the number of interfaces shared with other partitions to reduce the data to be communicated by reducing the number of edge cuts for two-dimensional meshes and face cuts for three-dimensional meshes.

For dynamically adaptive meshes, additional optimization objectives for optimized remeshing are given by minimizing communication data, considering the costs for meshes after load balancing [KSK03] and maximizing quality of load balancing. Examples for such graph-based meshing and partitioning tools are e.g. ParMETIS [KK98][1] and PT-Scotch [CP08][2].

Geometric partitioner using SFCs

Instead of using graph theory for optimized mesh partitioning, the underlying geometry and spatial placement of the mesh elements can be used. Such geometric partitioners can be based on e.g. SFCs curves. They yield one property which is of particular importance for partitioning: The surjective mapping of each point in an d-dimensional space to a one-dimensional interval, retaining the location of the d-dimensional space in the one-dimensional representation [MJFS01]. Thus, load balancing can be reduced to a one-dimensional representation: each partition is assigned an interval of the one-dimensional problem with similar workload [BZ00]. Libraries supporting SFC-based partitioning are e.g. amatos [BRH$^+$05][3] and Zoltan [BCCD12][4].

Graph- vs. SFC-based partitioning

Handling dynamic partitioning based on graph partitioners obviously involves complex graph operations [BZ00], resulting in an NP-hard problem [Zum00]. Even with a multilevel diffusive scheduling to compensate the NP-hard problem, a partitioning based on SFCs still leads to faster load balancing while still yielding partition optimality regarding the edge cut close to graph-based partition algorithms [Mit07].

3.4.2 Simulation software including grid generation

We give an overview of well-known frameworks with a built-in grid generation. Again, we also do not aim for a complete survey, but only consider the work which is of most relevance for our considered simulations. We compare the possibilities of each framework regarding our requirements described in the previous sections.

[1] `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`
[2] `http://www.labri.fr/perso/pelegrin/scotch/`
[3] `http://www.amatos.info/`
[4] `http://www.cs.sandia.gov/Zoltan/ug_html/ug_intro.html`

- *OpenFOAM:* Features of OpenFOAM include a PDE toolbox implemented with an embedded language, capabilities for simulations of CFD with different models, and also dynamic mesh generation. For adaptive mesh refinement which is required for efficient simulation of wave propagations, OpenFOAM requires additional memory to store e.g. connectivity information on cells.

- *Dune:* The Dune Project[5] (Distributed and Unified Numerics Environment) also offers dynamic adaptive grids with the ALUGrid module with ParMETIS [BBD+08]. Dune stores connectivity information on each cell and therefore also requires additional memory similar to OpenFOAM.

- *p4est:* The p4est [BWG11] software library offers mesh management for parallel dynamic AMR-based forest of octrees and is only partly an entire simulation framework by itself. It already proofed its high scalability [BGG+08] also in the context of DG simulations. Since it is based on octrees, this obviously yields hanging nodes. Since we require triangular grids, inserting edges to each cell can be used to reconstruct triangles, however this would lead to requirements of rearranging mesh triangles in case of additional hanging nodes created or removed in a cell.

- *Peano:* This framework is a mesh-based PDE solver on Cartesian grids with the mesh generated by the Peano SFC. Outstanding features are e.g. support for arbitrary dimensions and dynamically adaptive mesh refinement with a recursive trisection in each dimension [Nec09, Wei09]. Since it is based on Cartesian grids, the drawbacks are similar to those of *p4est* with our requirements of triangle cells.

With the memory efficiency requirement for dynamically adaptive simulations on triangular grids being the main driving requirement of this work, none of the frameworks mentioned above, and to our best knowledge also no other frameworks fulfill our demands.

3.4.3 Related software development

This section outlines our contributions compared to the preceding work on the serial and parallel simulation with the Sierpiński stack- and stream-based approach. Based on the dissertation of Csaba Vigh [Vig12], our major algorithmic differences and achievements are the following:

- Our adaptivity automaton and the cluster-based parallelization allows skipping of adaptivity traversals with already consistent clusters.

- The parallelization in [Vig12] is only focused on MPI, whereas our run-length encoded communication scheme and the way how the meta information is updated provides an efficient parallelization for shared- and distributed-memory systems.

- We introduced an efficient node-based communication scheme for the parallelization.

- The Riemann solvers in our work are more sophisticated and require an improved communication scheme.

- In [Vig12], a data migration is presented with an enumeration of cells following the SFCs. These numbers are propagated via edges after refinement and coarsening operations. In contrast, our approach does not require such a global information, resulting e.g. in a considerable improvement for data migration.

[5] `http://www.dune-project.org`

3.4.4 Impact on and from related work

The serial traversal with the stack- and stream-based approach used in this thesis is based on the preceeding research on serial traversals, e.g. [GMPZ06,BSVB08]. In our work, we developed a software design and algorithms for an efficient parallelization on shared- as well as distributed-memory systems.

There are several groups working on SFC-based traversal and organization schemes for PDE solvers; to our knowledge, only the work of Prof. Michael Bader and his co-workers is also Sierpiński -based and is called $sam(oa)^2$. Their work has the closest relation to this thesis, and that's why we mention it here. Their focus was initially on a software design and a parallelization approach which only supports MPI parallelization, see e.g. [Vig12] which was developed in cooperation with Kaveh Rahnema.

Several discussions with him and his group led to certain assimilations of our approaches. However, the major difference still lies in their focus on SFC cuts for partitioning. These SFC cuts lead to further research-related challenges such as the construction of consistent meta information.

4
Serial implementation

This chapter describes the development of the non-parallel simulation based on the stack- and stream-based grid traversals following the Sierpiński SFC. We present interface demands to run discontinuous Galerkin simulations on dynamically adaptive grids and to visualize the results. This serial implementation is the basis of the parallel development described in the next chapter.

- **Section 4.1: Grid generation with refinement trees**
  We start with the grid generation based on a SFC *refinement tree*.
- **Section 4.3: Stack-based communication**
  Based on the grid generation, the *data exchange* via shared edges and vertices is described.
- **Section 4.4: Classification of data lifetime**
  This is followed by a *classification* of the required data access.
- **Section 4.5: Stack- and stream-based simulation on a static grid**
  This section shows a concrete implementation of a *stack- and stream-based simulation*.
- **Section 4.6: Adaptivity**
  A description of how to handle the adaptivity with simulations on *dynamically adaptive grids* is then presented.
- **Section 4.7: Verification of stack-based edge communication**
  So far, we assumed our communication via the stack system to be correct. This section gives a formal proof for edge-based communication.
- **Section 4.8: Higher-order time stepping: Runge-Kutta**
  The implementation of higher-order Runge Kutta methods with our stack- and stream-based simulation approach is described.
- **Section 4.9: Software design, programmability and realization**
  For usability reasons, the software was developed with a layered framework approach. This software design is presented and gets important for the new parallelization method.
- **Section 4.10: Optimization**
  We also developed optimizations, some of them presented in this Section.
- **Section 4.11: Contributions**
  Finally, we summarize our contributions to the serial development.

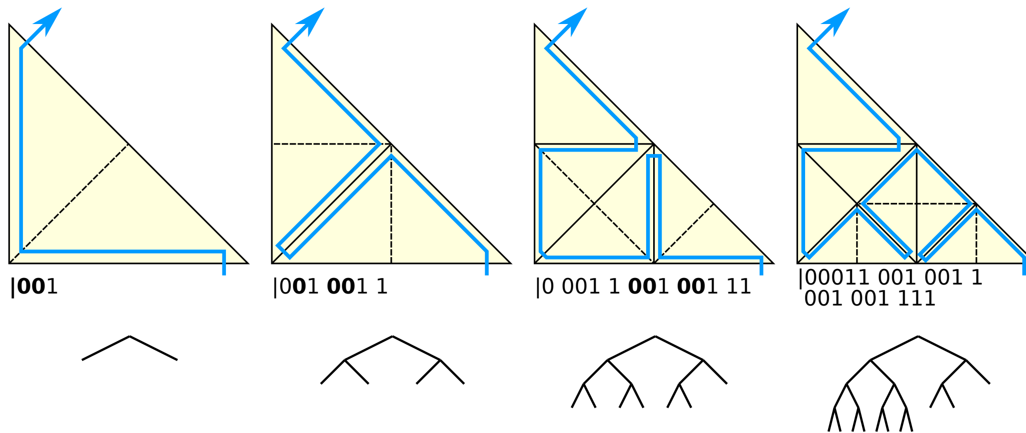


Figure 4.1: Top row: bitstream array and illustration for successively refined grids. 0 represents a leaf node and 1 an inner tree node. The symbol "|" denotes the beginning of the array. During a spacetree traversal, the array is read from right to left. Bold face numbers represent cells which are refined in the next right handed image. Bottom row: respective cell tree structure.

4.1 Grid generation with refinement trees

With spacetrees [Fra00, Gün04, Pög04, Mun06, Nec09, Wei09], a particular domain Ω is initially represented in its non-refined state by a single cell. In a tree-like structure, this cell is represented by the root node of a tree. For refining a cell, two or more nodes are appended to the corresponding leaf tree node [Mit07], one node for each additional cell. Following this refinement scheme, the grid can be represented in a hierarchical tree with the leaf nodes representing the entire grid with non-overlapping cells. A serialization of the spacetree is then given by a depth-first traversal of the spacetree.

The spacetree used in this work is based on triangles and refinements of a cell based on the so-called newest vertex bisections [Mit07]. This bisection splits the triangle cell with the inserted refinement edge starting at the latest newest inserted vertex and the other vertex placed on the opposite edge. Such a bisection of triangles allows an adaptive mesh generation with an underlying binary tree with the Sierpiński SFC [HDJ04]. Bisecting an isosceles right-angled triangle, this refinement creates triangles of the same shape, only different in rotation, orientation and size. However, this does not limit the applicability to other triangle shapes: we can map triangle cells to a different shape and present this possibility for simulations on a sphere (Section 6.5).

We can store the *grid structure* with a bit-stream (0 and 1) representing the spacetree structure. Traversing the grid can then be accomplished by successively reading elements of this *structure stream* and following the tree in a depth-first tree traversal in case of a 1 marker. In case of reading a 0 marker, we reached a leaf element which is a representative for a grid cell.

Refining a cell with newest vertex bisection can then be accomplished by replacing a single leaf-child bit 0 with $0, 0, 1$ assuming that the bit stream is read from right to left. Coarsening of a cell is achieved by joining two cells which are children on the recursive tree traversal, thus replacing a sequence $0, 0, 1$ with 0. Examples of different refinement states are given in Fig. 4.1. Note, that this can also lead to hanging nodes whereas we require a conforming grid (i.e. a grid without hanging nodes). The generation of a conforming grid with this property is further discussed in Section 4.6.

4.2 Stacks

We assume that the structure of a grid is stored on a stack. Then, we can traverse the grid by successively fetching the top-most element of this stack on each spacetree node.

Formally, a stack S^k is modified via *push* and *pop* functions, respectively, pushing data to the top of the stack or removing a data item from the top of the stack.

Given a stack $S^k = (s_1^k, s_2^k, \ldots, s_{|S^k|}^k)$ with its stack elements s_i^k, a push of element α can be formalized via

$$push : (S^k, \alpha) \mapsto (s_1^k, s_2^k, \ldots, s_{|S^k|}^k, \alpha) \tag{4.1}$$

and pop by returning a tuple with the stack and the fetched data, yielding

$$pop : (S^k) \mapsto ((s_1^k, s_2^k, \ldots, s_{|S^k|-1}^k), s_{|S^k|}^k). \tag{4.2}$$

Regarding the implementation of such a stack access, the push and pop operations are offered with different implementations: Pushing and fetching of single or multiple stack elements. The topmost element is referenced via a pointer. Stack size modifications can, thus, be achieved by increments and decrements on this pointer only. Due to the small number of operations related to stack access, all methods are marked to be inlined by the compiler, thus no function call is involved for the execution of a push or pop operation. The stack is preallocated with the maximum capacity of possible elements being stored on the stack during a traversal (see Section 4.10.6).

The bit stream for a forward spacetree traversal used so far is not directly applicable for a backward traversal. Using our stack structure as the input stream only and following the input/output scheme suggested in [GMPZ06,Nec09,Wei09], this demands for creating a structure stack usable for backward traversals.

For generation of the backward structure stack, we use post-order push operations of the structure markers to a stack. The spacetree traversal is then based on bits fetched from $S_{forward}^{structure}$ and setting up a backward structure stack $S_{backward}^{structure}$ by pushing the structure bits in postfix order, after the children have been visited. This creates a structure stack with its input usable for traversals in reversed direction.

4.3 Stack-based communication

Since our simulations require data exchange between grid cells, we describe a cache-aware way of exchanging data stored for different grid cells based on stack operations.

The high variety of possible application requirements on the grid traversals and data exchange patterns demands for a code generator. Otherwise, these variety requirements would lead to either (a) generic code including computations not required for the particularly considered application, thus wasted computation time, or to (b) handwritten code, specialized for each application which leads to code which has to be modified each time before being capable of using them for different grid traversal and communication requirements. Therefore, we decided to use a code generator and continue by introducing a structured formulation of the grid traversal and communication approach to allow the creation of the traversal code with a code generator (See Section 4.9.5). This is contrary to the approach taken before for a stack- and stream-based SFC traversal which was based only on hand-written traversal code.

4.3.1 SFC-labeled grid generation

We extend the grid generation approach from Section 4.1 by labeling each cell with additional information which is described in this and the next section. Such labels yield properties which

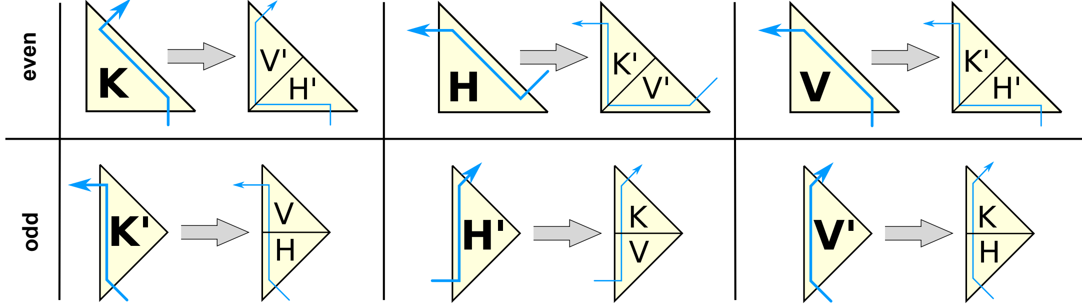


Figure 4.2: *Geometric refinement grammar* we use in our scheme. The bisection is described with the tick arrow. The right side of a thick gray arrow shows application of the grammar rule with the resulting children types. Triangles of type *even* are denoted with K, H and V and those of type *odd* are marked with an additional prime: K', H' and V' [SBB12].

are getting beneficial in upcoming Sections.

The Sierpiński SFC can be defined via a grammar [BSVB08] being recursively applied with different basic triangle traversal types given in

$$\mathbb{G} := \{K, V, H\}. \tag{4.3}$$

The notation of these types are related to the location of the SFC entering and leaving the triangle, see Fig. 4.2. K traversals enter a triangle via a cathetus (***K**athete* in German), H via the hypotenuse and V entering and leaving the triangle via a cathetus, creating a V-like shape. Additional properties are used for default and mirrored traversal by

$$\mathbb{O} := \{even, odd\}. \tag{4.4}$$

The marker *odd* mirrors the traversal direction along the newest vertex bisection edge. The tuple

$$\mathbb{T} := (\mathbb{G}, \mathbb{O}) \tag{4.5}$$

then defines all SFC traversal combinations (see Fig. 4.2 for an overview) required for our communication scheme. For visualization, we always draw the SFC close to the hypotenuse of each triangle as suggested in [Sch06].

To recursively traverse the spacetree, the function *childT* stores the grammar applied to the tree traversal (see cells on the right side of the gray arrow in Fig. 4.2).

$$childT : (\mathbb{G}, \mathbb{O}) \to ((\mathbb{G}, \mathbb{O}), (\mathbb{G}, \mathbb{O}))$$

$$\begin{array}{rrcl}
childT : & (K, even) & \mapsto & ((H, odd), (V, odd)) \\
& (H, even) & \mapsto & ((V, odd), (K, odd)) \\
& (V, even) & \mapsto & ((H, odd), (K, odd)) \\
\\
& (K, odd) & \mapsto & ((H, even), (V, even)) \\
& (H, odd) & \mapsto & ((V, even), (K, even)) \\
& (V, odd) & \mapsto & ((H, even), (K, even))
\end{array} \tag{4.6}$$

Theorem 4.3.1 (Shared edge with SFC order) *The SFCs provide a unique order of the cells. This implies that each cell C_{i+1} shares exactly one edge with C_i.*

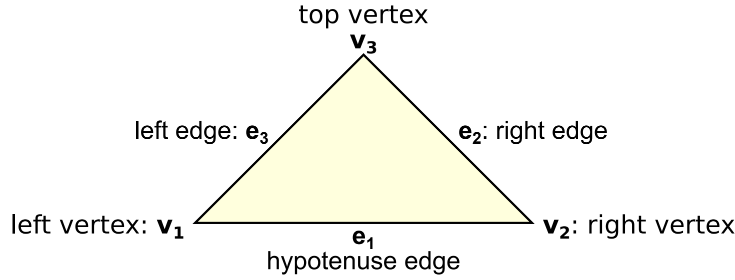


Figure 4.3: Normalized triangle with enumerated edges e_i and vertices v_i. This normalization is similar to the reference space used in the DG simulation. Here, we use it for a unique enumeration of all communication primitives without considering the orientation of the triangle.

Proof: The proof is given by the recursive refinement grammar based on the Sierpiński SFC (see Fig. 4.2): the SFC pierces exactly one edge which is inserted by newest vertex bisection. This is the edge shared by the two child triangles for a refined cell. ∎

The edges of each triangle in its normalized orientation (see Fig. 4.3) are enumerated e_1, e_2 and e_3 by starting at the hypotenuse and then enumerating the triangle legs in counter-clockwise order yielding the unique identifiers

$$\mathbb{E} := \{e_i | 1 \leq i \leq 3\}. \tag{4.7}$$

The vertices are enumerated similarly starting at the vertex left to the hypotenuse, resulting in unique identifiers

$$\mathbb{V} := \{v_i | 1 \leq i \leq 3\}. \tag{4.8}$$

4.3.2 Communication access order and edge types

We continue with the requirements of efficient data exchange between cells. Using the Sierpiński SFC, we can exchange data among adjacent cells via pushing and fetching data to and from a stack system (see next Section).

Using such a stack-based communication demands for (a) particular *access order* in each cell and (b) an *edge-type labeling*:

(a) **Access order:** This order is directly related to the SFC induced order of the interfaces shared by adjacent cells and whether the communication edge or vertex, further referred to as *communication primitives*, is either on the *left* or *right side* of the SFC:

$$\mathbb{S} := \{left, right\}$$

A distinction between left- and right-handed communication primitives is required for our correct stack-based communication system.

We follow the suggestion for an access order of edge and vertex primitives in [BSVB08], and write down the order in a formal way including edges e_i and vertices v_i with the function $A : (\mathbb{T}, \mathbb{S}) \rightarrow \bigcup_{i \in \{2,3,4\}} (\mathbb{E} \cup \mathbb{V})^i$:

$$
\begin{array}{rlrlrl}
A: & ((K,even),left) & \mapsto & (e_3,v_3,e_2,v_2) & ((K,odd),left) & \mapsto (v_2,e_1) \\
& ((K,even),right) & \mapsto & (v_1,e_1) & ((K,odd),right) & \mapsto (e_2,v_3,e_3,v_1) \\
\\
& ((H,even),left) & \mapsto & (v_1,e_3,v_3,e_2) & ((H,odd),left) & \mapsto (e_1,v_1) \\
& ((H,even),right) & \mapsto & (e_1,v_2) & ((H,odd),right) & \mapsto (v_2,e_2,v_3,e_3) \\
\\
& ((V,even),left) & \mapsto & (e_3,v_3,e_2) & ((V,odd),left) & \mapsto (v_2,e_1,v_1) \\
& ((V,even),right) & \mapsto & (v_1,e_1,v_2) & ((V,odd),right) & \mapsto (e_2,v_3,e_3)
\end{array}
\tag{4.9}
$$

To give an example, $A((K,even),left)$ describes the communication primitive access order of the even K triangle type (see top left image in Fig. 4.2). The access order is given by the SFC traversal direction. With the SFC drawn close to the hypotenuse, the left edge e_2 is the first primitive on the left side of the SFC traversal where the SFC enters the triangle. Therefore, primitive e_3 is the first one, followed by the top vertex v_3 and the edge e_2. Since the SFC leaves the triangle via the hypotenuse, the right vertex v_2 is also on the left side of the SFC and, thus, is the last primitive regarding primitives at the left side to the SFC traversal for triangle type $(K, even)$.

Lemma: 4.3.2 *Distinguishing cases of A for different $\mathbb{G}$ for edge communication becomes obsolete.*

Proof: Restricting A to edge communication e_i only, i.e. dropping all v_i, the following condition holds:

$$\forall \alpha \in \mathbb{O}, \beta \in \mathbb{S} : A((K,\alpha),\beta) == A((V,\alpha),\beta) == A((H,\alpha),\beta) \quad \blacksquare$$

(b) **Edge type labels:** We further introduce cell-local labels to each edge:

- The edge is of type *new*, if data is created for this edge. The cell adjacent to this edge is then traversed during the remaining SFC based grid traversal.
- The same edge on the adjacent cell is then marked with *old*. This accounts for data being previously prepared and pushed to a stack system by the adjacent cell and read by the current cell from the very same stack system.
- An edge on a domain boundary is labeled as *boundary edge* demanding for appropriate boundary handling.

The labels for edges are then given by

$$\mathbb{C} := \{n, o, b\}^3 \tag{4.10}$$

for *new*, *old* or *boundary* edge types on the three edges enumerated in anti-clockwise order with the enumeration starting at the hypotenuse, see Fig. 4.3. An example for labeled edges is given in Fig. 4.4. The communication itself can then be realized in a cache-aware manner based on a stack system (see Section 4.3.2).

The communication information C for the first and second child, respectively, due to cell bisection is given by

$$childC_{first/second} : (\mathbb{C}, \mathbb{T}) \to \mathbb{C}^3$$

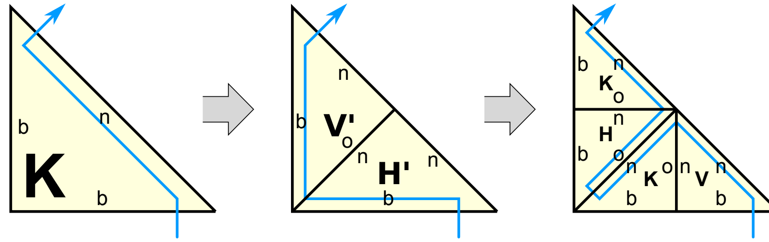


Figure 4.4: Edge types are recursively inherited [SBB12].

with

$$\begin{array}{lllllll} childC_{first}(c, even) & \mapsto & (c_3, c_1, n) & \quad & childC_{first}(c, odd) & \mapsto & (c_2, n, c_1) \\ childC_{second}(c, even) & \mapsto & (c_2, o, c_1) & \quad & childC_{second}(c, odd) & \mapsto & (c_3, c_1, o) \end{array} \tag{4.11}$$

only considering edge communication information so far.

4.3.3 Edge-based communication and edge-buffer stack

We use two stacks, the *left* and *right* communication stack which we continue to use for our edge-based communication, see e.g. [BRV12]. This communication is based on the inherited edge types and uses push and pop operations acting on stacks. A *push* operation is applied for edges of type *new*, storing new data on the stack which is to be fetched by a *pop* operation from another cell for the corresponding edge of type *old*.

With this push and pop operations to/from the respective stacks, this yields our *algorithm for edge-based communication*: For each traversed leaf-node, we apply push and pop operations on the *left* and *right* communication stack. The stack access is specified with our access order $A : (\mathbb{T}, \mathbb{S})$. The information whether data has to be read (pop) or written (push) to the stack is given by the edge types $\mathbb{C}$.

During a *forward traversal*, this communication scheme is able to transfer information to cells subsequently traversed by the current grid traversal. Due to this propagation direction only towards subsequently traversed cells, we require an additional *backward traversal* for transferring data to cells in the other traversal direction. Therefore we require a grammar with reversed access order, given by changing *push* and *pop* operations to communicate in the reversed order and also by reversing the node and edge communication order from function A.

Data forwarded during the forward traversal can be either processed directly, e.g. by updating cell-local data or can be stored to an *edge buffer stack* with data being read from this buffer during the backward traversal. An example of the latter approach is illustrated in Fig. 4.5.

4.3.4 Vertex-based communication

For applications such as node-based flux limiter [KT04], finite element methods [Sch06] and visualization (see Section 6.4.2), efficient data transfer via vertices is required.

We use such vertex-based communication schemes for a visualization of a water surface based on a DG shallow water simulation. Since DG discretizations are discontinuous at the cell boundary, a direct visualization of cell-wisely reconstructed water surface would lead to gaps in the surface. To overcome this non-intuitive visualization distracting the observer due to visible background color, we can construct a closed surface for visualization at each cell's boundary, see Section 6.4.2 for details on the surface reconstruction.

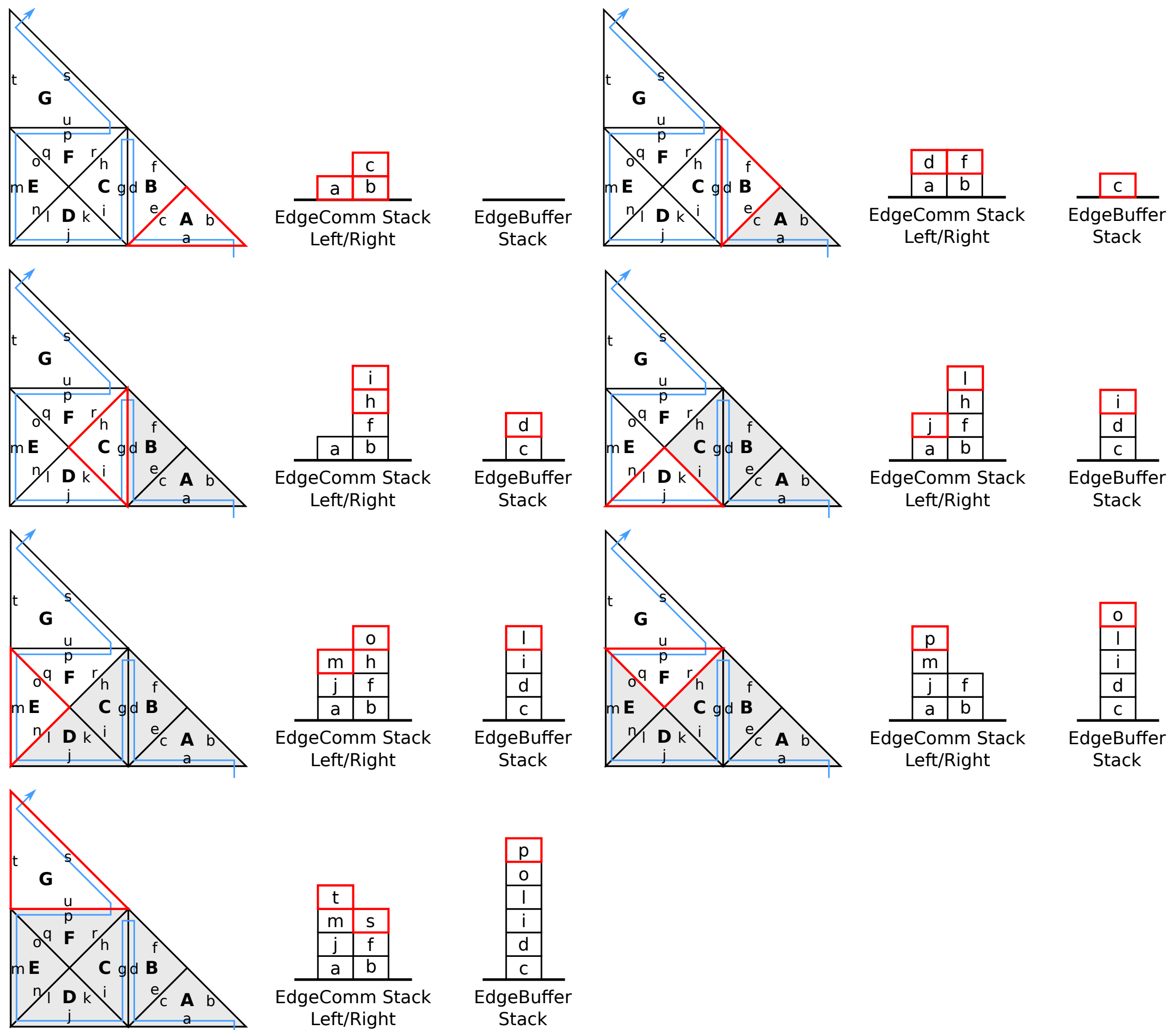


Figure 4.5: Illustration of an edge-based communication scheme for successively traversed leaf cells with the domain boundary edges labeled with type *new*. The data to be transferred is drawn close to each edge. The edge types and communication order is inferred based on the rules given in Section 4.3.2 and are not visualized. For the first leaf triangle A, the edge data b and c are stored in the SFC order to the right communication stack and the edge data a to the left communication stack. In the second leaf triangle B, c is first read from the communication buffer and stored to the edge communication buffer to be processed during the backward traversal. Then, the data f associated to the edge right handed to the SFC is stored to the respective right edge communication stack and data d to the left one. After processing the communication data for the last cell G, all communication data for edges pierced by the SFC are stored consecutively on the edge buffer stack.

Edge communication with each edge shared by up to two cells requires *storing* of, *receiving* of and running *computations* on pairs of edge communication data only. For vertices, more than two cells typically share a vertex. Therefore, additional access policies are required:

Vertex touch policies:
During a grid traversal, vertices shared by other cells are either accessed the first, the last or between the first and last time. We follow the terminology *first*-, *middle*- and *last*-touch [Nec09, Wei09]:

- *first-touch*:
  The vertex is accessed for the first time of the traversal.
- *middle-touch*:
  The vertex data is assumed to be alrady first-touched with the last touch pending.
- *last-touch*:
  This vertex data is accessed for the last time of the traversal.

Instead of fetching an element from the stack, updating it and pushing it back, we only fetch the reference of this element and use the reference to update this data. This does not strictly follow the algorithmic and formal stack access pattern, but yields less operations and thus more performance.
We just described different vertex access policies involved during grid traversal but still miss the knowledge which policy to apply. These policies are inferred based on the edge type labels only, with

$$\begin{aligned} \mathcal{P}: \quad & (\mathbb{C} \times \mathbb{C}) \rightarrow \mathbb{P} \\ & (e_i, e_j) \mapsto \{first, middle, last\} \end{aligned} \tag{4.12}$$

and both edges communication types (e_i, e_j) spatially touching the vertex as parameters. The vertex-access policy is then determined in the following way:

- (new, new): first-touch policy
  If both edges are of type new, the corresponding vertex data on the vertex communication stacks cannot exist since adjacent cells were not visited yet. This is due to the continuous Sierpiński SFC curve: The curve continues traversing the cells sharing the vertex with (new, new) edge types. This yields a first touch policy.
- (new, old) or (old, new): middle-touch policy
  An edge of type old denotes an already visited adjacent cell, thus the vertex data on the stack is already allocated. An edge of type new concludes the adjacent cell to be visited and the vertex data thus still required to remain on the stack. Thus this vertex data is accessed with a middle touch policy.
- (old, old): last-touch policy
  In case that the vertex is not shared with successively traversed cells, using similar arguments than for the (new, new) edge type constellation, it is touched for the last time.

This also leads to the order of pair-wise edge communication types not being relevant for vertex access policy, yielding

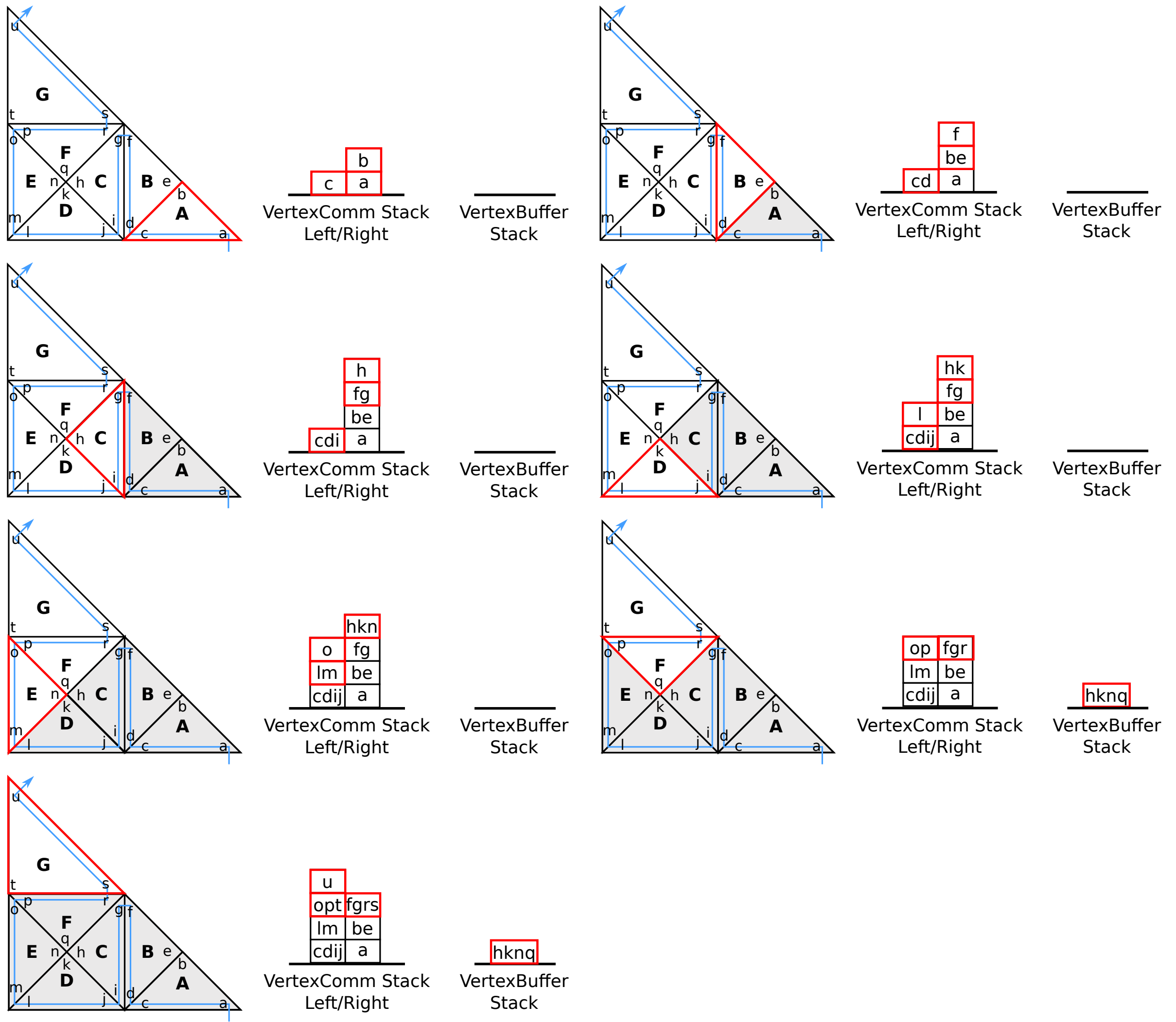


Figure 4.6: Illustration of vertex-based communication scheme for successively traversed leaf cells with the domain boundary edges labeled with type *new*. The vertex data to be transferred is depicted close to each vertex.

$$\mathcal{P} : (e_i, e_j) \mapsto \begin{array}{lll} \textit{first} & \textit{if} & (e_i, e_j) \in \{(\{n, b\}, \{n, b\})\} \\ \textit{middle} & \textit{if} & (e_i, e_j) \in \{(\{n, b\}, o), (o, \{n, b\})\} \\ \textit{last} & \textit{else.} & \end{array} \tag{4.13}$$

Finally, using the communication access order and edge types derived for each leaf element (see Section 4.3.2), the vertex-based communication for semi-persistent vertices via the stack system can be derived and applied with the information given above. An illustration is given in Fig. 4.6.

We like to emphasize, that we optimized the inference of the vertex access policies with a code generator, see Sec. 4.10.1. Hence, there are no if-branchings involved to distinguish between the different access policies presented above.

4.4 Classification of data lifetime

We continue by showing variants of how different data is accessed with a stack system.

(a) **Persistent access:** Persistent data is never released during a traversal by either updating the associated data only or fetching it from one buffer and pushing it to another one.
Examples:
Structure stack: With the stack data handled during a *forward* and *backward traversal*, fetching data from one and pushing all data to the other stack such as the structure stack can keep the grid structure persistently in memory.
Cell data stack: This stack stores the data which is stored for every cell, e.g. the DoF stored for the DG simulation. This data can be kept persistently either by updating the data on the stacks or by copying the data from one source stack to another destination stack and using the destination as the source stack in the next traversals.

(b) **Semi-persistent access:** For this type of access, we distinguish between *creating access* and *clearing access*, respectively, for the forward and backward traversal:

- **Creating access:**
For communication via edges and vertices, the data is first *created* with a forward traversal and stored to an additional buffer for reusage during backward traversal.

  Examples:
Flux preprocessing: Nodal quadrature points on the edges are stored during the forward traversal[1].

  Vertex computations: Storing surface vertex data, e.g. for visualization: initialization and updating such data stored at vertices shared by cells.

- **Clearing access:**
A *clearing access* starts with an already filled buffer and reads the data resulting in an empty buffer.
Examples:
Flux postprocessing: After the forward traversal and flux computations, the flux updates are fetched from the buffers and are further processed with time step integration for the cells.

  Vertex computations: With the vertex data stored to the buffer system during the forward traversal, it is fetched from the buffer system for processing, e.g. writing vertex data to a vertex buffer array, resulting in an empty buffer system.

(c) **Non-persistent access:**
This access starts with an empty stack and finishes with an empty stack. Thus, it is an interplay of creating and clearing accesses.

Adaptivity information: To refine or coarsen cells, adaptivity information requesting refinement or coarsening for cells is forwarded to adjacent cells via a stack system with the adjacent cell's adaptivity states immediately updated. After either the forward or backward traversal is finished, the stack is empty.

[1]Depending on the implementation, this can also be a non-persistent access by moving the data to another buffer

This work puts its focus on data which has persistent cell- and structure-stacks only. The other stacks are assumed to be semi-persistent such as the edge communication data. Here, we describe two ways how to handle persistent edge and node data with a stack system:

- The first approach is *storing the node and edge data in cell data*, resulting in as many duplicates as cells share the node or edge primitives. This approach is directly applicable with this development by mapping cell data to edges and vertices.
- A second approach uses *stack systems* for storing node and edge data only once and separated from the cell data. This avoids storing duplicated data in grid cells resulting in less memory consumption. The Peano framework [Nec09,Wei09] uses a storage system for such persistent vertices. An implementation with the serial version using the Sierpiński SFC can be e.g. found in [Vig12] and is not part of this thesis.

4.5 Stack- and stream-based simulation on a static grid

We continue with a concrete description on how we execute a stack- and stream-based simulation on a static grid. Due to the static grid, we assume the grid structure already stored in the structure stream.

4.5.1 Required stacks and streams

In order to run a simulation for solving the hyperbolic equations with our stack-based communication system, we continue determining further buffer requirements. We introduce different stack types:

- *Communication stacks* $\mathbf{S}_{lr}$: Two stacks are required for communication patterns via edges or vertices due to stack operations distinguishing between the left and right side of the SFC.
- *Buffer stack* $\mathbf{S}_t$: This stack is used for semi-persistent data, e.g. to store edge communication data during the forward traversal and fetch it from the buffer to perform the time step integration.
- *Traversal persistent stacks* $\mathbf{S}_{fb}$: Two stacks are used for forward and backward traversal direction fetching data from one stack and pushing it to the other stack.

Then, the basic stack for grid storage, usable for a pure grid traversal without any computations, is given by

Stack S	Description of purpose	Classification
$\mathbf{S}_{fb}^{structure}$	Grid representation	persistent

Depending on the simulation requirements, additional stacks are used. For a simulation with a DG method on a static grid and flux computations requiring an edge-based communication, this leads to the following additional stacks:

Stack S	Description of purpose	Classification
$\mathbf{S}_{fb}^{simCellData}$	Storage for cell data (e.g. DoF)	persistent
$\mathbf{S}_{lr}^{simEdgeComm}$	Edge data communicated to adjacent cells	semi-persistent
$\mathbf{S}_{t}^{simEdgeBuffer}$	Buffer for data exchange via edges	semi-persistent

The buffer $\mathbf{S}_t^{simEdgeBuffer}$ is required to buffer the communication data during the forward traversal. This data from adjacent cells is then fetched during the backward traversal.

4.5.2 DG simulation with stacks and streams

With our knowledge on the basics of a hyperbolic simulation, we are going to assemble a simulation with the stack- and stream-based approach introduced in the previous Sections.

For flux computations, we do not follow the approach of splitting the flux computations as suggested in [BBSV10] which basically avoids computing the flux based on the DoF on each cell's edge. Such an approach is not applicable with the Rusanov flux solvers from Section 2.10 as well as many other solvers.

A single time step is then computed with the following algorithmic pattern using a forward and backward traversal.

Forward traversal:

(a) **Storing flux paramters**: For each cell traversed along the SFC, we distinguish between edge types (see Section 4.3.2).

- *new*: For edge types *new*, the DoF on an edge required for the flux evaluation are first computed (see Section 2.7) and then communicated to the cell adjacent to the edge. Due to the stack-based communication, each edge of type *new* is followed by a corresponding traversal of the cell adjacent to the edge.
- *old*: For each edge type *old*, the previously written flux parameters are read from the communication stacks and pushed to the semi-persistent stack $\mathbf{S}_t^{simEdgeBuffer}$, directly followed by a push of the cell-local edge coefficients. *This operation converts non-persistent data to semi-persistent data.*
- *boundary*: In case of a boundary condition, the parameters for the flux computation are reconstructed based on the selected type of boundary condition (see Section 2.11) and are, together with the corresponding DoF of the local edge, pushed to the $\mathbf{S}_t^{simEdgeBuffer}$.

Additionally to the flux parameters, the *inner cell radius* is also transfered for computing the *CFL condition* (see Section 2.13) to allow a time step size computation which depends on the inner cell radius of each cell.

Traversal-intermediate processing:

(b) **Compute fluxes**: After storing the cell size as well as all flux DoF consecutively on the *simEdgeBuffer* stack, the fluxes are evaluated and the flux updates written to the same buffer storage (see Section 2.10).

(c) **Compute time-step size**: Based on the flux computations, the maximum time-step size is directly derived from the CFL condition, the maximum wave speed and the cell size (see Section 2.13). The cell size, in addition to the flux parameters, has also been stored to the edge buffer stack.

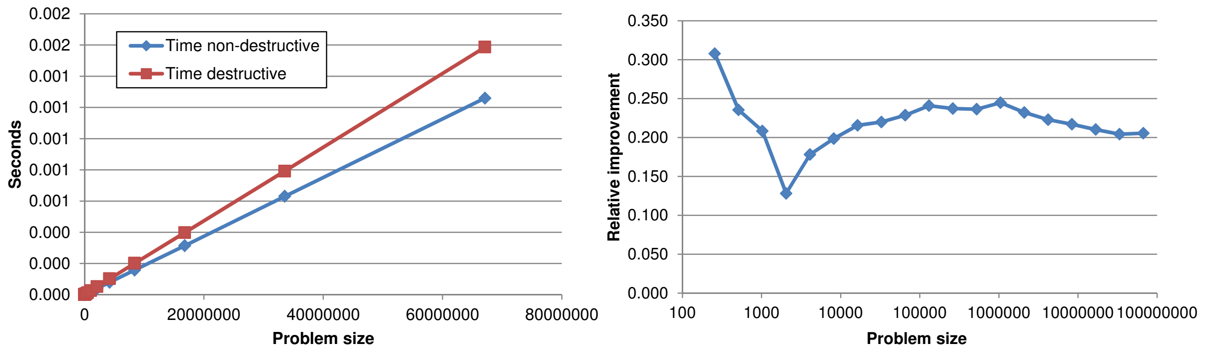


Figure 4.7: Benchmark for non-destructive streams vs. destructive streams with different stream sizes showing robust improvement for larger problem sizes with non-destructive streams.

Backward traversal:

(d) **Receive flux updates** (Section 2.7): During the backward traversal, for each edge type *new*, the *pair of flux-updates* on the shared edge is read from the buffer. The flux update associated to the currently traversed cell is then used for the time stepping and the other flux update is pushed to the communication stacks to be processed by the adjacent cell.

Flux updates for edges of type *old* have previously been pushed to the corresponding left or right edge communication stack and are read from the respective stack.

(e) **Advance the time step on the data stored at each cell** (Section 2.14): The flux-updates for each edge of a cell are available either via the edge buffer for edges of type *new* or via the stack system for edges of type *old*. Thus the cell can be advanced in time based on the flux updates and the computed time-step size.

An alternative approach for flux computations presented in the traversal-intermediate processing is computing the flux updates during the first forward traversal before pushing the flux parameters to the edge buffer (see [SBB12] for an implementation). This avoids the mid-processing and, thus, also additional bandwidth requirements; However, it does not yield the optimization of vectorized flux evaluation for finite volume simulations (see Section 4.10.4) and hence was not considered in this work.

4.5.3 Non-destructive streams

With a software concept using input- and output-streams and temporary stacks only (cf. [MRB12, Nec09, Wei09]), all the data has to be read from one memory location and written to another memory location. Among others, this results in increased cache utilization due to additional memory being accessed and additional cache blocks used. Another performance aspect are additional index computations for the second stack.

We tested this hypothesis with two different benchmarks based on an array of size n. Both benchmarks operate on an array of integers and successively increment the stored values. The first benchmark operates in-situ, reading a floating point value, incrementing it and writing the data back to the same array entry, whereas the second benchmark writes the result to an additional array.

Figure 4.7 shows results for both benchmarks with different block sizes. It suggests that non-destructive streams are mandatory for memory performance. This leads to a robust performance improvement of more than 20% for larger streams using non-destructive streams.

Thus, we use an approach with non-destructive access to our stack structures and further motivate this by highlighting possible applications in our algorithm:

- *Avoid generation of structure stacks for reversed traversal*:
  Writing the structure stream to the memory with an additional memory buffer leads to increased pullution of the cache despite no changing grid. With adaptivity traversals introduced in the next section, several forward- and backward-traversals are executed obviously without requiring reconstruction of structure stacks during each traversal since the structure is fixed. Hence, we reuse the structure stacks.

- *In-situ cell updates for each time step*:
  During the forward traversal, an input/output stream would lead to obsolete copy operations of data stored per cell which does not require any modifications. Such data can be e.g. the Jacobi matrix, vertex coordinates or distorted grids.

This yields two different kinds of non-destructive streams. Assuming e.g. a grid traversal in forward direction accessing cell data, a *top-down stream* is used to iterate over the stack data top-down. A *bottom-up stream* can be used in a similar way to start iterating over the data from the very first element on the bottom of the stack. All these stream operations *do not remove or add* any data to stack, they only update the stack entries.

Optimizing our input/output scheme with these non-destructive streams, this modifies our stack system used for the simulation on static grids (Sec. 4.5.2) in the following way:

Forward traversal:

- *Structure input*: top-down stream (read only (RO))
  The forward structure stack is read with a non-destructive top-down stream which allows for being reused by successive traversals.

- *Cell data*: top-down stream (RO)
  The cell data is read with a non-destructive top-down stream with read access only.

- *Structure output*: stack (write-only (WO))
  The backward structure stack is cleared before the traversal and written with standard push operations to create the structure information for a backward traversal. In case of no modification of the grid structure and with the backward structure stack information already existing, we can skip creating this structure information.

Backward traversal:

- *Structure output*: NONE
  No structure output is required since the forward structure stack *already exists.*

- *Cell data*: bottom-up stream (read-write (RW))
  The backward structure stack is read with a non-destructive top-down stream with read-write access allowing update of cell data.

- *Structure input*: top-down stream (RO)
  The backward structure stack is read with a stream access. Since it was sucessively pushed post-order to the stack, it has to be read top-down.

We just presented the modifications of our stack system to a non-destructive system with a static grid. Next, we continue with the adaptivity traversals.

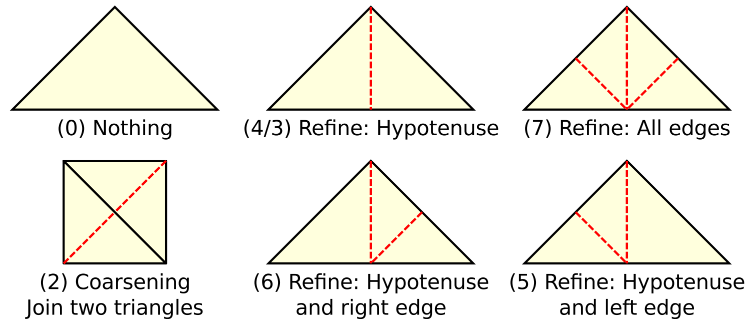


Figure 4.8: Different adaptivity states of each triangle. The red dashed lines represent the new edges of the refined triangle or the removed edges for a coarsening operation joining two triangles [SBB12].

4.6 Adaptivity

We introduce two additional stack systems for adaptivity. The first one accounts for the current adaptivity state in each cell, the second one for the adaptivity communication information to create a conforming grid, a grid without hanging nodes, by inserting additional edges.

Stack or stream S	Comment
$\mathbf{S}_{lr}^{adaptiveEdgeComm}$	Left and right stack for edge communication information to transfer the three different adaptivity markers $\mathbb{M}$
$\mathbf{S}^{adaptiveState}$	Adaptivity state for each cell, see Fig. 4.8

The stacks $\mathbf{S}_{lr}^{adaptiveEdgeComm}$ are then used to forward three different adaptivity markers

$$\mathbb{M} := \{M_R, M_C, M_0\}$$

to adjacent cells, requesting a refinement on an edge, a coarsening or no adaptivity request.

- *Refinement request:* For adaptivity states inserting edges and, thus, creating a hanging node, the adjacent cell also has to insert one or more corresponding edges finally avoiding the hanging node. We use a refinement marker M_R forwarding these insertion requests. The receiving cell can then use this refinement marker to switch to an adaptivity state (see Fig. 4.8) avoiding a hanging node.

- *Coarsening request:* The markers M_C forward coarsening requests along the triangle legs only. With the stack-based communication, this can be established by edge-based communication (see Section 4.6.2 for more information).

We store the adaptivity state on a separate stack $\mathbf{S}^{adaptiveState}$. This allows for the computation of the conforming transitions without touching the simulation cell data stack in the middle traversals to reduce memory bandwidth requirements during the adaptivity traversals.

The adaptivity process (for optimizations, see Sections 4.10.3 and 5.8.2) is then given with a three-pass system which we can execute iteratively:

1. *First forward traversal - marking of refine/coarsen operation requests*:
   Based on the adaptivity indicators (See Section 2.12.5), three adaptivity states are introduced: each cell either requests a (3) *local refine* operation on the cell, a (2) *joining* with another cell or (0) *no adaptivity* request, see Fig. 4.8. Adaptivity markers are then forwarded to adjacent cells similar to the middle traverals which are discussed next.

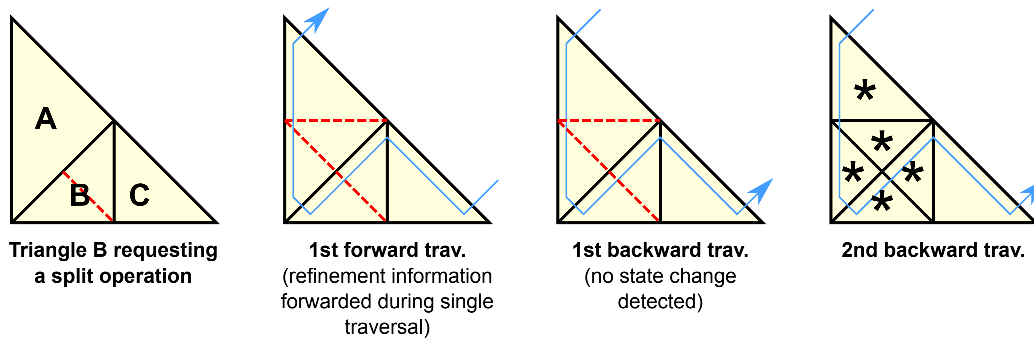


Figure 4.9: Basic adaptive refinement traversals. Adaptivity with a forward and two backward traversals. Note, that the 1st backward traversal was only required to determine a conforming adaptive state.

2. *Middle backward/forward traversals - determine conforming transition*:
   To create a conforming grid, a *conforming transition* of the grid cells to the new grid has to be determined. This requires two markers forwarding refinement and coarsening operations. The middle traversals are done by successively executing backward and forward traversals until a conforming grid state is reached. All edges then have to be marked with the same markers. Further details on the creation of this conforming transition are given in Section 4.6.3.

3. *Applying conforming transition to grid cell data*:
   The last backward traversal applies the transition states determined by refining/coarsening cells and the actual modification of the element data. Regarding the persistent cell data stored on the stack system, this is accomplished by streaming the cell data and inserting or removing additional cells. For interpolation and restriction operations, see Section 2.12.

4.6.1 Refinement

A single forward or backward traversal does not propagate information to all adjacent cells. This is due to edge communication information only propagated to cells which are traversed in the direction of the SFC.

Hence, additional forward and backward traversals are required. The determination of the conforming transition is thus similar to an iterative method: As many iterative smoother steps are done until the residual, in our case the conforming state, is reached.

Considering all possible refinement states required to create a conforming grid, additional split states for a refinement triggered via e_3, e_2 and via both edges are required (See Fig. 4.8 for refinement states (4) to (7)).

4.6.2 Coarsening

Using the knowledge on spacetrees and with the constraint of no hanging nodes, a coarsening operation is only permitted under particular circumstances. Figure 4.10 gives a sketch of a coarsening operation. Two constraints have to be fulfilled for a valid, conforming grid generating coarsening operation:

1. The first constraint is obviously given by the spacetree with only leaf nodes with the *same parent node* allowed to coarsen. Following the recursive grammar from Section 4.1, these

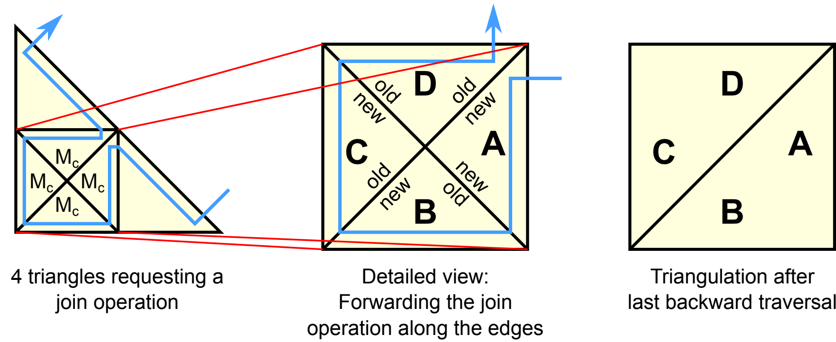


Figure 4.10: Coarsening agreement of cells due to adaptivity traversals. During the first forward traversal, coarsening markers are forwarded along the triangle legs of cells requesting a coarsening operation. A successful coarsening is applied with the last backward traversal [SBB12].

leaf nodes always share an edge with one of the triangle legs and different triangle legs (left or right) for the leaf nodes.

2. The second constraint is raised by the conforming grid property: if only both leaf nodes are joined to a single cell, the resulting triangle can have a hanging node on its hypotenuse. This also demands for coarsening of both cells adjacent to the hypotenuse of the coarsened triangle.

The requirements driven by both constraints lead to a "diamond"-like shape [HDJ04] of triangles involved in coarsening as depicted in Fig. 4.10. Note, that the diamond is created by the touching triangle legs of the four triangles. In case of a coarsening request, a conforming grid can only be obtained if all four involved triangles agree to the coarsening and thus forward coarsening markers M_C along all triangle leg edges. Using the cell traversal order induced by the Sierpiński SFC, a forward followed by a backward traversal is sufficient to propagate the coarsening information: during the first forward traversal, the coarsening request markers M_C are propagated via the triangle legs to the adjacent cells accounting for the diamond like shape. As soon as one cell does not request a coarsening, the marker is not forwarded anymore. In case of the coarsening information not being fully propagated to the last cell in the diamond, the coarsening requests are all invalidated during the backward traversal by the same approach.

In case of a *refinement marker* forwarded via an edge to a cell that requests a coarsening, this coarsening becomes invalid and the state is transferred to one of the corresponding refinement states (3)-(7).

Hence, the computational grid can be a superset of the required mesh.

4.6.3 Termination of adaptivity traversals

One approach for testing for a transition state resulting in a conforming grid was suggested in [BBSV10]. With this approach, the traversal is stopped if no change of state is determined (cf. Fig. 4.9).

This number of conforming grid traversals is limited even with the cascade of edge insertions to avoid hanging nodes. We consider the two possible reasons for cascades.

(a) A *pseudo cascade* forwards the information on the hanging node to the adjacent cell with both considered cells sharing only the hypotenuse. In this case, the forwarding of refinement markers directly leads to a conforming grid once the marker has been forwarded.

(b) A *real cascade* forwards the refinement marker via the hypotenuse to a cell sharing one of its cathetus. This adjacent cell is related to a leaf node one level higher in the refinement tree. Thus, the cascading process can only occur on higher levels, directly leading to the upper limit given by the refinement depth. With the refinement markers forwarded at least once in each pair of forward and backward traversal, this assures the termination of the adaptivity algorithm.

Joining adaptivity and time step traversals

The adaptivity traversals can be combined with the simulation traversals. Since its feasibility was already shown in [Nec09, Wei09], this was not further considered in this work.

4.7 Verification of stack-based edge communication

This section gives a *formal proof* of a valid edge-oriented communication between cells with the stack system using push and pop operations for the Sierpiński SFC. Using push operations on edges of type *new* and pop operations on edges of type *old*, the following theorem holds:

Theorem 4.7.1 (Valid stack-based communication) *Using the communication schemes with the pop and push operations on the communication stack yields correct data transfer to each cell adjacent to the corresponding edge.*

Proof: The proof of the theorem is supported by axiomatic rules closely related to those of the Hoare calculus [Hoa69]. We define inference rules given in the form $\frac{A}{B}$ with A stating the condition which has to hold true and B the condition to replace A. The conditions are given by a tuple of tuples $\{\{P\}S\{Q\}\}^n$ with the precondition P, the code-statement S and the postcondition Q.

We further assume push and pop operations specified via $push(S, a)$, pushing the element a to stack S, and $pop(S, a)$ fetching element a from the stack.

The push and pop operations are extended with the pre- and postconditions

$$\begin{array}{rcl} \{S^k\} & push(S^k, \alpha) & \{S^k = (s_1, s_2, \ldots, s_{|S^k|}, \alpha)\}, \\ \{S^k, S^k_{|S^k|} = \beta\} & pop(S^k, \beta) & \{S^k = (s^k_1, s^k_2, \ldots, s^k_{|S^k|-1})\}. \end{array} \tag{4.14}$$

with the superscript k generating a unique labeling of the stacks.

Reduction and commutative rules

Correct communication: The success of a correct pop operation by applying the grammar and inheritance of types lead to pop operations $pop(S, \alpha)$, *testing for correct fetching* of element α from the top of the stack. This leads to the rule

$$R6: \frac{\{S^k_{|S^k|} = \alpha\} pop(S^k, \alpha) \{S^k = (s^k_1, s^k_2, \ldots, s^k_{|S^k|-1})\}}{\{S^k_{|S^k|} = \alpha\} (S^k, \alpha) = pop(S^k) \{S^k = (s^k_1, s^k_2, \ldots, s^k_{|S^k|-1})\}}. \tag{4.15}$$

This converts the pop operation into the pop operation introduced in Section 4.2, which allows reduction:

Reduction: Correct communication via push/pop to/from stack S^k is recognized and reduced via

$$R1: \frac{\{\}push(S^k,\alpha)\{S^k=(s_1^k,s_2^k,\dots,s_{|S^k|}^k,\alpha)\};\{S_{|S^k|}^k=\alpha\}(S^k,\alpha)=pop(S^k)\{S^k=(s_1^k,s_2^k,\dots,s_{|S^k|-1}^k)\}}{\epsilon} \tag{4.16}$$

This assures correct communication information received by the communication partner by pushing the same data α to the stack and fetching the same data α *from the same stack*. A reduction statement tests for such a correct communication and removes the statements, resulting in ϵ. We are allowed to do so, since the stack is kept in an unmodified state for all other edges accesses.
Commutativity: Commutative property for accessing different communication stacks

$$\begin{aligned} R2: \frac{\{A\}push(S^m,\alpha)\{B\};\{C\}push(S^{n\neq m},\beta)\{D\}}{\{C\}push(S^n,\beta)\{D\};\{A\}push(S^m,\alpha)\{B\}}, \quad R3: \frac{\{A\}push(S^m,\alpha)\{B\};\{C\}(S^n,\beta)=pop(S^{n\neq m})\{D\}}{\{C\}(S^n,\beta):=pop(S^n)\{D\};\{A\}push(S^m,\beta)\{B\}} \\ R4: \frac{\{A\}pop(S^m,\alpha)\{B\};\{C\}pop(S^{n\neq m})\{D\}}{\{C\}pop(S^n,\beta)\{D\};\{A\}pop(S^m)\{B\}}, \quad R5: \frac{\{A\}(S^n,\alpha)=pop(S^n)\{B\};\{C\}push(S^{m\neq n},\beta)\{D\}}{\{C\}push(S^m,\beta)\{D\};\{A\}(S^n,\alpha)=pop(S^n)\{B\}} \end{aligned} \tag{4.17}$$

Those inference rules allow rearranging statements in case that they access different stacks. We emphasize here that this is only valid with constant data on each edge to be pushed to and fetched from stacks. This assumption would not be valid if e.g. data is fetched from one stack and data depending on this one pushed to another stack.

Joining consecutive communication data: In case of pushing or fetching edge data in the correct order with respect to the parent triangle allows to join data stored on the stack:

$$R7: \frac{\{A\}push(S^k,\alpha_{r,1})\{B\};\{C\}push(S^k,\alpha_{r,2})\{D\};}{\{A\}push(S^k,\alpha_r)\{D\}}, \quad R8: \frac{\{A\}pop(S^k,\alpha_{r,1})\{B\};\{C\}pop(S^k,\alpha_{r,2})\{D\};}{\{A\}pop(S^k,\alpha_r)\{D\}} \tag{4.18}$$

Here, α_r represents the edge data of the parent element. The edge data α_r on the hypotenuse of the parent's cell can be split into two edges creating edge data $\alpha_{r,1}$ and $\alpha_{r,2}$. This rule has to be proven to *assure correct refinement operations.*

Application of inference rules

We apply the rules $R1$-$R8$ to show the valid stack access for edge-based communication. Considering only *new* and *old* communication types, the proof is given without loss of generality by *boundary* edges assumed to not modify any communication stacks. For even traversals, the access order for the root triangle only considering the edge access (see Section 4.3.2) is given by

$$A_{(*,even),left} := (e_3,e_2) \qquad A_{(*,even),right} := (e_1) \ . \tag{4.19}$$

Distinguishing between K, V and H traversals is not required (see Theorem 4.3.2). We associate the edge information (a,b,c) to the edges (e_1,e_2,e_3). Splitting a triangle using newest vertex bisection would then splits the communication information on the hypotenuse a by appending $_{,1}$ or $_{,2}$ to the subindex into $(a_{,1},a_{,2})$. This would push two elements to the stack associated to e_1 in the correct order. We assume the inserted edge information to be equal to an arbitrarily choosen d.

We map each new and old communication type of edge data α to $push(S^k,\alpha)$ and $pop(S^k,\alpha)$ to/from stack S^k. To avoid verification for all push/pop combinations, we use a generalization of stack operations

$$pp : \mathbb{C} \to \{push, pop\}$$

for the parent edges only.

For the parent element of type *even*, this leads to the push and pop operations

$$pp(c_3)(S^{left}, c); \quad pp(c_2)(S^{left}, b); \quad pp(c_1)(S^{right}, a);.$$

It has to be proven that by applying the stack operations for both children, the stack modifications are equal to only applying the stack communication patterns of the parent element.

Our communication scheme with type inheritance

$$childC_{first}(c, even) \rightarrow (c_3, c_1, n) \qquad \text{and} \qquad childC_{second}(c, even) \rightarrow (c_2, o, c_1)$$

can then be applied resulting directly in the code for the stack operations being executed:

$$\begin{array}{rll}
 & //\ firstchild; & \\
\{M_1\} & pp(c_3)(S^{left}, c); & \{N_1\}; \\
\{M_2\} & pp(c_1)(S^{right}, a_1); & \{N_2\}; \\
\{\} & push(S^{right}, d); & \{S^r = (s^r_1, s^r_2, \ldots, s^r_{|S^r|}, d)\} \\
 & & \\
 & //\ secondchild; & \\
\{M_3\} & pp(c_2)(S^{left}, b); & \{N_3\}; \\
\{S^r, S^r_{|S^r|} = d\} & pop(S^{right}, d); & \{S^r = (s^r_1, s^r_2, \ldots, s^r_{|S^r|-1})\} \\
\{M_4\} & pp(c_1)(S^{right}, a_2); & \{N_4\};
\end{array}$$

Using commutative rules (R2-R5), we can exchange the stack operations to

$$\begin{array}{rll}
\{M_1\} & pp(c_3)(S^{left}, c); & \{N_1\}; \\
\{M_3\} & pp(c_2)(S^{left}, b); & \{N_3\}; \\
 & & \\
\{M_2\} & pp(c_1)(S^{right}, a_1); & \{N_2\}; \\
\{\} & push(S^{right}, d); & \{S^r = (s^r_1, s^r_2, \ldots, s^r_{|S^r|}, d)\} \\
\{S^r, S^r_{|S^r|} = d\} & pop(S^{right}, d); & \{S^r = (s^r_1, s^r_2, \ldots, s^r_{|S^r|-1})\} \\
\{M_4\} & pp(c_1)(S^{right}, a_2); & \{N_4\};
\end{array}$$

and by applying the reduction rule, this yields the *half-diamond operations*

$$\begin{array}{rll}
\{M_1\} & pp(c_3)(S^{left}, c); & \{N_1\}; \\
\{M_3\} & pp(c_2)(S^{left}, b); & \{N_3\}; \\
 & & \\
\{M_2\} & pp(c_1)(S^{right}, a_1); & \{N_2\}; \\
\{M_4\} & pp(c_1)(S^{right}, a_2); & \{N_4\};
\end{array} \quad . \tag{4.20}$$

Further applying our reduction rule R7/8 completes the first part of the proof since the reduction is only achieved in case of an exchange of data in correct order and correct order of push operations. This leads to

$$\begin{array}{rll}
\{M_1\} & pp(c_3)(S^{left}, c); & \{N_1\}; \\
\{M_3\} & pp(c_2)(S^{left}, b); & \{N_3\}; \\
 & & \\
\{M_2\} & pp(c_1)(S^{right}, a); & \{N_4\};
\end{array}$$

These stack operations on the data a, b and c are equal to the stack operations of the parent cell, assuming that this is the leaf cell. For *odd* element types, the verification is analogously given.

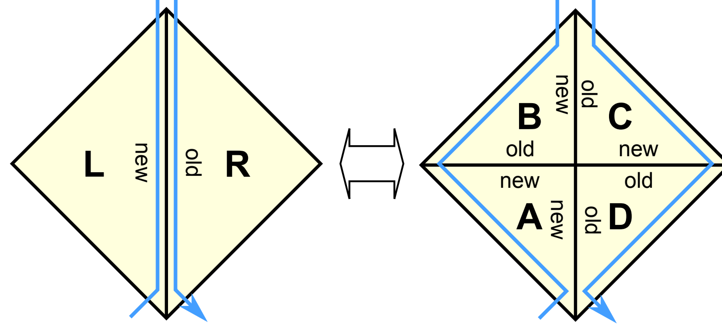


Figure 4.11: Coarsening and refine operations with diamond shaped structure

With our grids generated by the spacetree, we still have to prove the stack system for an entire grid. Without loss of generality, we only present this proof for diamond-shaped grid areas by applying coarsening rules, see Fig. 4.11.

We assemble two *half-diamond operations* (see Eqs. (4.20)) to operations of a full diamond and apply the coarsening operations. With operations on the right stack access represented by the symbol $[X]$, this yields the following code

$$
\begin{array}{lll}
\{M_1\} & pp(c_3)(S^{left}, b); & \{N_1\}; \\
\{M_2\} & pp(c_2)(S^{left}, c); & \{N_2\}; \\
\\
\{\} & push(S^{right}, a_1); & \{S^r = (s^r_1, s^r_2, \ldots, s^r_{|S^r|}, a_1)\}; \\
\{\} & push(S^{right}, a_2); & \{S^r = (s^r_1, s^r_2, \ldots, s^r_{|S^r|}, a_2)\}; \\
\\
 & \ldots [X] \ldots & \\
\\
\{S^r, S^r_{|S^r|} = a_2\} & pop(S^{right}, a_2); & \{S^r = (s^r_1, s^r_2, \ldots, s^r_{|S^r|-1})\}; \\
\{S^r, S^r_{|S^r|} = a_1\} & pop(S^{right}, a_1); & \{S^r = (s^r_1, s^r_2, \ldots, s^r_{|S^r|-1})\}; \\
\\
\{M_7\} & pp(c_3)(S^{left}, d); & \{N_7\}; \\
\{M_8\} & pp(c_2)(S^{left}, e); & \{N_8\};
\end{array}
.
$$

Applying the commutative and reduction rules, we can further reduce both push and pop operations to

$$
\begin{array}{lll}
\{M_1\} & pp(c_3)(S^{left}, b); & \{N_1\}; \\
\{M_2\} & pp(c_2)(S^{left}, c); & \{N_2\}; \\
\\
\{M_7\} & pp(c_3)(S^{left}, d); & \{N_7\}; \\
\{M_8\} & pp(c_2)(S^{left}, e); & \{N_8\};
\end{array}
$$

and with recursive induction, this results in a validated stack access. ∎

Theorem 4.7.2 (SFC stack communication order) *We consider a traversal with stack-based communication and communicate the cell identifiers I_i via edges. Here, the identifiers I_i are increasing by traversing the cells following the SFC. Then it holds that at any time during the traversal all identifiers are ordered on the communication stack.*

Proof: *By communicating cell identifiers to adajcent cells, we follow the SFC. Then all push operations push identifiers of cells with the same or higher index. With the previous proof, we also assured the correct communication.* ∎

We just introduced a method for validating the correct communication via the left- and right stacks for two-dimensional triangular grids. Such a way of validation can play an important role on the search for valid stack-communication properties for other SFC-induced grids such as hexagons or higher-dimensional shapes.

4.8 Higher-order time stepping: Runge-Kutta

With DG simulations and their higher-order spatial discretization, the time-stepping method should be of a similar (higher-)order. To determine the framework requirements, we selected the explicit Runge-Kutta (RK) method. Considering the demands and algorithms shown in Section 2.14, RK methods require storing conserved quantities at particular points in time to V_i and their corresponding derivative D_i. Computing RK time step updates can then be achieved by additional stacks $\mathbf{S}^{V_0}$ for V_0 and $\mathbf{S}^{D_i}$ for D_i computed in each stage [BBSV10]. For an explicit RKn method assuring accuracy up to n-th order with $V_1 := V_0$ due to $a_{k,k} = 0$, we compute each stage $i \in \{1, \ldots, n\}$ with the following algorithm:

Algorithm: RK time stepping

Before iterating over the RK stages, the cell data $\mathbf{S}_f^{simCellData}$ at the current time step is copied to $\mathbf{S}^{V_0}$.

For i *in* $(1, \ldots, n)$ do:

(a) *Compute* $D_i := R(V_i)$:
The simulation cell data stack $\mathbf{S}_f^{simCellData}$ is assumed to be set to V_i (see next step), the conserved quantities computed within the current RK stage. The time step-typical computations including edge communications are executed for $\mathbf{S}_f^{simCellData}$. However, instead of updating the conserved quantities, only the change of the conserved quantities over time is stored to $\mathbf{S}_f^{simCellData}$, yielding D_i.

(b) *Compute* $V_i := V_0 + \Delta t \sum_{j=1}^{n} a_{i,j} D_j$:
After the grid traversal, D_i is copied to $\mathbf{S}_f^{D_i}$. Then V_i is computed and stored to $\mathbf{S}_f^{simCellData}$ by iterating over all elements of the stacks associated to V_0 and D_j and applying equation (2.29).

Finally the time step is computed with

$$\hat{U}(t + \Delta t) := V_0 + \Delta t \sum_{i=1}^{n} b_i D_i.$$

Since we use pointers to mark the beginning of the stack for both push and pop operations, it is not necessary to copy stack data when assigning e.g. $V_0 := U$. Instead of copying the entire stack, we can efficiently swap the stack pointers.

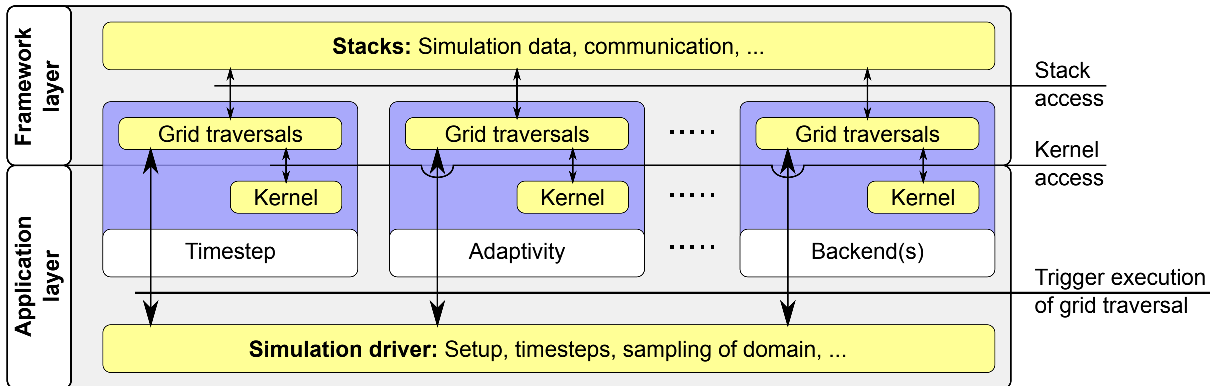


Figure 4.12: Overview of the serial framework design.

4.9 Software design, programmability and realization

Using the introduced communication via stacks and streams, no direct access of adjacent primitives is possible[2]. Also implementing SFC grid traversals with a stack- and stream-based approach, a solution was already suggested in the Peano framework [Wei09]: the application developer has to implement a kernel with the grid traversal calling corresponding kernel methods. However, this framework uses only interfaces for vertices and cells whereas we require additional access to e.g. semi-persistent edge- and node-based data. Therefore, we follow a similar software concept compared to the one of the Peano framework for the vertices and introduce additional interfaces as well as additional framework considerations. The building blocks used for the software design, capable of running single-threaded simulations, are given in Fig. 4.12.

4.9.1 Framework and application layer

For the application layer, we follow the concept *"what to do rather than how to do it"* from Intel's Array Building Blocks [NSL⁺11]. With a kernel-based software design, the framework user thus specifies what to compute on the grid rather than how to traverse and access the grid data.

Framework layer: How to access grid data and how to store grid and communication data is based on the stack- and stream-based approach presented in the previous Sections. Which data is associated to grid primitives and in which way these are accessed (see Section 4.4 on classification of stack data access) typically depends on the simulation requirements and, thus, is only known and has to be specified by the application developer. "How to" manage the data stored on the grid and to hide the way how it is accessed is then the task of the framework layer. "What to do" with the data offered by the framework layer then has to be specified by the application developer.

Application layer: Based on the framework layer providing access to the grid data, the application developer then specifies operations to this grid data in the *kernels*.

4.9.2 Simulation driver

Starting grid traversals and corresponding operations on grid data is initiated by the *simulation driver*. Such traversals can be the simulation time step itself, traversals for adaptivity, writing

[2]Here, we assume that no additional index access for the grid primitives is used

simulation data to permanent storage using backends, etc. This driver is either implemented by the application developer or provided by a default driver from the framework.

4.9.3 Grid traversals and kernels

Grid traversals manage the data access by executing push and pop operations on stacks. These operations depend on the data access demands specified by the application developer. The data references are then forwarded to the kernels to store, update or read the data which is managed by the grid traversal.

4.9.4 Kernel interfaces

For efficiency reasons, we use specialized kernel interfaces based on the grid data access requirements instead of providing all possible grid data accesses, leading to data management overhead in the case that obsolete data is accessed (e.g. if no vertex data is required) or has to be generated. To give an example, vertex coordinates are not required for mid-adaptivity traversals. Next, we describe how to specify the specialized kernel interfaces.

We can specify the grid data access relations of a single traversal in a matrix. This matrix relates the input (row) and output (column) relations between data stored at grid primitives $\mathcal{C} := \{cell, edge, vertex, adaptiveState\}$. The additional *adaptivityState* is required for the adaptivity state information used by the adaptivity traversals and is stored on an additional stack structure.

A single forward or backward traversal can then be represented with the matrix $\mathcal{M}_{(a,b)}$ with $a, b \in \mathcal{C}$ describing the operations executed on data associated to the cell primitives. Then each matrix entry describes the standard grid data access type (*persistent*, *creating*, *clearing*, see Section 4.4).

We give an example for the forward traversal of the simulation which computes the flux parameters. Here, we can specify the kernel interface requirements with the following matrix and only include columns and rows which are relevant:

		Output	
		cell	*edge*
Input	*cell*		*flux creating*
	edge		

Whereas these matrices account for accessing the grid data itself, further kernel parameters can be e.g. cell vertices and cell normals which are computed during the grid traversal on-the-fly. Such kernel parameters are computed during the grid traversal and are not stored in each cell to reduce the bandwidth requirements. We can optimize grid traversals without any requirements of such kernel parameters by avoiding computations of obsolete kernel parameters during the traversal. E.g. traversals such as the middle adaptivity traversal only update the adaptivity state information and do not require cell's vertex coordinates and normals.

Furthermore, we restrict the kernel interfaces for accessing grid data to one or two grid primitives. Hence, interfaces such as storing cell-based data to a vertex and to an edge are not allowed. Those interfaces are described by

$$c_1[_to_c_2[_\mathcal{P}]](parameters)$$

with $c_{\{1,2\}} \in \{cell, \mathbb{E}, \mathbb{V}\}$ and the brackets [] denoting optional interfaces depending on the requirements [3].

[3] This interface description was partly derived in collaboration with Oliver Meister.

1. $c_1(parameters)\{\ldots\}$:
   In case of only c_1 given, this describes kernel interfaces accessing data stored in the primitive c_1, e.g. a cell. Input data stored on primitives such as edges and vertices required to update the data associated to c_1 is then handed over via the parameters, e.g. results of flux computations stored on edges. We differentiate between different input data primitives with a C++ language feature by using *different types* for data stored at primitives.

   For our DG simulation, we use this interface type to update the conserved quantities stored in each cell ($c_1 = \{cell\}$) based on the computed fluxes stored on edges, $\{edge_1, edge_2, edge_3\} \in parameters$.

   For the visualization tasks, such a type of kernel function is also used to generate the triangle data to be used for visualization, based on vertex data, $\{vertex_1, vertex_2, vertex_3\} \in parameters$

2. $c_1_to_c_2(parameters)\{\ldots\}$:
   For an interface name depending on c_1 and c_2, this describes kernel interfaces for storing data to the primitive c_2, based on the primitive c_1. To give an example with our DG simulation, we use it to compute the DoF on the edges with the concrete interfaces given by $c_1 = \{cell\}$ and $c_2 = \{edge_1, edge_2, edge_3\}$. For DG boundary conditions (see Section 2.11), the edge primitives are further extended with boundary edge interfaces $c_2 = \{edge_1, edge_2, edge_3, boundaryEdge_1, boundaryEdge_2, boundaryEdge_3\}$.

   In the case that $c_1 = c_2$, this accounts for updating the data stored on both primitives. This is e.g. used for edges to compute the fluxes after storing the DoF to the edge buffers.

3. $c_1[_to_c_2[_\mathcal{P}]](parameters)\{\ldots\}$:
   For vertex primitives, we require additional access policies $\mathcal{P}$ (see Sec. 4.3.4) differentiating between first-, middle- and last-touch operations.

   These interfaces are then used for the visualization to store the DG surface data to vertex primitives $c_2 = \{vertex_1, vertex_2, vertex_3\}$.

For the parallel implementation, additional framework interfaces are used for the synchronization of the replicated shared interfaces with only partly updated data.

4.9.5 Code generator

All the possible parameter combinations discussed in the previous section lead to a manifold of grid traversal variants.

Creating different optimized kernels using template features of C++ allows for a modification of particular features and requirements of grid traversals; however, it does not allow for the modification of the number of parameters e.g. to disable computing vertex coordinates forwarded via parameters during recursive grid traversal.

As previously mentioned, we decided to create grid traversals using a code generator and discuss it here in more detail. This generator creates optimized code for the grid traversal which is *tailored to the grid traversal requirements* based on the *knowledge of the application developer.*

Configuration parameters for this generator besides T (see Section 4.9.4) are e.g.

- Traversal direction: forward or backward direction.
- Input for kernel parameters: cell vertex coordinates, edge normals, grid depth, etc.
- Adaptivity requirements: min/max limiters for adaptivity traversals.

- Special demands: flux computations, conforming grid indicators, create forward/backward structure stack, etc.

For efficiency reasons, vtable calls have been avoided for the interfaces by inheriting a kernel class to the grid traversal class since overhead optimizations of such vtable calls mainly depend on the used compiler.

Simulation traversal

For a forward DG simulation traversal storing fluxes, we get

		Output	
		cell	*edge*
Input	*cell*		*flux creating*
	edge		

with *flux clearing* as a special tag to create specialized code for flux computations. For the backward traversal the matrix is given by

		Output	
		cell	*edge*
Input	*cell*		
	edge	*flux clearing*	

for executing cell operations with fluxes (edge data) as input.

Adaptivity traversal:

We can then implement the first adaptivity traversals with

		Output		
		cell	*adaptiveState*	*edge*
Input	*cell*		*creating*	*non-persistent*
	adaptiveState			
	edge			

,

thus computing the adaptive state based on cell-wise stored data and forwards adaptivity markers via edges. In case that the adaptivity state flags were created during the backward traversal of the simulation time step, we can skip this forward traversal and start directly with the middle adaptivity traversal.

The middle adaptivity traversals are required to assure a conforming grid and are specified by

		Output		
		cell	*adaptiveState*	*edge*
Input	*cell*			
	adaptiveState		*persistent*	*non-persistent*
	edge			

.

This operates *on the adaptive state flag only.* Thus, no access of simulation cell data is required.

Finally, a last backward traversal refines and coarsens cells depending on the adaptivity state flags while still transferring adaptivity markers along edges. This transfer is required to transfer coarsening "disagreement" information to adjacent cells in case of no middle traversal. Otherwise, this can lead to a coarsening which creates a hanging node. Furthermore, the last adaptivity traversal is specially tagged to execute kernel methods to compute the DoF of the refined or coarsened cells.

		Output		
		cell	*adaptiveState*	*edge*
Input	*cell*	*persistent*		
	adaptiveState	*clearing*		(*non-persistent*)
	edge			

Presistent cell data is updated (refine or coarsen cells) and the adaptive states are cleared from the stacks. Creating non-persistent adaptivity information on the edges is an important component for the meta communication information used for our parallelization which is discussed in Section 5.2.3.

Vertex-based communication for visualization:

To show the applicability of our vertex-based communication, we compute the vertex data based on the data stored in each cell. This results in the following communication matrix

		Output	
		cell	*vertex*
Input	*cell*		*creating*
	vertex		*non-persistent*

which computes the vertex data based on cell-wise stored DoF. Here, the non-persistent vertex access type is implemented with a reduce operation on each vertex. The backward traversal is then given by

		Output	
		cell	*vertex*
Input	*cell*		
	vertex	*clearing*	

finally assembling the vertex information for each cell.

4.10 Optimization

So far, we presented several framework requirements and how they can be accomplished with a serial traversal of the Sierpiński SFC. This section presents several performance improvements with hardware optimizations and algorithmic developments.

4.10.1 Parameter unrolling

Stack push and pop operations depend on the edge (Section 4.3.3) or vertex types (Section 4.3.4) as well as the order in which they are accessed (Section 4.3.2). A straightforward implementation would lead to inheritance of such types and utilizations of if-branchings for distinguishing between whether a push or pop operation has to be done as well as on which stack such an operation has to be executed.

We only consider *new* and *old* edge types without loss of generality and start by describing the issues with the non-optimized version: For different if-branches used to distinguish between these two edge types, a typical branch prediction would lead to a miss rate of 50% in average. We expect this to lead to less performance due to discarding instructions in the pipeline and thus reduced efficiency for the grid traversal.

With our code generator (Section 4.9.5), we can consider the recursive spacetree traversal code be given by

$$S := foobar(P_1, P_2, \dots, P_n, \mathbf{...})\{Source(P_1, P_2, \dots, P_n, \mathbf{...})\}$$

With $Source(\dots)$ representing a source code with P_i as parameters. Here, the considered parameters P_i are within a relatively small parameter range $P_i \in (p_i^1, \dots, p_i^{n_i})$ with n_i the number of possible parameters for the i-th parameter. Such a parameter range can e.g. be the edge types $\{new, old, boundary\}$, the orientation, etc. With the traversal source code, we can create specialized code for each state which we further refer to as *parameter unrolling.* This yields $\prod_i P_i$ possible specializations of the form

$$S_{P_1_P_2_\dots_P_n}(\mathbf{...})\{Source_{P_1,P_2,\dots,P_n}(\mathbf{...})\}$$

which on the first glimpse seems to be a drawback due to increased lines of code and hence increased code size. However, this approach allows avoiding all if-branchings required by different parameters P_i, thus avoiding if-branch mispredictions induced by *new* and *old* edge type labels.

We compared grid traversals with and without parameter unrolling on an Intel(R) Xeon(R) CPU X5690 with 3.47GHz. A regular grid resolution with 22 refinement levels is used with 12 floating point operations in each cell. The vertices are computed during the grid traversal on-the-fly.

On the one hand, this leads to increased size of machine code and therefore a severely increased instruction cache miss rate by a factor of 33.7. On the other hand, the if-branching mispredictions are reduced by a factor of 4.7 in average. Having a look at concrete runtime results shows that the avoidance of if-branching mispredictions outperforms the increased instruction cache misses:

	parameter unrolling	**with parameters**	**performance increase**
GNU Compiler	5.066 sec	6.044 sec	19.3%
Intel Compiler	3.806 sec	5.130 sec	34.8%

Hence, such an optimization is important for kernels with only a few operations.

4.10.2 Recursive grid traversal and inlining

Due to the recursive grid traversal, we should be aware of the overhead of recursive methods. This overhead involves calls of our methods as well as the parameter handling of the methods. Therefore, kernels and traversal methods are prefixed with the `inline` statement, instructing the compiler not to use a function call to this method but to inline the code of the method directly. This aims at avoiding function calls. Similar inline optimizations are also used for the stack and other core operations.

4.10.3 Adaptivity automaton

With the adaptivity traversals presented in Section 4.6, a single traversal only considers the change of adaptivity state, but not the direction of propagation of the adaptivity markers.

In case that a refinement marker is forwarded via an edge and the traversal direction propagates this information to the adjacent cell within the same traversal, this assures the processing of the refinement marker within the same traversal. Thus forwarding and processing the refinement marker in the same traversal is assured for edges of type *new.* If all refinement markers can be forwarded in the traversal, also the grid is assumed to be in conforming state.

		incoming edge marker							
	state t	000	001	010	011	100	101	110	111
no request	**0**	000,**0**	100,**5**	100,**6**	100,**7**	000,**4**	000,**5**	000,**6**	000,**7**
INVALID	**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**
local coarsening req.	**2**	000,**2**	100,**5**	100,**6**	100,**7**	000,**4**	000,**5**	000,**6**	000,**7**
local refine request	**3**	100,**4**	100,**5**	100,**6**	100,**7**	000,**4**	000,**5**	000,**6**	000,**7**
refined: hyp	**4**	000,**4**	000,**5**	000,**6**	000,**7**	000,**4**	000,**5**	000,**6**	000,**7**
refined: hyp, left	**5**	000,**5**	000,**5**	000,**7**	000,**7**	000,**5**	000,**5**	000,**7**	000,**7**
refined: hyp, right	**6**	000,**6**	000,**7**	000,**6**	000,**7**	000,**6**	000,**7**	000,**6**	000,**7**
ref.: hyp, right, left	**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**

Table 4.1: Transistion table for our adaptivity automaton. Each row represents the current transition state t and each column the incoming state change request based on the incoming adaptivity markers M_R (bit is set) and M_0 (bit is unset). The new *transition state* (f,t) is then given by the transition to the adaptivity state t and bitencoded information on edges for which edge markers M_R still have to be forwarded to generate a conforming grid.

We construct an automaton with initial states given by the adaptivity states, transition states based on the adaptivity markers read via edge communication and remaining adaptivity markers to be forwarded. This automaton can be written in a tabular format (see Table 4.1) and is used as follows.

Each grid cell has its own adaptivity *state* $S := (f, t)$ which consists out of the refinement information which still has to be forwarded via each edge in a bit field $f := \{0, 1\}^3$ for edges (e_1, e_2, e_3); t represents one of the adaptivity states from (0) to (7).

Considering refinement operations only, the *transition* function is then based on a table lookup using the current adaptivity state f selecting the row and the input bit fields storing the incoming edge markers selecting the column in Table 4.1. The new *state* of a single cell is then given by the entry (f, t) of the table cell. Bits set in f_{update} represents *the required* refinement information which still has to be forwarded via edges.

(a) *Initialization:* For initialization, the adaptivity state t is set to (0) for no adaptivity request, (2) for a local coarsening request and (3) for a local refinement request. The *forward information bits* f are set to (100) for state (3) avoiding the hanging node, otherwise to (000). The transition state tuple (f, t) is then stored to the adaptivity state stack and updated during the following grid traversals to generate a conforming grid state.

(b) *Transition:* The transition is based on the incoming edge markers for refinement requests which we store to the tuple $T := (e_1, e_2, e_3)$.

Based on the current transition state (f, t), the tuple of the refinement requests t to be forwarded to adjacent cells and the new transition state can be determined via a lookup in the automaton table with the current incoming refinement requests T, yielding (f_{update}, t_{new}).

In case of a cell transition set to coarsening (2), the propagation of the coarsening agreements marker M_C is immediately stopped in case of an incoming marker fetched via a cathetus is not of type M_C.

Forwarding of refinement information: Since the information to be forwarded is already included in f, required refinement markers are also stored in f_{update}. A bitmask is used with *bits set for edges of type old.* This bitmask is ANDed to f to account for edges of type (a) *new* with refinement information propagated during the current cell traversal and (b) *boundary* which do not require forwarding of refinement information.

This adaptivity automaton is able to avoid at most one traversal for serial traversals. However it turns into a crucial component for the optimization with the cluster skipping approach in the parallelization (See Section 5.8.2).

4.10.4 CPU SIMD optimizations for inter-cell computations (fluxes)

With the trend of computing cores requiring operations on data stored in vectors to get close to the peak of floating point operations, utilization of vector instructions is mandatory for HPC-oriented developments.

Regarding the flux computations on the edges for higher-order DG simulations, multiple nodal points for flux evaluation are available on each edge. This allows a straightforward optimization with SIMD operations.

For 0th-th order discretization, however, only DoF of a single node on the edge is stored per edge. Here, SIMD optimizations are still possible for a pair of nodal points e.g. by computing multiple square roots in the Rusanov solver with a single SIMD operation.

In this section, we like to focus on an alternative method for 0th-th order discretizations. We consider, that the nodal-wise given conserved quantities are stored consecutively on the edge buffer stacks. [4] Since the DoF on the edge buffer stack are stored in the format of *arrays of structure* and since SIMD operations typically demand for a *structure of arrays*, the data has to be rearranged. We consider a block of multiple conserved quantities U_i^+, U_i^- for the left and right state of an edge stored to the edge buffer stack:

$$\begin{aligned}
S &= (U_{e1}^+, U_{e1}^-, U_{e2}^+, U_{e2}^-, U_{e3}^+, U_{e3}^-, U_{e4}^+, U_{e4}^-) && (4.21)\\
&= ((q_{e1}^{+1}, q_{e1}^{+2}, q_{e1}^{+3}, q_{e1}^{+4}), (q_{e1}^{-1}, q_{e1}^{-2}, q_{e1}^{-3}, q_{e1}^{-4}), && (4.22)\\
&\quad (q_{e2}^{+1}, q_{e2}^{+2}, q_{e2}^{+3}, q_{e2}^{+4}), (q_{e2}^{-1}, q_{e2}^{-2}, q_{e2}^{-3}, q_{e2}^{-4}), && (4.23)\\
&\quad (q_{e3}^{+1}, q_{e3}^{+2}, q_{e3}^{+3}, q_{e3}^{+4}), (q_{e3}^{-1}, q_{e3}^{-2}, q_{e3}^{-3}, q_{e3}^{-4}), && (4.24)\\
&\quad (q_{e4}^{+1}, q_{e4}^{+2}, q_{e4}^{+3}, q_{e4}^{+4}), (q_{e4}^{-1}, q_{e4}^{-2}, q_{e4}^{-3}, q_{e4}^{-4})). && (4.25)
\end{aligned}$$

Assuming that this is an 8×4 matrix, we can transpose this matrix. This yields four components of the same conserved quantity on either the left or the right side of the edge in each vector:

$$\begin{aligned}
S \quad = \quad & ((q_{e1}^{+1}, q_{e2}^{+1}, q_{e3}^{+1}, q_{e4}^{+1}), && (4.26)\\
& (q_{e1}^{+2}, q_{e2}^{+2}, q_{e3}^{+2}, q_{e4}^{+2}), && (4.27)\\
& (q_{e1}^{+3}, q_{e2}^{+3}, q_{e3}^{+3}, q_{e4}^{+3}), && (4.28)\\
& (q_{e1}^{+4}, q_{e2}^{+4}, q_{e3}^{+4}, q_{e4}^{+4}), && (4.29)\\
& (q_{e1}^{-1}, q_{e2}^{-1}, q_{e3}^{-1}, q_{e4}^{-1}), && (4.30)\\
& (q_{e1}^{-2}, q_{e2}^{-2}, q_{e3}^{-2}, q_{e4}^{-2}), && (4.31)\\
& (q_{e1}^{-3}, q_{e2}^{-3}, q_{e3}^{-3}, q_{e4}^{-3}), && (4.32)\\
& (q_{e1}^{-4}, q_{e2}^{-4}, q_{e3}^{-4}, q_{e4}^{-4})). && (4.33)
\end{aligned}$$

With a typical flux solver such as the Rusanov flux solver, an optimized SIMD evaluation of the conserved quantities is then possible. After flux computations, the data layout has to be converted back from the structure-of-arrays to the arrays-of-structure format. Such a reordering in its generic form is also known as gather and scatter operations, respectively, for packing data into a vector-processable format and inverting this packaging (cf. [EHB+13]). In our case, a matrix transposition was sufficient.

[4] The idea of computing fluxes on this edge buffer stacks was also developed independently by Oliver Meister and Kaveh Rahnema.

As a proof of concept, we conducted benchmarks based on the augmented Riemann solver [Geo06] which were SIMD optimized in [Höl13]. In this work, the SIMD-optimized augmented Riemann solver is implemented with single precision. We compare the SIMD performance improvements with this solver based on simulations with a radial dam break szenario. The simulation is executed with an initial refinement depth of $d = 16$ and with $a = 10$ additional levels of adaptive grid refinement. The SIMD optimization which we used for the benchmarks stores 4 single-precision components in each vector, and we execute the simulation for 100 time steps.

We measure the optimization for the edge communication and adaptivity traversal times given in seconds:

	SIMD disabled	SIMD enabled
Time step	10.86	6.66
Adaptivity	3.99	3.93

The time to compute a time step is improved by 38.7% but the maximal theoretical improvement of 75% was not reached. We account for that by (a) the branch divergence inside the flux solver resulting in non-parallalelized sections and (b) the grid traversal time which was not optimized with the SIMD flux evaluation. The runtime for the adaptivity traversals is not reduced since the SIMD optimized solvers are only used in the time step traversals.

4.10.5 Structure of arrays for cell-local computations

Considering the flux computations of conserved quantities with higher-order basis functions (see Section 2.3), we can use SIMD instructions by storing the weights for the basis functions in a structure of arrays. We store the n weights q_j^i with $j \in \{1, \ldots, n\}$ for a particular conserved quantity i of a cell consecutively in memory. For four conserved quantities, this yields

$$\left(q_1^1, q_2^1, \ldots, q_n^1, \quad q_1^2, q_2^2, \ldots, q_n^2, \quad \ldots \quad q_1^4, q_2^4, \ldots, q_n^4\right).$$

Using this structure of arrays with the matrix formulation of the DG time stepping scheme (see Part II), allows us to compute the time step integration mainly with matrix-vector or matrix-matrix multiplications.

In this work, we followed this structure-of-arrays format for our conserved quantities, however, we did not further focus on optimization of such matrix-matrix computations.

4.10.6 Prospective stack allocation

A straightforward implementation of the stack system with the C++ std::vector class from the standard library would lead to a performance slowdown: an obvious example is given by frequent memory free() and alloc() operations if not pre-allocating sufficient memory and thus re-utilization of already allocated memory. Pre-allocating far more memory than required leads to a severe additional memory consumption, possibly exceeding the available memory.

With a prospective stack size allocation based on an upper limit for the maximum required allocated stack, we can overcome testing for exceeding stack access and avoid any stack resize operations during grid traversals. Using the knowledge on the number of cells c (See [Vig12]) for our simulation domain assembled by a triangle, upper limits for the storage requirements on all stacks can be determined. The number of cells c after all adaptivity traversals on a grid can be derived based on the adaptivity state information before updating the grid structure.

The number of grid cells are increased for each inserted edge and decreased for a pair of coarsening requests stored on the adaptivity state stack. See e.g. Fig. 4.8 with the red-dashed lines representing the inserted edges for the refinement operations (3) to (7).

Here, we infer the upper limits of the required stack storage:

Simulation cell data $\mathbf{S}^{simCellData}$:

The cell data stack size is directly set to c.

Structure stack $\mathbf{S}^{structure}$:

An upper limit of the size of the structure stack is directly given by the recursive structure. With a cell split achieved by replacing 0 with 100 on the structure stack, this creates 2 additional bits for each new cell. Starting with a single cell represented by a stack storing |0 only, i.e. only one entry, this yields

$$\max(|\mathbf{S}^{structure}|) := 1 + 2(c-1) = 2c - 1$$

Edge communication data $\mathbf{S}^{simEdgeComm}$, $\mathbf{S}^{adaptiveEdgeComm}$:

Edge communication data such as $\mathbf{S}^{simEdgeComm}$ for running the simulation or $\mathbf{S}^{adaptiveEdgeComm}$ for adaptivity traversals is limited by recursive bisection of the communication edges. We follow a greedy approach successively splitting triangles targeting at creating as many edges as possible at the triangle boundary at the root spacetree node. This leads to a sequence of maximum edge communication elements E with E_i representing the upper limit of required stack storage for the i-th inserted edge. For the right communication stack, this yields $E^{right} := (1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, \ldots)$ and $E^{left} := (2, 2, 3, 4, 4, 5, 5, 6, 6, 7, 7, \ldots)$ for the left communication stack. With $E_i := max(E_i^{left}, E_i^{right}) = E_i^{left}$, an upper limit is given by

$$\max(|\mathbf{S}^{\{sim,adaptive\}EdgeComm}|) := floor(\tfrac{c}{2}) + 2.$$

By splitting the root triangle into two children and considering the maximum number of communication edges between both children, the cardinality is still less or equal to $floor(\frac{c}{2}) + 2$.

Edge buffer data $\mathbf{S}^{simEdgeBuffer}$:

For the root triangle, at least six edge buffer elements are required due to storing up to six boundary edge data elements on the buffer stacks. With each refinement of child triangles, two additional edges are created for the shared edge and two additional edges are created due to splitting the edge on the hypotenuse. This demands for additional storage of four elements on the stack system. Thus, the maximum number of required edge buffer data for edge data is

$$\max(|\mathbf{S}^{simEdgeBuffer}|) := 6 + 4(c-1) = 2 + 4\,c.$$

Vertex communication data $\mathbf{S}^{vertexComm}$:

For vertex data, only two nodes can be stored on each communication stack. Following a greedy edge insertion focussing on creating as many vertices as possible on the boundary of the domain created by the spacetree, this yields

$$v^{left} := (3, 3, 4, 5, 5, 6, 6, 7, 7, \ldots)$$

$$v^{right} := (2, 3, 3, 4, 4, 5, 5, 6, 6, \ldots)$$

With $max(v_i^{left}, v_i^{right}) = v_i^{left}$, an upper limit is given by

$$\max(|\mathbf{S}^{vertexComm}|) := floor(\tfrac{c}{2}) + 3 \ .$$

Vertex data buffer $\mathbf{S}^{vertexBuffer}$:

For the visualization, the vertex buffer only requires data storage for the vertices inside the domain since the boundary vertex data is stored to the vertex communication stacks. Each insertion of an edge creates one additional vertex. Since those inner vertices can be created only with more than 4 cells and one additional vertex is generated for each new cell, this yields:

$$\max(|\mathbf{S}^{vertexBuffer}|) := \max(c - 4, 0)$$

We like to emphasize, that e.g. a node-based communication for flux limiters of DG simulations can lead to other limits.

Resizing stacks:

We derived maximum stack size capacities to store all data during a traversal based on a given number of simulation cells c. This new snumber of simulation cells was derived with the adaptivity state flags on the stack after the conforming adaptivity state was detected.

Before the last backward adaptivity traversal, we then reallocate the output stacks with the new capacity. The other stacks are reallocated after the last backward adaptivity traversal. This avoids resize operations during all traversals.

However this would still yield a frequent reallocation of stack data if the number of cells changes. Therefore, we introduce two additional padding values: $P^{padding}$ and $P^{undershoot}$. These thresholds are applied in case of exceeding and undershooting a particular number of simulation cells.

A reallocation is done only if $|\mathbf{S}^{simCellData}| < c$ or $|\mathbf{S}^{simCellData}| - P^{undershoot} > c$. This reallocates the stacks for $c + P^{padding}$ simulation cells.

4.11 Contributions

So far, we presented a single-threaded framework for DG simulations based on the Sierpiński SFC with stack- and stream-based communication via edges and vertices. These communication schemes are based on previous research for vertex- [BSVB08] and edge-based [BBSV10] simulations. Our new contributions are summarized here:

- We gave a formal introduction of the Sierpiński SFC including a formal proof of the correct stack-communication system.
- We developed a framework for such communication schemes and introduced clear interfaces offered to the application developer to hide the stack-based communication complexity.
- The code generator does not only lead to an efficient method to generate tailored and thus optimized code based on the user requirements, but also leads to optimizations such as parameter unrolling to avoid most if-branching mispredictions.
- We separated the stacks into their functional utilization, see Section 4.5. This avoids obsolete access of memory e.g. by separation of structure, cell-data and adaptivity state stacks.

- An automaton table considers the propagation direction of adaptivity information.
- SIMD optimizations allow vectorized computation of fluxes for finite volume simulations.
- A prospective stack reallocation for a good balance between memory requirements and frequent stack reallocation is derived.

5 Parallelization

The parallelization of inherently serial code as it is the case for the Sierpiński SFC grid traversal with its stack- and stream-based data management is a challenging task. This chapter is on the extension of the so far serial grid traversal to a cluster-based parallelization approach.

- **Section 5.1: SFC-based parallelization methods for DAMR**
  We start with an overview of common parallelization methods with dynamic adaptive mesh refinement to show differences, but also common aspects of our approach.
- **Section 5.2: Inter-partition communication and dynamic meta information**
  To offer efficient data exchange between our partitions, we developed communication meta information which is represented by a run-length encoding. We present the concept of this encoding, its applicability to edge- and vertex-based communication and how to use it for the dynamically changing grids.
- **Section 5.3: Parallelization with clusters**
  Despite the parallelization of a serial code, the abstraction layers introduced for the serial simulation should be maintained for usability reasons. Also the parallelization should be hidden from the application developer with an appropriate software design. We account for both issues with a cluster-based software solution.
- **Section 5.4: Base domain triangulation and initialization of meta information**
  For the simulation domain itself, a typical requirement is a flexible assemblation of simulation domains of different kind of shapes.
- **Section 5.5: Dynamic cluster generation**
  With the dynamically changing grids, we have to cope with load imbalances. This is accomplished by dynamically generating clusters. With our clusters based on tree-splits, we use tree-splits and -joins and show implicit handling of the communication meta information for these splits and joins.
- **Section 5.6: Shared-memory parallelization**
  The shared-memory parallelization describes the cluster-to-thread scheduling, the dynamic cluster generation strategies and the used threading libraries.
- **Section 5.7: Results: Shared-memory parallelization**
  Results on the shared-memory parallelization with different cluster generation strategies on different platforms are presented in this section.
- **Section 5.8: Cluster-based optimization**
  With clustering at hand, we present different optimization possibilities.

- **Section 5.9: Results: Long-term simulations and optimizations on shared-memory**
  Scalability tests of parallelization concepts are typically executed only for a few time steps of the simulation. Here, we present the impact of long-term simulation runs on the scalability, resulting in a different optimal choice of cluster generation and scheduling compared to the short-term simulations.

- **Section 5.10: Distributed-memory parallelization**
  The extension to distributed memory parallelization is presented in this section. Whereas the RLE meta information solves the efficiency issues of our distributed-memory communication scheme, the cluster-based software design allows straightforward cluster migration.

- **Section 5.12: Results: Distributed-memory parallelization**
  With the distributed-memory parallelization introduced in the previous section, here we present small- and large-scale scalability studies.

- **Section 5.13: Summary and Outlook**
  The last section concludes and highlights the new contributions developed in this thesis.

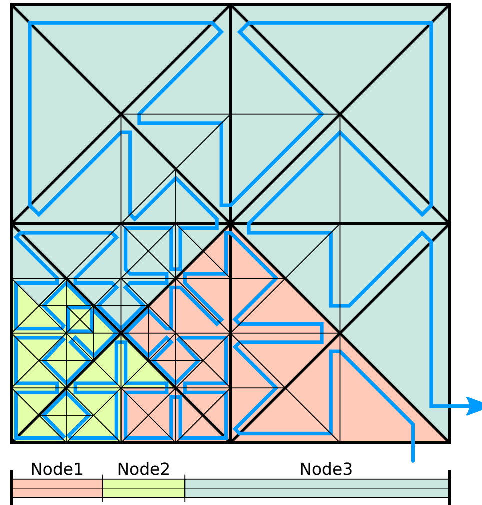


Figure 5.1: Domain partitioning with SFC cuts. Top image: 2D Sierpinski partitioning with each partition given in a different color. Bottom image: 1D representation of the partitioning, each interval representing a partition.

5.1 SFC-based parallelization methods for DAMR

Parallelization of simulations with (dynamic) adaptive mesh refinement has a rich history in scientific computing. SFC-based domain decomposition and load-balancing strategies are considered to be among the most efficient regarding our requirements of a changing grid in each time step (see related work in Section 3.4.1), and we continue with a more detailed introduction to SFC-based domain decomposition methods.

5.1.1 SFC-based domain partitioning

We start with an SFC-based domain decomposition of a discretized domain $\Omega_d = \bigcup_i \{C_i\}$ with cells C_i. By ordering and *enumerating all cells along the SFC*, a partitioning into N non-overlapping partitions $P_k \in \Omega_d$ with $1 \leq k \leq N$ can be achieved: This associates cells to a partition k by generating an interval for each partition with the start cell id S_k and an end id given by the next partition's cell start id S_{k+1},

$$P_k := \left\{ \bigcup_i C_i | S_k \leq i < S_{k+1} \right\}, S_k \in \mathbb{N}^+$$

with $S_{N+1} := |C| + 1$. The communication interfaces $\mathcal{I}$ between two different partitions P_i and P_j with $i \neq j$ are given by a set of hyperfaces

$$\mathcal{I}_{i,j} := \{P_i \cap P_j\}.$$

We further refer to *hyperfaces* created by the Sierpiński SFC to be edges (hyperfaces of dimension $d-1$) and nodes (hyperfaces of dimension $d-2$).

With the spacetree-based grid greneration inducing a serialization of the underlying grid cells with the SFC, there are two common ways on partitioning such a grid:

- **SFC cuts:**
  With *SFC cuts* [Beh05,DBH+05], partitions are generated by cutting the one-dimensional

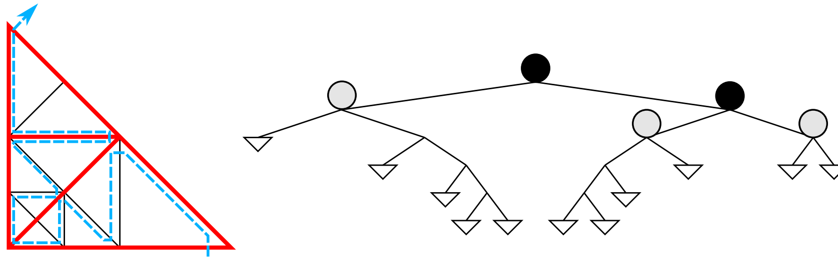


Figure 5.2: Partitioning of a triangular-shaped domain partitioning with tree splits. Left image: domain triangulation with each partition marked with a thick red border. Right image: representation of the tree split domain partitioning with the refinement tree. The triangle nodes represent the cells, the gray-filled circles the subtree's root node.

representation of the SFC into equally sized chunks. This aims at improving the load balancing by cutting the SFC at appropriate positions. An example of generated partitions based on the Sierpiński SFC is given in Fig. 5.1.

- **Tree splits:**
  With the grid generation based on the recursively defined spacetree, we can generate partitions by using the naturally given bisection. A partitioning is then based on tree splits (see also [Wei09]) with cells represented by leaf nodes of *subtrees*, see Fig. 5.2. These subtrees are a special case of the SFC cuts.

The communication, data migration and code-generator-based optimizations (see Sec. 4.10.1) which we derive in this thesis can be applied to spacetree splits and also SFC cuts. Due to historical reasons and the existing code generator for recursive traversals of subtrees, we decided to continue with the parallelization based on the tree splits.

5.1.2 Shared- and replicated-data scheme

We distinguish between parallelization approaches by considering methods with shared and replicated parallelization [SWB13b] [1]. Both methods are based on a domain decomposition into multiple partitions, each partition sharing hyperfaces of dimension $d-1$ or less with adjacent partitions. We refer to these shared hyperfaces as *shared interfaces* dP_k. With each compute unit executing operations and modifying data associated on each partition in parallel, data on these shared interfaces is accessed in parallel and has to be kept consistent. Based on our grid generated with a spacetree, we consider two different data access schemes, each one resulting in a different parallelization approach:

- **Shared data scheme (shared access synchronization):**
  The SFC induces a serialization of the domain data into a stream. With a *shared data scheme* following the SFC input stream, multiple compute units can operate on the same input data stream, but on different chunks of the input stream. Due to accessing the same data, an *access synchronization* to avoid race conditions is required. This would lead to a parallelization approach that requires frequent access synchronizations using e.g. mutices, or spin locks.

[1] Parallelization methods with multiple threads executing operations for each cell (see e.g. [NUW12]) are not further considered since we do not consider them to be scalable due to overheads for executing tasks in each cell.

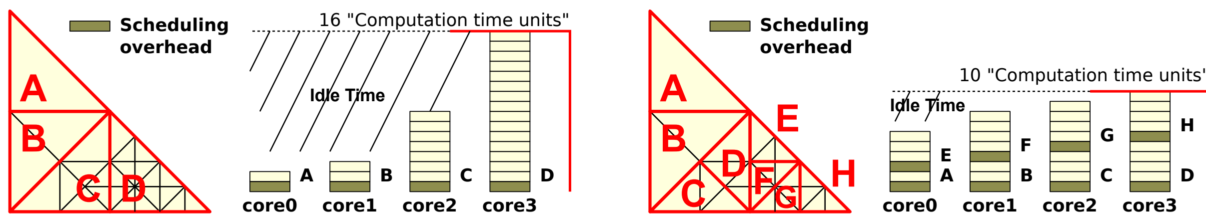


Figure 5.3: 1:1 (left) vs. N:1 (right) scheduling with partitions generated by subtrees. Each yellow block is representative for a single grid cell. The blocks in dark-yellow color represent execution overheads to run computations on a partition. The N:1 scheduling leads to less idle time for typical grid structures of dynamically changing grids [SBB12].

- **Replicated data scheme (replicated data synchronization):**
  Using a *replicated data scheme*, the data on shared interfaces is considered to be replicated. This leads to a parallelization approach with computations on each partition executed massively in parallel without any synchronization, followed by *data synchronization*, e.g. a reduce operation on the replicated data on the shared interfaces (see [Vig12]).

The replicated data scheme with separated data buffers (stacks and streams, e.g.) for each partition is typically used for distributed-memory environments since replicated data can be sent and reduced after the receive operation by using distributed-memory messaging. For shared-memory environments, such communication interfaces are typically not available or, lead to additional overhead. In this work, we developed a run-length encoding of meta information to make the replicated data scheme also feasible on shared-memory systems (see Section 5.2). Our method does not only avoid these overheads for shared-memory systems, but also leads to an elegant solution for distributed-memory parallelization. We continue to use the *replicated-data* scheme in the present work.

5.1.3 Partition scheduling

Once the partitions are generated, several scheduling possibilities exist to assign computations on SFC-based domain partitions to compute units.

- **1:1 scheduling:**
  With a 1:1 assignment of partitions to compute units, each partition of the domain is assigned to a single compute unit. We refer to this as a 1:1 scheduling. Using a stack- and stream-based communication scheme, this approach was taken so far for SFC cuts with the Sierpiński SFC (see e.g. [Vig12]) as well as tree splits with the Peano SFC [Wei09] for distributed memory only, both assigning a compute unit to a single partition.

- **N:1 scheduling:**
  With a partitioning approach based on tree splits, a 1:1 scheduling approach would clearly lead to high idle times due to workload imbalances (left image in Fig. 5.3). An alternative to this approach is massive splitting [SBB12] creating by far more subtree-oriented partitions than there are compute units available, resulting in an N:1 scheduling (right image in Fig. 5.3).

For SFC-cuts, a 1:1 partition scheduling allows an optimized implementation due to avoiding object-oriented overheads with single-threaded MPI (cf. [Vig12]). With the focus of our partition generation based on tree splits, such a 1:1 scheduling would lead to the above mentioned

load imalances. Therefore, an N:1 scheduling as well as an object-oriented software approach gets mandatory to tackle the N:1 load balancing.

5.2 Inter-partition communication and dynamic meta information

With partitions generated by the SFC intervals, we present communication patterns by generating and synchronizing data shared interfaces of partitions with a replicated data scheme. First of all, a replicated data scheme requires independent communication buffers. To account for these replicated data scheme, we use additional stack-based communication buffers for each partition. We continue with a static number of partitions and refer to Section 5.5 for dynamic partition generation.

5.2.1 Grid traversals with replicated data layout

For our Sierpiński stack- and stream-based traversals, we first describe the replicated data layout in an abstract way. It can be implemented with a forward- and backward-traversal in the following way:

(a) *Forward traversal:*
During the forward grid traversal of partition P_i, the edge types of all shared interfaces on the partition are set to *new*. With our grid traversal based on recursion and inherited edge types, the edge types for the shared interfaces can be directly set to *new* and *old* at the sub-tree's root node.

By executing an SFC-based grid traversal, communication data is written to the corresponding edge- or vertex-communication stacks. Since those output buffers are replicated, this access is race condition free. Now, the output stacks are filled with communication data of dP_i.

(b) *Synchronization:*
Working on replicated data, we execute a reduce operation for the data stored at shared interfaces to synchronize the replicated data on the communication buffers. Information on the placement of shared interface data dP_i in memory buffers is provided in Section 5.2.3.

(c) *Backward traversal:*
Finally, a backward SFC traversal is reading the data from communication stacks by setting the edge types of shared interfaces to *old*. This results in reading the communication data on which the reduce operation was executed on.

A parallelization with a replicated data scheme allows the forward traversals being executed on all partitions in parallel and also in arbitrary order. The same holds for the synchronization operation and backward traversals. Therefore, the only required access-synchronizations are between the forward, reduce and backward traversals. So far, this is a similar approach which was also taken in [Vig12].

5.2.2 Properties of SFC-based inter-partition communication

We next discuss important properties by using the Sierpiński SFC with a stack- and stream-based communication.

Lemma: 5.2.1 (Order of replicated data) *After the first traversal, the elements on the communication stack are ordered with their creating SFC cell indices.*

Proof: This theorem directly follows from Theorem 4.7.2 on page 64 due to correct order of the elements on the communication stacks also during the grid traversal. ∎

With communication data ordered with the SFC cell indices, this also leads to additional and for our development mandatory properties regarding the cardinality and uniqueness of data exchange:

Theorem 5.2.2 (Unique adjacent partition) *All communication data for an adjacent partition P_k are consecutively stored on the communication stack. This induces an unique adjacency of partitions.*

Proof: The proof is given by reductio ad absurdum with the communication element order from Lemma 5.2.1. Let at least two consecutively stored non-empty communication data sets S_1 and S_3 on the communication stack shared with an adjacent partition P_a and a set S_2 associated to P_b be given. S_1, S_2 and S_3 are consecutively stored on the communication stack. Further, let the communication data consist out of SFC-ordered cell indices. Then, there has to be at least one partition P_b with $a \neq b$ accessing the elements stored between the communication elements which are stored for partition P_a on the communication stack. However, this leads to a contradiction to Lemma 5.2.1, page 84: with all cell indices from P_a within a particular SFC interval, the cell indices from P_b then have to be within the range of P_a. With our grid partitioning approach (see Section 5.1.1), this is not possible due to consecutive intervals (without gaps) assigned exclusively to each partition. ∎

5.2.3 Meta information for communication

So far, we assume the knowledge on which blocks of data stored on the communication stacks are associated to which partition to be already given. We refer to this knowledge as *communication meta information.*

We can store this information per partition and not per cell due to two properties given by the SFC stack-based communication:

1. The adjacency information per cell is not required for inner-partition hyperfaces due to stack-based communication.
2. We can store and manage data on shared interfaces $\mathcal{I}_{i,j}$ efficiently with a run-length encoding per partition which we present next:

Run-length encoded adjacency information:

We continue with a description of the meta information for communication and how it is managed. Searching for adjacency data for each communication element can be time-consuming if iterating over all adjacency information stored for all shared hyperfaces or stored per cell. For node-based communication with each node adjacent to up to 8 cells, managing adjacency information can be also very memory demanding.

We present a solution based on both previously derived theorems:

- Lemma 5.2.1 provides the property of all shared interfaces particularly ordered on the stack system with respect to the SFC-based index of the cell generating the data. Using this property, we can *access the data at the adjacent partition en bloc without considering per-hyperface* meta information. This is due to the same quantity of data also being stored *ordered and en bloc* on the communication buffer of the other partition. Using the order of both replicated chunks of communication data, the corresponding replicated data placement for each hyperface can be induced.

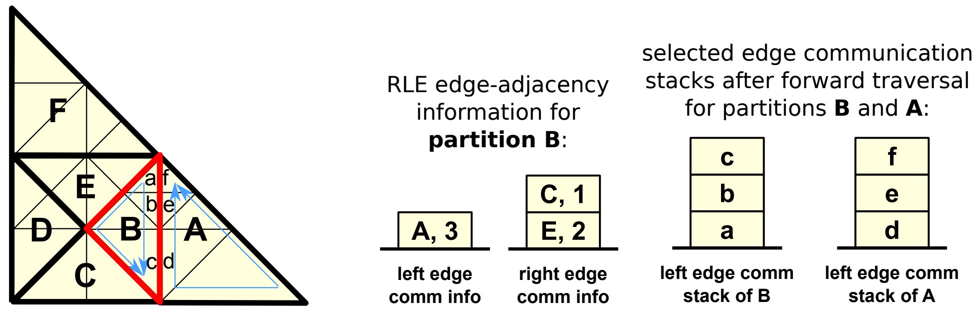


Figure 5.4: Example for an RLE edge communication meta information. The domain is partitioned via subtrees of the spacetree into partitions A-F. Here, we discuss the RLE edge meta communication information for partition B. Regarding the left edge communication stack, three edge communication data elements (a, b, c) are stored to the left edge communication stack after the first forward traversal. These shared edges are adjacent to partition A and encoded with the RLE $(A, 3)$. For the right communication stack and following the SFC induced edge access with the forward traversal in partition B, the first adjacent partition is E via two edges, hence using the RLE entry $(E, 2)$. This is followed by partition C with only one edge which is encoded with RLE entry $(C, 1)$.

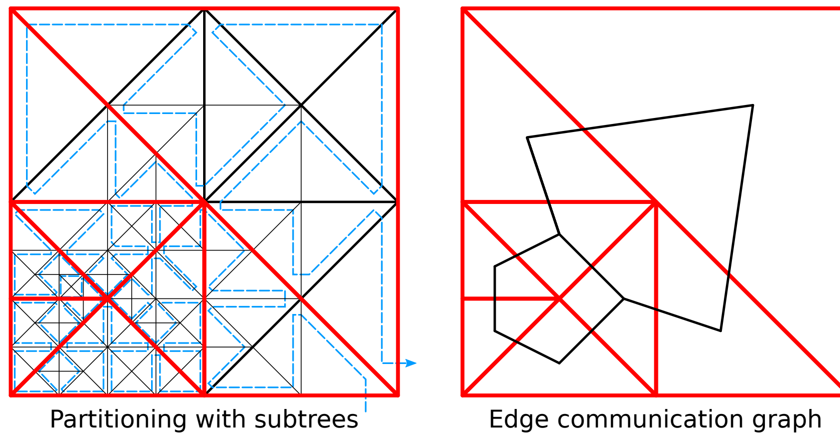


Figure 5.5: An example of a sparse communication graph representing our RLE meta information for edges. Each graph edge represents one entry in the RLE meta information.

- Theorem 5.2.2 assures that each consecutive chunk of data is *stored only once* per adjacent partition. For partition P_i, let the consecutive set of shared interfaces $I_{i,k}$ be given preceeded and followed by shared interfaces associated to other partitions. Then, there is no other set of shared interfaces which is associated to partition P_k. For partitions generated by spacetree splits, we can also use the geometric convexity of the subtree to assure the uniqueness of a set of shared interfaces associated to an adjacent partition.

This allows introduction of a *run-length-encoded (RLE)* representation of the adjacency information.

We first consider one-dimensional shared interfaces (edges) only and the information on communication edges for adjacent partitions A, B and C, respectively, given by 2, 4 and 3 shared edges. Without RLE, we can store the meta information with the tuple $(A, A, B, B, B, B, C, C, C)$. Using our RLE, we represent m entries for edge communication referring to the same adjacent partition P with the tuple $R := (P, m)$. We can compresses this with an RLE scheme to

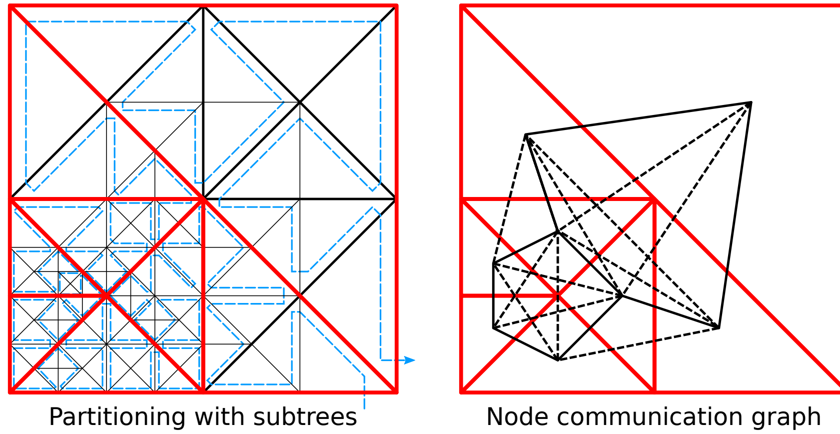


Figure 5.6: Sparse communication graph for edges and nodes. Left image: grid representation. Right image: communication graph for the given underlying grid. Solid line graph edges represent communication for edges whereas a dashed graph represents communication for nodes of partitions not represented by RLE for edge communication meta information.

$((A, 2), (B, 4), (C, 3))$. Figure 5.4 gives an example of such an RLE edge communication for a single partition and Figure 5.5 shows the underlying representation of the sparse edge communication graph.

For the zero-dimensional hyperfaces (nodes), we can extend the RLE representations for the edges directly accounting for nodes: We first require that *no zero-length encoded communication information for edges* may be stored. This allows us to use RLE elements with $m = 0$ to describe a vertex shared with an adjacent partition which was not considered by the edge-based communication so far.

To give an example, we assume adjacent partitions A, B and C, sharing hyperfaces setup with 2 edges, 1 vertex and 3 shared edges, respectively. Storing the information on edges and vertices separately would result in edge encodings (A, A, C, C, C) for the edge communication information and (A, A, A, B, C, C, C, C) for the vertex communication information. Unifying both representations with tuples and additional markers e and v representing edges and vertices yields $((A, e), (A, e), (B, v), (C, e), (C, e), (C, e))$. Here, we assume that two consecutive and identical entries also account for a vertex. Using our RLE scheme, this meta information can be further compressed to $((A, 2), (B, 0), (C, 3))$. The underlying sparse communication graph is given in Fig. 5.6. This RLE meta information is stored separately for the left and right communication buffers.

We summarize the main benetifs of using such an RLE compared to meta information stored for each shared hyperface:

(a) Less memory is required to store adjacency information.

(b) We can use block-wise communication for shared- and distributed-memory communication.

(c) We can implement an efficient implicit management of RLE meta information for our adaptivity traversals (see e.g. Section 5.2.6).

By considering the communication of cell ids, the SFC traversal at the adjacent partition generates the data on the communication stacks in reversed, descending, order. This leads to the requirement of *reversing the order of elements transferred block-wise.*

5.2.4 Vertices uniqueness problem

So far, we ignored the vertices at the cell touched at first and last during traversal of a partition. The data associated to this vertex can be stored to either the left or right communication stacks. We consider two solutions for this uniqueness problem:

- *SFC traversal aware:*
  The algorithm for determining the position of the communication data considers whether the vertices have been stored to the left or right communication stack. This can be done by taking the SFC traversal for the first and last entered cell into account.

- *SFC traversal modification:*
  An alternative approach is to modify the SFC traversal and, thus, forcing the first and last node to be either stored to the left or right stack.

We decided to use the *modification of the SFC traversal* for the present work: with our recursive subtree traversal, we override the underlying SFC traversal grammar $\mathbb{G}$ on the subtree's root node to force the placement of the first and last vertex to the left or right communication stack. Then, we use this uniqueness to derive the knowledge on the placement of the vertices on the communication stacks.

5.2.5 Exchanging communication data and additional stacks

So far, we know the amount of data and from which adjacent partition to read the data from. With our parallelization based on the replicated data scheme (see Sec. 5.1.2), we use separated buffers for the inter-partition shared hyperfaces. However, exchanging data with adjacent partitions using the same stacks for receiving data as for writing data leads to race conditions. Therefore we extend the stack system with buffers storing the exchanged or, in combination with the communication buffer, the reduced data in case of a reduce operation. This additional *exchange stack* is then used by the backward traversal instead of the communication stack during the forward traversal. Such additional exchange stack requirements lead to a *duplication of each communication stack* due to our replicated data scheme.

We further differentiate between shared- and distributed-memory data exchange:

- For *shared-memory systems*, an access to communication data can be directly achieved by *additionally storing a pointer* to the adjacent partition in each RLE information. This allows accessing the adjacent partition directly and looking up the position of the replicated interface data on the adjacent communication stack efficiently with the RLE meta information, see Sec. 5.2.3.

- For *distributed-memory implementations*, we extend the RLE entry with a rank and set this to the rank which owns the adjacent partition. We can then distinguish between partitions stored on the same rank or on another rank by either comparing the rank in the RLE entry with the local rank id or by introducing a special rank, e.g. -1 to represent partitions in the same memory context.

  In case that partitions are existing in the same memory context, the data exchange method is identical to the shared-memory implementation.

  If both partitions are stored on different MPI ranks, we send the local replicated data block-wise by using our RLE meta information. Hwere, we lookup the memory location on the communication stack and the rank information for the destination of the message, followed by a non-blocking send. The receive operation is analogous to the send operation,

but uses the exchange communication stack as the receive buffer. Further information is available in Section 5.10.

5.2.6 Dynamic updating of run-length-encoded adjacency information

The communication to/from cells adjacent via hyperfaces can be accomplished with our communication stack system and the RLE adjacency information. Due to our dynamically adaptive grids, this adjacency information has to be updated appropriately for dynamically changing grids. To avoid a reconstruction of the meta information, e.g. based on the recursive spacetree traversal, we use additional information stored on the communication stack during our last adaptivity traversal.

Instead of running only the last backward adaptivity traversal to refine and coarsen cells, we also transfer additional information on inserted and removed edges via the edge communication stacks. To generate data for the partition boundary dP_i, we set the edge types of the partition boundaries to *new* and forward the following markers via edges:

- Refine marker M_R:
  The marker M_R is pushed to the edge communication stack for edge e_i in case that a refine operation demands inserting an edge creating a vertex at edge e_i.

- Coarsening marker M_C:
  For a coarsen operation, the marker M_C is written to both edges associated to the triangle legs. This accounts for this edge being involved into a coarsening operation. We consider two cases: (a) the edge is shared between the two cells which are joined with the coarsening. Then, this edge is removed and also the forwarded coarsening marker M_C is fetched from the stack system. (b) the edge is not shared between the two cells which are joined with the coarsening. Then, this edge is not joined with the edge of the other cell involved in the coarsening process. Hence, the marker M_C is written twice to the communication stack.

- No operation M_0:
  In case that neither the marker M_R, nor the markers M_C was transferred via the edge, the marker M_0 is pushed to the edge communication system. This accounts for an edge not being modified due to adaptivity.

After the grid traversal, the left and right edge communication stacks then store adaptivity markers on split (M_R: inserting a vertex) and joined ($2 \times M_C$: removing a vertex) edges for the partition boundary dP_i. This allows us to update the RLE meta information of the modified grid based on these markers only. An example is given in Fig. 5.7.

Updating the left and right RLE meta information is then accomplished by iterating over the respective adaptivity communication stack to which the adaptivity markers were written to. These markers describe the change in the RLE meta information due to the adaptivity step.

Algorithm: Updating communication meta information

- For markers M_R, the RLE information associated to the adjacent partition has to be incremented by 1 due to an additional edge inserted.

- The marker M_C is handled in a different way: due to the diamond coarsening shape (Fig. 4.10), only pairs of M_C are allowed. The corresponding RLE is then decremented by 1 *for each pair* of coarsening markers M_C stored on the communication stack.

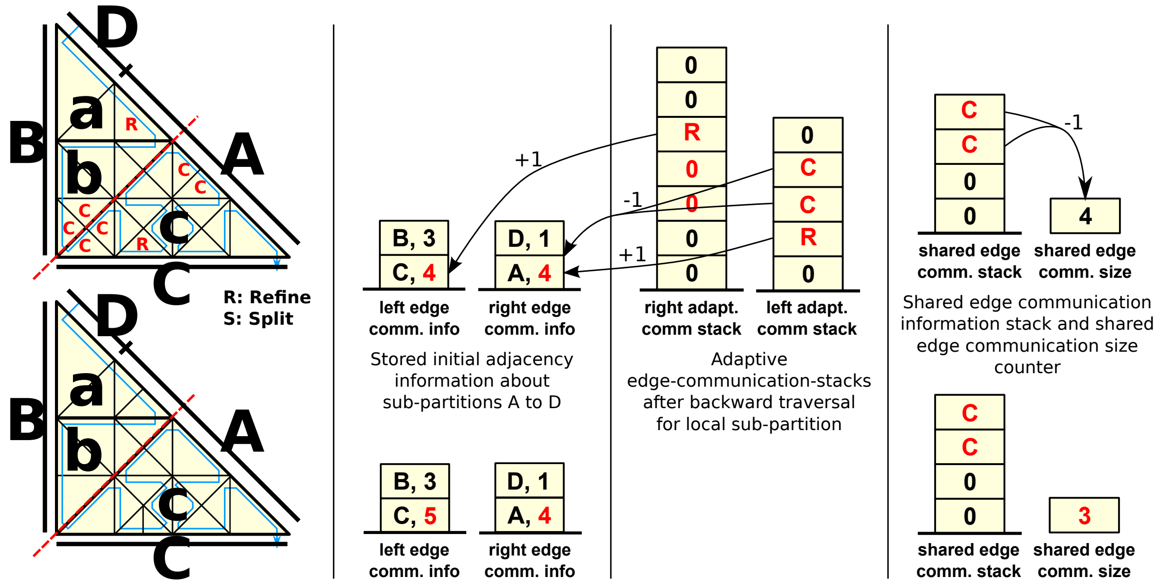


Figure 5.7: Updating meta information for the communication being based on adaptivity refinement ($M_R = R$) and coarsening markers ($M_C = C$). Note that the adaptivity markers are stored in backward traversal direction. For a single refinement marker R, the corresponding RLE entry has to incremented by 1 to account for the inserted edge. Reading two coarsening markers C from the adaptivity communication stack, this is a representative for a removed single edge. Therefore the RLE entry is decremented by 1 [SBB12].

Updating the vertex communication data is also transparent to this adaptivity process. By adding or removing edges via updating the RLE, this also accounts for updating the vertex-based communication information due to three properties:

- Adding an edge by increasing the RLE also considers the additional vertex.
- Removing an edge by decreasing the RLE also considers the removed vertex.
- Explicitly stored vertices with RLE of 0 are transparent to edge insertions since all refinement states do not modify 0-length encoded vertex information.

Special care has to be taken for *partitions consisting only of a single cell* since a coarsening operation would lead to a partition only consisting of half a cell. These coarsening operations must be deactivated by invalidating the coarsening state on the adaptivity state stack.

The presented communication schemes are also applicable to partitions generated by SFC cuts.

5.3 Parallelization with clusters

Based on the previous sections which gave a description on our domain decomposition approach and the communication between partitions, we now describe implementation details of our parallelization approach: how to store and manage partitions. This finally leads to an efficient clustering of the grid as suggested in different contexts such as cluster-based local

time stepping [CKT09] and clustering based on *intervals of one-dimensional representations of SFCs* preserving the locality of multi-dimensional grid traversals [MJFS01]. With our dynamic adaptive grid, we extend these ideas to a *dynamic clustering*.

5.3.1 Cluster definition

We start with our definition of a cluster by its data storage, grid management, communication meta information and traversal meta information properties:

- *Bulk of connected cell data*:
  In general, a bulk of cells connected via hyperfaces is associated uniquely to a cluster. There is a path from each cell to all other cells inside the cluster via shared hyperfaces of cells in the cluster. Using a continuous SFC, the connectivity is directly given by our partitioning approach based on SFC intervals and Theorem 4.3.1.

- *Indepent memory areas*:
  The simulation data and the meta information associated to each cluster is allocated in a way to be race-condition free with other clusters. In this work, we allocate the data for each cluster on a separate heap memory. Among others, this allows shrinking and growing of the number of grid cells in each cluster without forcing reallocations of data areas of other clusters.

- *Communication information*:
  The inter-partition communication should be managed efficiently for shared- and distributed-memory parallelization. In our case, we store such communication meta information to adjacent clusters using RLE. For the intra-partition communication, we use the stack-based communication approach. Communicating via stacks makes our communication invariant to the memory or rank location of the cluster, which becomes an important property for data migration.

- *Traversal meta information and user-specified data (optional)*:
  Each cluster also has to store traversal meta data, e.g. initial vertex coordinates to start the grid traversal, minimum and maximum adaptive refinement depth limiters and also possibly required user-specified data. The traversal meta data is required to know where to start traversals in the spacetree. The user-specified data can consist of e.g. parameters for kernels and the face identifier of the cubed sphere, see Sec. 6.5.

For cluster-based local time stepping [CKT09], a decomposition of the domain in non-overlapping bulks with communication schemes capable of independent time steps was suggested, see Fig. 5.8. Since our development yields the same possibilities with a domain decomposition based on the heuristic of the Sierpiński SFC, we continue to use the terminology *cluster*, referring to a partitioning and software design fulfilling the previously mentioned properties.

5.3.2 Cluster-based framework design

With clustering at hand, we introduce a parallel framework design with a top-down approach by assembling domains with a set of clusters: First, we encircle the simulation data, the grid traversals as well as the corresponding kernels from the serial framework design (see Fig. 4.12) into a cluster container. A sketch of the resulting structures and abstractions inside such a container are depicted in Fig. 5.9.

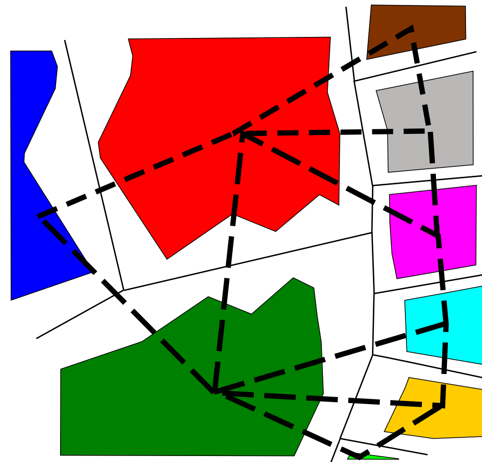


Figure 5.8: Examples of alternative clustering strategy: Shapes of clusters used for cluster-based local time-stepping method [CKT09]. The connectivity information is given via the edges shared by adjacent clusters. Such clusters are not generated by the approach presented in this thesis.

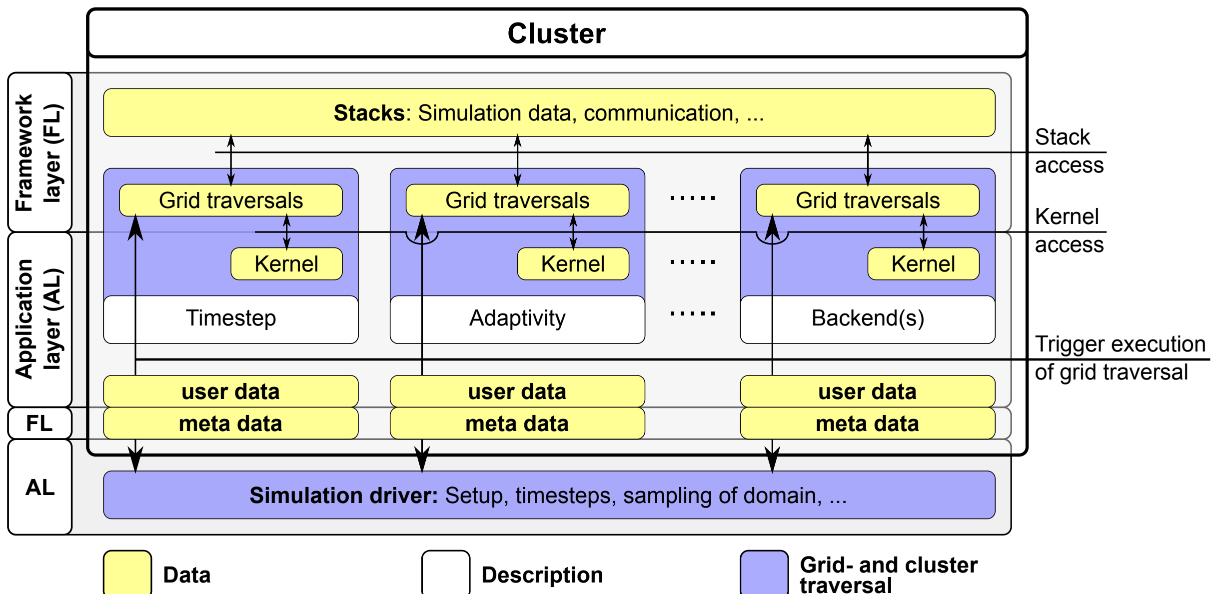


Figure 5.9: Cluster building block based on serial framework design.

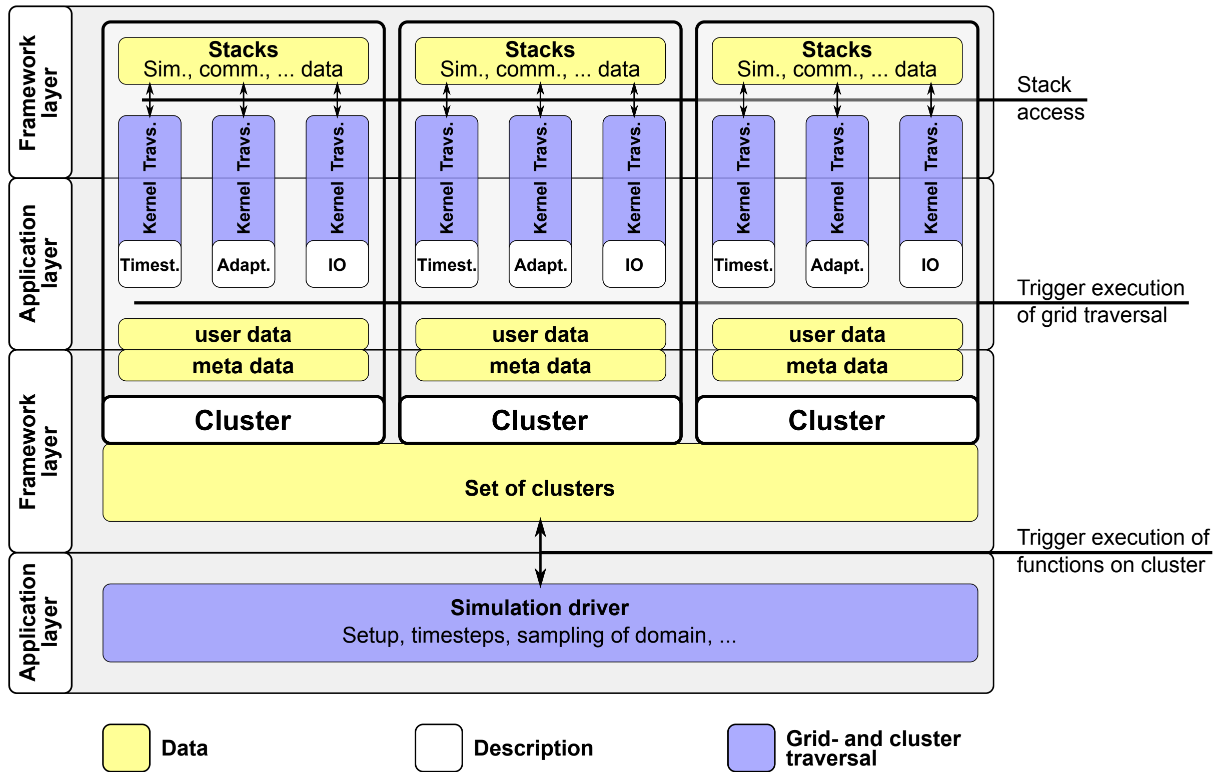


Figure 5.10: Overview of the parallel cluster-based framework design. See the text for a detailed description.

For multiple clusters, we then extend this serial cluster-based framework design as it is depicted in Fig. 5.10. We continue with a top-down description under consideration of the serial framework design from Fig. 4.12:

- Top *framework layer*: each cluster has associated its own grid data which is stored on stacks.
- On the *application layer* below, kernels can be executed in parallel for each cluster without influencing each other due to replicated data scheme. The user data can be used to store cluster-specific information, e.g. the face id for the cubed-sphere domain triangulation or cluster-specific boundary conditions.
- The meta data is required for communication and information on grid traversals and hence belongs to the *framework layer*. All clusters are kept in an efficient cluster management structure, the set of clusters, which is discussed in Section 5.3.3.
- The simulation driver then executes operations specified via C++ lambda functions. These lambda functions are executed on all clusters with the cluster as the parameter. Such an operation in a lambda function can be e.g. a forward traversal or setting the cluster parameters.

5.3.3 Cluster set

The dynamical creation and deletion of clusters demands for an efficient cluster management data structure. Such an efficient management can be e.g. achieved with (double-)linked lists,

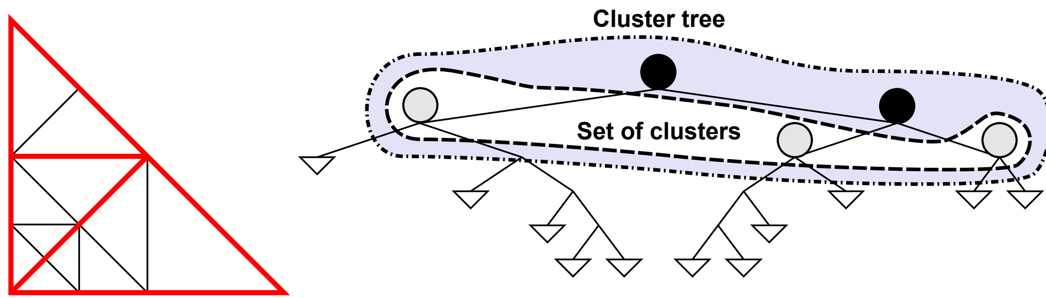


Figure 5.11: Set of clusters based on tree-splits of the spacetree.

vectors and maps. All these containers represent a set which stores the clusters. With our dynamic cluster generation based on subtree, we keep the recursive structure and use a binary tree structure, the *cluster tree*, to insert and remove clusters.

In case of a cluster *split*, two nodes are attached to the formerly leaf node and the cluster data is initialized at both leaf nodes.

For clusters stored at two leaves sharing the same parent, a *join* operation creates new cluster data at the parent node. Here, we can also *reuse the cluster storage of one children* to avoid copy operations. After *joining* two clusters, both leaf nodes are removed.

See Section 5.5 for a detailed description of dynamic clustering. An example of a cluster set based on a tree is given in Fig. 5.11.

Regarding the *base triangulation* which allows us to assemble domains with triangles being their building blocks (see Section 5.4), we follow the idea from p4est [BWG11] and combine multiple cluster trees at the leaf of a super-cluster tree. This extends the cluster tree to a forest of trees, a *cluster forest*, embedded into the super-cluster tree. We avoid joining clusters which were not created by a cluster split, i.e. initially belonging to the base triangulation. Here, we limit the cluster-join operations by considering the depth of the cluster in the super-cluster tree. For sake of convenience, we continue referring to the super-cluster tree as the cluster tree.

5.3.4 Cluster unique ids

For identifying each cluster uniquely, we generate unique ids directly associated to the cluster's placement in the cluster tree. This is based on the parent's unique id for a split operation and childrens' unique ids for join operation.

The cluster tree root id is initially set to 1_b with the subscript b denoting binary number notation. With the id of the parent node stored in *parentId*, the first child's id traversed by the SFC and the second child's id following the first child is given by

$$firstChildId := 2 \cdot parentId \quad \text{and} \quad secondChildId := 2 \cdot parentId + 1.$$

Using this unique id inference results in a cluster forest's root node id of 1_b - otherwise the same id would be assigned to the first child. Based on one of the child ids, the parent's unique id can be inferred by

$$parentId := \left\lfloor \frac{ChildId}{2} \right\rfloor.$$

This recursive unique id generation also provides information on the placement of the cluster within the tree. This feature is used for cluster-based data migration in Section 5.10.3 to update adjacency information about cluster stored on the same MPI node.

Furthermore these unique ids inherently yield an *order of the cluster along the SFC*. Given id_1 and id_2, we can compute the order with the following algorithm: for each unique id, the depth of the cluster in the cluster tree is given with

$$clusterDepth_i := bsr(id_i)$$

(bit scan reversed), returning the position of the most significant set bit in id_i. E.g. $bsr(001001_2)$ would yield 3. With the maximum depth of both clusters given by

$$maxClusterDepth := max(clusterDepth_1, clusterDepth_2),$$

we shift both unique ids to be on the same cluster tree level using

$$sid_i := id_i << (maxClusterDepth - clusterDepth_i)$$

and finally get the order by direct comparison of both sid_i using less-than relations on sid_i represented with integer numbers. We can use this order to avoid duplicated reduce operations on replicated data shared by two clusters (see Section 5.8.1).

Alternative approaches would be e.g. based on a mix of MPI ranks and the MPI-node-local cluster enumeration. This also leads to properties which can be similarly used in the next sections. However, we decided to use the approach described above, since this unique id gets beneficial if searching for a cluster in the cluster tree.

5.4 Base domain triangulation and initialization of meta information

Simulations frequently demand being executed on domains with a different shape than a triangle or quadrilateral, e.g. a rectangular-shaped domain. The grid generation based on the Sierpiński SFC with the communication via stacks was so far accomplished only for a continuous SFC (see e.g. [Vig12]) traversal of the grid. This leads to limitations of the shape of the simulation domain. Solutions which are also based on the linearized form of the leave nodes of a forrest of spacetrees (see [NCT09], this work is based on the Hilbert curve) introduced discontinuous SFC traverals. Such discontinuities in the SFC traversal allow spatial jumps on the simulation grid leading to an increased flexibility in domain configurations.

Since spacetree traversal is based on recursion, this makes such discontinuities challenging. Hence, we use an alternative domain assemblation: with our clustering based on subtrees, this allows domains of flexible shape being assembled by triangle primitives. This assemblation is considered to be valid, regarding our stack-based communication scheme, as long as all edges of the triangles are either not shared with other triangles or shared with exactly one triangle, see Figure 5.12 for examples. This requires setting up correct meta information in each cluster. Such information is e.g. run-length encoded communication information to/from adjacent clusters for communication with adjacent triangles and is further discussed in the next Section.

5.4.1 Initial communication meta information

We initialize the inter-cluster communication based on clusters initially representing an entire spacetree. Each spacetree can also be based on a single leaf node. We refer to the spatial representation of these clusters as *base triangles*.

With such base triangles, the number of edge-communication elements on the left and right communication stack is given with

$$|\text{edges on each cathetus}| = 2^{\lfloor \frac{d}{2} \rfloor}$$

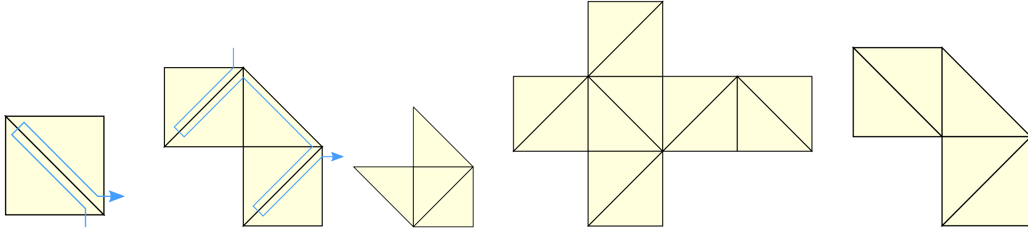


Figure 5.12: First two domains: examples for domain base triangulations with a traversal on a continuous Sierpiński SFC. Next three domains: Examples for domain base triangulations which cannot be represented by continuous 2D Sierpiński SFC traversal. The fourth domain is of particular interest for the cubed sphere grid, see Sec. 6.5.

$$|\text{edges on hypotenuse}| = 2^{\lfloor \frac{d+1}{2} \rfloor}$$

with d the initial refinement depth.

For each base triangle, we determine the base triangles which are adjacent via all three triangle edges by searching for all possible adjacent base triangles via testing for shared edges vertices. This $O(n)$ complexity for n clusters to search for base triangles sharing an edge can be assumed to be negligible for a small number of base triangles setting up the simulation domain.

The reconstruction of the vertex-based meta information requires further processing and is described in Sec. 5.5.4.

5.5 Dynamic cluster generation

For our clusters based on tree splits, we use tree-split and -join operations for the cluster generation.

5.5.1 Splitting

To split a cluster, we require information on (a) the number of shared hyperfaces along the separating hyperfaces and (b) the association of persistent data to both child clusters.

We achieve inferring this information by *stopping our grid traversal at particular positions* and *implicitly derive the required quantities on the number of hyperfaces and cells* based on the communication stacks (meta communication information) and persistent data stack (simulation data). We refer to this information as *split information*.

For optimization reasons, we can further join the determination of split information with the last backward adaptivity traversal and present the algorithm based on this backward traversal which is in reversed direction, see Fig. 5.13. The algorithm is based on the spacetree nodes on the 2$^{\text{nd}}$ level relative to the cluster root tree node and follows the SFC in reversed direction with (a, b, c, d). Here, we consider partitions (A, B, C) respectively set up by leaf nodes of subtrees $(a, b, c \cup d)$.

The application of our algorithm is given in Fig. 5.13 with the left and right communication stacks terminology used for the last adaptivity (backward) traversal and for grammars of type even. We denote the stacks for the left and right adaptivity communication stack with $\mathbf{S}_{left}$ and $\mathbf{S}_{right}$, respectively. The cell stack is given by $\mathbf{C}$, and we determine the number of cells of the parent element by considering the elements stored on the cell stack: $c^{(parent)} := |\mathbf{C}|$.

The information on the split operation is stored in $s^{(1)}_{\{left,right\}}$ for the first and in $s^{(2)}_{\{left,right\}}$ for the second child. We also require the number of edges shared by both split clusters in

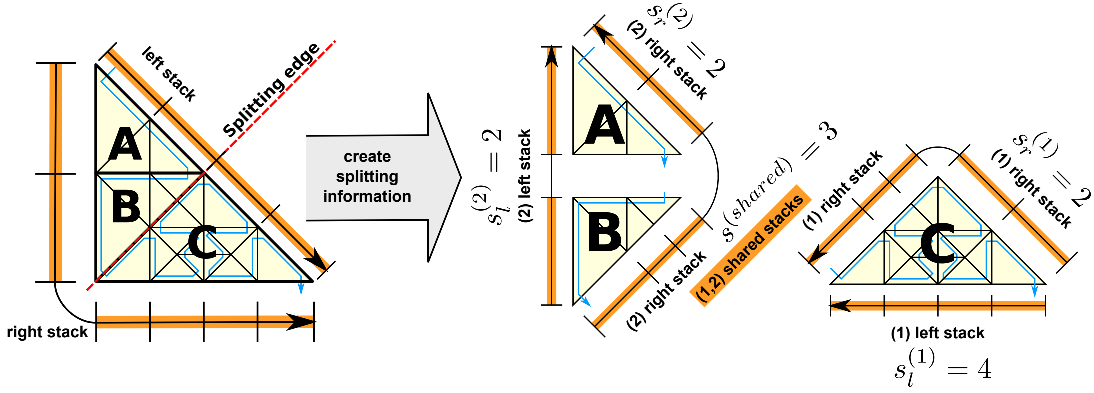


Figure 5.13: Backward traversal generating split information. The stacks generation direction are given in forward traversal direction while the traversal itself is done in reversed direction during the last backward adaptivity traversal [SBB12].

$s^{(shared)}$. The number of cells associated to the first and second child partition is stored to $c^{(1)}$ and $c^{(2)}$.

1. After traversal of cells in subcluster A, the communication edges on the left stack for the 2nd subcluster are memorized with

$$s_{right}^{(2)} := |\mathbf{S}_{left}|$$

   We like to emphasize again, that the left and right labels are reversed due to the information derived in backward traversal whereas RLE meta information is stored in forward-traversal direction.

2. After traversal of cells in B, the shared edges are computed by

$$s^{(shared)} := |\mathbf{S}_{left}| - s_{right}^{(2)}$$

   and the number of parent's inter-cluster edges for the 2nd subcluster on the right stack are saved in

$$s_{left}^{(2)} := |\mathbf{S}_{right}|.$$

   The processed number of cells which is then associated to the 2nd cluster created by the split operation is inferred by

$$c^{(2)} := c^{(parent)} - |\mathbf{C}|$$

   with $|\mathbf{C}|$ the remaining number of cells to be processed. This also represents the number of cells assigned to the first subpartition C:

$$c^{(1)} := |\mathbf{C}|.$$

3. Due to the adaptivity traversal, the information on the shared edges $s^{(shared)}$ only represents the quantity of shared hyperfaces before refining or coarsening the grid based on the adaptivity states. Therefore, the amount of hyperfaces $s^{(shared)}$ has to be updated based on the refinement and coarsening markers M_R and M_C stored on the adaptivity marker communication stack, see Section 5.2.6.

4. After traversal of cells in C, the information to reconstruct the communication information is then given with

$$s^{(1)}_{right} := |\mathbf{S}_{left}| - s^{(2)}_{right},$$

and

$$s^{(1)}_{left} := |\mathbf{S}_{right}| - s^{(2)}_{left}.$$

Clusters with leaf elements stored on level 1 or 2 require to be handled appropriately in case that subclusters A and B do not exist. E.g. with a domain only consisting of two cells, we can directly set $s^{(2)}_{right} := 1$, $s^{(2)}_{left} := 1$, $s^{(shared)} := 1$.

With the derived information, we can apply the split operation: The *cell data* on the stack can be directly split based on the split information $c^{(\{1,2\})}$. The *traversal meta information* (providing e.g. the starting point of the grid traversal) has to be updated to account for the new traversal start. The *communication meta information* of both children can also be determined, based on the information on the number of shared edges of both children given in $s^{(\{1,2\})}_{\{left,right\}}$, $s^{(shared)}$ and $c^{(\{1,2\})}$. For the odd grid grammar, the inference of the required information is similar and therefore not further discussed here.

The presented algorithm only accounts for updating the local cluster information and would lead to inconsistencies to the meta information stored at adjacent cluster. Updating this meta information on adjacent cluster is presented in Section 5.5.3.

The simulation data on the stacks is currently split after the traversal of the entire cluster. However, copying chunks of stack data from the other clusters can result in stream-like copy operations and result in a bandwidth-limited problem. A direct streaming of the persistent simulation data during the last adaptivity traversal to the stacks of the corresponding new child clusters has the potential to avoid this additionally required stream-copy operation but is not implemented, yet.

5.5.2 Joining

Using clustering based on tree splits, our cluster-based approach is restricted to joining two clusters only if they share the same parent node in the cluster tree. Joining two clusters is accomplished by

(a) concatenating the cell data storage,

(b) joining both traversal meta information and

(c) joining both communication meta information and removing the RLE information about the edges shared by both child cluster.

5.5.3 Split and join updates of meta communication information

So far, we can split and join clusters and update the meta information based on adaptivity markers written to the edge communication stacks. However, we also have to consider possible split and joins of adjacent clusters, e.g. if an RLE element has to be split to account for the associated cluster being split. This requires updating the edge communication meta information. We present an algorithm by modifying this edge communication information to synchronize with the adjacent meta communication information in case of splits or joins. This algorithm is based on *local cluster and directly adjacent cluster access only and does not involve any global communication operations.*

Transfer State	Description
NO_TRANSFER	No split or join was executed on this cluster.
SPLIT_PARENT	This node was split and both children have the state SPLIT_CHILD.
SPLIT_CHILD	Cluster generated by a split operation are set to this state.
JOINED_PARENT	This node was created by joining both children which have state JOINED_CHILD.
JOINED_CHILD	This cluster represents a former child node which was joined with its sibling. This cluster is not deleted to allow accessing its communication meta information.

Table 5.1: Different state flags assigned to each node in the cluster tree.

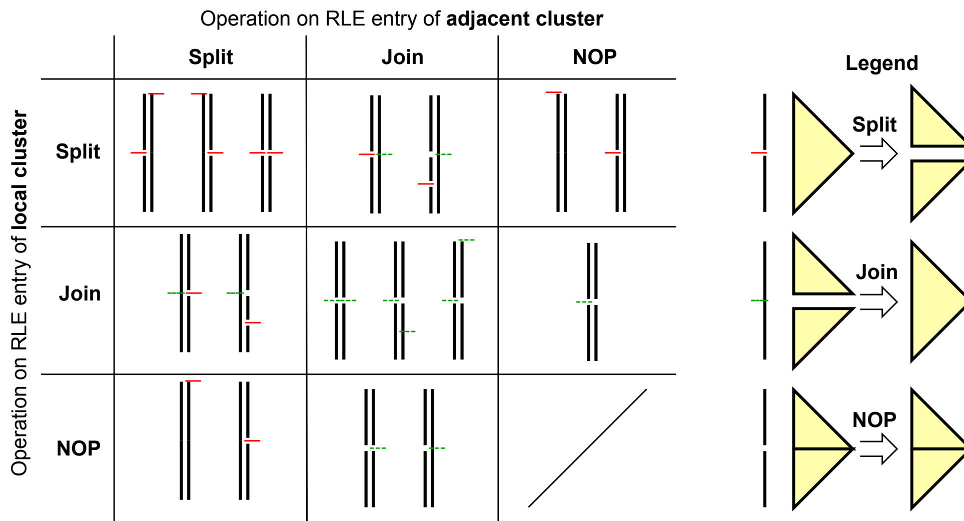


Figure 5.14: Possible split and join constellations for adjacent clusters. Each thick black line represents one RLE meta information after an update. The left thick line corresponds to the *local meta information* and the right one to the *adjacent meta information*. Different split and join constellations are possible: Split operations are marked with a red line, join operations with a green dashed line. See Fig. 5.15 for a concrete example.

We continue refering to the cluster for which the RLE communication meta information is updated as the *local cluster* and a cluster described by a single RLE communication data stored for a local cluster as *an adjacent cluster*.

Updating edge-based meta communication is based on flags stored per cluster which describe the split- and join-states of the cluster. All required states for markers of adjacent clusters are given in Table 5.1 and an overview of all relevant cases which have to be considered in Fig. 5.14.

Shared memory

Similar to the exchange of edge communication data, updating the communication information is accomplished by read-only access of adjacent cluster data for shared memory access. Examples for the split-split constellations are given in Fig. 5.15.

The algorithm then consists of the following steps for each of the left and right RLE meta information lists:

Algorithm: Outer loop of RLE consistency

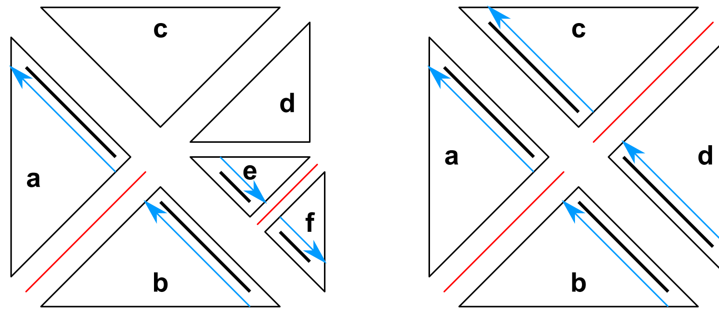


Figure 5.15: Two different split-split cluster states with the SFC traversal drawn closely to the shared interface either in opposite or identical direction. The blue arrow shows the traversal direction of the SFC curve close to the considered edge.

1. An iterator is set to the first RLE entry in the meta information list.
2. For each iteration on the next RLE entry, we test for *adjacent cluster changes*.
3. In case that neither a split nor a join operation was detected in the adjacent cluster given by the RLE entry, we advance the iterator and continue with (3).
4. The RLE entry requires additional processing due to possible inconsistencies created by split/join operations of the adjacent cluster. Since we are updating the communication information and modifying data in parallel, we have to avoid race conditions. These race conditions can occur since a processing of other clusters in parallel can lead to accessing the meta information of this cluster.

   If this is the first time of a modification of one entry in the local RLE list during the list iteration, we duplicate the list of local RLE communication meta information and continue iterating on the duplicate. The iterator is set to the same entry in the duplicated data and further operations are only executed on the duplicated data.
5. We continue by distinguishing different split or join cases. First, a check of the *adjacent transfer state* is done which can be NO_TRANSFER, SPLIT_PARENT or JOINED_CHILD, respectively, abbreviated with $(NOP, SPLIT, JOIN)$. The second decision is the type of the *local transfer state* which can be NO_TRANSFER, SPLIT_CHILD or JOINED_PARENT, also respectively abbreviated with $(NOP, SPLIT, JOIN)$. An overview of possible cases is given in Fig. 5.14.
6. For split transfer states of the adjacent partition, an additional counter *remainingCommunicationElements* is introduced and set to the number of edges of the currently processed local RLE entry.
7. Update the local RLE entry with one of the algorithmic blocks below, advance the iterator and continue at 3.

We distinguish the method for updating the RLE meta information depending on the *transfer state* of the adjacent cluster to decide in which access order to traverse the RLE-related adjacent

clusters:

Algorithm: Local RLE entry update - **adjacent transfer state: SPLIT_PARENT**

- **Local transfer state: SPLIT_CHILD**:
  For this transfer state constellation, we further distinguish between the following cases:

  1. The split operations for both adjacent clusters lead to RLE entries with equal cardinality, see e.g. the right image in Fig. 5.15. Then only the information on the new placement (pointer/rank) of the adjacent cluster (this information changed due to the parent's split) has to be updated.
  2. The RLE meta information is on a split of both adjacent clusters with the quantities not matching, see e.g. left image in Fig. 5.15.

  In any case, we have to figure out the traversal order of the corresponding adjacent child cluster nodes.

  Whether the *first or second adjacent child node is traversed first*, plays an important role to append additionally required RLE information in the correct order. This information is given by D with 0 for a traversal in the order "first, then second child" and 1 for "second, then first child". We initialize our flag with $D := 0$. The traversal direction is then updated based on the local and adjacent SFC traversal direction on the shared cluster boundary:

  – *Rule 1) Traversal order of adjacently split children:* If the cluster boundary traversal directions of the local and the adjacent parent's cluster are equal, the traversal order of the adjacent children has to be in the same direction by setting $D := 0$, otherwise reversed by setting $D := 1$.

  – *Rule 2) Traversal order of locally split children:* The traversal order is reversed if the RLE of the second local child is updated. This leads to a search of matching RLE entries from the end of the adjacent parent's meta information.

  An example for applying these rules is given below. Once the order of adjacent child access directions is known, the update process can start:

  1. *remainingCommunicationElements* is initialized with the number of local RLE elements for which the update is done.
  2. An adjacent child is accessed according to the adjacent traversal direction D, and the edge communication elements are updated in case of matching RLE elements including decreasing the variable *remainingCommunicationElements* with the number of edge communication elements in the adjacent cluster.
  3. If the variable *remainingCommunicationElements* is less or equal to zero, no further updates are required.
  4. Otherwise, the next child is searched for a matching RLE entry with meta information of exactly *remainingCommunicationElements* elements.

- **Local transfer state: JOINED_PARENT**
  If the local cluster is a parent cluster which was created by joining two child clusters, the first adjacent cluster is searched for corresponding RLE elements. In case of a

perfect match of the RLE number of shared edges, no update is required. Otherwise, a new RLE entry with the changed quantity of shared edges has to be inserted and the search is continued on the next adjacent child cluster for the remaining matching RLE element.

- **Local transfer state: NO_TRANSFER**
  In case that the local cluster was not modified, rules similar to the SPLIT_CHILD case can be applied. Because of this similarity, these are not further discussed here.

Given the rules introduced in the previous algorithm, we discuss the relevant scenarios given in left image in Fig. 5.15 for cluster b by traversing the adjacent clusters.

Left image:

- Rule 1) The traversal on the cluster boundary of the shared edges of cluster b is counter-clockwise and also the direction of e and f. This leads to a reversed child traversal order ($D := 1$).
- Rule 2) Since cluster b is the first child's triangle, the adjacent traversal order is not changed.
- With $D == 1$, this leads to a reversal of the traversal of the adjacent child clusters, a second-first order. This means, that first cluster f is searched for adjacent edge parts, followed by cluster e. Each traversal leads to an insertion of corresponding new RLE entries to the RLE meta information. These entries replace the RLE entry associated to the split parent cluster.

Right image:

- Rule 1) The SFC traversal direction on the cluster boundary of shared edge of cluster b is counter-clockwise and thus not equal to the cluster boundary direction of c which is clockwise, leading to a non-reversed adjacent cluster tree traversal ($D := 0$).
- Rule 2) Since the triangle b is the first child triangle, the adjacent child traversal order is not changed ($D := 0$).
- Combining both rules, the child traversal of the adjacent node is done in first-second order. Thus cluster d is searched for adjacent edge parts first, followed by cluster c.

Algorithm: Local RLE entry update - **adjacent transfer state: JOINED_CHILD**

In case of a joined adjacent child, the local RLE is tested for the possibility to join it with the next local RLE. Such a join is producing a consistent state in case that both consecutively stored adjacently joined children have the same parent. Otherwise, this would result in a violation of our assumption of a unique RLE existing for shared interfaces, see Sec. 5.2.3. Other updates of the RLE meta information can be applied similarly to the split case above.

Algorithm: Local RLE entry update - **adjacent transfer state: NO_TRANSFER**

With all local RLE entries being updated directly in case of split and join operations, no modifications of the RLE entry are required in case of an adjacent cluster without split/join.

Algorithm: Postprocessing RLE updates

Finally, the possibly RLE meta information which was duplicated to avoiding race conditions (see step 4 in the 'outer loop' part of the algorithm above) is used as the new RLE meta information.

This algorithm computes a consistent meta information fulfilling all theorems described so far in this thesis.

Distributed memory

We like to give a short outlook to distributed-memory parallelization: Here, we use two-sided communication and the read-only push/pull principle: We separate the updates of the meta information into local and adjacent operations. Instead of looking up the required split/join information by accessing the adjacent clusters stored at another MPI node, our updates of the meta information for each local RLE entry rely on a cooperative communication:

All local clusters send the necessary split/join information about their local changes to the adjacent clusters (push). The adjacent clusters can then receive (pull) this data and update the meta communication information similar to the shared memory synchronization. This leads to a straightforward extension of the shared-memory algorithm presented above.

5.5.4 Reconstruction of vertex communication meta information

We have not considered the vertex communication meta information for dynamic clustering, yet. After cluster splits and joins, the vertex communication meta information can be in an inconsistent state since zero-length encoded vertex RLE entries can be missing. However, we can reconstruct the vertex communication information based on a valid edge communication from the previous section.

The algorithmic idea is based on the assumption that all clusters sharing the vertex also store one or two edge-related RLE meta information about edge-adjacent clusters (or a boundary) which also share the vertex. By accessing the edge-adjacent clusters, we can continue our search to the other vertex-sharing clusters. Then all clusters sharing a vertex can be traversed via following the RLE edge communication information associated to this vertex. This allows generation of a trace of all clusters sharing the vertex and, based on this trace, a reconstruction of our zero-length encoded vertex RLE meta information.

Detailed algorithm

Without loss of generality, we only show the algorithm with the assumption that both SFC traversal directions along the hyperfaces shared by two clusters are *in opposite directions.*

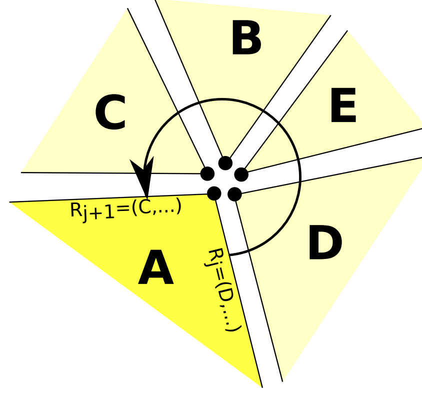


Figure 5.16: Sketch of reconstruction of vertex communication meta information. The meta information is reconstructed for the vertex shared by edge communication elements R_j and R_{j+1} of cluster A.

For each cluster P_i, let the left and right RLE *edge communication meta information* be given by $R_k^{(\{left,right\})} = (Q, m)_k$ with Q the adjacent cluster, $m > 0$ the number of edges shared with the adjacent partition and $1 < k < N_{(\{left,right\})}$ selecting the RLE entry. The cluster associated to the first entry in the tuple $R_j = (Q, m)$ is given by $P(R_j) := R_j[1] == Q$.

We use a flag $F \in \{front, back\}$ describing whether the vertex for which the meta information is determined is associated to the front or back of the currently processed edge-related RLE entry. For the sake of convenience, we *formally*[2] combine $R^{(\{left,right\})}$ to a single ring-like buffer of RLE meta information

$$R := \left(R^{(right)}_{i=1,2,\dots,N_{(right)}}, R^{(left)}_{i=N_{(left)},N_{(left)}-1,\dots,1} \right)$$

with N_{left} and N_{right} the number of tuples in the left and right RLE meta information list, respectively. We further define a periodic padding with $R_0 := R_{|R|}$ and $R_{|R|+1} := R_1$. For *domain boundaries*, we further introduce a special RLE tuple $(-1, -1)$ to account for the boundary edges.

Algorithm: Reconstruction of vertex meta information for cluster P_i

For all $R_j \in R$ of the cluster P_i, execute the following operations:

- *Initialize vertex tracing:* We use flag F which is initialized to *back* since the vertex is associated to the last edge processed by the RLE edge meta information. The previously visited cluster is kept in $P_{(prev)}$ and initialized to P_i as the cluster to reconstruct the vertex information for. The next processed cluster then becomes $P_{(current)} := P(R_j)$.

  In case that $P_{(current)}$ represents a boundary, we set the flag F to $front$ and continue the search with $P_{(current)} := P(R_{j+1})$. If this is also a boundary, the vertex is not shared by any other cluster and we stop the algorithm.

- *Trace generation:*

  1. For the currently processed cluster $P_{(current)}$, the communication meta information $R_{(conn)}$ connecting the currently visited cluster to the previous one is determined by $conn := \{i | P(R_i) = P_{(prev)}\}$

[2] The implementation directly iterates over the left and right meta information.

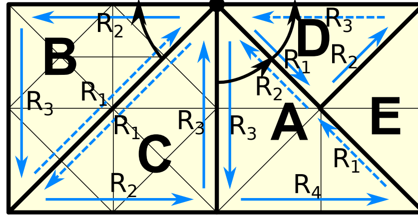


Figure 5.17: Generation of vertex communication data based on RLE edge communication for cluster C. Edge-based RLE information is given by blue arrows and the left and right information combined to R (see text for further description).

2. Using flag F, we load the next ($F = front$: $R_{(next)} := R_{(conn+1)}$) or previous ($F = back$: $R_{(next)} := R_{(conn-1)}$) RLE communication meta information to $R_{(next)}$. The next cluster to visit is then given by $P(R_{(next)})$.
   (a) If the adjacent cluster is of type boundary ($P(R_{(next)}) == -1$), we consider two cases:
   (I) In case that this is the second time that we run into a boundary, we visited all adjacent clusters which are connected via edges and the algorithm terminates.
   (II) Otherwise, we rerun our trace generation (1) on the start cluster $P_{(prev)} := P_i$ but traversing in the other direction $P_{(current)} := P(R_{j+1})$. The flag F is then set to $front$.
   (b) In case of visiting the original cluster $P_{(current)} == P_i$, we found all clusters connected to the vertex and finish the trace generation.
   (c) The next cluster is visited by the following steps: First, we set the previously visited cluster to the current one $P_{(prev)} := P_{(current)}$ and set the current cluster to the next cluster to visit $P_{(current)} := P(R_{j+1})$. Then we continue with our trace generation (1).

Based on the visited clusters, we can reconstruct the required RLE vertex meta information. For each RLE, at most 8 adjacent clusters which share the vertex are visited. Therefore, this algorithm has a *constant runtime* for each RLE. Furthermore, the algorithm is based on *local* (only vertex-adjacent) access operations only. Giving the average number of RLE elements per cluster in $avgRLE$, the runtime complexity is $O(\#Cluster \cdot avgRLE)$.

Example

We give a concrete example of this algorithm for reconstructing the RLE information for cluster C and the middle vertex at the top in Fig. 5.17. Here, we start with the RLE entry $R_3 := (A, .)$ and initialize the algorithm to $P_{(prev)} := C$, $P_{(current)} := A$ and $F := back$.

- Visiting cluster A, we first determine the RLE associated to the previous cluster to be $R_3 = (C, .)$. Since the F is set to *back*, we load the previous RLE R_2, yielding $R_{(next)} := R_2 = (D, .)$.
- We update the previous and current cluster to process to $P_{(prev)} := P_{(current)} = A$ and $P_{(current)} := P(R_{(next)}) = D$

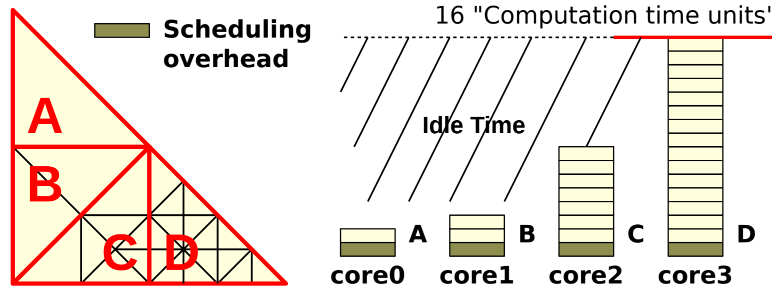


Figure 5.18: Parallelization by splitting cluster in a quantity equal to the number of processors. This leads to a high idle time for adaptive grids [SBB12].

- Visiting cluster D, we determine the adjacent RLE to be $R_1 = (A, .)$. Since F is set to *back*, we load the previous RLE $R_0 = R_3$, yielding $R_{(next)} := R_3 = (-1, -1)$.

  Due to the detected boundary, we set $F := front$, $P_{(prev)} := C$ and $P_{(current)} := R_{j+1} = B$.

- Processing cluster B, the RLE entry associated to the cluster C is given by $R_1 = (C, .)$. F is set to $front$, thus we load the next RLE R_2, yielding $R_{(next)} := R_2 = (-1, -1)$. Since this is the second time that we run into a boundary, the algorithm terminates.

We also like to mention an alternative way which was not implemented in this work: The information on the adjacently generated vertices can also be inferred similar to the reconstruction of edge meta information, see Sec. 5.5.3. With this method, we can also search on the adjacent split/joined cluster meta information for a possible requirement of inserting or removing vertex RLE meta information.

5.6 Shared-memory parallelization

We still have to choose when to split and join clusters as well as which computation units shall execute operations on which clusters.

We start with annotations used for cluster generation and scheduling decisions. Considering a parallelization by creating as many clusters as there are compute units available, this results in a 1:1 scheduling with an example given in Section 5.1.3. With our dynamic clustering based on tree splits, this would clearly lead to high idle times due to severe load imbalances, e.g. created by refined grid cells in a single cluster, see Fig. 5.18.

In this work, we developed two different split approaches: massive-splitting and load-balancing oriented splits. Both methods are based on annotating each cluster i with the workload W_i, e.g. the number of cells in each cluster. Then, the entire simulation workload is given by $\mathbf{W} := \sum_i W_i$. For the load-balanced-oriented splits, we further label each cluster with a range-related information $R_i := R_{i-1} + W_{i-1}$ and set $R_1 := 0$. These labels are computed with a parallel traversal of the cluster tree (see Section 5.3.3) with subindices lhs, rhs and $parent$ denoting the left or right child of the parent tree node, respectively:

- We run through the cluster tree bottom-up and annotate each inner cluster tree node with $W := W_{lhs} + W_{rhs}$.

- During a second top-down traversal, we start at the root node and annotate it with $R_{root} := 0$. During the top-down traversal, we then annotate each left child node with $R_{lhs} := R_{parent}$ and each right child node with $R_{rhs} := R_{parent} + W_{lhs}$.

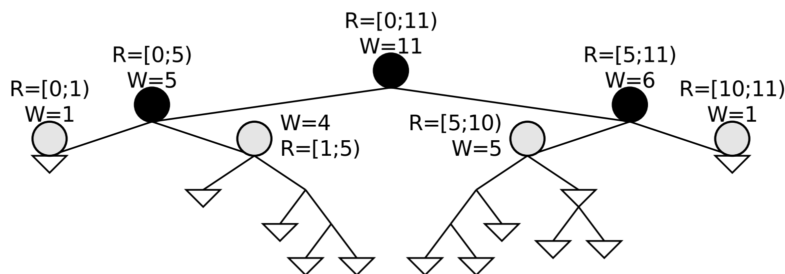


Figure 5.19: Annotation of a cluster with weights W_j and ranges R_j from left to right [SWB13a]

.

Then, W_j represents the workload of all child clusters for a particular cluster tree node j and R_j the range $[R_j, R_j + W_j)$ of workloads, see Fig. 5.19. We continue using these annotations for scheduling and generation decisions.

5.6.1 Scheduling strategies

We consider different scheduling strategies to assign the tasks on clusters to compute units. In this context, tasks are any kind of operations to be executed on clusters: time step, particular adaptivity traversal, backend, setup, etc.

Owner compute and affinities

With p compute units, let $W_{avg} := \frac{\mathbf{W}}{p}$ be the average workload for each compute unit. Each cluster i is then assigned to compute unit

$$j := \left\lfloor \frac{R_i + \frac{W_i}{2}}{W_{avg}} \right\rfloor . \tag{5.1}$$

To avoid computations of this association for each traversal of the cluster tree, we cache the range of thread ids i fulfilling Eq. (5.1) and store it to each leaf node during the dynamic cluster-generation traversals. This yields a unique ownership of each cluster. Furthermore, we also reduce this information bottom-up and extend the annotation of each node in the cluster tree by a range of compute units which own at least one leaf node of the currently processed node.

We implement two different scheduling strategies by either traversing only tree nodes for which Equation (5.1) holds or by setting task affinities for tasks created on each node. These two scheduling strategies are denoted with *owner-compute* and *affinities* and these strategies can be used to consider the *spatial locality* of each cluster on a NUMA memory hierarchy.

Task flooding

Using the task-flooding scheduling, we create a task for each cluster tree node. This leads to operations (task) being executed via work-stealing mechanisms with the used shared-memory parallelization models.

5.6.2 Cluster generation strategies

We present two different strategies for the cluster generation in this section.

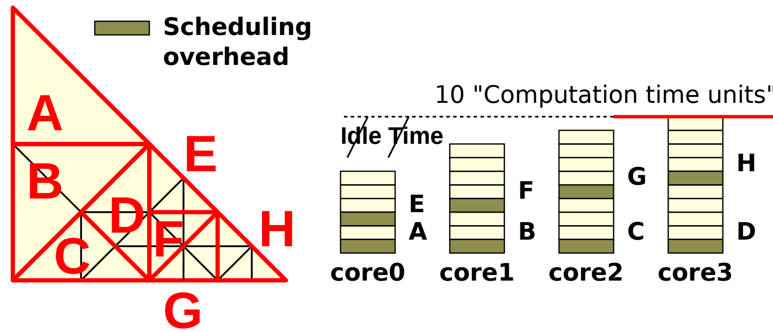


Figure 5.20: Using massive splitting creating by far more clusters than there are processors.

Load-balanced splitting

An owner-compute scheme belongs to the classical domain decomposition since it directly aims at assigning computations in a static way to compute units. For perfect load balancing, equal workload should be assigned to each compute unit. Assigning clusters to compute units, the load balancing can be improved by splitting clusters in which cells can be assigned to more than one compute unit. Given p compute units, the average workload per compute unit is given by $W_{avg} := \frac{\mathbf{W}}{p}$. This leads to splitting a cluster i, if

$$\left\lfloor \frac{R_i}{W_{avg}} \right\rfloor \neq \left\lfloor \frac{R_i + W_i - 1}{W_{avg}} \right\rfloor .$$

This targets at improved balancing of work decomposition and considers only the pure workload under the assumption that there are overheads until the initialization of computations on each cluster.

Massive splitting

An alternative parallelization method is given by massive splitting. We use the assumption that the overhead for starting computations on a cluster is relatively small compared to the computational workload in each cluster. This allows us to create more clusters compared to the number of compute units available and to efficiently process them due to the small overhead. Contrary to a 1:1 scheduling of tree splits which would yield high idle time due to work imbalances, we expect decreased idle time by creating by far more clusters than there are compute units available (See Fig. 5.20).

For massive splitting, we split each cluster if its *workload exceeds a particular threshold* T_s and join two cluster, if the *workload of both child clusters undershoots* T_j. The join threshold can therefore be evaluated cluster-locally only, and we only compare the boolean agreement to the join operation of both child clusters.

In this work, we use a join threshold $T_j := \left\lfloor \frac{T_s}{2} \right\rfloor$ with T_s representing the split threshold. Selecting a join threshold in this way assures that a join of two cluster does not directly result in a split due to joined clusters directly exceeding the split threshold.

5.6.3 Threading libraries

With the growing number of cores on shared-memory systems, a high variety of threading libraries are available. Considering dynamically changing resources (see Part IV), also support for such extensions to a threading model is required.

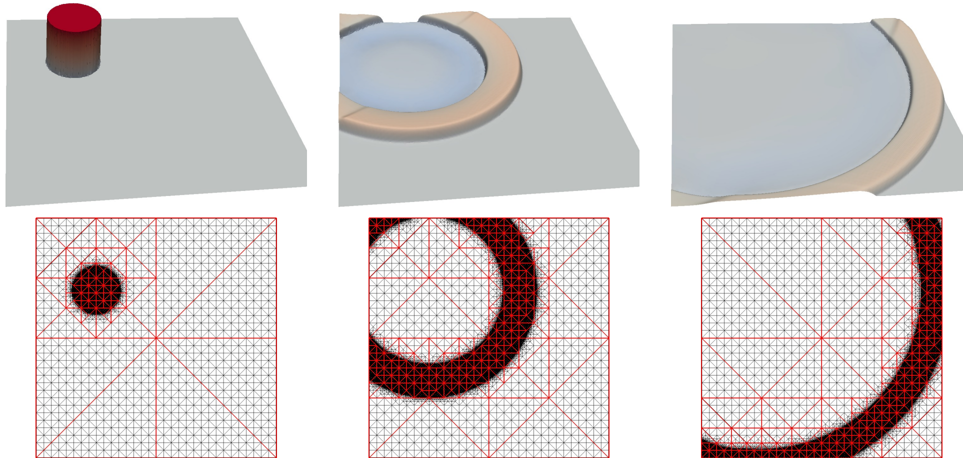


Figure 5.21: Visualization of three different time steps for a representative radial dam break test scenario. A radial dam break is used for most of the benchmarks in this section [SWB13a].

To introduce the parallel execution of operations on each cluster (see Section 5.6), we consider two different threading tools which are representative for other thread-based parallelization models: OpenMP[3] and Threading Building Blocks (TBB)[4]. Both threading libraries offer parallelism in different ways. With *OpenMP*, language extensions via pragma precompiler directives are used to express parallelization in the program code, whereas parallelization features of *TBB* are made available to the application developer via C++ and even more convenient features via C++11 language features. Since information on both parallelization methods is available in other work and reference guides, we refer the interested reader to the corresponding websites and reference guides for an introduction and only highlight how these parallelization models are used in our development.

Both threading extensions provide support for the very basic parallelization features required for our cluster-based framework: parallel processing of a *for-loop* as well as *tasking*. We implemented three different ways for a parallel execution of operations on clusters.

(a) An *owner-compute* implementation starts a task for each compute unit via parallel execution of a *for-loop* over the range of all compute units. Each for-loop iteration and, thus, each compute unit executes operations only for a subset of clusters for which Eq. (5.1), page 107 holds.

(b) Creating a *task for each cluster-tree node*. In case of massive splitting, this leads to a flooding of the overall system with tasks and uses work stealing with both considered threading libraries. During the traversal of our cluster tree, the implementation emits a new task for each cluster tree node.

(c) We can further use features offered by TBB to create *tasks with affinities* to compute units. Such an attribute leads to enqueuing tasks into the working queue of the thread for which the affinity was set for.

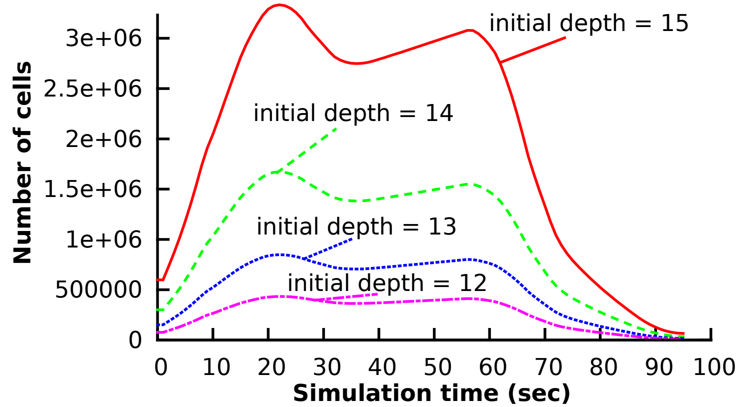


Figure 5.22: Changing number of cells over time for shallow water simulations executed with different initial refinement depths. See the text for detailed information on the scenario [SBB12].

5.7 Results: Shared-memory parallelization

We conducted several benchmarks with the test platforms further denoted as Intel and AMD. All computations are done in single precision. A sketch of a shallow water benchmark which is simulating a radial dam break is given in Fig. 5.21. Detailed information on both systems is given in Appendix A.2.

The first benchmark scenario is given by a flat sea-ground 300 meters below the sea surface and a domain length of 5 km. We use up to 8 levels of refinements and allow the refinement/coarsening triggered according to the surface above/below 300.02/300.01 meter. Our domain is set up by two triangles forming a square centered at the origin. The initial refinement depth is given by d, resulting in 2^{d+1} coarse triangles without adaptivity. The dam is placed with its center at $(-1250m, 1000m)$ with a radius of $250m$. The positive water displacement is set to $1m$. Performance results are given in “Million Cells Per Second” (MCPS) that can be computed.

The dynamically changing number of cells over the entire simulation is given in Fig. 5.22. A visualization of a similar scenario with an initial radial dam break is given in Fig. 5.21.

Threshold-based split size

With the threshold-based cluster generation approach (see Sec. 5.6.2), we expect overheads introduced depending on the number of clusters (i.e. on the maximum number of cells in each cluster). The following benchmark is based on a regular and non-adaptive grid. We tested different initial refinement depths d, each depth resulting in a grid with 2^{d+1} cells. The simulation is based on 1st-order spatial- and time-discretization of the shallow water equations.

The results in Fig. 5.23 show the dependency of the efficiency on the cluster split threshold. We account for these performance increases and decreases with *tasking overheads*, *synchronization requirements* and *load imbalances*. These aspects are further described and performance increase and decrease is depicted with ↑ and ↓, respectively, in the following table:

[3] `http://www.openmp.org/`
[4] `http://threadingbuildingblocks.org/`

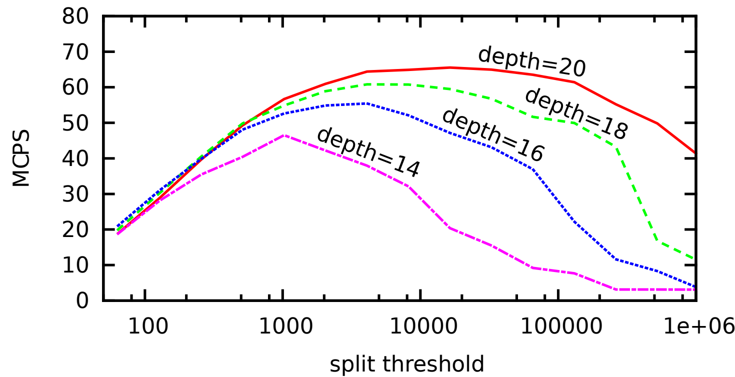


Figure 5.23: MCPS for a simlation on a regular static grid. The performance is given for different split thresholds and simulation workload (d={14,16,18,20})executed on 40 cores on platform Intel [SBB12].

	Smaller thresholds	**Larger thresholds**
Tasking overhead	↓ Smaller thresholds lead to more clusters, therefore *more overheads* for starting tasks.	↑ With larger thresholds, less clusters are created, thus reducing the overhead related to the number of tasks.
Replicated interfaces	↓ More clusters lead to more replicated interfaces and thus to *more reduction operations*.	↑ For larger thresholds, less additional reduction operations are required.
Load imbalances	↑ Smaller clusters increase the potential of *improving the load balancing*.	↓ For larger thresholds, the potential of load imbalance is increased.

Thus, the optimal performance depends on the problem size, the number of cores and the splitting of the clusters.

Threading overhead

We continue with a more detailed analysis of the overheads introduced by threading and, if not otherwise mentioned, fix the split threshold size to $T_s := 32768$, i.e. the number of clusters is kept constant. Several benchmark settings were used:

- *Serial:* This version is based on the serial software design, see Fig. 4.12. In particular, no cluster set is used and no threading overheads exist.
- *Parallel, no splitting, 1 thread:* This version and the upcoming described versions are based on the parallel software design, see Fig. 5.10. The splitting threshold was set to infinity.
- *Parallel, 1 thread:* The simulation was executed with a single thread and the cluster generation is based on the cluster split threshold size T_s.
- *Parallel, N threads:* The simulation was executed with N threads in parallel with the cluster split threshold size T_s.

The threads are pinned to non-hyperthreaded cores in a compact way. A simulation domain set up by a single base triangle with a typical radial dam break scenario was used. The maximum refinement depth was set to 8 and 100 simulated seconds were computed. The results, based on

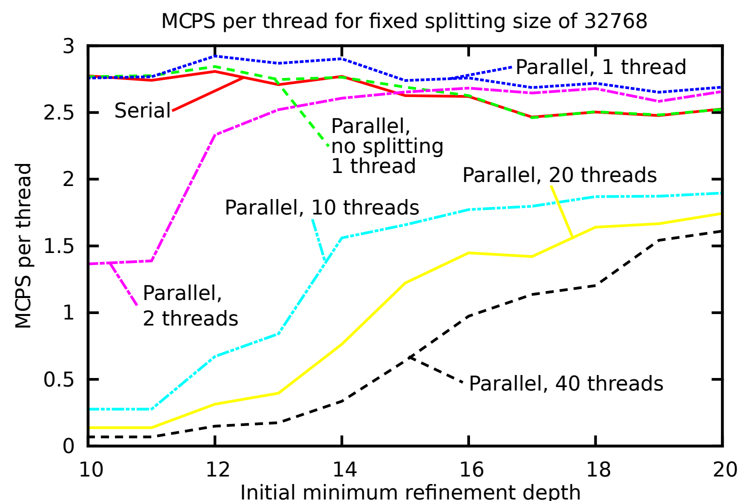


Figure 5.24: Million cells per second (MCPS) for simulations on on a regular static grid. The performance is given for the serial version and the parallel version with different number of threads and split thresholds on 40 cores on platform Intel [SBB12].

the million processed cells per second divided by the number of threads, are given in Fig. 5.24. The benchmarks were conducted on the platform Intel and we discuss the benchmark data in the order of the test runs.

- *Serial:* The serial version slows down after a refinement depth of 12 to 14. The reason for this is the problem size exceeding a cache level size.
- *Parallel, no splitting, 1 thread:* This version almost coincides with the serial version. Thus, we can conclude that *almost no overhead is introduced by the additional layers for the parallelization.*
- *Parallel, 1 thread:* With a larger problem size, we see that a parallel approach leads to an improved performance compared to the serial version only. Detailed traversal statistics show a performance improvement in both the simulation and adaptivity traversals.

  We account for this improved performance with a *cache-blocking* effect: Our clustering splits the domain into independent chunks with a size of equal or less than $2^{15} = 32768$. This generates clusters with a size also fitting at least into the last-level cache. To account for the temporal data reaccesses which is required for such a cache effect, we reconsider our stack access strategies between traversals:

  (a) In the edge-based communication scheme used in this implementation, we pushed the flux DoF stored on both edges to a buffer and computed the fluxes after the cluster's grid was traversed. Thus, the DoF stored on this buffer can still be available in cache and accessed with less latency.

  (b) Regarding the adaptivity traversals, we only use the structure and adaptivity stack as input. In our implementation, we only access 2 bytes on the structure and 1 byte for the adaptivity state stack for each traversed cell during the traversals generating a conforming grid. This perfectly fits into a higher cache level. By atomically executing the forward traversal directly after a backward traversal in one operation executed on the cluster, the structure and adaptivity information is still kept in a higher cache level and thus does not have to be written back to main memory.

- *Parallel, N threads:* The simulation was executed with N threads in parallel. Effects based on the cache-optimizations of the single-threaded version above could also be measured for the 2-threaded simulation.

  For higher number of threads (10, 20 and 40), no further improvements due to cache-blocking could be gained for the considered problem sizes. Simulations and the scalability with a larger number of threads are discussed in the next sections.

We conclude that our approach leads to optimizations with cache-aware cluster sizes and, hence, can lead to a memory bandwidth reduction for dynamically adaptive meshes.

Strong-scaling benchmarks

We continue with strong-scaling benchmarks on shared-memory test platforms Intel and AMD (see Appendix A.2). On the Intel platform, the first 40 cores are the physical ones, followed by 40 cores which are the hyperthreaded ones. The simulations were initialized with a minimum refinement depth of 20. During the setup of the radial dam break, we use up to 8 additional refinement levels. We computed 100 time steps, leading to about 19 Mio. grid cells processed in average per time step. Tasking from TBB and OpenMP was used for the evaluation.

If not stated differently, we consider NUMA domain effects and try to avoid preemptive scheduling with context switches by pinning threads to computing cores. This is also referred as setting the thread affinities. We evaluated three different core-affinity strategies:

- *no affinities:* This implementation does not use any pinnings. Thus, the threads can be preemted and their execution continues on another core.

- *A1*: This pins all threads in a linearly increasing manner following the core enumeration. On platform Intel, the enumeration is priorized first with all non-hyperthreaded cores on a socket, then over all sockets, followed by all hyperthreaded cores on all sockets. The first 10 threads are then pinned to the non-hyperthreaded cores on the first socket, the next 10 threads to the second socket, etc. Threads 41 to 50 are pinned to the hyperthreaded cores.

- *A2*: Assuming a core enumeration starting with 1, this pinning skips every second core until all odd-numbered cores are consecutively assigned. The next threads are then pinned to the even-numbered cores. With our platform Intel and using 40 threads with A2 affinities, this results in the first 20 threads being assigned to the odd-numbered physical cores followed by the odd-numbered hyperthreaded cores on all sockets.

The results for the platform Intel and AMD are given in Fig. 5.25 and 5.26 and are discussed next.

- *Platform Intel:*
  Using TBB, the best scalability can be measured for this platform with A1 scheduling. With the A2 scheduling, the scalability gets worse after 20 threads due to threads 21 to 40 being assigned to the hyperthreaded cores with their non-hyperthreaded cores already used for computations.

  With OpenMP, less scalability is gained in general for the considered test case. We created tasks with the untied clause for each cluster tree node. Due to known issues for OpenMP task constructs (see e.g. [OP09]), the scalability is not as good for unbalanced trees as by using TBB for parallelization.

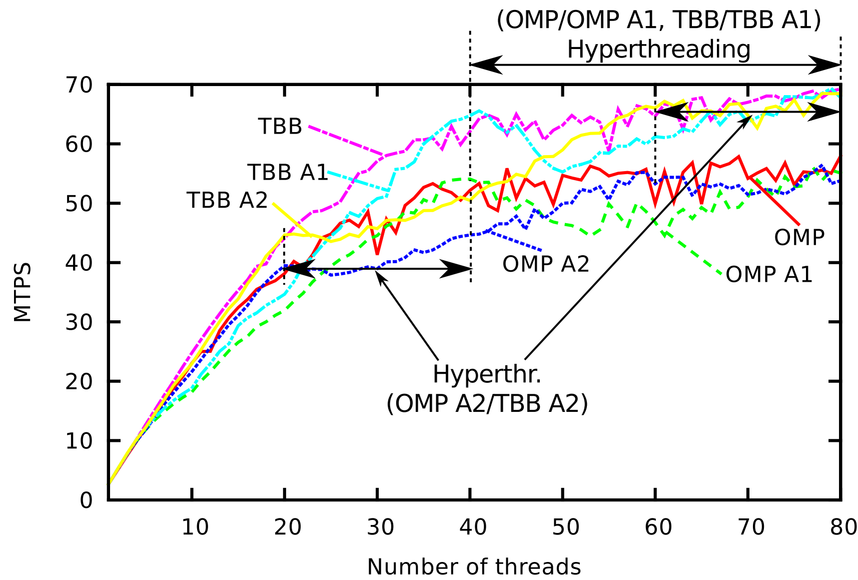


Figure 5.25: Strong scaling on a 40-core Intel platform with 40 additional hyperthreads [SBB12].

- *Platform AMD*:
  Sufficient scalability could be also determined for the AMD platform with an overall performance close to the Intel system. Since the considered AMD platform shares the floating point units (FPU) for 2 particular cores in the architecture, we account for the bend visible at multiples of 8 for A1 and A2 affinities by this FPU sharing.

Strategies for scheduling and cluster generation

We next consider different combinations for scheduling, determining which thread executes computations on which clusters (see Sec. 5.6.1) and how to dynamically generate clusters (see Sec 5.5). The benchmark scenario is based on a 2nd-order discretization in space and 1st-order discretization in time.

Regarding the *scheduling stragtegies*, we shortly recapitulate them in the following table:

Scheduling	Description
Owner-compute	The clusters are assigned in a fixed way to the threads. No work stealing is possible.
Affinities	Only affinities are set, enqueuing tasks executed for a cluster to the worker queue of the owning thread. This still allows work stealing.
Task-flooding	No affinities are used. With current threading models TBB and OpenMP, this enqueues the task to the worker queue of the enqueuing thread.

We also developed two different cluster-generation approaches shortly recapitulated in the next table:

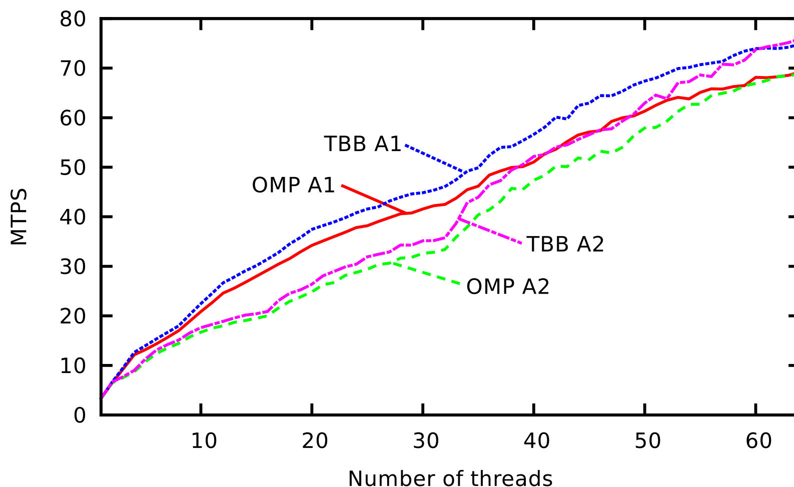


Figure 5.26: Million cells per second (MCPS) for strong scaling on a 64-core AMD platform [SBB12].

Cluster generation	Description
Threshold-based	The clusters are split or joined based on whether the local workload (cells) exceeds or undershoots the thresholds T_s or T_j. Thus, this is a *local cluster generation criterion.*
Load-balanced splitting	The clusters are split and joined based on the range information (see Sec. 5.6.2) and split in case that the cluster can be assigned to more than a single thread. *This is a global cluster generation criterion.*

We conducted benchmarks with different combinations of the above mentioned scheduling and cluster generation strategies (See Fig. 5.27). Here we see that a cluster generation approach with range-based treesplits does not result in improvements as expected, but leads to less performance. We account for that by additional splits and joins compared to the pure threshold-based method. Such splits and joins of clusters result in additional overhead due to synchronizing updated meta information and additional memory access to split and join the cluster. Using a threshold-based cluster generation, by far less changes due to cluster splits and joins are required. This leads to improved performance for the threshold-based clustering.

5.8 Cluster-based optimization

We continue with an overview of possible optimizations:

- With clustering based on a replicated data scheme, reduce operations are required in order to synchronize the replicated data. This synchronization can be achieved via duplicated reduce operations executed on each cluster or a single reduce operation per shared hyperfaces by considering the SFC order of the clusters. Details on this optimization are given in 5.8.1.
- We expect performance improvements by cluster-based local-time stepping (see Section 5.6). With communication interfaces (sequentially stored and duplicated) similar to dynamic

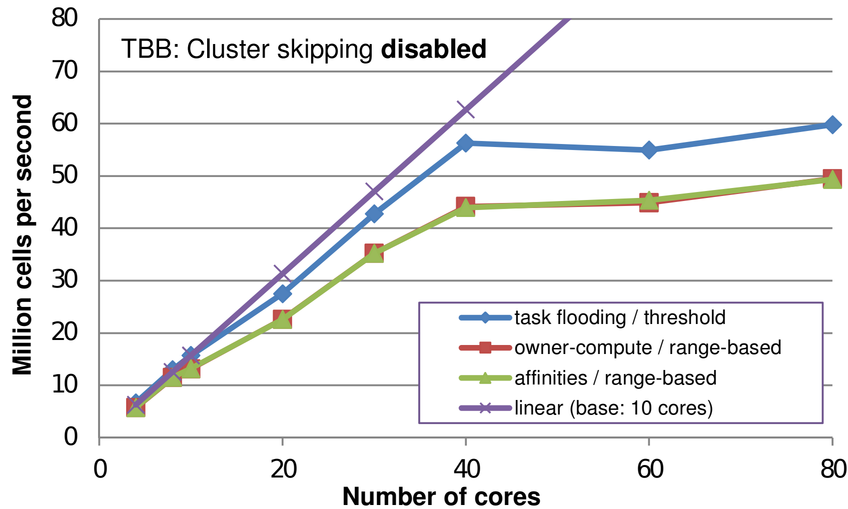


Figure 5.27: Overview of scalability with different cluster generation strategies. The legend is given in the format (scheduling strategy / cluster generation) [SWB13a].

adaptive block-structured grids in the Peano framework [UWKA13], this would yield a similar algorithm, and thus we did not focus on such an implementation.

- Considering iterative solvers, we can skip computations on clusters. This allows implementation of the local relaxation method [Rüd93] with multiple clusters per compute unit.

- With grid traversals only required for a particular spatial area, traversals on clusters which are out of this area can be directly skipped during traversals. The bounding triangles for our tree-split based clusters is directly available in the meta information which is stored for each cluster. Thus, optimizations such as software-controlled frustum culling [GBW90, LP01] are possible. We optimized the sampling of simulation data at particular points (e.g. buoy data for Tsunami simulations or values for analytical benchmarks) by skipping clusters which do not cover the sampling point, see e.g. Section 6.3 for an application.

- One of the main building blocks of this framework is a conforming grid within each cluster. The consistency traversals for a conforming grid generation are typically executed by traversing the entire grid. However, for clusters already in a conforming grid state and without any edge creation requests from adjacent clusters, no additional adaptivity traversals would be required. More information on this optimization is given in Section 5.8.2.

- With the parallelization typically getting stuck during the output of simulation data, we can use a threaded parallelization model and a dedicated writer task or thread. This task/thread stores the output data to persistent storage whereas the other threads can continue running computations. These optimizations are presented in the Section 6.4.1.

5.8.1 Single reduce operation for replicated data

Using the replicated communication scheme (see Section 5.2), there are basically two ways to reduce the replicated data:

- The first one is accessing the replicated data of the adjacent cluster in a *read-only manner and executing the reduce operation twice for each cluster*. This strictly follows a

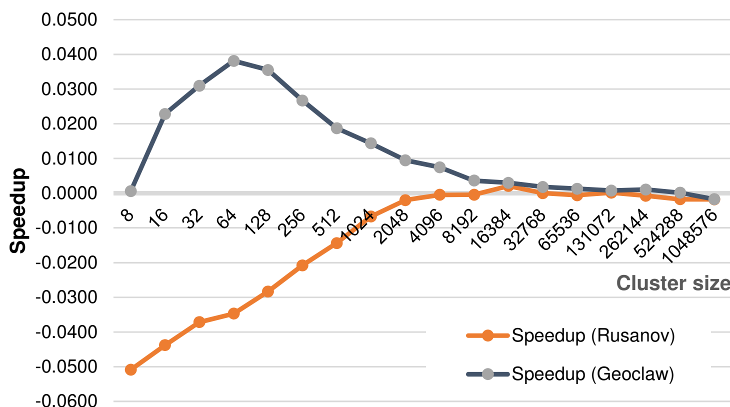


Figure 5.28: Performance speed up and speed down if avoiding duplicated reduction of replicated data.

distributed-memory parallelization with the replicated data sent to the adjacent cluster, followed by a reduce operation.

- We present an alternative approach by executing the *reduce operation only once for the replicated data*.

  First, we have to make a decision which cluster should compute the reduction. We can determine such a cluster which is responsible for computing the reduction by considering the SFC order of the first cell in each cluster or the order of the unique ids assigned to each cluster since they also follow the SFC order (see Section 5.3.4). In our development, we use the cluster's unique ids to determine the "owner" of the reduction operation.

  Second, the owner computes the reduce operation and stores the reduced data which is to be processed by the adjacent cluster to the communication buffer. Thus, this range of the communication buffer does not contain anymore the data which is read for the reduction, but the already reduced data. An *additional synchronization* is required to the next step, which reads the reduced data to avoid race conditions.

  Third, the "non-owner" can fetch the data from the communication buffer which is already reduced e.g. by a flux computation and the data can be processed avoiding duplicated reduce operations.

We conducted two experiments based on a shallow water simulation with a radial dam break, 18 initial refinement levels and up to 8 additional levels of adaptivity. Since we assume that the benefits of avoiding duplicated reduce operations depend on the costs of the flux solvers, we run two different benchmarks - each one with a different flux solver: the relatively *lightweight* Rusanov solver and the *heavyweight* augmented Riemann solver from the GeoClaw package [Geo08].

The saved computations also depend on the number of shared cluster interfaces, hence we executed benchmarks parametrized with different split thresholds. The results are based on a single-core execution and are given in Fig. 5.28 and they show, that possible performance improvements depend on the utilized flux solver.

1. For the *lightweight flux solver*, avoiding reduce operations results in a performance decrease for smaller clusters. An explanation for this effect is given by the threading overheads: since we require additional synchronization operations, there is an additional overhead

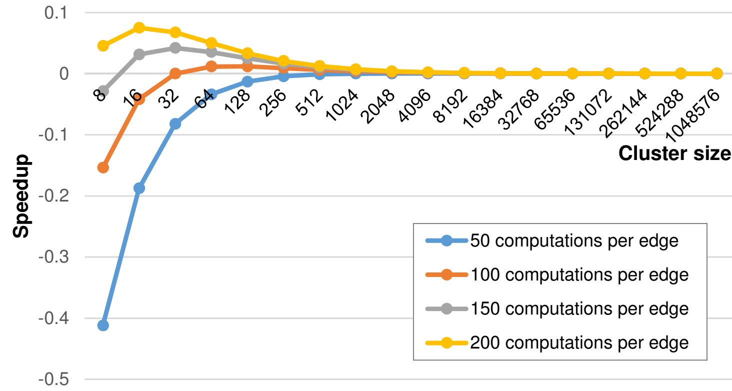


Figure 5.29: Performance speed up and speed down if avoiding duplicated reduction of replicated data.

introduced by executing additional tasks. For the Rusanov flux solver, the synchronization overhead is higher than the benefits.

2. Using *heavyweight solvers*, we see a speedup for optimization and assume that the computational intensity of the solver outperforms the tasking overheads for the additional synchronization.

To verify these assumptions, we develop a simplified model based on the following quantities:

- d: We relate the number of cells in the cluster to the subtree's refinement depth d.
- o: This accounts for the tasking overhead costs per cluster, e.g. the computations to dequeue and start a task.
- e: Computational time per edge.
- c: Computational time per cell.
- b: The number of cluster boundaries which we approximate to linearly depend on d. We approximate this with $2d$.

Since the computational speedup can also be computed for each cluster, our model only considers the computational costs for a single cluster. These costs for a cluster with a single reduce operation are then given by

$$C_{(single)}(o,e,c,d) := \underbrace{o}_{\text{Additional synch. overhead}} + \underbrace{2^d \cdot c}_{\text{Computations for cells}} + \underbrace{1.5 \cdot 2^d \cdot e}_{\text{Computations for edges}}$$

whereas the costs with two reduce operations per shared hyperface are given by

$$C_{(double)}(o,e,c,d) := \underbrace{2^d \cdot c}_{\text{Computations for cells}} + \underbrace{1.5 \cdot 2^d \cdot e}_{\text{Computations for edges}} + \underbrace{2d \cdot e}_{\text{Additional comp. for edges}} .$$

We compute the speedup using the single evaluation with $\frac{C_{(double)}-C_{(single)}}{C_{(double)}}$ and use rule-of-thumb values $o = 1000$, e.g. 1000 cycles to dequeue a task from the worker's queue and $c = 100$ as the computational workload per cell. The speedup graph for different refinement depths and thus cluster sizes is given in Fig. 5.29 for different computational costs for edges.

The shapes of our experimentally determined results in Fig. 5.28 match to our model. We find the agreement, that in case of a low computational cost involved in each edge, there's no benefit in using a single reduce operation. For computational intensive edge reductions as it is the case for the augmented Riemann solver, using a single reduce operation can be beneficial. However, as soon as the number of cells in each cluster is getting higher, the benefits tend towards zero.

We conclude, that for small cluster sizes, avoiding duplicated reduction operations can be beneficial for small cluster, e.g. required for strong-scaling problems and the algorithm presented in the next section.

5.8.2 Skipping of traversals on clusters with a conforming state

So far, we only considered adaptivity conformity traversals being executed by traversing the entire domain grid. For parallelization, we decomposed our domain with clusters and replicated data scheme to execute traversals massively parallel also in a shared-memory environment. For adaptivity traversals aiming at creating a conforming grid, the cluster approach is similar to a red-black coloring approach with information on hanging nodes possibly not forwarded to adjacent clusters in the current adaptivity traversal. Thus more traversals can be required in case of parallelization.

However, a clustering approach offers a way to compensate these conformity traversals if there are more than one clusters per compute unit: adaptivity traversals can be skipped on clusters that (a) already have a conforming grid and (b) do not receive a request of an adjacent cluster to insert an edge (adaptivity marker M_R). Both requirements are discussed next:

(a) To determine whether the adaptivity states already lead to a conforming cluster, we tag each cluster with a "non-conforming" boolean which is set to *false* before any adaptivity traversals. During the adaptivity traversals, as soon as information on inserting an edge is not propagated with the current traversal, this boolean is set to *true*. Such information on non-propagated hanging nodes is directly given by the *forward information bits* f (see Section 4.10.3): If these bits are not equal to zero, adaptivity information still has to be propagated by an additional traveral.

(b) Testing for adaptivity request propagations from adjacent clusters is accomplished by checking the left and right adaptivity communication buffers stored at the adjacent cluster which store adaptivity markers M_R for corresponding requests. In case that all values are not set to M_R, no adaptivity edge-insertion requests are propagated from the adjacent cluster.

This algorithm is based on the sparse graph representation (see Fig. 5.5 of our clustered domain: using this graph representation, our algorithm can be represented by a Petri net [SWB13a].

Algorithm: Skipping of clusters with a conforming adaptivity state

Here, each cluster is a node and the directed arcs exist for each RLE entry. Tokens represent requests for additional conformity traversals. We then distinguish between *local consistency traversal tokens* and *adjacent consistency tokens* with the latter one forwarded to, or received from adjacent clusters.

- *Local consistency tokens:*
  These tokens are generated by local consistency traversals (see (a) above) and are

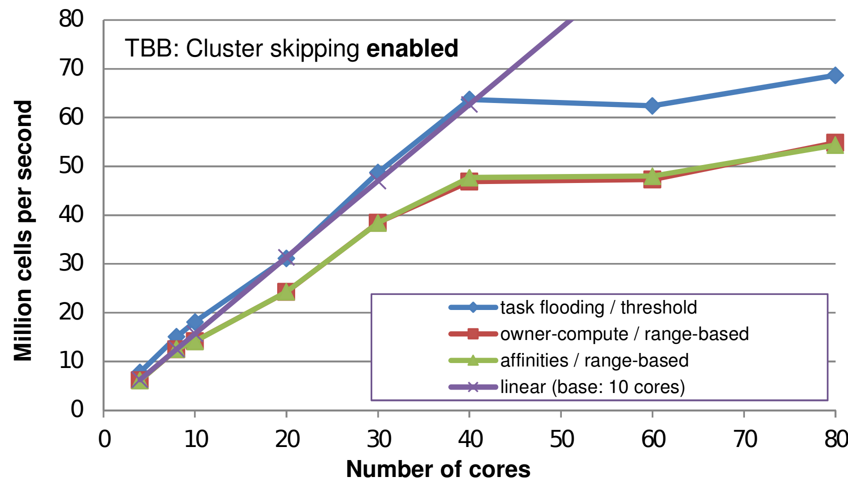


Figure 5.30: Overview of scalability using different cluster generation strategies with adaptive conforming cluster skipping enabled. The legend is given in the format "scheduling strategy / cluster generation" [SWB13a].

injected to each cluster's graph node in case of the cluster still not in a conforming grid state. These tokens are removed after each traversal, e.g. by a transition requiring one or more tokens and firing no token.

- *Adjacent consistency tokens:*
  Tokens are fired via the connectivity graph edges to an adjacent cluster in case that a refinement marker is stored on the corresponding RLE encoded communication data. In case that such an adjacent consistency token is received from an adjacent cluster, an additional conformity traversal is started.

Results for short-term simulations:

We conducted several runs for the SWE benchmark scenario from Sec. 5.7. The results are given in Fig. 5.30 with the linear scalability shown for the implementation without adaptive conforming cluster skipping. In compliance with the results for the non-skipping version, the range-based cluster generation suffers of additional overheads due to more frequent cluster generation operations. However, considering the pure task-flooding with a threshold-based cluster generation, we get a higher performance compared to the basic implementation due to algorithmic cluster-based optimizations.

Detailed statistics on the skipping algorithm:

We further analyze the robustness of the adaptive cluster skipping and detailed statistics on the run time separated into the three major phases of a single simulation time step: (a) generating clusters, (b) adaptivity traversals and (c) time step traversals. The results are given in Fig. 5.31 for a different number of threads executed on the same benchmark scenario. For all tested numbers of threads, the skipping algorithm yielded a robust performance improvement. The time spent for cluster generation is negligibly small. We account for that by the required decreased

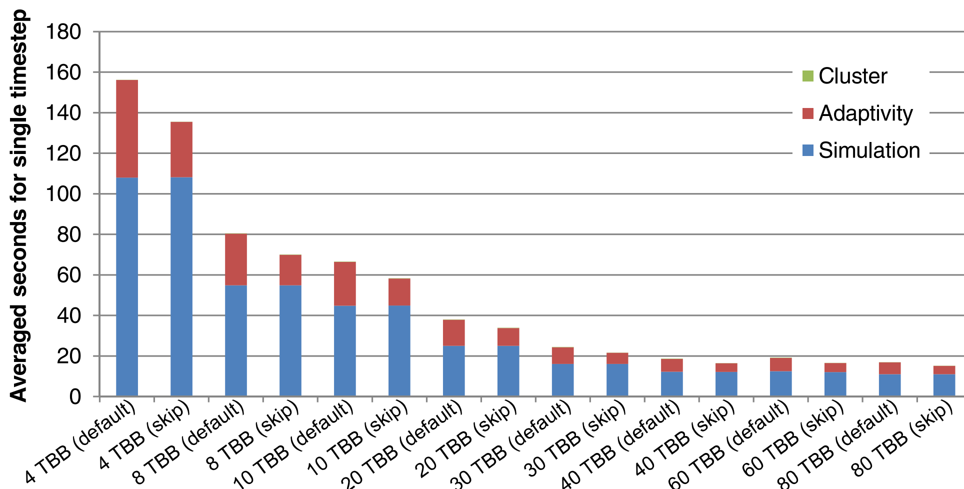


Figure 5.31: Comparing run time of skipping and non-skipping of conforming cluster. Here, we distinguish between the cluster generation time (Cluster) which is negligibly small, time for adaptivity tarversals (Adaptivity) and time to run the computations for the time step (Simulation) [SWB13a].

time step size due to the higher-order spatial discretization. This also leads to less changes in the grid and therefore less executions of cluster generation. The presented adaptive cluster skipping also leads to robust performance improvements on distributed memory simulations, see Sec. 5.12.1.

5.8.3 Improved memory consumption with RLE meta information

Our run-length encoding provides an optimization for the blockw-wise data communication for stack-based communications. This section highlight the potential of saved memory storage by using our RLE meta information.

Improved memory consumption results based on simulation

With statistics gathered for executed simulations, we compare the number of RLE-stored meta information to the number of elements required if we would store the communication primitive separately [SWB13b].

Here, we assume the meta information on vertices associated to one RLE edge communication entry is not stored, but inferred by the per-hyperface stored edge meta information. To give an example, we assume an RLE encoding $((A,2),(B,0),(C,3))$ to be stored with $((A,e),(A,e),(B,v),(C,e),(C,e),(C,e))$, see Sec. 5.2.3. We then compute the ratio between the quantity of entries required for our RLE scheme $Q_R := |((A,2),(B,0),(C,3))|$ to the number of non-RLE encoded entries $Q_N := |((A,e),(A,e),(B,v),(C,e),(C,e),(C,e))|$. This ratio $Q := \frac{Q_N}{Q_R}$ then yields the factor of memory saved with our RLE meta information compared to storing the meta-communication information separately for the communication hyperfaces.

We conducted empirical tests with a hyperbolic simulation based on (a) the Euler and (b) the shallow water equations. Here, we used edge-based communication for the simulation and node-based communication for a visualization in each time step. The simulation is started with an initial refinement depth of $d = 6$ and up to $a = 14$ additional refinement levels. The simulations are initialized with the same radial dam break, using the density as the height for

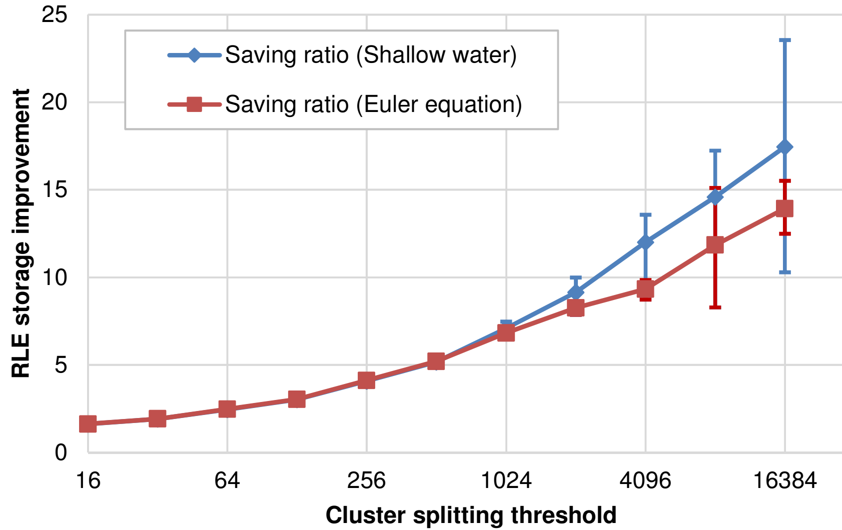


Figure 5.32: Ratio of entries for RLE meta information to non-RLE meta information for different cluster sizes for the simulation based on the shallow water and Euler equation. The error bars show the minimum and maximum savings of RLE meta information over the entire simulation time [SWB13b].

the Euler simulation, and we simulate 100 seconds. This resulted in execution statistics given in the table below:

	time steps	min. number of cells	max. number of cells
Euler	31	34593	40228
Shallow water	733	34593	178167

We plot the ratio Q for our simulation scenario above in Fig. 5.32. Larger clusters lead to an increased ratio due to more shared hyperfaces per cluster and hence more entries to be compressed with our RLE meta information. The min/max bounds for the largest tested cluster size with the Euler simulation is not as large as for the shallow water equation. We account for that by two statistical effects: the first one is a shorter run time which leads to less possible values involved in the min/max computation. The second one is, that less time steps for the Euler simulation also lead to less cells generated in average with a maximum of 40228 cells over the entire simulation (see the table above). This leads to only a low average number of clusters during the entire simulation, almost no dynamic cluster generation in the simulation and thus less values involved into the min/max error range. For typical cluster sizes of more than 4000 cells, the memory requirements for the meta information are reduced by factor of 9.

RLE vs. per-cell stored meta information

For simulations executed on unstructured grids, the meta communication information is typically stored for each cell. This is considered to be especially memory consuming for meta information on vertices due to more than a single adjacent cell compared only a single adjacent cell with edges. However, the ratio of the meta information overhead per-cell compared to our cluster-shared-hyperface RLE encoding depends on the number of degrees of freedom stored in each cell which we further analyze with a model [SWB13b]. Here, we model the payload (DoF) per cell with W and assume a regular refinement depth d, yielding 2^d cells per cluster. We also assume that the non-RLE meta information takes the same

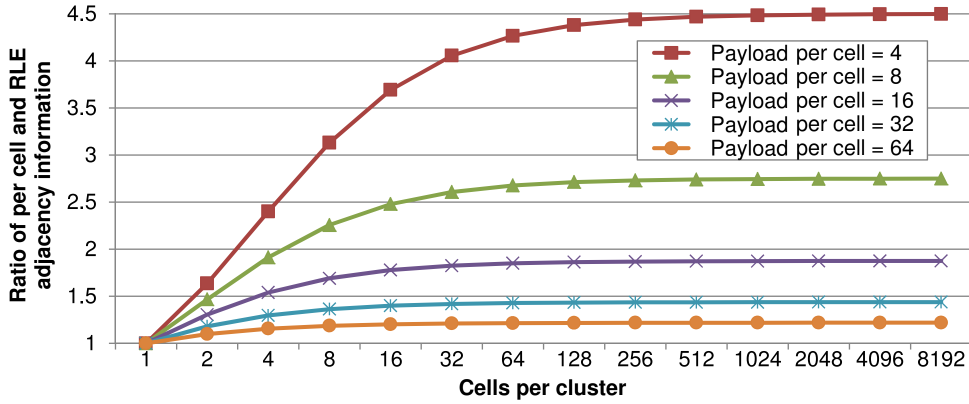
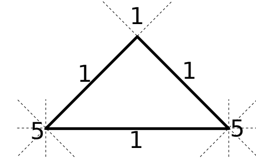


Figure 5.33: Ratio of saved memory by using our RLE encoding compared to per-cell stored adjacency information [SWB13b].

amount of memory as a single DoF. Furthermore, we neglect the requirement of storing the cell orientation since this can be bit-compressed and stored in a cell requiring less storage than a single DoF. Such a cell orientation can be required if accessing the adjacent cell. With the grid generated by the bipartitioning Sierpiński SFC and based on our assumption of a regular grid, the required memory for the meta information on adjacent cells for each triangle cell and the workload for each cell are given by



$$\begin{aligned} S := \quad & 3 && \text{(3 edge adjacency information)} \\ & + 2 \cdot 5 + 1 && \text{(vertex adjacency information)} \\ & + W \end{aligned}$$

with 3 edge-adjacent cells and $2 \cdot 5 + 1$ required meta information on the per-vertex additional cells, see right handed figure.

Storing the meta information only for the cluster boundaries, this yields $R := 2 \cdot (3+2 \cdot 5+1)$ RLE entries to store the adjacent information with the factor 2 accounting for the two-element tuple for each entry.

The ratio of the number of cells times the memory requirements to the RLE encoded memory requirements then yields the memory saved for each cluster:

$$\frac{S \cdot 2^d}{R + W \cdot 2^d} = \frac{(3 + 2 \cdot 5 + 1 + W) \cdot 2^d}{R + W \cdot 2^d} \overset{2^d \to \infty, L.H.}{=} \frac{3 + 2 \cdot 5 + 1 + W}{W} = \frac{14}{W} + 1$$

We get an upper bound of $\frac{14}{W} + 1$ which only depends on the per-cell payload W.

We plotted this ratio for different cluster sizes and DoF per cell in Fig. 5.33. First, this plot shows that *less memory is saved for smaller cluster sizes*. We account for this by clusters with less cells requiring the same amount of RLE meta information compared to clusters with more cells. Hence, for smaller clusters, the benefit of the ratio of cells to RLE encoding is getting smaller. Second, for larger cluster sizes, the plot shows the *potential of memory savings* for simulations with a *small number of DoF per cell*. According to the simplified model, this can reduce the memory consumption up to a factor of 4.5. Third, with a larger number of DoF per cell, as it is the case for higher-order simulations or by storing regular sub-grids for each cell, the benefit of our stack- and stream-based simulations with RLE meta information is below 1.5 for more than 32 DoF per cell. Though, the advantages of the RLE-based communication are still given e.g. by the block-wise communication and cluster-based data migration for distributed-memory systems (Section 5.10.3).

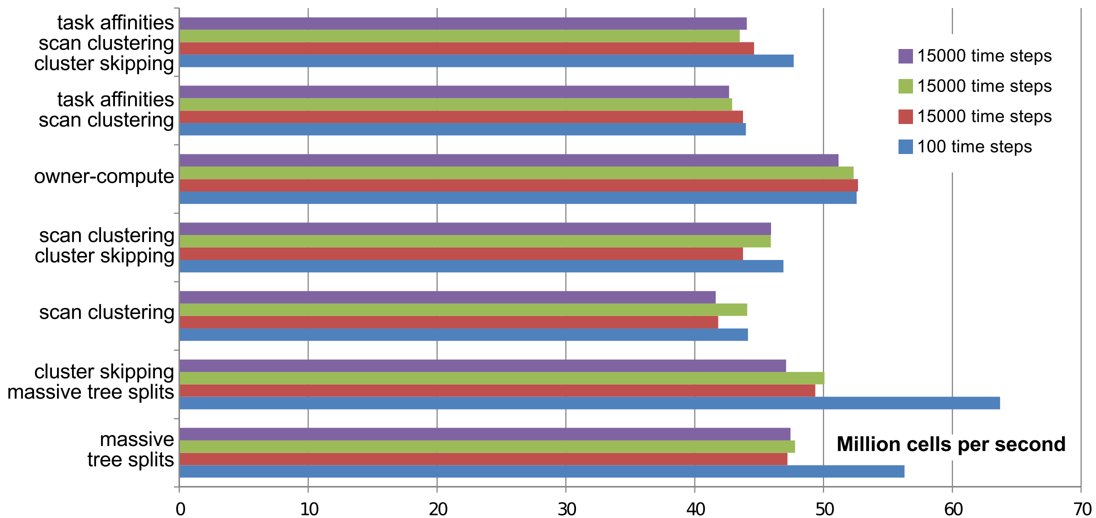


Figure 5.34: Overview of runtimes for simulations with different cluster-generation strategies with and without skipping of adaptive conforming clusters [SWB13a].

5.9 Results: Long-term simulations and optimizations on shared-memory

In the previous sections, we conducted several benchmark experiments based on different cluster generation apporaches, affinities and optimizations. However, these benchmarks did not outline possible effects of long-term simulations. Therefore, we conducted long-term simulation runs with 15000 time steps and surveyed different cluster generation, computation scheduling and optimization strategies on our test platform Intel (see Appendix A.2). Such a long-term simulation run also induces more grid changes compared to the short-term run, and thus, also more cluster generations.

For a better comparison, the benchmark scenario is identical with the one already used in Sections 5.7 and 5.8.2. We present the results of cells processed in average per second in Fig. 5.34. Due to the longer run time of 15000 time steps, we also expect an increased noise in the simulation run time. Therefore, we execute the long-term simulation three times.

With our benchmark results at hand, we first search for the formerly best choice, the cluster skipping with the massive tree splitting, which provides the best short-term simulation performance with 100 time steps. However, for long-term simulation runs, the best performance is not anymore provided by this so far best choice: the *owner-compute scheme pays off* the work stealing mechanism. We account for this by the additional grid changes, causing more cluster generations. With the owner-compute scheme, the clusters are generated in a memory-locality preserving way.

Since the owner-compute scheme also outperforms the affinities using TBB, this scheme is the best choice for long-term simulations for this benchmark.

5.10 Distributed-memory parallelization

Our distributed-memory parallelization approach takes advantage of our cluster-based software design with a replicated data scheme also for shared-memory parallelization. This also accounts for the two major issues for distributed memory systems: (a) efficient block-wise communication via RLE meta information and (b) efficient data migration with the cluster-based software

design. Furthermore, our software design makes the extensions for distributed memory almost transparent for a framework user since only minor additional requirements for the framework user are induced such as the data migration of user-specified data.

Support for distributed memory can then be accomplished with the following extensions to the framework design (Fig. 5.10):

1. **Simulation driver:**
   The simulation driver is responsible for controlling the overall simulation. This includes e.g. the determination of the amount of workload required for load balancing and the global time-step size. Therefore we extend the simulation driver with e.g. min-reduce operations on the time step sizes of each rank to compute the correct global time-step size.

2. **Inter-cluster communication:**
   The data associated to hyperfaces which are shared with clusters of other ranks has to be sent and received in a correct way (Section 5.10.1).

3. **Dynamic cluster generation:**
   The dynamic cluster generation has to be extended to distributed memory to support updating the RLE communication meta information for splits and joins of adjacent clusters (Section 5.10.2).

4. **Cluster-based data migration:**
   Each cluster is extended with instructions for transferring its meta-, stack- and user-specified data to another rank. Also updating the RLE communication meta information has to be considered (Section 5.10.3).

5. **Base triangulation:**
   Not all base triangulations are valid anymore since our communication schemes for distributed memory relies on the SFC order of communication data (Section 5.10.4).

Considering the components relevant from an algorithmic point of view, the distributed memory parallelization is then based on items (2)–(5) which are further discussed in detail.

5.10.1 Intra- and inter-cluster communication

Regarding the cell communication, requirements on (a) intra- and (b) inter-cluster communication are discussed next:

(a) *Intra-cluster communication*:
We can use the property of the data exchange between cells being accomplished via the stack system. Communication via this stack system does not depend on any explicitly stored adjacency information. Therefore, our stack-based communication method is invariant to the memory location[5] and rank. This leads to an *intra-cluster communication not depending on the memory- and rank-location of the cluster*.

(b) *Inter-cluster communication*:
For inter-cluster communication, only a minor modification to support distributed memory is required. The RLE communication meta information about adjacent clusters is *extended by the information on the adjacent rank* with which to exchange the data.

[5] E.g. changed by reallocating the stack system

Regarding the inter-cluster data exchange, we use non-blocking send operations to transfer the replicated data to the rank of the adjacent cluster. Using our RLE, this data transfer can be obviously accomplished *directly block-wise*, e.g. without gathering of smaller chunks of data to be transferred.

We further use MPI send-recv tags to label each communication with the communication data being either stored to the communication buffers in clockwise or counter-clockwise order of the SFC close to the inter-cluster shared hyperfaces. This is required for base triangulations consisting e.g. only of two triangles with periodic boundary conditions. This avoids the communicated data stored for clockwise edge directions being read for edges with counter-clockwise direction and vice versa.

Depending on the method of data exchange on distributed-memory systems, our approach can further demand for a particular *order and tagging*. To receive the data blocks in correct order, the send operations follow the SFC order of the clusters and the receive operations are then executed in reversed SFC order. Also the RLE meta information entries have to be iterated in reversed order to consider the opposite direction of the traversal of the clusters at the other rank (see Section 5.2.5 for information on reversing communicated data). Reversing these receive operations assures the correct order.

5.10.2 Dynamic cluster generation

For load-balancing reasons, it can be advantageous to split a cluster in case that its one-dimensional SFC interval representation can be assigned to several cores.

The cluster generation approach considered in this work is based on the rank-local number of cells only. This makes the cluster generation approach independent of the global load distribution. An extension with a global parallel prefix sum based on the number of cells [HKR+12] to generate cluster in a global-aware manner is not further considered in this work.

After splitting and joining of clusters, we also require reconstruction of the communication meta information to account for possible splits and joins of adjacent clusters. Updating local RLE meta information that is associated to a cluster stored at another rank is not possible anymore with our shared-memory approach, see Sec. 5.5.3. This is due to a missing direct access to the meta information of the adjacent cluster. Therefore, the adjacent cluster is responsible for deriving and sending the required split/join information. The local cluster then receives and uses this information to update the RLE meta information accounting for adjacent split/join operations.

5.10.3 Cluster-based load balancing

With dynamically adaptive grids, this results in load imbalances across several ranks. Data migration is one of the typical approaches to migrate workload to another rank. With our cluster-based data migration, we present a highly efficient method for migrating a set of clusters. This efficiency results from three major components of our software design:

- *Stack- and stream-based system:*
  All simulation DoF are stored on a stream system and the intra-cluster communication via stacks is based on the cluster's structure stack. Hence, this intra-cluster communication is independent to the memory location. Thus we can directly transfer this data in *raw-format*[6] to an adjacent cluster without requirements on updating adjacencies, e.g. by updating pointers or relative indices of adjacent primitives stored in each cell. Since our

[6]assuming the same system, e.g. the same endian format on all compute nodes

stacks only store the pure payload and the grid structure in a bitstream[7], we assume this to be quasi-optimal w.r.t. to the amount of transferred memory.

- *Meta information and user-defined data:*
  Besides the communication meta information, only the remaining cluster-local data has to be migrated separately.

- *RLE update:*
  With all information on adjacent communication stored implicitly using the RLE communication scheme and communication buffer, only the RLE communication meta information has to be updated to yield a consistent communication meta information.

The migration algorithm of our cluster then consists of the following steps:

1. *Destination labeling:*
   We label each cluster with the rank to which it has to be migrated to.

2. *Prospective update of communication meta information:*
   For each RLE meta information entry and in case that this meta information represents a communication with a cluster stored at another rank, *send the destination rank* to the adjacent rank, otherwise use ϵ to represent *no migration*. Then the local RLE meta information about the adjacent cluster is updated in case of a migration of the adjcent cluster to another rank. This assures that each adjacent cluster can update its meta information to the new destination of the migrated cluster.

   For efficiency reasons, we joined this step with updating the RLE information during forwarding the information for dynamic clustering (Section 5.10.2).

3. *Cluster migration:*
   The cluster data (stacks, meta and user information, ...) is then migrated to the adjacent rank, see Section 5.10.3. This migration already includes the updated communication meta information from the previous step.

4. *Synchronize the cluster-local RLE information*:
   We distinguish between received and send clusters:

   After receiving all clusters at a rank, the pointers to the adjacent clusters which are stored on the same rank have to be recovered, e.g. for accessing meta communication data. By also keeping the cluster-unique id in each RLE meta information entry, we can find the clusters efficiently with the cluster set based on a binary tree structure. The pointers from the adjacent cluster to the currently processed one can then be directly corrected by writing to the RLE meta information of the adjacent cluster.

   We emphasize that these writing operations to adjacent clusters violate our *push and pull* concept (*no writing access to adjacent clusters*). However, we consider this to be the most efficient way to update the communication meta information for clusters stored on the same memory context.

   For clusters migrated away from the considered rank, the RLE meta information of the adjacent clusters on the same considered rank have to be updated. These entires are set to the new rank to which the cluster was migrated to. This new rank is available with the destination rank label, see the first item (1) of this list.

[7] In our implementation, we store each bit in one byte to avoid bit shift and and operations during the grid traversals.

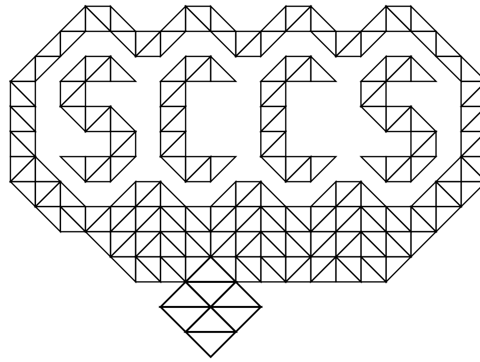


Figure 5.35: Possible domain triangulation for simulations with a shared-memory parallelization, forbidden for distributed-memory runs

The information where to send the clusters is derived based on the local and MPI *parallel prefix sum* on the number of cells in each cluster. Then, each rank can determine the current range of cells within the global number of cells of the entire simulation. The clusters are tagged with a rank which leads to improved load balancing. Such a load balancing can be e.g. improved by sending the clusters only to the *next or previous rank*. This is advantageous, since only the next and previous rank have to test for a cluster migration from an adjacent node. For severe load-imbalances, such a migration is not feasible anymore since only an iterative data migration can solve such a load imbalance. Hence, we can tag the cluster to be directly sent to the rank which should own the cluster for improved load balancing. Here, the rank is computed by

$$rank := \left\lfloor \frac{R_i + \frac{W_i}{2}}{W_{avg}} \right\rfloor$$

with R_i the *global start id* of the first cell in the cluster, W_i the workload in the cluster and W_{avg} the average number of cells per rank (cells). This can be interpreted as extension of the range information presented in Sec. 5.6 to distributed-memory systems. Then, an all-to-all communication is used to inform ranks to receive one or more clusters from a rank. Further details on such a load balancing can be e.g. found in [HKR+12]. Finally, we like to emphasize, that the data migration of clusters still has to conserve the SFC order of the clusters on all MPI ranks.

Since our cluster migration is based on setting the destination rank followed by the cluster migration, this allows implementing a generic load-balancing interface [DHB+00] and, thus, also different well-studied load-balancing and migration strategies, see e.g. [ZK05, Cyb89, Hor93].

5.10.4 Distributed base triangulation

With our communication scheme relying on the correct SFC order of the underlying grid cells, we are not allowed anymore to generate domain triangulations such as the one given in Fig. 5.35. In general, traversals with an initial base triangulation and an SFC traversal close to shared edges in the same direction (see e.g. [NCT09] for a cubed sphere SFC traversal with the Hilbert SFC) result in inconsistent communication schemes.

A different view of a correct communication order for base triangulations on distributed-memory systems is the possibility of embedding our grid into a (regularly) recursively structured grid based on the Sierpiński SFC. This consequently leads to the space-forrest assembled by the nodes on a particular level of a regularly refined spacetree. Examples for invalid and valid

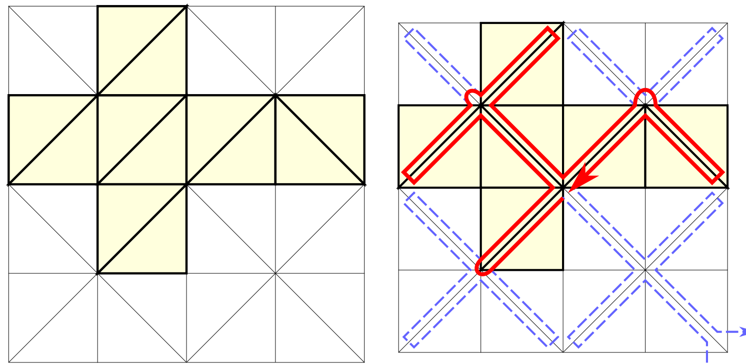


Figure 5.36: Different base triangulations. Left image: Invalid base triangulation with the middle two triangles not embeddable to the regularly refined grid. Right image: valid base triangulation.

base triangulations with the corresponding embedding into a regular refinement are given in Fig. 5.36.

The initial association of base triangles to ranks is accomplished by aiming for balanced workloads. Embedding our base triangles into an SFC generated grid, we can again use the 1D representation to assign clusters to ranks. Each rank then initializes the clusters assigned to it. Regarding the cluster tree, each rank removes all leaf nodes not assigned to the current rank and also the inner nodes which do not have any further child nodes.

5.10.5 Similarities with parallelization of block-adaptive grids

We continue by highlighting *similarities between clustering and dynamically adaptive block-adaptive grids*.

- *Communication meta information*:
  With our RLE communication meta information, the quantity and the location of the data to be communicated is directly given. For block-adaptive grids, this quantity is also directly known by the ranges in each dimension of the underlying grid in each block.

- *Block-wise load-balancing*:
  In our case, we accomplish load-balancing by migrating the cluster-associated simulation data and by updating the communication meta information. For migration of blocks, also the block-associated simulation data is migrated, followed by the meta information which is typically stored for each block.

Hence, a cluster-based parallelization approach allows similar optimizations such as hybrid parallelizations, latency hiding, etc. with some of them being also further researched in this work.

5.11 Hybrid parallelization

The number of cores on cache-coherent memory domains considerably increased during the last decade. Shared-memory systems with several threads per CPU are nowadays omnipresent and with Intel's XeonPhi, even more than 100 threads have to be programmed in a shared-memory environment in an efficient way. Such a hybrid parallelization yields several advantages; some of them are:

- Sampling of datasets:
  Storage of either the entire or only a part of the bathymetry data for Tsunami simulations for each single-threaded MPI rank could lead to severe memory consumptions. To give a concrete example, we consider the ocean bathymetry datasets from General Bathymetric Chart of the Oceans (GEBCO) [IOC] with the entire dataset of size of less than 2GB. This alrady exceeds the sizes of memory typically available per core. E.g. on the current generation of the SuperMUC, 16 cores share 32GB memory [EHB+13]. Thus with 2GB memory consumption per thread to store the entire GEBCO datasets, this already occupies all available memory.

  With a hybrid parallelization, the datasets can be directly shared among several threads. This allows storing the dataset only once in each program context, resulting in more memory available for simulation data. We used this hybrid parallelization for the Tsunami benchmarks in Section 6.3 with the entire bathymetry data loaded into each rank's memory.

- Reduced data migration:
  Using single-threaded MPI can result in severe communication overheads in case of several clusters being migrated at the same time. This can lead to a memory transfer similar to a streaming benchmark due to migrated stacks and streams compactly stored in memory and transferred block-wise. Using a hybrid parallelization, some data migration can be avoided. In case of the clusters required to be migrated to a thread (considered to be a rank for single-threaded MPI implementation) which executes tasks in the same memory space in which the cluster is stored at, the cluster can be directly processed by the other thread without requiring any cluster migration process.

We discuss two alternative approaches for the inter-cluster communication presented in Section 5.10.1.

- The first approach can be used to overcome a sequentialization of the iteration over the clusters in reversed order to receive the data on the shared interfaces. An extension of the send/recv tag with *unique communication tags associating two clusters* can be used, e.g. involving both cluster unique IDs. This unique communication tags also assure a unique message tag and, thus, no particular order has to be considered to read the message. However, the MPI interface standard 3.0 [For12] only assures a tag range from 0 to 32767. This range can be exceeded by our cluster-based approach being based on tree-splits: First, we require at least one bit for distinguishing between the left and right communication stack. This would restrict our remaining tag range to ≈ 16383. Second, a massive splitting can lead to by far more clusters than the available tag range, possibly *violating the requirements given by the MPI implementation.*

- The second considered alternative is based on the *thread ids* instead of the cluster ids. Since the number of threads is limited, the utilization of a valid range of tags can be assured. A set of clusters can then be deterministically assigned to each thread (e.g. by using the affinity ids) and each thread processes the set of cluster in parallel. This requires an extension of the RLE communication meta information by also adding a thread id next to the MPI rank (see [Mav02] for a similar concept).

Since our results already yield sufficient efficiency for hybrid parallelization to simulate Tsunamis on distributed-memory systems, we did not implement these alternatives.

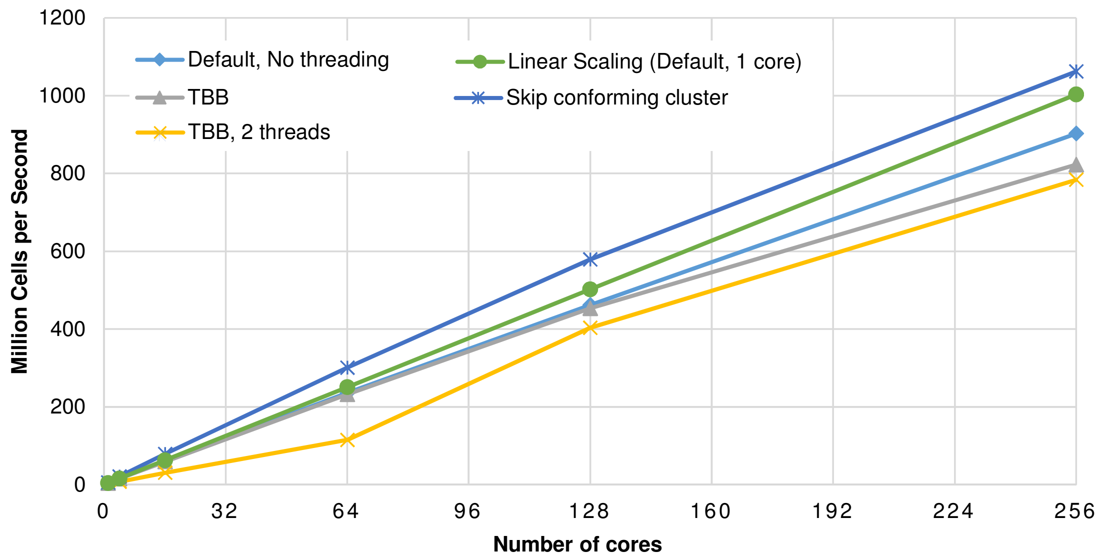


Figure 5.37: Strong scalability on MAC cluster for different parallelization models and workload. The default method is based on the massive-splitting cluster generation.

5.12 Results: Distributed-memory parallelization

Based on the distributed-memory extensions, we executed benchmarks on a small-scale system with up to 256 cores and large-scale system with thousands of cores. Compared to higher-order spatial discretization schemes, a finite-volume discretization leads to larger time steps due to a larger CFL condition and thus more requirements on load-balancing. Therefore, we used a finite-volume simulation with the SWE and the Rusanov flux solver. The scenario was set up with a radial dam break scenario on a quad-shaped domain. All computations are done in single precision.

5.12.1 Small-scale distributed-memory scalability studies

We conducted small-scale scalability studies on the MAC cluster test platform Intel (see Appendix A.2.3). The initial refinement depth was set to 22 with 8 additional refinement levels for dynamically adaptive mesh refinement. With this fixed problem size, we compute a strong scalability problem.

The dynamic cluster generation is controlled by a massive-splitting method with a split threshold of 4096. Cluster-based load balancing, grid adaptivity traversals and cluster generations are executed between each simulation time step. The cell throughput is given in Fig. 5.37 on up to 256 cores for different parallelization models and optimization activated. The linear scalability is based on a single-core execution of the default massive-splitting cluster generation. Applying our optimization with the conforming-cluster skipping algorithm (see Sec. 5.8.2 and “Skip conforming cluster” in the plot), we generate robust improvements with this algorithmic optimization also on distributed-memory systems.

The TBB threading has a clear overhead compared to our non-threaded version. We account for this by the relatively small computational amount in each cluster and by the additional requirement of a thread-safe MPI libary. However, the drop in performance comparing the single-threaded and the double-threaded TBB benchmark is below 5% for the strong scalability on 256 cores.

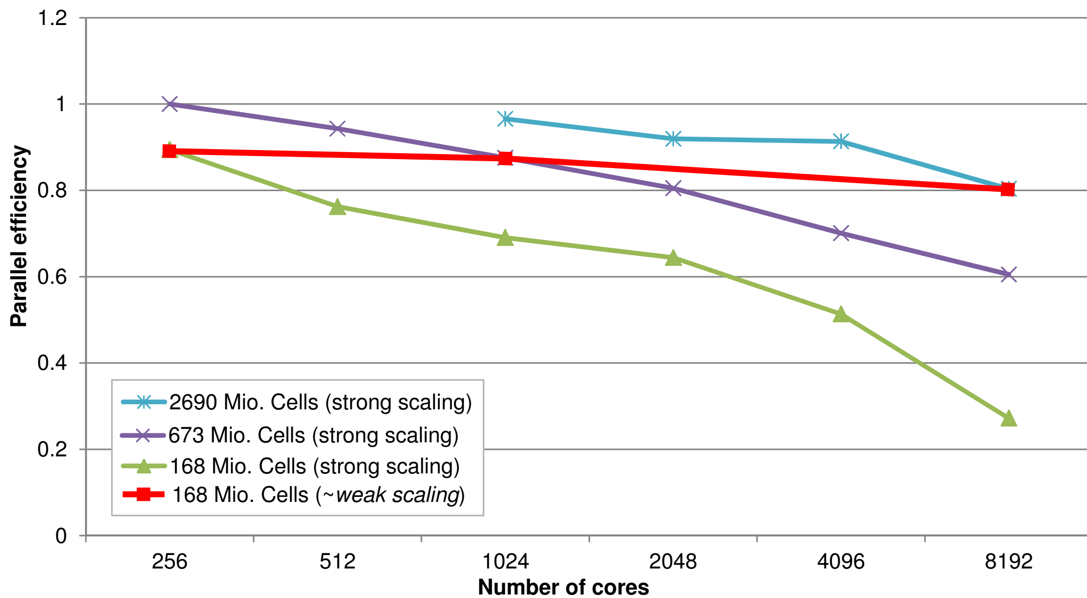


Figure 5.38: Strong and weak scalability graphs on SuperMUC for different workloads for a shallow water simulation with finite volumes.

5.12.2 Large-scale distributed memory strong-scalability studies

We conducted the large-scale scalability studies with several strong scaling benchmarks on the SuperMUC.

For cluster generation, we used a massive-splitting with a threshold size of 32768. We tested different initial refinement depths $d = \{24, 26, 28\}$ with up to $a = 8$ additional dynamically adaptive refinement depths and executed 100 time steps with each simulation. The measurement is started after the last adaptivity traversals and until no further cluster migrations are requested.

In this benchmark, we only migrated clusters to the next or previous ranks based on the parallel MPI prefix sum over the workload, see [HKR+12] for further information. The results for different initial problem sizes are given in Fig. 5.38. The baseline is given with the simulation on 256 cores, an initial refinement depth of 26 and up to 8 allowed refinement levels. This baseline has a throughput of 1122.76 mio. cells per second.

The simulation with the smaller problem size resulted in worse scalability for a high number of cores. We account for that by considering the cluster threshold size of 32768. This leads to $\frac{168000000}{32768 \cdot 8192} \approx 0.626$ clusters in average per rank and, hence, accounts for the dropdown in scalability due to severe load imbalances. Smaller cluster sizes or using a range-based splitting (see Sec. 5.6.2) are expected to improve the scalability and is part of our ongoing research.

Executing the simulation with 2690 million cells was not possible on less cores than 1024 cores due to memory requirements exceeding the physically available memory per compute node. We expect that this can be overcomed by reducing the additional paddings for the prospective stack allocations (see Sec. 4.10.6). However, executing problems of this size on such a relatively small number of cores would never allow computing entire simulation runs due to the high workload and thus very long computation time per core. Therefore we did not further investigate such workloads per core.

Considering weak scaling, the efficiency is still above 90% for 8192 cores with the baseline at 256 cores.

5.13 Summary and Outlook

We summarize our major developments and contributions, followed by an outlook to future work:

- *RLE meta communication information:*
  We used properties of the stack-based simulation to develop a parallelization based on a run-length encoding which leads to the following advantages. Our communication meta information is stored for inter-cluster shared hyperfaces only and is run-length encoded. This RLE edge meta information is updated implicitly, based on the adaptivity markers. We can also encode the vertex meta information efficiently with our RLE using a zero-run-length encoding, resulting in a reduced memory consumption. The data access on shared- and distributed-memory systems can then be accomplished block-wise. Considering the DG simulations, the edge- and vertex-based communication allows an implementation of possibly required flux limiters for edges and vertices.

- *Parallelization:*
  Regarding the *parallelization*, we developed a cluster-based software design. Here, one or more independent chunks of the simulation grid reside in the same memory context. These chunks can be traversed in arbitrary order and the communication is accomplished based on run-length encoded meta information.

- *Dynamic cluster generation:*
  With the dynamically changing grids leading to a different number of grid cells in each cluster, we derive the meta information after tree-splits and -joins in an efficient way. Our dynamic cluster generation implicitly derives the new meta information based on the number of entries stored on the stacks. Two different ways of dynamic cluster generation have been investigated. The range-based clustering generates the clusters aiming for load balancing. However, our tested scenario showed overheads compared to a threshold-based cluster generation, leading to the best results.

- *Parallelization models:*
  Our software and communication design results to direct applicability of different parallelization models.

  On shared-memory systems, we evaluated TBB and OpenMP on a 40-core NUMA system yielding high scalability for short- and long-term simulations with NUMA-aware scheduling. Here, the owner-compute scheme with a threshold-based splitting showed the best results.

  On distributed-memory systems, our software concept leads to an efficient data migration with the clustering concept. Our benchmarks show a weak scalability of over 80% on more than 8000 cores with the baseline at 256 cores.

- *Skipping of conforming cluster traversals:*
  We used the adaptivity automaton with the skipping of clusters with an already conforming grid state. This results in robust performance improvements, also compensating additionally required conforming grid traversals due to the domain decomposition.

- *Data migration:*
  After the dynamic clustering phase, we can migrate clusters efficiently to adjacent MPI ranks. This requires migrating the cluster's raw data which is stored compactly on the

stacks. Since all connectivity information is stored implicitly with the structure bit stream and since the simulation stacks only contain the pure payload of the simulation (e.g. only the DoF values), we assume the amount of transferred memory to be quasi-optimal. Furthermore, only pre- and postprocessing of the RLE meta information is required which is also quasi-optimal due to the run-length encoding.

We envision the following possible further developments:

- *SFC cuts*:
  Our dynamic cluster generation is based on spacetree splits. This has drawbacks for our load-balanced-aware splitting (see Sec. 5.6.2) with two of them mentioned here: First of all, we can only iteratively split the spacetree to improve the load balancing. Second, in case of two adjacent clusters processed by the same thread (e.g. using owner-compute), their shared hyperfaces still require a reduction operation despite that they are processed by the same thread. We expect that a cluster generation based on SFC cuts can also solve the aforementioned issues.

- *Loosening the conforming grid requirement*:
  For a conforming grid generation, we forward the adaptivity requirements with the stack-based edge communication. This (a) requires multiple consistency traversals, (b) can lead to additional traversals due to the domain decomposition and (c) involves additional overheads for reduction operations on conforming grid states.

  Regarding issue (a), we propose to generate an indexing structure to avoid any hanging nodes inside each cluster in a single traversal. This indexing structure can be generated with a single traversal by forwarding the current cell index with the edge communication. The result is a directed-acyclic graph with its edges directed towards the adjacent cells with a higher SFC-enumerated index. The root node is the latest traversed cell and its last node is the first cell. Based on this directed graph, we can reconstruct the indexing to all cells, yielding an bidirected graph. This indexing makes the direct forwarding of hanging nodes markers within each cluster possible, hence not requiring additional traversals.

  Considering issue (b), the synchronization barriers can be circumvented by allowing hanging nodes on the cluster boundaries. We can represent hanging nodes by splitting an RLE entry with appropriate handling required.

  No global synchronizations, e.g. reductions on adaptivity conformity states, are required due to solving (a) and (b).

- We assume, that the RLE meta communication information can also be used for Cartesian and hexagonal grids and that it can be also applied to higher-dimensional grids, e.g. grids generated by the Peano SFC.

6 Application scenarios

The previous chapters focused on the algorithms developed for the parallelization of simulations on dynamically adaptive meshes. However, such an algorithmic description does not assure the applicability of the presented algorithms in real scenarios. Hence, we show possible application scenarios and discuss the benefits of our dynamically adaptive mesh refinement and the clustering in the upcoming sections.

6.1 Prerequisites

Before running benchmarks with the shallow water equations, we need (a) solvers with the capability of running accurate simulations, (b) an error norm for objective statements on the accuracy, (c) an error indicator to trigger refinement and coarsening requests, and (d) an extension of the refinement and coarsening operations due to bathymetry. These issues are briefly addressed in the upcoming sections.

6.1.1 GeoClaw solver

For a correct handling of wave propagations, we decided to use the augmented Riemann solver from the GeoClaw package [Geo08,BGLM11]. This solver is well-tested in the context of shallow water and Tsunami simulations [MGLT] and operates on the cell averaged conserved quantities (h, hu, hv, b), respectively the *distance of the water surface to the bathymetry*, *the momentum in x-direction*, *the momentum in y-direction* and *the bathymetry relative to the horizon*. Using the augmented Riemann solver, the flux computations with the adjacent cells are replaced by solvers computing so-called net updates. These solvers compute the net flux from one cell to another one.

Then the conserved quantities are directly improved based on the computed net updates for h, hu and hv and we use the b variable to store the wave speed for the accurate computation of the time-step size. For usability issues, we reuse the reference space (see Sec. 2.2) and the same framework interfaces as for the higher-order simulation.

6.1.2 Error norm

In the following benchmarks, we require an error norm for an objective comparison of simulations conducted with different parameters. We followed the suggestion in [RFLS06] to use the L1 error norm on the surface heights over time for shallow water simulations. However, instead of using the relative error, we decided to use the absolute error to avoid any influence in the computed error induced by our adaptively changing bathymetry. Let the interval to compute the error be given with $t \in [T_s, T_e]$. The L1 norm is then applied to the absolute difference of

our computed solution $h(t)$ to the baseline $b(t)$:

$$E(b,h) := \frac{1}{T_e - T_s} \int_{T_s}^{T_e} (||b(t) - h(t)||) \,.$$

In its discrete form based on N sampling points, where the first and last one are related to T_s and T_e, respectively, this yields

$$E^d(b,h) := \frac{1}{N} \sum_{i=0}^{N-1} (||b(T_s + i\Delta t) - h(T_s + i\Delta t)||)$$

with

$$\Delta t = \frac{T_e - T_d}{N - 1}.$$

With a given data set at discrete sampling points for comparison with our baseline, we use spline interpolation and compute equidistantly distributed sampling points which are then used to compute the error with E^d. We chose the number of equidistant sampling points to a robust value which does not lead to significant changes in the computed error norm (typically larger than 20000).

6.1.3 Error indicator

With our main focus on computing the solution within given error bounds as fast as possible, only grid cells with a particular contribution (feature rich areas) to the final result should be refined. We implemented two different adaptivity criteria, each one strictly depending on the per-cell stored data to avoid additional edge-communication data.

- *Horizontal deviation:*
  This adaptivity criterion is based on the absolute value of the water surface deviation from the horizon, yielding
  $$I_{horizon} := |b + h|$$
  with b the bathymetry which is negative for the sea ground and h the water surface distance to the bathymetry.

  This error indicator does not consider the size of the cell and is therefore *unaware of the refinement depth*. This would result in possible refinement operations up to the maximum of the allowed refinement depth.

- *Net-updates:*
  With net-update-based solvers, we can use an error indicator based on the net-update component Δh which represents the amount of fluid streamed over an edge with a unit length and a unit time-step size. We use this to determine a steady state with respect to the water surface elevation. In case that as much water is flowing into the cell as it is flowing out (i.e. a balanced state), the amount of fluid in a cell is not modified. This yields the following adaptivity criteria for the per edge communicated net-update components Δh_i:
  $$I_{\text{net-update}} := \sum_{i=1,2,3} \Delta h_i |e_i|$$
  with $|e_i|$ the length of the triangle edge i.

  This error indicator is based on the *edge length* and is therefore *sensitive to the refinement depth.*

Then, each cell requests a refinement operation in each time step in case of $I > \alpha_{\text{refine}}$ and the cell agrees to coarsening in case of $I < \alpha_{\text{coarsen}}$.

For the benchmark studies in the following chapter, only the net-update based simulation was successfully applied to improve the computational time with dynamically adaptive mesh refinement for the considered simulations. Therefore, we restrict the presentation of our results to these indicators. We like to emphasize, that this does not induce, that the height-based adaptivity criteria cannot be used for efficient dynamic adaptive grids, but that our considered parameters did not yield satisfying results.

Alternative error indicators are e.g. based on the derivative of the surface height [RFLS06] and require additional data transfers via edges. With the focus of our studies on justification of the dynamically adaptive grids and since we will see that our results already justify the dynamical adaptivity, we did not further investigate alternative error indicators.

A final remark is given on the net-update-based error indicator: we expect, that this indicator cannot be applied to flooding and drying scenarios since the water surface height close to the shoreline tends towards zero. Hence our error indicator does not lead to refining the grid in this area. Since the sampling points in our considered scenario are in a simulation area with a relatively deep water compared with the average depth, a modification of the net-update-based solver for flooding and drying scenarios, e.g. by dividing it with the average depth of the cell, was therefore not required.

6.1.4 Refinement and coarsening with bathymetry

In contrast to the previous shallow-water test scenarios, the scenarios from the following section have a non-constant bathymetry value. Due to refinement and coarsening operations with our dynamically adaptive grid, we require an extension for the refinement and coarsening operations for the conserved quantities including bathymetry.

We discuss the conservation schemes based on the conserved quantities $U := (h, hu, hv, b)$ and denote the conserved quantities in the parent cell with U_p and for both children with $U_{1/2}$. We determine the bathymetry data $b_{1/2}$ by sampling the bathymetry data set at the cell's center of mass.

We further assume that the water surface should stay on the same horizontal level after a refinement operation to avoid spurious gravitation-induced waves. We can assure this by initializing the water height for both cells with

$$h_1 := \max(0, (b_p + h_p) - b_1) \qquad h_2 := \max(0, (b_p + h_p) - b_2)$$

using the *max* operator to avoid non-physical negative mass.

With the velocity of the moving wave being one of the most important features for wave-propagation dominated schemes, we used a *velocity conserving scheme*: We compute the velocity of the parent cell with $u_p := \frac{(hu)_p}{h_p}$, $v_p := \frac{(hv)_p}{h_p}$ and initialize the momentum of each child cell with $(hu)_{1/2} := h_{1/2} u_p$, $(hv)_{1/2} := h_{1/2} v_p$, respectively. The coarsening operation uses the averaged velocity of both joined cells to reconstruct the conserved quantities in the parent cell.

Using the velocity conservation yielded the best results for our simulations executed in the ongoing sections. Alternative approaches such as conserving the momentum resulted in less stable simulations and were thus not further considered.

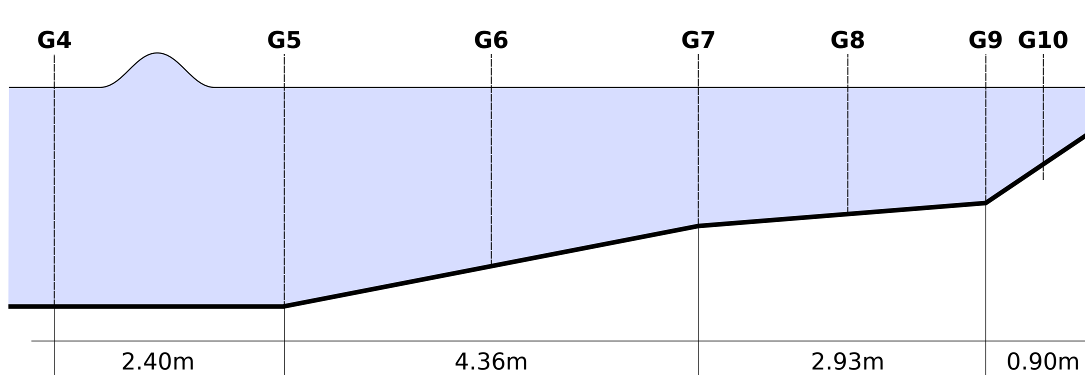


Figure 6.1: Scenario sketch for the solitary wave on composite beach benchmark.

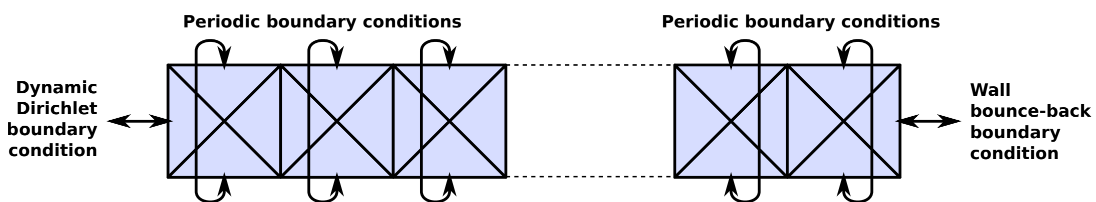


Figure 6.2: Base triangulation and boundary conditions for a solitary wave on composite beach.

6.2 Analytic benchmark: solitary wave on composite beach

We selected the Solitary Wave On Composite Beach (SWOCB) benchmark of the NOAA[1] benchmarks which is suggested for Tsunami model validation and verification[2] to show the correctness and applicability of the run-time adaptivity of the developed framework in the context of an analytic benchmark. Despite its complex simulation scenario with non-constant bathymetry, there is an analytical solution available [UTK98] which makes this benchmark very interesting. *Acknowledgement:* This benchmark was implemented in collaboration with Alexander Breuer.

6.2.1 Scenario description

We use scenario A of the benchmark, with a sketch of its scenario given in Fig. 6.1. This benchmark is based on a wave which is first moving over a bathymetry with a constant depth of 0.218 and then entering an area with three non-constant bathymetry segments, with a slope of $\frac{1}{53}$, $\frac{1}{150}$, and $\frac{1}{13}$ respectively.

Three different boundary conditions are required (see Fig. 6.2). We set the *wall boundary condition* on the right side of Fig. 6.1 to bounce back boundary conditions (see Sec. 2.11.3). The boundaries on the scenario sides (top and bottom side in Fig. 6.2) are set to be *periodic conditions*, thus simulating an infinitely wide scenario. For the *input boundary condition* on the left side with the wave moving in, we do not model the initial wave form and its momentum, but use the boundary conditions of the analytical solution from the GeoClaw group provided via the git repository[3].

[1] `nctr.pmel.noaa.gov`
[2] `http://nctr.pmel.noaa.gov/benchmark/Solitary_wave/`
[3] `https://github.com/rjleveque/nthmp-benchmark-problems/`

With our framework, we can setup such a geometry by assembling the domain with a quadrilateral strip where each one is assembled by four base triangles, see Fig. 6.2. Due to our requirement of uniquely shared hyperfaces (see Theorem 5.2.2), we are not allowed to assemble a quadrilateral by only two triangles, since this would lead to the same adjacent cluster in the RLE meta communication information for the left and right communication stacks. Therefore, we used four triangles to assemble a quadrilateral, circumventing this issue for edge communication. We initialize the domain with a strip of 128 quadrilaterals, hence resulting in 2^9 initial triangles for refinement depth 0. Then, for initial refinement depth d, the domain is initialized with $2^{(9+d)}$ triangle cells.

6.2.2 Gauge plots and errors

We first analyze the approximation behavior with different regular refinement depths without the dynamic adaptivity enabled. The plots for water surface height at different water gauge stations G5 - G10 are given in Fig. 6.3. We can see

- a good matching with the analytical solution and
- a convergence behavior to the analytic solution.

A more detailed analysis of the convergence error based on the L1 norm with the analytic solution (see Sec. 6.1.2) is given in Fig. 6.4 for all water gauge stations. This is based on the L1 error norm computed over the interval $[270, 295]$ and is decreasing for higher resolutions.

6.2.3 Dynamic adaptivity

Next, we select water gauge G8 for testing the possibilities of dynamic adaptivity based on the L1 error norm computed for this gauge station. We executed several parameter studies with the initial refinement depths $d \in \{0, 2, 4, 6, 8\}$ and additional adaptive levels $a \in \{0, 2, 4, 6, 8\}$ with the constraint $d + a < 8$. Hence, we do not allow any spacetree depths exceeding 8.

A good justification for dynamic adaptivity is to show improved accuracy results with less cells involved in the computation. Following this idea, we executed the benchmark on a regular grid with $d = 6$, yielding $2^{9+6} = 32768$ grid cells, computed the error norm (6.1.2) resulting in an error of $3.80e{-}5$ and use this as a *baseline* for comparison with simulations on dynamically adaptive grids.

We execute the benchmark studies within the set of allowed d and a parameters described above and compare the results using the net-update-based error indicators (see Sec. 6.1.3). The refinement adaptivity parameter was chosen as $r := 10^{-n}$ with $n \in \mathbb{N}$ and the coarsening paramter in a very conservative manner with $c := \frac{r}{10}$. For a better overview, we only focus on simulations yielding improved L1 error norm results compared to our baseline. The cell distribution over time for these simulations is plotted in Fig. 6.5.

In the next step, we have a closer look on the parameter studies which do not only yield improved results, according to the L1 error norm, but also require less cells in average. We use $d = [initialrefinementdepth]/[dynamicadaptiverefinementlevels]$ to abbreviate the adaptivity parameters. A plot for the corresponding cell distributions is given in Fig. 6.6 and detailed information is presented in the next table:

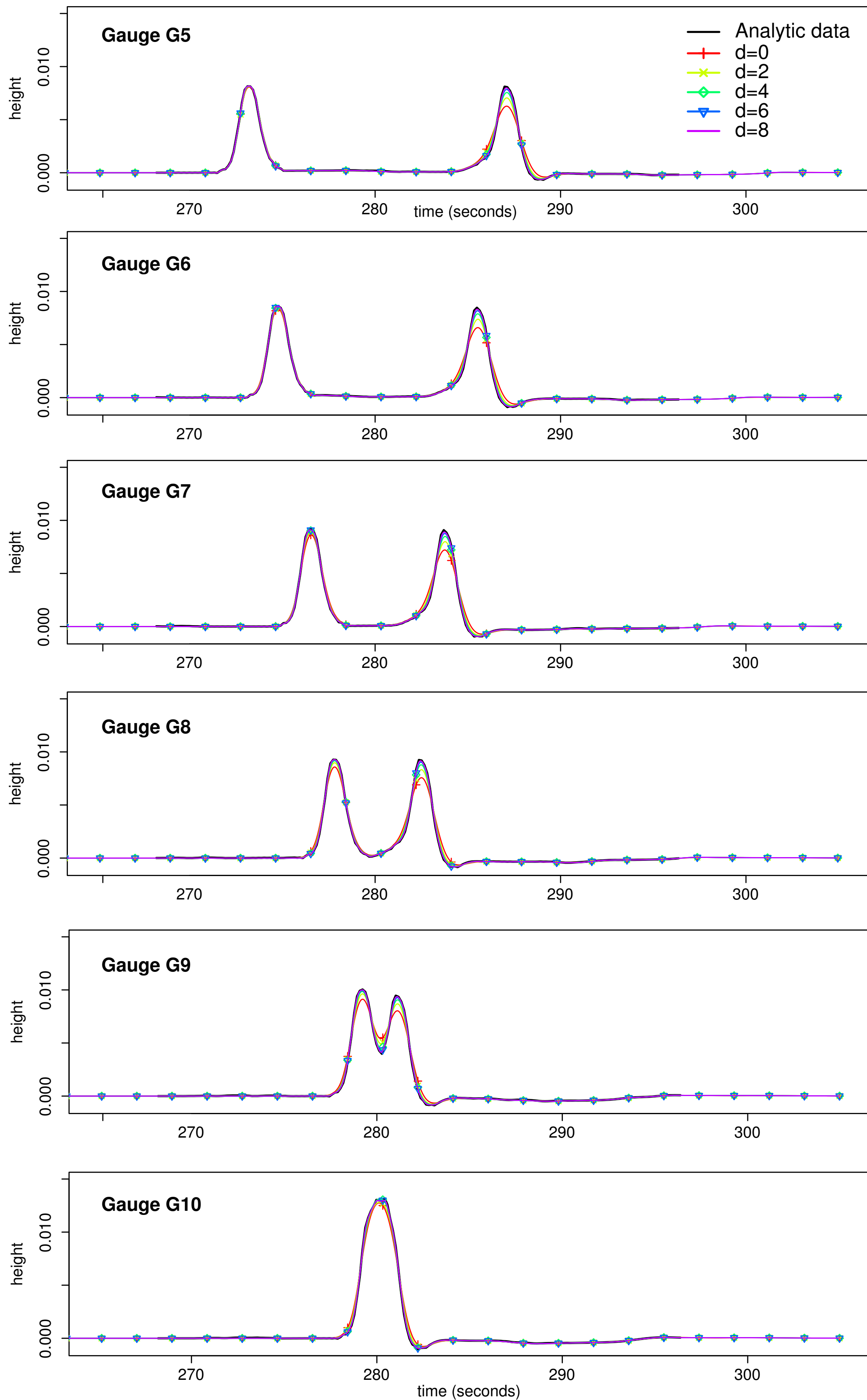


Figure 6.3: Analytic and computed solution for the surface elevation at the water gauge stations.

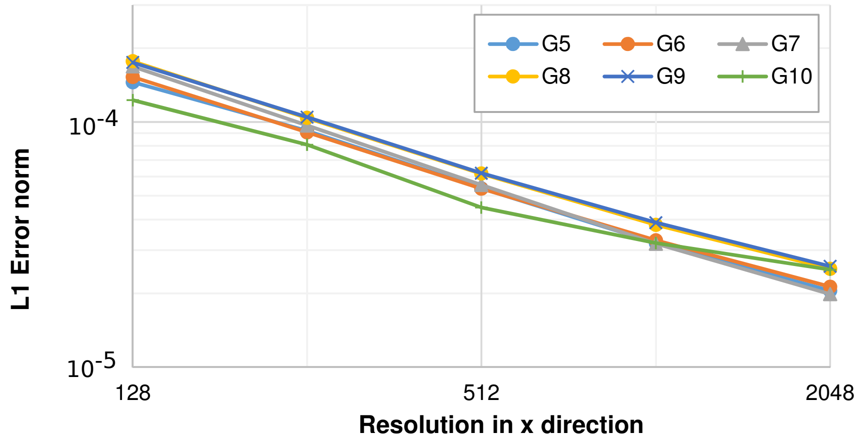


Figure 6.4: Error computed with the L1 error norm showing difference in analytic and computed solution for the surface elevation at different water gauge stations. The resolution is given for the edges on one of the long quad strips boundary. Both axes are in log scaling.

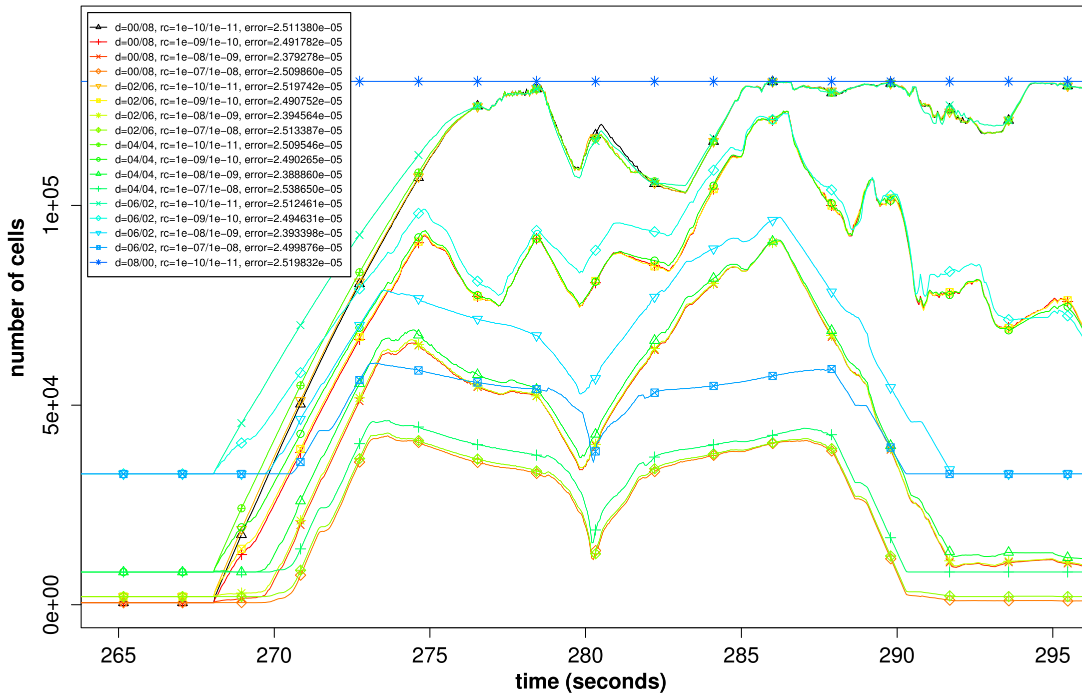


Figure 6.5: Cell distributions over time for solitary wave on composite beach benchmark scenario. The elements in the legend are encoded with "d=[initial refinement depth]/[dynamic adaptive refinement levels], rc=[refine threshold]/[coarsen threshold], error=[L1 error to baseline]".

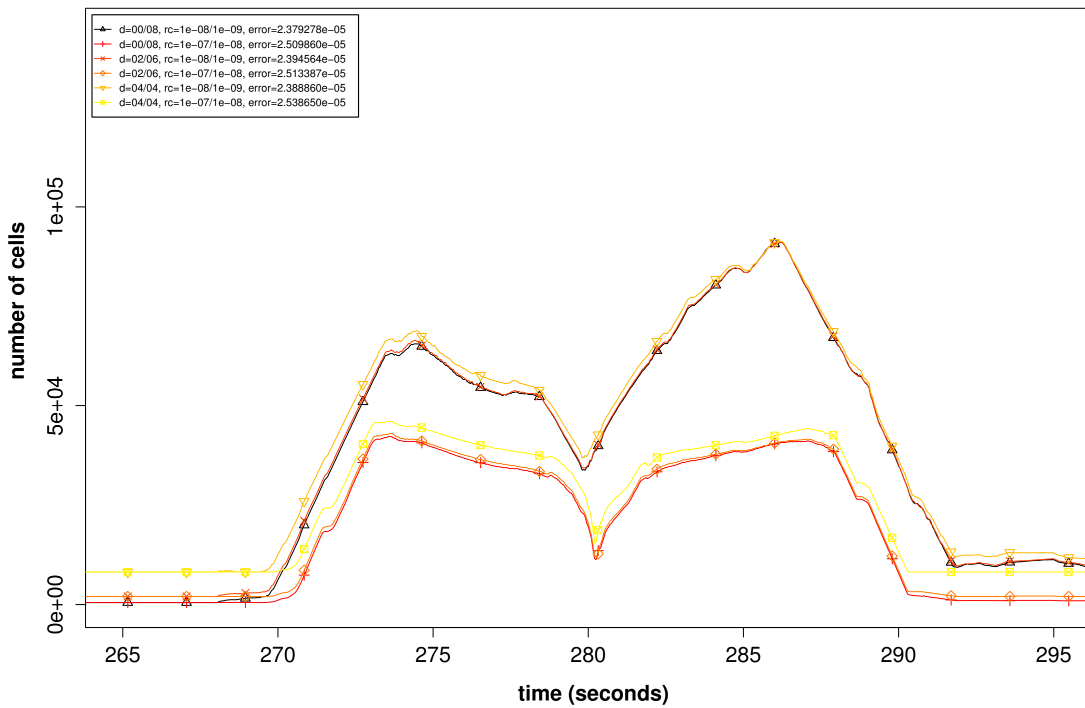


Figure 6.6: Cell distributions over time for a simulation of a solitary wave on composite beach benchmark on dynamically adaptive grids which **require less cells and yield improved results**. The elements in the legend are encoded with "d=[initial refinement depth]/[dynamic adaptive refinement levels], rc=[refine threshold]/[coarsen threshold], error=[L1 error to baseline]".

Benchmark parameter	error	avg. cells	time steps	saved cells per time step
d=0/8, r=0.0001/0.00001	2.38e-5	23556.13	38839	28.1%
d=0/8, r=0.001/0.0001	**2.51e-5**	**11795.13**	**30009**	**64.0%**
d=2/6, r=0.0001/0.00001	2.39e-5	24326.79	40303	25.8%
d=2/6, r=0.001/0.0001	2.51e-5	12924.97	31354	**60.6%**
d=4/4, r=0.0001/0.00001	2.39e-5	28169.67	43500	14.0%
d=4/4, r=0.001/0.0001	2.54e-5	18075.68	36732	**44.8%**
d=6/0 (baseline)	**3.80e-5**	**32768.00**	**37553**	**0.0%**
d=8/0 (baseline 2)	2.52e-5	131072.00	75105	-300.0%

We can see, that for the "d=0/8, r=0.001/0.0001" parameter settings this yields an improvement of $\frac{2^{15}-11795.13}{2^{15}} \approx 64\%$ cells used in average per time step and on the other hand, we get more accurate results. Also considering the total amount of cells involved in the computations, the benefit is increased to $\frac{37553 \cdot 2^{15}-30009 \cdot 11795.13}{37553 \cdot 2^{15}} \approx 71.2\%$ for the considered parameters.

With a domain regularly resolved with a refinement depth $d = 8$, the computed error is $2.52e{-}5$ after 75105 time steps. We compare this to the parameter study "d=0/8, r=0.001/0.0001" which yielded the best result for the same order of magnitude . This leads to $\frac{75105 \cdot 2^{17}-30009 \cdot 11795.13}{75105 \cdot 2^{17}} \approx 96.4\%$ less cells involved in the computations.

These benefits are based on conservatively chosen adapvitiy parameters and on simulations on a relatively small domain. Hence, we expect more improvements by further parameter studies and larger domains. In the next chapter, we evaluate the potential of the dynamic adaptivity on a realistic Tsunami simulation.

6.3 Field benchmark: Tohoku Tsunami simulation

With the simulation of the Tohoku Tsunami of 11 March 2011, a realistic benchmark is given to show the potential of the developed algorithms for dynamically adaptive mesh refinement. Here, we assume that the information on the water surface displacement information due to earthquakes is available a short time after the earthquake.

Acknowledgements for the Tsunami simulation

The first dynamically adaptive Tsunami parameter studies with our cluster-based parallelization approach were simulated in the beginning of 2012. These simulations were developed in collaboration with Alexander Breuer who contributed, among others, the required C++-interfaces to the GeoClaw Riemann solver, the bathymetry and displacement datasets. Also Sebastian Rettenberger contributed his development ASAGI [Ret12] to this studies to access the bathymetry datasets.

However, this was not used anymore for the studies in this thesis due to reasons discussed in the following section. We also like to thank the Clawpack- and Tsunami-research groups for providing their software and the scripts as Open Source and a very good documentation to reproduce their results.

Bathymetry and multi-resolution sampling

Due to dynamic adaptivity, the bathymetry data has to be sampled during run time.

The bathymetry datasets we used in this work are based on the GebCo[4] dataset with its highest resolution requiring 2GB of memory. With the different cell resolutions, sampling the

[4] `http://www.gebco.net/`

bathymetry dataset only on the finest level would lead to aliasing effects, and special care has to be taken with interpolation [PHP02]. Therefore, we additionally preprocessed the bathymetry computing multi-resolution bathymetry datasets. Using multiple resolutions, the coarser levels then require $\frac{1}{4}$ of memory compared to the next higher-resolved level, thus the overall memory requirement is less than 3GB to store all levels. This memory requirement is considerably lower than the total memory available per shared-memory compute node. Therefore, we decided to use a native loader for the bathymetry data which loads the entire bathymetry datasets into the memory.

For the Tohoku Tsunami simulation, the GebCo dataset is preprocessed with the Generic Mapping Tools [WS91]; they map the bathymetry data given in longitude-latitude format to the area of interest, see Fig. 6.7. We used a length-preserving mapping, which conserves the length from each point on the bathymetry data to the center of the displacement data.

Initialization

For the Tohoku Tsunami, an earthquake resulted in displacements of the sea ground which led to a change of the water surface height. We consider a model which assumes a change in the water surface only at the beginning of the simulation. Hence, for the initial time step of the Tsunami simulation, we require the information on the displacements describing these surface elevation and follow the instructions provided within the Clawpack package[5]. The seismic data we use is provided by the UCSB[6] and used as input to the Okada model [SLJM11] computing the displacements.
Our initialization is based on an iterative loop:

(1) In each iteration, the conserved quantities are reset to the state at time $t = 0$. These conserved quantities are the water surface height including the displacement. The bathymetry data is sampled from the multi-resolution datasets, and both momentum components are set to zero.

(2) Then, a single time step is computed and the grid is refined with adaptivity requests based on the net-update parameters (see Sec. 6.1.3).

(3) If the grid structure changed, continue at (1), otherwise continue with the simulation.

This setup relies on the local extrema of the displacement datasets already detectable by the net-update error indicator.

Adaptivity parameters

Similar to the analytic benchmark in Sec. 6.2, we conducted several benchmarks with different initial refinement parameters $d = \{10, 16, 22\}$ and up to $a = \{0, 6, 12\}$ additional refinement levels. The refinement thresholds used by the error indicators are $r = \{50, 500, 5000, 50000\}$ and $c = \frac{r}{5}$ for the coarsening thresholds.

Comparison with buoy station data

We first conducted studies by comparing the smiulation data with the water-surface elevation measured at particular buoy stations. This elevation data is provided by NOAA[7]. The tidal

[5] http://depts.washington.edu/clawpack/users/quick_tsunami.html
[6] http://www.geol.ucsb.edu/faculty/ji/big_earthquakes/2011/03/0311_v3/Honshu.html
[7] National Oceanic and Atmospheric Administration, National Data Buoy Center

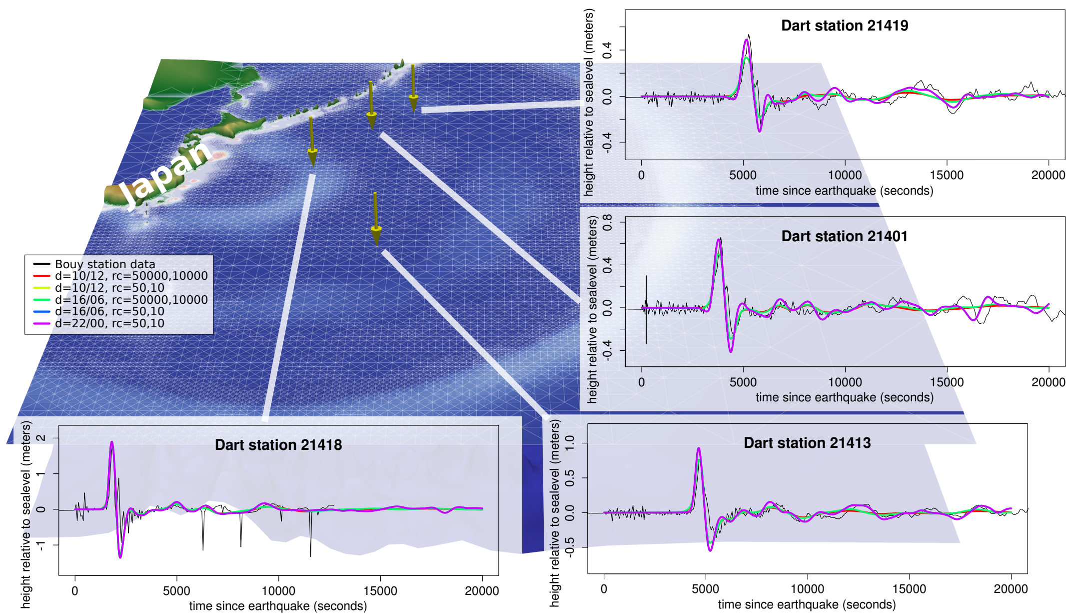


Figure 6.7: Tohoku Tsunami simulation with different buoy stations marked with yellow arrows. The measured and simulated water surface displacements at the four relevant buoy stations are plotted for selected adaptivity parameters.

waves are not modeled within our simulation, but included in the buoy station data. To remove this tidal-wave induced water-surface displacement, we use the detide scripts (see [BGLM11]).

The simulation domain and the simulated and measured displacements of the buoy stations are visualized in Fig. 6.7. This visualization of the simulation grid and the water and bathymetry data is based on a simulation with adaptivity paramters chosen for a comprehensible visualization of the grid.

Regarding the time of the wave front hitting the buoy station, the results show a very good agreement with the data recorded by the buoy stations. However, we should consider that the underlying simulation is based on displacement data computed with a model. Therefore, no concrete statement in the direction of a realistic simulation should be made here, but we continue determining the possibilities with our dynamically adaptive simulations.

Dynamically adaptive Tsunami simulations

We executed Tsunami simulations with the adaptivity parameters described in the previous section. To reduce the amount of data involved in our data analysis, we selected a particular buoy station. We expect, that wave propagations to buoy stations over a longer time are more influenced by factors such as the grid resolution and other parameters compared to wave propagations which take only a short time to reach a buoy station. Therefore we select buoy station 21419, with the peak of the first wave front arriving at the latest point in time compared to the other stations.

We compute the error norm (6.1.2) on the time interval $[0, 20000]$ for several adaptivity parameters; the results are given in Table 6.1. Figure 6.8 shows bar plots of the errors and the normalized average number of cells relative to the maximum value. Dynamically adaptive simulations require an improvement in the error and the number of cells. Detailed results are discussed next.

We use $da = 20/0$ (initial refinement depth of 20, no additional refinement levels) as the

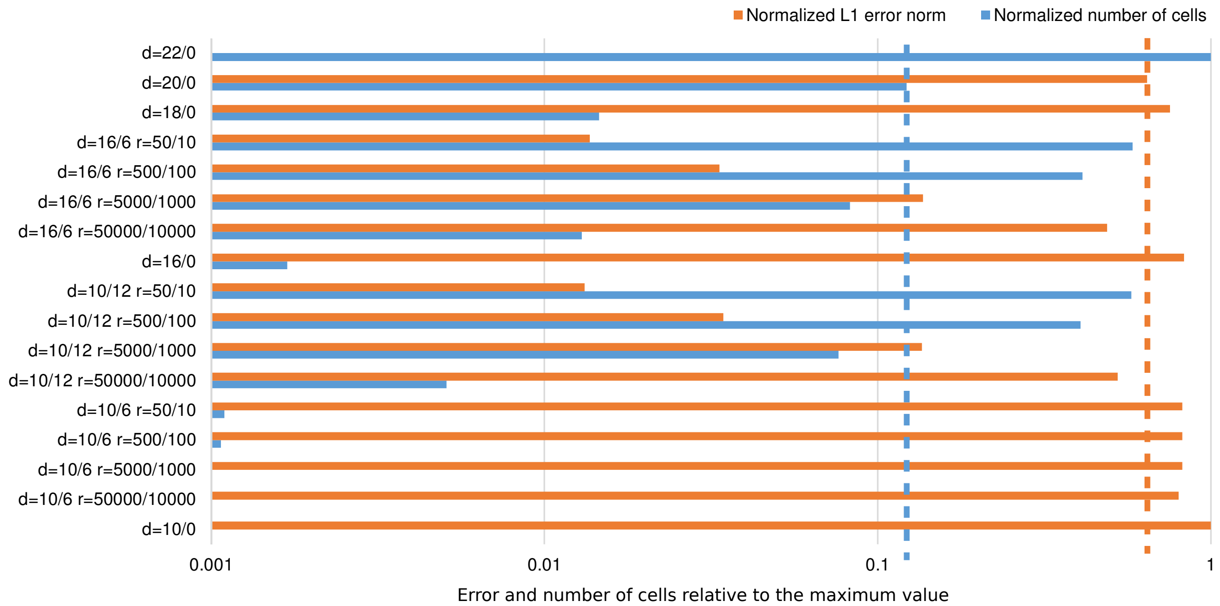


Figure 6.8: Visualization of computed error vs. required number of cells for the entire simulation. More efficient implementations require smaller bars, both for the error and the number of cells used in the entire simulation.

Parameter study	**L1 error**	**Processed mio. cells**	**Saved cells**
da=22/0	0	88936.02	-719.16%
da=20/0 (baseline)	**0.026121**	**10856.96**	0%
da=18/0 r=50/10	0.0306024	**1298.14**	88.04%
da=16/6 r=50/10	**0.0005550**	51775.64	-376.89%
da=16/6 r=500/100	**0.0013608**	36650.05	-237.57%
da=16/6 r=5000/1000	**0.0055551**	**7342.58**	**32.37%**
da=16/6 r=50000/10000	**0.0198121**	**1151.38**	**89.40%**
da=16/0	0.0337187	**150.34**	98.62%
da=10/12 r=50/10	**0.0005356**	51275.72	-372.28%
da=10/12 r=500/100	**0.0013975**	36127.60	-232.76%
da=10/12 r=5000/1000	**0.0055124**	**6784.04**	**37.51%**
da=10/12 r=50000/10000	**0.0213234**	**452.25**	**95.83%**
da=10/6 r=50/10	0.0333180	**97.43**	99.10%
da=10/6 r=500/100	0.0333127	**95.11**	99.12%
da=10/6 r=5000/1000	0.0333143	**75.89**	99.30%
da=10/6 r=50000/10000	0.0325013	**22.86**	99.79%
da=10/0	0.0406040	**0.26**	100.00%

Table 6.1: Different parameter studies and computed error relative to the baseline da=22/0 and saved cells relative to the baseline da=20/0.

baseline for comparison with our dynamically adaptive grids. Comparing this baseline with $da = 22/0$ shows that about 8 times more cells are involved in the computation. This is

(a) due to two additional refinement levels resulting in 4 times more cells and

(b) due to the reduced CFL condition.

With the computed error of 0.026121 for our baseline $da = 20/0$, we first highlighted each computed error with bold face and each number of processed cells below 10856.96 mio. in Table 6.1. Then, the percentage of saved cells compared to the baseline is given in the right column. The last column shows the possibilities with the impact of computational amount with our dynamic adaptivity: we can save more than 95% of the cells required for yielding improved results according to the L1 error norm.

We continue with a more detailed execution of the simulation on a Westmere 32-core shared-memory system to determine the possible savings in computation time. The cluster split threshold is automatically chosen, keeping the number of clusters close to 512. The regular resolution created 512 clusters in average, yielding 16 clusters per core. We measure the time after the initialization of the earthquake induced displacements. The timings for the simulation traversals, adaptivity traversals and split/join phases are presented separately in the following table:

	Simulation parameters		
	da=10/12 r=50000/10000	**da=20/0**	**da=22/0**
Simulation traversals	13.63 sec	288.09 sec	2370.19 sec
Adaptivity traversals	10.44 sec	23.05 sec	157.47 sec
Split/join operations	7.84 sec	5.41 sec	11.22 sec
Sum	**43.71 sec**	**336.03 sec**	**2557.11 sec**

The sum of all values is slightly different to the sum of all measured program phases due to, e.g., overheads induced by measurement. We can see, that the adaptivity traversals and the time spend into the split/join operations exceed the time invested for the simulation traversals. However, this relative overhead pays off due to the reduced overall computation time compared to the regular grid resolution.

(a) $da = 20/0$:
We start by comparing the simulation-traversal time for running the wave propagation on the regularly resolved domain $da = 20/0$ with the entire simulation time (time stepping, adapvitity, split/joins) required by our dynamically adaptive simulation $da = 10/12, r = 50000/10000$. Compared to the simulation $da = 22/0$, this yields a performance improvement of $\frac{288.09}{43.71} = 6.6$.

We next analyze the theoretical maximum performance improvements based on the average number of cells per time step: for the simulation on a dynamically adaptive grid, only 69070 cells per time step are used in average whereas for the regular grid, 2097152 cells were processed per time step, yielding a factor of 30.4 as the expected performance improvement. However, we only gained a factor of 6.6 for which we account by the following three issues:

- The size of the dynamically adaptive grid with only 69070 cells in average per time step is very small, resulting in a relatively low scalability.
- Additional overheads are introduced by the dynamically changing grid structure
- The number of required time steps is increased from 5177 for the $da = 20/0$ simulation to 6543 for the simulation on the dynamically adaptive grid. This is due to smaller grid cells on the dynamically adaptive grid lead to a decreased time-step size.

(b) $da = 22/0$:
Here, we assume that the simulation on the dynamically adaptive grids are sufficiently accurate so that we can compare the refinement depth of the regularly resolved domain to the maximum refinement depth of the dynamical adaptive simulation. This also avoids the aforementioned issue (3) with smaller time steps. Then, we can compare the runtime of 43.71 seconds for the dynamically adapative simulation $da = 10/12, r = 50000/10000$ with the simulation runtime of 2370.19 seconds for a regularly resolved domain $da = 22/0$. Here, we get a performance improvement of $\frac{2370.19}{43.71} = 54.2$. To give another example for the dynamically adaptive simulation with $da = 10/12, r = 10000/2000$, we still get a speedup of $\frac{2370.19}{144{,}42} = 16.4$. We want to emphazise again, that these speedups only hold under the assumption, that the results of the dynamically adaptive simulation are sufficiently accurate.

With simulations on dynamically changing grids, this also results in dynamically changing resource requirements over the simulation runtime. E.g. considering the results for the dynamically adaptive simulation in the aforementioned benchmark scenario (a), the simulation is not able to scale very well on the assigned number of cores due to its small problem size. With concurrently executed applications with dynamically changing resource requirements, e.g. multiple simulations for parameter studies on dynamically adaptive grids, an over-runtime changing resource assignment can result in increased efficiency on which we put our focus on in part IV of this thesis.

6.4 Output backends

The presented application scenarios are mainly driven to gain some insight into the simulated scenario. A visualization of the entire or a fraction of simulation data is one of the most frequently used way to gain this insight and we present different visualization backends.

We aim at generality of our backend infrastructure by considering both on- and off-line backends. For our off-line backend, we use VTK binary file output for off-line visualization with Paraview. Our on-line backend is based on OpenGL and also offers interactive steering of the simulation. The greatest common divisor for OpenGL and VTK backends regarding their simulation data storage format is storing of geometry and primitive data in separated arrays which we use as the input-data format to both backends.

Our main goals are then given by

(a) the development of an efficient off-line backend by writing the simulation data to persistent storage while continuing the simulation in the background and

(b) the visualization of a closed surface for DG simulations.

6.4.1 VTK file backends

With the interest of scientists to analyze the simulation data at different time steps, this data has to be made available for further processing. Using the VTK file backend, the data has to be written to persistent memory. However, typically only a very low bandwidth is available to access such a persistent memory compared to the main memory. Hence, also writing large datasets to it would result in severe bottlenecks and thus idling cores. Here, we studied several implementations to write the output data to persistent memory:

- *No output*:
  This benchmark does not write any files to determine the peak performance.

- *Default (blocking)*:
  The default output method blocks until the function which is called to write all simulation data to persistent storage finished its execution.

- *pthread*:
  A separate thread is started which is writing the simulation data in background to the harddisk while still continuing with the execution of the simulation on all available cores. This can lead to a single core shared among the writer and a simulation thread.

- *pthread, lastcore*:
  This execution is similar to the aforementioned execution above, but does not use the last core for the simulation. This aims at avoiding resource conflicts with the executed writer thread.

- *Writer task*:
  Using TBB for thread initialization, we can use TBB fire-and-forget tasks [MK11]. These tasks are enqueued to a working queue without any thread waiting for the finishing of the task. The idea is to solve the issues with both pthread versions:

  (a) The default pthread version results in potential resource conflict due to preemption with other threads. Using a writer task avoids this due to work stealing.

  (b) The lastcore variant results in an idling core, once the output was written to the persistent memory.

  Using a writer task, other tasks can be processed by the same thread, e.g. with work stealing, after the task finished writing data to persistent memory.

The domain triangulation is based on a quadrilateral and the simulation grid is initialized with $d = 10$ and with up to $a = 16$ additional refinement levels. The simulation computes a radial dam break with the Rusanov flux solvers for 201 time steps. The output data itself is preprocessed in parallel by using all available cores. Such a simulation results in 4.55 mio. cells processed in average per simulation time step and with binary VTK file sizes above 300 MB. We used our Intel platform (see appendix A.2) and write the simulation output data to persistent memory (Western Digital Hard Drive of the typ Red 2 TB with a 64 MB cache and a theoretical transfer rate of up to 6 Gb/s). Results for different frequencies of writing output files to persistent memory are given in Fig. 6.9.

The *blocking version* shows a clear disadvantage compared to the other methods since cores idle until the function which writes the output data finished writing the data. Such idling cores are compensated with the pthread versions. Both *pthread* versions show an improvement. However, the dedicated writer core which we implemented to avoid oversubscription of cores leads to decreased performance of 0.85%, 3.68% and 0.42% percent respectively for writing output files each $B = (25, 50, 100)$ time steps. Hence, avoiding resource conflicts does not result in a robust performance improvements for the tested simulation parameters. Here, the *oversubscription of cores should be used.*

With *TBB* fire-and-forget tasks, we get a robust performance improvement compared to all other writer methods. Furthermore, for writing the simulation data only after more than 100 time steps, the performance loss for writing data to persistent memory is only at 4% compared to writing no simulation data.

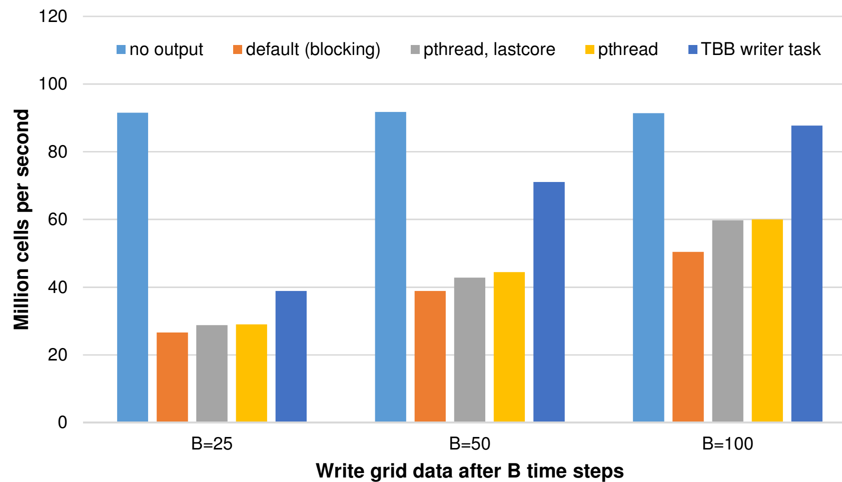


Figure 6.9: Benchmark statistics with million cells per second processed and for different output backends. The parameter B specifies the number of time steps when to write data to persistent memory.

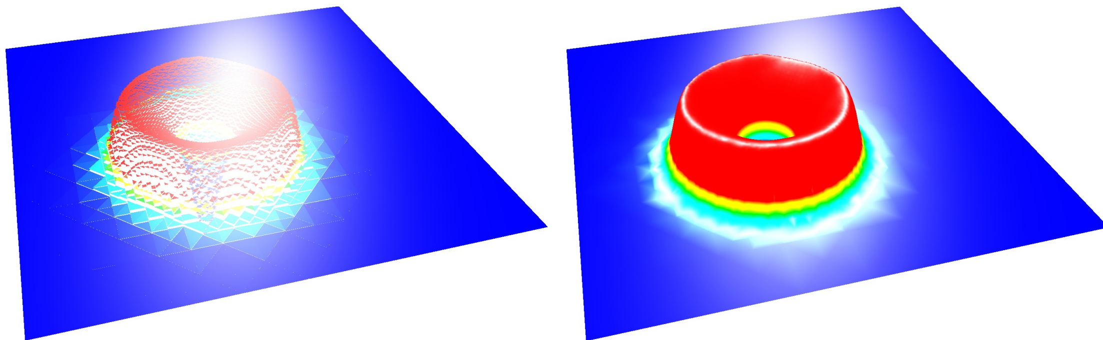


Figure 6.10: Visualization methods of the surface of a shallow-water simulation. Left image: the direct visualization of the finite-volume simulation leads to gaps in the surface. Right image: the closed surface leads to less distraction and improved analysis of the data.

6.4.2 OpenGL

Besides the interactive steering possibilities of our OpenGL backend, here we like to focus on the reconstruction of a closed surface for visualization with the OpenGL backend with the vertex-based communication which was originally developed for node-based flux limiters. For shallow-water simulations, a direct visualization of the approximated solution with simulations based on the DG method leads to a surface with gaps. An examplanory visualization of a particular time step for a radial dam break is given in Fig. 6.10. Such gaps lead to a distraction of the person analyzing the data. For the visualization of a closed surface for shallow water DG simulations, cell data such as the water surface height can be averaged based on the surface height in cells sharing the vertex.

A generation of triangle strips for visualization was already considered with algorithms based on the Sierpiński SFC [PG07]. To our best knowledge, no visualization was developed so far which is capable of computing both vertex and normal data on-the-fly for surface reconstruction with dynamically adaptive triangular grids based on simulation data with a close to $O(\#cells)$ complexity.

We compute a closed surface with our vertex-based communication scheme. Here, the

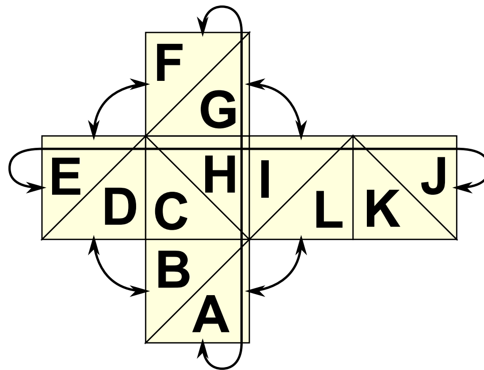


Figure 6.11: Cubed sphere domain triangulation with periodic boundary conditions.

per-cell approximated height is averaged at the vertices and used as the vertex for the water surface visualization. However, only considering vertex coordinates with a vertical displacement, e.g. based on water surface displacement, would not result in proper shading since normals are required at vertices. Therefore, we continue with additional traversals computing the normals associated to the previously computed vertices. This is based on the face orientations and quantitative properties for each triangle, see [JLW05] for further information.

Since traversing the cells is $O(n)$ with a negligible overhead ϵ for reduce operations for a large cluster, and by using a fixed number of grid traversals, this also yields an $O(n + \epsilon)$ complexity for the reconstruction of our closed surface including the normals at each vertex. Other algorithms to reconstruct a closed surface such as the Voroni triangulation require at least an $O(n \log n)$ algorithm in the worst-case [AK00], whereas our SFC traversal yields a robust $O(n + \epsilon)$ algorithm for the surface reconstruction, including cluster-based parallel processing.

Examples of the resulting surface visualization with the OpenGL backend are e.g. given in Fig. 6.10 and Fig. 6.12.

6.5 Simulations on the sphere

Here, we present the possibility of simulations on the sphere based on our development. Using a two-dimensional quadrilateral domain shape and mapping this domain onto a sphere would lead to so-called pole singularities with sharp angles. Several solutions are available such as an icosahedral, a cubed sphere and a ying-yang grid (see [Beh06]). In this work, we use the cubed sphere, which was originally intended to be used for quadrilateral-like primitives.

With our framework being optimized for triangular grids created with a bisection, we start by assembling two triangles to a quadrilateral. Then each quadrilateral can be used as a cube face. This cube can be unfolded to a two-dimensional mesh which assembles the faces of a cube with a base triangulation given in Fig. 6.11. This figure also shows the cube-periodic boundary conditions with the arrows.

The cube's center is assumed to be placed into the sphere's center with a grid on each side of the cube. Then, each cube side is projected to the surface of the sphere and, due to the changes in angles, creates a distorted grid. Projection methods can then be e.g. equidistant or equiangular ones [NTL05].

To account for such a distortion of grid cells, the terms of our weak formulation of the continuity equation (see Eq. (2.8)) need to be extended to account for the projection of the cell from "sphere space" to "reference space", see [Gir06] for detailed information. To compute these distortions, we require the knowledge on the cube's face as well as the vertex coordinates

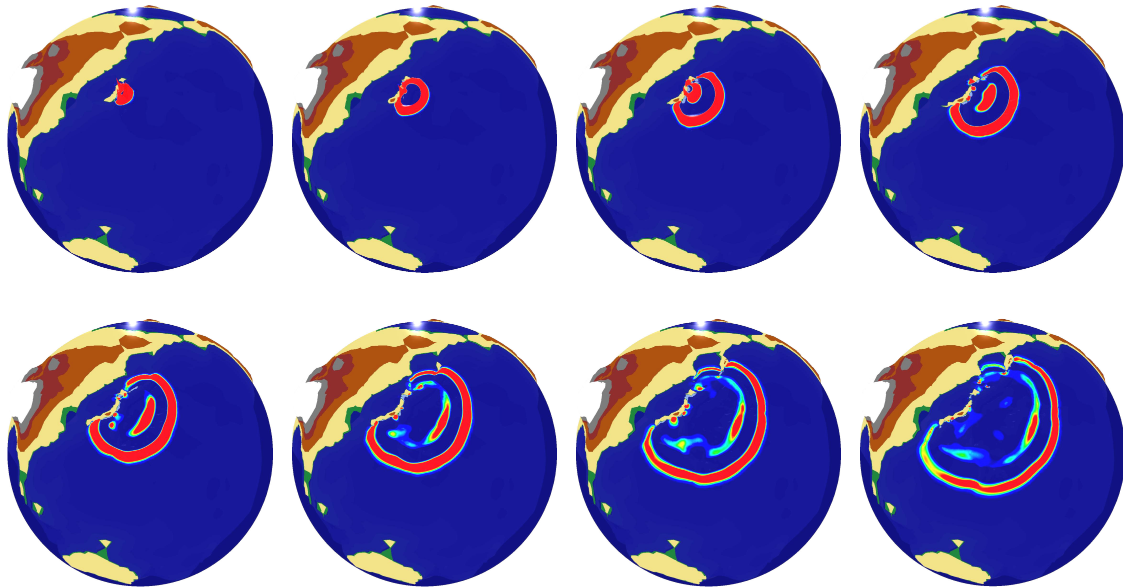


Figure 6.12: Visualization of the experimental earth-scale Tohoku Tsunami simulation executed on dynamically adaptive grids with the OpenGL backend.

on the sphere. Based on the two-dimensional vertex coordinates, we can then

(a) determine the cube's face id,

(b) compute the coordinates of the triangles on the spherical surface and compute the distortion matrices with equations given in [Gir06, CLDL09].

Further extensions such as the Coriolis force and also the verification of the implementation of these projections is work under progress.

Combining this simulation with our OpenGL output backend which was discussed in Section 6.4.2, we can compute and directly visualize the propagation of the Tohoku Tsuami simulation over the entire earth globe based on the augmented Riemann solvers [Geo08]. Examplanory screenshots are given in Fig. 6.12.

6.6 Multi-layer simulations

With the shallow water equations, a simplification of an originally three-dimensional model is used. This allows computationally more efficient simulations with results close to the three-dimensional formulation. This is not possible in all cases such as weather and climate simulations. Considering e.g. the model used by the Deutscher Wetter Dienst (DWD), a multi-layer discretization in the vertical direction is used to simulate three-dimensional effects.

We also extended our framework with such a multi-layer approach. Here, we present the multi-layer simulation of the Euler equation. A constant number of layers is assumed in each grid cell. The two-dimensional cell-data storage is then used to store a pile of three-dimensional cells. We introduce a new terminology for this extension: the three-dimensional cells are further denoted as volumes. Edges are further described as *adjacent faces* and the shared interfaces of two piled cells are named *local face*, see Fig. 6.13.

For a basic 3D DG simulation, the following major building blocks of a multi-layer simulation are required:

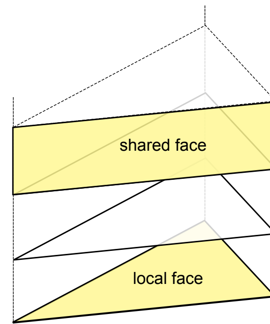


Figure 6.13: Multi-layer approach: multiple cells are stored triangular grid cell.

1. *Gather flux* parameters on faces.
2. *Compute fluxes* on adjacent faces and local faces.
3. Compute *time step size*.
4. Based on flux updates and time step size, *integrate the time* in each volume.

We also have to compute the *time step size for the local faces*. This would require an extension of the framework with additional interfaces. However, we overcome this by utilization of (a) the cluster-local user data to temporarily store flux updates and (b) the kernel interfaces for storing edge communication data:

- *Executing flux computations for local faces:*
  For communication of flux parameters via edges, the corresponding interface is executed exactly once for each triangle edge and for each time step.

  Hence, we can use one kernel handler, e.g. for the hypotenuse (*cell_to_hyp*). This allows us to compute fluxes for the local faces.

- *Storing flux updates:*
  After computing the fluxes for the local faces, these fluxes have to be temporarily stored until the time step size is known. Since we aim at memory efficiency computations, we do not store the fluxes in each cell, since they are e.g. not required during adaptivity, but we extend the *cluster-local user data* with an additional stack system to temporarily push computed fluxes to this stack.

- *Computing time step size:*
  We also store the wave speed computed for the local faces to cluster-local user-data. After computation of the fluxes with the adjacent faces of the cluster, the wave speed from the local faces is involved in returning the per-cluster maximum wave speed required for computing the maximum time step size.

- *Time stepping:*
  With the flux updates for the adjacent faces and for the local faces fetched from the cluster-local stack, the DoF are advanced in time.

This extension finally leads to the capability of handling multi-layered simulations transparently to the framework which was originally only developed for two-dimensional simulations.

6.7 Summary and outlook

We presented several application scenarios to show the applicability and benefits of dynamically adaptive grids. Based on an analytic benchmark, we showed the correct implementation and that over 96% cells can be saved with a dynamically adaptive grid. Then a Tsunami simulation was implemented to show the applicability of the dynamic adaptivity within a realistic scenario. With a relatively small benchmark scenario, the computation time for the simulation was already improved by a factor of 6.6 with results of higher accuracy compared to the simulation on a regular grid. Assuming, that the results with the dynamically adaptive simulation are sufficiently accurate, the runtime improvement is larger than a factor of 54. For the data analysis with an offline processing, we tested several output backends for online and offline processing. Here, online processing allows direct visualization of the simulation as well as interactive steering methods. We further developed several ways of writing output data to persistent memory. Here, writer tasks result in only 4% loss in performance compared to a simulation which is not writing any output data. Finally, we presented our extensions for simulations on the sphere and a multi-layer discretization.

There are a couple of issues that deserve further investigation in the future. Among others, these are given as follows:

- *Cluster-based local-time stepping:*
  With the naming "clustering" originating from the cluster-based local-time stepping idea, this is obviously one possible utilization of our approach. Since our cluster-based approach is similar to block-structured grids (see Sec. 5.10.5), we expect that an extension to cluster-based local-time stepping can be accomplished in a similar way as in the PeanoClaw [UWKA13] framework.
- *Resiliancing:*
  The independency of clusters allows an efficient duplication and forwarding of one or more clusters to other compute nodes for resiliancing. A replacement of a computation node can then be accomplished by initializing the simulation at another node, based on the duplicated clusters and a reconstruction of RLE adjacency information only on the cluster-adjacent MPI ranks.
- *Dynamically changing simulation data:*
  Considering our GUI which also offers interactive steering methods such as setting parameters during the simulation's runtime and modifying the cell data and grid structure, this yields further applications. Some of them are e.g. the interactive testing of flooding scenarios or the simulation of dynamic earthquake-induced displacement data.

PART IV

INVASIVE COMPUTING

Current batch-job schedulers of super-computing centers rely on a static number of resources assigned to a program during its execution time. This number of resources is typically specified by the application developer at the time of enqueuing the application to the batch system.

Considering the overall system's state with multiple applications executed in parallel, such a static resource allocation is incapable to account for runtime-changing resource requirements of applications. These changing resource requirements are e.g. induced by simulations with dynamically adaptive mesh refinement. Hence, there is a demand for dynamic resource allocation which leads to challenging issues such as coping with the dynamic resource allocation for concurrently executed applications on HPC systems.

With the Invasive Computing paradigm, a promising approach is suggested for a dynamic resource management. This paradigm was initially suggested for multiprocessor System-on-Chip (MPSoC), in particular tightly-coupled processor arrays [Tei08] focusing on the efficient utilization of a two-dimensional array of computing cores for prospective computing architectures with hundreds and thousands of cores. There, the authors introduce the idea of the ability of programs to copy and execute themselves on computing resources in their proximity. Three basic operations are suggested; here we give a brief description on the operations which are relevant for this work:

- With *invade*, each computation resource can request additional resources. Desired properties of the requested resources can be specified by an additional constraints. Such constraints can be e.g. the number of cores, floating-point support, etc.
- The next step is *infect* which replicates the program into the successfully invaded cores and starts the program.
- Once the computations are finished, a *retreat* frees the previously invaded cores and makes them available for further invades.

Despite originally designed for MPSoCs, the Invasive Computing paradigm also yields the potential of being applied to our HPC computing issues.

Chapter 7: Invasive Computing with invasive hard- and software
We first give a brief introduction to our development on invasive extensions for algorithms developed within the Transregio research project "Invasive Computing". Since this thesis focuses on Invasive Computing on HPC shared-memory architectures, we only show the multigrid algorithm as a representative application from the area of scientific computing to show the differences of Invasive Computing on MPSoCs to Invasive Computing on HPC shared-memory systems.

Chapter 8: Invasive Computing for shared-memory HPC systems
For Invasive Computing in HPC on shared-memory systems, a concrete realization of Invasive

Computing is presented in this chapter. This is followed by results on the paradigm applied to simulations on dynamically adaptive grids, including Tsunami parameter studies.

7 Invasive Computing with invasive hard- and software

We give a brief overview of our work on Invasive Computing within the "Invasive Computing"[1] transregio (TR) project which inspired the application of the invasive paradigm also in HPC.

Thousands and more heterogeneous cores are expected on future multi-processor systems on chips [Tei08]. For concurrently executed applications or applications with changing resource requirements during runtime, this demands an orchestration of resources. Our approach is based on giving the application the possibility to specify resource requirements during run time with the invasive commands $invade$, $infect$ and $retreat$ described in the previous section. E.g. if an application fails to yield a particular parallel efficiency due to reduced workload, it can retreat from resources. This makes the resources available to other applications. To support these invasive commands in an efficient way, several challenges have to be solved such as scheduling of resource-competing applications, programmability, algorithmic redesign, new hardware components, etc. Contrary to our HPC approach which relies on extensions to standard parallelization models and software extensions only, the invasive hardware platform developed in the TR also offers invadable hardware components on a multiprocessor systems-on-a-chip (MPSoCs) and an invasive software tool chain.

Our main task in this TR is to provide the knowledge on algorithms from the area of scientific computing. Based on the requirements of our algorithms, we took part in the decision making of several developments of the Invasive System on the hard- as well as software layers. To show the differences to our HPC approach, we continue to give a brief introduction to the Invasive System.

7.1 Inavsive hardware architecture

Here we give a short overview of the Invasive System architecture from the HPC perspective. An example configuration of the Invasive Chip is given in Fig. 7.1.

- *Network on Chip:*
  The hardware architecture is build upon a multi-tile infrastructure. The network on chip (NoC) connects all tiles with a two-dimensional mesh network.
- *Tile:*
  Each tile can have a local memory, a CiC, standard Leon cores and special accelerator cores.
- *CiC:*
  Programs are able to start computations on the same or on another tile. They enqueue the kernels in the Core i-let Controller (CiC) of the target tile. The CiC is then able to start the execution of kernels (i-lets) on the cores without software scheduling overheads.

[1] `http://invasic.informatik.uni-erlangen.de`

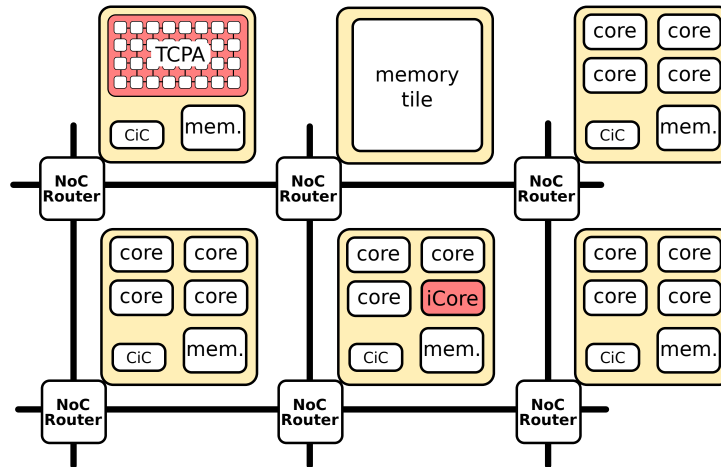


Figure 7.1: Example configuration of the invasive multi-tile architecture. See the text for a further information.

- *Accelerators:*
  Besides standard Leon cores, two different types of cores are available with the Invasive system: On the one hand, tightly-coupled processor arrays (TCPAs) which are based on a regular mesh of cores. The resources used by an existing programs can be extended by infecting processing elements in their proximity or shrunk by retreating from processing elements. On the other hand, i-Cores provide a run-time adaptive instruction set and adaptive micro architecture. This allows uploading of special instructions and modification of the core parameters to support applications which have different requirements.

- *Memory model:*
  Each tile has a local memory with cache-coherent access on each tile. The global memory on the memory tile is segmented and each segment is assigned to a single tile. For the global-memory model and the the vision of thousands of invadable cores on one System on Chip (SoC) making a cache coherency very challenging, we decided to use no cache coherency between tiles.

7.2 Invasive software architecture

Considering algorithms on a system without cache coherency among the tiles, this requires extensions similar to distributed memory systems, e.g. a message-passing interface (MPI). Parallelizing an application on such distributed-memory systems typically results in an increased complexity.

This complexity is compensated by the development of our algorithms in the *X10 language* [ESS05]. Additionally, an *X10 compiler* [BBMZ12] is developed to generate optimized code for the Invasive Chip and the *OctoPOS* operating system [OSK+11] running on the chip.

A decentralized *agent system* [KBL+11] then schedules the resources among the executed applications, based on the constraints and possibly other input parameter from hardware monitors.

The X10 language supports a so-called partitioned global address space (PGAS). This improves the usability for accessing arrays which are distributed among several tiles via a virtual continuously stored array structure. This hides the physical placement of the data chunk from

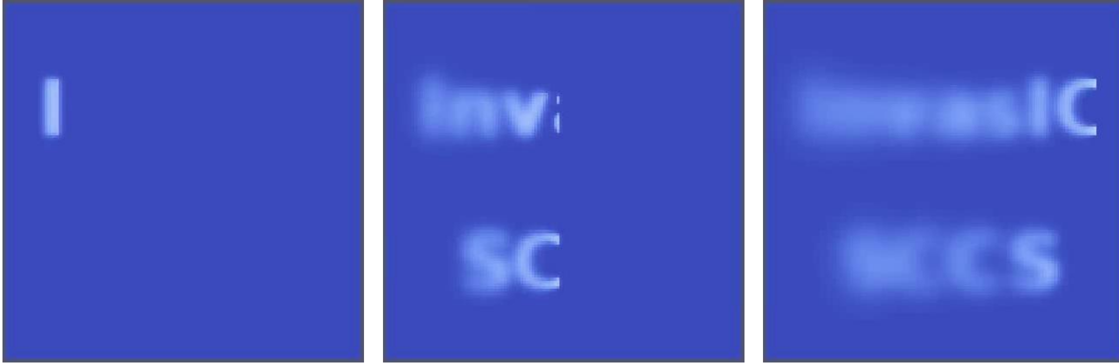


Figure 7.2: Heat distribution on a metal plate with a laser engraving symbols from left to right onto the metal [BRS+13].

the application developer. For our algorithms, we use the *distributed arrays* from the X10 library.

It is important to mention, that the PGAS features in X10 already compensate the additional burden which is put on the developer due to the distributed-memory system. However, there's still an increased complexity due to required extensions of algorithms to cope with *dynamically changing number of compute resources.*

For the programmability of the Invasive Computing paradigms, we contributed a variety of algorithmic patterns to the research groups of Prof. Dr. G. Snelting and Prof. Dr. J. Teich, resulting in the invadeX10 framework. This framework offers basic invasive computing interfaces such as invade, infect and retreat via the *invadeX10* [BRS+13]. Furthermore, additional extensions were required to compensate the complexity which is introduced by the Invasive Computing paradigms. Such extensions are e.g. the redistribution of distributed arrays after a change of resources to improve the data locality, infections supporting recursive reduce operations to compute a surplus, etc.

7.3 Invasive algorithms

With the knowledge about algorithms from the area of scientific computing, we developed numerical core algorithms in invadeX10. Such algorithms are a multigrid solver with changing workload due to restrictions and prolongations, a dynamic adaptive quadrature based on recursion and lightweight tasks and a Peano-SFC-based matrix-matrix multiplication [Bad08].

Since the multigrid solver with its multi-resolution access is one of the most interesting algorithms for invasion, we selected this solver to highlight the required changes in the program structure and like to refer to [TOS00] for a detailed introduction to such multigrid solvers.

As a representative application scenario, we selected a laser engraving symbols on a two-dimensional metal plate, see Fig. 7.2. This process can be simulated with the discretization of the heat equation which is discussed next, followed by a brief introduction to the multigrid algorithm which is then used to solve the system of equations.

Let the change in heat distribution over time for a two-dimensional problem be given by

$$\frac{dT(x,y,t)}{dt} = \alpha \Delta T(x,y,t) + E(x,y,t), \quad (x,y) \in \Omega.$$

on a domain $\Omega = [0,1]^2$ with the temperature T, the external energy E (e.g., a laser) and the thermal diffusivity coefficient α. We use Dirichlet boundary conditions of 0 on the domain

boundaries $d\Omega$. For the discretization in time, we use 1st-order forward differences and an implicit update scheme for the time stepping, yielding

$$\frac{T(x,y,t+\Delta t) - T(x,y,t)}{\Delta t} = \Delta T(x,y,t+\Delta t) + E(x,y,t).$$

This can be formulated with a system of linear equations $\mathbf{A}\vec{x} = \vec{b}$ which can be solved with iterative solvers to compute an approximated solution.

Using a Jacobi solver, a single iteration mainly smooths the high-frequency errors, only. In case of low-frequency errors, multigrid solvers are commonly used. Here, we use the error-correction scheme of the multigrid solvers which restricts the residual $\vec{r} := \vec{b} - \mathbf{A}\vec{x}^*$ successively to coarser levels. Then, on each coarser level, we apply a Jacobi smoother iteration to compute the residual and restrict the residual to the coarser level. Each restriction operation then results in a reduction of the workload on each level by a factor of 4. After the execution of the smoother on the coarsest level[2], the computed error-correction is successively prolongated to the higher-resolved levels and an additional smoother iteration is executed before prolongating it to the next level.

With our heat equation and with a size of the simulation domain of 128×128, 7 levels with different resolutions are used. Since each level has a changing workload, this also results in a dynamical resource requirement.

For the parallelization, we use the distributed arrays from X10 to store the approximated solution $\vec{x}$ of the iteration, the right side $\vec{b}$ and the residual $\vec{r}$ for each level.

Next, we compare a pseudo code of a non-invasive and an invasified multigrid algorithm in Fig. 7.1. To show the applicability of the dynamical resource management, we shrink the number of compute resources during the restriction.

<table>
<tr><th>Standard multigrid</th><th>Invasive multigrid</th></tr>
<tr><td>vcycle(N, x, b):</td><td>vcycle(N, x, b, homeClaim):</td></tr>
<tr><td>r = computeResidual(N, x, b)</td><td>r = computeResidual(N, x, b, homeClaim)</td></tr>
<tr><td>while |r| > threshold:</td><td>while |r| > threshold:</td></tr>
<tr><td>vcycleIteration(N, x, b)</td><td>vcycleIteration(N, x, b, homeClaim)</td></tr>
<tr><td>r = computeResidual(N, x, b)</td><td>r = computeResidual(N, x, b, homeClaim)</td></tr>
<tr><td>vcycleIteration(N, x, b):</td><td>vcycleIteration(N, x, b, claim):</td></tr>
<tr><td>smoother(N, x, b) # Pre−smooth</td><td>smoother(N, x, b, claim) # Pre−smooth</td></tr>
<tr><td>r = residual(N, x, b) # Residual</td><td>r = residual(N, x, b, claim) # Residual</td></tr>
<tr><td></td><td>nc = reinvade(Nr, claim) # Reinvade Claim</td></tr>
<tr><td>Nr = N/2 # Restricted Level</td><td>Nr = N/2 # Restricted Level</td></tr>
<tr><td>rr = restrict(N, r) # Restrict Residual</td><td>rr = restrict(N, r, nc) # Restrict Residual</td></tr>
<tr><td># To new Claim</td><td># to new Claim</td></tr>
<tr><td>er = floatArray(Nr, 0) # Setup with 0</td><td>er = floatArray(Nr, 0, nc) # Setup with 0</td></tr>
<tr><td>vcycleIteration(Nr, er, rr) # V−Cycle</td><td>nc2 = vcycleIteration(Nr, er, rr, nc) # vcycle</td></tr>
<tr><td></td><td># Redistribute due to changed Claim</td></tr>
<tr><td></td><td>if (nc != nc2):</td></tr>
<tr><td></td><td>nc = nc2 # Update Claim</td></tr>
<tr><td></td><td>x.redistribute(Nr, nc) # Data Migration</td></tr>
<tr><td></td><td>b.redistribute(Nr, nc) # Data Migration</td></tr>
<tr><td>e = prolongate(Nr, er) # Prolongate error</td><td>e = prolongate(Nr, er, nc) # Prolongate Error</td></tr>
<tr><td>x = x + e # Apply correction</td><td>x = x + e # Apply Correction</td></tr>
<tr><td>smoother(N, x, b) # Post−smooth</td><td>smoother(N, x, b, nc) # Post−Smooth</td></tr>
<tr><td>return</td><td>return nc # Return possibly modified claim</td></tr>
</table>

Table 7.1: Comparison of pseudo code for non-invasive and invasive versions of the X10 multigrid [BRS+13].

Based on the invadeX10 framework, each v-cycle iteration is extended with the *claim* as a parameter which describes the currently invaded processing elements. The resources in these claims can be dynamically changed with a *reinvade*, based on the number of slices of the solution

[2] Typically a direct solver is used here

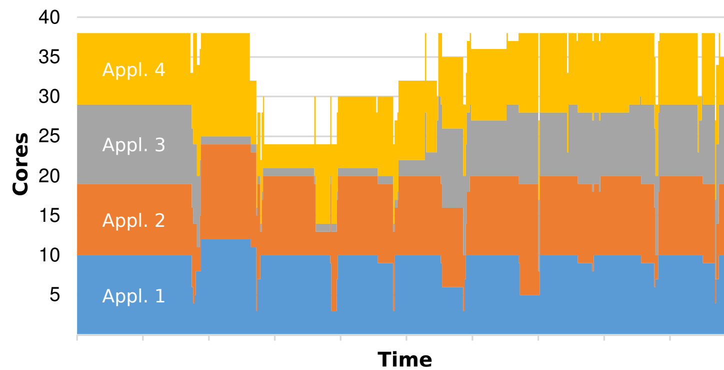


Figure 7.3: Dynamic resource distribution with four concurrently executed multigrid algorithms. The huge gap represents the first v-cycle of all applications [BRS+13].

array on the currently restricted multigrid level. In case that a *reinvade* led to a change of resources (nc != $nc2$), also the distributed arrays are *redistributed* to improve the locality to the computing resources.

7.4 Results

To evaluate the feasibility of dynamical reassignment of compute resources with concurrently running applications, we setup a test environment on a 40-core HPC system to show the functionality of the dynamic resource redistribution. Four multigrid applications, which are started concurrently (see Sec. 8.4), compute the same simulation of a laser engraving symbols on a metal plate, see Fig. 7.2. The feasibility of the dynamic resource redistribution is shown in Fig 7.3.

Here, the v-cycle of the gray application is clearly visible, whereas the resources in the claims of the other applications are not immediately released. Due to hardware issues, an evaluation on the invasive System is not possible, yet.

8 Invasive Computing for shared-memory HPC systems

Today's batch-job schedulers for HPC systems lead to an execution of jobs with a static resource allocation over their runtime. Despite this static resource assignment being omnipresent on HPC systems, this comes with two major drawbacks:

- First, starting applications is not possible in case of *insufficient resources available*. This is due to compute resources used by other applications which cannot be reassigned to other applications without stopping them. This lack of resource reallocation in the applications then leads to idling resources until the remaining amount of requested resources becomes available.
- Second, for concurrently executed applications with *unforeseeable changing resource requirements* as it is the case for our simulations on dynamically changing grids, a static resource allocation is obviously unable to cope with a changing resource requirement.

We suggest addressing these issues by applications adapting to changing resources and a resource manager (RM) which optimizes the resource distribution towards improved application throughput. Despite our framework is capable of being executed on distributed-memory systems, we only consider Invasive Computing for shared-memory systems here as a proof of concept. This was partly developed in collaboration with the research group of Prof. Dr. M. Gerndt, see e.g. [GHM+12].

In this work, each application is assumed to be parallelized with OpenMP or TBB. The required extensions for Invasive Computing on these parallelization models are discussed in Sec. 8.1. The invasive client layer offers the API to request changing resources and to update the resource requirements if requested by the resource manager, see Sec. 8.2. Scheduling of the resources is based on information provided by the invasive applications with the scheduling decisions evaluated in the RM which is described in Sec. 8.3.

8.1 Invasion with OpenMP and TBB

Our Invasive Computing extensions are build on existing functionality of *OpenMP*[1] and *Intel TBB*[2]. Both parallelization models offer parallelization via pragma language extensions or via embedding into the C++ language with a library, respectively. A parallelization on shared-memory has similar restrictions compared to distributed-memory systems which are not considered in HPC standard threading libraries so far:

- On shared-memory systems, an application can always be started using all available resources. However, an application should not be started when some of the accessed computing resources are used by other applications. Otherwise this leads to preemption and

[1] `http://openmp.org/`
[2] `https://www.threadingbuildingblocks.org/`

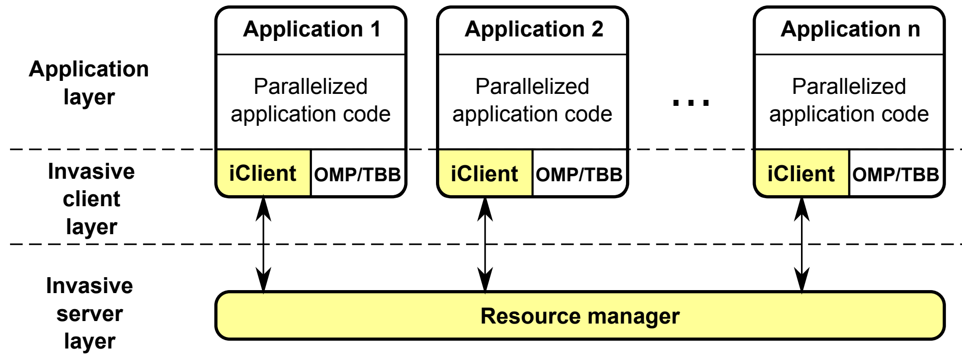


Figure 8.1: Overview of Invasive Computing layers on shared-memory system. Each application is extended by a client layer adopting the resources and communicating with the resource manager.

caches shared among both running applications [KCS04], hence leading to a severe loss of performance. This is in particular important for urgent computing (see e.g. [BNTB07]) with requirements of starting an application despite other applications already use the required resources.

- A changing scalability of algorithms cannot be considered in an a-priori thread allocation. Our DAMR simulations introduced in Part III with their changing workload over the simulation leads to a strongly varying scalability over runtime. For significantly smaller workloads, see e.g. Tsunami parameter studies, this also leads to an underutilization of resources if not dynamically and efficiently shared with other concurrently running applications.

The applications considered in this work are based on time-stepping schemes. Here, we assume a loop, iterating over the time steps required for the simulation and the parallelization only inside the loop. Due to insufficiencies of OpenMP and TBB to change the number of threads inside a parallel region (see e.g. [Ope08] for OpenMP), we allow changes of threads only at the very beginning of each loop, thus only between each simulation time step. To support invasion of cores, we then have to (a) change the number of threads capable of work stealing and (b) set the pinning of the work stealing threads to physical compute cores.

(a) For OpenMP, we set the number of threads with *omp_set_num_threads(#cores)*, and using TBB, the worker threads are set by *tbb::scheduler_init(#cores)*.

(b) Regarding the pinning, we accomplish this by executing a single task for each thread, e.g. using a parallel for loop over the number of available cores and a chunk size of 1. For TBB, we first set the affinity of each task to the corresponding thread which is used to invade a core. In each thread, mutices are then used to avoid work stealing. Otherwise, such work stealing can result in unpinned threads or even a thread pinned to the wrong core. Inside the task, the affinity of the executing thread is then set to the invaded core, based on information provided by the RM.

We only update the number of active threads in each application and their pinning to cores every time if there's a change in resources either in the number of threads or their pinning.

Considering the previously mentioned requirements, this leads to a software design presented in Fig. 8.1. This extends each application with an *invasive client layer* which offers the invasive

commands which are discussed in the next section. OpenMP and TBB are supported by this client-side extension. The resource manager then orchestrates the resources for all registered invasive applications.

8.2 Invasive client layer

The Invasive paradigm was originally developed for specialized Invasive hardware, whereas we apply this paradigm on HPC shared-memory systems. This leads to different programming interfaces which are discussed next.

- *Joined invade and infect call:*
  We can join the *invade* and *infect* calls to a single *invade*. This is due to the shared-memory systems not requiring to infect a compute resource by replicating the kernel code since it is already accessible in each thread's memory. As soon as the Invasive Computing paradigm is extended to HPC with distributed-memory systems, the *infect* call is required, e.g. to synchronize the simulation data such as the number of time steps.
- *No claims:*
  For our target application, the main simulation loop leads to invasions only requiring a single claim which represents the currently invaded resources. Thus, we assume a *single claim* existing for each application and invasions modifying this claim only.

This results in a simplified command space without infects and also no claim for Invasive Computing with our iteration-based shared-memory application. We next discuss the required constraint system for our simulations on dynamically changing grids (Sec. 8.2.1), the communication to the resource manager (Sec. 8.2.2) and the Invasive Computing API for shared-memory systems (Sec. 8.2.3).

8.2.1 Constraints

To schedule the compute resources for which the applications compete for, the applications have to provide information on their current state. Such a constraint specification can lead to very complex structures if requiring a general applicability on heterogeneous systems. E.g. an application can either require 3 cores and one accelerator or in case that the accelerator was not invadable, require more than three cores. This can be expressed with a tree-structure, resulting in a constraint-hierarchy (see [BRS+13]). Then, different constraint constellations can be combined with AND or OR relations. On the one hand, this constraint hierarchy yields a high flexibility, whereas on the other hand, forwarding this information to the resource manager can lead to expensive communication overheads to/from the resource manager. This is due to serialization of the constraint tree, sending the serialized version to the RM, receiving it and unpacking it by de-serialization. Additional complexity is generated inside the RM which has to evaluate the constraint hierarchy by using the constraint trees of different applications to optimize the resource assignment concurrently scheduled applications.

With our HPC shared-memory systems considered in this work which are only based on homogeneous cores, we decided to avoid such a constraint hierarchy by using a fixed number of non-hierarchical constraint properties in a list. Our constraint system is then based on the following properties:

- *min/max cores*: This constraint limits the number of requested resources to the specified range of resources which are claimed.

- *scalability graph*: Such a scalability graph is specified with a one-dimensional array with each entry representing the scalability for the number of cores.
- *workload*: This hint can be used to distribute more cores to applications with more workload and vice versa.

Once specified by the application, these constraints then have to be communicated to the resource manager with details on the resource scheduling for scalability graphs and distribution hints further described in Section 8.4.

8.2.2 Communication to resource manager

A centralized resource manager is used which is responsible to optimize the resources based on constraints received from applications. Our client-server design (see Fig 8.1) suggests a message-based communication with the clients transferring constraints of a typically small message size to the server running as a dedicated process. Then the server (resource manager) requires receiving the message, processing it and sending one or more messages to the client (application). To overcome message processing latencies, also the ability of asynchronous communication is required.

All these requirements are fulfilled by the *IPC System V message queues (MQ)*, supporting small message sizes which can be efficiently exchanged with the server. MQs also support testing whether a message can be dequeued for non-blocking communication.

During its setup, the centralized resource manager creates a message queue with a unique identifier. This makes the message queue singleton-like for all invasive applications which use the same identifier to communicate to the RM. Both the client and the server applications can then en- and dequeue messages using this message-queue identifier. Each message further has to be marked with a token specifying the receiver. We then mark the RM with identifier 0 and each invasive application with the application's system-wide unique process id.

8.2.3 Invasive Computing API

The following invasive interfaces are then offered to the application developer:

- **setup()**
  This has to be executed directly after the program starts. It registers the application at the resource manager which only sends back a message if at least a single core was allocated to the application. Here we aim at avoiding initial resource conflicts.
- **shutdown()**
  This sends a signal to the RM to release all resources associated to the application. The application is then expected to exit immediately.
- **invade_blocking(constraints)**
  We further refer to the initially suggested *invade* call as a *blocking invade* since we also introduce non-blocking invasions. To highlight the differences, we first describe message processing of the blocking invade call:

  (a) A message including the constraints is sent to the RM *and the program looses the control flow.*

  (b) Then the RM optimizes the resource utilization based on all available constraints of each invasive application. This is further discussed in Sections 8.3 and 8.4.

(c) After evaluations and optimizations inside the RM, possible changes of resources are sent to the application and the corresponding resources are marked to be used by the application. Such a resource update message then includes the core ids which are infected by setting appropriate affinities (see Sec. 8.1).

(d) Finally, the *control flow is given back to the program.*

- **invade_nonblocking(constraints)**
  As previously described, using a blocking call for invasions involves latencies when waiting for feedback from the RM. These overheads lead to idling of all threads of an application while waiting for the RM's response. With non-blocking invasion, the RM sends resource-update messages to the applications without a preceding invade. This allows the application to directly process resource update messages similar to the blocking invade by changing the number of cores and setting the pinning appropriately. Then, the constraints are forwarded to the RM, but without waiting for the reply message with the invaded resources.

  In contrast to the blocking-invade call, a message including the constraints which describe the changing requirements is sent to the RM with the control flow directly given back to the application after the message was enqueued. Thus, we can overcome an idling of cores.

- **reinvade_blocking()** and **reinvade_nonblocking()**
  With the standard invade calls, constraints are always specified and forwarded to the RM. This leads to overheads of, first of all, serializing the constraint data and, secondly, evaluating and possibly optimizing the resource distribution on the RM side. However, if the resource requirements do not change, we avoid this serialization and optimization procedure with the *reinvade* interfaces. Here, the RM assumes that the same constraints which were previously forwarded to the RM should be used for optimization. Even if the current resource requirement did not change for application A, a change in resource requirements of application B can result in a change of resources for the application A.

- **retreat()**
  To hand back the currently used resources, a *retreat* can be executed by the application. Then, all resources except a single one is released with the remaining thread continuing execution of the application's master thread.

8.3 Invasive resource manager

The content and structure of this section is related to our work [SRNB13b] which is currently under review. A separate process runs in the background on one thread without pinning and executes the resource manager (RM). The task of the resource manager is then the optimization of the resource distribution and is based on the information provided by the applications via constraints. Such constraints can be e.g. scalability graphs, workload and range constraints, see Sec. 8.2.1. For sake of clarity, Table 8.1 gives an overview of the symbols used in this and the upcoming section.

Realization

The RM aims at optimizing the core-to-application assignment stored in the vector $\vec{C}$. Here, each entry represents the association of the $R = |\vec{C}|$ physical cores to the applications. The

Symbol	Description
R	Number of system-wide available computing resources
N	Number of concurrently running processes
$\vec{A}$	List of running applications or MPI processes
ϵ	Placeholder for "no application"
$\vec{C_r}$	State of resource assignments to applications
$\vec{D_i}$	Optimal resource distribution assigning D_i cores to application A_i
$\vec{P_i}$	Optimization information (scalability graphs, e.g.) for application i
$\vec{T_i}$	Optimization targets (throughput, energy, etc.) for each application
$\vec{G_i}$	Number of resources currently assigned to application i
$\vec{F_i}$	List of free resources
$\vec{W_i}$	Workload for application i
$T(c)$	Throughput for c cores
$S_i(c)$	Scalability graph for application i.

Table 8.1: (source: [SRNB13b]) Overview of the symbols which are used in the data structures of the resource manager.

application id is stored to $\vec{C_i}$ if core i is assigned to the application. In case of no core assignment, ϵ is used as a placeholder.

Scheduling information

Here, we describe our algorithm which optimizes the resource distribution based on the constraints provided by the applications. Again, let R be the amount of system-wide available compute resources. Further, let N be the amount of concurrently running applications, ϵ a marker for a resource not assigned to any application and $\vec{A}$ a *list of identifiers* of concurrently running applications, with $|\vec{A}| = N$. Then, we distinguish between management data inside the RM: uniquely *per-application* and *system-wide* data.

Per-application data: For each application $\vec{A_i}$, there is a $\vec{P_i}$ storing the currently specified constraints which were previously send to the RM via a (non-)blocking invade. The RM uses these constraints for optimizations, depending on the desired optimization targets which are discussed in Section 8.4.

System-wide data: The system-wide management data is defined with the current resource assignment $\vec{C}$ and an optimization target. Such optimization targets e.g. request a maximization of the application throughput or for future applications the minimization of energy consumption. Then,

$$\vec{C} \in (\{\epsilon\} \cup \vec{A})^R,$$

is the current state on the resource assignment. This assigns each compute resource uniquely to either an application $a \in \vec{A}$ or to none ϵ. Then an optimization target is given e.g. by the *optimal resource distribution*

$$\vec{D} \in \{0, 1 \ldots, R\}^N.$$

Here, each entry $\vec{D_i}$ stores the number of cores which are assigned to the i-th application $\vec{A_i}$.

We further demand

$$\sum_i \vec{D}_i \leq R \tag{8.1}$$

to *avoid oversubscription* of these resources. This avoids assignment of more resources than there are available on the system. The resource collision itself is avoided by assigning the resources via the vector $\vec{C}$. Here, each core can be assigned to only a single application. Cores which are currently assigned to an application are additionally stored in a list for releasing them without a search operation on $\vec{C}$.

Optimization loop

A loop is used inside the RM which successively optimizes the resource distribution. Here, the resource distribution is updated based on the constraints. Further, the *current resource distribution* $\vec{C}$ is optimized towards the optimal *target resource distribution* $\vec{D}$. The optimization loop can be separated into three parts:

- *Computing target resource distribution* $\vec{D}$*:*
  New parameters for computing the target distribution are made available to the RM via constraints during setup, shutdown and invade messages. Here, the setup message yields the constraint with a single core, whereas the shutdown message includes a constraint which frees all cores.

  The optimization function is executed every time if a new one is available (setup), a constraint is updated (invade) or removed (shutdown). This optimization function is given by

  $$(\vec{D}^{(i+1)}, \vec{C}^{(i+1)}) := f_{optimize}(\vec{D}^{(i)}, \vec{C}^{(i)}, \vec{P}, \vec{T}) \tag{8.2}$$

  in its general form. Here, the vector of optimization targets is given in $\vec{T}$, e.g. targets such as throughput or load distribution. $\vec{P}$ contains the application constraints and the current distribution of cores to applications is given in $\vec{C}^{(i)}$.

  The computation of the target distribution with $f_{optimize}$ is further described in Section 8.4. Then, $\vec{D}^{(i+1)}$ contains the configuration of the computing cores to which the resource distribution has to be updated and the superscript (i) annotates the i-th execution of the optimization function.

  For applications which are sensitive to non-uniform memory access (NUMA), the target core-to-application can be beneficial and is also returned in $\vec{C}^{(i+1)}$. In the current implementation, this core-to-application assignment is not used and we continue using only the quantitative optimization given in $\vec{D}^{(i+1)}$.

- *Optimizing current resource distribution* $\vec{C}$*:*
  The RM successively updates the current resource distribution in $\vec{C}$ based on the theoretically optimal resource distribution $\vec{D}^{(i+1)}$. A direct release of a core from an application is only possible under special circumstances, e.g. if the core to be released is associated to the application which is currently executing the (re)invade call. Otherwise, a message is send to the application which has to release the core and the core may only be set as free in the resource manager if the application sends a corresponding response answer.

  Given the list $\vec{A}$ of applications, the *resource redistribution* is then optimized either by assigning additional cores or releasing cores for each application. Here,

  $$\vec{G}_i := |\{j | \vec{A}_i = \vec{C}_j, \forall j \in \{1, \dots, R\}\}|$$

is the number of resources which are currently assigned to application $\vec{A}_i$. We then use an iterative process over all applications $\vec{A}_i$ to redistribute the resources over all applications:

- $\vec{G}_i = \vec{D}_i$: No update
  No further change in resources is required.
- $\vec{D}_i < \vec{G}_i$: Release resources
  If less resources should be used by the application $\vec{A}_i$, a message with this new core constellations is send to the application. For non-blocking communication, the message is send immediately to the application and for blocking invades, the resources can be directly assumed to be released since the application directly updates the number of used threads after waiting for the message of the RM. For non-blocking simulations, the current resource distribution $\vec{C}$ is not yet updated to avoid assigning these resources to other applications.
- $\vec{D}_i > \vec{G}_i$: Add resources
  If additional resources should be assigned to the application, a search is executed in the list of free resources $\vec{F}$ with $\vec{C}_{\vec{F}_j} = \epsilon$. Then, it assigns up to $k \leq \vec{D}_i - \vec{G}_i$ resources to the application with

$$\forall j \in \{\vec{F}_1, \ldots, \vec{F}_k\} : \vec{C}_j := \vec{A}_i.$$

- *Client-side resource update messages:*
  Every time the RM receives a resource update message from one of the applications, further optimizations are executed since the change in resource utilization can lead to further possibilities of resource optimizations. This executes the previously described iterative process of resource optimizations.

8.4 Scheduling decisions

The content and structure of this section is related to our work [SRNB13b] which is currently under review. The optimized target resource distribution $\vec{D}$ is computed based on the previously introduced data structures $\vec{T}$ as the specified optimization target and $\vec{P}$ as an per-application specified information.

We further drop the core dependencies of our original optimization function (8.2), yielding the simplified optimization function

$$\vec{D}^{(i+1)} := f_{optimize}(\vec{D}^{(i)}, \vec{P}, \vec{T}). \tag{8.3}$$

Optimizations are then applied with the constraints of all applications depending on $\vec{P}$ and the per-application optimization target $\vec{T}$.

Requirements on constraints: The constraints which are forwarded by resource-aware applications to the RM are then kept in $\vec{P}$ with one entry for each application. Then, the RM schedules the available resources based on the optimization target and these constraints. Here, we distinguish between *local* (optimizing resources for a single application) and *global constraints* (optimizing resources for multiple applications).

Local constraints: With constraints given by the range of cores between 1 and the maximum number of cores, an application can request a particular range of cores. These constraints make is challenging to optimize concurrently running applications since no knowledge on their performance state for a changing number of resources can be inferred and we refer to such constraints as local ones.

Global constraints: Such constraints can be evaluated by the optimization function in a way which optimizes the resources targeting at a global optimum of all applications.

- *Application's workload*: If running similar applications, the *workload* can be used to schedule resources. This is due to the workload also used e.g. in load balancing. Our target function then redistributed R compute resources to each application A_i with

$$\vec{D}_i := \left\lceil \frac{R \cdot \vec{W}_i}{\sum_j \vec{W}_j} \right\rceil - \alpha_i, \quad \alpha_i \in \{0, 1\}.$$

  Hence, it assigns $\vec{D}_i$ resources to the application $\vec{A}_i$. To avoid over-subscription (see Eq. (8.1)), α has to be chosen in a particular way.

  With the assigned number of resources $\vec{D}_i$, the problem can be reformulated to a scalability optimization. Here, we assume that each application has a perfect strong scalability $S(c)$ for c cores within the range $[1; \vec{D}_i]$ and no performance gain beyond $\vec{D}_i$ cores:

$$S(c) := \min(c, \vec{D}_i).$$

  We can then assign the cores to the applications by adding a core until this leads to no further gain in performance. Obviously, this approximation with a scalability graph (explained in the next paragraph) is only an approximation of the real scalability graph. Using real scalability graphs provided by application are discussed next as an alternative.

- *Application's scalability graph*: Here, we assume that each application messages its overtime changing *strong scalability graph* to the RM. There is a linear dependency of the applications scalability and the workload throughput.

  For a given number of cores c, the throughput $T(c)$ represents the fraction of time to compute a solution for a fixed problem size $w = \vec{W}_i$. Next, we consider the throughput improvement for a changing number of cores with the baseline given at the throughput of a single core: $\frac{\frac{w}{T(c)}}{\frac{w}{T(1)}} = \frac{T(1)}{T(c)} =: S(c)$. This yields the relation to the scalability graph of strong scalability $S(c)$. Hence, we can use the scalability graph as an optimization hint for the application's throughput. Furthermore, we can use a scalability graph to optimize for the theoretical global maximum throughput even among heterogeneous applications due to the normalization $S(1) = 1$ for a single core.

  Let the scalability graph $S_i(c)$ be provided by application $\vec{A}_i$. We further require a strictly monotonously increasing behavior

$$S_i(c) - S_i(c-1) > 0 \ ,$$

  as well as a concavity

$$S_i(c+1) - S_i(c) \leq S_i(c) - S_i(c-1).$$

  This also assumes that no super-linear speedups are possible.

  The global throughput is then maximized by searching the most efficient resource-to-application combination in $\vec{D}$. This yields a multi-variate maximization problem in $\vec{D}_j$ for our optimization target

$$\max_{\vec{D}} \left(\sum_i S_i(\vec{D}_j) \right) . \tag{8.4}$$

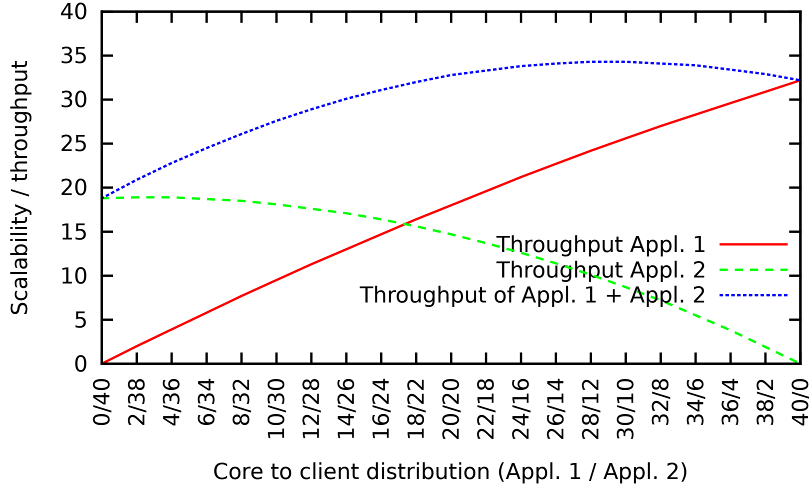


Figure 8.2: Two scalability graphs are given. The one for application A has an increasing number of cores from left to right and the one for application B has an increasing number of cores from right to left. The maximum throughput searches for the resource allocation to both application which maximizes the overall throughput [SRNB13a].

Again, we avoid over-subscription with the side constraint $\sum_j \vec{D}_j \leq R$.

Fig. 8.2 shows an example of such an optimization. Two applications are considered which provide a scalability graph. The graph with the red solid line represents the scalability of the first application with increasing number of resources from left to right and the green dashed line represents the scalability graph of the second application with the numbers of resources increasing from right to left.

To optimize the resource distribution for our "maximizing throughput" optimization target, the optimal point is given by the maximum of the sum of both throughput graphs.

However, we can use our requested properties of strictly monotonously increasing and concave scalability graphs. This allows us to solve the maximization problem for an arbitrary number of applications with an iterative gradient method which is related to the steepest descent solver [FP63] and assuming that there is only one optimum, the global optimum.

Initialization: The iteration vector $\vec{B}^{(k)}$ for the k-th iteration is introduced which assigns $\vec{B}_i$ computing cores to each application i. We start with $\vec{B}^{(0)} := (1, 1, \ldots, 1)$ which assigns each application a single core at the start. This is required since each application demands for at least a single core on which it is executed.

Iteration: We then compute the *throughput improvement* for each application i, for a single core additionally assigned to it:

$$\Delta S_i := S_i(\vec{B}_i + 1) - S_i(\vec{B}_i). \tag{8.5}$$

Then, the application n, which yields the maximum throughput improvement is given by $\Delta S_n := \max_j \{\Delta S_j\}$.

We can update the resource distribution for the $k+1$-th iteration by

$$\vec{B}^{(k+1)} := \vec{B}^{(k)} + \delta_{i,n} \tag{8.6}$$

with the Kronecker symbol δ.

Stopping criterion: If all resources are distributed ($\sum_i \vec{B}_i^{(k)} = R$), we can stop our optimization and the last iteration vector $\vec{B}$ contains the optimized target resource distribution for $\vec{D}^{(k+1)}$.

8.5 Invasive programming patterns

Using Invasive Computing on shared-memory systems requires implementation of invasive interfaces at particular positions in the code. This section presents the invasive programming pattern for iteration-based DAMR simulations and shows required extensions for simulations which rely on cached thread associativity.

8.5.1 Iteration-based simulation

We assume that our application uses an iterative setup (initial refinement) and an iterative time-stepping scheme (simulation) with the pseudo code given in the left column of Table 8.2.

Standard simulation steps	Invasive simulation steps
;	**setup**();
gridRefinement = True	gridRefinement = True
while gridRefinement:	while gridRefinement:
;	**invade_nonblocking**([**constr**.]);
gridRefinement = setupGrid();	gridRefinement = setupGrid();
for i in timesteps:	for i in timesteps:
;	(**re**)**invade_**(**non**)**blocking**([**constr**.]);
timestepAndAdaptivityTraversals();	timestepAndAdaptivityTraversals();
;	**shutdown**();

Table 8.2: Comparison of *pseudo code* for our simulation based on a time stepping and setup loop (left) with an invasified version (right). This invasified version only works for code without owner-compute scheme.

An extension with our Invasive Computing API then requires the following interfaces:

- **setup():**
  The setup routine registers the program at the invasive RM. In case of no resource available, this call blocks until at least one resource can be assigned to the application.

- **invade_(non)blocking(constraints):**
  Different variants of the invasive command space can be used for updating the resources with the forwarding of constraints.

- **reinvade_(non)blocking():**
  Different variants of the reinvade are used for updating the resources without updating the already forwarded constraints.

- **shutdown():**
  After the simulation is finished, the RM is informed about the shutdown of the application, making the resources available to other applications.

8.5.2 Iteration-based with owner-compute

Applying the Invasive Computing programming pattern to our owner-compute scheme (see Section 5.6.1) can lead to invalid cluster-tree traversals. The reason can be found in the caching of the thread id range on each node owning one of the leave nodes in the cluster tree. With dynamically changing numbers of active threads in a program, this thread id can be associated to a thread which does not exist anymore. In case of a shrinking number of resources, this leads to missing traversals of nodes in the cluster tree. For a growing number of resources, one or more cores remain unused. Hence, we additionally require to account for these cached thread ids and show the required extension in the pseudo code presented in Table 8.3.

```
setup();

for i in timesteps:
  // return new number of threads
  new_number_of_threads =
    (re)invade_(non)blocking([constraints], postponeThreadUpdate);

  if new_number_of_threads < old_threads:   // shrink
    updateCachedThreadIDs(new_number_of_threads);
    updateNumberOfRunningThreads(new_number_of_threads);

  elif new_number_of_threads > old_threads: // grow
    updateNumberOfRunningThreads(new_number_of_threads);
    updateCachedThreadIDs(new_number_of_threads);

  old_threads = new_number_of_threads;

  timestepAndAdaptivityTrav();

shutdown();
```

Table 8.3: Pseudo code for changing number of threads with an owner-compute scheme.

Here, the function *(re)invade_(non)blocking* also returns the new number of threads and the additional constraint *postponeThreadUpdate* requests, that the number of threads are not directly updated with the invade call. Instead, an additional function *updateNumberOfRunningThreads* is provided which has to be executed for updating the threads based on the last reinvade call.

In case that the number of threads was decreased, we then update the cached thread ids in the cluster tree (*updateCachedThreadIDs*), followed by updating the number of actively running threads including their pinning. For a growing number of threads, we can directly update the number of running threads followed by updating the cached thread ids.

8.6 Results

This section presents several studies in the context of our formerly described Invasive Computing implementation for HPC shared-memory systems.

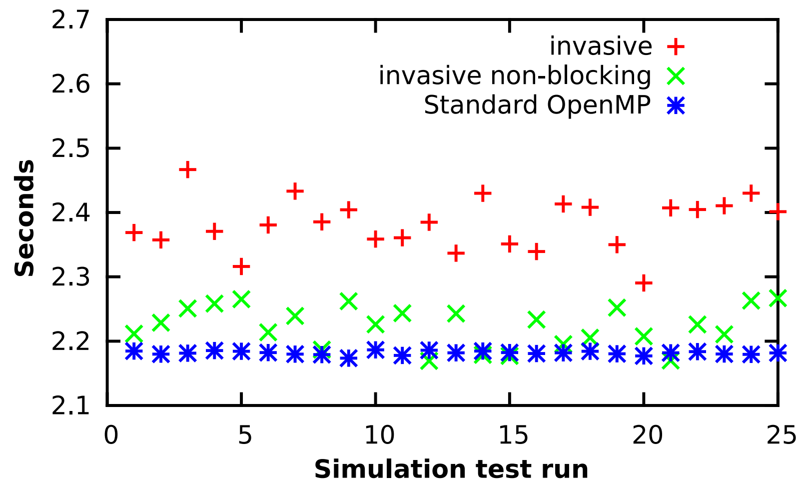


Figure 8.3: Plot of the seconds taken for each of 25 simulation runs and for each application type: non-invasive with OpenMP, invasive with blocking communication to the RM and invasive with non-blocking communication.

8.6.1 Micro benchmarks of invasive overheads

We start with overhead measurements of our Invasive Computing interfaces comparing the blocking with the non-blocking (re)invade calls. For a measurement in a realistic environment, we use the simulation framework presented in the previous part and start several small shallow water simulations on a grid which is regularly refined with 128 cells. The invasive versions are realized with extensions to OpenMP.

We analyzed the invasive overheads with three different versions:

(a) *invasive*: Here, we use blocking communication to the RM.

(b) *invasive non-blocking*: This uses non-blocking communication and asynchronous communication to the RM.

(c) *OpenMP*: No invasive commands are used and the parallelization is only accomplished with OpenMP parallelization.

The invasive versions of the simulations send one invade request to the resource manager between each time step. The measurements of the overall application's runtime were conducted for 25 executions of a single application. It is sufficient to consider a single application, since we are currently only interested in the overheads of the RM. The required simulation times on a single core are printed in Fig. 8.3. The blocking version of the invasion leads to an overhead of 15% in average compared to the non-invasive version. However, this invasion is compensated by the non-blocking version, yielding a robust improvement compared to the blocking one and results in an overhead of only 5% in average compared to the non-invasive version. Hence, we continue running our invasive benchmarks with the non-blocking version.

8.6.2 Dynamic resource redistribution with scalability graphs

Our main motivation for Invasive Computing in the area of HPC was driven by simulations on dynamically adaptive grids. Next, we use our shallow water simulation to show the functionality of dynamic resource distribution based on scalability graphs.

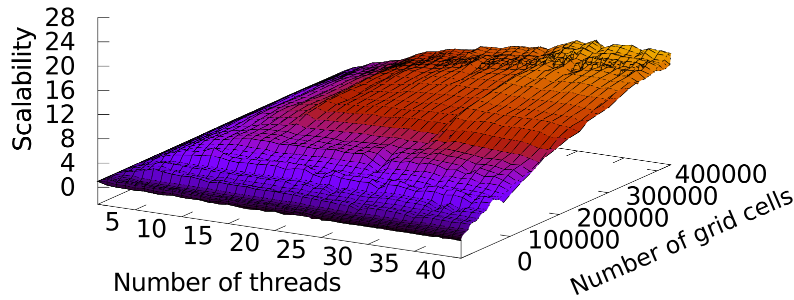


Figure 8.4: Scalability plot for different number of cells of a shallow water simulation on a dynamically adaptive grid [BBS12].

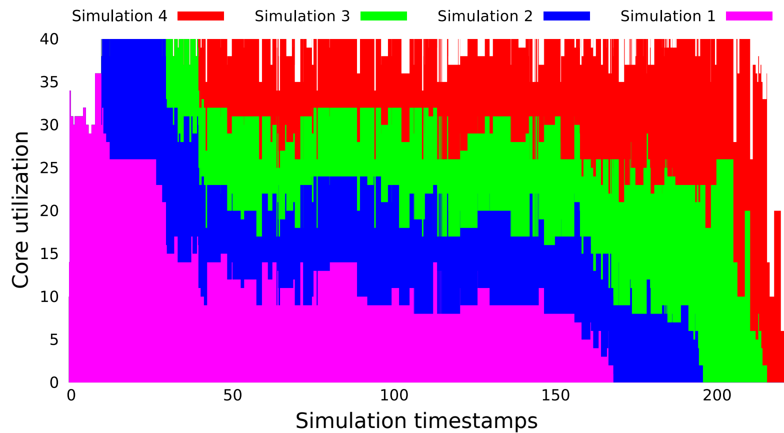


Figure 8.5: Dynamic resource redistribution for four shallow water simulations. The white gaps represent idling resources until another application invades them [BBS12].

The scalability graphs we use are determined in advance with the execution of simulations for a few time steps on different cores and different refinement parameters. An example of such scalability graphs with different grid cells is given in Fig. 8.4, it was computed on the platform Intel, a 40-core shared-memory system (see Appendix A.2). Given a particular number of grid cells, e.g. based on the current number of cells used in the dynamically adaptive simulation, the scalability graph can be extracted by a slice.

We can then optimize the resource distribution with the precomputed scalability graphs: Given the number of current grid cells, we determine the scalability graph which is closest to the given number of grid cells. Such a scalability graph is one slice of the plot in Fig. 8.4. If the scalability graph is different to the previous one, it is forwarded to the RM. We can then run multiple shallow water simulations in parallel without oversubscribed resources and the resources optimized by the RM with the scalability graph.

For the benchmark scenario, we use four identical shallow water simulations. Each one is started slightly delayed to the previous one with 10, 20, and 10 seconds. The simulation mesh is initialized with an initial refinement depth of 4 and 14 additional adaptivity levels. The splitting threshold is set to 8192.

Based on the scalability graphs, the dynamic resource redistribution of concurrently executed applications which are started shortly delayed is presented in Fig. 8.5. The resource distribution is based on the scalability optimization algorithm presented in Section 8.4.

Having a closer look on the execution times, the invasive version took 266 seconds for its execution. We compare this execution time with an OpenMP parallelization which executes the simulations one after another. This resulted in an execution time of 521 seconds. Our

approach is also competitive to a TBB parallelized version which starts the simulation as soon as it is enqueued, taking 491 seconds for the execution. Here, the Invasive Computing shows a clear benefit with an improvement of 49% compared to the OpenMP parallelization and 46% if comparing it to the TBB parallelized execution.

8.6.3 Invasive Tsunami parameter studies

To show the real potential of Invasive Computing in the context of an application with an iterative time-stepping scheme, we use a scenario of several Tsunami parameter studies in parallel. Here, we use the Tohoku Tsunami simulation which was presented in Sec. 6.3. Here, the simulation first loads the bathymetry datasets, then preprocesses it to a multi-resolution format and initializes the simulation grid. Finally, the wave propagation is simulated.

As a parameter study, we executed five simulations with slightly different adaptivity parameters. Since starting all simulations at the same time does not yield a realistic HPC scheduling, the enqueuing of the executions of each of these simulations is delayed with the seconds given in the following vector: $(10, 15, 15, 15)$.

We challenge OpenMP and TBB with our Invasive Computing approach, yielding three different parallelization methods:

- *OpenMP:*
  With the standard OpenMP environment, the execution of multiple applications in parallel on all available cores would result in resource conflicts e.g. by frequent preemption of applications and hence shared caches. Therefore, we have to execute them one after another.

- *TBB:*
  We further used the TBB parallelization of our simulation. One of the TBB features is e.g. that in case of an idling worker thread, TBB hands back the control to the operating system via the *yield* system call instead of doing a busy waiting. Such a busy waiting results in a core utilization, hence the core cannot be used exclusively by another application.

- *Invasive Computing:*
  For the invasive execution, we use the non-blocking invade to forward the constraints to the RM. Regarding the constraints, we avoided the scalability graphs since our current implementation requires to determine them in advance of the simulation. Here, we use the workload constraint and the number of cells involved in the current simulation as the workload.

The overall execution time for several scenarios, each one based on five simulations, is depicted in Fig. 8.6 for different problem sizes. These problem sizes of the simulations in each scenario are increased from left to right by increasing the initial refinement depth. The adaptivity refinement depth of $(10, 10, 8, 8, 7)$ was used for the five simulations.

For smaller problem sizes, TBB yielded optimizations similar to our Invasive Computing approach. However, these problem sizes were only considered for testing purpose and do not yield any practical relevant data. For larger simulations as they were considered for the analysis of our Tsunami simulations in Section 6.3, TBB looses its performance improvement compared to the OpenMP non-invasive execution.

We further depicted the results of the scenarios E and F with a larger problem size and a linear scaling in the right image in Fig. 8.6. Here, our Invasive Computing approach leads

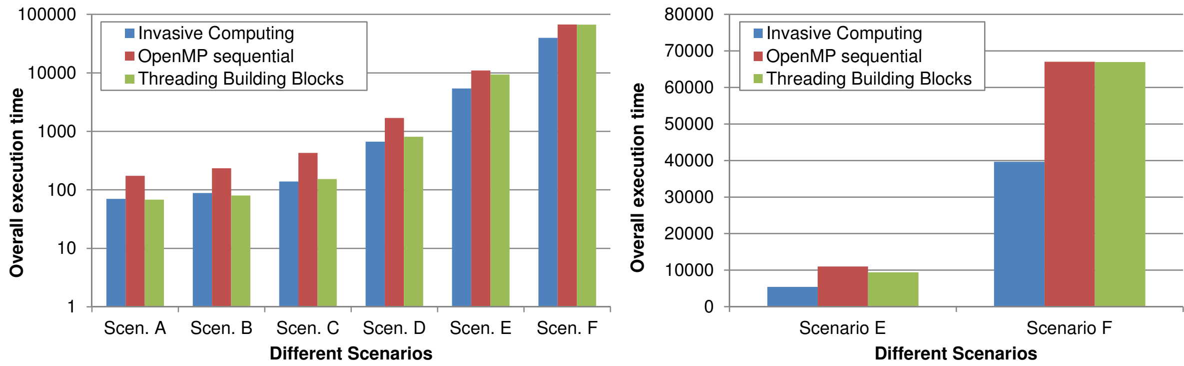


Figure 8.6: Left image: execution of several Tsunami parameter studies with a growing problem size from left to right. Right image: benchmark scenarios E and F visualized with linear scaling.

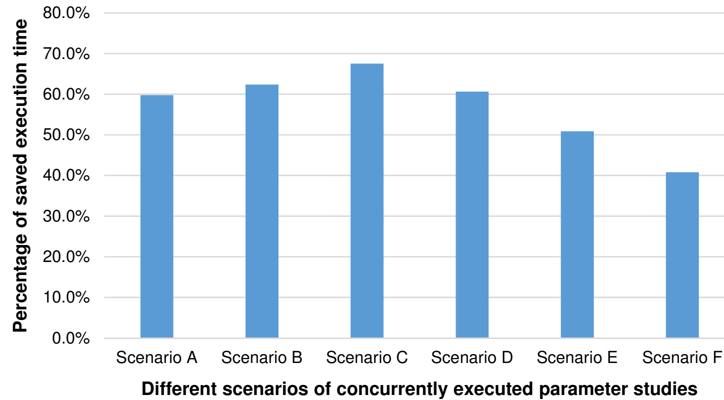


Figure 8.7: Performance improvements of invasive benchmark scenarios A to F compared to OpenMP consecutive execution.

to a robust performance improvement of 45% compared to a non-invasive parallelization with standard OpenMP and TBB.

Assuming a simulation executed with a sufficiently large workload, a close-to-linear scalability can be reached, see Sec. 5.7. Since the performance improvements of Invasive Computing rely on a non-linear scalability of applications and furthermore a scalability which is changing over the runtime, we further analyzed the behavior of the performance improvements of the invasive execution from small to large simulation scenarios. We plotted the reduction in runtime in percentage comparing the invasive to the non-invasive execution in Fig. 8.7. Here, we can see that the performance improvements with Invasive Computing are less for larger simulation scenarios. We account for that by the improved scalability of larger workloads.

We close this chapter by relating our results to a concrete example for the $da = 22/0$ Tsunami simulation from Sec. 6.3. This simulation has an overall execution time of 2557.11 seconds and we assume, that the execution time of scenarios E has the closest similarity regarding the runtime. Then, assuming concurrently executed parameter studies of this workload, the improvement in efficiency is above 50%.

9 Conclusion and outlook

Despite that the Invasive Computing paradigm was originally developed for TCPAs, it showed its applicability also in areas of MPSoCs as well as HPC shared-memory systems.

For the Invasive MPSoC System with its heterogeneous computing components, we selected the multigrid solver as a representative algorithm and presented the challenges for extensions with Invasive Computing in X10. With the Invasive Computing interfaces offered by invadeX10, this considerably improves the programmability of Invasive Computing on systems with no inter-tile cache coherency.

Applying the Invasive Computing paradigm on shared-memory HPC systems comes with benefits, but also drawbacks: An obvious drawback is the fact, that applications have to provide information on their current resource requirements via the constraint system. Deriving resource requirements such as scalability graphs is challenging, however the computational workload yields an alternative. The application developer also has to consider the possibly changing number of computing resources. For applications which make use of cached thread affinities such as those shown in Sec. 8.5.2, another programming pattern has to be used. Besides these drawbacks, we proofed that Invasive Computing in HPC can lead to severe performance benefits if executing several resource-competing applications with Invasive Computing. We gained an improvement of 45% in execution time for Tsunami parameter studies compared to other parallelization models such as TBB or OpenMP.

With our large-scale simulations using dynamically adaptive mesh refinement on distributed-memory systems (see Sec. 5.12) which is e.g. required to simulate wave propagations on earth scale (see Sec. 6.5) and with multi-layers (see Sec. 6.6), we expect even more changing resource requirements for each application, hence more optimization possibilities with Invasive Computing. This demands for Invasive Computing also on HPC distributed-memory systems. Contrary to the hard- and software development for the Invasive MPSoCs, Invasive Computing on HPC large-scale systems has be based on a standard parallelization model for distributed-memory systems to allow the invasification of existing codes. However, this yields several challenges such as modifications of the MPI communicator, replication of the program instance to new computational resources and the synchronization of the simulation state. With such Invasion-enabled large-scale simulations on distributed-memory systems, we expect even more improved efficiency.

PART V

SUMMARY

In this work, we focused on the efficient implementation of dynamically adaptive mesh refinement for optimizing the execution of numerical simulations. In particular, we considered wave-propagation dominated problems such as Tsunami simulations. We started with an introduction to the very basics of a discontinuous Galerkin (DG) discretization scheme which yielded our communication requirements with edge- and vertex-based communications for the considered DG solvers. For the discretization, we decided to use triangles as grid primitives and use a grid which is generated by the triangle-bisecting Sierpiński SFC.

Regarding the serial implementation, we first gave an introduction to existing work on stack- and stream-based simulations. Then, we introduced clear interfaces for the communication and data access to run DG simulations on dynamically adaptive grids. A code generator is developed which does not only lead to an efficient method to generate traversal code with interfaces tailored to the user requirements, but also yields optimizations with parameter unrolling, avoiding most of the if-branching instructions. Furthermore, we avoid obsolete access of memory e.g. by separation of structure, cell-data and adaptivity-state stacks. An automaton table considers the propagation direction of adaptivity information and is further used for optimizations. With a prospective stack reallocation, a good balance between memory requirements and frequent stack reallocation was presented.

For the parallelization, we first developed a run-length encoded meta communication information with its connectivity information implicitly updated by adaptivity markers written on the communication stacks. We use zero-length encodings for vertices which are not already represented by the edge-based meta information. Such a run-length encoded (RLE) meta communication information further allows an efficient block-wise communication. Then, we introduced clustering based on SFC-induced cell intervals and the aforementioned RLE communication meta information. Our cluster generation is based on tree-splits and -joins and we infer the new meta information after splits and joins implicitly via the elements stored on the stacks during a traversal. Our software and communication methods lead to a direct applicability of different parallelization models. We developed different cluster generation and scheduling methods on shared- and distributed-memory systems. On a 40-core shared-memory system, the owner-compute scheme yielded the best results due to the improved NUMA-domain awareness. Besides the RLE meta encoding, we developed further algorithmic optimizations with one of them the skipping of already conforming clusters. This results in a robust performance improvement, also compensating the additionally required conforming grid traversals required due to the domain decomposition. On distributed-memory systems the clustering leads to efficient block-wise communication as well as cluster-based data migration. With a dynamically adaptive grid in each time step, the strong scalability measured with a baseline at 256 cores is over 60% and the weak scalability is over 90% on 8192 cores.

Dynamically changing grids also lead to dynamically changing resource requirements. To cope with these changing resource requirements, we achieved a dynamic resource allocation by

extending standard shared-memory HPC parallelization models with the Invasive Computing paradigm. We presented a resource manager on a shared-memory system and the required extensions to the simulation framework. This allows an optimization of resources based on application-specific information. We conducted several benchmarks with a dynamical redistribution of the computing resources among concurrently running applications on shared-memory systems, resulting in throughput improvements for concurrently executed Tsunami simulations of more than 43%.

A
Appendix

A.1 Hyperbolic PDEs

A.1.1 Gauss Lobatto Points

The following table gives examples of basis functions based on Gauss-Lobatto points and their nodal points up to order 2.

	Polynomial	Nodal point
Degree 0:	$\varphi_0(x,y) := 1$	$(\frac{1}{3},\frac{1}{3})$
Degree 1:	$\varphi_0(x,y) := 1 - x - y$	$(\frac{1}{2},\frac{1}{2})$
	$\varphi_1(x,y) := x$	$(\frac{1}{2},0)$
	$\varphi_2(x,y) := y$	$(0,\frac{1}{2})$
Degree 2:	$\varphi_0(x,y) := 1 - 3x + 2x^2 - 3y + 4xy + 2y^2$	$(0,0)$
	$\varphi_1(x,y) := 4x - 4x^2 - 4xy$	$(\frac{1}{2},0)$
	$\varphi_2(x,y) := -x + 2x^2$	$(1,0)$
	$\varphi_3(x,y) := 4y - 4xy - 4y^2$	$(0,\frac{1}{2})$
	$\varphi_4(x,y) := 4xy$	$(\frac{1}{2},\frac{1}{2})$
	$\varphi_5(x,y) := -y + 2y^2$	$(0,1)$

A.1.2 Jacobi polynomials

The orthogonal Jacobi polynomials on triangle basis up to degree 2 are given in the following Table. Compared to the Gauss-Lobatto Points, the Jacobi polynomials are constructed hierarchically.

	Polynomial
Degree 0:	$\varphi_0(x,y) := \sqrt{2}$
Degree 1:	$\varphi_0(x,y) := \sqrt{2}$
	$\varphi_1(x,y) := -2 + 6y$
	$\varphi_2(x,y) := 2\sqrt{3}(2x - 1 + y)$
Degree 2:	$\varphi_0(x,y) := \sqrt{2}$
	$\varphi_1(x,y) := -2 + 6y$
	$\varphi_2(x,y) := (1 - 8y + 10y^2)\sqrt{6}$
	$\varphi_3(x,y) := 2\sqrt{3}(2x - 1 + y)$
	$\varphi_4(x,y) := 3\sqrt{2}(-1 + 5y)(2x - 1 + y)$
	$\varphi_5(x,y) := (1 - 2y + y^2 - 6x + 6xy + 6x^2)\sqrt{30}$

A.1.3 Mass Matrix

For nodal basis functions of degree 1 and 2 with Gauss-Lobatto points, the inverse mass matrices are given by

$$\begin{bmatrix} 18 & -6 & -6 \\ -6 & 18 & -6 \\ -6 & -6 & 18 \end{bmatrix} \begin{bmatrix} 72 & -3 & 12 & -3 & 12 & 12 \\ -3 & \frac{39}{2} & -3 & -\frac{27}{4} & -\frac{27}{4} & 12 \\ 12 & -3 & 72 & 12 & -3 & 12 \\ -3 & -\frac{27}{4} & 12 & \frac{39}{2} & -\frac{27}{4} & -3 \\ 12 & -\frac{27}{4} & -3 & -\frac{27}{4} & \frac{39}{2} & -3 \\ 12 & 12 & 12 & -3 & -3 & 72 \end{bmatrix}.$$

Using orthonormal triangle basis functions based on normalized Jacobi Polynomials, the inverse mass matrix is identical to the *identity matrix* for arbitrary degree.

A.1.4 Stiffness matrices

For Gauss-Lobatto nodal points, this leads to stiffness matrices each with a single zero-row for degree 1:

$$\begin{bmatrix} -1/6 & -1/6 & -1/6 \\ 1/6 & 1/6 & 1/6 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1/6 & -1/6 & -1/6 \\ 0 & 0 & 0 \\ 1/6 & 1/6 & 1/6 \end{bmatrix}$$

Computing the matrices x and y for degree 2, this still leads to almost dense matrices.

$$\begin{bmatrix} -1/15 & -1/10 & 1/30 & -1/10 & 1/30 & 1/30 \\ 1/10 & 0 & -1/10 & 2/15 & -2/15 & 0 \\ -1/30 & 1/10 & 1/15 & -1/30 & 1/10 & -1/30 \\ 1/30 & -2/15 & 1/30 & -\frac{4}{15} & -\frac{4}{15} & -1/15 \\ -1/30 & 2/15 & -1/30 & \frac{4}{15} & \frac{4}{15} & 1/15 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1/15 & -1/10 & 1/30 & -1/10 & 1/30 & 1/30 \\ 1/30 & -\frac{4}{15} & -1/15 & -2/15 & -\frac{4}{15} & 1/30 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1/10 & 2/15 & 0 & 0 & -2/15 & -1/10 \\ -1/30 & \frac{4}{15} & 1/15 & 2/15 & \frac{4}{15} & -1/30 \\ -1/30 & -1/30 & -1/30 & 1/10 & 1/10 & 1/15 \end{bmatrix}.$$

The matrices created for the orthonormal basis function have a higher sparsity pattern. Stiffness matrices for degree 1 are given by

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 2\sqrt{3}\sqrt{2} & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 3\sqrt{2} & 0 & 0 \\ \sqrt{3}\sqrt{2} & 0 & 0 \end{bmatrix}$$

and for degree 2

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2\sqrt{3}\sqrt{2} & 0 & 0 & 0 & 0 & 0 \\ 4 & 5\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3\sqrt{10} & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 3\sqrt{2} & 0 & 0 & 0 & 0 & 0 \\ -4/3\sqrt{3} & 10/3\sqrt{6} & 0 & 0 & 0 & 0 \\ \sqrt{3}\sqrt{2} & 0 & 0 & 0 & 0 & 0 \\ 2 & 5/2\sqrt{2} & 0 & 5/2\sqrt{3}\sqrt{2} & 0 & 0 \\ 2/3\sqrt{15} & -1/6\sqrt{30} & 0 & 3/2\sqrt{10} & 0 & 0 \end{bmatrix}.$$

Those matrices have clearly a sparser layout and are thus better suited for computation considering the stiffness matrices only so far.
However, using modal basis functions, they have to be transferred to nodal basis functions. Using Gauss-Lobatto nodal points and 1st degree basis functions, this results in the following

three matrices:

$$\begin{bmatrix} \sqrt{2} & -2 & -2\sqrt{3} \\ \sqrt{2} & -2 & 2\sqrt{3} \\ \sqrt{2} & 4 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 2/3\sqrt{3} & 2/3\sqrt{3} & 2/3\sqrt{3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 1/3\sqrt{3} & 1/3\sqrt{3} & 1/3\sqrt{3} \end{bmatrix}$$

and to the following matrices for 2nd degree

$$\begin{bmatrix} \sqrt{2} & -2 & \sqrt{6} & -2\sqrt{3} & 3\sqrt{2} & \sqrt{30} \\ \sqrt{2} & -2 & \sqrt{6} & 0 & 0 & -1/2\sqrt{30} \\ \sqrt{2} & -2 & \sqrt{6} & 2\sqrt{3} & -3\sqrt{2} & \sqrt{30} \\ \sqrt{2} & 1 & -1/2\sqrt{6} & -\sqrt{3} & -9/4\sqrt{2} & 1/4\sqrt{30} \\ \sqrt{2} & 1 & -1/2\sqrt{6} & \sqrt{3} & 9/4\sqrt{2} & 1/4\sqrt{30} \\ \sqrt{2} & 4 & 3\sqrt{6} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2/3\sqrt{3} & 0 & 2/3\sqrt{3} & 2/3\sqrt{3} & 0 \\ -1/4\sqrt{2} & 0 & -1/4\sqrt{2} & \sqrt{2} & \sqrt{2} & 1/2\sqrt{2} \\ -\frac{3}{20}\sqrt{30} & 0 & \frac{3}{20}\sqrt{30} & -1/5\sqrt{30} & 1/5\sqrt{30} & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ -1/6\sqrt{6} & -2/3\sqrt{6} & -1/6\sqrt{6} & 0 & 0 & 1/3\sqrt{6} \\ 0 & 1/3\sqrt{3} & 0 & 1/3\sqrt{3} & 1/3\sqrt{3} & 0 \\ -1/2\sqrt{2} & 0 & 1/4\sqrt{2} & 0 & \sqrt{2} & 1/4\sqrt{2} \\ -1/15\sqrt{30} & 2/15\sqrt{30} & 1/12\sqrt{30} & 0 & 1/5\sqrt{30} & -\frac{1}{60}\sqrt{30} \end{bmatrix}.$$

A.1.5 Flux matrices

Only a subset of conserved quantities is involved in the flux computation, see the following matrices for GL nodal points of order 2:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

Thus each row selects the conserved quantities for each nodal point. For modal representation, a similar approach to the stiffness matrices has to be used.

After flux computations, the flux updates only involve the nodal points of the edge for which the flux was computed for. Examples for such matrices generated for flux computations and Gauss Lobatto nodal points in the order of hypotenuse, right and left edge are given by

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -2/15\sqrt{2} & -1/15\sqrt{2} & 1/30\sqrt{2} \\ 0 & 0 & 0 \\ -1/15\sqrt{2} & -\frac{8}{15}\sqrt{2} & -1/15\sqrt{2} \\ 1/30\sqrt{2} & -1/15\sqrt{2} & -2/15\sqrt{2} \end{bmatrix} \begin{bmatrix} -2/15 & -1/15 & 1/30 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1/15 & -\frac{8}{15} & -1/15 \\ 0 & 0 & 0 \\ 1/30 & -1/15 & -2/15 \end{bmatrix} \begin{bmatrix} -2/15 & -1/15 & 1/30 \\ -1/15 & -\frac{8}{15} & -1/15 \\ 1/30 & -1/15 & -2/15 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

A.1.6 Butcher tableau

For RK with 2 stages, one possible tableau is

$a_{1,1} := 0$	$a_{1,2} := 0$
$a_{2,1} := \frac{1}{2}$	$a_{2,2} := 0$
$b_1 := 0$	$b_2 := 1$

yielding

$$\begin{array}{rcl} V_0 & := & U(t) \\ V_1 & := & V_0 + \Delta t(a_{1,1}D_1 + a_{1,2}D_2) = V_0 \\ D_1 & := & R(V_1) \\ V_2 & := & V_0 + \Delta t(a_{2,1}D_1 + a_{2,2}D_2) = V_0 + \Delta t\frac{1}{2}D_1 \\ D_2 & := & R(V_2) \\ U(t+\Delta t) & := & V_0 + \Delta t(b_1 D_1 + b_2 D_2) = V_0 + \Delta t D_2 \end{array} . \tag{A.1}$$

Another formulation for 2$^{\text{nd}}$ order accuracy is given by Heun's method, yielding the tableau

$a_{1,1} := 0$	$a_{1,2} := 0$
$a_{2,1} := 1$	$a_{2,2} := 0$
$b_1 := \frac{1}{2}$	$b_2 := \frac{1}{2}$

The tableau for RK3 [But64] yielding 3-rd order accuracy is given by

$a_{1,1} := 0$	$a_{1,2} := 0$	$a_{1,3} := 0$
$a_{2,1} := \frac{1}{2}$	$a_{2,2} := 0$	$a_{2,3} := 0$
$a_{3,1} := -1$	$a_{3,2} := 2$	$a_{3,3} := 0$
$b_1 := \frac{1}{6}$	$b_2 := \frac{2}{3}$	$b_3 := \frac{1}{6}$

The tableau for classical RK4 yielding 4-th order accuracy [SM03] is given by

$a_{1,1} := 0$	$a_{1,2} := 0$	$a_{1,3} := 0$	$a_{1,4} := 0$
$a_{2,1} := \frac{1}{2}$	$a_{2,2} := 0$	$a_{2,3} := 0$	$a_{2,4} := 0$
$a_{3,1} := 0$	$a_{3,2} := \frac{1}{2}$	$a_{3,3} := 0$	$a_{3,4} := 0$
$a_{4,1} := 0$	$a_{4,2} := 0$	$a_{4,3} := 1$	$a_{4,4} := 0$
$b_1 := \frac{1}{6}$	$b_2 := \frac{1}{3}$	$b_3 := \frac{1}{3}$	$b_4 := \frac{1}{6}$

A.1.7 Rotational invariance of Euler equations

Here, we present the rotational invariancy of the Euler equations [Tor01]: For an edge normal $\vec{n}_e$ pointing towards $(\cos(\alpha), \sin(\alpha))^T$, the rotation matrix $R(\alpha)_e$ is given by

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We test for rotational invariancy by putting the flux terms from Eq. (1.9) into the rotational invariancy formula given in Eq. 2.9 which yields for the left hand side

$$F(U) \cdot \vec{n}_e = \begin{pmatrix} \cos(\alpha)\, ru \\ \cos(\alpha)\left(ru^2 + p\right) \\ r\cos(\alpha)\, uv \\ \cos(\alpha)\,(E+p)\, u \end{pmatrix} + \begin{pmatrix} \sin(\alpha)\, rv \\ ru\sin(\alpha)\, v \\ \sin(\alpha)\left(rv^2 + p\right) \\ \sin(\alpha)\,(E+p)\, v \end{pmatrix} \tag{A.2}$$

and for the right hand side

$$R(\alpha)^{-1}F(R(\alpha)(U)) = R(\alpha)^{-1}F((\rho, \cos(\alpha)\rho u + \sin(\alpha)\rho v, -\sin(\alpha)\rho u + \cos(\alpha)\rho v, E)^T) \quad (A.3)$$

$$= \begin{pmatrix} r\left(\cos\left(\alpha\right)u + \sin\left(\alpha\right)v\right) \\ ru^2\cos\left(\alpha\right) + ru\sin\left(\alpha\right)v + \cos\left(\alpha\right)p \\ r\cos\left(\alpha\right)uv + rv^2\sin\left(\alpha\right) + \sin\left(\alpha\right)p \\ \left(E+p\right)\left(\cos\left(\alpha\right)u + \sin\left(\alpha\right)v\right) \end{pmatrix} \quad (A.4)$$

which is equal to $F(U) \cdot \vec{n}_e$ using basic trigonometric calculus.

A.2 Test platforms

We give a detailed description on the test platforms used in this thesis.

A.2.1 Platform Intel

The first platform denoted as *Intel* is based on a four socket system with each socket equipped with an Intel Xeon CPU (E7-4850@2.00GHz) with 10 cores per CPU and each core twice hyper threaded, resulting in 20 hyper threads.

Cache level	Size	Sharing information
L1	32kB	exclusive
L2	256kB	exclusive
L3	24MB	shared

Each CPU has its own memory controller assigned with 64GB of memory available via each controller. With 4 CPUs, this leads to 256 GB of main memory.

A.2.2 Platform AMD

We refer to the second platform as *AMD* and like to thank the Institute for Multiscale Simulation, Friedrich-Alexander Universität Erlangen-Nürnberg, for giving us access to their AMD cluster. This is based on 4 AMD Opteron(TM) Processors 6276, each one with 16 cores. On these CPUs, 2 cores share one FPU. The 16 cores are further separated in 2 modules, each module with its own last level cache and NUMA domain. The cache hierarchy then looks as follows

Cache level	Size	Sharing information
L1	16kB	exclusive
L2	2MB	shared by 2 cores
L3	6MB	shared by 8 cores

Each NUMA domain has 16GB of memory attached to each memory controller. This leads to 128GB available main memory.

A.2.3 Platform MAC Cluster

This cluster is based on 28 nodes, each one with a dual socket Intel SandyBridge-EP Xeon E5-2670 and 128 GB RAM and 8 cores per socket. Hence, up to 448 cores are available.

The secret to creativity is knowing how to hide your sources.

Albert Einstein (1879 - 1955)

Bibliography

[AK00] Franz Aurenhammer and Rolf Klein. Voronoi diagrams. *Handbook of computational geometry*, 5:201–290, 2000.

[AS98] Harold L Atkins and Chi-Wang Shu. Quadrature-free implementation of discontinuous Galerkin method for hyperbolic equations. *AIAA journal*, 36(5):775–782, 1998.

[Bad08] Michael Bader. Exploiting the locality properties of peano curves for parallel matrix multiplication. In *Euro-Par 2008–Parallel Processing*, pages 801–810. Springer, 2008.

[BBD+08] Peter Bastian, Markus Blatt, Andreas Dedner, Christian Engwer, Robert Klöfkorn, Markus Ohlberger, and Oliver Sander. A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework. *Computing*, 82(2-3):103–119, 2008.

[BBMZ12] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. *An x10 compiler for invasive architectures*. KIT, Fakultät für Informatik, 2012.

[BBS12] Michael Bader, Hans-Joachim Bungartz, and Martin Schreiber. Invasive Computing on High Performance Shared-memory Systems. In *Facing the Multicore-Challenge III*, volume 7686 of *Lecture Notes in Computer Science*, pages 1–12, September 2012.

[BBSV10] Michael Bader, Christian Böck, Johannes Schwaiger, and Csaba Vigh. Dynamically Adaptive Simulations with Minimal Memory Requirement-Solving the Shallow Water Equations Using Sierpinski Curves. *SIAM Journal on Scientific Computing*, 32(1):212–228, 2010.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[BCCD12] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering, and Colorin. *Scientific Programming*, 20(2), 2012.

[Beh05] Jörn Behrens. Multilevel optimization by space-filling curves in adaptive atmospheric modeling. 2005.

[Beh06] Jörn Behrens. *Adaptive atmospheric modeling: key techniques in grid generation, data structures, and numerical operations with applications*. Springer, 2006.

[BGG+08] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Georg Stadler, Eh Tan, Tiankai Tu, Lucas C Wilcox, and Shijie Zhong. Scalable adaptive mantle convection simulation on petascale supercomputers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 62. IEEE Press, 2008.

[BGLM11] Marsha J Berger, David L George, Randall J LeVeque, and Kyle T Mandli. The GeoClaw software for depth-averaged flows with adaptive refinement. *Advances in Water Resources*, 34(9):1195–1206, 2011.

[BGN00] P Brufau and P Garcia-Navarro. Two-dimensional dam break flow simulation. *International Journal for Numerical Methods in Fluids*, 33(1):35–57, 2000.

[BHK07] Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. deal. IIa general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):24, 2007.

[BL98] MJ. Berger and R. LeVeque. Adaptive mesh refinement for two-dimensional hyperbolic systems and the AMRCLAW software. *SIAM J. Numer. Anal*, 35:2298–2316, 1998.

[BNTB07] Pete Beckman, Suman Nadella, Nick Trebon, and Ivan Beschastnikh. SPRUCE: A system for supporting urgent high-performance computing. In *Grid-Based Problem Solving Environments*, pages 295–311. Springer, 2007.

[Bra07] Dietrich Braess. *Finite elements: Theory, fast solvers, and applications in solid mechanics*. Cambridge University Press, 2007.

[BRH+05] Jörn Behrens, Natalja Rakowsky, Wolfgang Hiller, Dörthe Handorf, Matthias Läuter, Jürgen Päpke, and Klaus Dethloff. amatos: Parallel adaptive mesh generator for atmospheric and oceanic simulation. *Ocean Modelling*, 10(12):171 – 183, 2005. The Second International Workshop on Unstructured Mesh Numerical Modelling of Coastal, Shelf and Ocean Flows.

[BRS+13] Hans-Joachim Bungartz, Christoph Riesinger, Martin Schreiber, Gregor Snelting, and Andreas Zwinkau. Invasive Computing in HPC with X10. In *X10 Workshop (X10'13)*, Seattle, Washington, June 2013.

[BRV12] Michael Bader, Kaveh Rahnema, and Csaba Attila Vigh. Memory-Efficient Sierpinski-Order Traversals on Dynamically Adaptive, Recursively Structured Triangular Grids. In Kristjan Jonasson, editor, *Applied Parallel and Scientific Computing - 10th International Conference, PARA 2010*, volume 7134 of *Lecture Notes in Computer Science*, pages 302–311. Springer, March 2012.

[BSA12] André R Brodtkorb, Martin L Sætra, and Mustafa Altinakar. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55:1–12, 2012.

[BSVB08] Michael Bader, Stefanie Schraufstetter, Csaba Vigh, and Jörn Behrens. Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves. *International Journal of Computational Science and Engineering*, 4(1):12–21, 2008.

[But64] John C Butcher. Implicit runge-kutta processes. *Mathematics of Computation*, 18(85):50–64, 1964.

[BWG11] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[BZ00] Jörn Behrens and Jens Zimmermann. Parallelizing an unstructured grid generator with a space-filling curve approach. In *Euro-Par 2000 Parallel Processing*, pages 815–823. Springer, 2000.

[CFL28] Richard Courant, Kurt Friedrichs, and Hans Lewy. Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen*, 100(1):32–74, 1928.

[CKT09] CE Castro, M Käser, and EF Toro. Space–time adaptive numerical methods for geophysical applications. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1907):4613–4631, 2009.

[CLDL09] Richard Comblen, Sébastien Legrand, Eric Deleersnijder, and Vincent Legat. A finite element method for solving the shallow water equations on the sphere. *Ocean Modelling*, 28(1):12–23, 2009.

[Coc98] Bernardo Cockburn. *An introduction to the discontinuous Galerkin method for convection-dominated problems*. Springer, 1998.

[CP08] Cédric Chevalier and François Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6):318–331, 2008.

[CS01] Bernardo Cockburn and Chi-Wang Shu. Runge–Kutta discontinuous Galerkin methods for convection-dominated problems. *Journal of scientific computing*, 16(3):173–261, 2001.

[Cyb89] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301, 1989.

[DBH+05] Karen D Devine, Erik G Boman, Robert T Heaphy, Bruce A Hendrickson, James D Teresco, Jamal Faik, Joseph E Flaherty, and Luis G Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2):133–152, 2005.

[DHB+00] Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St John, and Courtenay Vaughan. Design of dynamic load-balancing tools for parallel applications. In *Proceedings of the 14th international conference on Supercomputing*, pages 110–118. ACM, 2000.

[Dub91] Moshe Dubiner. Spectral methods on triangles and other domains. *Journal of Scientific Computing*, 6(4):345–390, 1991.

[Dun85] DA Dunavant. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *International journal for numerical methods in engineering*, 21(6):1129–1148, 1985.

[DZM07] Ramsay Dyer, Hao Zhang, and Torsten Möller. Voronoi-Delaunay duality and Delaunay meshes. In *Proceedings of the 2007 ACM symposium on Solid and physical modeling*, pages 415–420. ACM, 2007.

[EHB+13] Wolfgang Eckhardt, Alexander Heinecke, Reinhold Bader, Matthias Brehm, Nicolay Hammer, Herbert Huber, Hans-Georg Kleinhenz, Jadran Vrabec, Hans Hasse, Martin Horsch, et al. 591 TFLOPS multi-trillion particles simulation on SuperMUC. In *Supercomputing*, pages 1–12. Springer, 2013.

[ESS05] Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar. X10: an experimental language for high productivity programming of scalable systems. In *Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05)*. Citeseer, 2005.

[Fly66] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[For12] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard: Version 3.0*. High-Performance Computing Center, 2012.

[FP63] Roger Fletcher and Michael JD Powell. A rapidly convergent descent method for minimization. *The Computer Journal*, 6(2):163–168, 1963.

[FR89] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 247–252. ACM, 1989.

[Fra00] Anton Frank. *Organisationsprinzipien zur Integration von geometrischer Modellierung, numerischer Simulation und Visualisierung.* Dissertation, München, 2000.

[GBW90] Benjamin Garlick, D Baum, and J Winget. Interactive viewing of large geometric databases using multiprocessor graphics workstations. *SIGGRAPH90 course notes: Parallel Algorithms and Architectures for 3D Image Generation*, 1990.

[GE04] M Gopi and David Eppstien. Single-Strip Triangulation of Manifolds with Arbitrary Topology. In *Computer Graphics Forum*, volume 23, pages 371–379. Wiley Online Library, 2004.

[Geo06] David L George. *Finite volume methods and adaptive refinement for tsunami propagation and inundation*. PhD thesis, Citeseer, 2006.

[Geo08] David L George. Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation. *Journal of Computational Physics*, 227(6):3089–3113, 2008.

[GHM+12] Michael Gerndt, Andreas Hollmann, Marcel Meyer, Martin Schreiber, and Josef Weidendorfer. Invasive computing with iomp. In *Specification and Design Languages (FDL)*, pages 225 –231, September 2012.

[Gir06] Francis X Giraldo. High-order triangle-based discontinuous Galerkin methods for hyperbolic equations on a rotating sphere. *Journal of Computational Physics*, 214(2):447–465, 2006.

[GMPZ06] Frank Günther, Miriam Mehl, Markus Pögl, and Christoph Zenger. A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. *SIAM Journal on Scientific Computing*, 28(5):1634–1650, 2006.

[Gün04] Frank Günther. *Eine cache-optimale Implementierung der Finite-Elemente-Methode*. Dissertation, TU München, May 2004.

[GW08] FX Giraldo and T Warburton. A high-order triangular discontinuous Galerkin oceanic shallow water model. *International journal for numerical methods in fluids*, 56(7):899–925, 2008.

[HDJ04] Lok M Hwa, Mark A Duchaineau, and Kenneth I Joy. Adaptive 4-8 texture hierarchies. In *Proceedings of the conference on Visualization'04*, pages 219–226. IEEE Computer Society, 2004.

[HKR+12] Daniel F Harlacher, Harald Klimach, Sabine Roller, Christian Siebert, and Felix Wolf. Dynamic load balancing for unstructured meshes on space-filling curves. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1661–1669. IEEE, 2012.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–585, 1969.

[Höl13] Wolfgang Hölzl. Vectorization and GPGPU-Acceleration of an Augmented Riemann Solver for the Shallow Water Equations. Bachelor's thesis, Institut für Informatik, Technische Universität München, July 2013.

[Hor93] G Horton. A multi-level diffusion method for dynamic load balancing. *Parallel Computing*, 19(2):209–218, 1993.

[HT12] Alexander Heinecke and Carsten Trinitis. Cache-oblivious matrix algorithms in the age of multicores and many cores. *Concurrency and Computation: Practice and Experience*, 2012.

[HW08] Jan S Hesthaven and Tim Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, volume 54. Springer, 2008.

[IOC] IHO IOC. BODC, Centenary Edition of the GEBCO Digital Atlas. *British oceanographic data centre, Liverpool.*

[JLW05] Shuangshuang Jin, Robert R Lewis, and David West. A comparison of algorithms for vertex normal computation. *The Visual Computer*, 21(1-2):71–82, 2005.

[Ju07] Lili Ju. Conforming centroidal Voronoi Delaunay triangulation for quality mesh generation. *Inter. J. Numer. Anal. Model*, 4:531–547, 2007.

[KBL+11] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. DistRM: Distributed resource management for on-chip many-core systems. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 119–128. ACM, 2011.

[KCS04] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proc. of the 13th Int. Conf. on Par. Arch. and Compilation Techniques*, 2004.

[KD06] Martin Käser and Michael Dumbser. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes–I. The two-dimensional isotropic case with external source terms. *Geophysical Journal International*, 166(2):855–877, 2006.

[KDDLPI07] Martin Käser, Michael Dumbser, Josep De La Puente, and Heiner Igel. An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes–III. Viscoelastic attenuation. *Geophysical Journal International*, 168(1):224–242, 2007.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[KSK03] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis. *Parallel graph partitioning and sparse matrix ordering library. Version*, 2, 2003.

[KT04] Dmitri Kuzmin and Stefan Turek. High-resolution FEM-TVD schemes based on a fully multidimensional flux limiter. *Journal of Computational Physics*, 198(1):131–158, 2004.

[Lam32] H Lamb. Hydrodynamics Cambridge University Press. *Cambridge, UK*, 1932.

[LeV02] Randall J. LeVeque. *Finite-Volume Methods for Hyperbolic Problems.* Cambridge University Press, 2002.

[LNT07] Michael N. Levy, Ramachandran D. Nair, and Henry M. Tufo. High-order Galerkin methods for scalable global atmospheric models. *Computers & Geosciences*, 33(8):1022 – 1035, 2007.

[LP01] Peter Lindstrom and Valerio Pascucci. Visualization of large terrains made easy. In *Visualization, 2001. VIS'01. Proceedings*, pages 363–574. IEEE, 2001.

[Mav02] Dimitri J Mavriplis. Parallel performance investigations of an unstructured mesh Navier-Stokes solver. *International Journal of High Performance Computing Applications*, 16(4):395–407, 2002.

[MBGW10] Andreas Müller, Jörn Behrens, Francis X Giraldo, and Volkmar Wirth. An adaptive discontinuous Galerkin method for modeling cumulus clouds. 2010.

[MGLT] Breanyn T MacInnes, Aditya Riadi Gusman, Randall J LeVeque, and Yuichiro Tanioka. Comparison of earthquake source models for the 2011 Tohoku event using tsunami 2 simulations and near field observations 3.

[Mit07] William F Mitchell. A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. *Journal of Parallel and Distributed Computing*, 67(4):417–429, 2007.

[MJFS01] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):124–141, 2001.

[MK11] Andrey Marochko and Alexey Kukanov. Composable Parallelism Foundations in the Intel Threading Building Blocks Task Scheduler. In *PARCO*, pages 545–554, 2011.

[MNN11] Miriam Mehl, Tobias Neckel, and Ph Neumann. Navier–Stokes and Lattice–Boltzmann on octree-like grids in the Peano framework. *International Journal for Numerical Methods in Fluids*, 65(1-3):67–86, 2011.

[MRB12] Oliver Meister, Kaveh Rahnema, and Michael Bader. A Software Concept for Cache-Efficient Simulation on Dynamically Adaptive Structured Triangular Grids. In Koen De Boschhere, Erik H. D'Hollander, Gerhard R. Joubert, David Padua, and Frans Peters, editors, *Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22 of *Advances in Parallel Computing*, pages 251–260, Gent, May 2012. ParCo 2012, IOS Press.

[Mül12] Andreas Müller. *Untersuchungen zur Genauigkeit adaptiver unstetiger Galerkin-Simulationen mit Hilfe von Luftblasen-Testfällen.* Mainz, Univ., Diss., 2012, 2012.

[Mun06] Ralf-Peter Mundani. *Hierarchische Geometriemodelle zur Einbettung verteilter Simulationsaufgaben.* Dissertation, Aachen, 2006.

[NCT09] RD Nair, H-W Choi, and HM Tufo. Computational aspects of a scalable high-order discontinuous Galerkin atmospheric dynamical core. *Computers & Fluids*, 38(2):309–319, 2009.

[Nec09] Tobias Neckel. *The PDE Framework Peano: An Environment for Efficient Flow Simulations.* Dissertation, Institut für Informatik, Technische Universität München, June 2009. Dissertation erhältlich im Verlag Dr. Hut unter der ISBN 978-3-86853-147-3.

[NSL+11] Chris J Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, et al. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *Code generation and optimization (CGO), 2011 9th annual IEEE/ACM international symposium on*, pages 224–235. IEEE, 2011.

[NTL05] Ramachandran D Nair, Stephen J Thomas, and Richard D Loft. A discontinuous Galerkin transport scheme on the cubed sphere. *Monthly Weather Review*, 133(4):814–828, 2005.

[NUW12] Svetlana Nogina, Kristof Unterweger, and Tobias Weinzierl. Autotuning of Adaptive Mesh Refinement PDE Solvers on Shared-memory Architectures. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *PPAM 2011*, volume 7203 of *Lecture Notes in Computer Science*, pages 671–680, Heidelberg, Berlin, 2012. Springer-Verlag.

[OP09] Stephen L Olivier and Jan F Prins. Evaluating OpenMP 3.0 run time systems on unbalanced task graphs. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 63–78. Springer, 2009.

[Ope08] OpenMP Arch. Review Board. OpenMP Application Program Interface Version 3.0, 2008.

[OSK+11] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. OctoPOS: A parallel operating system for invasive computing. *Sventek, J.(Hrsg.)*, pages 9–14, 2011.

[PG07] Renato Pajarola and Enrico Gobbetti. Survey of semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer*, 23(8):583–605, 2007.

[PHP02] Nathaniel G Plant, K Todd Holland, and Jack A Puleo. Analysis of the scale of errors in nearshore bathymetric data. *Marine Geology*, 191(1):71–86, 2002.

[Pög04] Markus Pögl. *Entwicklung eines cache-optimalen 3D Finite-Element-Verfahrens für grosse Probleme.* Dissertation, Düsseldorf, August 2004.

[Ret12] Sebastian Rettenberger. Ein paralleler Server für adaptive Geoinformation in Strömungssimulationen. Master's thesis, Institut für Informatik, Technische Universität München, June 2012.

[RFLS06] Jean-François Remacle, Sandra Soares Frazao, Xiangrong Li, and Mark S Shephard. An adaptive discretization of shallow-water equations based on discontinuous galerkin methods. *International journal for numerical methods in fluids*, 52(8):903–923, 2006.

[Rüd93] Ulrich Rüde. Fully adaptive multigrid methods. *SIAM Journal on Numerical Analysis*, 30(1):230–248, 1993.

[Rus62] Vladimir Vasil'evich Rusanov. *Calculation of interaction of non-steady shock waves with obstacles.* NRC, Division of Mechanical Engineering, 1962.

[Sag94] Hans Sagan. *Space-filling curves*, volume 18. Springer-Verlag New York, 1994.

[SBB12] Martin Schreiber, Hans-Joachim Bungartz, and Michael Bader. Shared-memory Parallelization of Fully-Adaptive Simulations Using a Dynamic Tree-Split and -Join Approach. Puna, India, December 2012. IEEE International Conference on High Performance Computing (HiPC), IEEE Xplore.

[Sch03] Dirk Schwanenberg. *Die Runge-Kutta-Discontinuous-Galerkin-Methode zur Lösung konvektionsdominierter tiefengemittelter Flachwasserprobleme.* PhD thesis, 2003.

[Sch06] Stefanie Schraufstetter. Speichereffiziente Algorithmen zum Lösen partieller Differentialgleichungen auf adaptiven Dreiecksgittern. Diplomarbeit, TU München, July 2006.

[SES06] Joe Sampson, Alan Easton, and Manmohan Singh. Moving boundary shallow water flow above parabolic bottom topography. *Anziam Journal*, 47:C373–C387, 2006.

[She02] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry*, 22(13):21 – 74, 2002.

[SLJM11] Guangfu Shao, Xiangyu Li, Chen Ji, and Takahiro Maeda. Focal mechanism and slip history of the 2011 Mw 9.1 off the Pacific coast of Tohoku Earthquake, constrained with teleseismic body and surface waves. *Earth, planets and space*, 63(7):559–564, 2011.

[SM03] Endre Süli and David F Mayers. *An introduction to numerical analysis.* Cambridge University Press, 2003.

[SRNB13a] Martin Schreiber, Christoph Riesinger, Tobias Neckel, and Hans-Joachim Bungartz. Invasive compute balancing for applications with hybrid parallelization. In *Proceedings of the 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'13).* IEEE, October 2013.

[SRNB13b] Martin Schreiber, Christoph Riesinger, Tobias Neckel, and Hans-Joachim Bungartz. Invasive Compute Balancing for Applications with Shared and Hybrid Parallelization. 2013. submitted for publication.

[SSD03] Pavel Solin, Karel Segeth, and Ivo Dolezel. *Higher-order finite element methods.* CRC Press, 2003.

[SWB13a] Martin Schreiber, Tobias Weinzierl, and Hans-Joachim Bungartz. Cluster Optimization and Parallelization of Simulations with Dynamically Adaptive Grids. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013*, volume 8097 of *Lecture Notes in Computer Science*, pages 484–496, Berlin Heidelberg, 2013. Springer-Verlag.

[SWB13b] Martin Schreiber, Tobias Weinzierl, and Hans-Joachim Bungartz. SFC-based Communication Metadata Encoding for Adaptive Mesh. In *Proceedings of the International Conference on Parallel Computing (ParCo)*, October 2013. accepted.

[Syn91] Costas Emmanuel Synolakis. Tsunami runup on steep slopes: How good linear theory really is. In *Tsunami Hazard*, pages 221–234. Springer, 1991.

[Tei08] Jürgen Teich. Invasive Algorithms and Architectures, Invasive Algorithmen und Architekturen. *it-Information Technology*, 50(5):300–310, 2008.

[Tor01] EF Toro. Shock-capturing methods for free-surface shallow flows. *Chichester, etc.: Wiley*, 2001.

[TOS00] Ulrich Trottenberg, Cornelius W Oosterlee, and Anton Schuller. *Multigrid.* Access Online via Elsevier, 2000.

[UTK98] K UTKU. Long wave runup on piecewise linear topographies. *J. Fluid Mech*, 374:1–28, 1998.

[UWKA13] Kristof Unterweger, Tobias Weinzierl, David I. Ketcheson, and Aron Ahmadia. PeanoClaw - A Functionally-Decomposed Approach to Adaptive Mesh Refinement with Local Time Stepping for Hyperbolic Conservation Law Solvers. Technical Report TUM-I1332, 2013.

[Vig12] Csaba A. Vigh. *Parallel Simulation of the Shallow Water Equations on Structured Dynamically Adaptive Triangular Grids.* PhD thesis, Institut für Informatik, Technische Universität München, 2012.

[Wei09] Tobias Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids.* Dissertation, Institut für Informatik, Technische Universität München, München, 2009.

[Wel09] Hilary Weller. Predicting mesh density for adaptive modelling of the global atmosphere. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1907):4523–4542, 2009.

[WS91] Paul Wessel and Walter H. F. Smith. Free software helps map and display data. *Eos, Transactions American Geophysical Union*, 72(41):441–446, 1991.

[Xie13] Yuan Xie. Future memory and interconnect technologies. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 964–969. IEEE, 2013.

[ZK05] Gengbin Zheng and Laxmikant V Kale. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing.* Citeseer, 2005.

[Zum00] Gerhard Zumbusch. *On the quality of space-filling curve induced partitions.* Sonderforschungsbereich 256, 2000.