Technische Universität München
Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen

HIGH PERFORMANCE EARTHQUAKE SIMULATIONS

ALEXANDER NIKOLAS BREUER

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. Dr. Arndt Bode
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Michael G. Bader
	2. Univ.-Prof. Dr. Heiner Igel (Ludwig-Maximilians-Universität München)

Die Dissertation wurde am 22.09.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 03.12.2015 angenommen.

ABSTRACT

The understanding of earthquake dynamics is greatly supported by highly resolved coupled simulations of the rupture process and seismic wave propagation. This grand challenge of seismic modeling requires an immense amount of supercomputing resources, thus optimal utilization by software is imperative. Driven by recent hardware developments the increasing demand for parallelism and data locality often requires replacing major software parts to bring efficient numerics and machine utilization closely together.

In this thesis I present a new computational core for the seismic simulation package SeisSol. For minimal time-to-solution the new core is designed to maximize value and throughput of the floating point operations performed in the underlying ADER discontinuous Galerkin discretization method. Included are auto-tuned sparse and dense matrix kernels, hybrid parallelization from many-core nodes up to machine-size and a novel high performance clustered local time stepping scheme.

The presented computational core reduces time-to-solution of SeisSol by several factors and scales beyond 1 million cores. At machine-size the new core enabled a landmark simulation of the 1992 Landers earthquake. For the first time this simulation allowed the analysis of the complex rupture behavior resulting from the non-linear interaction of frictional sliding and seismic wave propagation at high geometric complexity.

ZUSAMMENFASSUNG

Das Verständnis der Erdbebendynamik wird von hochauflösenden, gekoppelten Simulationen des Bruchprozesses und der seismischen Wellenausbreitung unterstützt. Für die benötigten hohen Auflösungen wird eine immense Menge an Höchstleistungsrechenresourcen verwendet, daher ist eine optimale Ausnutzung durch die Software unerlässlich. Getrieben durch aktuelle Entwicklungen in der Hardware erfordern die höheren Anforderungen an Parallelisierung und Datenlokalität häufig das Ersetzen ganzer Softwareteile, um gleichzeitig eine effiziente Numerik und Maschinenauslastung zu gewährleisten.

In dieser Dissertation präsentiere ich einen neuen Rechenkern für die seismische Simulationssoftware SeisSol. Der neue Kern maximiert den Wert und Durchsatz der Gleitkommaoperationen in der zugrundeliegenden ADER-DG Diskretisierungsmethode, um die Rechenzeit zum gewünschten Ergebnis zu minimieren. Beinhaltet sind automatisch optimierte Matrixkernel, hybride Parallelisierung von Vielkernarchitekturen bis hin zum kompletten Großrechner, sowie ein hochperformantes gruppiertes lokales Zeitschrittschema.

Der präsentierte Kern reduziert die Rechenzeit von SeisSol um einen substantiellen Faktor und skaliert bis hin zu mehr als einer Millionen Recheneinheiten. Durch den Kern wurde eine wegweisende Simulation des Landers-Erdbebens von 1992 auf einem kompletten Großrechner ermöglicht. Zum ersten Mal erlaubte diese Simulation die Analyse des damit verbundenen komplexen Bruchprozesses, welcher aus der nichtlinearen Interaktion des Reibungsprozesses gekoppelt an die seismische Wellenausbreitung resultiert, in einer komplizierten Geometrie.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Earthquakes cover a broad spectrum of scales. Seismic waves travel through entire Earth and can be measured thousands of kilometers away from the source. In contrast, the generation of earthquakes is sensitive to a resolution of a few meters, critical for seismic hazard assessment and a highly nonlinear phenomenon. Only very few field observations are available in the source region of earthquakes. Thus computer simulations have become an indispensable tool to study the causes and effects of earthquakes.

DYNAMIC RUPTURE Physics-based dynamic rupture modeling is able to capture this source complexity. Dynamic rupture earthquake simulations include the source process as part of the solution by coupling seismic wave propagation and frictional sliding. Simpler kinematic approaches assume a prescribed rupture process and may lack information about physical earthquake dynamics.

However, accurate dynamic rupture simulations pose major challenges to the simulation environments. Realistic representations of the fault systems introduce a high geometric complexity, since the complex fault systems are described as non-planar interfaces (e.g. [50]). Common features include curved faults, kinks, branches or even fault roughness. Usually proper discretization for multiphysics dynamic rupture simulations is obtained by aligning faces or nodes of the computational mesh to the faults.

Once a proper discretization is found, initial fault stresses are prescribed and nonlinear friction laws are assumed for the rupture physics. After initiation of the rupture, e.g. by defining a nucleation zone, frictional sliding generates seismic waves. The seismic waves propagate through the volume of the computational domain and may trigger additional slip. Therefore accurate simulation of the rupture process and seismic wave propagation is equally important.

Seismic waves propagate at speeds dictated by heterogeneities of the material (e.g. [43]). Thus, heterogeneities of the fault system's surrounding medium have to be captured properly in the multiphysics modeling environment for both, accurate wave and accurate rupture propagation (e.g. [30, 17, 58]). Material features might be highly localized, which adds a second level of geometric complexity to the numerical discretization. Additionally seismic waves are reflected from surface topography. Thus, precise representations of surface topography further increase the geometric complexity.

Since no analytic solutions exist for realistic scenarios, the *SCEC/ USGS Spontaneous Rupture Code Verification Project* [29] aims at verification of the scientific community's software packages. As of March 2014, the project lists 39 benchmarks, 36 participating modelers and 14 sets of results for every benchmark in average [3]. The computational approaches use Finite Difference (FD) schemes (e.g. [17, 45]), Finite Element Methods (FEM), e.g. [4, 58, 49], or even hybrid FD-FEM approaches [48].

In comparison to FD packages, the flexibility of simulation environments utilizing FEM is advantageous, when the accurate discretization of complex fault systems, surface geometries and heterogeneities in the medium is required. The prize of this flexibility is an increased complexity of the implementations and increased computational work for FEM, at least for low-order simulations. Additionally, Discontinuous Galerkin (DG)-FEM do not impose continuity across the elements boundaries and thus the discontinuities of the rupture process are embedded very naturally into the formulations.

The SeisSol software package[1] is the topic of this thesis and uses, among others (e.g. [4, 58]), DG-FEM for spatial discretization. Together with the use of unstructured tetrahedral meshes and the ADER scheme in time, this allows for accurate discretization of fault systems, surface topography and material heterogeneities [23, 49, 22].

HPC Highly resolved, three-dimensional earthquake simulations demand an immense amount of supercomputing resources. This demand is driven by accurate representations of geometric features and the need for resolved, high frequencies in the simulations.

Thus, aside from numerical challenges, software packages simulating earthquakes are required to scale on state-of-the-art supercomputers. Recent forward simulations of seismic wave propagation successfully utilized large factions of some of the largest supercomputers worldwide (e.g. [31, 9, 32, 19, 63, 44, 6, 12, 60, 5]).

However, only very few of the performed landmark-simulations coupled dynamic rupture propagation directly to seismic wave propagation (e.g. [31], [18]). Taking the total number of simulation environments in the *SCEC/ USGS Spontaneous Rupture Code Verification Project* [29] into account, a gap between latest physics-driven developments and HPC capabilities is visible. Reason is the required, high degree of algorithmic development, optimization and testing required to exploit all levels of parallelism offered by state-of-the-art supercomputing architectures [6].

In this thesis I present a new computational core for SeisSol's computationally intensive wave propagation component. My core is tailored to complex geometries and heterogeneous materials, adopts re-

1 https://github.com/SeisSol/SeisSol

quirements of the targeted supercomputers in the algorithmic design phase, and includes careful optimization for all levels of concurrency.

COMPUTATIONAL CORE My new computational core for SeisSol, presented in this thesis, is designed to maximize value and throughput of the floating point operations performed in the underlying ADER DG-FE discretization. In contrast to a pure performance reengineering, the design of my core also restructures the algorithmic layout of SeisSol. This restructuring brings together efficient numerics and machine utilization.

The commonly used explicit time stepping schemes in earthquake simulations usually rely on the Courant–Friedrichs–Lewy (CFL)-condition for stability. This condition limits the maximum time step an element might have and is influenced by the order of convergence, occurring wave speeds and size of the considered element. In contrast to explicit schemes, unconditionally stable, implicit time integrators allow for larger time steps, but require the solution of large systems of equations.

Explicit, global time stepping (GTS) schemes use the same, minimum time step for all elements and thus ease implementation of the used time integration. However, when performing simulations with heterogeneous materials and adaptive meshes, this might result in a huge waste of computational resources if the time steps of many elements are largely underestimated.

A solution is offered by local time stepping (LTS) schemes (e.g. [24, 13, 53, 56, 25, 64, 54]). These schemes use different time steps for the elements in the computational domain. While the offered solutions successfully reduce the computational demands, large-scale implementations of the schemes are challenging due to the heterogeneous dependencies in time. Recent developments (e.g. [13, 53, 56]) suggest to reduce heterogeneity of the LTS algorithms by the introduction of clusters. The clusters summarize elements in the computational domain advancing with a common time step. Additionally the authors of [13, 53, 56] utilize multi-level partitioning to address distributed memory machines.

To address LTS challenges, the presented computational core features a novel high performance clustered LTS scheme. This scheme clusters elements with similar time steps together without requiring connectivity of clustered elements. Additionally, the presented, clustered LTS scheme follows ideas of multirate approaches (e.g. [55, 56]) by introducing a single fundamental time step. Here, all clusters advance with integer multiples of this time step. The resulting clustered local time stepping scheme is able to capture homogeneous and heterogeneous time step variations in the computational domain, maintains a large fraction of the theoretical speedup offered by LTS, and sustained petascale performance using a simple weighted partitioning.

On the engineering-side my computational core addresses all important performance characteristics of the used supercomputers. Here, my core implements a customized memory layout streamlined to my clustered local time stepping scheme. This memory layout features arbitrary vector-alignment, supports hardware prefetching, includes NUMA-aware initialization, and allows for in-place communication. The computational part of the core incorporates auto-tuned sparse and dense matrix kernels and includes hybrid, asynchronous parallelization.

In summary, this thesis studies the individual design decisions of my computational core, discusses the hardware-aware implementation and studies the obtained, unique algorithmic and computational performance in different setups.

OVERVIEW This thesis is split into the three major parts i Algorithm, ii Supercomputing, and iii Simulations.

First, Pt. i summarizes the fully discrete form of the used modal ADER-DG method for the elastic wave equations in Ch. 2.

Next, we derive the clustered LTS scheme of our new computational core in Ch. 3, the second chapter of Pt. i. Only the extensive algorithmic restructuring of SeisSol's initial LTS approach and the far-reaching design decisions in Ch. 3, rigorously aiming at node-level performance and scalability, enable our computational core for petascale performance in LTS setups. The discussed approaches cover all algorithmic parts of our clustered LTS scheme in multi-partition settings. This includes design decisions on a per-element basis in Ch. 3.1, the clustering of elements with similar time steps for regularity in Ch. 3.2, the partitioning and the introduction of communication layers for distributed memory machines in Ch. 3.3, and the time management and scheduling in chapters 3.4 and 3.5.

Pt. ii, the second part of this thesis, maps the clustered LTS scheme of Ch. 3 to supercomputing systems. After introducing our targeted supercomputers in Ch. 4, we systematically address all technical layers of the machines in our computational core. First, Ch. 5 introduces the data structures and derives a high performance memory layout supporting vectorization, shared memory parallelization and distributed memory parallelization. Next, in Ch. 6, we discuss how our computational core realizes hybrid parallelization. Our clustered LTS scheme is designed to be asynchronous, thus details of message progression using a dedicated communication core are covered. The last chapter of Pt. ii, Ch. 7, introduces the innermost kernels of our computational core. These kernels drive the single core performance of our computational core and discussions of arithmetic intensities and the value of the performed hardware floating point operations are included.

After the derivation of our clustered LTS scheme in Pt. i and the mapping to supercomputers in Pt. ii, the final part of this thesis, Pt. iii, studies the performance of our computational core in different setups. First, we study the single-node and small-scale performance of our computational core using different LTS configurations in Ch. 8. These small-scale results build the baseline for the large-scale evaluation up to machine-size in the remaining chapters. Ch. 9 performs a global time stepping weak scaling and shows that our computational core is prepared for simulations with more than 10^{12} degrees of freedom. Next, Ch. 10 studies the performance of multiphysics dynamic rupture earthquake simulations up to machine size. The presented, full-machine GTS production run of the 1992 Landers earthquake sustained petascale performance and shows the applicability of our computational core. Finally, Ch. 11 applies all of our computational core's functionality to seismic wave propagation in Mount Merapi. Especially the presented, sustained petascale performance of a clustered LTS simulation with production character summarizes the compelling results of the algorithmic design decision made in Pt. i and the engineering decisions made in Pt. ii.

Part I

ALGORITHM

This part covers all algorithmic design decisions of our clustered local time stepping scheme.

The first chapter of this part, Ch. 2, summarizes the fully discrete form of the used ADER-DG scheme for elastic wave equations.

In the second chapter, Ch. 3, we design a clustered local time stepping scheme aiming at high node-level performance and scalability. The individual algorithmic design decisions cover constraints on LTS relations of neighboring elements, the clustering, and the multi-partition layout of our clustered LTS scheme.

2 NUMERICS

This chapter gives a brief recapitulation of SeisSol's underlying numerics. We start this chapter with the introduction of basic concepts, used in our formulation of the ADER-DG scheme. First, Ch. 2.1 recapitulates the reference coordinates and the reference tetrahedron. Both, the reference coordinates and the reference tetrahedron, are used in the spatial DG discretization, together with mappings to and from the meshed tetrahedrons. Next, Ch. 2.2 defines a set of polynomial, hierarchical, orthogonal basis functions in terms of the reference element. These basis function are used in Ch. 2.3 to define the unique mass matrix, unique stiffness matrices and unique flux matrices.

Starting from the elastic wave equations in Ch. 2.4, we then derive the fully discrete formulation of the ADER-DG method. For this purpose chapters 2.5 and 2.6 apply the DG machinery for spatial discretization. First, we cover the ADER time integration in Ch. 2.7, followed by the volume integration in Ch. 2.8 and the numerical flux in Ch. 2.9. Finally, the individual integrators lead to the final fully discrete formulation in Ch. 2.10.

2.1 REFERENCE COORDINATES

In this chapter we recapitulate the reference tetrahedron T_R together with a reference $\xi_1\xi_2\xi_3$-coordinate system and face parameters χ_1 and χ_2. We use all three in the following chapters for the definition of the basis function and the corresponding matrix structures.

Our reference tetrahedron is defined by a set of sorted vertices $T_R = (\vec{o}, \vec{e}_1, \vec{e}_2, \vec{e}_3)$ with the following coordinates:

$$\vec{o} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \ \vec{e}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \ \vec{e}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \ \vec{e}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \tag{1}$$

Vertex $\vec{o}$ coincides with the origin and $\vec{e}_1, \dots, \vec{e}_3$ with the unit vectors of our reference $\xi_1\xi_2\xi_3$-coordinate system.

Assume a given tetrahedron $T_k = (\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4)$ in physical $x_1x_2x_3$-coordinates with vertices $\vec{x}_1, \dots, \vec{x}_4$. Further, we assume that the point $\vec{x}_1$ is the representation of the origin $\vec{o}$ in physical coordinates and that the vectors $\vec{x}_i - \vec{x}_1, \ i = 2, 3, 4$ represent the unit vectors $\vec{e}_1, \dots, \vec{e}_3$ of the $\xi_1\xi_2\xi_3$-system in physical coordinates. Then the coordinate transformation $\Xi(\vec{\xi})$ maps each point $\vec{\xi} \in T_R$ in reference coordinates to a unique point $\vec{x} \in T_k$ in physical coordinates [11, Ch. 3.5.4.5]:

$$\vec{x} = \Xi(\vec{\xi}) = \begin{pmatrix} l_1 & l_2 & l_3 \\ m_1 & m_2 & m_3 \\ n_1 & n_2 & n_3 \end{pmatrix} \vec{\xi} + \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}, \tag{2}$$

with

$$\begin{pmatrix} l_1 & l_2 & l_3 \\ m_1 & m_2 & m_3 \\ n_1 & n_2 & n_3 \\ p_1 & p_2 & p_3 \end{pmatrix} = \begin{pmatrix} x_{21} - x_{11} & x_{31} - x_{11} & x_{41} - x_{11} \\ x_{22} - x_{12} & x_{32} - x_{12} & x_{42} - x_{12} \\ x_{23} - x_{13} & x_{33} - x_{13} & x_{43} - x_{13} \\ x_{11} & x_{12} & x_{13} \end{pmatrix}. \tag{3}$$

By inverting the transformation matrix (incl. the translation) we obtain the back-transformation $\Xi^{-1}(\vec{x})$ from physical coordinates to reference coordinates:

$$\vec{\xi} = \Xi^{-1}(\vec{x}) = \begin{pmatrix} l_1 & l_2 & l_3 \\ m_1 & m_2 & m_3 \\ n_1 & n_2 & n_3 \end{pmatrix}^{-1} \left(\vec{x} - \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \right). \tag{4}$$

We additionally require parametrizations of the reference element's faces $(\partial T_R)_i,\ i \in 1, \ldots, 4$ in the upcoming scheme. Following [23] we define the face-parameters $(\chi_1, \chi_2)^T$ to obtain the face's volume coordinates $\vec{\xi}(i, (\chi_1, \chi_2)^T) \in (\partial T_R)_i$ in the reference system by:

$$\vec{\xi}(i, (\chi_1, \chi_2)^T) = \begin{cases} (\chi_2,\ \chi_1,\ 0)^T & \text{if } i = 1 \\ (\chi_1,\ 0,\ \chi_2)^T & \text{if } i = 2 \\ (0,\ \chi_2,\ \chi_1)^T & \text{if } i = 3 \\ (1 - \chi_1 - \chi_2,\ \chi_1,\ \chi_2)^T & \text{if } i = 4. \end{cases} \tag{5}$$

Two face-neighboring tetrahedrons in $x_1x_2x_3$-coordinates have different mappings to the reference element. We assume that the shared face in the reference tetrahedron is given by i for the first tetrahedron and j for the second tetrahedron. To use (5) for both faces we define the mapping $\vec{\chi}_n$ from parameters of an element-face to those of the neighboring element's face [23] :

$$\vec{\chi}_n(h, (\chi_1, \chi_2)^T) = \begin{cases} (\chi_2,\ \chi_1)^T & \text{if } h = 1 \\ (1 - \chi_1 - \chi_2,\ \chi_2)^T & \text{if } h = 2 \\ (\chi_1,\ 1 - \chi_1 - \chi_2)^T & \text{if } h = 3. \end{cases} \tag{6}$$

The index h again depends on the mappings of both tetrahedrons to the reference element. It summarizes the three possible vertex combinations two triangular faces can have.

Assuming $(\hat{\chi}_1, \hat{\chi}_2)^T$ are the given face-parameters of our first tetrahedron, $\vec{\xi}(i, (\hat{\chi}_1, \hat{\chi}_2)^T)$ gives the associated volume coordinates of the first tetrahedron in the reference system. Combining (5) and (6), the second tetrahedron's coordinates are given by $\vec{\xi}(j, \vec{\chi}_n(h, (\hat{\chi}_1, \hat{\chi}_2)^T))$.

2.2 BASIS FUNCTIONS

The discontinuous Galerkin Finite Element scheme recapitulated in the following chapters requires a discrete expansion basis. We approximate a quantity $q \in \mathbb{R}$ inside our reference tetrahedron T_R by $q_h \in \mathbb{R}$ via a modal expansion:

$$q(\vec{\xi}) \approx q_h(\vec{\xi}) = \sum_{b=1}^{B} \tilde{q}_b \phi_b(\vec{\xi}). \tag{7}$$

In (7) the constant coefficients $\hat{q}_b$ are the modes and $\phi_b(\vec{\xi})$ are the spatially dependent basis functions.

The choice of the basis functions ϕ_b not only dictates the performance-relevant sparsity patterns of important matrix structures (mass, stiffness, flux) but also has a great influence on the accuracy of the final discontinuous Galerkin scheme. In literature a variety of options for the proper choice of basis functions exists (e.g. [16, 39]).

In this thesis we utilize a commonly used set of polynomial, orthogonal and hierarchical basis functions based on the Jacobi Polynomials. As described in [39, Ch. 3.2] the construction of the used tetrahedral basis follows the ideas of tensor product modal expansions. The introduction of a collapsed coordinate system allows to introduce a set of one-dimensional primal functions ψ^a_p, $\psi^b_{p,q}$ and $\psi^c_{p,q,r}$:

$$\begin{aligned} \psi^a_p(\eta) &= P^{0,0}_p(\eta), \\ \psi^b_{p,q}(\eta) &= \left(\frac{1-\eta}{2}\right)^p P^{2p+1,0}_q(\eta), \\ \psi^c_{p,q,r}(\eta) &= \left(\frac{1-\eta}{2}\right)^{p+q} P^{2p+2q+2,0}_r(\eta), \end{aligned} \tag{8}$$

with the Jacobi polynomials $P^{\alpha,\beta}_n$ defined as:

$$P^{\alpha,\beta}_n(\eta) = \frac{(-1)^n}{2^n n!}(1-\eta)^{-\alpha}(1+\eta)^{-\beta}\frac{\partial^n}{\partial \eta^n}\left((1-\eta)^{\alpha+n}(1+\eta)^{\beta+n}\right). \tag{9}$$

Together with the mapping from tetrahedral $\xi_1\xi_2\xi_3$- to hexahedral $\eta_1\eta_2\eta_3$-coordinates,

$$\eta_1 = \frac{2\xi_1}{1-\xi_2-\xi_3} - 1, \quad \eta_2 = \frac{2\xi_2}{1-\xi_3} - 1, \quad \eta_3 = -1 + 2\xi_3, \tag{10}$$

we can then define the polynomial expansion up to degree P as product of the primal functions [39, Ch. 3.2]:

$$\phi_{p,q,r}(\vec{\xi}) = \psi^a_p(\eta_1)\psi^b_{p,q}(\eta_2)\psi^c_{p,q,r}(\eta_3), \quad p+q+r \le P. \tag{11}$$

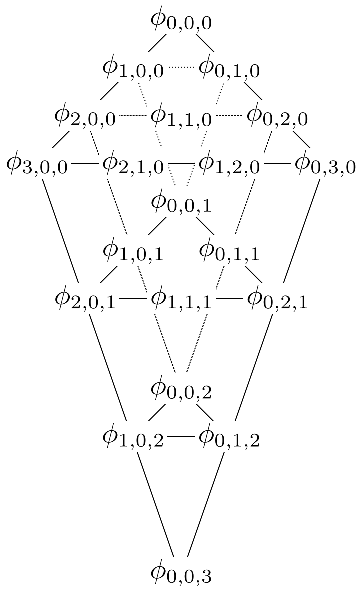

$\phi_{0,0,0}$
$\phi_{1,0,0}$ $\phi_{0,1,0}$
$\phi_{2,0,0}$ $\phi_{1,1,0}$ $\phi_{0,2,0}$
$\phi_{3,0,0}$ $\phi_{2,1,0}$ $\phi_{1,2,0}$ $\phi_{0,3,0}$
$\phi_{0,0,1}$
$\phi_{1,0,1}$ $\phi_{0,1,1}$
$\phi_{2,0,1}$ $\phi_{1,1,1}$ $\phi_{0,2,1}$
$\phi_{0,0,2}$
$\phi_{1,0,2}$ $\phi_{0,1,2}$
$\phi_{0,0,3}$

Figure 1: Illustration of the hierarchical expansion basis. The triangle in the front with nodes $\phi_{3,0,0}$, $\phi_{0,3,0}$ and $\phi_{0,0,3}$ contains all additional basis functions $P = 3$ adds to $P \leq 2$.

Fig. 1 illustrates the hierarchical structure of the basis. We see that every new polynomial degree $P + 1$ adds the basis functions $\phi_{p,q,r}$ with $p + q + r = P + 1$ to those of the previous degree with $p + q + r \leq P$. For the mapping of the triple (p, q, r) to the index b used in (7) we sort the basis functions on the outer level by the hierarchical structure. Every contribution of the hierarchical level is then sorted by the indices r, q and p:

$$\left.\begin{array}{l} \left.\begin{array}{l} \phi_1 = \phi_{0,0,0}, \;\} P = 1 \\ \phi_2 = \phi_{1,0,0},\; \phi_3 = \phi_{0,1,0},\; \phi_4 = \phi_{0,0,1} \end{array}\right\} P = 2 \\ \phi_5 = \phi_{2,0,0},\; \phi_6 = \phi_{1,1,0},\; \phi_7 = \phi_{0,2,0},\; \phi_8 = \phi_{1,0,1}, \\ \phi_9 = \phi_{0,1,1},\; \phi_{10} = \phi_{0,0,2} \end{array}\right\} P = 3 \qquad (12)$$
$$\vdots$$

2.3 GLOBAL MATRICES

Based on the reference tetrahedron (see Ch. 2.1) and the basis functions (see Ch. 2.2) we are now able to define the global set of matrices required in the upcoming discontinuous Galerkin scheme.

These are the mass matrix M, the three stiffness matrices K^{ξ_c}, the four left state flux matrices $F^{-,i}$ and the 48 right state flux matrices $F^{+,i,j,h}$:

$$\begin{aligned}
M &= \int_{T_R} \phi_l \phi_m \,\mathrm{d}\vec{\xi} \\
K^{\xi_c} &= \int_{T_R} \phi_l \left(\phi_m\right)_{\xi_c} \mathrm{d}\vec{\xi} && c,k \in 1,2,3 \\
F^{-,i} &= \int_{(\partial T_R)_i} \phi_l \left(\vec{\xi}(i,\vec{\chi})\right) \phi_m \left(\vec{\xi}(i,\vec{\chi})\right) \mathrm{d}\vec{\chi} && i,j \in 1,2,3,4 \\
& && l,m \in 1,\dots,B. \\
F^{+,i,j,h} &= \int_{(\partial T_R)_i} \phi_l \left(\vec{\xi}(i,\vec{\chi})\right) \phi_m \left(\vec{\xi}(j,\chi_n(h,\vec{\chi}))\right) \mathrm{d}\vec{\chi}
\end{aligned} \tag{13}$$

2.4 ELASTIC WAVE EQUATIONS

The *elastic wave equations* are a variable coefficient linear system of hyperbolic partial differential equations and the basic set of equations used for seismic wave propagation in SeisSol. In the homogeneous three-dimensional case they are given as [46]:

$$q_t + A^{x_1} q_{x_1} + A^{x_2} q_{x_2} + A^{x_3} q_{x_3} = 0. \tag{14}$$

q is the nine-dimensional space-time-dependent vector of quantities:

$$q(\vec{x},t) = \begin{pmatrix} \sigma^{11} & \sigma^{22} & \sigma^{33} & \sigma^{12} & \sigma^{23} & \sigma^{13} & u_1 & u_2 & u_3 \end{pmatrix}^T . \tag{15}$$

σ^{11}, σ^{22} and σ^{33} are the normal stresses and σ^{12}, σ^{23} and σ^{13} the shear stresses of the six-dimensional stress tensor. The particle velocities in physical x_1-, x_2- and x_3-direction are summarized by the quantities u_1, u_2 and u_3.

Furthermore the three space-dependent Jacobians A^{x_1}, A^{x_2} and A^{x_3} are defined by:

$$A^{x_1}(\vec{x}) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & -\lambda - 2\mu & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\lambda & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\lambda & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu \\ -\rho^{-1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\rho^{-1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\rho^{-1} & 0 & 0 & 0 \end{pmatrix}$$

$$A^{x_2}(\vec{x}) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda - 2\mu & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\mu & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\rho^{-1} & 0 & 0 & 0 & 0 & 0 \\ 0 & -\rho^{-1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\rho^{-1} & 0 & 0 & 0 & 0 \end{pmatrix} .$$

$$A^{x_3}(\vec{x}) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda - 2\mu \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\mu & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\rho^{-1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\rho^{-1} & 0 & 0 & 0 & 0 \\ 0 & 0 & -\rho^{-1} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} . \tag{16}$$

The parameters of the Jacobians summarize the properties of the material. $\lambda(\vec{x})$ and $\mu(\vec{x})$ are the Lamé parameters, whereby μ is the shear modulus and λ does not have a direct physical interpretation. The density of the material is given by $\rho(\vec{x}) > 0$.

The nine eigenvalues $s_1, \dots, s_9$ of the Jacobians determine the Riemann structure of the elastic wave equations:

$$\begin{aligned} & s_1(\vec{x}) = -\sqrt{\frac{\lambda+2\mu}{\rho}}, \quad s_2(\vec{x}) = -\sqrt{\frac{\mu}{\rho}}, \quad s_3(\vec{x}) = -\sqrt{\frac{\mu}{\rho}}, \\ & s_4(\vec{x}) = s_5(\vec{x}) = s_6(\vec{x}) = 0, \\ & s_7(\vec{x}) = \sqrt{\frac{\mu}{\rho}}, \quad s_8(\vec{x}) = \sqrt{\frac{\mu}{\rho}}, \quad s_9(\vec{x}) = \sqrt{\frac{\lambda+2\mu}{\rho}}. \end{aligned} \tag{17}$$

s_1 and s_9 correspond to the P-wave velocities, while s_2, s_3, s_7 and s_8 correspond to the S-wave velocities.

The corresponding matrix of A^{x_1}'s right eigenvectors R is given by:

$$R = \begin{pmatrix} \lambda+2\mu & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda+2\mu \\ \lambda & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \lambda \\ \lambda & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \lambda \\ 0 & \mu & 0 & 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mu & 0 & 0 & 0 & \mu & 0 & 0 \\ s_9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & s_1 \\ 0 & s_8 & 0 & 0 & 0 & 0 & 0 & s_2 & 0 \\ 0 & 0 & s_7 & 0 & 0 & 0 & s_3 & 0 & 0 \end{pmatrix}. \tag{18}$$

2.5 SPATIAL DISCRETIZATION

In this section we use the discontinuous Galerkin Finite Element method in a modal formulation to discretize the system (14).

First we split the given spatial domain $\Omega \in \mathbb{R}^3$ in K pairwise disjunct tetrahedrons T_k:

$$\Omega = \bigcup_{k=0}^{K} T_k \tag{19}$$

Using the modal basis functions $\phi_b(\vec{\xi})$ (see Ch. 2.2) up to polynomial degree $P \in \mathbb{N}$ leads to a spatial approximation of order $\mathcal{O} = P+1$. Additionally using the transformations $\Xi_k^{-1}(\vec{x})$ to map each tetrahedron k to reference coordinates (see Ch. 2.1), we obtain for $\vec{x} \in T_k$:

$$q(\vec{x},t) = \begin{pmatrix} q_1(\vec{x},t) \\ q_2(\vec{x},t) \\ \vdots \\ q_9(\vec{x},t) \end{pmatrix} \approx q_h(\vec{x},t) = \sum_{b=1}^{B_{\mathcal{O}}} \begin{pmatrix} \tilde{q}_{k,1,b}(t)\phi_b\left(\Xi_k(\vec{x})\right) \\ \tilde{q}_{k,2,b}(t)\phi_b\left(\Xi_k(\vec{x})\right) \\ \vdots \\ \tilde{q}_{k,9,b}(t)\phi_b\left(\Xi_k(\vec{x})\right) \end{pmatrix}. \tag{20}$$

The time-dependent modal coefficients $Q_k = \{\tilde{q}_{k,i,j}\}$ denote the $9 \times B_{\mathcal{O}}$ per-element degrees of freedom (DOFs), where $B_{\mathcal{O}}$ denotes the required number of basis functions.

2.6 WEAK FORMULATION

To derive the weak form we multiply (14) with test functions. Choosing our basis $\phi_m(\Xi_k^{-1}(\vec{x}))$ as test functions and additionally integrating over a control volume $T_k \in \mathbb{R}^3$ we obtain:

$$\int_{T_k} q_t \phi_m \, \mathrm{d}\vec{x} + \sum_{c=1}^{3} \int_{T_k} A^{x_c} q_{x_c} \phi_m \, \mathrm{d}\vec{x} = 0 \quad \forall m \in 1, \ldots, B_{\mathcal{O}}. \tag{21}$$

Application of the chain rule gives:

$$\int_{T_k} q_t \phi_m \, \mathrm{d}\vec{x} = \sum_{c=1}^{3} \left(\int_{T_k} A^{x_c} q \left(\phi_m\right)_{x_c} \mathrm{d}\vec{x} - \int_{T_k} A^{x_c} \left(q\phi_m\right)_{x_c} \mathrm{d}\vec{x} \right) \tag{22}$$

Next we apply the divergence theorem and obtain:

$$\int_{T_k} q_t \phi_m \, \mathrm{d}\vec{x} = \sum_{c=1}^{3} \left(\int_{T_k} A^{x_c} q \left(\phi_m\right)_{x_c} \mathrm{d}\vec{x} \right) - \int_{\partial T_k} F \phi_m \, \mathrm{d}\vec{s} \quad , \tag{23}$$

where we also introduced the numerical flux F as an approximation in the surface integral. The numerical flux is derived in Ch. 2.9 for a face-aligned coordinate system exploiting the rotational invariance of the system (14).

Approximation of the solution q in (23) via the discrete finite element space gives:

$$\begin{aligned} &\sum_{b=1}^{B} \left(\left((Q_k)_b\right)_t \int_{T_k} \phi_b \phi_m \, \mathrm{d}\vec{x} \right) \\ &= \sum_{b=1}^{B} \left(\sum_{c=1}^{3} A_k^{x_c} (Q_k)_b \int_{T_k} \phi_b (\phi_m)_{x_c} \, \mathrm{d}\vec{x} \right) - \int_{\partial T_k} F \phi_m \, \mathrm{d}\vec{s}. \end{aligned} \tag{24}$$

Here $(Q_k)_b \in \mathbb{R}^9$ is the vector of modal coefficients per basis function $b \in 1, \ldots, B_{\mathcal{O}}$ in element k.

2.7 TIME PREDICTION

We use Cauchy-Kovalewski procedure as a per-element time prediction of the DOFs and therefore as main ingredient for our time discretization. First we write the governing system (14) in terms of reference coordinates [41]:

$$q_t + A^{\xi_1} q_{\xi_1} + A^{\xi_2} q_{\xi_2} + A^{\xi_3} q_{\xi_3} = 0. \tag{25}$$

Here we introduce the Jacobians in reference coordinates A^{ξ_c}, which are a linear combination of the Jacobians A^{x_c}:

$$A^{\xi_{c_r}} = \sum_{c=1}^{3} \frac{\partial \vec{\xi}_{c_r}}{\partial \vec{x}_c} A^{x_c}, \quad c_r \in 1,2,3. \tag{26}$$

Following [41], we project (25) onto the basis ϕ_b and obtain, together with insertion of the spatial discretization q_h (see (20)), the time derivatives $(Q_k)_t$ of the DOFs:

$$(Q_k)_t = -\left(A^{\xi_1} Q_k \left(K^{\xi_1}\right)^T + A^{\xi_2} Q_k \left(K^{\xi_2}\right)^T + A^{\xi_3} Q_k \left(K^{\xi_3}\right)^T\right) M^{-1}. \tag{27}$$

M^{-1} is the inverse of the mass matrix and $(K^{\xi_c})^T$ the transposed stiffness matrices (see (13)).

The Taylor series of the DOFs Q_k for order $\mathcal{O} \in \mathbb{N}^+$ about expansion point t_0 is then given by:

$$Q_k(t_0, t) = \sum_{d=0}^{\mathcal{O}-1} \frac{(t - t_0)^d}{d!} \cdot \frac{\partial^d}{\partial t^d} Q_k(t_0). \tag{28}$$

Here, we use (27) recursively to obtain higher derivatives $\partial^d / \partial t^d Q_k$ with the DOFs itself as initial condition for the zeroth derivative: $\partial^0 / \partial t^0 Q_k = Q_k(t_0)$.

The time integrated DOFs $\mathcal{T}_k$ over interval $[\hat{t}, \hat{t} + \Delta t]$ are obtained by integrating the Taylor series approximation (28) analytically:

$$\begin{aligned} \mathcal{T}_k(t_0, \hat{t}, \Delta t) &= \int_{\hat{t}}^{\hat{t}+\Delta t} Q_k(t_0, t)\,\mathrm{d}t \\ &= \sum_{d=0}^{\mathcal{O}-1} \frac{\left(\hat{t} + \Delta t - t_0\right)^{d+1} - \left(\hat{t} - t_0\right)^{d+1}}{(d+1)!} \cdot \frac{\partial^d}{\partial t^d} Q_k(t_0). \end{aligned} \tag{29}$$

2.8 VOLUME INTEGRATION

To formulate the volume integration in reference coordinates, we apply the change of variables theorem to the second sum in (24):

$$\sum_{b=1}^{B} \left(\sum_{c=1}^{3} A_k^{x_c} (Q_k)_b \int_{T_k} \phi_b \left(\phi_m\right)_{x_c} \mathrm{d}\vec{x} \right) = \left(\sum_{c=1}^{3} |J_k| A_k^{\xi_c} Q_k K^{\xi_c} \right)_m . \tag{30}$$

$A_k^{\xi_c}$ are the linear combinations of $A_k^{x_c}$, obtained with (26). K^{ξ_c} are the stiffness matrices of (13) and $|J_k|$ the determinant of the Jacobian matrix corresponding to the transformation Ξ_k^{-1} from the reference tetrahedron T_R to T_k.

We use the time integrated DOFs (29) to define the final, fully discrete volume operator $\mathcal{V}_k$:

$$\mathcal{V}_k(\mathcal{T}_k) = \sum_{c=1}^{3} A_k^{\xi_c} \mathcal{T}_k K^{\xi_c} M^{-1}. \tag{31}$$

Note that the inverse mass matrix M^{-1} is required in the final formulation and that we drop $|J_k|$ on purpose as it will cancel out in the final formulation.

2.9 SURFACE INTEGRATION

In this chapter we formulate the solution of the face-normal generalized Riemann problem stemming from the surface integral in (24). To simplify the following considerations we rotate the quantities of two neighboring tetrahedrons T_k and T_{k_i} to a common face-aligned coordinate system first. This step exploits the rotational-invariance of (14) and allows us to formulate the face-normal Riemann problem. For a given tetrahedron k and face i with face-normal direction $\vec{n}_{k,i} = (n_{x_1}, n_{x_2}, n_{x_3})^T$ the rotation, $U_{k,i}$, of the DOFs to the face-aligned coordinate system is given by [23]:

$$U_{k,i} = \begin{pmatrix} U^{1,1}_{k,i} & U^{1,2}_{k,i} & 0_{3\times3} \\ U^{2,1}_{k,i} & U^{2,2}_{k,i} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & U^{3,3}_{k,i} \end{pmatrix}, \tag{32}$$

with

$$U^{1,1}_{k,i} = \begin{pmatrix} n^2_{x_1} & s^2_{x_1} & t^2_{x_1} \\ n^2_{x_2} & s^2_{x_2} & t^2_{x_2} \\ n^2_{x_3} & s^2_{x_3} & t^2_{x_3} \end{pmatrix},$$

$$U^{1,2}_{k,i} = \begin{pmatrix} 2n_{x_1}s_{x_1} & 2s_{x_1}t_{x_1} & 2n_{x_1}t_{x_1} \\ 2n_{x_2}s_{x_2} & 2s_{x_2}t_{x_2} & 2n_{x_2}t_{x_2} \\ 2n_{x_3}s_{x_3} & 2s_{x_3}t_{x_3} & 2n_{x_3}t_{x_3} \end{pmatrix},$$

$$U^{2,1}_{k,i} = \begin{pmatrix} n_{x_2}n_{x_1} & s_{x_2}s_{x_1} & t_{x_2}t_{x_1} \\ n_{x_3}n_{x_2} & s_{x_3}s_{x_2} & t_{x_3}t_{x_2} \\ n_{x_3}n_{x_1} & s_{x_3}s_{x_1} & t_{x_3}t_{x_1} \end{pmatrix},$$

$$U^{2,2}_{k,i} = \begin{pmatrix} n_{x_2}s_{x_1} + n_{x_1}s_{x_2} & s_{x_2}t_{x_1} + s_{x_1}t_{x_2} & n_{x_2}t_{x_1} + n_{x_1}t_{x_2} \\ n_{x_3}s_{x_2} + n_{x_2}s_{x_3} & s_{x_3}t_{x_2} + s_{x_2}t_{x_3} & n_{x_3}t_{x_2} + n_{x_2}t_{x_3} \\ n_{x_3}s_{x_1} + n_{x_1}s_{x_3} & s_{x_3}t_{x_1} + s_{x_1}t_{x_3} & n_{x_3}t_{x_1} + n_{x_1}t_{x_3} \end{pmatrix},$$

$$U^{3,3}_{k,i} = \begin{pmatrix} n_{x_1} & s_{x_1} & t_{x_1} \\ n_{x_2} & s_{x_2} & t_{x_2} \\ n_{x_3} & s_{x_3} & t_{x_3} \end{pmatrix}.$$

The two additional vectors $\vec{s}$ and $\vec{t}$ are orthogonal to each other and tangential to face i. The outer-pointing normal $\vec{n}_{k,i}$ of face i points from T_k to T_{k_i}. We call direction $T_k \to T_{k_i}$ *right* and direction $T_{k_i} \to T_k$ *left*. Further we use the notation $x < x_0$ for all values lying on the left side of the straight line through x_0 with direction $n_{k,i}$ and $x > x_0$ for all values on the right side.

In our discretization the face-normal Generalized Riemann Problem GRP_P of order P is the following one-dimensional, face-normal, initial value problem at point $x_0 \in \partial T_{k,i}$ and time t_0 [59, Ch. 19.2]:

$$q_t + A(\vec{x}) q_{x_1} = 0 \tag{33}$$

$$q(x, t_0) = \begin{cases} q_l = U_{k,i} q_k(x_0, t_0), & \text{if } x < x_0 \\ q_r = U_{k,i} q_{k_i}(x_0, t_0), & \text{if } x > x_0. \end{cases} \tag{34}$$

$$A(x) = \begin{cases} A_l = A_k^{x_1}, & \text{if } x < x_0 \\ A_r = A_{k_i}^{x_1}, & \text{if } x > x_0 \end{cases} \tag{35}$$

$q_k(x_0, t_0)$ and $q_{k_i}(x_0, t_0)$ are the degree P boundary extrapolated finite element approximations (20) of the two neighboring tetrahedrons T_k and T_{k_i}. A_l is the constant Jacobian $A_k^{x_1}$ in T_k and A_r the Jacobian $A_{k_i}^{x_1}$ in T_{k_i}.

As described in [59, Ch. 19.4] the solution of the GRP_P can be computed via P classical Riemann problems in the derivatives $\partial^j/\partial t^j q_l$ and $\partial^j/\partial t^j q_r$ at space-time point (x_0, t_0). Following [46, Ch. 9] we solve these classical Riemann problems via eigenvector decompositions of the jumps $\partial^j/\partial t^j q_r - \partial^j/\partial t^j q_l$. The eigenvectors of A_l associated to negative (left-going) waves combined with those of A_r associated to positive (right-going) waves gives R^{lr}, the combined matrix of eigenvectors:

$$\begin{aligned} R^{lr} &= \begin{pmatrix} r_1^{lr} & r_2^{lr} & \dots & r_9^{lr} \end{pmatrix} \\ &= \begin{pmatrix} \lambda_l + 2\mu_l & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda_r + 2\mu_r \\ \lambda_l & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \lambda_r \\ \lambda_l & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \lambda_r \\ 0 & \mu_l & 0 & 0 & 0 & 0 & 0 & \mu_r & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mu_l & 0 & 0 & 0 & \mu_r & 0 & 0 \\ (s_9)_l & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (s_1)_r \\ 0 & (s_8)_l & 0 & 0 & 0 & 0 & 0 & (s_2)_r & 0 \\ 0 & 0 & (s_7)_l & 0 & 0 & 0 & (s_3)_r & 0 & 0 \end{pmatrix}. \end{aligned} \tag{36}$$

Entries with subscript l refer to material parameters of T_k and those with r to parameters of T_{k_i}. The wave strengths α_j are obtained by decomposition of the jumps into eigenvectors:

$$\alpha_j = \begin{pmatrix} \alpha_{j,1} & \alpha_{j,2} & \dots & \alpha_{j,9} \end{pmatrix}^T = \left(R^{lr}\right)^{-1} \left(\frac{\partial^j}{\partial t^j} q_r - \frac{\partial^j}{\partial t^j} q_l \right). \tag{37}$$

We derive the constant left-side middle states q_j^m by jumping over the left-going waves starting from the left states $\partial^j/\partial t^j q_l$:

$$q_j^m = \frac{\partial^j}{\partial t^j} q_l + \sum_{p:(s_p)_l<0} \alpha_{j,p}\, r_p^{lr}. \tag{38}$$

Rearrangement of the terms in (38) together with (37) separates T_k's contribution to the middle states from that of T_{k_i}:

$$\begin{aligned} q_j^m = \frac{\partial^j}{\partial t^j} q_l \quad &- \begin{pmatrix} r_1^{lr} & r_2^{lr} & r_3^{lr} & 0_{9\times 6}\end{pmatrix} \left(R^{lr}\right)^{-1} \frac{\partial^j}{\partial t^j} q_l \\ &+ \begin{pmatrix} r_1^{lr} & r_2^{lr} & r_3^{lr} & 0_{9\times 6}\end{pmatrix} \left(R^{lr}\right)^{-1} \frac{\partial^j}{\partial t^j} q_r. \end{aligned} \tag{39}$$

We get the time-dependent left-side middle state $q^m(t)$ as solution of the GRP$_P$ by combining the middle states q_j^m of the derivatives via the Cauchy-Kovalewski procedure [59, Ch. 19.4]:

$$q^m(t) = \sum_{j=0}^{P} \frac{t^j}{j!} q_j^m. \tag{40}$$

Additionally multiplying the left-side middle-state q^m with matrix A_l and rotating back from face-coordinates with $U_{k,i}^{-1}$ defines our numerical flux:

$$\begin{aligned} F_{k,i}(t) = &U_{k,i}^{-1} \left(A_l - A_l \begin{pmatrix} r_1^{lr} & r_2^{lr} & r_3^{lr} & 0_{9\times 6}\end{pmatrix} \left(R^{lr}\right)^{-1} \right) \sum_{j=0}^{P} \frac{t^j}{j!} \frac{\partial^j}{\partial t^j} q_l \\ &+ U_{k,i}^{-1} A_l \begin{pmatrix} r_1^{lr} & r_2^{lr} & r_3^{lr} & 0_{9\times 6}\end{pmatrix} \left(R^{lr}\right)^{-1} \sum_{j=0}^{P} \frac{t^j}{j!} \frac{\partial^j}{\partial t^j} q_r. \end{aligned} \tag{41}$$

(41) shows that the complete flux computation reduces to multiplication with two face-local matrices once time-derivatives have been obtained from the Cauchy-Kovalewski procedure (27). We call these matrices *flux solvers* and define them as:

$$\begin{aligned} \hat{A}_{k,i}^{-} &= U_{k,i}^{-1} \left(A_l - A_l \begin{pmatrix} r_1^{lr} & r_2^{lr} & r_3^{lr} & 0_{9\times 6}\end{pmatrix} \left(R^{lr}\right)^{-1} \right) U_{k,i} \\ \hat{A}_{k,i}^{+} &= U_{k,i}^{-1} A_l \begin{pmatrix} r_1^{lr} & r_2^{lr} & r_3^{lr} & 0_{9\times 6}\end{pmatrix} \left(R^{lr}\right)^{-1} U_{k,i}. \end{aligned} \tag{42}$$

To derive the fully discrete formulation we rewrite the surface integral in (24) and insert the flux (41) together with the flux matrices and the flux solvers introduced in (13) and (42):

$$\int_{\partial T_k} F \phi_m \, \mathrm{d}s = \sum_{i=1}^{4} |S_i| \left(\hat{A}_{k,i}^{-} \hat{Q}_k F^{-,i} + \hat{A}_{k,i}^{+} \hat{Q}_{k_i} F^{+,i,j,h} \right). \tag{43}$$

S_i is a scalar coming from change of variables theorem and denotes the area of face i in physical $x_1x_2x_3$-coordinates. $\hat{Q}_k$ are the time predictions of tetrahedron k's DOFs obtained via Cauchy-Kovalewski (27) and $\hat{Q}_{k_i}$ those of k's ith face neighbor.

Integration of (43) in time with (29) over the interval $[\hat{t}_k, \hat{t}_k + \Delta t_k]$ gives the per-face formulation of the surface operator:

$$\mathcal{F}_{k,i}(\mathcal{T}_k, \mathcal{T}_{k,i}) = \mathcal{F}^-_{k,i}(\mathcal{T}_k) + \mathcal{F}^+_{k,i}(\mathcal{T}_{k_i}) \tag{44}$$

with

$$\mathcal{F}^-_{k,i}(\mathcal{T}_k) = \frac{|S_i|}{|J_k|} \hat{A}^-_{k,i} \mathcal{T}_k F^{-,i} M^{-1}, \tag{45}$$

$$\mathcal{F}^+_{k,i}(\mathcal{T}_{k_i}) = \frac{|S_i|}{|J_k|} \hat{A}^+_{k,i} \mathcal{T}_{k_i} F^{+,i,j,h} M^{-1}. \tag{46}$$

Similar to the volume operator we include the effect of the inverse mass matrix M^{-1} and $1/|J_k|$ already in this definition to account for the final fully discrete formulation.

2.10 TIME STEPPING

By integrating $(Q_k)_t$ in (24) in time and by the definition of the mass matrix M we get:

$$\sum_{b=1}^{B} \left(\int_{t_k^n}^{t_k^n + \Delta t_k} ((Q_k)_b)_t \, \mathrm{d}t \int_{T_k} \phi_b \phi_m \, \mathrm{d}\vec{x} \right) = \left(\left(Q_k^{n_k+1} - Q_k^{n_k} \right) |J_k| M \right)_m, \tag{47}$$

where $Q_k^{n_k+1}$ are the DOFs at the next time step $t_k^{n_k+1} = t_k^{n_k} + \Delta t_k$ and $|J_k|$ is the determinant coming from the change of variables theorem.

We get the final fully discrete form via integration of (24) in time and inserting the time operator (29), the volume operator (31) and the surface operator (44):

$$Q_k^{n_k+1} = Q_k^{n_k} + \mathcal{V}_k(\mathcal{T}_k) - \sum_{i=1}^{4} \mathcal{F}_{k,i}\left(\mathcal{T}_k, \mathcal{T}_{k,i}\right). \tag{48}$$

The effect of the inverse mass matrix M^{-1} and $|J_k|$ is already included in the volume and surface operator.

Additionally, we split (48) to get a two-step scheme. The first step updates the DOFs with all local contributions of tetrahedron k itself:

$$Q_k^{*,n_k+1} = Q_k^{n_k} + \mathcal{V}_k(\mathcal{T}_k) - \sum_{i=1}^{4} \mathcal{F}^-_{k,i}\left(\mathcal{T}_k\right). \tag{49}$$

The second step uses the time integrated DOFs of the face-neighbors k_i only to compute the complete DOFs $Q_k^{n_k+1}$ at time step $t_k^{n_k+1}$:

$$Q_k^{n_k+1} = Q_k^{*,n_k+1} - \sum_{i=1}^{4} \mathcal{F}^+_{k,i}\left(\mathcal{T}_{k_i}\right). \tag{50}$$

For stability the per-element time step Δt_k is limited by Δt_k^{CFL}, the maximum allowed time step imposed by the CFL-condition: $0 < \Delta t_k \leq \Delta t_k^{\text{CFL}}$. We use a maximum, which resembles 50% of the stability of Runge-Kutta-schemes [23, 24]:

$$\Delta t_k^{\text{CFL}} = \frac{1}{2\mathcal{O} - 1} \cdot \frac{d}{\max_i |s_i|}. \tag{51}$$

d is the insphere diameter of tetrahedron T_k. s_i are k's eigenvalues in (17), thus $\max_i |s_i| = |s_1| = s_9$ holds.

SUMMARY

This chapter introduced the fully discrete formulation of our modal ADER-DG discretization for the elastic wave equations. First, we defined a unique reference element in Ch. 2.1 and defined a set of hierarchical, polynomial basis functions in Ch. 2.2. Together with appropriate mappings of our tetrahedrons in the computational domain to the reference element, this allowed us to use a set of global, unique matrices (mass, stiffness, flux) in our ADER-DG discretization.

The derivation of the spatial discretization in Ch. 2.5 and the weak formulation in Ch. 2.6 led us to the time integrator in Ch. 2.7, the volume integrator in Ch. 2.8 and the surface integrator in Ch. 2.9. The combination of all integrators resulted in our two-step scheme of Ch. 2.10. Here, we split our discrete formulation into a local contribution (49) and a neighboring contribution (50). Finally, we formulated the per-element time step limitations resulting from the CFL-condition (51).

In theory we are able to implement a solver for the elastic wave equations at this point. However, since we aim at efficient, large scale earthquake simulations, multiple design decision are outstanding from an algorithmic and performance engineering perspective. For this purpose we derive a clustered local time stepping scheme in the next chapter, Ch. 3, and exploit the locality of the CFL-condition (51). This scheme trades some of the theoretical optimality, where every element advances with its maximum allowed time step, for regularity requirements of modern supercomputers. Afterwards, Pt. ii maps the derived clustered local time stepping scheme to supercomputing systems.

3 CLUSTERED LOCAL TIME STEPPING

Eq. (48) allows for arbitrary per-element time steps Δt_k as long as the element-local CFL-requirements are met [24]. To utilize Local Time Stepping (LTS), [24] and [13] introduce per-element flux-buffers to which neighboring tetrahedrons write their contributions. The proposed LTS scheme in [24, 13] then sweeps over the mesh and updates an element's DOFs only if the flux-buffer contains all neighboring contributions of possible intermediate neighboring time steps. Whenever a tetrahedron is updated it also updates the neighboring flux-buffers. Here the tetrahedron uses its time derivatives to compute $\mathcal{T}_{k_i}$ in the neighboring contribution of the boundary operator $\mathcal{F}$ in (46) to update the flux-buffers.

In contrast to [24] and [13] we follow a different paradigm for the derivation of our LTS scheme. We force the elements to provide time data for LTS, which is accessed in a read-only fashion by face-neighbors in the update step. Additionally clustering similar time steps together reduces the irregularity of our scheme and removes the requirement to use sweeps touching all elements. We start the derivation of our LTS scheme in Ch. 3.1 by deriving time stepping relations two neighboring tetrahedrons are allowed to have. Based on this derivation Ch. 3.2 describes the clustering of the LTS scheme and the individual steps required to perform the setup. Ch. 3.3 extends our clustering with a layout capable of handling multiple partitions and thus featuring distributed memory systems. Additionally, Ch. 3.4 derives a time management scheme to formalize our elements' dependencies in time. Finally, Ch. 3.5 presents a scheduling method favoring critical work in multi-partition settings aiming at large-scale simulations.

3.1 LTS RELATIONS

Our LTS scheme limits the possible relations neighboring elements can have to two general cases. Assume a given tetrahedron k with a face-neighbor k_i and corresponding time steps Δt_k and Δt_{k_i}. Both elements are initially in sync, $t_k^{n_k} = t_{k_i}^{n_{k_i}}$. We differentiate between the case $\Delta t_k = \frac{1}{r} \cdot \Delta t_{k_i},\ r \in \mathbb{N}^+$, and the case $\Delta t_k = r \cdot \Delta t_{k_i},\ r \in \mathbb{N}^+$. The Global Time Stepping (GTS) relation $\Delta t_k = \Delta t_{k_i}$ $(r = 1)$ is part of both cases and the decision which implementation is used for GTS depends on the neighboring elements' LTS configurations. Note that limiting the time step differences to $r \in \mathbb{N}^+$ instead of arbitrary rates

$r \in \mathbb{R}^+$ is imposed for efficiency reasons of the final scheme rather than being a limitation of the ADER-scheme.

In the first case, $\Delta t_k = \frac{1}{r}\Delta t_{k_i}$, we require the neighboring tetrahedron k_i to store its derivatives $\mathcal{D}_{k_i}$:

$$\mathcal{D}_{k_i} = \begin{pmatrix} Q_{k_i}^{n_{k_i}} & \frac{\partial}{\partial t} Q_{k_i}^{n_{k_i}} & \frac{\partial^2}{\partial t^2} Q_{k_i}^{n_{k_i}} & \dots & \frac{\partial^{\mathcal{O}-1}}{\partial t^{\mathcal{O}-1}} Q_{k_i}^{n_{k_i}} \end{pmatrix}. \tag{52}$$

We can use the formulation of the ADER time integration in (29) to compute the time integrated DOFs in arbitrary intervals after the expansion point t_0. Thus k_i's time derivatives $\mathcal{D}_{k_i}$ computed at expansion point $t_0 = t_{k_i}^{n_{k_i}}$ can be used for integration intervals $[t_{k_i}^{n_{k_i}} + \Delta t_k^l, t_{k_i}^{n_{k_i}} + \Delta t_k^u]$ within the limits of the CFL-condition: $0 \leq \Delta t_k^l < \Delta t_k^u \leq \Delta t_{k_i}^{\text{CFL}}$ [24].

For our neighboring elements k and k_i with $\Delta t_k^{\text{CFL}} \geq \Delta t_k = \frac{1}{r}\Delta t_{k_i} \leq \Delta t_{k_i}^{\text{CFL}}$, $r \in \mathbb{N}^+$ we use k_i's derivatives $\mathcal{D}_{k_i}$ in one or multiple time steps of k to derive the time integrated DOFs $\mathcal{T}_{k,i}$ in (50), e.g. three times if $\Delta t_k = \frac{1}{3}\Delta t_{k_i}$ $(r = 3)$.

On the contrary the second case, $\Delta t_k = r \cdot \Delta t_{k_i}$, requires us to evaluate the boundary operator $\mathcal{F}_{k,i}^+$ in (50) over the entire interval $[t_k^{n_k}, t_k^{n_k} + \Delta t_k]$. This interval covers multiple time steps of k_i if $r > 1$. The linearity of $\mathcal{F}_{k,i}^+$ allows us to sum up multiple successive time integrated DOFs of k_i together and evaluate the corresponding neighboring flux contribution only once when updating k's DOFs:

$$\begin{aligned} &\sum_{l=0}^{r-1} \hat{A}_{k,i}^+ \mathcal{T}_{k_i}(t_k^{n_{k_i}} + l\Delta t_{k_i}, t_k^{n_{k_i}} + l\Delta t_{k_i}, \Delta t_{k_i}) F^{+,i,j,h} M^{-1} \\ &= \hat{A}_{k,i}^+ \left(\sum_{l=0}^{r-1} \mathcal{T}_{k_i}(t_k^{n_{k_i}} + l\Delta t_{k_i}, t_k^{n_{k_i}} + l\Delta t_{k_i}, \Delta t_{k_i}) \right) F^{+,i,j,h} M^{-1}. \end{aligned} \tag{53}$$

Here i, j and h denote the respective indices of the shared k-k_i-face from k's perspective.

The complete time information of k_i, required for the update of k, is only available after the r^{th} time prediction of k_i. Thus we require the neighboring tetrahedron k_i to sum its time integrated DOFs in a time buffer $\mathcal{B}_{k_i}$ until the entire time interval $[t_k^{n_k}, t_k^{n_k} + \Delta t_k]$ is covered:

$$\mathcal{B}_{k_i} = \sum_{l=0}^{r-1} \mathcal{T}_{k_i}\left(t_k^{n_k} + l\Delta t_{k_i}, t_k^{n_k} + l\Delta t_{k_i}, \Delta t_{k_i}\right). \tag{54}$$

EXAMPLE In the following we examine a simple example with two neighboring tetrahedrons k_1 and k_2. For illustrative purposes we consider updates and dependencies between k_1 and k_2 only. Of course the final scheme is required to respect all dependencies and perform all updates imposed by the four face-neighbors of every tetrahedron. Assume elements k_1 and k_2 have the LTS relation $\Delta t_{k_1} = 3 \cdot \Delta t_{k_2}$ and neighbor each other via k_1's local face i_{k_2} and k_2's local face i_{k_1}. Initially both tetrahedrons are synchronized, $t_{k_1}^{n_1} = t_{k_2}^{n_2}$. k_2 updates three times more often than k_1, thus requires k_1 to provide derivatives $\mathcal{D}_{k_1}$. Conversely k_1 only updates once for every three updates of k_2, therefore k_2 is required to provide a buffer $\mathcal{B}_{k_2}$.

First k_1 and k_2 compute their time predictions via (27) and (29). k_1 directly stores the derivatives and k_2 initializes the buffer B_{k_2} with the time integrated DOFs over interval $[t^{n_2}_{k_2}, t^{n_2}_{k_2} + \Delta t_{k_2}]$:

$$\begin{aligned} \mathcal{D}_{k_1} &= \begin{pmatrix} Q^{n_1}_{k_1} & \frac{\partial}{\partial t} Q^{n_1}_{k_1} & \frac{\partial^2}{\partial t^2} Q^{n_1}_{k_1} & \dots & \frac{\partial^{\mathcal{O}-1}}{\partial t^{\mathcal{O}-1}} Q^{n_1}_{k_1} \end{pmatrix}, \\ \mathcal{B}_{k_2} &= \mathcal{T}_{k_2}\left(t^{n_2}_{k_2}, t^{n_2}_{k_2}, \Delta t_{k_2}\right). \end{aligned} \tag{55}$$

Directly after computing the individual time information both tetrahedrons update their DOFs with the element-local contributions according to Eq. (49) (see Fig. 2a):

$$\begin{aligned} Q^{*,n_1+1}_{k_1} &= Q^{n_1}_{k_1} + \mathcal{V}_{k_1}\left(\mathcal{T}_{k_1}(t^{n_1}_{k_1}, t^{n_1}_{k_1}, \Delta t_{k_1})\right) \\ &\quad - \mathcal{F}^{-}_{k_1, i_{k_2}}\left(\mathcal{T}_{k_1}(t^{n_1}_{k_1}, t^{n_1}_{k_1}, \Delta t_{k_1})\right), \\ Q^{*,n_2+1}_{k_2} &= Q^{n_2}_{k_2} + \mathcal{V}_{k_2}\left(\mathcal{T}_{k_2}(t^{n_2}_{k_2}, t^{n_2}_{k_2}, \Delta t_{k_2})\right) \\ &\quad - \mathcal{F}^{-}_{k_2, i_{k_1}}\left(\mathcal{T}_{k_2}(t^{n_2}_{k_2}, t^{n_2}_{k_2}, \Delta t_{k_2})\right). \end{aligned} \tag{56}$$

In the next step only k_2 is allowed to complete its first time step. k_1 has to wait until k_2's buffer contains all three successive time predictions of k_2. Fig. 2b shows the neighboring update of k_2, which updates k_2's DOFs with k_1's contribution via (50):

$$Q^{n_2+1}_{k_2} = Q^{*,n_2+1}_{k_2} - \mathcal{F}^{+}_{k_2, i_{k_1}}\left(\mathcal{T}_{k_1}(t^{n_1}_{k_1}, t^{n_2}_{k_2}, \Delta t_{k_2})\right). \tag{57}$$

Here we use k_1's derivatives $\mathcal{D}_{k_1}$ to get $\mathcal{T}_{k_1}(t^{n_1}_{k_1}, t^{n_1}_{k_2}, \Delta t_{k_2})$. Now k_2's DOFs are at time $t^{n_2+1}_{k_2} = t^{n_2}_{k_2} + \Delta t_{k_2}$.

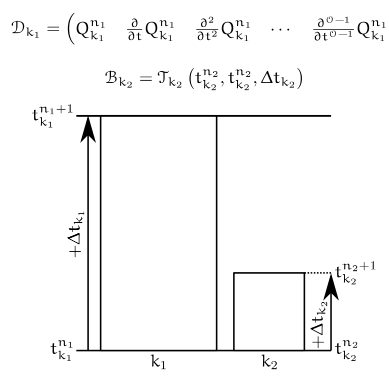
Next we compute the time integrated DOFs $\mathcal{T}_{k_2}(t^{n_2+1}_{k_2}, t^{n_2+1}_{k_2}, \Delta t_{k_2})$ and update k_2's buffer $\mathcal{B}_{k_2}$ and DOFs accordingly (see Fig. 2c):

$$\mathcal{B}_{k_2} = \mathcal{T}_{k_2}\left(t^{n_2}_{k_2}, t^{n_2}_{k_2}, \Delta t_{k_2}\right) + \mathcal{T}_{k_2}(t^{n_2+1}_{k_2}, t^{n_2+1}_{k_2}, \Delta t_{k_2}) \tag{58}$$

$$\begin{aligned} Q^{*,n_2+2}_{k_2} &= Q^{n_2+1}_{k_2} + \mathcal{V}_{k_2}\left(\mathcal{T}_{k_2}(t^{n_2+1}_{k_2}, t^{n_2+1}_{k_2}, \Delta t_{k_2})\right) \\ &\quad - \mathcal{F}^{-}_{k_2, i_{k_1}}\left(\mathcal{T}_{k_2}(t^{n_2+1}_{k_2}, t^{n_2+1}_{k_2}, \Delta t_{k_2})\right). \end{aligned} \tag{59}$$

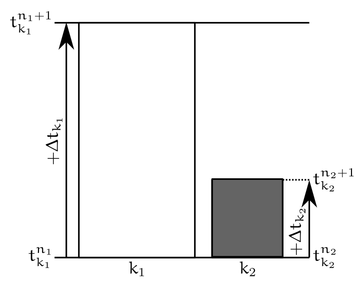
k_1 still requires k_2's time integrated DOFs over the interval $[t_{k_1} + 2 \cdot \Delta t_{k_2}, t_{k_1} + 3 \cdot \Delta t_{k_2}]$ in the buffer $\mathcal{B}_{k_2}$. Thus only k_2 is allowed to complete its second time step by using k_1's derivatives $\mathcal{D}_{k_1}$ again (see Fig. 2d):

$$Q^{n_2+2}_{k_2} = Q^{*,n_2+2}_{k_2} - \mathcal{F}^{+}_{k_2, i_{k_1}}\left(\mathcal{T}_{k_1}(t^{n_1}_{k_1}, t^{n_2+1}_{k_2}, \Delta t_{k_2})\right). \tag{60}$$

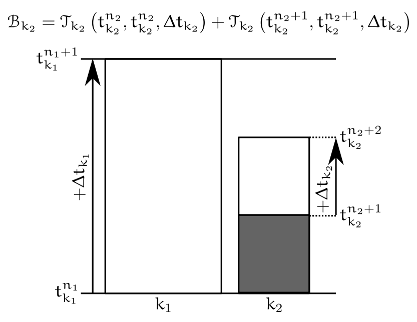


$\mathcal{D}_{k_1} = \left(Q^{n_1}_{k_1} \quad \frac{\partial}{\partial t}Q^{n_1}_{k_1} \quad \frac{\partial^2}{\partial t^2}Q^{n_1}_{k_1} \quad \cdots \quad \frac{\partial^{O-1}}{\partial t^{O-1}}Q^{n_1}_{k_1}\right)$
$\mathcal{B}_{k_2} = \mathcal{T}_{k_2}\left(t^{n_2}_{k_2}, t^{n_2}_{k_2}, \Delta t_{k_2}\right)$
$t^{n_1+1}_{k_1}$
$+\Delta t_{k_1}$
$t^{n_1}_{k_1}$
$t^{n_2+1}_{k_2}$
$+\Delta t_{k_2}$
$t^{n_2}_{k_2}$
k_1
k_2

(a) k_1 and k_2 compute time predictions, set the derivatives $\mathcal{D}_{k_1}$ and the buffer $\mathcal{B}_{k_2}$ accordingly, and update their DOFs with element-local contributions.

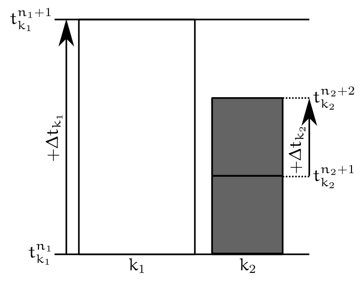


(b) k_2 finishes its first time step by using k_1's derivatives $\mathcal{D}_{k_1}$ in the neighboring flux computation.

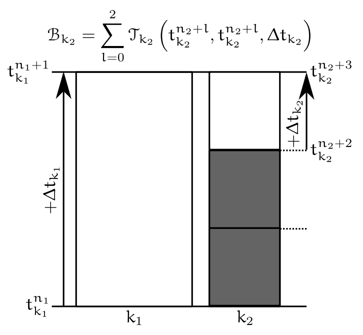


$\mathcal{B}_{k_2} = \mathcal{T}_{k_2}\left(t^{n_2}_{k_2}, t^{n_2}_{k_2}, \Delta t_{k_2}\right) + \mathcal{T}_{k_2}\left(t^{n_2+1}_{k_2}, t^{n_2+1}_{k_2}, \Delta t_{k_2}\right)$
$t^{n_1+1}_{k_1}$
$+\Delta t_{k_1}$
$t^{n_1}_{k_1}$
$t^{n_2+2}_{k_2}$
$+\Delta t_{k_2}$
$t^{n_2+1}_{k_2}$
k_1
k_2

(c) k_2 computes its next time prediction, updates the buffer $\mathcal{B}_{k_2}$ accordingly, and updates its DOFs with element-local contributions.

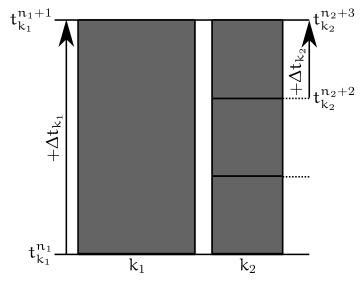


(d) k_2 finishes its second time step by using k_1's derivatives $\mathcal{D}_{k_1}$ in the neighboring flux computation.



$\mathcal{B}_{k_2} = \sum_{l=0}^{2} \mathcal{T}_{k_2}\left(t^{n_2+l}_{k_2}, t^{n_2+l}_{k_2}, \Delta t_{k_2}\right)$
$t^{n_1+1}_{k_1}$
$+\Delta t_{k_1}$
$t^{n_1}_{k_1}$
$t^{n_2+3}_{k_2}$
$+\Delta t_{k_2}$
$t^{n_2+2}_{k_2}$
k_1
k_2

(e) k_2 computes its next time prediction, updates the buffer $\mathcal{B}_{k_2}$ accordingly, and updates its DOFs with element-local contributions.



(f) k_1 finishes its first time step by using k_2's buffers $\mathcal{B}_{k_2}$ in the neighboring flux computation. k_2 finishes its third time step by using k_1's derivatives $\mathcal{D}_{k_1}$ in the neighboring flux computation.

Figure 2: Time marching scheme for two neighboring tetrahedrons k_1 and k_2 with $\Delta t_{k_1} = 3 \cdot \Delta t_{k_2}$ and initial times $t^{n_1}_{k_1} = t^{n_2}_{k_2}$. (a), (c) and (e) show the time prediction steps, which set the derivatives $\mathcal{D}_{k_1}$ of k_1 and the time buffer $\mathcal{B}_{k_2}$ of k_2, combined with the element-local updates of the DOFs (white). (b), (d) and (f) show the update with the neighboring time information (gray). Only dependencies between k_1 and k_2 are considered in the given example ignoring additional neighbors for illustrative purposes.

At this point k_2's DOFs are at time step $t_{k_2}^{n_2+2} = t_{k_2}^{n_2} + 2 \cdot \Delta t_{k_2}$, while k_1's DOFs still contain k_1's local contribution of the first time step only.

Now k_2 computes the time integrated DOFs $\mathcal{T}_{k_2}(t_{k_2}^{n_2+2}, t_{k_2}^{n_2+2}, \Delta t_{k_2})$ and updates its time buffer $\mathcal{B}_{k_2}$ and DOFs (see Fig. 2e):

$$\mathcal{B}_{k_2} = \sum_{l=0}^{2} \mathcal{T}_{k_2}(t_{k_2}^{n_2+l}, t_{k_2}^{n_2+l}, \Delta t_{k_2}), \tag{61}$$

$$\begin{aligned} Q_{k_2}^{*,n_2+3} = Q_{k_2}^{n_2+2} + \mathcal{V}_{k_2}\left(\mathcal{T}_{k_2}(t_{k_2}^{n_2+2}, t_{k_2}^{n_2+2}, \Delta t_{k_2})\right) \\ - \mathcal{F}^{-}_{k_2,i_{k_1}}\left(\mathcal{T}_{k_2}(t_{k_2}^{n_2+2}, t_{k_2}^{n_2+2}, \Delta t_{k_2})\right). \end{aligned} \tag{62}$$

After this step the time buffer $\mathcal{B}_{k_2}$ covers the entire interval $[t_{k_1}^{n_1}, t_{k_1}^{n_1} + \Delta t_{k_1}]$. Thus next, not only k_2 is allowed to finish the time step by using $\mathcal{D}_{k_1}$ once again, but also k_1 by using k_2's buffer $\mathcal{B}_{k_2}$ (see Fig. 2f):

$$\begin{aligned} Q_{k_1}^{n_1+1} &= Q_{k_1}^{*,n_1+1} - \mathcal{F}^{+}_{k_1,i_{k_1}}(\mathcal{B}_{k_2}) \\ Q_{k_2}^{n_2+3} &= Q_{k_2}^{*,n_2+3} - \mathcal{F}^{+}_{k_2,i_{k_1}}\left(\mathcal{T}_{k_1}(t_{k_1}^{n_1}, t_{k_2}^{n+2}, \Delta t_{k_2})\right). \end{aligned} \tag{63}$$

As illustrated in Fig. 2f after the single time step of k_1 and the three of k_2 both tetrahedrons are synchronized again, $t_{k_1}^{n_1+1} = t_{k_2}^{n_2+3}$.

3.2 CLUSTERING

In this section we derive the clustering of our LTS scheme. Here we consider the entire mesh containing all tetrahedrons, cluster elements together based on their time step and normalize the setup to match all of our requirements.

INITIAL CLUSTERING We define the minimum time step $\Delta t_{\min}^{\text{CFL}}$ and maximum time step $\Delta t_{\max}^{\text{CFL}}$ over all time steps:

$$\Delta t_{\min}^{\text{CFL}} = \min_{k=1,\dots,K} \Delta t_k^{\text{CFL}}, \qquad \Delta t_{\max}^{\text{CFL}} = \max_{k=1,\dots,K} \Delta t_k^{\text{CFL}}. \tag{64}$$

Our clusters C_l's time intervals C_l^r are pairwise disjunct and overlap the entire interval of possible time steps:

$$\begin{aligned} C_1^r &= \left[\Delta t_{\min}^{\text{CFL}}, \quad r_1 \cdot \Delta t_{\min}^{\text{CFL}}\right), \\ C_2^r &= \left[r_1 \cdot \Delta t_{\min}^{\text{CFL}}, \quad r_1 \cdot r_2 \cdot \Delta t_{\min}^{\text{CFL}}\right), \\ &\vdots \\ C_L^r &= \left[r_1 \cdot \ldots \cdot r_{L-1} \cdot \Delta t_{\min}^{\text{CFL}}, \quad r_1 \cdot \ldots \cdot r_L \cdot \Delta t_{\min}^{\text{CFL}}\right), \end{aligned} \tag{65}$$

$$\bigcup_{l=1}^{L} C_l^r \supset [\Delta t_{\min}^{\text{CFL}}, \Delta t_{\max}^{\text{CFL}}]. \tag{66}$$

For sizes of the clusters we choose integers $r_l \in \mathbb{N}^+$, which matches our two possible LTS relations of Ch. 3.1. In the initial clustering phase we assign every tetrahedron $k \in 1, \ldots K$ to the matching cluster based on the local time step Δt_k^{CFL}.

Note that our initial clustering does not require any connectivity of tetrahedrons sharing the same cluster. This property proves to be crucial when using LTS in application runs. Here, the used unstructured, adaptive, tetrahedral meshes tend to result in clusters scattered loosely throughout the computational domain. We discuss application-specific details of LTS in chapters 8 and 11.

NORMALIZATION To reduce the LTS overhead we allow per tetrahedron k only a single buffer $\mathcal{B}_k$ and a single set of derivatives $\mathcal{D}_k$. This step reduces the amount of data stored per element, but also the number of possible LTS configurations. While the derivatives $\mathcal{D}_k$ allow for face-neighbors with multiple different smaller time steps, the limitation to a single buffer allows for face-neighbors with a unique larger time step only. Additionally we limit the dependency of clusters to a single time level. Therefore we restrict face-neighbors k_i of a tetrahedron $k \in C_l$ to the direct neighboring clusters only: $k_i \in C_{l-1}$, if $l > 1$ or $k_i \in C_l$ or $k_i \in C_{l+1}$, if $l < L$. This step reduces heterogeneity of the final algorithm, e.g. by reducing the number of possible sends and receives in a distributed memory setting.

As shown in lines 1-15 of Alg. 1, we impose these requirements by iterating over the mesh. If the cluster id l of an element $k \in C_l$ is larger than the follow-up id $l_{\text{min}} + 1$ of the minimum neighboring cluster id l_{min}: $l > l_{\text{min}} + 1$, we move element k to cluster $C_{l_{\text{min}}+1}$.

Moving an element to a lower cluster might lead to violations of our normalization-criterion after the iteration, if we touched any face-neighbor of this element already. Therefore in lines 17-22 of Alg. 1 we repeat our procedure until we reach a consistent state.

3.3 MULTIPLE PARTITIONS

In this chapter we discuss the high-level layout of our clustered local time stepping algorithm for settings with multiple partitions. The design aims at scalability when utilizing high node-numbers in distributed memory settings using the Message Passing Interface (MPI). We recapitulate that only the second step (50) of our two-step scheme depends on face-neighboring information. In combination with the clustering (65) and our normalization we can define an efficient asynchronous communication scheme for multiple partitions.

PARTITIONING SeisSol uses static, adaptive, unstructured, tetrahedral meshes for discretization of the spatial domain. In a multi-partition setting we have to assign the elements of our mesh to dif-

Algorithm 1 Normalize clustering

```
1: procedure maximumDifference
2:     r=0
3:     for $k \in 1, \dots, K$ do
4:         $l_k = c_{\text{id}}[k]$                          ▷ Get cluster of the element.
5:         $l_{\min} = l_k$
6:         for $k_i \in k_1, k_2, k_3, k_4$ do               ▷ Iterate over face-neighbors.
7:             $l_{\min} = \min(l_{\min}, c_{\text{id}}[k_i])$   ▷ Derive minimum id of face-neighbors.
8:         end for
9:         if $l_k > l_{\min} + 1$ then
10:            $c_{\text{id}}[k] = l_{\min} + 1$              ▷ Lower time step / associated id.
11:            r = r + 1                                  ▷ Increase reduction counter.
12:        end if
13:    end for
14:    return r
15: end procedure
16:
17: procedure normalizeClustering
18:    r=1
19:    while r > 0 do
20:        r = maximumDifference()
21:    end while
22: end procedure
```

ferent computational resources. Therefore in the partitioning step, which is part of preprocessing, we have to find a balance between *load-balancing* and *communication*.

We use the graph partitioning tool METIS [40] to accomplish this goal. METIS creates the dual-graph of our mesh. The vertices in the dual-graph are associated with our tetrahedrons and edges connecting vertices in the dual-graph are equivalent to tetrahedral faces. Every vertex and edge in the dual-graph has an associated weight, which defaults to 1. Given the dual-graph and corresponding weights we use k-way partitioning of METIS. Objective function is to balance the sum of the vertex weights per partition equally (load balancing) and to reduce the overall edge-cut of edges connecting partitions (communication avoidance).

In our clustered LTS scheme we use for all elements of a cluster C_l the ratio between the cluster's time step $r_1 \cdot \ldots \cdot r_{l-1} \cdot \Delta t^{\text{CFL}}_{\text{min}}$ and the time step $\Delta t^{\text{CFL}}_{\text{min}}$ of the first cluster to derive the vertex weights in the dual-graph. An element of the first cluster C_1 updates $r_1 \cdot \ldots \cdot r_{l-1}$ more often than an element of C_l. Therefore for a cluster C_l we set the weight of every associated vertex to $w_l = 1/\left(r_1 \cdot \ldots \cdot r_{l-1}\right)$. In practice we only use the initial clustering for our partitioning strategy, because the normalization step (see Alg. 1) reduces the time step of very few elements only. Ch. 8 and Ch. 11 apply our clustering to two application-specific examples using different cluster-configurations. Here, the normalization reduces the time step of less than 1.1 ‰ of the elements in every considered configuration.

Note that our approach is fairly simple, but proves to be efficient at scale. However the partitioning step is completely decoupled from the actual simulation and therefore might be optimized in future work. Possible extensions could derive more advanced models for the vertex weights, e.g. also consider required integrations from derivatives via (29) in LTS settings. Additionally, non-constant weights for the edges could be used to account for network topologies and LTS dependent communication frequencies and communication volumes between elements. As we will see in chapters 7-11 SeisSol's performance characteristics are completely known a-priori. It follows that in the ideal case the partitioning would include an auto-tuning step, which also resolves cluster dependencies and adjusts the partitioning accordingly. This is possible even without running the actual setup for which the partitioning is derived.

COMMUNICATION LAYERS Inside a partition p we denote the individual elements belonging to cluster C_l with $C_{l,p}$. Note that not necessarily all cluster-partition combinations $C_{l,p}$ exist, since a partition might only have elements of a subset of clusters. To ease notation we also use the term *cluster* when referring to $C_{l,p}$.

For every cluster $C_{l,p}$ we introduce a communication layer and strictly separate it from interior elements. The communication layer contains all elements required for computation, which are involved in communication. In contrast the interior is completely independent from communication within a time step of $C_{l,p}$.

A communication layer itself is split into a ghost layer, face-neighbors of elements in $C_{l,p}$ which reside in neighboring partitions, and a copy layer, elements of $C_{l,p}$ in our partition p, which have face-neighbors located in neighboring partitions. Both, the ghost and the copy layer, are further subdivided in communication regions. Each region is associated with a neighboring pair correlating uniquely to a specific cluster-partition combination of the neighboring elements.

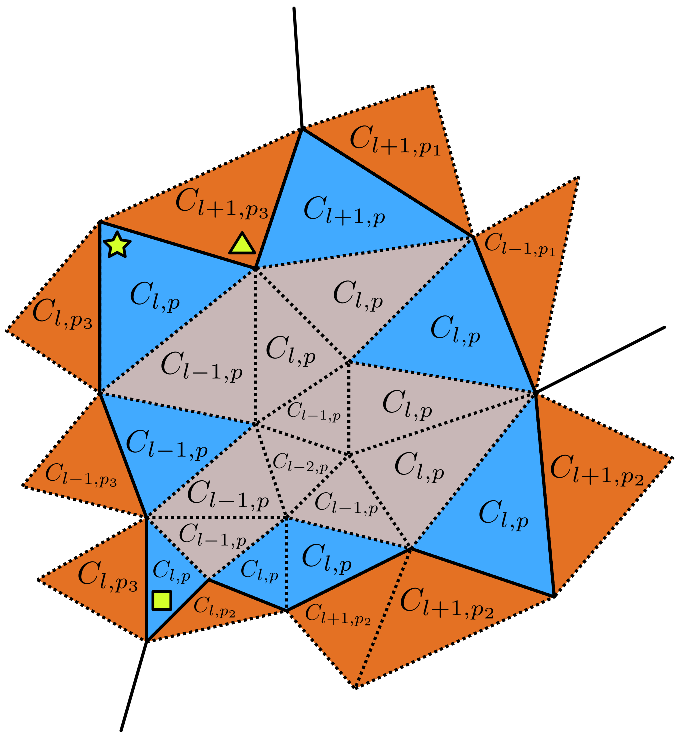
EXAMPLE Fig. 3 visualizes our clustered LTS scheme in a distributed memory setting. For illustrative purposes our example consists of a small triangular partition p including communication layers. Neighboring partitions are p_1, p_2 and p_3. Partition-boundaries are highlighted via solid lines in Fig. 3. Partition p holds elements of time stepping clusters C_{l-2}, C_{l-1}, C_l and C_{l+1} annotated with $C_{l-2,p}$, $C_{l-1,p}$, $C_{l,p}$, $C_{l+1,p}$ respectively. The copy and ghost layers cover elements of several cluster-partition combinations. In Fig. 3a we color all elements of the copy layers in blue, those of the ghost layers in orange and interior elements in gray.

Fig. 3b shows the specific layout of time stepping cluster C_l in partition p: $C_{l,p}$. The interior of $C_{l,p}$ consists of four elements and is highlighted in gray. The six elements of $C_{l,p}$'s copy layer are colored in blue. Our copy layer is subdivided in five different copy regions. Every of the copy regions is visualized using a different pattern in the coloring scheme. We use the same patterns for the ghost regions matching the respective patterns of the associated copy regions. For example the two-element copy region of $C_{l,p}$ communicating with C_{l+1,p_2} has a flat color. The corresponding ghost region consists of three elements of C_{l+1,p_2} and is orange.

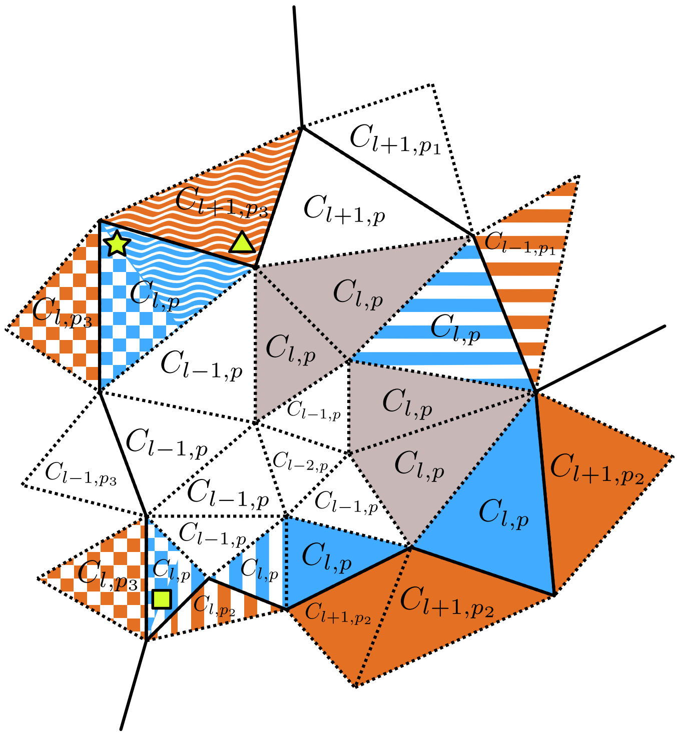
A special case is the copy-element marked with a yellow star. This element neighbors the two time stepping clusters C_l and C_{l+1} in partition p_3. Thus the same element can be part of multiple copy regions. In Fig. 3b the two copy regions are illustrated via the checkerboard and wavy pattern.

The copy-element marked with a yellow square is also part of two copy-regions. Here the element neighbors the same cluster C_l in two different partitions. Consequently it is part of two copy regions. These are the vertically-striped region related with C_{l,p_2} and the checkered region associated with C_{l,p_3}. Analogue the ghost-element with a yellow triangle is part of two ghost-regions.

We address this issue in the implementation-specific Ch. 5.3 by duplicating the corresponding elements in memory.



(a) Elements of the interior, copy layers and ghost layers are colored for all time stepping clusters.



(b) Only the interior, copy layer and ghost layer of $C_{l,p}$ are colored. Patterns in the coloring denote different communication regions in the copy and ghost layer.

Figure 3: Exemplary clustered local time stepping configuration for a partition p. Interior elements of the partition are gray, copy layer elements blue and elements of the ghost layers orange. The elements marked with a yellow star, square and triangle are part of more than one communication region.

3.4 TIME MANAGEMENT

Following our considerations of chapters 3.1 and 3.2 we are able to derive a clustered LTS setup starting from arbitrary per-element time steps Δt_k. Additionally, in Ch. 3.3, we derived the high-level structure of clustered LTS in multiple partitions. This step included partitioning and the introduction of copy and ghost regions for communication between neighboring partitions. From an algorithmic viewpoint the only missing part for an operative clustered LTS scheme is the *time management*.

WORK-ITEMS AND WORK-GROUPS In time management we decide dynamically how to advance our clusters forward in time. For this purpose we dynamically generate and (re-)order individual work-items at runtime.

For the definition of our work-items we distinct between elements in the copy layer and interior elements of every $C_{l,p}$ (see Ch. 3.3). We further use our two-step scheme in (49) and (50) to define two work-items each for the copy layer and for the interior.

In every partition p we group the local work-items in work-groups. As abbreviations for our work-groups we use $\mathcal{L}_p^{\text{int}}$ for the local operations (see (49)) on interior elements and $\mathcal{L}_p^{\text{cop}}$ for those on copy layer elements. Analogue we use $\mathcal{N}_p^{\text{int}}$ and $\mathcal{N}_p^{\text{cop}}$ for the neighboring updates (see (50)) of the interior and copy layer. For example adding cluster $C_{l,p}$ to work-group $\mathcal{L}_p^{\text{cop}}$ implicitly means that we generate a work-item, which is required to perform the local operations for all copy layer elements of $C_{l,p}$.

WORK-ITEM GENERATION To utilize our work-items and -groups we have to decide when we are allowed to create a new work-item by adding a cluster $C_{l,p}$ to the respective work-group. The generation of work-items is required to respect dependencies in time discussed as part of Ch. 3.1. For this purpose we define a set of conditions under which a new work-item is allowed to be created. Our design will be completely partition-local, even though work-items of the copy layer depend on data of neighboring partitions. Thus, after derivation of our local generation procedures, we discuss how to decide dynamically if a cluster in $\mathcal{L}_p^{\text{cop}}$ or $\mathcal{N}_p^{\text{cop}}$ is eligible for an update or required to wait for communication.

Whenever the processing of a work-item is finished and the corresponding cluster is dropped from any of the work-groups, we might be able to create a new work-item.

Alg. 2 shows the complete process of our item generation. In lines 1-11 we decide based on six conditions if a per-partition cluster $C_{l,p}$ meets all requirements to be added to $\mathcal{L}_p^{\text{int}}$ and $\mathcal{L}_p^{\text{cop}}$. Input arguments of the procedure are $C_{l,p}$ itself, the synchronization time t^{sync} and the

Algorithm 2 Generation of work items

```
1:  procedure generateLocal($C_{l,p}$, $t^{\text{sync}}$, $t^{\text{dofs}}_{l-1,p}$, $t^{\text{pred}}_{l+1,p}$, $t^{\text{dofs}}_{l+1,p}$)
2:      $e \leftarrow ( C_{l,p} \notin \mathcal{L}^{\text{int}}_p) \wedge (C_{l,p} \notin \mathcal{L}^{\text{cop}}_p)$
3:      $e \leftarrow ( C_{l,p}.t^{\text{dofs}} < t^{\text{sync}} ) \wedge e$
4:      $e \leftarrow ( C_{l,p}.t^{\text{pred}} \leq t^{\text{dofs}}_{l-1,p} ) \wedge e$
5:      $e \leftarrow ( C_{l,p}.t^{\text{pred}} = C_{l,p}.t^{\text{dofs}} ) \wedge e$
6:      $e \leftarrow ( (C_{l,p}.t^{\text{pred}} < t^{\text{pred}}_{l+1,p}) \vee (C_{l,p}.t^{\text{pred}} \leq t^{\text{dofs}}_{l+1,p}) ) \wedge e$
7:      if $e$ then                ▷ Add local items if all conditions are met.
8:          $\mathcal{L}^{\text{int}}_p \leftarrow C_{l,p} \cup \mathcal{L}^{\text{int}}_p$
9:          $\mathcal{L}^{\text{cop}}_p \leftarrow C_{l,p} \cup \mathcal{L}^{\text{cop}}_p$
10:     end if
11: end procedure
12:
13: procedure generateNeighboring($C_{l,p}$, $t^{\text{sync}}$, $t^{\text{pred}}_{l-1,p}$, $t^{\text{pred}}_{l+1,p}$)
14:     $e \leftarrow ( C_{l,p} \notin \mathcal{N}^{\text{int}}_p) \wedge (C_{l,p} \notin \mathcal{N}^{\text{cop}}_p)$
15:     $e \leftarrow ( C_{l,p}.t^{\text{dofs}} < t^{\text{sync}} ) \wedge e$
16:     $e \leftarrow ( C_{l,p}.t^{\text{pred}} \leq t^{\text{pred}}_{l-1,p} ) \wedge e$
17:     $e \leftarrow ( C_{l,p}.t^{\text{pred}} > C_{l,p}.t^{\text{dofs}} ) \wedge e$
18:     $e \leftarrow ( C_{l,p}.t^{\text{pred}} \leq t^{\text{pred}}_{l+1,p} ) \wedge e$
19:     if $e$ then      ▷ Add neighboring items if all conditions are met.
20:         $\mathcal{N}^{\text{int}}_p \leftarrow C_{l,p} \cup \mathcal{N}^{\text{int}}_p$
21:         $\mathcal{N}^{\text{cop}}_p \leftarrow C_{l,p} \cup \mathcal{N}^{\text{cop}}_p$
22:     end if
23: end procedure
24:
25: procedure generateWorkItems($C_{l,p}$, $t^{\text{sync}}$)
26:     $t^{\text{pred}}_{l-1,p} \leftarrow t^{\text{dofs}}_{l-1,p} \leftarrow$ limits::max()
27:     $t^{\text{pred}}_{l+1,p} \leftarrow t^{\text{dofs}}_{l+1,p} \leftarrow$ limits::max()
28:
29:     if $\exists C_{l-1}, p$ then        ▷ Get times of previous cluster if existent
30:         $t^{\text{pred}}_{l-1,p} \leftarrow C_{l-1,p}.t^{\text{pred}}$
31:         $t^{\text{dofs}}_{l-1,p} \leftarrow C_{l-1,p}.t^{\text{dofs}}$
32:     end if
33:     if $\exists C_{l+1}, p$ then            ▷ Get times of next cluster if existent
34:         $t^{\text{pred}}_{l+1,p} \leftarrow C_{l+1,p}.t^{\text{pred}}$
35:         $t^{\text{dofs}}_{l+1,p} \leftarrow C_{l+1,p}.t^{\text{dofs}}$
36:     end if
37:
38:     generateLocal($C_{l,p}$, $t^{\text{sync}}$, $t^{\text{dofs}}_{l-1,p}$, $t^{\text{pred}}_{l+1,p}$, $t^{\text{dofs}}_{l+1,p}$)
39:     generateNeighboring($C_{l,p}$, $t^{\text{sync}}$, $t^{\text{pred}}_{l-1,p}$, $t^{\text{pred}}_{l+1,p}$)
40: end procedure
```

time levels $t^{\text{dofs}}_{l-1,p}$, $t^{\text{pred}}_{l+1,p}$ and $t^{\text{dofs}}_{l+1,p}$ of the neighboring clusters. In the context of Alg. 2, *pred* always refers to the time level of the predictions, which is equivalent to the time of the last element local update (49). Analogue *dofs* refers to the time of the last full time step via (50). Lines 26-36 initialize the neighboring time levels with a very large value, if cluster C_{l-1} or cluster C_{l+1} does not exist in partition p or at all ($l = 1$ or $l = L$). This initialization ensures that in the corresponding corner-cases all of the following conditions are still valid.

The first condition in line 2 of Alg. 2 ensures that work-groups $\mathcal{L}^{\text{int}}_p$ and $\mathcal{L}^{\text{cop}}_p$ are not in possession of $C_{l,p}$ already. Next, in line 3, we check if the DOFs of $C_{l,p}$ are at the desired synchronization time, which all clusters' elements have to reach. We create work items only if this is false. Otherwise, after all clusters reached our synchronization time, we would either process the state via external routines, for example by writing output, and continue with the next synchronization point or shutdown our simulation completely because we reached the final simulation time.

New time predictions update the buffers or derivatives or both of the previous time step. Therefore we are allowed to generate local operations only, if no face-neighbors of elements in $C_{l,p}$ require $C_{l,p}$'s current time predictions in their neighboring flux computations. Due to our normalization in Alg. 1 it is sufficient to check this condition for $C_{l-1,p}$, $C_{l,p}$ and $C_{l+1,p}$. In line 4 of Alg. 2 we verify that the DOFs of $C_{l-1,p}$ reached the same time as $C_{l,p}$'s time prediction and therefore $C_{l-1,p}$ is finished using $C_{l,p}$'s current derivatives. Similar in line 5 we verify that $C_{l,p}$ used its own predictions for the neighboring update. The statement in line 6 is more complex, because $C_{l+1,p}$ operates on buffers of $C_{l,p}$, which are summed over multiple time steps of $C_{l,p}$. Here generation of new local work items is valid in two cases. In the first case $C_{l,p}$'s prediction time is before $C_{l+1,p}$'s prediction time. This means that $C_{l,p}$ is required to add at least one more set of time integrated DOFs to its respective buffers, before $C_{l+1,p}$ is able to use this data. In the second case $C_{l,p}$'s and $C_{l+1,p}$'s prediction times are in sync. Here we have to ensure that $C_{l+1,p}$ already used $C_{l,p}$'s buffers in the neighboring update step (50).

In lines 13-23 we decide if we are allowed to add a cluster to work-groups $\mathcal{N}^{\text{int}}_p$ and $\mathcal{N}^{\text{cop}}_p$ for neighboring updates. Compared to the generation of local work-items, the neighboring updates only change the elements' DOFs. Consequently no information, which might still be required, gets overwritten. However we have to ensure that all time predictions for the neighborings updates are available. Again in lines 26-36 of Alg. 2 the input time levels $t^{\text{pred}}_{l-1,p}$ and $t^{\text{pred}}_{l+1,p}$ are initialized with a large number if $C_{l-1,p}$ or $C_{l+1,p}$ or both are not available.

The condition in line 14 ensures that $C_{l,p}$ is neither scheduled in $\mathcal{N}^{\text{int}}_p$ nor in $\mathcal{N}^{\text{cop}}_p$. As for the local items we only add neighboring items if the synchronization time is not reached (line 15). Line 16 checks that the predictions of $C_{l-1,p}$ are available, line 17 checks for those of $C_{l,p}$ and line 18 for the predictions of $C_{l+1,p}$

3.5 SCHEDULING

In Ch. 3.3 we divided the elements of each per-partition time stepping cluster $C_{l,p}$ into an interior part and a copy layer. Additionally we subdivided every copy layer again into copy regions and introduced ghost regions as the counterparts of the copy regions. During a simulation we have to send time predictions of the elements in a copy region to the neighboring partition. Conversely predictions of elements in a ghost region have to be received. Ch. 3.4 introduced work-items and -groups within a partition, but left synchronization across partition boundaries open. In this chapter we bring both concepts together and discuss how and when our work-items are processed. This includes introduction of our asynchronous inter-partition communication.

CLUSTER OPERATIONS Alg. 3 shows the operations a cluster $C_{l,p}$ might perform. The first procedure *localCopy* in lines 1-9 returns immediately in the case of ongoing sends (lines 2-4). Thus we ensure that $C_{l,p}$'s predictions in copy regions reached the respective ghost regions located in neighboring partitions successfully before an update.

The generation of local work-items for cluster $C_{l,p}$ directly follows a full update of the DOFs (see Alg. 2, line 5). Therefore in line 5 of Alg. 3 we issue asynchronous receives for new time predictions in the ghost regions. Whether a certain ghost region requires new data depends on the relation of $C_{l,p}$ and its neighboring cluster. We request new data from neighbors with a smaller or identical time step in every of $C_{l,p}$'s time steps. Neighbors with a larger time step send derivatives. Following (65) we evaluate the derivatives r_l times in general (synchronization is the only exception). Consequently for neighbors with larger time steps we issue receives initially after synchronization and then after every r_l^{th} time step.

The call of *localCopyOps* for $C_{l,p}$ in line 6 updates the time predictions of all elements in the copy layer. Additionally it updates the DOFs of these elements with the local contributions according to (49).

In line 7 we call the procedure *sendCopyRegions*, which is again communication related. Following our time predictions computed in *localCopyOps* we might have new data available other partitions rely on. Analogue to the receives in line 5 the communication frequency depends on the time step relation of $C_{l,p}$ and the neighboring clusters associated with the respective copy regions. We asynchronously send time predictions for copy regions neighboring smaller or equal time step clusters in every of $C_{l,p}$'s time steps. If a copy regions is associated with a neighboring cluster with a larger time step, we have to

Algorithm 3 Cluster Operations

```
procedure localCopy()
    if $\neg$this.sendsFinished() then
        return false        ▷ Immediately return if sends are ongoing.
    end if
    this.receiveGhostRegions()
    this.localCopyOps()
    this.sendCopyRegions()
    return true
end procedure

procedure neighboringCopy()
    if $\neg$this.receivesFinished() then
        return false ▷ Immediately return if receives are ongoing.
    end if
    this.neighCopyOps()
    return true
end procedure

procedure localInterior()
    this.localIntOps()
end procedure

procedure neighboringInterior()
    this.neighIntOps()
end procedure
```

sum up r_l time steps in the buffers before sending them. Therefore we issues sends of these copy regions every r_l time steps. The only exception is a synchronization point in time, where we might send out the data earlier, because the buffers might be completed in less than r_l time steps.

The two procedures *localInterior* and *neighboringInterior* in lines 19-25 of Alg. 3 operate on the interior elements of $C_{l,p}$. *localInterior* computes the time predictions and updates the DOFs with the local contributions according to (49). *neighboringInterior* updates the DOFs of all interior elements with the neighboring elements' contributions (see (50)).

SCHEDULING Alg. 4 shows our scheduling scheme as part of the *advanceInTime* procedure. At the entry point of *advanceInTime* we assume that all clusters are synchronized in time and have to reach a common synchronization point t^{sync}.

Due to the synchronization we are allowed to perform a single time step of element-local operations for all clusters. Therefore, before entering our time marching loop (lines 6-34), in lines 2-4 we call *generateWorkItems* for all clusters and effectively add all clusters of partition p to $\mathcal{L}_p^{\text{cop}}$ and to $\mathcal{L}_p^{\text{int}}$.

Next we enter the time marching loop covering lines 6-34. Line 6 defines the only exit point of our time marching scheme. It states that all work-groups $\mathcal{L}_p^{\text{cop}}$, $\mathcal{N}_p^{\text{cop}}$, $\mathcal{L}_p^{\text{int}}$ and $\mathcal{N}_p^{\text{int}}$ have to be empty on exit. Whenever we finish a work-item, we instantly check if the changed state of the system allows us to create new work-items. Thus empty work-groups are equivalent to reaching the synchronization time t^{sync} in all clusters.

Inside the time marching loop our first inner-loop iterates over all clusters in $\mathcal{L}_p^{\text{cop}}$. Every-time the local operations of the copy layer elements were successful, we remove the cluster from $\mathcal{L}_p^{\text{cop}}$ (line 9) and generate new work-items if possible (line 10). In the case that *localCopy* returned unsuccessful, the current cluster has ongoing sends (see Alg. 3) and we leave the cluster in $\mathcal{L}_p^{\text{cop}}$ for the time being.

Our next inner-loop in lines 14-19 iterates over all clusters in $\mathcal{N}_p^{\text{cop}}$. Similar to our first inner-loop, we leave a cluster untouched in $\mathcal{N}_p^{\text{cop}}$ if a receive request for any of the cluster's ghost regions is ongoing in *neighboringCopy*. Else we remove the cluster from $\mathcal{N}_p^{\text{cop}}$ (line 16) and generate new work-items if possible (line 17).

The remaining statements in the time marching loop of Alg. 4 operate on the inner elements of the clusters. First, if available, we get the cluster on top of work-group $\mathcal{L}_p^{\text{int}}$ (line 22) or $\mathcal{N}_p^{\text{int}}$ (line 29). Then we compute the local operations (line 23) or neighboring updates (line 30) for the interior elements of the respective cluster. The last two steps, operating on both work-groups, remove the cluster from $\mathcal{L}_p^{\text{int}}$

Algorithm 4 Scheduling

```
1: procedure advanceInTime($t^{\text{sync}}$)
2:     for all $C_{l,p} \in p$ do
3:         generateWorkItems($C_{l,p}$, $t^{\text{sync}}$)
4:     end for
5:
6:     while $\mathcal{L}_p^{\text{cop}} \neq \emptyset \wedge \mathcal{N}_p^{\text{cop}} \neq \emptyset \wedge \mathcal{L}_p^{\text{int}} \neq \emptyset \wedge \mathcal{N}_p^{\text{int}} \neq \emptyset$ do
7:         for all $C_{l,p} \in \mathcal{L}_p^{\text{cop}}$ do
8:             if $C_{l,p}$.localCopy() then
9:                 $\mathcal{L}_p^{\text{cop}} = \mathcal{L}_p^{\text{cop}} \setminus C_{l,p}$
10:                generateWorkItems($C_{l,p}$, $t^{\text{sync}}$)
11:            end if
12:        end for
13:
14:        for all $C_{l,p} \in \mathcal{N}_p^{\text{cop}}$ do
15:            if $C_{l,p}$.neighboringCopy() then
16:                $\mathcal{N}_p^{\text{cop}} = \mathcal{N}_p^{\text{cop}} \setminus C_{l,p}$
17:                generateWorkItems($C_{l,p}$, $t^{\text{sync}}$)
18:            end if
19:        end for
20:
21:        if $\mathcal{L}_p^{\text{int}} \neq \emptyset$ then
22:            $C_{l,p} \leftarrow \mathcal{L}_p^{\text{int}}$.top()
23:            $C_{l,p}$.localInterior()
24:            $\mathcal{L}_p^{\text{int}} = \mathcal{L}_p^{\text{int}} \setminus C_{l,p}$
25:            generateWorkItems($C_{l,p}$, $t^{\text{sync}}$)
26:        end if
27:
28:        if $\mathcal{N}_p^{\text{int}} \neq \emptyset$ then
29:            $C_{l,p} \leftarrow \mathcal{N}_p^{\text{int}}$.top()
30:            $C_{l,p}$.neighboringInterior()
31:            $\mathcal{N}_p^{\text{int}} = \mathcal{N}_p^{\text{int}} \setminus C_{l,p}$
32:            generateWorkItems($C_{l,p}$, $t^{\text{sync}}$)
33:        end if
34:    end while
35: end procedure
```

(line 24) or $\mathcal{N}_p^{\text{int}}$ (line 31) and generate new work-items if possible (line 25 or 32).

To maximize the overlap of communication and computation our scheduling prioritizes all cluster-operations on copy layer elements. In every iteration of the time marching loop we iterate over all clusters in $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{N}_p^{\text{cop}}$, but only work on a single cluster in $\mathcal{L}_p^{\text{int}}$ and $\mathcal{N}_p^{\text{int}}$. In practice work-items in $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{N}_p^{\text{cop}}$ have low loads, because of the low surface-area-to-volume ratio of our partitions. Thus we use a simple *first in, first out*-queue for $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{N}_p^{\text{cop}}$.

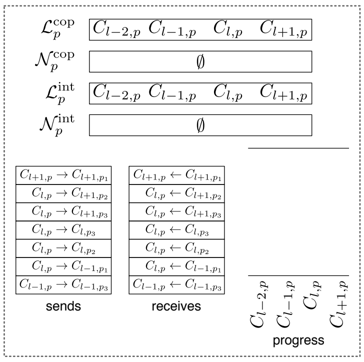
In contrast $\mathcal{L}_p^{\text{int}}$ and $\mathcal{N}_p^{\text{int}}$ have a high load. Here we assume that clusters with small time steps define the crucial path of our simulation and use sorted queues for $\mathcal{L}_p^{\text{int}}$ and $\mathcal{N}_p^{\text{int}}$. Whenever a cluster $C_{l,p}$ is added to either $\mathcal{L}_p^{\text{int}}$ or $\mathcal{N}_p^{\text{int}}$, we use the cluster's id l for the sorting. Hence the top of the queue (lines 22 and 29) always refers to the smallest time step cluster in the respective queue.

EXAMPLE In the following example we use the same triangular LTS configuration for the layout of a partition p as in Ch. 3.3's example. The interior regions, ghost and copy layers of all clusters $C_{l-2,p}$, $C_{l-1,p}$, $C_{l,p}$ and $C_{l+1,p}$ are illustrated in Fig. 3a. Fig. 3b shows the interior, copy regions and ghost regions for cluster $C_{l,p}$.

Fig. 4 and and Fig. 5 show an example execution of our time marching scheme for partition p. The illustrations of the individual states are split into three parts. The upper part shows the current status of our work-groups, empty work-groups are marked with $\emptyset$. Priority in the queues $\mathcal{L}_p^{\text{cop}}$, $\mathcal{N}_p^{\text{cop}}$, $\mathcal{L}_p^{\text{int}}$ and $\mathcal{N}_p^{\text{int}}$ reads from left to right. For example in Fig. 4a $C_{l-2,p}$ has highest priority in $\mathcal{L}_p^{\text{cop}}$.

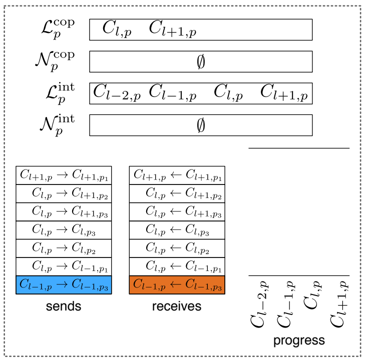
The second part of our states are the two arrays *sends* and *receives* on the lower-left side. *sends* illustrates all outgoing, asynchronous communication, while *receives* shows all incoming, asynchronous communication. Ongoing sends are colored in blue and ongoing receives in orange. Using Fig. 4f as an example, we see two ongoing sends and two ongoing receives. Here $C_{l+1,p}$ sends the time predictions of its respective copy region to C_{l+1,p_1}, analogue $C_{l,p}$ for a region to C_{l,p_3}. In terms of incoming communication, $C_{l+1,p}$ receives ghost region data from C_{l+1,p_1} and $C_{l,p}$ from C_{l+1,p_3}. Further we can map $C_{l,p}$'s active communication to Fig. 3b. In this case $C_{l,p} \rightarrow C_{l+1,p_3}$ sends data of the blue, wavy copy region and $C_{l,p} \rightarrow C_{l,p_3}$ data of the blue, checkered copy region. The receive $C_{l,p} \leftarrow C_{l+1,p_3}$ refers to receiving data for the orange, wavy ghost region.

The third and last part of our states is the lower-right diagram, which shows the *progress* in time of all clusters in partition p. Initially all clusters are in sync (lower horizontal line) and have to reach the next synchronization time t^{sync} in Alg. 4 (upper horizontal line). The latests time predictions (t^{pred} in Alg. 2) for each clusters are visualized by white boxes. The full updates of the DOFs (t^{dofs} in Alg. 2)



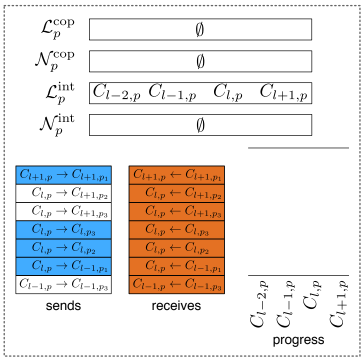
$\mathcal{L}_p^{\text{cop}}$ $C_{l-2,p}$ $C_{l-1,p}$ $C_{l,p}$ $C_{l+1,p}$
$\mathcal{N}_p^{\text{cop}}$ $\emptyset$
$\mathcal{L}_p^{\text{int}}$ $C_{l-2,p}$ $C_{l-1,p}$ $C_{l,p}$ $C_{l+1,p}$
$\mathcal{N}_p^{\text{int}}$ $\emptyset$
sends
receives
progress

(a) Initial state before entering the time-marching loop.



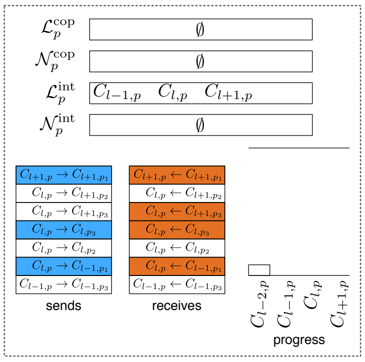
$\mathcal{L}_p^{\text{cop}}$ $C_{l,p}$ $C_{l+1,p}$
$\mathcal{N}_p^{\text{cop}}$ $\emptyset$
$\mathcal{L}_p^{\text{int}}$ $C_{l-2,p}$ $C_{l-1,p}$ $C_{l,p}$ $C_{l+1,p}$
$\mathcal{N}_p^{\text{int}}$ $\emptyset$
sends
receives
progress

(b) State after finishing the first two work-items. One send and one receive are ongoing.



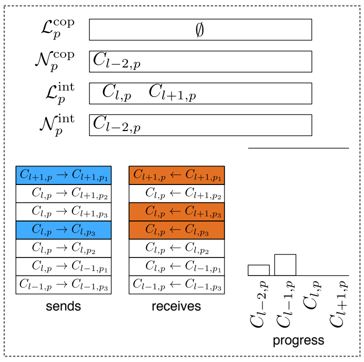
$\mathcal{L}_p^{\text{cop}}$ $\emptyset$
$\mathcal{N}_p^{\text{cop}}$ $\emptyset$
$\mathcal{L}_p^{\text{int}}$ $C_{l-2,p}$ $C_{l-1,p}$ $C_{l,p}$ $C_{l+1,p}$
$\mathcal{N}_p^{\text{int}}$ $\emptyset$
sends
receives
progress

(c) State after emptying $\mathcal{L}_p^{\text{cop}}$ for the first time. Four sends and seven receives are ongoing.



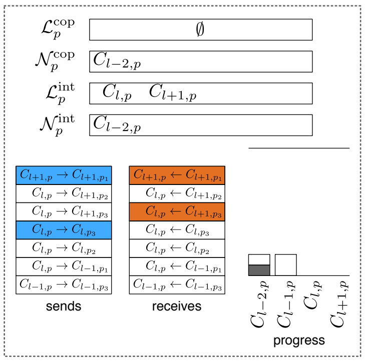
$\mathcal{L}_p^{\text{cop}}$ $\emptyset$
$\mathcal{N}_p^{\text{cop}}$ $\emptyset$
$\mathcal{L}_p^{\text{int}}$ $C_{l-1,p}$ $C_{l,p}$ $C_{l+1,p}$
$\mathcal{N}_p^{\text{int}}$ $\emptyset$
sends
receives
progress

(d) $C_{l-2,p}$ finished its first time prediction. Three sends and four receives are ongoing.



$\mathcal{L}_p^{\text{cop}}$ $\emptyset$
$\mathcal{N}_p^{\text{cop}}$ $C_{l-2,p}$
$\mathcal{L}_p^{\text{int}}$ $C_{l,p}$ $C_{l+1,p}$
$\mathcal{N}_p^{\text{int}}$ $C_{l-2,p}$
sends
receives
progress

(e) $C_{l-1,p}$ finished its first time predictions. Two sends and three receives are ongoing.



$\mathcal{L}_p^{\text{cop}}$ $\emptyset$
$\mathcal{N}_p^{\text{cop}}$ $C_{l-2,p}$
$\mathcal{L}_p^{\text{int}}$ $C_{l,p}$ $C_{l+1,p}$
$\mathcal{N}_p^{\text{int}}$ $C_{l-2,p}$
sends
receives
progress

(f) $C_{l-2,p}$ finished its first time step and computed another time prediction. Two sends and two receives are ongoing.

Figure 4: Possible series of states resulting from execution of Alg. 4. The four uppermost bars show the current configuration of the work-groups $\mathcal{L}_p^{\text{cop}}$, $\mathcal{N}_p^{\text{cop}}$, $\mathcal{L}_p^{\text{int}}$ and $\mathcal{N}_p^{\text{int}}$. Priority of the work-items reads from left to right. The two arrays *sends* and *receives* show all possible communication. Ongoing sends are blue, ongoing receives orange. The right diagram summarizes the states' *progress*. Finished time steps are gray, time predictions white.

are illustrated via gray boxes. Fig. 5d, for example, shows a progress with five finished full updates and eight computed time predictions. Three of these predictions still reach further in time than the corresponding DOFs. At this state cluster $C_{l-2,p}$ performed three full time steps and four time predictions, one of which still covers future updates of $C_{l-2,p}$. Cluster $C_{l-1,p}$ finished one complete time step and is in possession of a time prediction further in time than its DOFs. In contrast, $C_{l,p}$'s DOFs are at the same level as its latest prediction. However, $C_{l-1,p}$ still operates on $C_{l,p}$'s latest time predictions, while $C_{l+1,p}$ will use $C_{l,p}$'s respective summed time integrated DOFs later. $C_{l+1,p}$ only computed a single time prediction in Fig. 5d.

Obviously, in partition p, $C_{l-2,p}$ has the smallest time step and the other clusters' time steps are defined by the rates $r_{l-2} = 2$, $r_{l-1} = 2$ and $r_l = 3$ (see (65)). This means that every of $C_{l-1,p}$'s elements updates two times more of than the single element of $C_{l-2,p}$, $C_{l,p}$'s elements updates four time more often, and those of $C_{l+1,p}$ twelve times more often than $C_{l-2,p}$'s element.

Now we discuss the execution of our *advanceInTime* procedure in Alg. 4 and how it could lead to the different states in our example. This is only one possible control flow starting from the initial synchronized state, because of the used asynchronous communication and the dynamics of our scheduling scheme in case of ongoing communication. However note that our final results are still deterministic and bit-reproducible since we preserve the logical order of the operations.

The first state in Fig. 4a shows the initial generation of work-items after execution of lines 2-4 in Alg. 4. Here all clusters are initially added to both work-groups $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{L}_p^{\text{int}}$ performing local operations.

Next our execution enters the time-marching loop in lines 6-34. We do not have any ongoing receives, thus the first inner-loop of Alg. 4 computes the local operations for all clusters' copy layers. Fig. 4b shows partition p's state after the first two iterations. The associated work items of clusters $C_{l-2,p}$ and $C_{l-1,p}$ finished successfully. $C_{l-2,p}$ consists of a single interior element only, thus all copy layer operations return immediately and no communication is issued. Cluster $C_{l-1,p}$ has a single ghost and copy region, consequently a single send and a single receive are issued. We asynchronously send the time prediction of $C_{l-1,p}$'s single-element copy region to C_{l-1,p_3} and receive the time prediction of the corresponding single-element ghost region (see Fig. 3a).

The state after all iterations over $\mathcal{L}_p^{\text{cop}}$ is illustrated in Fig. 4c. Now all clusters finished their first local work-items and work-group $\mathcal{L}_p^{\text{cop}}$ is empty. We have a total of four ongoing sends and and six receives. No communication was issued for $C_{l,p} \rightarrow C_{l+1,p_2}$ and $C_{l,p} \rightarrow C_{l+1,p_3}$, since $C_{l,p}$'s time buffers are incomplete by now. The only finished request is $C_{l-1,p} \rightarrow C_{l-1,p_3}$ and was issued in the previous state (see Fig. 4b) already. Therefore, while working on $C_{l,p}$ and $C_{l+1,p}$, cluster $C_{l-1,p}$ sent its copy layer information.

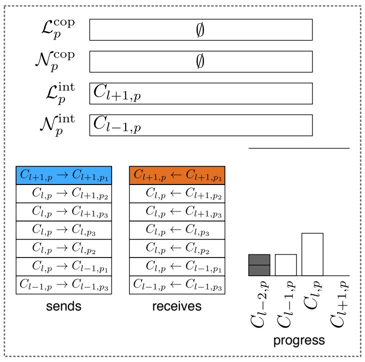
Work-group $\mathcal{N}_p^{\text{cop}}$ is empty and a local, interior work-item of $\mathcal{L}_p^{\text{int}}$ is computed next. Fig. 4d shows the state after the first execution of lines 21-26 in Alg. 4. We prioritized the smallest time step cluster $C_{l-2,p}$ in $\mathcal{L}_p^{\text{int}}$ and computed its interior local operations. Now $C_{l-2,p}$ successfully finished both of its first local work-items. Accordingly our progress diagram shows that $C_{l-2,p}$'s time prediction of the first time step is available. Meanwhile one send and three receives finished successfully.

The execution of our time-marching loop in Alg. 4 continues and we encounter that work-groups $\mathcal{N}_p^{\text{int}}$, $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{N}_p^{\text{cop}}$ are empty. We reach lines 21-26 again and compute a single, prioritized work-item of $\mathcal{L}_p^{\text{int}}$ again. This time cluster $C_{l-1,p}$ is on top of $\mathcal{L}_p^{\text{int}}$ and finishes all of its first time step's predictions. The call to *generateWorkItems* encounters that $C_{l-2,p}$ qualifies for neighboring updates and adds the cluster to groups $\mathcal{N}_p^{\text{cop}}$ and $\mathcal{N}_p^{\text{int}}$ in Fig. 4e's associated state. Additionally send $C_{l,p} \rightarrow C_{l-1,p_1}$ and receive $C_{l,p} \leftarrow C_{l-1,p_1}$ finished in background.

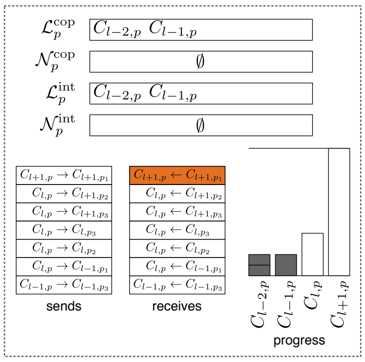
The transition from Fig. 4e to Fig. 4f covers the computation of multiple work-items associated with $C_{l-2,p}$. We start with the two work-items in $\mathcal{N}_p^{\text{int}}$ and $\mathcal{N}_p^{\text{cop}}$. After removing $C_{l-2,p}$ from $\mathcal{N}_p^{\text{cop}}$ we determine, in line 17 of Alg. 4, that $C_{l-2,p}$ qualifies for insertion in $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{L}_p^{\text{int}}$. Thus we compute both recently added work-items and add $C_{l-2,p}$ once again to $\mathcal{N}_p^{\text{cop}}$ and $\mathcal{N}_p^{\text{int}}$ when calling *generateWorkItems* in line 10. While executing all of the these work-items we assume that receive $C_{l,p} \leftarrow C_{l,p_3}$ finished in background. The resulting state is illustrated in Fig. 4f.

All of the following example states cover computation of multiple work items. To reach the state in Fig. 5a from Fig. 4f's state, we start with $\mathcal{N}_p^{\text{cop}}$ and drop $C_{l-2,p}$. Next a single work-item of $\mathcal{L}_p^{\text{int}}$ is processed in lines 21 to 26. $C_{l,p}$ is on top of $\mathcal{L}_p^{\text{int}}$, reaches the next time prediction level for all elements and allows us to generate work-items in $\mathcal{N}_p^{\text{cop}}$ and $\mathcal{N}_p^{\text{int}}$ for $C_{l-1,p}$'s neighboring updates. Lines 28-33 come next in our time-marching loop. We once again compute the single neighboring update of $C_{l-2,p}$'s element and drop $C_{l-2,p}$ from $\mathcal{N}_p^{\text{int}}$. However, we are are not allowed to add $C_{l-2,p}$ to $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{L}_p^{\text{int}}$ because $C_{l-1,p}$ relies on $C_{l-2,p}$'s summed time buffer for its neighboring updates.

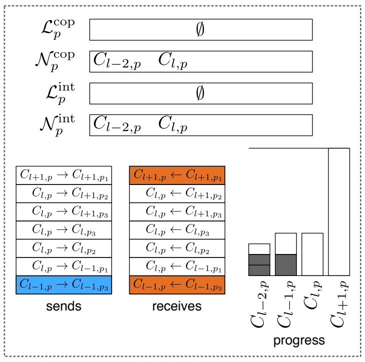
Work-group $\mathcal{L}_p^{\text{cop}}$ remains empty and we continue with $\mathcal{N}_p^{\text{cop}}$ in lines 14-19. $\mathcal{N}_p^{\text{cop}}$ contains cluster $C_{l-1,p}$ at this state. We process the remaining work-item of $\mathcal{N}_p^{\text{cop}}$, which is the neighboring operations of $C_{l-1,p}$'s copy layer element. This is the first time we require ghost layer information from a neighboring partition. In our example we received the required information from C_{l-1,p_3} already in the state il-


$\mathcal{L}_p^{\text{cop}}$ $\emptyset$
$\mathcal{N}_p^{\text{cop}}$ $\emptyset$
$\mathcal{L}_p^{\text{int}}$ $C_{l+1,p}$
$\mathcal{N}_p^{\text{int}}$ $C_{l-1,p}$
sends
receives
progress

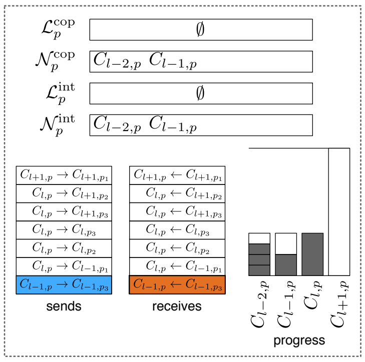
(a) $C_{l-2,p}$ finished its second time step. $C_{l,p}$ computed its first set of time predictions. One send and one receive are ongoing.


$\mathcal{L}_p^{\text{cop}}$ $C_{l-2,p}$ $C_{l-1,p}$
$\mathcal{N}_p^{\text{cop}}$ $\emptyset$
$\mathcal{L}_p^{\text{int}}$ $C_{l-2,p}$ $C_{l-1,p}$
$\mathcal{N}_p^{\text{int}}$ $\emptyset$
sends
receives
progress

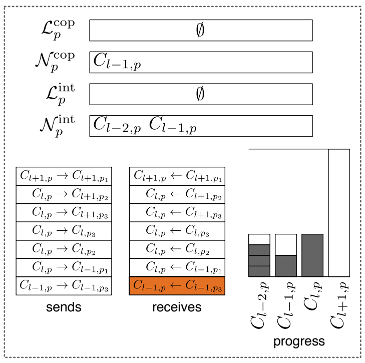
(b) $C_{l-1,p}$ finished its first time step. $C_{l+1,p}$ computed its first complete set of time predictions. One receive is ongoing.


$\mathcal{L}_p^{\text{cop}}$ $\emptyset$
$\mathcal{N}_p^{\text{cop}}$ $C_{l-2,p}$ $C_{l,p}$
$\mathcal{L}_p^{\text{int}}$ $\emptyset$
$\mathcal{N}_p^{\text{int}}$ $C_{l-2,p}$ $C_{l,p}$
sends
receives
progress

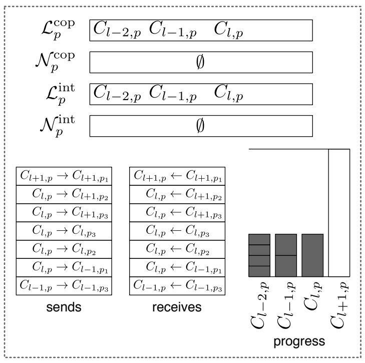
(c) $C_{l-2,p}$ computed its third time prediction and $C_{l-1,p}$ its second set of time predictions. One send and two receives are ongoing.


$\mathcal{L}_p^{\text{cop}}$ $\emptyset$
$\mathcal{N}_p^{\text{cop}}$ $C_{l-2,p}$ $C_{l-1,p}$
$\mathcal{L}_p^{\text{int}}$ $\emptyset$
$\mathcal{N}_p^{\text{int}}$ $C_{l-2,p}$ $C_{l-1,p}$
sends
receives
progress

(d) $C_{l-2,p}$ finished its third time step and computed its fourth time predictions. $C_{l,p}$ finished its first time step. One send and one receive are ongoing.


$\mathcal{L}_p^{\text{cop}}$ $\emptyset$
$\mathcal{N}_p^{\text{cop}}$ $C_{l-1,p}$
$\mathcal{L}_p^{\text{int}}$ $\emptyset$
$\mathcal{N}_p^{\text{int}}$ $C_{l-2,p}$ $C_{l-1,p}$
sends
receives
progress

(e) Only $C_{l-2,p}$ finished its work-item in $\mathcal{N}_p^{\text{cop}}$. $C_{l-1,p}$ is blocked by the ongoing receive.


$\mathcal{L}_p^{\text{cop}}$ $C_{l-2,p}$ $C_{l-1,p}$ $C_{l,p}$
$\mathcal{N}_p^{\text{cop}}$ $\emptyset$
$\mathcal{L}_p^{\text{int}}$ $C_{l-2,p}$ $C_{l-1,p}$ $C_{l,p}$
$\mathcal{N}_p^{\text{int}}$ $\emptyset$
sends
receives
progress

(f) Clusters $C_{l-2,p}$, $C_{l-1,p}$ and $C_{l,p}$ reached a common time level. No communication is ongoing.

Figure 5: Continuation of the possible series in Fig. 4 resulting from an example execution of Alg. 4.

lustrated in 4d, thus we can finish the work-item. The current state together with two more finished communication requests is illustrated in Fig. 5a.

Our time-marching loop continues by computing the local, interior operations of $C_{l+1,p}$ as the only work-item of $\mathcal{L}_p^{\text{int}}$. At this state we are not allowed to add any new work-items to our groups when calling *generateWorkItems* in line 25. $C_{l,p}$ is blocked because it requires $C_{l-1,p}$ to add another set of time predictions to the respective buffers. $C_{l+1,p}$ is required to wait until $C_{l,p}$ finished computing the corresponding time buffers.

The only non-empty work-group $\mathcal{N}_p^{\text{int}}$ comes next in our time-marching loop (see lines 28-33 of Alg. 4). At this point we finish $C_{l-1,p}$'s neighboring updates, drop the cluster from $\mathcal{N}_p^{\text{int}}$ and call *generateWorkItems* in line 32. The finished time step of $C_{l-1,p}$ allows us to add $C_{l-2,p}$ to $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{L}_p^{\text{int}}$ because $C_{l-2,p}$'s time buffer is allowed to be overwritten. Similar $C_{l-1,p}$ does not require its own time predictions anymore and we also add the cluster to $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{L}_p^{\text{int}}$. Fig. 5b illustrates the current state, where we additionally assume that send $C_{l+1,p} \rightarrow C_{l+1,p_1}$ finished.

We continue with $C_{l-2,p}$ and $C_{l-1,p}$ in $\mathcal{L}_p^{\text{cop}}$. The update of $C_{l-1,p}$ is valid because the send $C_{l-1,p} \rightarrow C_{l-1,p_3}$ finished already in the state shown in Fig. 4c. However, the call of *localCopy* in line 8 of Alg. 4 also issues send $C_{l-1,p} \rightarrow C_{l-1,p_3}$ and receive $C_{l-1,p} \leftarrow C_{l-1,p_3}$.

Next, we process and drop $C_{l-2,p}$'s work-item from $\mathcal{L}_p^{\text{int}}$ but are not allowed to add any new items to our work-groups. $\mathcal{L}_p^{\text{int}}$ is now the only non-empty work-group and contains cluster $C_{l-1,p}$ only. We compute the local, interior operations for $C_{l-1,p}$ and are allowed to add $C_{l-2,p}$ and $C_{l,p}$ to $\mathcal{N}_p^{\text{cop}}$ and $\mathcal{N}_p^{\text{int}}$. This state is shown in Fig. 5c.

$C_{l-2,p}$ in $\mathcal{N}_p^{\text{int}}$ comes next followed by the work-group $\mathcal{N}_p^{\text{cop}}$ containing $C_{l-2,p}$ and $C_{l,p}$. We process both work-items and add $C_{l-2,p}$ to work-groups $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{L}_p^{\text{int}}$ after dropping $C_{l-2,p}$ from $\mathcal{N}_p^{\text{cop}}$. Next, we process $C_{l-2,p}$ in $\mathcal{L}_p^{\text{int}}$ and $C_{l,p}$ in $\mathcal{N}_p^{\text{int}}$. But only after processing the last remaining work-item, $C_{l-2,p}$ in $\mathcal{L}_p^{\text{cop}}$, we are allowed to add $C_{l-2,p}$ and $C_{l-1,p}$ to $\mathcal{N}_p^{\text{cop}}$ and $\mathcal{N}_p^{\text{int}}$ once again. Further assuming that the receive $C_{l+1,p} \leftarrow C_{l+1,p_1}$ finished, we see the current state in Fig. 5d.

The next processed work-group is $\mathcal{N}_p^{\text{cop}}$. We can drop $C_{l-2,p}$ immediately, because of the missing copy layer. In contrast $C_{l-1,p}$ returns instantly and unsuccessfully from *neighboringCopy* in line 15 of Alg. 4. Reason is the ongoing receive $C_{l-1,p} \leftarrow C_{l-1,p_3}$ bouncing back in line 13 of Alg. 3. Thus our scheduling leaves $C_{l-1,p}$ untouched in $\mathcal{N}_p^{\text{cop}}$ as illustrated in Fig. 5e.

Instead we continue with $\mathcal{N}_p^{\text{int}}$ and finish $C_{l-2,p}$'s neighboring update. Assuming that the receive $C_{l-1,p} \leftarrow C_{l-1,p_3}$ finished in the meantime, we now compute the neighboring contribution for $C_{l-1,p}$'s copy layer and remove $C_{l-1,p}$ from $\mathcal{N}_p^{\text{cop}}$. The neighboring, interior operations of $C_{l-1,p}$ are the only remaining work-item. We finish those, drop $C_{l-1,p}$ from $\mathcal{N}_p^{\text{int}}$ and generate new work-items by adding $C_{l-2,p}$, $C_{l-1,p}$ and $C_{l,p}$ to $\mathcal{L}_p^{\text{cop}}$ and $\mathcal{L}_p^{\text{int}}$. This last state of our example is shown in Fig. 5f.

SUMMARY

This chapter introduced our clustered LTS scheme, which exploits heterogeneous time steps in the computational domain. Here, we started our considerations on the basis of two neighboring elements in Ch. 3.1. By limiting the possible LTS relations to two general cases, we allowed for every element a single time buffer or a single set of time derivatives or both. Next, Ch. 3.2 introduced our clustering and a normalization step. We allowed for arbitrary elements in our clusters, thus did not impose connectivity of a cluster's elements. Ch. 3.3 extended our clustered LTS scheme to multiple partitions. This led to a strict separation of elements in the clusters' interior regions, copy regions and ghost regions. Finally, Ch. 3.4 introduced our time management and Ch. 3.5 our scheduling. Ch. 3.5 closed with an extensive example covering all of the introduced aspects.

The derivation of our clustered LTS scheme covered all algorithmic design decisions of our computational core. However, our final implementation requires an appropriate layout of the involved data structures in memory and parallelization at all levels. For example, we assumed asynchronous communication just to happen in background, but left the actual implementation of the asynchronous data exchange open. For this purpose we switch in the next part, Pt. ii, to an engineering perspective. After the definition of our targeted supercomputers, this leads us to the complete formulation of our computational core, capable of running earthquake simulations at petascale performance. A detailed performance evaluation of all introduced concepts then follows in Pt. iii.

Part II

SUPERCOMPUTING

This part maps the clustered LTS scheme of Pt. i to supercomputing systems.

We start by introducing our targeted machines in Ch. 4. The remaining chapters of this part systematically address all technical layers of the targeted machines.

Ch. 5 studies all data structures of our computational core and derives a memory layout supporting vectorization, shared memory parallelization and distributed memory parallelization.

Next, Ch. 6 introduces the hybrid parallelization of our computational core aiming at machine-size simulations. Discussion and auto-tuning of our innermost kernels in Ch. 7 follows our hybrid parallelization. The innermost kernels drive the single-core performance of our computational core and thus vectorization is covered as last level of parallelism.

4 SYSTEMS

This chapter describes the four supercomputing systems *SuperMUC-1*, *SuperMUC-2*, *Stampede* and *Tianhe-2*. All four systems are used in small- and large-scale simulations throughout the remaining chapters and build the targeted architectures for our computational core.

SuperMUC-1 and SuperMUC-2 are homogeneous supercomputers and the floating point performance is delivered by fat-core Intel Xeon CPUs. In contrast Stampede and Tianhe-2 are accelerated with Intel Xeon Phi coprocessors and thus heterogeneous. Here the largest fraction of the floating point performance is offered by the Intel Xeon Phi cards connected to the host CPUs via PCI-E.

Most top-ranked systems in the June 2015 Top 500 list[1] are either GPU-accelerated, BlueGenes or built of our targeted Intel Xeon processors and Intel Xeon Phi coprocessors. In recent years a shift towards GPU- and Intel Xeon Phi accelerated systems was visible for the highest ranked systems of the Top 500 lists. Considering future systems, Intel's main focus for next-generation Intel Xeon Phis is socketed. Presumably most upcoming supercomputers using the Intel Xeon Phi technology will be homogeneous. Examples are the planned 27+ PFLOPS *Cori Phase-II*[2] of the National Energy Research Scientific Computing Center and the planned 180+ PFLOPS *Aurora*[3] of the Argonne Leadership Computing Facility. From this perspective, the results obtained on the homogeneous SuperMUC-2 have the highest relevance for upcoming socketed Intel Xeon Phi systems.

SUPERMUC-1 The SuperMUC-1 system is located at the Leibniz Supercomputing Center in Munich, Germany. With a sustained HPL-performance of 2.9 PFLOPS and a theoretical peak performance of 3.2 PFLOPS in double-precision it is listed at position 20 of the June 2015 Top500 list.

SuperMUC-1 consists of 9,216 dual-socket Intel Xeon E5-2680 v3 nodes. The nodes are connected with a commodity InfiniBand FDR10 network in a fat-tree topology. The fat-tree is organized in 18 islands with 512 nodes each and a 4:1 bandwidth ratio of intra-island to inter-island communication.

1 https://web.archive.org/web/20150905161941/http://top500.org/list/2015/06/

2 https://web.archive.org/web/20150905132454/http://www.nersc.gov/users/computational-systems/cori/

3 https://web.archive.org/web/20150911154757/http://www.alcf.anl.gov/articles/introducing-aurora

Every socket has a total of 8 cores with a base frequency of 2.7 GHz. The theoretical double-precision peak performance of 346 GFLOPS per node stems from two 256-bit (FMUL, FADD) floating point execution units per core. We refer to a core or node of SuperMUC-1 with the abbreviation *SNB-MUC*.

SUPERMUC-2 SuperMUC-2 is also located at the Leibniz Supercomputing Center. SuperMUC-2 ranks directly behind SuperMUC-1 at position 21 of the June 2015 Top 500 list. It sustained a HPL-performance of 2.8 PFLOPS and has a theoretical peak performance of 3.6 PFLOPS.

SuperMUC-2 contains 3,072 dual-socket Intel Xeon E5-2697 v3 nodes. The nodes are connected via Infiniband FDR14 with a fat-tree topology. Analogue to SuperMUC-1, the machine contains 512 nodes in every of its 6 islands. Again the ratio of intra-island to inter-island bandwidth is 4:1.

Every socket has 14 cores with a AVX base frequency of 2.2 GHz and a marked TDP frequency of 2.6 GHz. The theoretical peak performance in double-precision per socket is 493 GFLOPS at 2.2 GHz. Here a core features two 256-bit floating point execution units capable of performing FMA-operations. We refer to a core or node of SuperMUC-2 as *HSW-MUC*.

STAMPEDE Stampede is located at the Texas Advanced Computing Center in Austin, TX, USA. The June 2015 Top500 list ranks Stampede at position 8 with a sustained HPL-performance of 5.2 PFLOPS. Stampede is build of 6,400 nodes connected with a Mellanox FDR inifinband network in a two-level fat-tree topology. Each node is equipped with two Intel Xeon E5-2680 CPUs and accelerated with an Intel Xeon Phi SE10P coprocessor.

The host CPUs operate at 2.7 GHz and deliver, analogue to SuperMUC-1, 346 GFLOPS per node. An Intel Xeon Phi SE10P coprocessor has 61 cores operating at 1.1 GHz. Each of the coprocessor's cores is able to execute a 512-bit FMA-operation per cycle. Thus the floating point performance of a single coprocessor is 1.1 TFLOPS. This is equivalent to an aggregate node performance of 1.4 TFLOPS and a theoretical, system-wide peak performance of 9.1 PFLOPS.

TIANHE-2 The last system, Tianhe-2, is based at the National Super Computer Center in Guangzhou, China. Tianhe-2 is the top-system of the June 2015 Top500 list with a reported HPL-performance of 33.9 PFLOPS in double-precision. A total of 16,000 nodes are connected with a custom interconnect in a fat-tree topology. Each of the nodes is equipped with two Intel Xeon E5-2692 v2 CPUs and accelerated with three Intel Xeon Phi 31S1P coprocessors.

The two 12-core CPUs per node operate at 2.2 GHz and deliver 422 GFLOPS per node. A single Intel Xeon Phi 31S1P coprocessor has 57 cores and operates at 1.1 GHz. This leads to a floating point performance of 1.0 TFLOP per card and an aggregated, theoretical performance of 54.9 PFLOPS for the entire system.

5 DATA STRUCTURES

This chapter introduces the data structures of our computational core. Ch. 5.1 starts by defining computational matrices and examining the sparsity patterns of our time, volume and surface integrator. Next, Ch. 5.2 encodes the per-element LTS information we require for our clustered local time stepping scheme in a bitmask. Finally Ch. 5.3 introduces the memory layout of our computational core. Here we consider all levels of parallelism by sorting and aligning our data.

5.1 MATRIX STRUCTURES

In this chapter we discuss the final matrix patterns of our computational core stemming from the derivative computation in (27), the local update step (49) and the neighboring update step (50). The dimensions of all matrix operators exclusively depend on the number of quantities or the number of basis functions or both. Our elastic wave equations (14) have a fixed number of nine quantities. Thus we consider the number of basis functions as only variable parameter. Considering the definition of our basis in (11), the number of basis functions $B_{\mathcal{O}}$ in dependency of the convergence rate $\mathcal{O}$ of our ADER-DG scheme is given by:

$$B_{\mathcal{O}} = \frac{\mathcal{O} \cdot (\mathcal{O}+1) \cdot (\mathcal{O}+2)}{6}. \tag{67}$$

COMPUTATIONAL MATRICES For all following chapters we redefine the DOFs, time derivatives and time integrated DOFs as $B_{\mathcal{O}} \times 9$ matrices. This matches SeisSol's view on the matrix operations, eases visualization and is equivalent to transposing (27), (49) and (50). To prepare the formulation of our final computational core, we also introduce a new set of global and element-local matrices. Additionally to transposing our existing matrices we also enhance them. Our new *computational stiffness* and *computational flux* matrices also carry the effect of the (diagonal) inverse mass matrix and our *computational jacobians* and *computational flux solvers* the effect of the scalars J_k and $S_{k,i}$.

We define our new global matrices K^{ξ_c}, $\bar{\mathrm{K}}^{\xi_c}$, $\mathrm{F}^{-,i}$ and $\mathrm{F}^{+,i,j,h}$ as:

$$\begin{aligned}
\mathrm{K}^{\xi_c} &:= M^{-1}(K^{\xi_c})^T \\
\bar{\mathrm{K}}^{\xi_c} &:= M^{-1}K^{\xi_c} & c,h \in 1,2,3 \\
\mathrm{F}^{-,i} &:= M^{-1}(F^{-,i})^T & i,j \in 1,2,3,4. \\
\mathrm{F}^{+,i,j,h} &:= M^{-1}(F^{+,i,j,h})^T
\end{aligned} \tag{68}$$

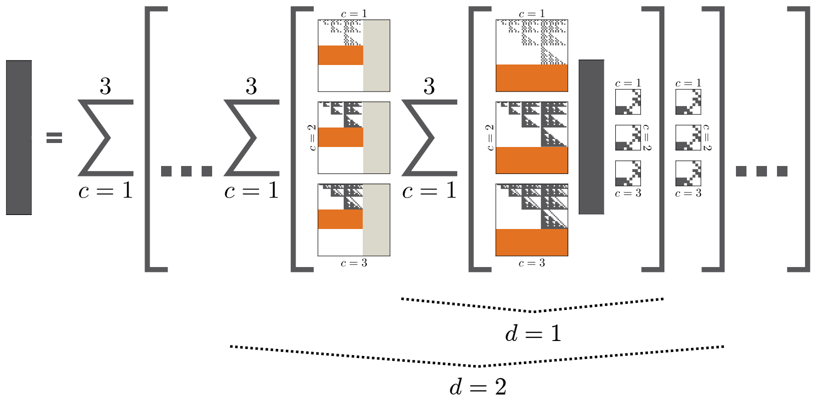


Figure 6: Structure of the recursive time derivative computation via (70) for a sixth order scheme. Non-zero entries are gray, zeros white. Zero-blocks of the stiffness matrices generating zeros in the derivatives are colored in orange. Blocks of the stiffness matrices hitting zero-blocks of the derivatives are colored silver.

Additionally we get for the element-local matrices $\mathrm{A}^{\mp}_{k,i}$ and $\mathrm{A}^{\xi_c}_{k}$:

$$\begin{aligned} \mathrm{A}^{\mp}_{k,i} &:= -\frac{|S_{k,i}|}{|J_k|}(\hat{A}^{\mp}_{k,i})^T \\ \mathrm{A}^{\xi_c}_{k} &:= |J_k|(A^{\xi_c}_{k})^T \end{aligned} \qquad c \in 1,2,3. \tag{69}$$

This leads us to the definition of the recursive time derivative computation (27):

$$\frac{\partial^{d+1}}{\partial t^{d+1}}\mathrm{Q}_k = -\sum_{c=1}^{3}\bar{\mathrm{K}}^{\xi_c}\left(\frac{\partial^d}{\partial t^d}\mathrm{Q}_k\right)\mathrm{A}^{\xi_c}_{k}. \tag{70}$$

We obtain the element-local operations (49) as:

$$\mathrm{Q}^{*,n_k+1}_{k} = \mathrm{Q}^{n_k}_{k} + \sum_{c=1}^{3}\mathrm{K}^{\xi_c}\mathcal{T}_k\mathrm{A}^{\xi_c}_{k} + \sum_{i=1}^{4}F^{-,i}\mathcal{T}_k\mathrm{A}^{-}_{k,i}. \tag{71}$$

And we obtain the neighboring updates (50) as:

$$\mathrm{Q}^{n_k+1}_{k} = \mathrm{Q}^{*,n_k+1}_{k} + \sum_{i=1}^{4}\mathrm{F}^{+,i,j_{k,i},h_{k,i}}\mathcal{T}_{k_i}\mathrm{A}^{+}_{k,i}. \tag{72}$$

SPARSITY PATTERNS Fig. 6 illustrates the recursive computation of the time derivatives (70) for a sixth order scheme ($B_6 = 56$). Non-zero entries are gray, while zero-entries are white. The orange rectangles highlight different zero-blocks in the matrices $\bar{\mathrm{K}}^{\xi_c}$. All matrices have zero entries in rows 36-56. This is a result of the choice of our hierarchical expansion basis in Ch. 2.2. Computation of the first derivative ($d = 1$) and thus evaluation of the innermost sum in Fig. 6 leads to derivatives with non-zero entries in rows 1-35 only.

The next recursive derivative computation, annotated with $d = 2$ in Fig. 6, uses these first derivatives as input. The silver blocks covering columns 36-56 of the matrices hit the previously generated

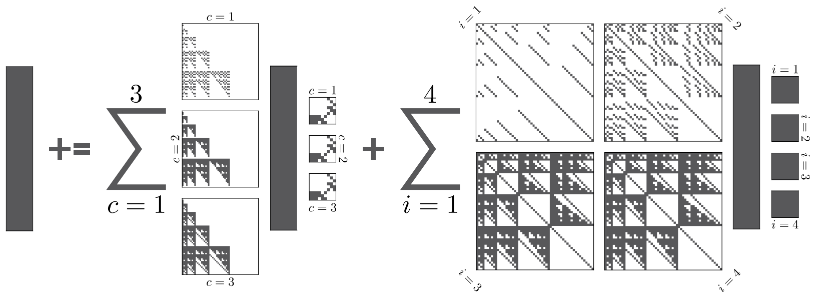


Figure 7: Sparsity patterns of matrix operations in the element-local contribution to a time step (71) for a sixth order scheme. Non-zero entries are gray and zero entries white.

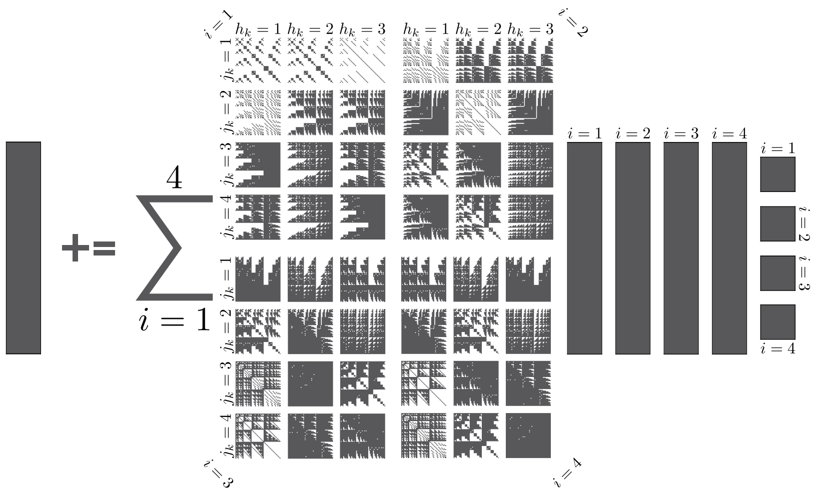


Figure 8: Sparsity patterns of matrix operations in the neighboring elements' contributions to a time step (72) for a sixth order scheme. Non-zero entries are gray and zero entries white.

zero blocks. Thus the orange blocks in rows 21-35 of the second stiffness multiplication generate zero-entries in rows 21-35 of the second derivative.

The scheme continues accordingly and the higher order derivatives shrink in every step analogue to our hierarchical basis. We exploit this property in Ch. 5.3 to reduce the size of the derivatives if stored or communicated and in Ch. 7.1 to reduce the computational effort of the derivative computation.

Fig. 7 illustrates the sparsity patterns of all matrices in the local update step (71) for a sixth order scheme. The first sum corresponds to the volume integration and the second sum to the elements' contribution to the surface integral. In contrast to matrices $A_k^{\xi_c}$ with 24 non-zero entries, matrices $A_{k,i}^{-}$ are dense in general.

The sparsity patterns of the neighboring update step (72) are illustrated in Fig. 8. Compared to the local flux contribution, the choice of the flux matrices in (72) is mesh-dependent. Thus in Fig. 8 we have to choose one out of 12 matrices in every summand. Addition-

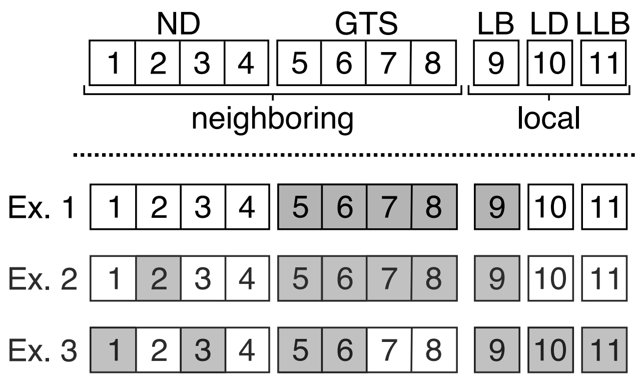


Figure 9: Bitmask used to encode the LTS configuration of a single element. The upper part shows the general layout of the mask. The lower part illustrates three example configurations. Zero-bits in the examples are colored white, while 1's are colored gray.

ally we operate on the four different time integrated DOFs of our face-neighbors and effectively increase the pressure on the memory subsystem compared to the local contributions.

5.2 LTS BITMASK

In Ch. 3.1 we introduced different relations two neighboring elements might have in our clustered local time stepping scheme. This led to the permanent storage of time derivatives or buffers or both per element. Our time integration, volume integration and surface integration introduced in chapters 2.7, 2.8 and 2.9 operate on the time derivatives or buffers or both. In this chapter we define a compact encoding of a single element's LTS configuration.

As illustrated in Fig. 9, we use a 11-bit bitmask for this purpose. In implementation, our bitmask is embedded in the `unsigned short` C++-data-type, which is guaranteed to have at least 16 bits. We use the first four bits of our bitmask (*ND* in Fig. 9) to encode the type of time data our face-neighbors provide for our element's neighboring updates. A neighbor provides time derivatives if the respective bit is 1 and buffers if it is set to 0. Later we use this information to decide if we have to compute time integrated DOFs prior to a neighboring update.

Bits 5-8 (*GTS* in Fig. 9) encode if we are in a global or local time stepping relation with one of our face-neighbors. A neighbor has the same time step as our element and is thus in GTS relation if the respective bit is set to 1.

Bits 9-11 are element-local and describe what information our element stores permanently. In the case of a 1 in bit 9 (*LB* in Fig. 9) our element stores a time buffer. Setting bit 10 (*LD* in Fig. 9) means that our element stores time derivatives. By setting bit 11 (*LLB* in Fig. 9) we are able to encode if the element's time buffer is used to sum multiple time steps.

Fig. 9 also includes three example LTS configurations an element might have. The first example, *Ex. 1*, encodes the classical GTS relation where only bits 5-9 are set. The 0s in bits 1-4 tell us that our element operates on face-neighboring time buffers. Bits 5-8 encode that the element is in GTS relation with all face-neighbors. Because of bit 9 our element permanently stores its time integrated DOFs in a buffer. However bit 11 shows that the buffer is overwritten with new time integrated DOFs in every time step. Finally bit 10 encodes that no derivatives are stored permanently.

Ex. 2, the second example in Fig. 9, is almost identical to *Ex. 1*. The only difference is bit 2, which is set in this case. Therefore we use the second face-neighbor's time derivatives for our neighboring update. This case is special, because bit 6 still tells us that we are in a GTS relation with our second face neighbor. Aside from boundary or dynamic rupture faces, this case occurs when our face-neighbor uses its time buffer to sum up multiple time steps.

Fig. 9's third example, *Ex. 3*, is one of many possible local time stepping configurations. We operate on time derivatives of face-neighbors 1 and 3 and on time buffers of neighbors 2 and 4. Neighbors 1 and 2 are in GTS relation, while 3 and 4 have an LTS relation with our element. Combining the information of bits 3-4 and 7-8 with the layout of our clustering scheme in Ch. 3.2, we know that our second neighbor is part of the next larger time step cluster and our fourth neighbor is part of the next lower time step cluster.

LOCAL TIME INTEGRATION Our neighboring elements' contribution to the surface integral (72) operates on time integrated DOFs. We use our bitmask to process the neighboring time predictions prior to an element's update. If the face neighbors provides time buffers and thus the corresponding *ND*-bit is not set, we can directly proceed with the buffers.

However, if the *ND*-bit is set, we have to derive the time integrated DOFs from the neighbor's derivatives via (29) first. Since Δt in (29) is simply given by the element's time step and $\hat{t}$ by the the time of the element's last complete time step ($C_{l,p}.t^{\text{dofs}}$ in Alg. 2), we seek for the scalar t_0. t_0 is the expansion point and thus the time of the associated neighboring cluster's last complete time step. Due to our normalization step in Ch. 3.2, we have two options for the time step of the neighboring element's cluster. The first option is global time stepping and the corresponding *GTS*-bit is set in Fig. 9. In this case $t_0 = \hat{t}$ and both scalars simply cancel out in (29). The second option is true, if the corresponding *GTS*-bit is not set in Fig. 9. Here, we use the last full update time of the next cluster for $\hat{t}$ in (29).

5.3 MEMORY LAYOUT

This chapter discusses the mapping of our ADER-DG scheme to memory. Goal is a suitable memory layout for high-order discretizations. In our considerations a *suitable* layout minimizes the final time-to-solution of our simulations. It is important to note that this is not necessarily the most compact, latency-optimized, throughput-optimized or most cache-friendly storage format. Instead we aim at a layout, which maximizes the performance of our entire computational core. Our design includes memory alignment, considers prefetching and is the backbone of our shared and distributed memory parallelization.

COLUMN-MAJOR STORAGE Starting at convergence rate 3 our DG discretization has more basis functions than elastic quantities (10 vs. 9). Thus vectorization in the basis functions is beneficial and we use a column-major storage for all our matrix structures in (70), (71) and (72). As visualized in Fig. 6, Fig. 7 and Fig. 8 our column-major storage is linear in the basis functions of our DOFs Q_k, derivatives $\partial^d / \partial t^d Q_k$ and time integrated DOFs $\mathcal{T}_k$. This supports vectorization of the dominating matrix-matrix products.

MEMORY ALIGNMENT For optimal performance we align large parts of our data to certain boundaries in the memory hierarchy. The alignment of entire memory blocks is simple. For example the `posix_memalign` or `mm__malloc` functions implement memory alignment for the heap. Similar for the stack we use the attribute `aligned` to ensure aligned allocations.

However, alignment of the memory blocks only ensures that the respective base pointers are aligned to proper boundaries. Our ADER-DG scheme operates within memory blocks, for example during a mesh iteration, and performs vector instructions on the respective data. Load and store operations access cache lines having a size of 64 bytes on all considered architectures [36, 38]. Vector loads and stores of data spanning across cache line boundaries impose performance penalties. For example every second 256 bit (32 byte) operation on unaligned memory is across a cache line split.

Accordingly inside a memory block, accessed via vector loads or stores, our memory layout supports optimal alignment with respect to the used vector instruction set. This translates to 16-byte alignment for SSE3, 32-byte alignment for AVX and AVX2, and 64-byte for the Knights Corner and AVX-512 instruction sets.

We perform vector operations in matrix-columns having the number of basis functions as size. Consequently our memory layout ensures proper alignment of these columns by zero-padding. This means that we introduce artificial rows, if the size of the columns in bytes is not already a multiple of the desired vector length.

COMPRESSED DERIVATIVES Ch. 5.1 discussed the structure of the time kernel in terms of increasing-size zero-blocks in higher derivatives. We exploit this to reduce memory requirements of derivatives. Instead of storing the full $B_\mathcal{O} \times 9 \times \mathcal{O}$-block, we use a compressed storage format. For the d^{th} derivative we only store the corresponding non-zero block of size $B_{\mathcal{O}-d} \times 9$. This results in a total requirement of $D_\mathcal{O}$ non-zeros:

$$D_\mathcal{O} = \sum_{d=0}^{\mathcal{O}-1} B_{\mathcal{O}-d} \cdot 9. \tag{73}$$

We then realize vector alignment by using zero-padding for the individual non-zero blocks.

GLOBAL MATRICES Ch. 2.3 defined the mass matrix, stiffness matrices and flux matrices in terms of a unique reference tetrahedron. Therefore all computations of our ADER-DG scheme share the same *global matrices* in a mesh iteration. Following (68) we have to store the three matrices $\bar{\mathrm{K}}^{\xi_c}$, the three matrices K^{ξ_c} and the 52 matrices $\mathrm{F}^{-,i}$ and $\mathrm{F}^{+,i,j,h}$.

Ch. 7.2 introduces our matrix kernels, the innermost and most performance critical kernels. These matrix kernels exclusively operate on our global data. Depending on the number and distribution of non-zero entries in the matrices we use either dense or sparse matrix-matrix multiplications. For this purpose, we vector-align the first entry of all matrices, but only introduce zero-padding for columns of matrices stored as dense. For sparse matrices we linearly store the non-zero entries (column-major).

To avoid jumps in memory, we allocate a single, memory page-size aligned memory block for all global matrices. Our ordering of the matrices inside our memory block supports the stride detection of the hardware prefetcher (e.g. [36, 51]) by linearly following our order of execution.

The first element-local operation is the computation of time derivatives (70). Thus we store the three matrices $\bar{\mathrm{K}}^{\xi_c}$ first. Following the discussion of matrix structures in Ch. 5.1, we can exploit the zero-blocks of matrices $\bar{\mathrm{K}}^{\xi_c}$ for dense storage. In this case we only store the vector-aligned non-zero block of size $B_{\mathcal{O}-1} \times B_\mathcal{O}$ for a convergence rate $\mathcal{O}$ scheme.

Next we compute the volume integration in (71). Consequently we store the three matrices K^{ξ_c} next. Again, following Ch. 5.1, we reduce the dense size and store the vector-aligned non-zero block of size $B_\mathcal{O} \times B_{\mathcal{O}-1}$ only.

The final part of the element-local step (71) is the element's local contribution to the surface integration. Therefore we store the four matrices $\mathrm{F}^{-,i}$ next.

The remainder of the memory block is occupied by the matrices $\mathrm{F}^{i,j,h}$ required for the neighboring contribution (72). Due to the used

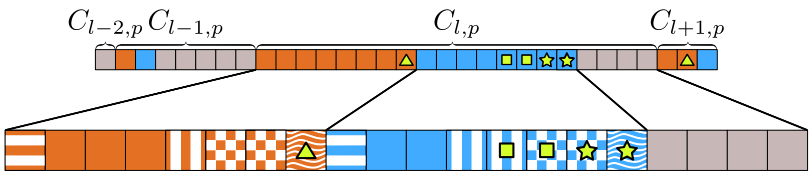


Figure 10: Memory-aware sorting of elements in the two-dimensional example shown in Fig. 3. The upper part shows the sorting of the entire partition with respect to clusters, the clusters' ghost layers, copy layers and interior regions. The lower part also shows the sorting by communication regions of $C_{l,p}$'s ghost and copy layer.

unstructured meshes we are not able to derive a fixed linear ordering in this case and simply store the matrices $F^{+,i,j,h}$ in ascending order.

ELEMENT-LOCAL DATA We call all data, depending on the tetrahedrons, *element-local*. Our memory layout follows a common strategy for all element-local memory blocks. The idea of our strategy is to store element-local data consistent with the operations of our clustered LTS scheme introduced in Ch. 3. This ensures consecutive access of all non-neighboring information in our computational loops over elements. As for our global matrices this technique simplifies predictions of memory accesses and supports hardware prefetching.

We ensure linear memory access of non-neighboring data in our work-items by sorting our elements. On the outermost level we sort the elements by their local time stepping clusters. Next we sort the individual elements of a cluster by the communication-related structures. First comes the ghost layer, then the copy layer and finally the interior of every cluster. Further, we sort the elements in the ghost and copy layers by their individual communication regions. If a tetrahedron is a member of more than a single ghost or copy region (see Ch. 3.3), we duplicate it in every region. This duplication of elements allows us to store all communication-related data of all copy and ghost regions linearly in memory. Thus our sends and receives are able to operate on this data directly. Otherwise we would have to implement an additional step collecting the data in send and receive buffers before issuing our communication.

Fig. 10 shows the sorting applied to our simple two-dimensional example in Fig. 3 of Ch. 3.3. In the upper part we see that the element-local data is sorted with respect to the individual LTS clusters in the partition. Here inside every cluster the ghost layer data comes first, followed by the copy layer and interior. Additionally the two elements, which are part of multiple copy regions, are duplicated and marked with a yellow star and square. The element, which is part of ghost region of $C_{l-1,p}$ and $C_{l,p}$, is also duplicated and marked with a yellow triangle.

The lower part of Fig. 10 shows the logical structure of $C_{l,p}$'s element-local data. Additionally following Fig. 3b the individual communication regions are visualized with different patterns and $C_{l,p}$'s copy and ghost layer in the partition are sorted accordingly.

CONSTANT MATRICES As *constant matrices* we consider $A_k^{\xi_c}$, $A_{k,i}^-$ and $A_{k,i}^+$ in (69). For a single element the entries of these matrices depend on the material properties and the orientation and shape of the tetrahedron. The material parameters and spatial discretization are fixed for an entire run, thus all matrix entries are constant. Additionally we only access time buffer or time derivative data of ghost layer elements, therefore we store constant matrices only for copy and interior elements.

We use two memory blocks to store our constant matrices. The first block contains matrices $A_k^{\xi_c}$ and $A_{k,i}^-$ used in the derivative computation (70) and element-local update (71). As discussed we follow our ordering of element-local data to support hardware prefetching. Additionally for every element k we store the three matrices $A_k^{\xi_c}$ first and then the four matrices $A_{k,i}^-$. In sum this respects, in harmony with the storage of the global matrices, our order of execution. The second memory block contains the element-local matrices $A_{k,i}^+$ used for the neighboring updates in (72). By following our sorting of element-local data, we ensure linear access of matrices $A_{k,i}^+$ in the neighboring update work-items.

We support either storing the non-zeros only or the dense 9×9 block of matrices $A_k^{\xi_c}$. Matrices $A_{k,i}^-$ and $A_{k,i}^+$ are fully occupied in general and we store them as dense. No vector alignment is required for any of the constant matrices.

INTERNAL STATE The *internal state* covers the DOFs, time buffers and time derivatives. Together with the constant matrices, the internal state is responsible for almost all of SeisSol's memory consumption.

However, in contrast to the constant data, the memory consumption of the internal state depends on the order $\mathcal{O}$ of the ADER-DG method. As a consequence the dimensions of all matrix-columns in the internal state depend on the number of basis functions. Therefore we use zero-padding in the columns of all of the internal state's matrices to impose an alignment suitable for the respective vector instruction set.

We allocate a single page-size aligned memory block for our DOFs. Analogue to the constant data, our memory block only covers elements in the copy layers and interior regions. As before we ensure linear access of the DOFs in our clusters' work-items by imposing our ordering of element-local data.

Our storage scheme for the time data is the most complex. Here we allocate three page-size aligned memory blocks, one for the copy layers, one for the ghost layers and one block for the interior regions.

After applying our sorting, the individual memory chunks of the copy regions in the copy layers belong to a local cluster $C_{l,p}$ and have a unique associated LTS id of the neighborings partition's cluster. Additionally, we logically duplicate elements, which are part of more than one copy region. Dependent on their respective LTS configurations, elements in the clusters' copy layers store vector-aligned time buffers or vector-aligned compressed derivatives or both. Inside every copy region we first store all buffers and second all derivatives. Copy regions, which are not in global time stepping relation with their neighboring partition, send either buffers or derivatives only, never both. Thus, in this case, all communication relevant data is stored linearly in memory. Copy regions having a global time stepping relation with their neighboring partition are a special case. Following Ch. 5.2 the neighboring partition of a copy region expects us to send, dependent on the LTS configurations of our elements, time buffers and time derivatives within the same copy region. Therefore we apply an additional sorting step and store copy elements providing derivatives to the neighboring partition in front of those providing buffers. By doing so we store all communication related data linearly in memory again. Here, the corresponding communication-related memory chunk begins after the buffers of the elements providing time derivatives and reaches into the memory of the time derivatives until all respective elements providing derivatives are covered. Our layout of the ghost regions mirrors the communication related data of the corresponding copy regions. Hence we perform the same splitting of time buffers and derivatives together with our sorting. In sum, our memory layout allows us to send copy region data and receive ghost region data *as is*.

The time data of the interior elements covers our third page-aligned memory block. Once again we apply our element-local sorting and store in every region the buffers first, followed by the derivatives.

MEMORY CONSUMPTION Tab. 1 shows the memory consumption of a single LTS element for convergence orders 2-8 in double precision (8 bytes per value).

The second column shows the amount of memory in bytes an element's DOFs or time integrated DOFs occupy. Inside the parentheses we also see the overhead in percent our 16-byte, 32-byte or 64-byte vector-alignment adds to this. For example, for $\mathcal{O} = 5$ we introduce a 3% overhead for 16-byte and 32-byte alignment and a 14% overhead for 64-byte alignment. This correlates to the 35 basis functions or rows of a fifth order scheme, which are zero-padded to 36 rows for 16-byte and 32-byte alignment and to 40 rows for 64-byte alignment. In general certain convergence rates naturally feature alignment. For example the 56 basis functions of $\mathcal{O} = 6$ are a multiple of 8 (64 bytes

$\mathcal{O}$	$Q_k \vee \int Q_k \, dt$	$\frac{\partial^d}{\partial t^d} Q_k$	$\frac{\partial^d}{\partial t^d} Q_k$, compressed	internal%
2	288 (0, 0, 100)	576	360 (20, 60, 220)	9, 10, 14
3	720 (0, 20, 60)	2,160	1,080 (7, 33, 113)	20, 24, 30
4	1,440 (0, 0, 20)	5,760	2,520 (3, 14, 60)	33, 41, 48
5	2,520 (3, 3, 14)	12,600	5,040 (3, 9, 37)	47, 57, 64
6	4,032 (0, 0, 0)	24,192	9,072 (2, 5, 21)	58, 69, 75
7	6,048 (0, 0, 5)	42,336	15,120 (1, 3, 14)	68, 79, 83
8	8,640 (0, 0, 0)	69,120	23,760 (1, 2, 9)	75, 85, 88

Table 1: Absolute and relative memory consumption in dependency of the order of convergence using double-precision. The first column shows the considered order of convergence. Columns two, three and four show the memory requirements of (time integrated) DOFs, derivatives and compressed derivatives in bytes. Values in parenthesis represent the additional overhead in percent for 16-byte, 32-byte and 64-byte vector alignment. The fourth column compares the non-zero memory requirements of the internal state to the constant data. Given are the relative requirements in percent for the DOFs plus buffers (first number), DOFs plus compressed derivatives (second number) and DOFs plus buffers plus derivatives (third number).

in double precision). Thus, in this case, it is sufficient to align the base pointers only

The third column of Tab. 1 contains the memory consumption of the derivatives in bytes. Compared to the number of bytes required for our compressed storage in the fourth column, we see the benefit of our storage scheme. For example, considering non-zeros only, we reduce the memory consumption by 2.7× for a sixth order scheme. The parentheses in the fourth column again show the overhead our alignment adds to the compressed storage. Especially for 64-byte alignment this overhead is more severe than the alignment-overhead for the (time integrated) DOFs shown the second column. The reason is the alignment of the last derivatives containing only 1, 4 and 10 non-zero rows. For 64-byte vector-alignment these are padded to 8 and 16 rows respectively, adding an overhead of 113%. In future implementations it might be beneficial to introduce a mixed format and use unaligned storage for derivatives having a small numbers of rows.

The fifth and last column of Tab. 1 compares the memory consumption of the internal state's non-zeros to those of the constant matrices. Here we assume sparse storage for the three constant matrices $A_k^{\xi_c}$. This results in a total memory consumption of $3 \cdot 24 + 8 \cdot 9 \cdot 9$ entries for matrices $A_k^{\xi_c}$ and $A_{k,i}^{\mp}$ and thus a total of 5760 bytes in double precision. The first number in the fifth column of Tab. 1 shows the relative memory consumption of the internal state if an element stores DOFs and time buffers only. Additionally the second number shows the relative consumption for storage of DOFs and compressed derivatives, while the third number shows the consumption for storage of DOFs, buffers and compressed derivatives.

In summary the fifth column's numbers show the benefit of using high orders from a memory consumption's viewpoint. Low-order simulations use only a small fraction of the memory to encode the actual solution. The break-even point of constant to non-constant data depends on the LTS configurations, vectors alignment and partitioning, but is around orders $\mathcal{O} = 4$ to $\mathcal{O} = 5$ (see Tab. 1).

SUMMARY

This chapter introduced the data structures of our computational core together with an appropriate memory layout. First, Ch. 5.1 discussed the used matrix structures and the corresponding sparsity patterns. In the following memory layout, we exploited especially the recursively generated zero-blocks in the time derivatives. Next, Ch. 5.2 encoded the LTS information of every element in a compact bitmask. Ch. 5.3 discussed our final memory layout. The layout features arbitrary vector-alignment, supports hardware prefetching, and duplicates elements in the copy layer for in-place communication. A discussion of our computational core's memory requirements concluded Ch. 5.3.

The next chapters of this part, Pt. ii, use our computational core's memory layout to address the different concurrency levels of our targeted supercomputers. First, we introduce our hybrid, asynchronous single- and multi-node parallelization in Ch. 6. This is followed by the introduction of our low-level ADER-DG kernels in Ch. 7, targeting vectorization within a single core. Finally, Pt. iii evaluates the small- and large-scale performance of our complete computational core in different setups.

6 SHARED AND DISTRIBUTED MEMORY

This chapter introduces the shared and distributed memory parallelization of our computational core. The dominating programming model for our targeted supercomputers (see Ch. 4) is MPI+OpenMP and used for parallelization in this chapter.

Ch. 6.1 introduces the shared memory parallelization and Ch. 6.2 the asynchronous distributed memory parallelization. Afterwards, the last remaining building block for our computational core is the single-core optimization and covered in Ch. 7.

6.1 OPENMP

Due to the structure of our clustered local time stepping scheme, we are able to use a simple but efficient approach for the shared memory parallelization of our computational load. We recapitulate that our work-items of Ch. 3.4 either perform local operations or neighboring operations on all elements covered by the work-item. Further our memory layout of Ch. 5.3 ensures a linear ordering of all elements belonging to a specific work-item.

In sum this translates to two simple computational `for`-loops, one for the local operations and one for the neighboring operations, for implementation of the functions calls *localCopyOps()*, *neighCopyOps()*, *localIntOps()* and *neighIntOps()* in lines 6, 15, 20 and 24 of Alg. 3. Both loops iterate from a first to a last element, which are simply the first and last element of the respective work-item.

We use the OpenMP API[1] for our shared memory parallelization. Here, the simple pragma `#pragma omp parallel for schedule(static)` in front of our two computational loops is sufficient for parallelization. Since the per-element load in a work-item is almost constant, static scheduling is sufficient and efficient.

In addition to the allocation of our data in Ch. 5.3, we add an initialization step of all element-local data directly after its allocation. The allocations get mapped to memory in hardware when touched for the first time. OpenMP follows a first-touch policy, meaning that a memory page is allocated in the memory of the Non-Uniform Memory Access (NUMA) node on which the thread touching the page for the first time resides on. We mirror the OpenMP parallelization of our two computational loops to the initialization and simply set all floating values to zero initially. By doing so we ensure that all element-local data is mapped to memory close to the cores using it

1 http://openmp.org/

the most and reduce the NUMA-effects for compute nodes sensitive to NUMA.

6.2 MESSAGE PASSING INTERFACE

Ch. 3.5 introduced our asynchronous communication scheme from a high-level perspective and simply assumed that communication happens in background, once high-level send and receive requests are issued. Next, in Ch. 5.3, we discussed the final memory layout and how the communication data is embedded via ghost and copy regions. In the internal state each communication region has an associate memory chunk which contains the communication data. Thus for all of our partitions we know exactly the memory addresses and size of the data, which should be sent to neighboring partitions and should be received from neighboring partitions.

In this chapter we discuss the usage of the Message Passing Interface (MPI) to implement our asynchronous communication scheme. For this purpose we use a dedicated communication thread, which manages all communication internally. Our communication thread's usage of the MPI-standard is limited to the three functions `MPI_Isend`, `MPI_Irecv` and `MPI_test`.

`MPI_Isend` begins a non-blocking send, `MPI_Irecv` begins a non-blocking receive and `MPI_test` tests for completion of a communication request. The motivation for our dedicated communication thread is the implementation of the non-blocking communication in most commonly used MPI-implementations. A call to `MPI_Isend` and `MPI_Irecv` returns immediately after the start of the communication and we are able to continue with our computations. However, most MPI-implementations do not progress the messages in background. In practice our communication only continues when we call the MPI-implementation again, for example via `MPI_test`. Thus, aside from managing the communication and issuing sends and receives, the main task of our communication thread is to ensure background progression by constantly calling `MPI_test` for pending communication requests. This type of MPI-progression is computationally intensive and the reason why we pin our communication thread to a dedicated core. On many-core architectures this is a cheap price to pay for the benefits of truly asynchronous communication. In future implementations it might be beneficial to consider replacing our send-receive-model by a Remote Memory Access (RMA) approach to benefit from a more direct communication.

COMMUNICATION THREAD Our communication thread has to provide several arguments to the MPI-functions `MPI_Isend` and `MPI_Irecv`. The first three input arguments are the initial address of the message buffer, the number of elements to send and the type of the elements.

For all of our ghost and copy regions all three arguments are already properly defined, since we included the requirements of MPI already in our memory layout in Ch. 5.3. In terms of outgoing messages issued via `MPI_Isend` the initial addresses correspond to the start of the communication-related data in our copy regions' time data. The size is defined by the number of vector-aligned values in the time buffers or time derivatives or both. The type is the respective MPI datatype for single- or double precision. Analogue for incoming messages issued with `MPI_Irecv` the initial addresses are the starts of the ghost regions' time data, the number of elements equivalent to the number of values and the datatype equivalent to the respective MPI datatype of single- or double-precision.

Since we use `MPI_COMM_WORLD` as communicator and pass it via the respective input argument, the remaining two input arguments uniquely define the receiving LTS cluster C_{l_r,p_r} for `MPI_Isend` and the sending LTS cluster C_{l_s,p_s} for `MPI_Irecv`. The first of the two remaining input arguments is simply the destination p_r or source p_s of the data. We further distinguish multiple messages between two partitions by their MPI tag, which is the last open input argument. Thus, we uniquely define our tag in `MPI_Isend` and `MPI_Irecv` as $l_s \cdot L + l_r + O_{\text{time}}$, where L is the total number of global LTS clusters in (65) and $O_{\text{time}} \in \mathbb{N}$ a constant offset for all communication of time data.

The only output argument of `MPI_Isend` and `MPI_Irecv` is the communication request, which we use to query the status of the communication. Our communication thread defines a send-queue and a receive-queue for every LTS cluster in a partition. Whenever a new communication request is returned by either `MPI_Isend` and `MPI_Irecv`, we add the respective communication request to the corresponding queue.

Our communication thread runs permanently in background, thus we require a way to asynchronously interact with the thread. Relevant are the four calls in lines 2, 5, 7 and 12 of our cluster operations in Alg. 3. We require the communication thread to provide information about the current status of the communication. Here, for every cluster, we have to know if all sends are finished (line 2) and if all receives are finished (line 12). Further we need a way to tell the communication thread that a new set of non-blocking sends (line 7) or a new set of non-blocking receives (line 5) should be issued for a specific cluster. For this purpose we define two volatile signaling variables for every cluster, one for the sends and one for the receives. Every signaling variable has three possible states *complete*, *initiate* and *ongoing*.

Our communication thread now continuously iterates over the signaling variables of all clusters. Whenever the state of a send-variable is set to *initiate*, the communication threads calls `MPI_Isend` for the cluster's copy regions communicating in the time step, adds all send requests to the cluster's send-queue and changes the state to *ongoing*.

Similar if the state of a receive-variable is set to *initiate*, our communication thread calls `MPI_Irecv` for the clusters ghost regions communicating in the time step, adds the request to the receive-queue and switches the state to *ongoing*.

If our communication thread encounters a signaling variable with the state *ongoing*, it iterates over the respective send- or receive-queue and calls `MPI_test` for the queue's requests. `MPI_test` tests for completion of the respective send or receive and returns `true` if the communication is finished. We remove all requests of finished communication from the queue. A send-queue is empty if the cluster finished sending all its copy layer data and a receive-queue empty if the cluster finished receiving all its ghost region data. In these cases our communication thread changes the state of the signaling variable from *ongoing* to *complete*.

From the perspective of our cluster operations, our function *sendsFinished()* in line 2 of Alg. 3 returns `true` only if the cluster's signaling variable for the sends is in state *complete*. Comparable, *receivesFinished()* in line 12 returns `true` only if the signaling variable for the receives is in state *complete*. To tell our communication thread that the cluster's ghost or copy layer is ready for new communication, we switch the respective signaling variable from *complete* to *initiate* when calling *receiveGhostRegions()* in line 5 or when calling *sendCopyRegions()* in line 7 of Alg. 3.

7 COMPUTATIONAL KERNELS

This chapter discusses the layout of the computational kernels covering almost all of the computational load in our simulations. Our kernels are completely element-local and access face-neighboring data in a read-only fashion. Consequently all of the following considerations refer to a single element only. The only level of parallelization considered at this low-level is the SIMD-paradigm of Intel Xeon CPUs and the Intel Xeon Phi coprocessor.

First, Ch. 7.1 introduces the low-level implementation of our ADER-DG kernels implementing the time, volume and surface integration. Here, we additionally study memory accesses in hardware and discuss the usage of streaming stores for the permanent storage of derivatives. Next, Ch. 7.2 discusses the parameters and generation of our innermost sparse and dense matrix kernels. The matrix kernels dominate the floating point performance of our computational core and are the place where we perform the SIMD-parallelization. Ch. 7.3 studies the arithmetic intensity of our computational core from a theoretical standpoint. Last, Ch. 7.4 discusses an auto-tuning approach for the choice between sparse or dense matrix multiplications for the matrix kernels.

7.1 ADER-DG KERNELS

CODING CONVENTIONS This chapter uses C++ code-snippets to discuss the low-level implementation of our ADER-DG kernels. All of the following code-snippets follow a set of coding conventions. For example, Lst. 1 utilizes many of the most important conventions.

Functions parameters are prepended by `i_` if the parameter is accessed read-only by the function, prepended by `o_` if the parameter is accessed write-only and by `io_` if the functions reads and writes the parameter. For example the call-by-value parameter `i_timeStep` in line 1 of Lst. 1 is accessed in a read-only way. Inside function bodies we prepend `l_` for local variables and `m_` for member variables of the surrounding C++ class.

We use the typedef-name `real` as alias for floating-point values in our computational scheme. This allows us to switch between the native C++ types `float` and `double` at compile time. `float` translates to single-precision (32 bytes per value), while `double` is double-precision (64 bytes per value). All variables related to our time-management, such as `i_timeStep` in line 1 of Lst. 1, are not performance-relevant. Therefore we fix these to double-precision for improved accuracy.

Preprocessor variables are named in uppercase. Examples in Lst. 1 are `CONVERGENCE_ORDER` or `NUMBER_OF_ALIGNED_DOFS`. Before invoking the compiler, we use the preprocessor to replace all occurrences of these variables with values matching our requirements.

TIME KERNEL The time prediction scheme introduced in Ch. 2.7 is our first ADER-DG kernel. Lines 1-6 of Lst. 1 contain the function parameters of our kernel.

`i_timeStep` is the time step Δt used to compute the time integrated DOFs via (29). `computeAder` in Lst. 1 computes time integrated DOFs of the tetrahedron's complete time step in any case, here $\hat{t} = t_0$ holds and the respective scalars cancel out.

`i_stiffnessMatrices` are the three locations in the memory hierarchy, which contain the values of our matrices $\bar{K}^{\xi_1}$, $\bar{K}^{\xi_2}$ and $\bar{K}^{\xi_3}$. The DOFs $Q_k(t_0)$ of the tetrahedron are passed via `i_degreesOfFreedom` and the matrices $A_k^{\xi_1}$, $A_k^{\xi_2}$ and $A_k^{\xi_3}$ via `i_starMatrices`. `o_timeIntegrated` is the location to which our kernel writes the time integrated DOFs.

Analogue, `o_timeDerivatives` might point to the location where we store the derivatives. Recapitulating the possible LTS relations discussed in Ch. 3.1, we only have to store derivatives permanently if a tetrahedron k is required to provide time derivatives $\mathcal{D}_k$ for its face-neighbors. Our `computeAder` function in Lst. 1 assumes that we pass the null-pointer constant `NULL` if no derivatives have to be stored.

Considering the data-heavy structures in the kernel's arguments, the global matrices $\bar{K}^{\xi_c}$ most likely are in a low cache-level, since the time kernel is called over and over for a large number of elements. The DOFs in `i_degreesOfFreedom` and matrices $A_k^{\xi_c}$ in `i_starMatrices` are usually touched for the first time. However our sorting step of element-local data in Ch. 5.3 ensures linear ordering and thus supports hardware prefetching. The computational load in the local ADER-DG kernels is high for high-order simulations and, in this case, the memory-accesses are most likely shadowed behind computations. The pointer `o_timeIntegrated` points either to a temporary data structure, overwritten in every call of the time kernel, or points to memory for global time stepping relations. The derivatives `o_timeDerivatives` are accessed write-only.

In line 8 of Lst. 1 we initialize the scalar of the zeroth derivative ($d = 0$) summand in (29). Lines 11-14 define two arrays, we use to store intermediate results of our time kernel. Both arrays are aligned to page size boundaries. In lines 17-21 of Lst. 1 we initialize the zeroth derivative with the DOFs Q_k and the intermediate time integrated DOFs with $\Delta t \cdot Q_k$. All higher derivatives are reset to zero in lines 23-26.

The function call `streamstoreFirstDerivative` in lines 30-31 wraps a set of low-level intrinsic functions performing streaming stores for the zeroth derivatives if requested (`o_timeDerivatives != NULL`). Since our

Listing 1: Low-level time kernel computing time integrated DOFs and time derivatives (if requested).

```
void computeAder( double i_timeStep,
                  real** i_stiffnessMatrices,
                  real*  i_degreesOfFreedom,
                  real   i_starMatrices[3][STAR_NNZ],
                  real*  o_timeIntegrated,
                  real*  o_timeDerivatives ) {
  // scalars in the taylor-series expansion
  real l_scalar = i_timeStep;

  // temporary result
  real l_temporaryResult[NUMBER_OF_ALIGNED_DOFS]
    __attribute__((aligned(PAGESIZE_STACK)));
  real l_derivativesBuffer[NUMBER_OF_ALIGNED_DERS]
    __attribute__((aligned(PAGESIZE_STACK)));

  // initialize time integrated DOFs and derivatives
  for( unsigned int l_dof = 0;
       l_dof < NUMBER_OF_ALIGNED_DOFS; l_dof++ ) {
    l_derivativesBuffer[l_dof] = i_degreesOfFreedom[l_dof];
    o_timeIntegrated[l_dof]  = i_degreesOfFreedom[l_dof] * l_scalar;
  }

  for( unsigned int l_dof = NUMBER_OF_ALIGNED_DOFS;
       l_dof < NUMBER_OF_ALIGNED_DERS; l_dof++ ) {
    l_derivativesBuffer[l_dof] = 0.0;
  }

  // stream out first derivative (order 0)
  if ( o_timeDerivatives != NULL ) {
    streamstoreFirstDerivative( i_degreesOfFreedom,
                                o_timeDerivatives );
  }

  // compute derivatives and contributions to time integrated DOFs
  for( unsigned l_derivative = 1;
       l_derivative < CONVERGENCE_ORDER; l_derivative++ ) {
    // iterate over dimensions
    for( unsigned int l_c = 0; l_c < 3; l_c++ ) {
      // compute $K_{\xi_c}.Q_k$ and $(K_{\xi_c}.Q_k).A*$
      m_matrixKernels[ (l_derivative-1)*4 + l_c ] (
        i_stiffnessMatrices[l_c],
        l_derivativesBuffer+m_derivativesOffsets[l_derivative-1],
        l_temporaryResult,
        NULL, NULL, NULL );

      m_matrixKernels[ (l_derivative-1)*4 + 3   ] (
        l_temporaryResult,
        i_starMatrices[l_c],
        l_derivativesBuffer+m_derivativesOffsets[l_derivative],
        NULL, NULL, NULL );
    }

    // update scalar for this derivative
    l_scalar *= i_timeStep / real(l_derivative+1);

    // update time integrated DOFs
    integrateInTime( l_derivativesBuffer,
                     l_scalar,
                     l_derivative,
                     o_timeIntegrated,
                     o_timeDerivatives );
  }
}
```

time kernel permanently stores the derivatives in a write-only fashion, we bypass the cache and directly write to memory. This avoids Read for Ownership (RFO), where the entire cache-line is read from memory before we would overwrite it completely [36, 38].

The loop in lines 35-62 of Lst. 1 performs in an iteration a single step of the recursive derivative computation (70). Additionally, following (29), we add the effect of the current iteration's derivatives directly to the time integrated DOFs `o_timeIntegrated`. The inner-loop in lines 38-51 iterates over dimensions ξ_1, ξ_2 and ξ_3. In every of the inner iterations we perform two matrix-matrix multiplications by calling function-pointers of `m_matrixKernels`. The first call multiplies the previous derivative with the matrices $\bar{K}^{\xi_c}$ from the left and stores the intermediate results to `l_temporaryResult`. The jump in memory for the derivatives on the stack via `m_derivativesOffsets` is equivalent to our vector-aligned storage scheme for compressed derivatives (see Ch. 5.3). The second call multiplies the intermediate results with the matrices $A_k^{\xi_c}$ from the right and adds the result to the respective derivative in `l_derivativesBuffer`.

The array of function pointers `m_matrixKernels` allows us to use optimized matrix-multiplication kernels for the individual operations. Since the array-index for the multiplication from the left with $\bar{K}^{\xi_c}$ depends on both loop-indices, `l_derivative` and `l_c`, we are able to hardwire the current level of recursion and the current dimension in the respective matrix kernel. For the second call, we only hardwire the level of recursion in our matrix kernels, since we assume identically shaped matrices $A_k^{\xi_c}$ (see Ch. 5.1 and Fig. 6). We choose the individual matrix-multiplication kernels via auto-tuning based on the used architecture, order of convergence and precision. Details are covered in the next chapters following our current discussion of the ADER-DG kernels.

The remaining lines 53-61 of Lst. 1 add the current derivative's contribution to the time integrated DOFs `o_timeIntergrated` and permanently store the derivative, if requested. Again we reduce the pressure on the memory subsystem by using streaming stores in the function `integrateInTime` for storing the derivative.

VOLUME KERNEL The first sum in the local update step (71) computes the contribution of the volume integration. Lst. 2 shows the volume kernel `computeVolume` performing this operation. `computeVolume` 's first function parameter is the array `i_stiffnessMatrices` containing the locations of the three matrices K_c^{ξ} in the memory hierarchy. The second parameter `i_timeIntegrated` points to the time integrated DOFs of the element. Matrices $A_k^{\xi_c}$ are passed via `i_starMatrices` and the element's DOFs Q_k via `io_dofs`.

It is very likely that all input data already resides in a low cache-level, since we call the volume kernel directly after our time ker-

Listing 2: Low-level volume kernel computing the contribution of the volume integration to a time step.

```
 1 void computeVolume( real** i_stiffnessMatrices,
 2                     real*  i_timeIntegrated,
 3                     real   i_starMatrices[3][STAR_NNZ],
 4                     real*  io_dofs ) {
 5   // temporary result
 6   real l_temporaryResult[NUMBER_OF_ALIGNED_DOFS]
 7     __attribute__((aligned(PAGESIZE_STACK)));
 8
 9   // iterate over dimensions
10   for( unsigned int l_c = 0; l_c < 3; l_c++ ) {
11     m_matrixKernels[l_c]( i_stiffnessMatrices[l_c],
12                           i_timeIntegrated,
13                           l_temporaryResult,
14                           NULL, NULL, NULL );
15     m_matrixKernels[3](   l_temporaryResult,
16                           i_starMatrices[l_c],
17                           io_dofs,
18                           NULL, NULL, NULL );
19   }
20 }
```

nel shown in Lst. 1. The time kernel computes the time integrated DOFs in `i_timeIntegrated`. In addition it uses the three matrices $A_k^{\xi_c}$ in `i_starMatrices` and the DOFs in `io_dofs` for the derivative computation. Further the matrices K^{ξ_c} are global and used repeatedly in the element-local update (71) of every element. This is supported by our execution-aware storage of the global matrices introduced in Ch. 5.3. Here the matrices K^{ξ_c} directly follow matrices $\bar{K}^{\xi_c}$, thus hardware-prefetching is encouraged.

The structure of the volume kernel is similar to the derivative computation in the time kernel. Again, in lines 6-7, we define a page-size aligned array for the temporary data. Next, lines 10-19 iterate over the three dimensions ξ_c in the reference coordinate system and compute the corresponding volume contributions. Inside the loop we call a set of optimized matrix-multiplication kernels stored as function pointers in `m_matrixKernels`. This array belongs exclusively to the volume kernel and thus holds different pointers in comparison to the corresponding array of the time kernel. The first call in lines 11-14 of Lst. 2 multiplies the time integrated DOFs from the left with matrices K^{ξ_c} and stores the intermediate results. Next, in lines 15-18, we multiply the results of the previous matrix-matrix multiplication with $A_k^{\xi_c}$ from the right and add the result to the DOFs.

By using our function pointers `m_matrixKernels`, we are again able to hardwire individual matrix kernels for all four operations. We discuss details, including the respective tuning step, in the next chapters after the introduction of all ADER-DG kernels.

Listing 3: Low-level local surface kernel computing the element's contribution of the surface integration to a time step.

```
 1 void computeLocalSurface( enum faceType i_faceTypes[4],
 2                           real         *i_fluxMatrices[52],
 3                           real         *i_timeIntegrated,
 4                           real          i_fluxSolvers[4]
 5                             [NUMBER_OF_QUANTITIES*NUMBER_OF_QUANTITIES],
 6                           real         *io_dofs ) {
 7   // temporary product
 8   real l_temporaryResult[NUMBER_OF_ALIGNED_DOFS]
 9     __attribute__((aligned(PAGESIZE_STACK)));
10
11   for( unsigned int l_face = 0; l_face < 4; l_face++ ) {
12     // Riemann problem for dynamic rupture faces is solver separately
13     if( i_faceTypes[l_face] != dynamicRupture ) {
14       // compute neighboring elements contribution
15       m_matrixKernels[l_face]( i_fluxMatrices[l_face],
16                                i_timeIntegrated,
17                                l_temporaryResult,
18                                NULL, NULL, NULL );
19
20       m_matrixKernels[52](     l_temporaryResult,
21                                i_fluxSolvers[l_face],
22                                io_dofs,
23                                NULL, NULL, NULL );
24     }
25   }
26 }
```

LOCAL SURFACE KERNEL Our local surface kernel in Lst. 3 updates the DOFs with the second sum of (71). The first input parameter is the enumeration `i_faceType`, which encodes the types of the four faces. A pointer to the 52 global matrices $F^{-,i}$ and $F^{+,i,j,h}$, of which we only use the first four matrices $F^{-,i}$, is the second parameter. The remaining arguments pass the element-local data, which are the time integrated DOFs, the four matrices $A^{-}_{k,i}$ and the element's DOFs.

Considering the data-heavy arguments once again, the matrices $F^{-,i}$ most likely reside in low-level cache, because we repeatedly call the local surface kernel for our elements. The time integrated DOFs in `i_timeIntegrated` and DOFs in `io_dofs` have been touched in the previous calls of the time and volume kernel, thus most likely are still in cache. Matrices $A^{-}_{k,i}$ are untouched in general. However, for high-order simulation we assume that the hardware prefetcher recognizes our linear sorting of element-local data (see Ch. 5.3) and is able to shadow the loads from memory.

In lines 8-9 we once again define storage for our two-way matrix multiplications. The loop over the four faces of the tetrahedron is realized in lines 11-25. We only continue with the computation of a face's contribution if the face is not a dynamic rupture face (line 13), which is true for the vast majority of faces. Dynamic rupture faces use a specialized surface integration [49]. Next, lines 15-18 multiply the time integrated DOFs from the left with matrices $F^{-,i}$ and store the result in our intermediate storage. The second call of our matrix kernels in lines 20-23 multiplies the intermediate result from the right with a matrix $A^{-}_{k,i}$ and updates the DOFs with the contribution of the current face.

Again the class member `m_matrixKernels` allows us to hardwire optimized matrix-multiplication kernels for the five calls. However, the local surface kernel shares `m_matrixKernels` with the neighboring surface kernel, discussed in the next paragraph.

NEIGHBORING SURFACE KERNEL The neighboring elements' contribution to the surface integral (72), shown in Lst. 4, is our last ADER-DG kernel. While the time kernel, volume kernel, and local surface kernel are all part of our local work-items in Ch. 3.5, the neighboring surface kernel is the only ADER-DG kernel in the neighboring work-items.

The function parameters of the kernel are given in lines 1-7. The first parameter `i_faceTypes` contains, analogue to the local surface kernel in Lst. 3, the types of the four faces. Argument `i_neighboringIndices` contains, in dependency of the local face i, the id of the neighboring face $j_{k,i}$ and the orientation $h_{k,i}$ with respect to the reference element (see Ch. 2.1). `i_fluxMatrices` again contains the 52 matrices $F^{-,i}$ and $F^{+,i,j,h}$ (see Ch. 5.3).

The function parameter `i_timeIntegrated` in Lst. 4 points to the time integrated DOFs. If a face-neighbor provides derivatives, we have to assemble the time integrated DOFs via (29) first, before passing them to `computeNeighSurface` in Lst. 4. For this purpose, prior to the call of `computeNeighSurface`, we follow the procedure shown in Ch. 5.2 by computing the time integrated DOFs from the derivatives based on the LTS bitmask and storing them in a temporary array. Therefore, recently computed time integrated DOFs in `i_timeIntegrated` are most probably in a low cache-level. The location of the time buffers, which are used directly and not copied to the temporary array, depends on the data-locality of the neighboring elements.

However, in total, the hardware prefetcher most likely fails to predict these accesses of derivatives and buffers initially due to our unstructured tetrahedral meshes. For high-order simulations the neighboring accesses put the highest pressure on the memory subsystem. In future implementations software prefetches could be introduced in `computeNeighSurface` to ensure prefetching of neighboring data. In fact, the last three parameters of the functions in `m_matrixKernels`, currently addressed with `NULL`, prepare this step.

The last two parameters in lines 5-7 of Lst. 4 point to the matrices $A^{-}_{k,i}$ and the DOFs. Here, our memory layout ensures linear accesses and the matrices most likely reside in a low cache-level.

As in all other ADER-DG kernels, lines 10-11 define page-size aligned memory for our two-step matrix-matrix multiplications. The loop

Listing 4: Low-level neighboring surface kernel computing the neighboring elements' contribution of the surface integration to a time step.

```
void computeNeighSurface( enum faceType i_faceTypes[4],
                          int           i_neighboringIndices[4][2],
                          real         *i_fluxMatrices[52],
                          real         *i_timeIntegrated[4],
                          real          i_fluxSolvers[4]
                            [NUMBER_OF_QUANTITIES*NUMBER_OF_QUANTITIES],
                          real         *io_dofs ) {

  // temporary product
  real l_temporaryResult[NUMBER_OF_ALIGNED_DOFS]
    __attribute__((aligned(PAGESIZE_STACK)));

  // iterate over faces
  for( unsigned int l_face = 0; l_face < 4; l_face++ ) {
    // absorbing: no contribution, dynamic rupture: separate
    if( i_faceTypes[l_face] != outflow &&
        i_faceTypes[l_face] != dynamicRupture ) {
      // id of the flux matrix (0-3: local, 4-51: neighboring)
      unsigned int l_id;

      // compute the neighboring elements flux matrix id.
      if( i_faceTypes[l_face] != freeSurface ) {
        // derive memory and kernel index
        l_id = 4                                   // jump: F^{-, i}
              + l_face*12                          // jump: i
              + i_neighboringIndices[l_face][0]*3  // jump: j
              + i_neighboringIndices[l_face][1];   // jump: h
      }
      else { // free surface: fall back to F^{-, i}
        l_id = l_face;
      }

      // compute neighboring elements contribution
      m_matrixKernels[l_id]( i_fluxMatrices[l_id],
                             i_timeIntegrated[l_face],
                             l_temporaryResult,
                             NULL, NULL, NULL );

      m_matrixKernels[53](   l_temporaryResult,
                             i_fluxSolvers[l_face],
                             io_degreesOfFreedom,
                             NULL, NULL, NULL );
    }
  }
}
```

over the faces in lines 14-44 only processes a face if it is not a dynamic rupture face and no outflow boundary conditions are set (see lines 16-17). In the case of dynamic rupture faces the surface integral is computed separately [49] and in the case of outflow boundary conditions no neighboring surface-contribution is applied at all [23]. We derive the face's id in `i_fluxMatrices` in lines 21-31. In the case of free-surface boundary conditions we use $\mathrm{F}^{-,i}$ [23]. In all other cases we follow our ascending order of Ch. 5.3 to get the right matrix $\mathrm{F}^{+,i,j,h}$.

In lines 34-37 we multiply the time integrated DOFs from the left with matrices $\mathrm{F}^{+,i,j,h}$ or $\mathrm{F}^{-,i}$ in the case of free-surface boundary conditions and store the intermediate result. Finally, in lines 39-42, we multiply the intermediate result from the right with $\mathrm{A}^{+}_{k,i}$ and update the DOFs with the face's contribution. After all faces are processed in Lst. 4, the element is at the next time step.

The class member `m_matrixKernels` is identical to the local surface kernel in Lst. 3. In addition to the four shared matrix-multiplication kernels performing the multiplications with $\mathrm{F}^{-,i}$ from the left, we are able to hardwire 49 additional kernels for the multiplications with $\mathrm{F}^{+,i,j,h}$ and $\mathrm{A}^{+}_{k,i}$.

7.2 MATRIX KERNELS

The performance of our four ADER-DG kernels in Ch. 7.1 is dominated by small-size matrix-matrix multiplications in their inner-loops. For this purpose all three embedding classes define an array `m_matrixKernels` to hardwire optimized routines for this task.

In this chapter we introduce the identifiers formalizing our matrix-matrix operations and the usage of the libxsmm-library[1] for the generation of our low-level matrix kernels.

The next chapter discusses our auto-tuning approach in preprocessing, which chooses an optimized set of matrix kernels for a given configuration. These sets are then hardwired to the ADER-DG kernels using the preprocessor.

IDENTIFIERS We use standard BLAS-DGEMM identifiers for our matrix-matrix operations. Fig. 11 illustrates our identifiers for the operation $C = \beta C + AB$ with $\beta \in \{0,1\}$. M is the number of rows of matrices A and C, N is the number of columns of matrices B and C and K is A's number of columns and B's number of rows. $\mathrm{ld}A$, $\mathrm{ld}B$ and $\mathrm{ld}C$ are the leading dimensions of our matrices and thus encode our zero-padding used for vector-alignment. As discussed in Ch. 5.3 we allow sparse storage of matrices $\bar{\mathrm{K}}^{\xi_c}$, K^{ξ_c}, $\mathrm{F}^{-,i}$, $\mathrm{F}^{+,i,j,h}$ and $\mathrm{A}^{\xi_c}_{k}$. In this case no leading dimension is defined and we set $\mathrm{ld}A = -1$ for a sparse matrix in $\bar{\mathrm{K}}^{\xi_c}$, K^{ξ_c}, $\mathrm{F}^{-,i}$, $\mathrm{F}^{+,i,j,h}$ or $\mathrm{ld}B = -1$ if matrices $\mathrm{A}^{\xi_c}_{k}$ are sparse. Based on the order of convergence and vector-alignment we

1 https://github.com/hfp/libxsmm

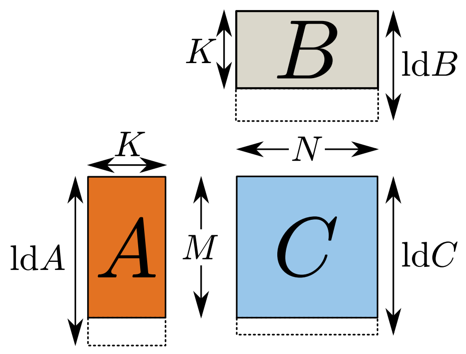


Figure 11: Illustration of the BLAS-DGEMM identifiers for the matrix-matrix product $C = \beta C + AB$.

	$\bar{\mathrm{K}}^{\xi_c} \frac{\partial^d}{\partial t^d} Q_k$						$\left(\bar{\mathrm{K}}^{\xi_c} \frac{\partial^d}{\partial t^d} Q_k\right) \mathrm{A}_k^{\xi_c}$					
d	M	N	K	$\mathrm{ld}A$	$\mathrm{ld}B$	$\mathrm{ld}C$	M	N	K	$\mathrm{ld}A$	$\mathrm{ld}B$	$\mathrm{ld}C$
1	35	9	56	36	56	36	35	9	9	36	9	36
2	20	9	35	36	36	20	20	9	9	20	9	20
3	10	9	20	36	20	12	10	9	9	12	9	12
4	4	9	10	36	12	4	4	9	9	4	9	4
5	1	9	4	36	4	4	1	9	9	4	9	4

Table 2: BLAS-DGEMM identifiers of the time kernel for a sixth order method. Shown are the identifiers for double-precision and 32-byte vector-alignment in dependency of the d^{th} derivative computation.

are now able to derive the identifiers for all matrix-matrix multiplications in our ADER-DG kernels of Ch. 7.1. This leaves the sparse-dense decision as only open parameter for our auto-tuning in the next chapter.

Tab. 2 shows our identifiers for a sixth order time kernel (see Lst. 1) in double-precision with 32-byte vector-alignment. The identifiers are given in dependency of the d^{th} derivative computation in lines 35-62 of Lst. 1. Identifiers M, N, K only operate on non-zero blocks and therefore exploit the decreasing number of non-zeros discussed in Ch. 5.1. The leading dimensions match our vector-aligned memory layout of Ch. 5.3. The first multiplication in lines 40-44 overwrites the intermediate array, therefore we additionally set $\beta = 0$. After multiplication with $\bar{\mathrm{K}}^{\xi_c}$ from the left, the intermediate result already contains the newly generated zero-block. Therefore in Tab. 2 the M of the multiplication with $\mathrm{A}_k^{\xi_c}$ from the right (see lines 46-50 of Lst. 1) is identical to the M of the first multiplication. Our second multiplication with matrices $\mathrm{A}_k^{\xi_c}$ updates the current derivative with the contribution of dimension ξ_c. Thus we add the results by setting $\beta = 1$.

The identifiers for the volume kernel (see Lst. 2) are shown in Tab. 3. In this case we show all dense identifiers for orders 2-7 and 32-byte

	$\mathrm{K}^{\xi_c}\mathcal{T}_k$						$(\mathrm{K}^{\xi_c}\mathcal{T}_k)\,\mathrm{A}^{\xi_c}_k$					
$\mathcal{O}$	M	N	K	ld$A$	ld$B$	ld$C$	M	N	K	ld$A$	ld$B$	ld$C$
2	4	9	1	4	4	4	4	9	9	4	9	4
3	10	9	4	12	12	12	10	9	9	12	9	12
4	20	9	10	20	20	20	20	9	9	20	9	20
5	35	9	20	36	36	36	35	9	9	36	9	36
6	56	9	35	56	56	56	56	9	9	56	9	56
7	84	9	56	84	84	84	84	9	9	84	9	84

Table 3: BLAS-DGEMM identifiers of the volume kernel for orders 2-7. Shown are the identifiers for double-precision and 32-byte vector-alignment in dependency of the order $\mathcal{O}$.

	$\mathrm{F}^{-,i}\mathcal{T}_k \vee \mathrm{F}^{+,i,j,h}\mathcal{T}_k$						$(\mathrm{F}^{-,i}\mathcal{T}_k)\,A^{-}_{k,i} \vee (\mathrm{F}^{+,i,j,h}\mathcal{T}_k)\,A^{+}_{k,i}$					
$\mathcal{O}$	M	N	K	ld$A$	ld$B$	ld$C$	M	N	K	ld$A$	ld$B$	ld$C$
2	4	9	4	4	4	4	4	9	9	4	9	4
3	10	9	10	12	12	12	10	9	9	12	9	12
4	20	9	20	20	20	20	20	9	9	20	9	20
5	35	9	35	36	36	36	35	9	9	36	9	36
6	56	9	56	56	56	56	56	9	9	56	9	56
7	84	9	84	84	84	84	84	9	9	84	9	84

Table 4: BLAS-DGEMM identifiers of the local and neighboring surface kernel for orders 2-7. Shown are the identifiers for double-precision and 32-byte vector-alignment in dependency of the order $\mathcal{O}$.

vector-alignment. Since the sparsity patterns of matrices K^{ξ_c} are the transposed patterns of the time kernel's matrices $\tilde{\mathrm{K}}^{\xi_c}$ (see Ch. 5.1), we adjust the size of the multiplication with matrices K^{ξ_c} from the left (see lines 11-14 of Lst. 2) via K. The leading dimensions of these multiplications match our vector-alignment of Ch. 5.3 and we set $\beta = 0$. The multiplication with matrices $\mathrm{A}^{\xi_c}_k$ from the right in the volume kernel (see lines 15-18 of Lst. 2) is similar to those of the time kernel. However, this time no zero-blocks are generated in the first multiplication and identifiers M, ld$A$ and ld$C$ match the full number of basis functions. Again we set $\beta = 1$ since this time we update the DOFs with dimension ξ_c's contribution.

Tab. 4's identifiers are our last set and show the settings for the local surface kernel in Lst. 3 and the neighboring surface kernel in Lst. 4. When multiplying our time integrated DOFs with matrices $F^{-,i}$ and $F^{+,i,j,h}$ from the left (see lines 15-18 of Lst. 3 and lines 34-37 of Lst. 4) we can not exploit any zero blocks and thus cover the full number of basis functions in M, K and their aligned equivalents ld$A$, ld$B$ and ld$C$. Again we set $\beta = 0$ for the first multiplication. The identifiers for multiplication with matrices $A^-_{k,i}$ and $A^+_{k,i}$ from the right are identical to those of the volume kernel in Tab. 3 and we set $\beta = 1$. However, this is the only case where we are not allowed to use sparse-multiplications, since matrices $A^-_{k,i}$ and $A^+_{k,i}$ are fully populated in general. All other matrix operations allow for sparse-settings and thus allow setting ld$A = -1$ for multiplications from the left or ld$B = -1$ for multiplications from the right.

KERNEL GENERATION We use the libxsmm-library for the actual generation of our matrix kernels. libxsmm targets at small-size matrix-matrix multiplications and supports the SSE3, AVX, AVX2, AVX-512 and Knights Corner vector instruction sets. Initially the used code-generator was developed as part of SeisSol and later integrated into libxsmm [8, 9, 31, 10].

libxsmm uses our DGEMM-identifiers and the desired vector instruction set as input. The result is a generated, optimized function for the specified sparse or dense matrix-matrix multiplication. Our DGEMM-identifiers are sufficient for dense matrix kernels. For the generation of sparse matrix kernels, we also pass the sparsity patterns to libxsmm using the Matrix Market format [7].

Three different approaches are used within libxsmm for the code-generation of our three possible sparse-dense combinations.

In the first case we multiply a sparse matrix from the left to a dense matrix and obtain a dense matrix as result (ld$A = -1$). Here libxsmm summarizes continuous matrix-entries in the columns of A and generates vectorized code by using intrinsic functions. Similar to the more general code-unrolling approach, the respective sparsity patterns are hardwired into the generated source code [8].

The second case multiplies a sparse matrix from the right to a dense matrix and results in a dense matrix (ld$B = -1$). Since we use a column-major format, the vectorization of this operation is a simple code unrolling. Here libxsmm generates C++-code, which is vectorized by the compiler's auto-vectorizer [8].

The last case assumes an all-dense matrix kernel. In this case the library generates optimized assembly code, which directly translates into machine instructions. Compared to intrinsic functions, the assembly code completely bypasses the compiler. This allows for custom, optimized register management [10].

7.3 ARITHMETIC INTENSITY

The definition of our ADER-DG kernels leads to the question what performance we can expect from our implementation in practice. For this purpose we derive the *arithmetic intensity* of our computational core. The arithmetic intensity summarizes how many floating point operations we perform in relation to memory transfers. With the arithmetic intensity at hand we can estimate the performance of our computational core for different architectures. Here one of the most important characteristics is the *machine balance*. The machine balance sets the sustained memory bandwidth of a machine in relation to its peak floating point performance (GiBS/GFLOPS). Combining arithmetic intensity and machine balance, we can estimate if our computational core's workload is in the memory-bound or compute-bound regime. Typical state-of-the-art architectures have a machine balance between 0.1 and 0.25 in double precision. Thus an arithmetic intensity of at least 4-10 double-precision floating point operations per byte is required to fully utilize the computational capabilities of the respective architecture.

We limit the complexity of our derivation by considering global time stepping and dense-matrix kernels only. Additionally we ignore our vector-alignment for the time-being and assume that neither the data required for the local operations nor the data required for the neighboring operations completely fits into last-level cache.

Our loop over the elements performing local operations successively computes the time kernel, volume kernel and local surface kernel for every element (see Ch. 6.1). Thus we have to read the DOFs, the three matrices $\text{A}_k^{\xi_c}$ and the four matrices $\text{A}_k^{-,i}$ for every element from memory. Since we write time buffers in global time stepping, we also have to load the element's buffer to satisfy the read-before-write requirement. The only data written back to memory are the updated DOFs and the time buffer. Every of the constant matrices requires to transfer 648 bytes in double-precision, while the size of the DOFs and buffers follows Tab. 1.

As floating point operations we count the dominating matrix-matrix multiplications of the ADER-DG kernels only. Here we have to consider the multiplications with $\bar{\text{K}}^{\xi_c}$ and $\text{A}_k^{\xi_c}$ in the recursive derivative computation of Lst. 1. Following the general scheme of the sixth order example in Tab. 2, the number of floating point operations for every multiplication is given by $M \cdot N \cdot K \cdot 2$. Additionally, we use the matrix identifiers in Tab. 3 and Tab. 4 for the derivation of floating point operations in the volume kernel (see Lst. 2) and the local surface kernel (see Lst. 3).

Our second loop in Ch. 6.1 computes the neighboring surface kernel for every element. We require the time buffers of the element itself and those of its four face neighbors. Depending on the mesh structure this data might have been accessed within the same iteration and still be in cache. Thus in addition to considering the sizes following Tab. 1, we also model the cache hits. Here we assume fixed ratio of buffer-data already in cache for every call of the neighboring surface kernel in Lst. 4. In addition to the time buffers we have to read the element's

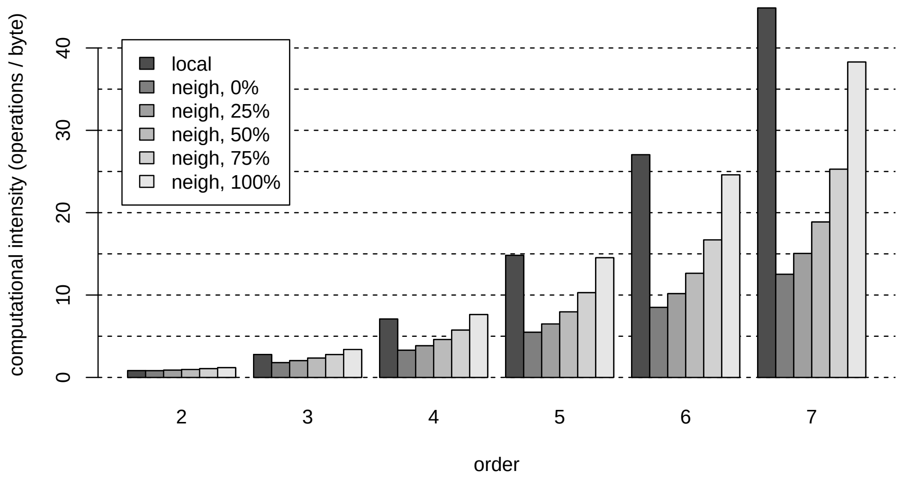


Figure 12: Arithmetic intensity of the computational core using global time stepping and double-precision arithmetic. The intensities for the two element-loops performing local and neighboring operations are given separately. For the neighboring loop different cache-hit ratios (0%, 25%, 50%, 75%, 100%) with respect to time buffer accesses are assumed.

DOFs and the four matrices $A_k^{+,i}$. After finishing all computations the DOFs are the only data, which gets written back to memory.

In terms of floating point operations we have to account for the four multiplications with $F^{+,i,j,h}$ and those with $A_k^{+,i}$. For this purpose we use the identifiers M, N and K given in Tab. 4.

Fig. 12 shows the resulting arithmetic intensities resulting from our considerations. We distinguish between the arithmetic intensity of the local loop and that of the neighboring loop, since these are different in general. For example it might be possible to have a compute-bound workload in the local loop, while the neighboring loop is bound by memory bandwidth. The arithmetic intensities of the neighboring loop are further split into different assumed cache-hit ratios when reading the time buffer data.

In general we observe a clear trend of growing arithmetic intensities for higher orders. Assuming a machine balance of 0.2, we would leave the memory-bound regime at order $\mathcal{O} = 4$ for the local loop and, depending on the cache-hit ratio, at order $\mathcal{O} = 4$ or order $\mathcal{O} = 5$ for the neighboring loop.

7.4 AUTO-TUNING

To decide whether computing a specific matrix-matrix product as sparse or as dense is advantageous, we perform an auto-tuning step in preprocessing. Due to the design of our ADER-DG kernels in Ch. 7.1, we are able to hardwire a specialized implementation to all our function pointers. For every supported architecture, order of convergence and precision, we have to consider the $(\mathcal{O}-1)\cdot 4$ matrix kernels of the time derivative computation in Lst. 1, the 4 kernels of the volume kernel in Lst. 2 and the 54 kernels of the surface integration in Lst. 3 and Lst. 4. The individual matrix kernels are combined in our ADER-DG kernels, which are again combined via element-loops over (71) and (72). Therefore, our auto-tuning has to model the dependencies of the matrix kernels and their background accesses to the memory hierarchy. For example tuning every possible matrix kernel separately on a hot L2-cache is not sufficient, since data accesses might dominate or cache trashing could happen when multiple matrix kernels are executed in series.

TUNING SETUP We use a proxy application to mimic the performance characteristics of our computational core on a single node. For this purpose, we assume a given number of elements and imitate the unstructured tetrahedral mesh by a random data structure. Thus all all required initialization, such as reading the mesh, can be skipped in our tuning runs. To every fake-element our data structure simply assigns four random elements as "neighboring" elements, random identifiers j and h for the matrices $F^{i,j,h}$ and a global time stepping LTS setup. The remaining layout is identical to our computational core's memory layout in Ch. 5.3 and our shared-memory parallelization in Ch. 6.1.

The proxy's error for single node-performance using global time stepping is typically in the order of a few percent, thus we are able to mimic our computational core's performance for an arbitrary number of elements and time steps without a significant initialization overhead. However, due to the large number of possible sparse-dense combinations, we have to further reduce the complexity of our tuning runs. Recapitulating that we compute the local and neighboring operations independently in Ch. 6.1 and for a large number of elements in general, we are able to split our auto-tuning into two steps. The first step tunes all matrix kernels in the time kernel, volume kernel and local surface kernel, while the second steps tunes the matrix kernels in the neighboring surface kernel only.

Further, inside an ADER-DG kernel, we assume that a global matrix with less non-zero entries performs better in sparse than one with more non-zero entries. Consider for example the four matrix kernels of the global matrices $F^{-,i}$ in the local surface kernel in Lst. 3. As visualized for a sixth order scheme in the second sum of Fig. 7, the matrices $F^{-,1}$, $F^{-,2}$, $F^{-,3}$ and $F^{-,4}$ have an increasing number of non-zeros. Here we assume that if computing $F^{-,2}$ as sparse is beneficial then this is also the case for $F^{-,1}$. Thus our tuning runs test computing only $F^{-,1}$ as sparse, computing $F^{-,1} \wedge F^{-,2}$ as sparse, $F^{-,1} \wedge F^{-,2} \wedge F^{-3}$ as sparse and also test computing $F^{-,1} \wedge F^{-,2} \wedge F^{-3} \wedge F^{-3}$ as sparse. Configurations testing, for example, if computation of only $F^{-,2}$ or

$F^{-,3} \wedge F^{-,3}$ as sparse is advantageous are skipped. We follow the same approach for the matrices $F^{+,i,j,h}$, but sort them by the number of non-zero entries first and only test computing the first 16 matrices as sparse.

We reduce the complexity of our auto-tuning further by using the same sparse decision for all matching recursive matrix kernels in the time derivative computation of Lst. 1. For example, when the multiplication from the left with $\bar{K}^{\xi_1}$ is sparse on the first level ($d = 1$) in Lst. 1, we use a sparse matrix kernel for $\bar{K}^{\xi_1}$ on all following levels ($d \geq 1$). Also we simultaneously switch matrices $A_k^{\xi_c}$ to sparse in the time and volume kernel.

In summary our auto-tuning runs for the time, volume and local surface kernel cover a total of 160 configurations:

- Compute matrices $A_k^{\xi_c}$ as dense or sparse.
- Compute either all matrices $\bar{K}^{\xi_c}$ as dense or test setting $\bar{K}^{\xi_1}$ or $\bar{K}^{\xi_1} \wedge \bar{K}^{\xi_2}$ or $\bar{K}^{\xi_1} \wedge \bar{K}^{\xi_2} \wedge \bar{K}^{\xi_2}$ sparse.
- Compute either all matrices K^{ξ_c} as dense or test setting K^{ξ_1} or $K^{\xi_1} \wedge K^{\xi_2}$ or $K^{\xi_1} \wedge K^{\xi_2} \wedge K^{\xi_2}$ sparse.
- Compute either all matrices $F^{-,i}$ as dense or test setting $F^{-,1}$ or $F^{-,1} \wedge F^{-,2}$ or $F^{-,1} \wedge F^{-,2} \wedge F^{-,3}$ or $F^{-,1} \wedge F^{-,2} \wedge F^{-,3} \wedge F^{-,4}$ sparse.

For our neighboring surface kernel we consider the 17 sparse-dense configurations given by the non-zero sorting:

- Compute either all matrices $F^{+,i,j,h}$ as dense or test setting $F^{+,1,1,3}$, $F^{+,1,1,1}$, $F^{+,1,1,2}$, $F^{+,2,2,2}$, $F^{+,2,1,1}$, $F^{+,1,2,1}$, $F^{+,4,3,1}$, $F^{+,3,4,1}$, $F^{+,2,4,2}$, $F^{+,3,2,1}$, $F^{+,4,2,2}$, $F^{+,2,3,1}$, $F^{+,2,1,2}$, $F^{+,2,1,3}$, $F^{+,1,2,2}$ and $F^{+,1,2,3}$ sparse in ascending order ($F^{+,1,1,3}$, or $F^{+,1,1,3} \wedge F^{+,1,1,1}$, etc.).

Every time step of our tuning runs computes both, the local and the neighboring loop in Ch. 6.1. However we leave all matrix kernels of the loop not involved in the current tuning as dense.

SNB-MUC AND HSW-MUC Next, we discuss the details of our auto-tuning for the machines SuperMUC-1 and SuperMUC-2. To measure reliable overall runtime improvements, we exclude caching effects across elements from our auto-tuning runs. For this purpose, in conjunction with our random neighboring accesses, we use a fixed number of fake-elements for every order, such that our element-local data (see Ch. 5.3) occupies at least 5 GiB of memory in every run. Noise in a single-node is accounted for by adjusting the number of time steps. Here, we ensure at least one minute of computational time in every tuning run, which results in 17 to 850 time steps, depending on the order and architecture. Additionally, we repeat every configuration 50 times and use a different node in every repeat to account for the machine's overall noise in terms of node performance.

For SuperMUC-1's 16-core nodes we use 16 OpenMP threads in our runs. Every thread is pinned to the first hyperthread of a core, which is faster than oversubscribing the cores via the Intel Hyper-Threading-Technology (HTT). All runs use the 2.7 GHz base frequency of the SNB-MUC nodes with Intel Turbo Boost Technology disabled. This matches the production configuration of SuperMUC-1.

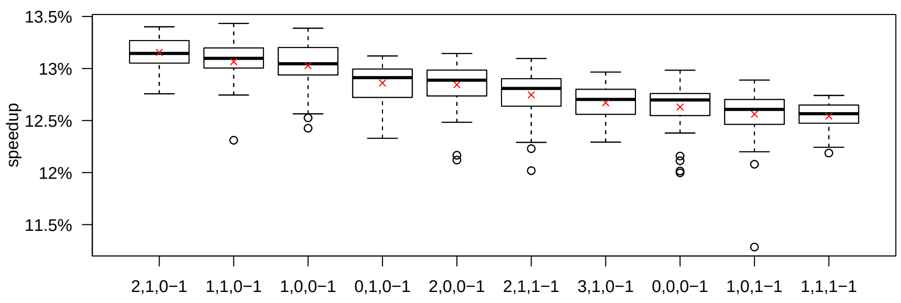
In our tuning runs for HSW-MUC we use 56 threads on the 28-core nodes of SuperMUC-2 and thus utilize HTT. This setting is slightly different from our multi-node settings in the next chapters, where one core is left for our communication thread (see Ch. 6.2). However, the relative results of the auto-tuning are not influenced by this. All runs on SuperMUC-2 use the 2.6 GHz base frequency of the HSW-MUC nodes. The cluster-on-die feature is enabled and can be fully utilized by our NUMA-aware memory layout of Ch. 5.3.

Fig. 13 and Fig. 14 show the results of our auto-tuning for convergence rates $\mathcal{O} = 2$ and $\mathcal{O} = 6$. For every tuning we derive the mean computational time of the 50 repeats using the all-dense configuration. On this basis we compute the speedup of every repeated sparse configuration. The speedup is illustrated with respect to the mean time of the all-dense configuration using boxplots. Additionally, for every configuration the mean speedup is visualized using a red cross. Every plot in figures 13 and 14 contains the ten best performing configurations with respect to their mean speedup.

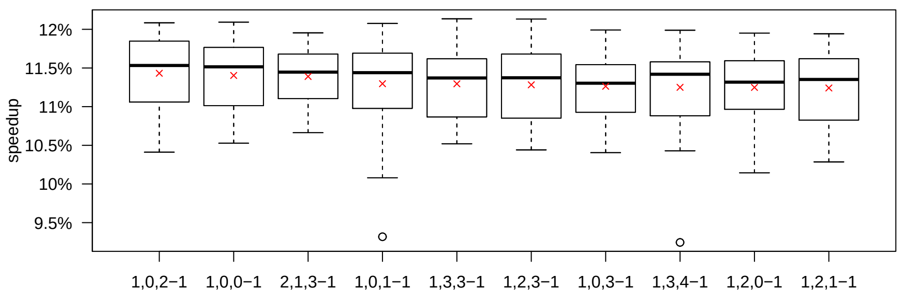
The results of the local tuning runs are given in figures 13a, 13b, 14a and 14b. The x-labels of the form a,b,c-d summarize the used sparse-dense configuration for the local operations, while the neighboring configuration stayed all-dense. Here, a is the number of (recursively) used sparse configurations for matrices $\bar{K}^{\xi_c}$, b the number of sparse configurations for K^{ξ_c} and c that of $F^{-,i}$. The last identifier d is 1 if the matrix multiplications with $A_k^{\xi_c}$ from the right are computed as sparse in the time and volume kernel. For example the fastest configuration in Fig. 13a is 2,1,0-1. This translates to a configuration which computed matrices $\bar{K}^{\xi_1} \wedge \bar{K}^{\xi_2}$, K^{ξ_1} and matrices $A_k^{\xi_c}$ as sparse.

Figures 13c, 13d, 14c and 14d show the results of the neighboring tuning runs. In this case the x-labels show the number of used sparse-configuration for matrices $F^{i,j,h}$. For example the label 3 of the best performing configuration in Fig. 14c summarizes sparse computation of matrices $F^{+,1,1,3} \wedge F^{+,1,1,1} \wedge F^{+,1,1,2}$.

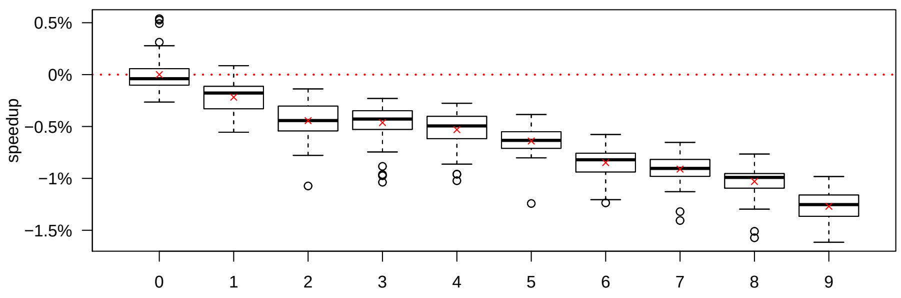
For the low-order runs ($\mathcal{O} = 2$) our best configuration for SNB-MUC achieves a speedup of 13.2%, while the best configuration on HSW-MUC reaches a speedup of 11.4%. Considering our discussion of arithmetic intensities in Ch. 7.3, these results match the expectations of a memory-bound configuration. In a dense configuration the local kernels access (r+w) a total of 5,112 bytes and the neighboring surface kernel accesses a total of 4,320 bytes. Computing the element-local matrices $A_k^{\xi_c}$ as sparse, as in all top configurations of Fig. 13a



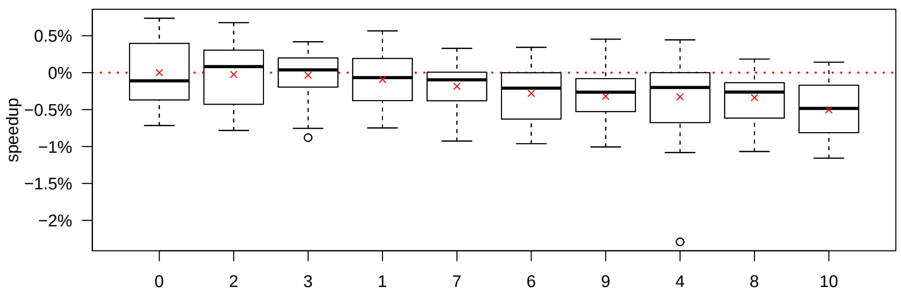
(a) Local: SNB-MUC.



(b) Local: HSW-MUC.



(c) Neighboring: SNB-MUC.



(d) Neighboring: HSW-MUC.

Figure 13: Results of the auto-tuning runs for order $\mathcal{O} = 2$. Shown are boxplots for the speedups in relation to the mean value of the all-dense configuration. Additionally red crosses mark the mean speedups of the configurations.
Local runs: The first value of the x-labels is the number of sparse configurations for $\bar{K}^{\xi_c}$, the second value is the number for K^{ξ_c} and the third value is the number for $F^{-,i}$. The last value is 1, if the matrices $A_k^{\xi_c}$ are computed as sparse in the time and volume kernel. Neighboring runs: The labels give the number of sparse matrix kernels for $F^{i,j,h}$.

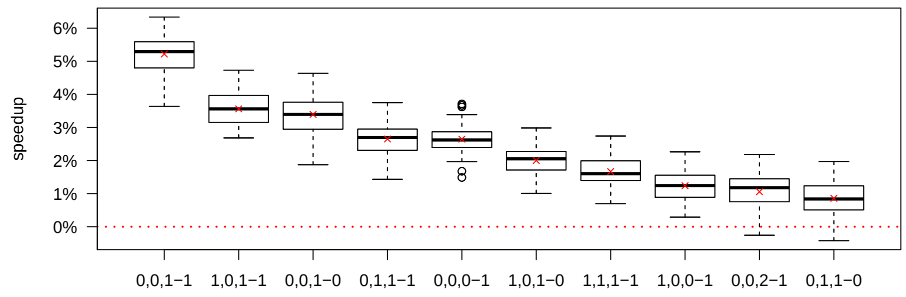
and Fig. 13b, reduces the memory accesses of the local kernels to a total of 3,744 bytes. This corresponds to a 14.5% reduction of memory transfers. In comparison to setting all matrix kernels involving $A_k^{\xi_c}$ to sparse, the remaining settings have a small influence on the performance of the top-settings for SNB-MUC and no distinguishable influence for HSW-MUC.

The results of the low-order neighboring tuning runs in Fig. 13c and Fig. 13d suggest computing all matrix kernels as dense on SNB-MUC and HSW-MUC. Due to the memory bottleneck this behavior is expected.

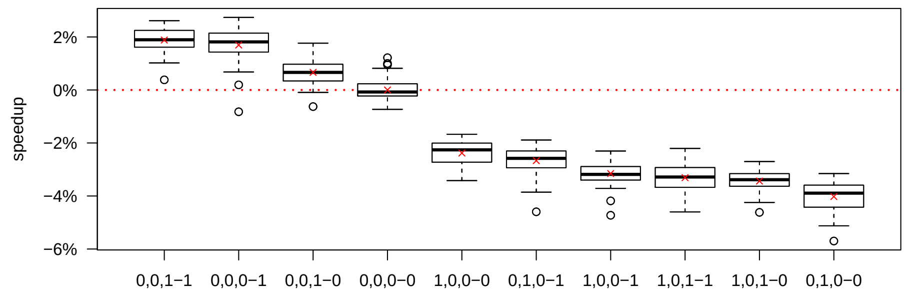
Moving to the high-order ($\mathcal{O} = 6$) runs in Fig. 14 we switch to the compute-bound regime of our auto-tuning. In general we now expect a positive or negative influence of all our sparse-dense decisions. The best performing configurations of our tuning for the local kernels are given in Fig. 14a for SNB-MUC and in Fig. 14b for HSW-MUC. For both architectures 0,0,1-1 is the best performing setting. This means that from a time-to-solution perspective matrix kernels involving $F^{-,1}$ and $A_k^{\xi_c}$ should be computed as sparse. For SNB-MUC an overall speedup (including the all-dense neighboring surface kernel) of 5.2% is achieved and for HSW-MUC a speedup of 1.9%. We can explain the optimal configuration, when setting our tuning-results in relation to the sparsity patterns shown in Fig. 6 and Fig. 7. Matrix $F^{-,1}$ is the global matrix in the local kernels with the fewest non-zero entries. Thus for the remaining global matrices the additional overhead of the sparse computations already dominates the potential performance improvement stemming from fewer floating point operations. In contrast, the matrices $A_k^{\xi_c}$ are multiplied to derivatives or time integrated DOFs from the right. Together with our column-major storage scheme in the basis functions the corresponding sparse kernels are perfectly vectorizable with a comparable small overhead introduced by the unrolling.

The results of the sparse-dense tuning for the neighboring surface kernel are shown in Fig. 14c for SNB-MUC and in Fig. 14d for HSW-MUC. Here, the best performing configuration for SNB-MUC computes matrices $F^{+,1,1,3} \wedge F^{+,1,1,1} \wedge F^{+,1,1,2}$ as sparse with a speedup of 0.6%. For HSW-MUC the best configuration achieves 0.4% by computing four matrices as sparse. In the case of the neighboring surface kernel, we have to remember that every matrix is called with the probability of $1/12$ due to our random initialization. Hence the possible impact of each individual sparse-dense switch is smaller than in the local tuning runs.

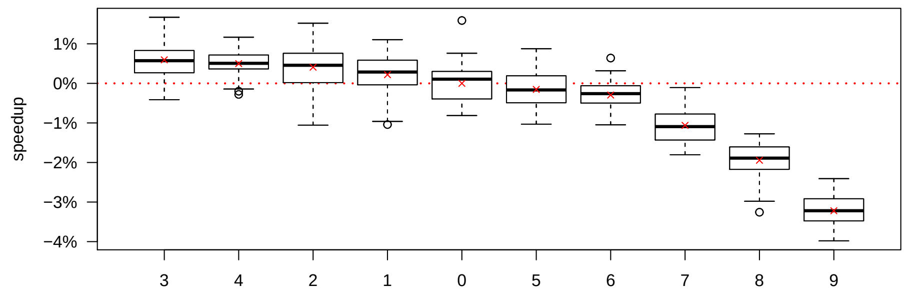
In summary, our results show that HSW-MUC has a worse sparse performance than SNB-MUC when executing high-order configurations. This is in agreement with the general trend of state-of-the-art computing architectures favoring floating point intensive, regular, vectorizable workloads.



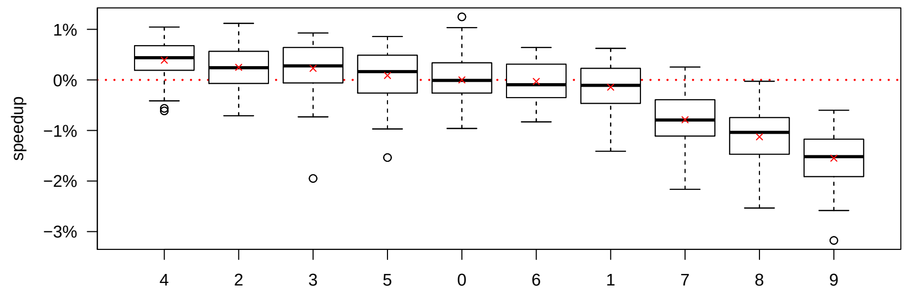
(a) Local: SNB-MUC.



(b) Local: HSW-MUC.



(c) Neighboring: SNB-MUC.



(d) Neighboring: HSW-MUC.

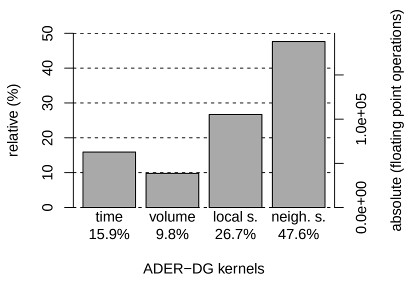
Figure 14: Results of the auto-tuning runs for order $\mathcal{O} = 6$. Shown are boxplots for the speedups in relation to the mean value of the all-dense configuration. Additionally red crosses mark the mean speedups of the configurations.
Local runs: The first value of the x-labels is the number of sparse configurations for $\bar{K}^{\xi_c}$, the second value is the number for K^{ξ_c} and the third value is the number for $F^{-,i}$. The last value is 1, if the matrices $A_k^{\xi_c}$ are computed as sparse in the time and volume kernel. Neighboring runs: The labels give the number of sparse matrix kernels for $F^{i,j,h}$.

FLOPS Our auto-tuning trades the irregular structures of most possible sparse matrix operators for the regular structures of the corresponding dense operators. However, using dense matrix kernels for sparse matrices comes with a non-optimal number of floating point operations. Thus, when considering the sustained performance, it is important to distinguish our floating point operations by their value. For this purpose we use the term *non-zero* floating point operations when referring only to operations contributing to the solution.

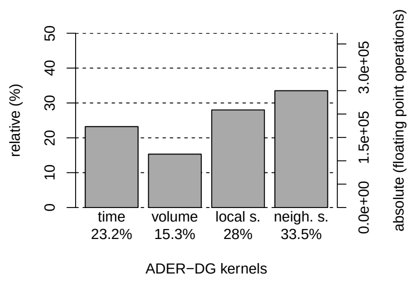
Fig. 15 shows our ADER-DG and matrix kernels in terms of floating point operations required for a single element-update using a sixth order method. All performed operations for an element-update in global time stepping are also computed in a local time stepping update. Ignoring the small overhead introduced by the computation of time integrated DOFs from derivatives, all following considerations directly translate to local time stepping on an element-update basis. Figures 15a and 15c show the non-zero floating point operations of an element-update, while figures 15b and 15d show operations in hardware and thus consider the computation of dense matrix kernels. We assume for all kernels that the matrices $F^{i,j,h}$ are used equally often and thus weight their contributions equally with $1/12$. In the case of hardware operations we assume computing the matrix kernels for $A_k^{\xi_c}$, $F^{-,i}$ and for $F^{+,1,1,3} \wedge F^{+,1,1,1} \wedge F^{+,1,1,2}$ as sparse.

With respect to non-zero operations in Fig. 15a, the local and neighboring loop perform almost the same number of floating point operations. The neighboring surface kernel performs most non-zero operations (47.6%), followed by the local surface kernel (26.7%), the time kernel (15.9%) and the volume kernel (9.8%). Considering the operations performed in hardware (see Fig. 15b) the loading of the kernels gets more homogeneous. Now the neighboring loop contains only 33.5% of the operations, while remaining operations are distributed among the local surface kernel with 28%, the time kernel with 23.2% and the volume kernel with 15.3%. Most importantly, by moving from non-zero operations in Fig. 15a to hardware floating point operations in Fig. 15b, we increase the absolute number of operations by a factor of 1.89. This can be interpreted as the price our ADER-DG scheme has to pay for minimal time-to-solution on state-of-the-art hardware.

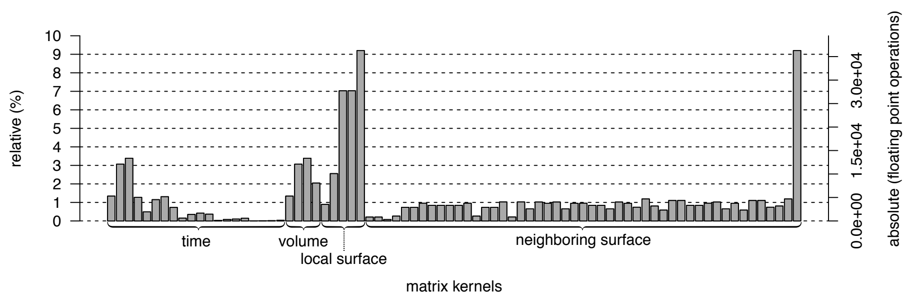
Fig. 15c and Fig. 15d go one step further and split the ADER-DG kernels in their individual matrix kernels. In this case, the illustration follows the ordering of our function pointers in listings 1, 2, 3 and 4. We keep the weighting of $1/12$ for matrix kernels using matrices $F^{i,j,h}$. However, we sum all the three calls of the matrix kernels computing the multiplications with matrices $A_k^{\xi_c}$ from the right in the time and volume kernel. Additionally, all four calls of matrix kernels computing multiplications with $A_k^{-,i}$ and $A_k^{+,i}$ from the right are combined in the surface kernels. The splitting of the time kernel, shown for non-zero operations in Fig. 15c and hardware operations in Fig. 15d,



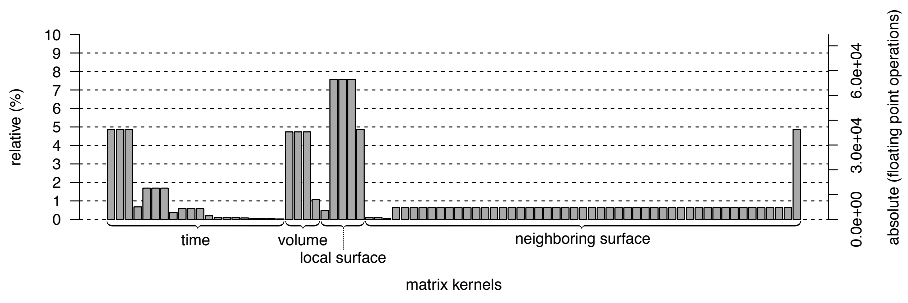
(a) Non-zero floating point operations of the ADER-DG kernels.



(b) Floating point operations of the ADER-DG kernels in hardware.



(c) Non-zero floating point operations of the matrix kernels.



(d) Floating point operations of the matrix kernels in hardware.

Figure 15: Relative and absolute number of floating point operations in the ADER-DG and matrix kernels for a sixth order configuration. A sparse-dense tuning of 0,0,1-1 is used for the local operations and of 3 for the neighboring operations. The left y-axis shows the relative number of performed floating operations with respect to an entire element update. The right y-axis gives the absolute number of floating point operations in a single element update. The matrix kernels of matrices $F^{i,j,h}$ are weighted with a factor of 1/12.

nicely shows the benefit of our zero-block exploit. While the kernels for matrices $\bar{K}^{\xi_c}$ in computation of the first derivative contribute each approximately 5% to the overall load in hardware, this already reduces below 2% for the second derivative and to ~0.5% for the third derivative.

We are now able to analytically derive the sustained GFLOPS for all our runs. For example the two local tuning runs in Fig. 14a and in Fig. 14b with the configuration 0,0,1-1 sustained in average 222 GFLOPS on SNB-MUC and 648 GFLOPS on HSW-MUC. This transfers to 115 non-zero GFLOPS on SNB-MUC and 337 non-zero GFLOPS on HSW-MUC. Note that the hardware to non-zero operations ratio is slightly higher than in the final setting because of the all-dense configuration for the neighboring surface kernel.

SUMMARY

Ch. 7.1 introduced our low-level ADER-DG kernels. The next chapter, Ch. 7.2, formalized the involved matrix-matrix multiplication via BLAS-DGEMM identifiers and described our usage of the libxsmm-library. Before moving to the auto-tuning, Ch. 7.3 discussed the arithmetic intensities of our ADER-DG kernels in dense settings and in dependency of the convergence rate and assumed cache-hit ratios in the neighboring surface kernel. Afterwards, Ch. 7.4 introduced our auto-tuning approach for the sparse-dense decision and presented tuning results for SuperMUC-1 and SuperMUC-2. A distinction of hardware and non-zero operations finalized our discussions of Ch. 7.4.

This chapter completed the derivation of our computational core. After the description of the clustered LTS scheme in Ch. 3, we introduced step-by-step a high performance implementation of the scheme addressing all levels of parallelism. From a technical standpoint our core now features a streamlined memory layout. The memory layout supports hardware prefetching by respecting the order of execution imposed by our clustered LTS scheme, features arbitrary vector-alignment, and features in-place communication. Our parallelization uses OMP for shared memory parallelization and MPI's send-receive-model for distributed systems. Additional, a dedicated communication thread ensures message progression and thus asynchronous communication. At the lowest level we use the libxsmm-library to ensure vectorization of our ADER-DG kernels.

In the next part, Pt. iii, we systematically evaluate the performance of our computational core. This evaluation covers single-node and small-scale setups, weak and strong scaling scenarios of our computational core on all targeted supercomputers, and multiple, full-machine landmark simulations.

Part III

SIMULATIONS

This part systematically evaluates the algorithmic design decisions made in Pt. i and the engineering decisions made in Pt. ii.

We start by evaluating the single-node and small-scale performance of our computational core in Ch. 8.

Next, the presented weak scaling of Ch. 9 shows that our computational core is ready for more than 10^{12} degrees of freedom.

Finally, the last two chapters present sustained petascale performance of our computational core for production character simulations. Ch. 10 applies our computational core to large-scale, GTS dynamic rupture earthquake simulations. These simulations are followed by large-scale clustered LTS, seismic wave propagation runs in Ch. 11.

8 SINGLE NODE AND SMALL-SCALE

Our first scenario aims at evaluation of our computational core's single node and small-scale performance. For this purpose we simulate the SCEC wave propagation test model Layer Over Half-space (LOH.1) [20]. The authors of [23] and [24] use the LOH.1 benchmark to compare the quality of SeisSol's GTS and LTS solution to those of different Finite Difference packages and a Finite Element package. In typical setups the discretization of the LOH.1 benchmarks fits on a small number of nodes. Therefore it was used extensively in [8, 9, 10] to study the performance of our computational core in different versions.

8.1 LOH.1

Our LOH.1 setup uses the spatial domain $\Omega = [-15\,\text{km}, 15\,\text{km}] \times [-15\,\text{km}, 15\,\text{km}] \times [0, 17\,\text{km}]$. Fig. 16 shows our mesh consisting of 386,518 tetrahedrons. The mesh adaptively refines the 1 km thick layer of the LOH.1 benchmark and thus increases the spatio-temporal resolution at the seismic receivers located at the free-surface.

For the layer we use the homogeneous parameters $\rho = 2600\,\text{kg/m}^3$, $\lambda = 20.8\,\text{GPa}$ and $\mu = 10.4\,\text{GPa}$. The half-space has parameters $\rho = 2700\,\text{kg/m}^3$, $\lambda = 32.4\,\text{GPa}$ and $\mu = 32.4\,\text{GPa}$. We use free-surface boundary conditions for the top of the layer ($z = 0$) and absorbing boundary conditions for the remaining three surfaces of our domain Ω. The seismic source is a point dislocation located at $(0, 0, 2\,\text{km})$ [20].

8.2 GLOBAL TIME STEPPING

Following the considerations in Ch. 5.3, our computational cores requires a total ~6.4 GiB memory for the LOH.1 setup using a seventh order global time stepping scheme in double-precision. Therefore, the LOH.1 setup is ideal to test the single-node performance of our computational core.

The global time stepping component of our computational core is extensively studied in [10]. Here the performance for orders $\mathcal{O} = 2 - 7$ on four different architectures is discussed. In addition to the convergence rate, the authors also investigate the influence of the clock rate. Fig. 17 shows the measured performance of [10] for a dual-socket Intel Xeon X5690 server (WSM-ISC), a dual-socket Intel Xeon E5-2670 v3 server (SNB-ISC), a dual socket Intel Xeon E5-2699 v3 server with enabled cluster-on-die (HSW-ISC) and an In-

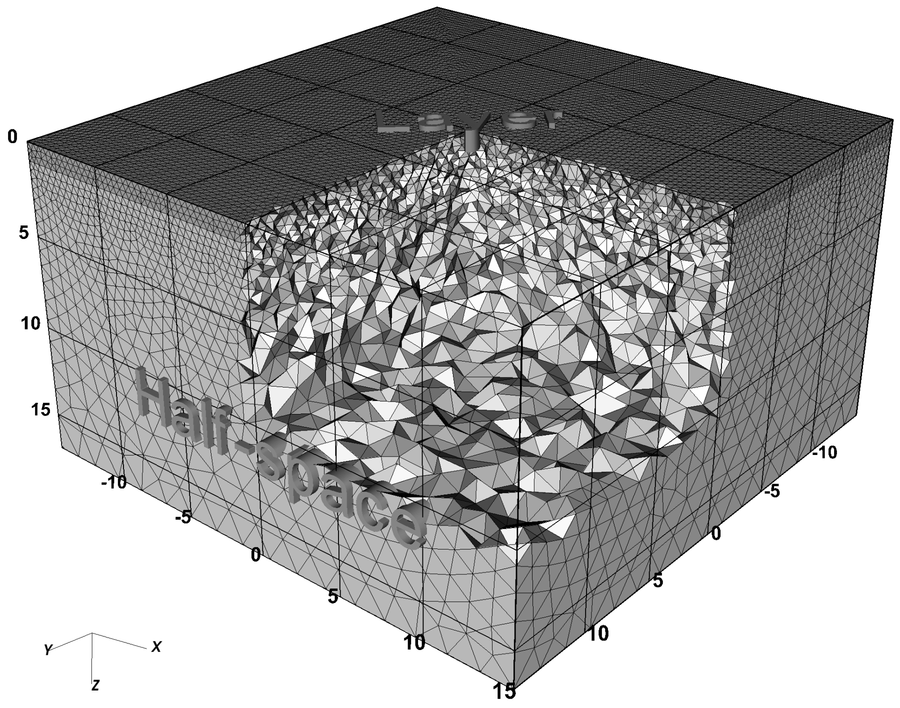


Figure 16: Illustration of the Layer Over Half-space (LOH.1) setup. Shown is the domain $\Omega = [-15\,\text{km}, 15\,\text{km}] \times [-15\,\text{km}, 15\,\text{km}] \times [0, 17\,\text{km}]$. The upper part of the domain is covered by the 1 km thick layer (dark gray) and the remainder by the half-space (gray). The structure of the mesh is illustrated by removing the elements in $[0, 15\,\text{km}] \times [0, 15\,\text{km}] \times [0, 10\,\text{km}]$.

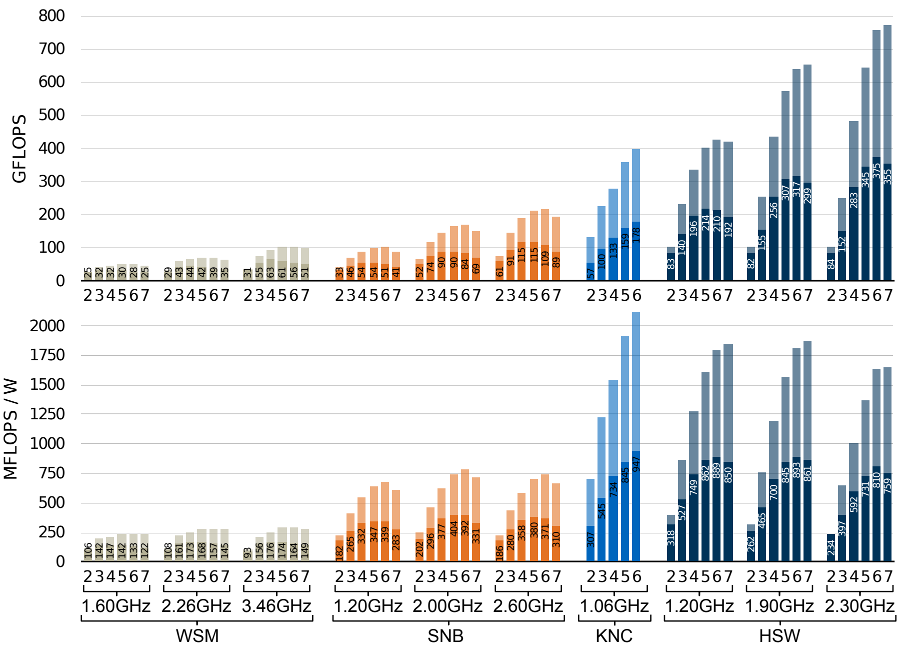


Figure 17: Single-node performance of the LOH.1 setup on four different architectures. Shown are the non-zero (dark colors) and hardware (light colors) MFLOPS/W and GFLOPS in dependency of the clock frequency and convergence rate. Source: [10].

tel Xeon Phi 5110P coprocessor (KNC-ISC). Analogue to our auto-tuning in Ch. 7.4, [10] interprets the results in terms of non-zero and hardware GFLOPS. Furthermore, [10] shows the power efficiency in terms of MFLOPS/W. The darker colors of the bars in Fig. 17 refer to non-zero operations, while the complete bars show hardware operations. Power measurements in [10] were performed on the AC-side using standard, air-cooled 2U rack-servers for WSM-ISC, SNB-ISC, and HSW-ISC. For KNC-ISC the Intel MIC System Management and Configuration application (micsmc) was used to isolate the power consumption of the coprocessor from the idling host.

The observed floating point performance in [10] underlines the general trend of decreasing machine balances. WSM-ISC is the oldest architecture in the mix with Q1'11 as launch date. Compared to the other systems WSM-ISC shows a flat performance profile across the orders of convergence. The profile is almost constant at 1.6 GHz and requires order $\mathcal{O} = 3 - 4$ for a high floating point throughput at 2.26 GHz and 3.46 GHz. 102 GFLOPS for fifth-order convergence at 3.46 GHz is the maximum sustained floating point performance for WSM-ISC. This corresponds to a peak efficiency of 61.4 % and a hardware to non-zero ratio of 1.67 in our setup. Additionally the high order simulations at 3.46 GHz have the highest energy efficiency. Thus the system benefits from the near-optimal frequency scaling, which reduces the relative energy consumption of non-CPU components.

SNB-ISC was launched in Q3'14 and doubles the theoretical peak performance of WSM-ISC with 333 GFLOPS. However in [10] the measured bandwidth of the STREAM triad benchmark [47] only increased by a factor of 1.8 to 75 GiB/s. Following Ch. 7.3, this lowers the machine balance by 9.7 % from 0.25 bytes/flop to 0.23 bytes/flop and thus requires a higher arithmetic intensity of our ADER-DG scheme to fully utilize the computational throughput of the machine. The measured floating point performance in Fig. 17 supports this theoretical result. In comparison to WSM-ISC, we observe a steepening performance profile. With 217 GFLOPs the highest hardware floating performance was measured for sixth order convergence at 2.6 GHz. This is equivalent to peak efficiency of 65 % and a hardware to non-zero ratio of 1.99. Energy-wise 2.0 GHz is the most efficient frequency for SNB-ISC.

The Intel Xeon Phi coprocessor KNC-ISC was launched in Q4'12 and has a theoretical peak performance of 1 TFLOPS. In [10] a bandwidth of 150 GiB/s was measured using the STREAM triad. Thus the floating point performance increased by 3.1 $\times$ compared to SNB-ISC, but only by 2 $\times$ in terms of memory bandwidth. This is equivalent to a machine balance of 0.14 bytes/flop and, as a result, the performance profile in Fig. 17 of KNC-ISC is steeper than the profile of SNB-ISC. Here, we require order $\mathcal{O} = 6$ to achieve maximum performance. Results for $\mathcal{O} = 7$ on KNC-ISC are not included in Fig. 17 due to memory limitations of the architecture. The maximum performance of 399 GFLOPS corresponds to a peak efficiency of 40 % and a hardware to non-zero ratio of 2.2. Power-wise KNC-ISC substantially more efficient than SNB-ISC and obtains the highest hardware power efficiency for all orders of convergence. Compared to the last architecture HSW-ISC this only holds in terms of non-zero operations for sixth order simulations. This is a result of the 512-bit vector instruction set, which requires higher hardware to non-zero ratios on KNC-ISC.

HSW-ISC is the last architecture in Fig. 17 and was launched in Q3'14. With 1.1 TFLOPS the theoretical peak performance of HSW-ISC is slightly higher at 1.9 GHz than the peak performance of KNC-ISC. However, the measured benchmark of the STREAM triad benchmark in [10] is only 105 GiB/s and thus 1.4 $\times$ less than that of KNC-ISC. As a result, HSW-ISC has the lowest machine balance with 0.1 bytes/flop. Note that HSW-ISC has a processor base frequency of 2.3 GHz, but guarantees full AVX2-performance only at 1.9 GHz to stay in TDP limits [37]. HSW-ISC has the lowest machine balance of all considered architectures and thus the steepest performance profile in Fig. 17. The maximum hardware performance is 773 GFLOPs at seventh order using 2.3 GHz with a hardware to non-zero ratio of 2.2. A peak efficiency of 60 % is reached considering the 651 GFLOPS at 1.9 GHz. Power-wise HSW-ISC is, together with KNC-ISC, the most efficient system. However, a clear descent is visible for all orders of convergence when going from 1.9 GHz to 2.3 GHz. Thus, from the perspective of energy consumption high-order simulations should run at the guaranteed frequency of 1.9 GHz. The low-order, memory bound simulations should run at the 1.2 GHz iso-frequency.

8.3 LOCAL TIME STEPPING

In Ch. 8.2 we discussed the cross-architecture global time stepping performance of our computational core on a single node or card. Now, we study the performance of our clustered LTS scheme for the LOH.1 setup. First, we discuss theoretical possible performance gains of LTS. The second step uses SuperMUC-2 to evaluate our scheme in practice. In addition to single node performance, we also study the strong scaling performance and thus the efficiency of our embedded distributed memory algorithm.

THEORETICAL PERFORMANCE To investigate the theoretical performance of LTS we first consider the per-element time steps imposed by the CFL-condition and second the resulting time steps of our clustering. Per-element time stepping builds the theoretical upper limit for the speedup of every LTS algorithm compared to global time stepping. In this case every element exactly updates with its CFL time step, but we assume GTS floating point performance for every update. Our clustered LTS scheme trades some of the theoretical, element-wise LTS speedup for homogeneity. Thus, our second performance characteristic is the theoretical speedup of our clustered LTS scheme assuming GTS floating point performance.

The ratio of the largest to the smallest CFL time step is 17.8 in our LOH.1 setup. Fig. 18 shows the non-uniform density of the time steps for the LOH.1 setup in $[\Delta t, 9\Delta t[$. Only very few elements have a time step in $[\Delta t, 2\Delta t[$. At $2\Delta t$ the density rapidly increases with two peaks at approximately $3\Delta t$ and $4\Delta t$. Afterwards, the density decreases again and is almost zero at $9\Delta t$. This indicates that very few elements have a time step larger than $9\Delta t$. The 25 % quartile (Q1) is at $3.2\Delta t$, the 50 % quartile (Q2) at $3.9\Delta t$ and the 75 % quartile (Q3) at $4.5\Delta t$. Taking all elements together, the upper limit for the LTS speedup is $4.0\,\times$.

Using a fixed rate of 2 in (65), we obtain a total of $L = 5$ clusters. The density of the first four clusters is shown as gray boxes in Fig. 18. The two clusters $C_2 = [2\Delta t, 4\Delta t[$ and $C_3 = [4\Delta t, 8\Delta t[$ have by far the highest density.

Fig. 19 depicts the spatial distribution of the clusters' elements with respect to the mesh shown in Fig. 16. Considering the spatial distribution of C_1's elements in Fig. 19a, we observe a scattered distribution in the vicinity of the material interface. This is the most critical cluster

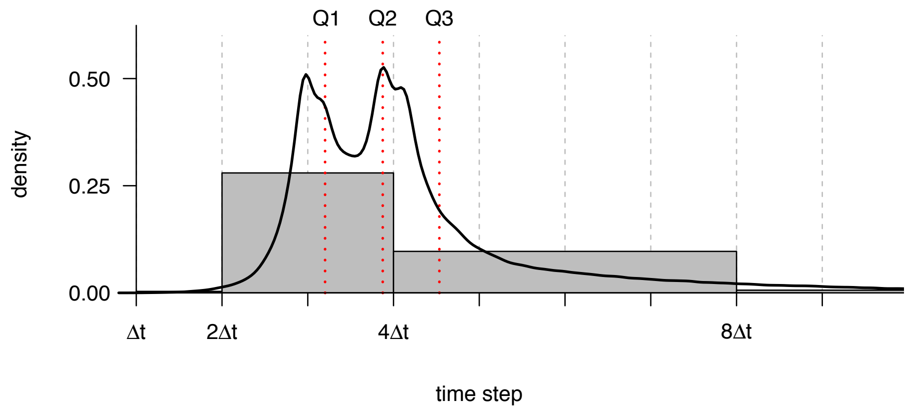


Figure 18: Density of time steps in the LOH.1 setup. The solid line shows the density in dependency of the elements' time steps. Red dotted lines show the location of the 25 % quartile (Q1), the 50 % quartile (Q2) and the 75 % quartile (Q3). Gray boxes represent a time step clustering with rate 2. The shown density is limited to elements with time steps in $[\Delta t, 9\Delta t[$.

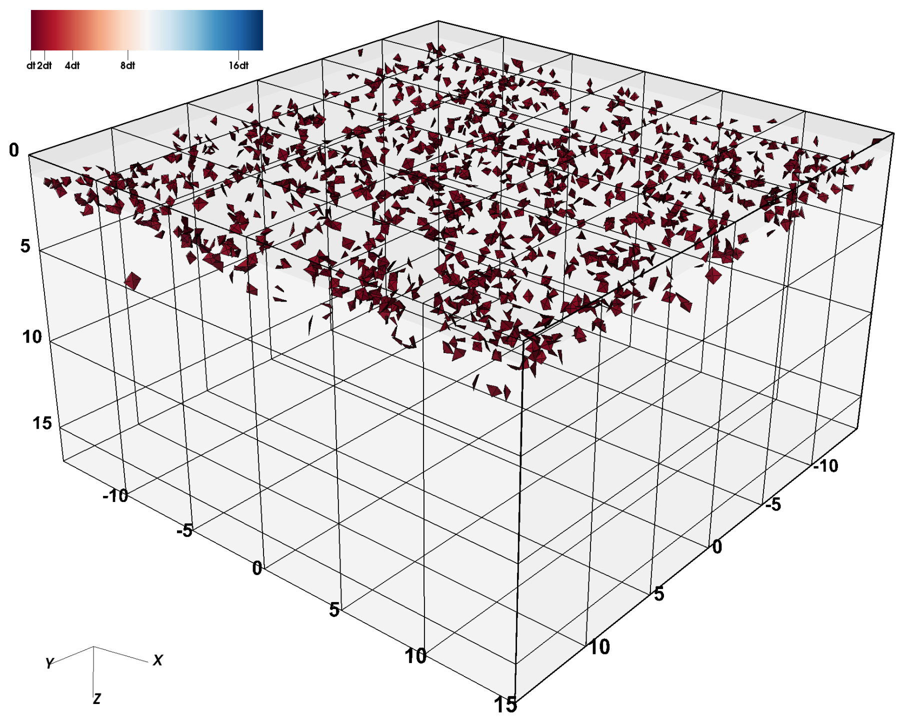
with the smallest time step and thus highest per-element load. Since we do not require connectivity of a cluster's elements, the flexibility of our clustered LTS scheme is a major advantage and minimizes the total number of elements in C_1. Taking the degenerated shape of the tetrahedrons into account, we mainly cover artefacts of the meshing process in C_1. Recapitulating our memory layout of Ch. 5.3, it is interesting to note that the element-local data of all scattered elements in Fig. 19a is stored linearly in memory.

Fig. 19b shows the second cluster $C_2 = [2\Delta t, 4\Delta t[$. Following the location of the 50 % quartile in Fig. 18 almost half of all elements are part of C_2. Most elements of C_2 are located densely below the interface at $z = 1\,\text{km}$. This is a result of our parameters in the half-space (see Ch. 8.1), which lead to $1.5\times$ faster P-wave velocities than in the layer. The remaining elements are loosely scattered in close proximity of the densely populated block. Similar to the elements of C_1, these elements are a result of coarsening in the meshing process.

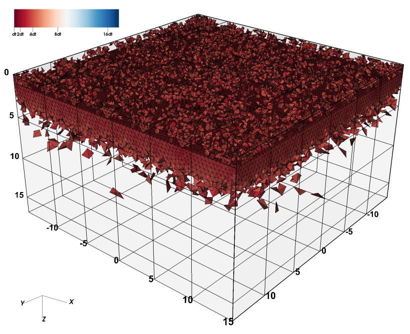
The second large cluster $C_3 = [4\Delta t, 8\Delta t[$ is shown in Fig. 19c. This cluster covers large parts of the layer and the remaining elements of the half-space up to a depth of approximately 10 km.

Cluster $C_4 = [8\Delta t, 16\Delta t[$ in Fig. 19d covers a major part of the computational domain. However, due to the adaptive mesh refinement and our clustered LTS scheme, the elements in C_4 are responsible for only a small fraction of the overall computational load. The last cluster $C_5 = [16\Delta t, 32\Delta t[$ in Fig. 19e contains only very few elements.

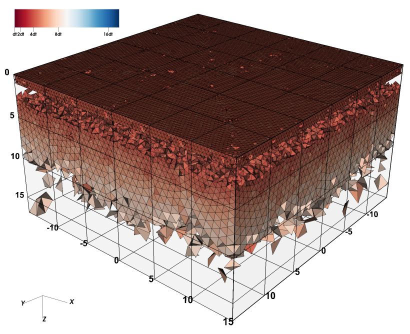
Fig. 20 shows the summed computational load of elements up to a certain time step in a theoretical, perfectly performing, per-element LTS algorithm. Here we assume that the cost of an element is reciprocal to its time step. Thus all possible, additional LTS overheads are neglected. We see that the first 25 % smallest time step elements (Q1)



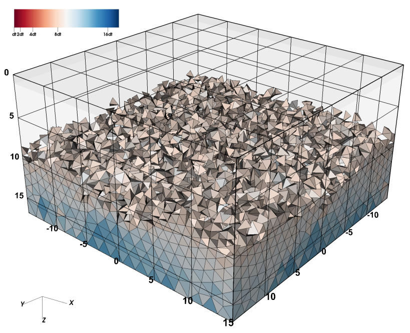
(a) $C_1 = [\Delta t, 2\Delta t[$.



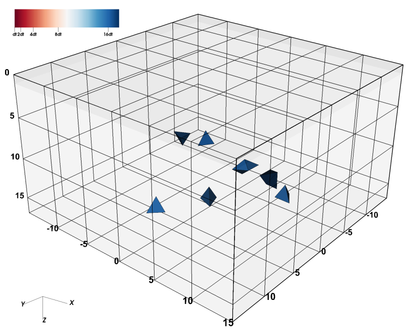
(b) $C_2 = [2\Delta t, 4\Delta t[$.



(c) $C_3 = [4\Delta t, 8\Delta t[$.



(d) $C_4 = [8\Delta t, 16\Delta t[$.



(e) $C_5 = [16\Delta t, 32\Delta t[$.

Figure 19: Spatial distribution of the clusters' elements for the LOH.1 setup and a rate-2 clustering. The location of the layer and the half-space is indicated by almost transparent gray boxes. The individual elements of the clusters are colored by their CFL time step.

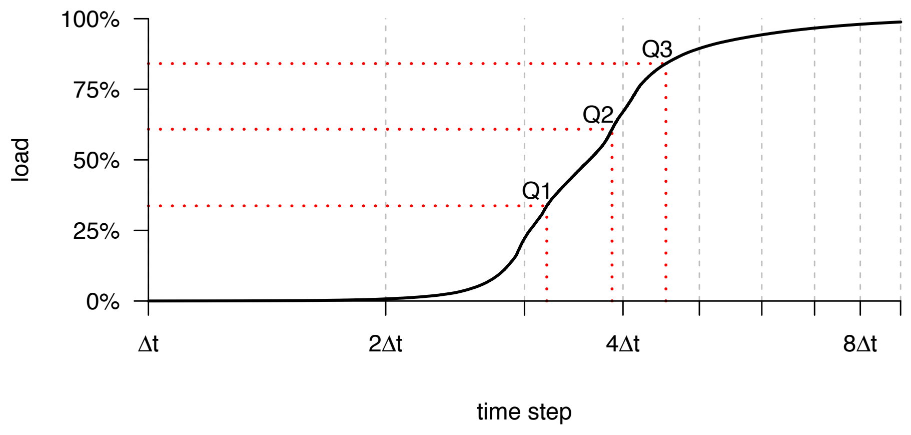


Figure 20: Distribution of the computational load in the LOH.1 setup assuming a theoretical, perfectly performing, per-element LTS algorithm. The solid line shows the summed load up in dependency of the time step. Red dotted lines show the location of the 25 % quartile (Q1), the 50 % quartile (Q2) and the 75 % quartile. The shown density is limited to time steps in $[\Delta t, 9\Delta t]$.

are responsible for 33.7 % of the overall load. The first 50 % (Q2) cover a load of 60.8 % and the first 75 % (Q3) a computational load of 84.1 %.

With respect to our clustered LTS scheme the maximum possible speedup for rate-2 clustering is 2.8 × and 2.3 × using fixed rates of 3. This is equivalent to 69.9 % of the maximum per-element speedup for rate-2 and 58.9 % of the maximum per-element speedup for rate-3 clustering.

STRONG SCALING Our study of the theoretical aspects shows the potential of local time stepping. Now, we discuss the observed performance of our clustered LTS scheme on SuperMUC-2 using sixth order of convergence. Our strong scaling covers 1-128 nodes and, similar to our auto-tuning on SuperMUC-2 in Ch. 7.4, uses the system with enabled cluster-on-die feature. Again we use HTT and explicitly pin two threads to each computational core. In contrast to the runs in Ch. 7.4, we leave the last core of every 28-core HSW-MUC node empty for our communication thread. For consistency this is also done in single-node runs, leaving 3.6 % of the node in busy waiting. Additionally, every run was performed at the guaranteed frequency of 2.2 GHz and the base frequency of 2.6 GHz.

Fig. 21 shows the strong scaling peak efficiencies of global time stepping (GTS), rate-2 (R2) and rate-3 (R3) clustered LTS for the LOH.1 setup. All runs were performed using sixth order convergence and double-precision arithmetic. For the calculation of the peak efficiencies we consider all 28 cores of HSW-MUC for the theoretical peak, including the communication core. Furthermore we assume that the

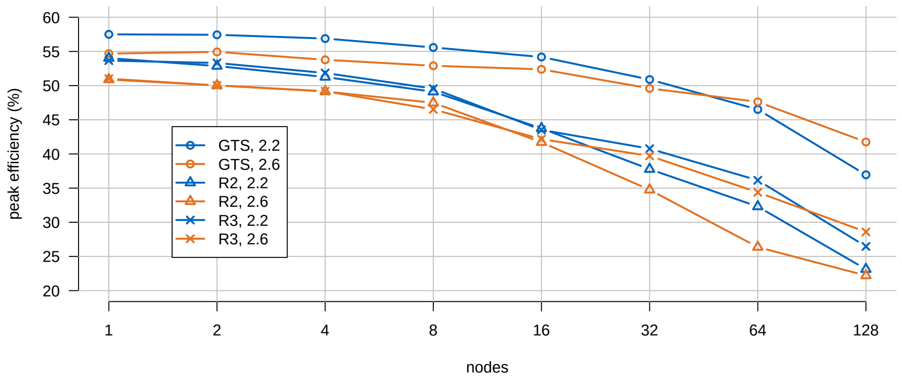


Figure 21: Strong scaling peak efficiencies of the computational core for the LOH.1 setup. Shown are the peak efficiencies of global time stepping (GTS), rate-2 clustered LTS (R2), and rate-3 clustered LTS (R3). All results are presented using HSW-MUC's guaranteed frequency of 2.2 GHz in blue and using the base frequency of 2.6 GHz in orange.

base frequency of 2.6 GHz is stable when fully utilizing the cores' floating point capabilities.

At the level of a single node our computational core achieves 57.5 % peak efficiency (567 GFLOPS) for GTS at 2.2 GHz and 54.7 % peak efficiency (637 GFLOPS) at 2.6 GHz. These results for HSW-MUC are in agreement with the results of Ch. 8.2 considering the higher core count (27 vs. 36) but lower frequencies of HSW-ISC.

The rate-2 clustering achieves 54.0 % peak efficiency at 2.2 GHz and 50.9 % peak efficiency at 2.6 GHz. Similar peak efficiencies of 53.7 % at 2.2 GHz and 51.0 % at 2.6 GHz are reached for rate-3 clustering on a single node in Fig. 21. In summary the single node results show that we are able to maintain the high GTS peak efficiencies of our computational core when using the LTS functionality. More importantly, we reduce the time-to-solution by a factor of 2.6 when using rate-2 clustering and by a factor of 2.2 for rate-3 clustering. Thus, in terms of Ch. 7.4, a hardware floating point operation of the LTS variants is more valuable than a non-zero operation of GTS.

Fig. 22 additionally shows the parallel efficiencies of the LOH.1 strong scaling setup. The strong scaling behavior of all time stepping variants in Fig. 22 is almost perfect up to eight nodes. Here GTS achieves a parallel efficiency of 96.7 % at 2.2 GHz and 2.6 GHz, while the LTS variants sustain between 90.9 % and 93.3 %.

Starting at 32 nodes the parallel efficiency drops for global time stepping due to the lowering amount of parallelism. 32 nodes using GTS are equivalent to approximately 447 elements per computational core and 159 MiB element-local data per node for our computational core.

The performance of local time stepping decreases noticeably starting at 16 nodes and thus earlier than GTS. This behavior is expected

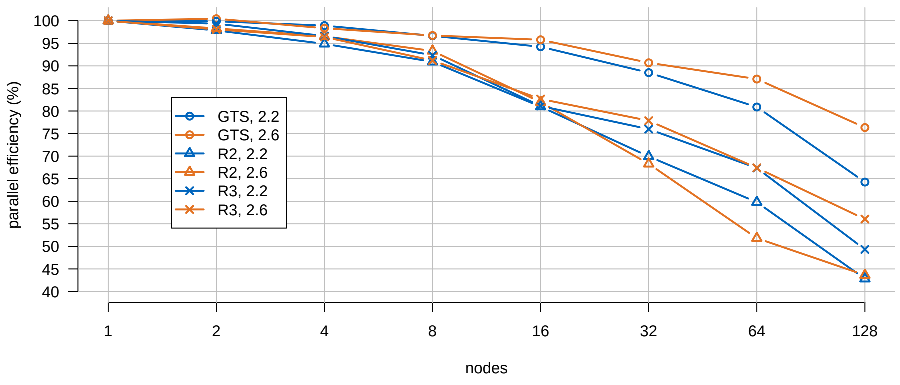


Figure 22: Strong scaling parallel efficiencies of the computational core for the LOH.1 setup. Shown are the parallel efficiencies of global time stepping (GTS), rate-2 clustered LTS (R2), and rate-3 clustered LTS (R3). All results are presented using HSW-MUC's guaranteed frequency of 2.2 GHz in blue and using the base frequency of 2.6 GHz in orange.

since our clustering reduces the amount of parallelism. Within a node the distribution of elements to clusters reduces the amount of elements available for our shared memory parallelization in Ch. 6.1. Further, in multi-node settings our load balancing leads to a higher variety in the element counts per node.

The maximum number of nodes in Fig. 21 and Fig. 22 is 128 and correlates to ~112 elements per core and a memory consumption of 40 MiB in GTS. Here, we still achieve a parallel efficiency of 64.3% for GTS at 2.2 GHz and 76.3 % at 2.6 GHz. R2 has a parallel efficiency of 42.8 % at 2.2 GHz and of 43.6 % at 2.6 GHz, while R3 reaches 49.3 % at 2.2 GHz and 56.0 % at 2.6 GHz. In this extreme case the speedups of the LTS variants reduce to 1.5 × - 1.7 ×.

SUMMARY

This chapter studied the single-node and small-scale performance of our computational core. For this purpose, we described the setup of the LOH.1 benchmark and studied the single-node, GTS performance of multiple Intel Xeon CPUs and the Intel Xeon Phi coprocessor. We observed high hardware peak efficiencies, above 60 % for the CPUs and around 40 % for the coprocessor, when increasing the order of convergence. Thus, the use of high orders is not only attractive from a perspective of mathematical convergence, but also efficient from a hardware viewpoint.

Following our single-node GTS performance-study, we investigated the theoretical possibilities of our clustered LTS scheme. We discussed the distribution of our elements to the possible time steps and the resulting load. Then we studied two different clustering strategies and derived theoretical peak speedups in comparison to GTS simulations.

Here, we identified an upper limit of 4.0 × for all possible LTS strategies and of 2.3 ×-2.8 × for our clustering strategies.

Our following GTS and LTS simulations on SuperMUC-2 underlined the efficiency of our algorithmic and engineering design decisions of Pt. i and Pt. ii. On a single node we were able to maintain the high GTS hardware peak efficiencies when using LTS and obtained efficiencies above 50 % for all configurations. These efficiencies translate to speedups between 2.2 × and 2.6 ×. Afterwards, we discussed GTS and LTS strong scaling results on up to 128 nodes of SuperMUC-2. Using eight nodes, we achieved an almost perfect strong scalability with parallel efficiencies above 90 % for all LTS and GTS configurations. Higher node counts resulted in decreasing parallel efficiencies due to the loss of concurrency.

The small scale results of this chapter build the baseline for the following large-scale setups in Ch. 9, Ch. 10 and Ch. 11. First, we drive the number of elements to the limits by performing a GTS weak scaling in Ch. 9. Next, Ch. 10 uses our computational core for a large-scale, multiphysics strong scaling and a full-machine production simulation of the 1992 Landers earthquake. Finally, Ch. 11 fully utilizes all of our computational core's functionality in a large-scale strong scaling. The final run of Ch. 11 performed a full-machine LTS simulation on SuperMUC-2 with sustained petascale performance under production conditions.

9 1+ TRILLION DEGREES OF FREEDOM

Before moving to production character simulations in the next chapters, we study the weak scaling behavior of our computational core in an artificial, cubic setup. This setup imitates the numerical convergence tests originally performed in [24]. The weak scaling setup was used extensively in [9, 31] to study the large-scale, GTS performance of our computational core in intermediate versions.

Additionally [10] studies the time- and energy-to-solution of the computational core on a single node or card by using the setup of [24] directly. Fig. 23 shows the numerical convergence of the cubic setup for uniform meshes. Here, we see the high order convergence of our computational core using single and double precision. In this setup our computational core hits machine precision between an error of 10^{-4} and 10^{-5} for single precision runs and at 10^{-11} for runs in double precision.

The use of high orders becomes especially appealing, when we consider the computational time required to reach a certain numerical error. Fig. 24 shows the L^{∞}-error for variable σ_{yz} and orders 2-7 in dependency of the runtime on HSW-ISC (see Ch. 8.2). In contrast to Fig. 23, these results include the effect of the time steps and the obtained hardware efficiencies. We see that every order of convergence lowers the error of the simulation by a magnitude in this benchmark. Thus, the high order simulations outperformed the low order runs in terms of hardware utilization and time-to-solution.

In the remainder of this chapter we start our weak scaling study by defining the cubic setup in Ch. 9.1. Next, Ch. 9.2 discusses the weak scaling performance obtained in [31] for an intermediate version of our computational core. Then we extend the combined studies of [9, 31, 10] with results on SuperMUC-2.

9.1 CUBE

We use sixth order of convergence and GTS in a cubic domain of extend $[-50m, 50m]^3$ for our weak scaling setup. The domain is discretized regularly by hexahedrons, each of which is subdivided into five tetrahedrons. Analogue to [24] we set up a sinusoidal P- and S-wave as initial condition and use homogeneous material parameters of $\rho = 1$, $\lambda = 1$ and $\mu = 1$. In contrast to [24] we use free-surface instead of periodic boundary conditions to ease the mesh generation process. In sum this allows us to fix the number of elements per node to 400,000 and perform a perfect load balancing for all studied node-

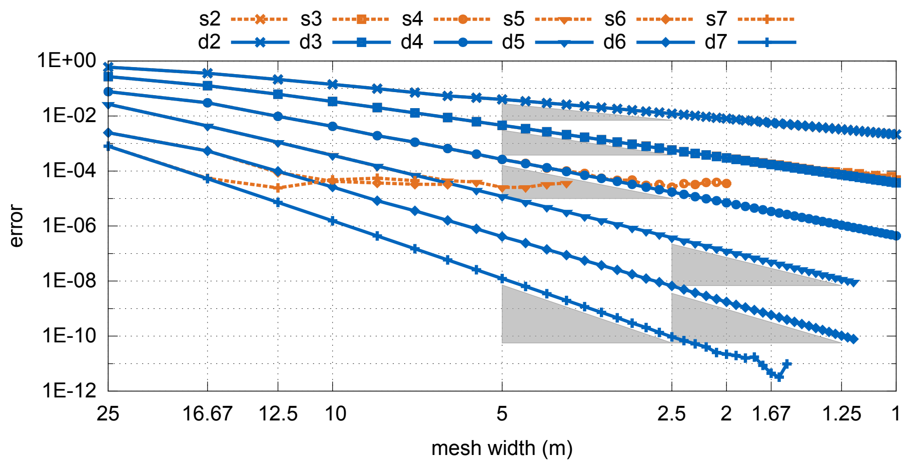


Figure 23: Convergence of the cubic setup for orders 2-7 using periodic boundary conditions. Shown is the L^∞-error for variable σ_{yz} in dependency of the cubic mesh width. Single-precision runs are colored in orange and double-precision runs in blue. The gray triangles illustrate the mathematical convergence rate. Source: [10].

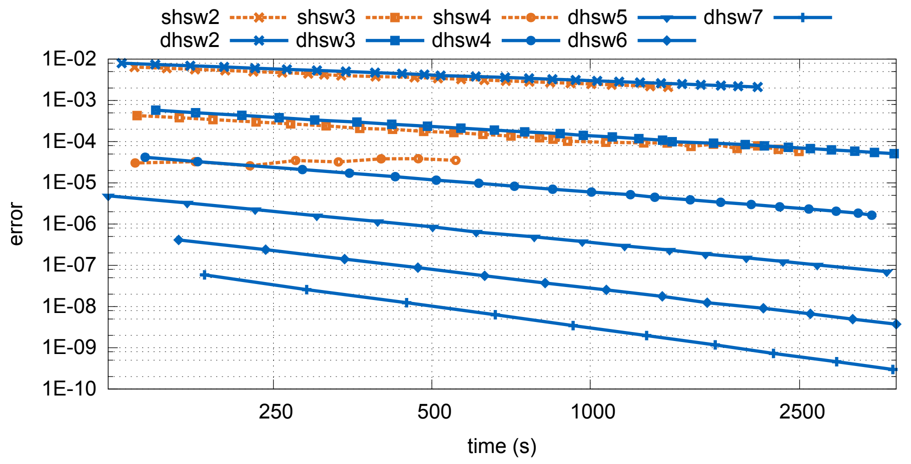
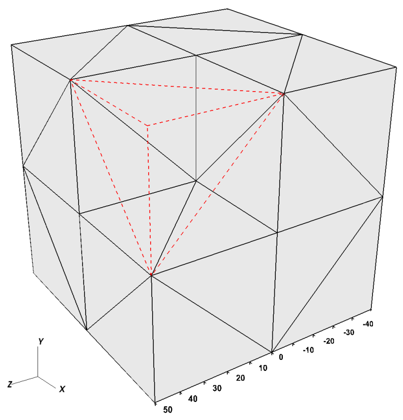
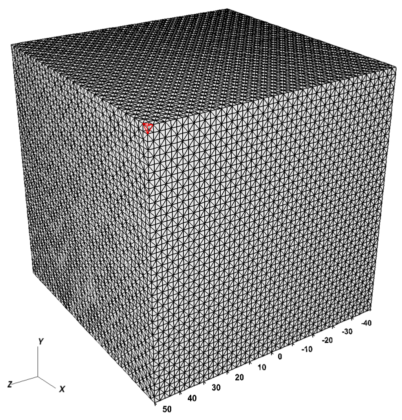


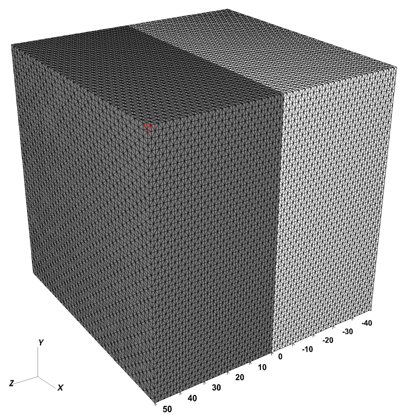
Figure 24: Time-to-solution of the cubic setup for orders 2-7 using periodic boundary conditions. Shown is the L^∞-error for variable σ_{yz} in dependency of the runtime. Single-precision runs are colored in orange and double-precision runs in blue.



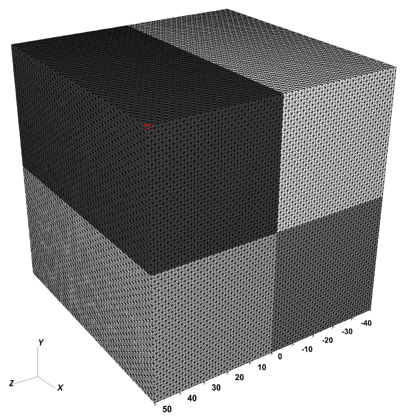
(a) $h_x = 50$, $h_y = 50$, $h_z = 50$.



(b) $h_x = 2.5$, $h_y = 2.5$, $h_z = 2$.



(c) $h_x = 2.5$, $h_y = 2$, $h_z = 1.25$.



(d) $h_x = 2$, $h_y = 1.25$, $h_z = 1.25$.

Figure 25: Four different meshes discretizing the cubic domain. The cubic domain is divided into hexahedral meshes with mesh width h_x, h_y and h_z. Each of the hexahedrons is further subdivided in five tetrahedrons. All edges of the hexahedron containing node $(50, 50, 50)$ are shown. The five of edges in the foreground are colored in red. Partitions are differentiated with shades of gray.

counts. Note that [31] used the same number of elements per node or card, while [9] used 960,000 elements per node.

Fig. 25 exemplary illustrates different meshes for the cubic domain. The first mesh in Fig. 25a is uniformly refined in all coordinate directions and contains a total of 40 tetrahedrons. All edges of the tetrahedrons in the cube touching node $(50, 50, 50)$ are shown. Such meshes are used in the convergence studies of [24] and [10].

The second mesh in Fig. 25b is used in our weak scaling study for a single node. Here, we have hexahedral mesh widths of $h_x = 2.5$, $h_y = 2.5$ and $h_z = 2$, thus a total of $40 \cdot 40 \cdot 50 \cdot 5 = 400,000$ elements.

We use the mesh illustrated in Fig. 25c for two nodes in our weak scaling. The mesh has a grid spacing of $h_x = 2.5$, $h_y = 2$ and $h_z = 1.25$. This is equivalent to $40 \cdot 50 \cdot 80 \cdot 5 = 160,000$ elements. We partition the mesh at $z = 0$ and obtain the two partitions illustrated as shades of gray in Fig. 25c.

The last mesh illustrated in Fig. 25d has a spacing of $h_x = 2$, $h_y = 1.25$ and $h_z = 1.25$ and leads to 1,600,000 elements. Together with partitioning along $y = 0$ and $z = 0$ we reach our target number of elements for four nodes.

9.2 WEAK SCALING

INTERMEDIATE VERSION The authors of [31] used an intermediate version of our computational core for a weak scaling study with the setup of Ch. 9.1. This version was missing some features our final computational core heavily relies on. First of all the version in [31] featured global time stepping only, while the final version simply uses our clustered LTS scheme in a global time stepping fashion for GTS setups.

Furthermore, the homogeneous SuperMUC-1 simulations in [31] utilized separate MPI copy buffers to which the data was copied before a blocking synchronization phase. Here, our final version communicates in-place and thus does not require to collect data before issuing communication. Also, the final version heavily utilizes asynchronous communication with a dedicated communication thread for MPI progression. The asynchronous layout is especially important for LTS simulations.

However, for the Intel Xeon Phi accelerated systems Stampede and Tianhe-2 the version of [31] featured a heterogeneous offload scheme. This scheme handles communication between the Host CPUs and Intel Xeon Phi coprocessors with the Intel Language Extensions for Offload (LEO). Additionally MPI communication between the Host CPUs is overlapped with computations on the Intel Xeon Phi coprocessors. From this viewpoint, the overlapping communication and computation of [31] is similar to our communication thread, however requires the transfers of MPI data from and to the coprocessors.

Fig. 26 shows the measured peak efficiencies and Fig. 27 the measured parallel efficiencies of [31]. Both figures show results for "SuperMUC, classic", the initial version of SeisSol running on SuperMUC-1. The time marching scheme of this version was completely replaced by our new computational core. The other curves show the performance of our computational core in the intermediate version of [31] running on SuperMUC-1 ("SuperMUC, gr. buff"), Stampede and Tianhe-2.

The hardware and non-zero peak efficiencies of the initial version "SuperMUC, classic" stagnates at 5 % in Fig. 26. The efficiencies of the new computational core in Fig. 26 are high in general but slightly lower than the more recent results for the LOH.1 benchmark, discussed in Ch. 8.2. Note that the results in Fig. 26 include the perfor-

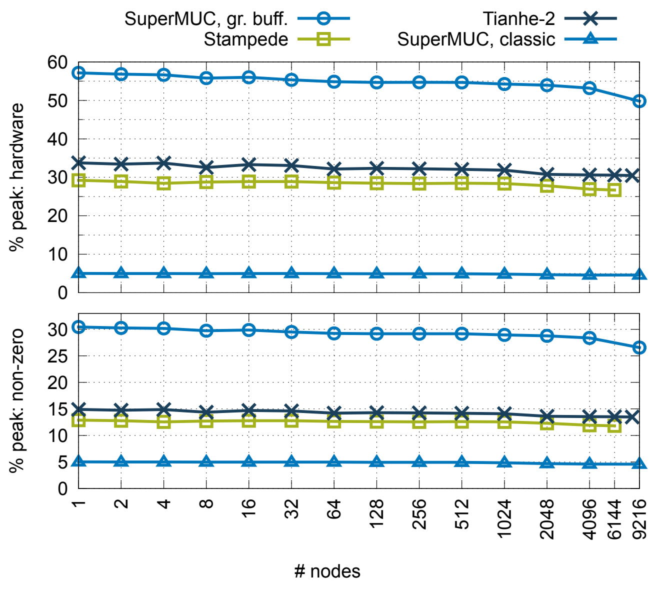


Figure 26: Weak scaling peak efficiencies using different versions of SeisSol for the cubic setup. Shown are the peak efficiencies for the initial version of SeisSol on SuperMUC-1 ("SuperMUC, classic") and an intermediate version of our new computational core on SuperMUC-1 ("SuperMUC, gr. buff"), Stampede and Tianhe-2. Source: [31].

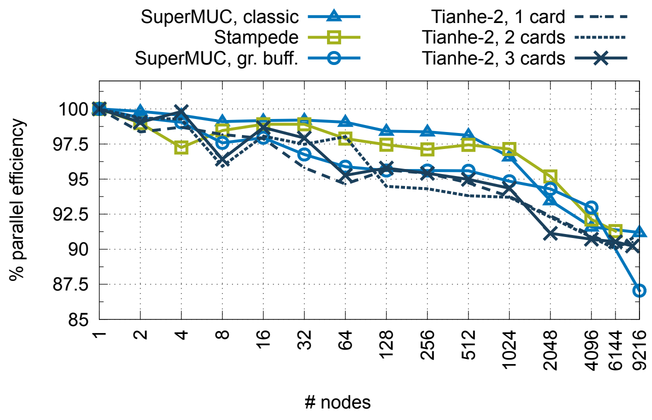


Figure 27: Weak scaling parallel efficiencies using different versions of SeisSol for the cubic setup. Shown are the parallel efficiencies for the initial version of SeisSol on SuperMUC-1 ("SuperMUC, classic") and an intermediate version of our new computational core on SuperMUC-1 ("SuperMUC, gr. buff"), Stampede and Tianhe-2. Source: [31].

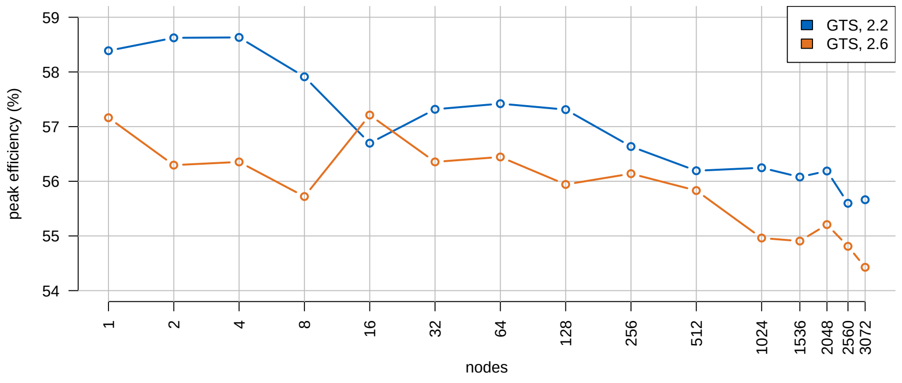


Figure 28: Weak scaling peak efficiencies of the final computational core for the cubic setup. Shown are the peak efficiencies of global time stepping using HSW-MUC's guaranteed frequency of 2.2 GHz in blue and using the base frequency of 2.6 GHz in orange.

mance of the host processors in the calculation of the peak efficiencies for Stampede and Tianhe-2. Since the heterogeneous offload scheme of [31] schedules the computationally heavy seismic wave propagation completely to the coprocessors, the host processors perform communication only.

At this point the largest homogeneous simulation reached 1.6 hardware PFLOPS using 9,216 SuperMUC-1 nodes. The largest heterogeneous simulation on Stampede sustained 2.3 PFLOPS using 6,144 nodes and 8.6 PFLOPS on Tianhe-2 using 8,192 nodes.

The parallel efficiencies for all machines in Fig. 27 are almost perfect. While the accelerated supercomputers Stampede and Tianhe-2 have efficiencies above 90 % for all core-counts, our computational core in the version of [31] drops below 87.5 % on 9216 nodes. This is a result of the blocking communication and in agreement with the higher parallel efficiency of "SuperMUC, classic" due to the lower peak efficiency of the initial SeisSol version.

FINAL VERSION Considering our computational core in its final version for GTS, we expect to mainly benefit from the overlapping communication and computation, and the high node level performance on HSW-MUC. In this context Fig. 28 shows the hardware peak efficiencies and Fig. 29 the parallel efficiencies on SuperMUC-2 for our weak scaling setup.

Again both figures show results for HSW-MUC's guaranteed frequency of 2.2 GHz and the base frequency of 2.6 GHz, but assume perfect frequency scaling for the calculation of the theoretical peak at 2.6 GHz. The peak efficiencies of all runs are high and above 54 % for all node counts, including the full-machine runs using 3,072 nodes. Comparing these results to those of the intermediate version in Fig. 26, we see that our peak efficiencies on SuperMUC-2 are similar to the previously observed SuperMUC-1 efficiencies. This behavior

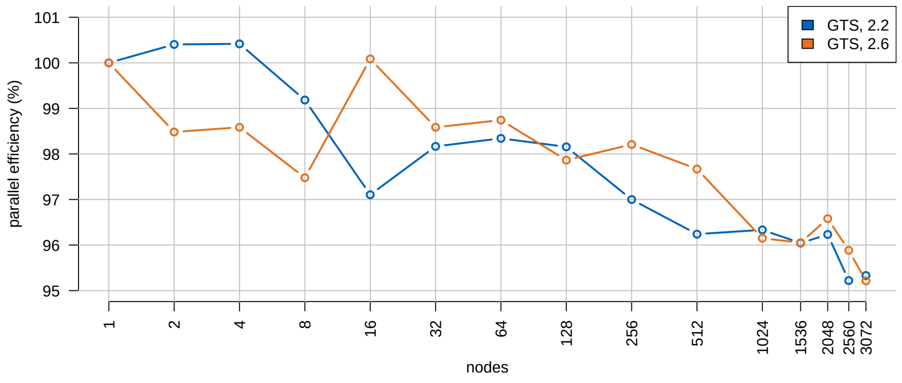


Figure 29: Weak scaling parallel efficiencies of the final computational core for the cubic setup. Shown are the parallel efficiencies of global time stepping using HSW-MUC's guaranteed frequency of 2.2 GHz in blue and using the base frequency of 2.6 GHz in orange.

is expected since the authors of [10] reported similar efficiencies on both architectures (see Fig. 17).

The full-machine size run at 2.2 GHz sustained 1.69 PFLOPS in hardware, while the run at 2.6 GHz sustained 1.95 PFLOPS. This is equivalent to a peak efficiency of 55.7 % at 2.2 GHz and of 54.4 % at 2.6 GHz. The high hardware utilization becomes even more imminent when considering SuperMUC-2's reported HPL performance of 2.81 PFLOPS. Here, our computational core reaches 69 % of the reported performance.

With respect to the parallel efficiencies in Fig. 29, we observe efficiencies above 95 % for all runs. A slight decrease of the parallel efficiency, despite our fully overlapping communication, is expected at high node counts. The reason are the performance variations of the nodes, where the worst performing node dominates our final performance. We already observed a similar behavior when running our auto-tuning in Ch. 7.4 on 50 different nodes.

We can now compare the homogeneous simulations of the intermediate computational core on SuperMUC-1 with our final version on SuperMUC-2. Here, we are able to increase the parallel efficiency by ~2.5 % on a 4,096 node basis and by 7.5 % on a 9,216 node basis of SuperMUC-1 compared to all 3,072 nodes of SuperMUC-2. A comparison of 3,072 SuperMUC-2 nodes with 9,216 nodes of SuperMUC-1 seems to be the fairest though, since both systems report an almost identical HPL-performance.

SUMMARY

This chapter presented a weak scaling study of our computational core on the supercomputers SuperMUC-1, SuperMUC-2, Stampede and Tianhe-2. We used 400,000 elements per node or card and observed almost perfect scaling behavior from a single node to half- or full-machine size runs. The sustained parallel efficiencies exceeded 90 % in heterogeneous and homogeneous configurations and corresponds to multi-PFLOP simulations on the Intel Xeon Phi accelerated machines. We obtained a maximum performance of 8.6 PFLOPS on Tianhe-2 using 8,192 nodes, which is equivalent to the use of 24,576 coprocessors or 1.4 Mio. cores. In this configuration 5.0×10^{12} degrees of freedom discretized our solution.

The runs on SuperMUC-2 showed the efficiency of our asynchronous, hybrid parallelization with parallel efficiencies above 95 % using the entire machine. Here, the highest node count of 3,072 nodes reached 1.95 PFLOPS in hardware or equivalently 69 % of SuperMUC-2's reported HPL-performance.

While this chapter showed the large-scale performance of our computational core using an artificial setup, a large-scale performance evaluation of simulations with production character is still outstanding. For this purpose we study the strong scaling behavior of a simulation of the 1992 Landers earthquake in Ch. 10 and of seismic wave propagation in Mount Merapi in Ch. 11.

10 PETASCALE MULTIPHYSICS

This chapter discusses a large-scale multiphysics setup of the 1992 Landers earthquake. The magnitude 7.3 earthquake occurred on June 28, 1992 near the town of Landers in California.

A rupture of multiple faults with a total rupture length of 85 km was reported. The average slip was between 3-4 m with a maximum slip of 6 m.[1] The 1992 Landers earthquake has a maximum Modified Mercalli Intensity of IX and resulted in 3 fatalities and more than 400 injured people.[2]

The large displacements caused by the 1992 Landers earthquake are still visible. Fig. 30 was taken more than 23 years after the earthquake and shows right-lateral offset of Linn Road in Landers.

In Ch. 10.1 we discuss the setup of our 1992 Landers scenario. Ch. 10.2 studies strong scaling of the setup on SuperMUC-1, SuperMUC-2, Stampede and Tianhe-2. As in the previous chapters, we discuss results of our computational core in an intermediate version before moving to the final version on SuperMUC-2. Finally, Ch. 10.3 studies two production runs of our setup on SuperMUC-1 and SuperMUC-2 respectively.

10.1 1992 LANDERS

The 1992 Landers setup discretizes the spatial domain with a total of 191,098,540 tetrahedrons. Adaptive mesh refinement is used to increase the resolution of the surface topography and of the fault system. 220,982 of the tetrahedral faces are aligned to the fault system inside the computational domain. For these faces dynamic rupture physics are solved. Effectively we replace our Riemann solver of Ch. 2.9 with a formulation introducing an artificial Godunov state satisfying a certain friction law [49].

An illustration of the tetrahedral mesh in conjunction with the fault system is given in Fig. 31. The Johnson Valley fault is shown in the foreground. We can see the increased resolution of the tetrahedral mesh in proximity of the fault system and the surface. Boundary conditions are free-surface at the surface and outflow everywhere else.

Material parameters in the domain are discretized using a one-dimensional, layered velocity profile. This velocity profile leads to

1 http://web.archive.org/web/20150830122227/http://scedc.caltech.edu/significant/landers1992.html

2 http://web.archive.org/web/20150830121958/http://earthquake.usgs.gov/earthquakes/states/events/1992_06_28.php

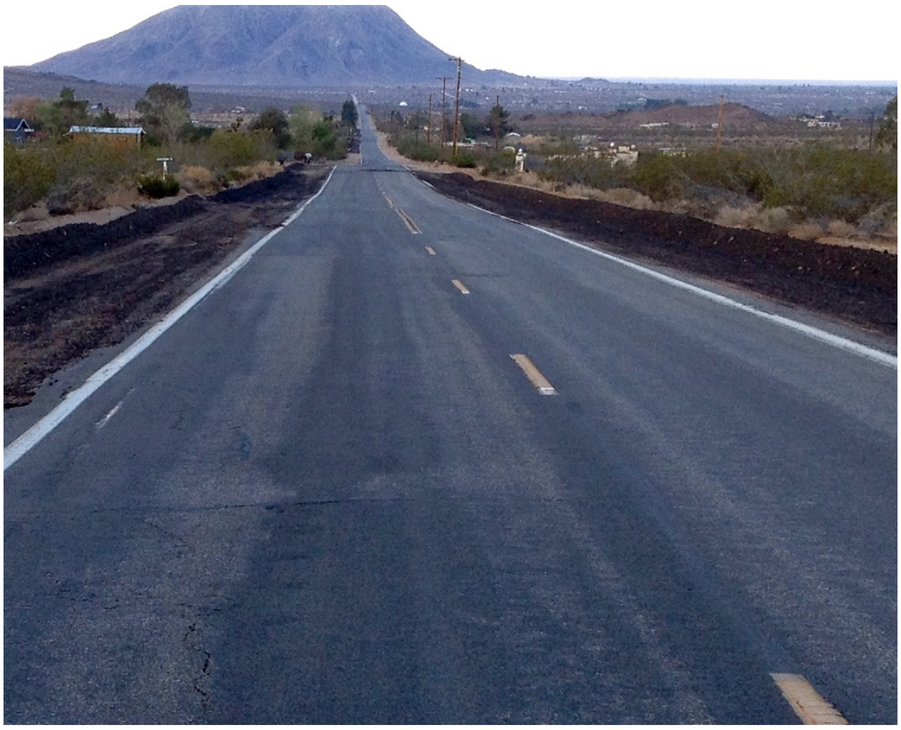


Figure 30: Right-lateral offset of Linn Road, Landers CA 92284, USA caused by the 1992 Landers earthquake. 23 years after the earthquake the offset is still visible. The picture was taken on July 5, 2015 looking eastward at 34.2954°, -116.4554°.

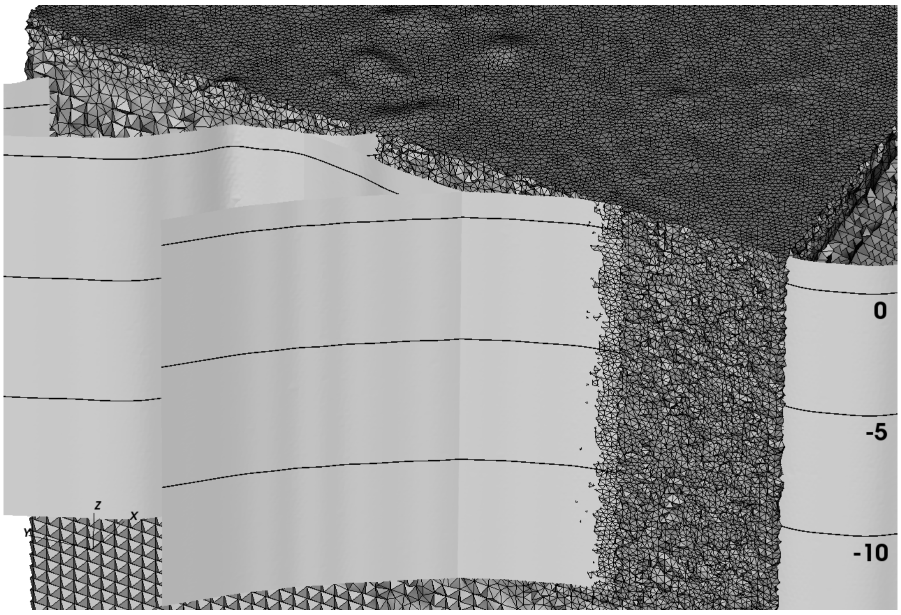


Figure 31: Illustration of the 1992 Landers setup. Shown is a part of the computational domain including the fault system and tetrahedral mesh. The mesh is cut along $x = 0\,\text{km}$, $y = -2\text{km}$ and $y = 25\,\text{km}$. On the faults depths of 0 km, -5 km and -10 km are given as contour lines.

gradually increasing wave speeds with increasing depth. For example our setup uses parameters $\rho = 2{,}300\,\mathrm{kg/m^3}$, $\lambda \approx 5.0\,\mathrm{GPa}$ and $\mu \approx 1.8\,\mathrm{GPa}$ up to a depth of 0.1 km. In the layer between 3-6 km depth the parameters are $\rho = 2{,}700\,\mathrm{kg/m^3}$, $\lambda \approx 27.9\,\mathrm{GPa}$ and $\mu \approx 21.2\,\mathrm{GPa}$. Following (17) this corresponds to an increase of $3.2\times$ in S-wave velocity and $2.64\times$ in P-Wave velocity.

The setups uses global time stepping and sixth order for the seismic wave propagation component. For the dynamic rupture computations a single quadrature point in time is used and multiple quadrature points in space are used [31]. Note that our computational core supports dynamic rupture physics only in GTS execution. While our considerations for the LTS wave propagation component (see Ch. 3) directly translate to dynamic rupture elements, extensive benchmarking is required to verify local time stepping in dynamic rupture workloads. Here, one can either decide to follow the LTS approach of our scheme and perform a minimal impact normalization only. Other options could enforce only neighboring dynamic rupture elements to have the same time step or enforce the same time step for all elements with dynamic rupture faces.

Analogue to Ch. 8.2, we are able to derive a total memory consumption of 2.4 TiB for our computational core using the given setup.

10.2 STRONG SCALING

INTERMEDIATE VERSION The authors of [31] use the 1992 Landers setup to study the strong scaling behavior of our computational core in an intermediate version. Analogue to the results discussed in Ch. 9, this study was carried out on the homogeneous supercomputer SuperMUC-1 and the two Intel Xeon Phi accelerated systems Stampede and Tianhe-2.

Fig. 32 shows the obtained peak efficiencies and Fig. 33 the obtained parallel efficiencies in dependency of the number of nodes. With respect to the weak scaling peak efficiencies (see Fig. 28), the authors of [31] are able to maintain the high efficiencies for all supercomputers on a 512 node basis. A slight decrease is expected due to the non-optimal load balancing in favor of minimized edge-cuts. At this point, our core consumes ~4.8 GiB per node for 512 nodes of SuperMUC and per card for 512 nodes of Stampede. The three cards per node in the Tianhe-2 system correspond to a consumption of ~1.6 GiB per card on a 512 node basis.

For higher node counts [31] observes decreasing parallel efficiencies as shown in Fig. 33. In contrast to the weak scaling results (see Fig. 27), the decrease is now highly dependent on the machine.

The parallel efficiencies of the homogeneous runs on SuperMUC-1 now drop faster than those in the weak scaling, but stay above 75 % for all node-counts. Here the relative amount of time spent in the

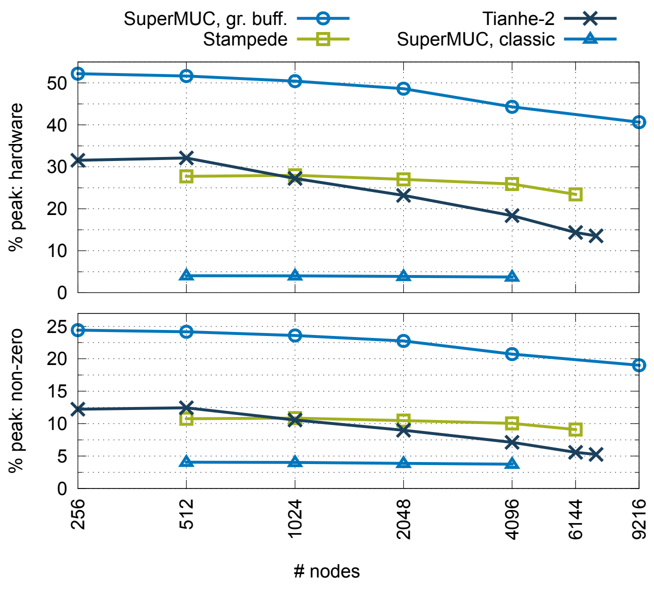


Figure 32: Strong scaling peak efficiencies using different versions of SeisSol for the 1992 Landers setup. Shown are the peak efficiencies for the initial version of SeisSol on SuperMUC-1 ("SuperMUC, classic") and an intermediate version of our new computational core on SuperMUC-1 ("SuperMUC, gr. buff"), Stampede and Tianhe-2. Source: [31].

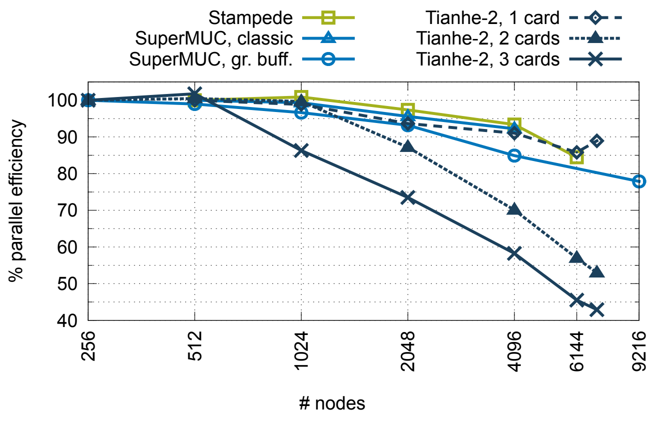


Figure 33: Strong scaling parallel efficiencies using different versions of SeisSol for the 1992 Landers setup. Shown are the parallel efficiencies for the initial version of SeisSol on SuperMUC-1 ("SuperMUC, classic") and an intermediate version of our new computational core on SuperMUC-1 ("SuperMUC, gr. buff"), Stampede and Tianhe-2. Source: [31].

blocking communication increases due to the lowering load per node. Further, the more expensive dynamic rupture computations are neglected in the partitioning of [31]. While the overall ratio of dynamic rupture evaluations to computations spend in the seismic wave propagation stays constant, the spatially local partitioning increases this ratio in the worst-case. As in the weak scaling case (see Fig. 27), the initial version of SeisSol sustains in [31] a higher parallel efficiency on 4,096 nodes of SuperMUC-1 than our computational core. This is again a result of the initial version's low peak performance.

Stampede scales nearly perfectly with a parallel efficiency of 84.5 % on 6,144 nodes. Compared to the homogeneous runs on SuperMUC-1, we have to consider the superior overlapping communication and computation of the heterogeneous offload scheme. Additionally, the authors of [31] overlap the seismic wave propagation component on the Intel Xeon Phi cards with dynamic rupture computations on the Host CPUs.

We observe the steepest decrease in Fig. 33 for Tianhe-2 using three Intel Xeon Phi cards per node. As discussed in [31], this is a result of the exposed MPI communication. Here, the high floating point performance of the nodes in combination with the halved bandwidth of Tianhe-2's network at the time of the runs are the reason [31]. This behavior changes when using less cards per node and thus increasing the bandwidth to floating point performance ratio. In fact, the single card runs on Tianhe-2 maintain above 85 % of parallel efficiency on 7,000 nodes.

Tianhe-2 achieved the highest performance of our computational core for the 1992 Landers strong scaling in [31]. With a parallel efficiency of 42.9 % the authors of [31] sustained 3.3 PFLOPS on 7,000 Tianhe-2 nodes. Stampede reaches 2.0 PFLOPS on 6,144 nodes. 1.3 PFLOPS with a parallel efficiency of 77.9 % are reached on the homogeneous SuperMUC-1 in the version of [31].

FINAL VERSION Equivalent to the weak scaling setup discussed in Ch. 9, we expect that the final version of our computational core outperforms the intermediate version of [31] in homogeneous runs. Again, we expect to mainly benefit from our asynchronous communication and the high peak efficiencies obtained for HSW-MUC.

Fig. 34 shows the peak efficiencies and Fig. 35 the parallel efficiencies of the final version on SuperMUC-2. Again, all results include SuperMUC-2's guaranteed frequency of 2.2 GHz and the base frequency of 2.6 GHz. The measured peak efficiencies for 128 nodes nearly maintain the weak scaling performance (see Fig. 28). Also in agreement with the weak scaling, the runs at 2.2 GHz reached a higher ratio of the theoretical peak performance. Since we use the full frequency of 2.6 GHz for the calculation of the peak performance, this behavior is expected.

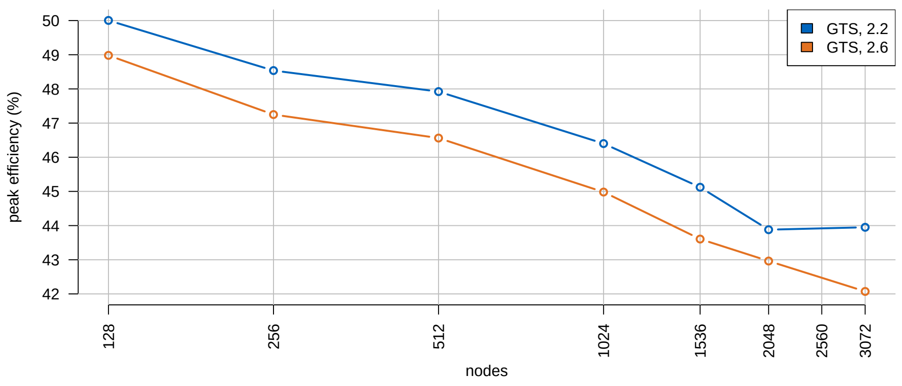


Figure 34: Strong scaling peak efficiencies of the final computational core for the 1992 Landers setup. Shown are the peak efficiencies of global time stepping using HSW-MUC's guaranteed frequency of 2.2 GHz in blue and using the base frequency of 2.6 GHz in orange.

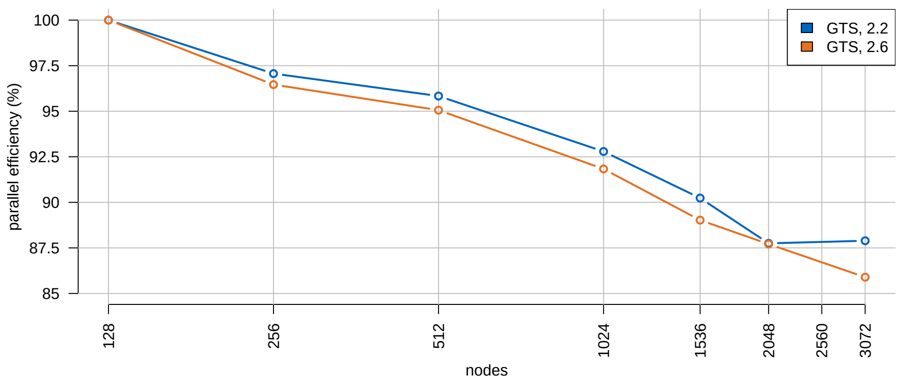


Figure 35: Strong scaling parallel efficiencies of the final computational core for the 1992 Landers setup. Shown are the parallel efficiencies of global time stepping using HSW-MUC's guaranteed frequency of 2.2 GHz in blue and using the base frequency of 2.6 GHz in orange.

Our computational core's final version is able to maintain a parallel efficiency of more than 85 %, even when scaling out to all 3,072 nodes of SuperMUC-2. SuperMUC-2 sustained 1.3 hardware PFLOPS with a parallel efficiency of 87.9 % using 3,072 nodes at 2.2 GHz. The run at 2.6 GHz reached 1.5 PFLOPS in hardware with a parallel efficiency of 85.9 %.

The peak efficiency of 3,072 nodes at 2.6 GHz is 42.1 %. This is equivalent to 53.3 % of SuperMUC-2's reported HPL-performance for our computational core in its final version. However, these results include strong scaling from 128 nodes to 3,072 nodes. This is an increase by 24 $\times$ in computational power with a memory requirement of ~821 MiB per node for our computational core's data in the largest configuration.

10.3 PRODUCTION

INTERMEDIATE VERSION The authors of [31] present results of a production run using the 1992 Landers setup (see Ch. 10.1). This run was carried out on all 9,216 nodes of SuperMUC-1 and reached a total simulated time of 42 seconds corresponding to 234,567 time steps. Analogue to the strong scaling in Ch. 10.2, the authors used our computational core in the intermediate version of [31]. For a total runtime of 7 hours and 15 minutes the authors sustained a performance of 1.25 PFLOPS in hardware. Fig. 36 shows the complex rupture behavior presented in [31]. Important features such as rupture branching or rupture jumps were observed and prove the value of multiphysics dynamic rupture earthquake simulations.

FINAL VERSION Since the output of the production run in [31] was limited to the fault system and seismic receivers, we discuss a repeated simulation on SuperMUC-2. In this simulation all 3,072 nodes of SuperMUC-2 were used at 2.6 GHz to run the final version of our computational core. This run additionally wrote the wave field containing the nine elastic variables to SuperMUC-2's GPFS Storage Servers (GSS). The overall amount of output was reduced by writing the values for the first, constant basis only. The run wrote the wave field output every 0.25 s of simulated time, resulting in a total of amount of 2.0 TiB. Additionally every 1.5 s the simulation wrote all data of our computational core to the GSS for the case of a hardware failure.

Since the simulation was carried out in an early access phase of SuperMUC-2, it had to be continued from checkpoints multiple times. The duration of the longest non-stop computation was 1 hour and 52 minutes. During this time our computational core advanced 11.75 s in simulated time and sustained 1.4 PFLOPS in hardware including output and checkpointing. Here, our computational core achieved 94.3 %

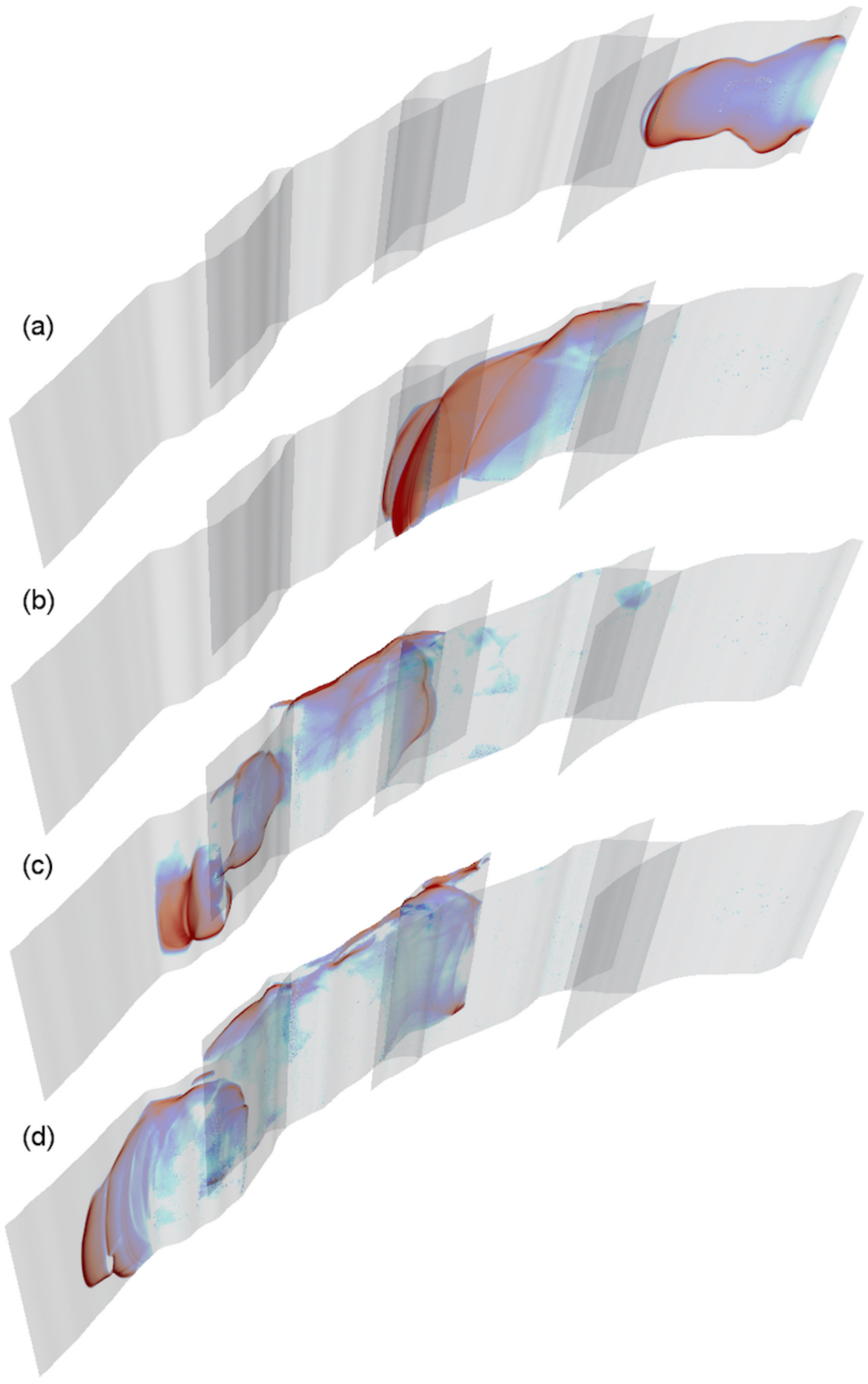


Figure 36: Dynamic rupture propagation on the fault system of the 1992 Landers setup. Shown is the slip rate throughout the simulation. Complex rupture propagation, including multiple rupture fronts, is visible. Source: [31].

of the strong scaling performance, discussed in Ch. 10.2. In summary, this repeated production on SuperMUC-2 not only enhanced but also outperformed the SuperMUC-1 run reported in [31].

Fig. 37 and Fig. 38 illustrate the obtained wave field output of the repeated production run. Fig. 37a shows the wave field after 9.25 s, Fig. 37b after 12 s, Fig. 38a after 19.5 s and Fig. 38b after 25 s. All figures show contours of the velocity magnitude for 0.2 m/s in gray, 2 m/s in red, 4 m/s in yellow and 6 m/s in light yellow. In addition the contours have a decreasing transparency with increasing velocity magnitude. The contour of 6 m/s is completely opaque.

For orientation three contours of the fault system (see Fig. 36) are shown at 0 km depth, 5 km depth and 10 km depth. Additionally a box of extend $[-80\text{ km}, 35\text{ km}] \times [-50\text{ km}, 80\text{ km}] \times [-15\text{ km}, 3\text{ km}]$ is drawn with grid lines in y-direction every 25 km and in z-direction every 5 km.

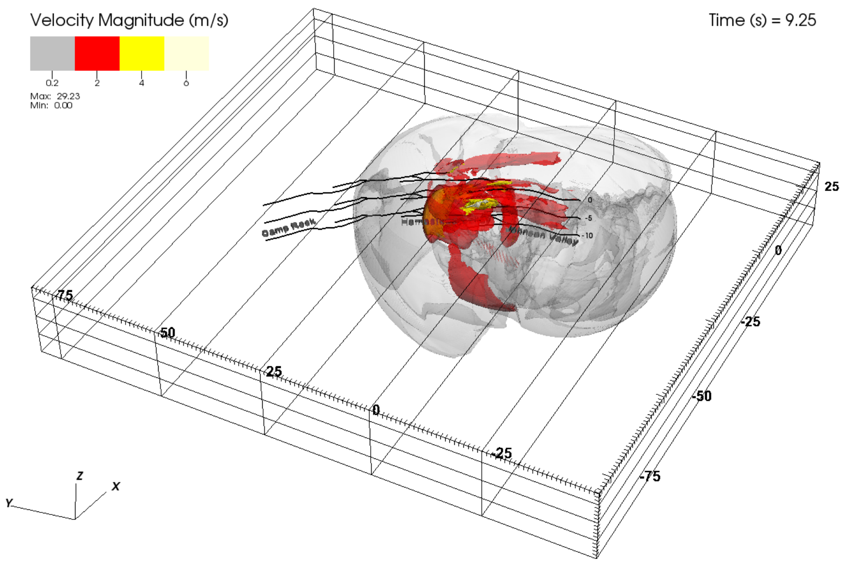
In the beginning of the simulation, shown in Fig. 37, the seismic waves originate from the propagating rupture and strong, local ground motion is visible at the surface. The snapshots at 19.5 s and 25 s in Fig. 38 show a highly heterogeneous structure of the wave field. The high velocity magnitudes of 2-6 m/s are mainly scattered in vicinity of the surface.

SUMMARY

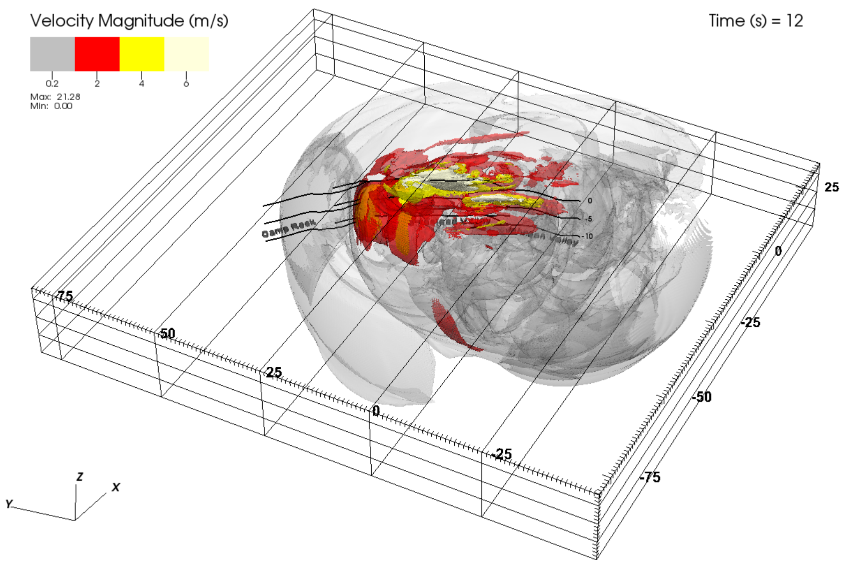
This chapter evaluated our computational core's performance when running GTS dynamic rupture earthquake simulations of the 1992 Landers setup. The setup uses sixth order of convergence in the wave propagation component and a total of 191,098,540 for spatial discretization. Our evaluation included, in Ch. 10.2, strong scaling results of our targeted supercomputers, SuperMUC-1, SuperMUC-2, Stampede and Tianhe-2. Again, we sustained petascale performance on all machines with multi-PFLOPS performance on the Intel Xeon Phi accelerated systems Stampede and Tianhe-2.

In comparison to the homogeneous results, presented in [31], our asynchronous communication scheme increased the parallel efficiencies in the strong scaling study on SuperMUC-2. Here, we reached parallel efficiencies above 85 % for all configurations when scaling from 128 to 3,072 nodes. Using all of SuperMUC-2's 3,072 nodes, we achieved 1.5 PFLOPS in hardware, which translates to a parallel efficiency of 85.9 % with 128 nodes as baseline. With respect to SuperMUC-2's reported HPL-performance, we reached 53.3 % with this run.

Finally, Ch. 10.3 presented results of a repeated production simulation of the 1992 Landers earthquake. However, in comparison to the earlier run on SuperMUC-1, presented in [31], we wrote wave-field output and checkpoints. Including the output, our production simu-

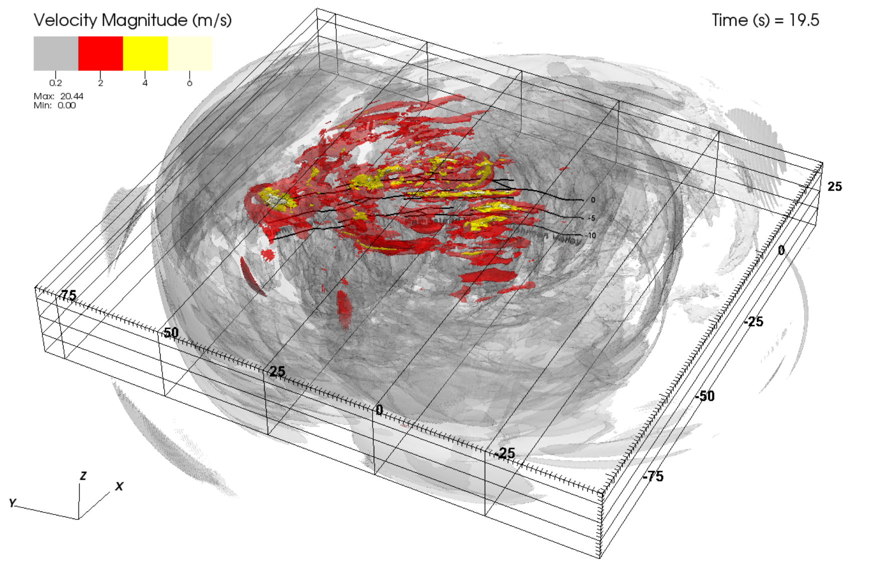


(a) Wave field after 9.25 s of simulated time.

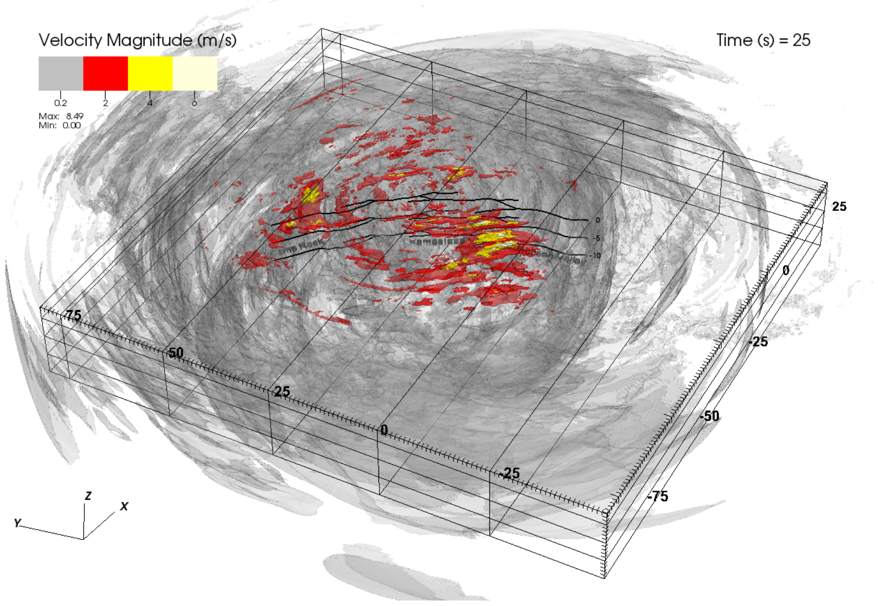


(b) Wave field after 12 s of simulated time.

Figure 37: Wave field of the machine-size 1992 Landers production run on SuperMUC-2 after 9.25 s and 12 s of simulated time. Shown are contour plots for different velocity magnitudes. For orientation three contours of the fault system at a depth of 0 km, 5 km and 10 km are shown.



(a) Wave field after 19.5 s of simulated time.



(b) Wave field after 25 s of simulated time.

Figure 38: Wave field of the machine-size 1992 Landers production run on SuperMUC-2 after 19.5 s and 25 s of simulated time. Shown are contour plots for different velocity magnitudes. For orientation three contours of the fault system at a depth of 0 km, 5 km and 10 km are shown.

lation sustained 1.4 PFLOPS on all 3,072 nodes of SuperMUC-2. This corresponds to 94.3 % of the observed strong scaling performance and verifies the value of the scaling studies. Considering SuperMUC-2's HPL-performance again, we reached 53.3 % in this production simulation. Ch. 10.3 closed with plots of the obtained wave-field showing the complexity of the interaction between seismic wave propagation and the rupture process.

11 PETASCALE LOCAL TIME STEPPING

Our last setup studies the strong scaling behavior of our computational core utilizing the full clustered local time stepping functionality. For this purpose we simulate seismic wave propagation in Mount Merapi. Mount Merapi is a volcano on the island of Java, Indonesia. The volcano is highly active and has caused multiple fatalities in recent years.

Ch. 11.1 discusses the numerical configuration of the Mount Merapi setup. Next, we study the theoretical aspects of local time stepping in Ch. 11.2. Strong scaling of our computational core on SuperMUC-1 and SuperMUC-2 is discussed in Ch. 11.3, followed by a simulation with production character in Ch. 11.4.

11.1 MOUNT MERAPI

The Mount Merapi setup uses sixth order of convergence and a total of 99,831,401 elements for spatial discretization. The setup uses parameters $\rho = 2400\,\mathrm{kg/m^3}$, $\lambda \approx 3.3\,\mathrm{GPa}$ and $\mu \approx 4.7\,\mathrm{GPa}$ inside the volcano. All coordinates are relative to $(0,0,0)$ located below Mount Merapi's peak at mean sea level. Here, we assume that every element inside the sphere with center at $(0,0,4\,\mathrm{km})$ and a radius of 5.1 km is inside the volcano. The remaining computational domain has parameters $\rho = 2000\,\mathrm{kg/m^3}$, $\lambda \approx 2.3\,\mathrm{GPa}$, $\mu \approx 2.4\,\mathrm{GPa}$.

Boundary conditions are set to free-surface at the surface and outflow everywhere else. The seismic source is a double-couple point source approximation at $(0,0,0)$.

Adaptive mesh refinement is employed to resolve the surface topography, the material contrast of the two-region model and the outflow boundaries. Fig. 39 illustrates the tetrahedral mesh. For illustrative purposes the mesh is extracted in a cylinder with a radius of 5 km. The overall memory consumption of our computational core is approximately 1.3 TiB when running the setup with global time stepping.

11.2 LTS IN THEORY

Analogue to the LOH.1 discussion in Ch. 8, we first analyze local time stepping for the Mount Merapi setup from a theoretical perspective.

Fig. 40 shows the density of the time steps. As in Ch. 8, the solid line refers to per-element time steps, while the gray boxes illustrate rate-2 clustering. The density of the per-element time steps in Fig. 40

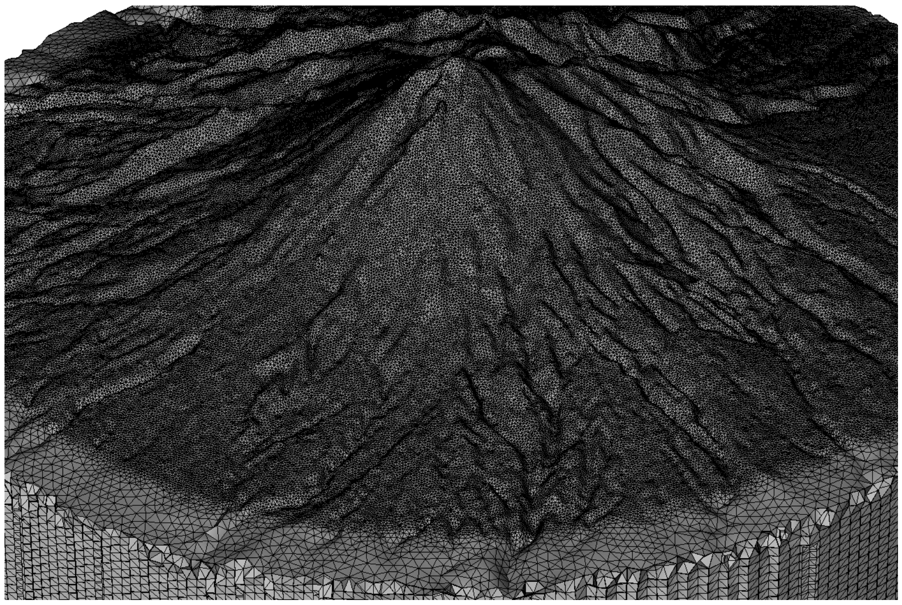


Figure 39: Illustration of the tetrahedral mesh in the Mount Merapi setup. The mesh is shown only for a cylinder with 5 km radius.

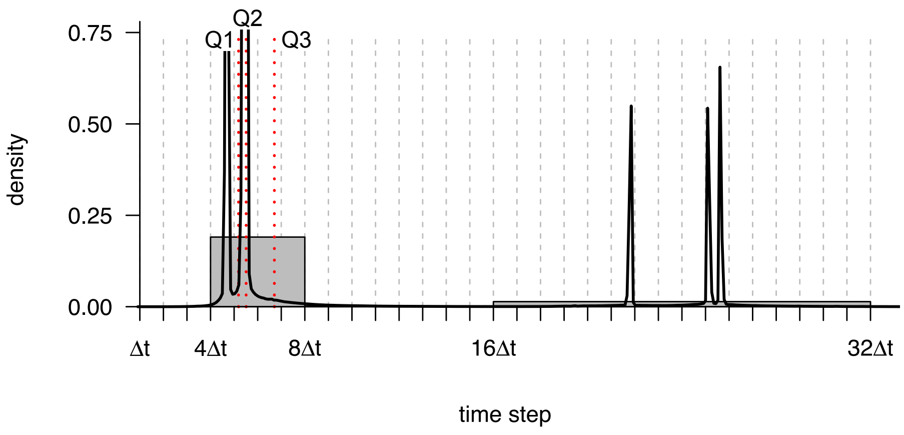


Figure 40: Density of time steps in the Mount Merapi setup. The solid line shows the density in dependency of the elements' time steps. Red dotted lines show the location of the 25 % quartile (Q1), the 50 % quartile (Q2) and the 75 % quartile (Q3). Gray boxes represent a time step clustering with a rate of 2. The shown density is limited to elements with time steps in $[\Delta t, 32\Delta t[$ and cut off for densities exceeding 0.75.

is highly heterogeneous with large peaks in the interval $[4\Delta t, 6\Delta t]$ and smaller peaks in the interval $[21\Delta t, 26\Delta t]$. The sharpness of the peaks is a result of the used regular mesh once the target resolution is reached and geometric features are absent. This regular mesh structure is for example visible on the boundaries of the extracted cylinder in Fig. 39. Considering the illustrated rate-2 clustering in Fig. 40, cluster $C_3 = [4\Delta t, 8\Delta t[$ contains almost all elements.

The cluster's elements for rate-2 clustering are shown in Fig. 41. To support the visualization two regions are cut out of the volcano. The left regions in Fig. 41 are bound by planes with normals $(1, -0.2, 0)$ and $(1, -1.3, 0)$. Planes with normals $(1, -0.25, 0)$ and $(1, 0.5, 0)$ were used to cut the right regions out of the mesh. As in the LOH.1 visualization, shown in Fig. 19, the color of the tetrahedrons refers to the per-element time step required by the CFL-condition. The two material regions are shown as almost transparent shades of gray.

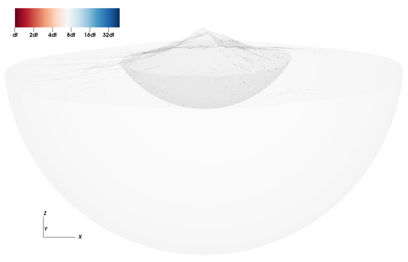
The first cluster $C_1 = [\Delta t, 2\Delta t[$ in Fig. 41a contains only very few sliver elements loosely scattered along the topography of the volcano and in proximity of the material interface. Cluster $C_2 = [2\Delta t, 4\Delta t[$ in Fig. 41b has a higher number elements scattered in the same region. The few elements of these two clusters lower the time step in global time stepping by factor of 4. Analogue to the LOH.1 example, almost no connectivity exists between the scattered elements. This proves the necessity of our clustered LTS scheme's flexibility with respect to the clustering. Requiring connectivity of the LTS clusters would either result in a high number of clusters with equal time steps or require us to impose connectivity by lowering the time steps of all elements in vicinity of the surface and the material interface.

The third cluster $C_3 = [4\Delta t, 8\Delta t[$ in Fig. 41c covers almost all interior elements of the Mount Merapi setup. In addition a few sliver elements are part of C_3 and scattered along the outer boundary. The dense block of elements in the interior corresponds to the regular structures already observed in the density plot (see Fig. 40).

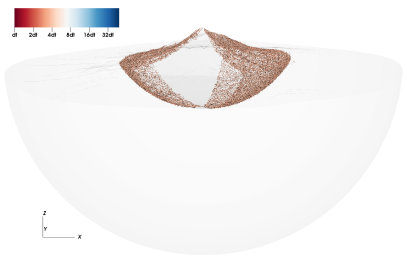
Cluster $C_4 = [8\Delta t, 16\Delta t[$ contains elements at the boundary of our computational domain and the material interface. The outer counterpart of C_3 is given by $C_5 = [16\Delta t, 32\Delta t[$. C_5 densely covers the volume of the outer material region. Again the dense block corresponds to the regular mesh structures, visible in the illustration of the mesh (see Fig. 39) and the time step density (see Fig. 40). The last cluster $C_6 = [32\Delta t, 64\Delta t[$ consist of elements scattered loosely along the boundary of our computational domain.

Fig. 42 shows the load of all elements up to a certain time step. We see a stair case pattern with jumps at the location of the peak densities. This behavior is significantly different from the smooth increase of the LOH.1 setup in Fig. 20 and is a result of the regular structures.

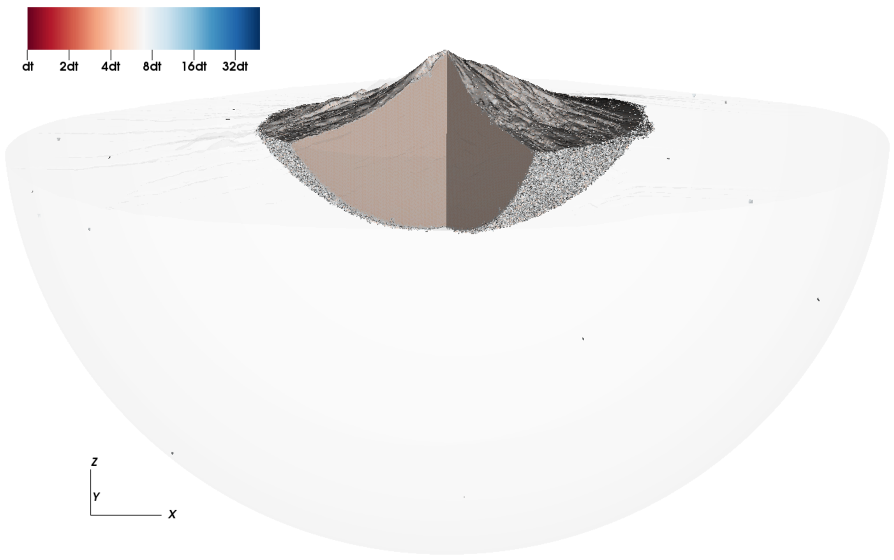
While the largest time step is more than 45 times bigger than the smallest time step, most elements are within $[4\Delta t, 6\Delta t]$ as the loca-



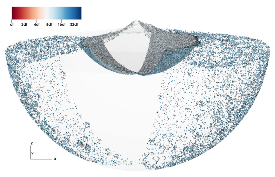
(a) $C_1 = [\Delta t, 2\Delta t[$.



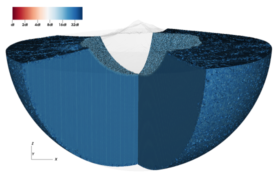
(b) $C_2 = [2\Delta t, 4\Delta t[$.



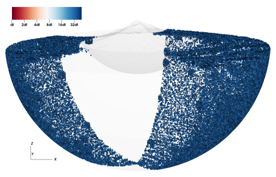
(c) $C_3 = [4\Delta t, 8\Delta t[$.



(d) $C_4 = [8\Delta t, 16\Delta t[$.



(e) $C_5 = [16\Delta t, 32\Delta t[$.



(f) $C_6 = [32\Delta t, 64\Delta t[$.

Figure 41: Spatial distribution of the clusters' elements for the Mount Merapi setup and a rate-2 clustering. The two material regions are visualized by two almost transparent shades of gray. The individual elements of the clusters a colored by their CFL time step.

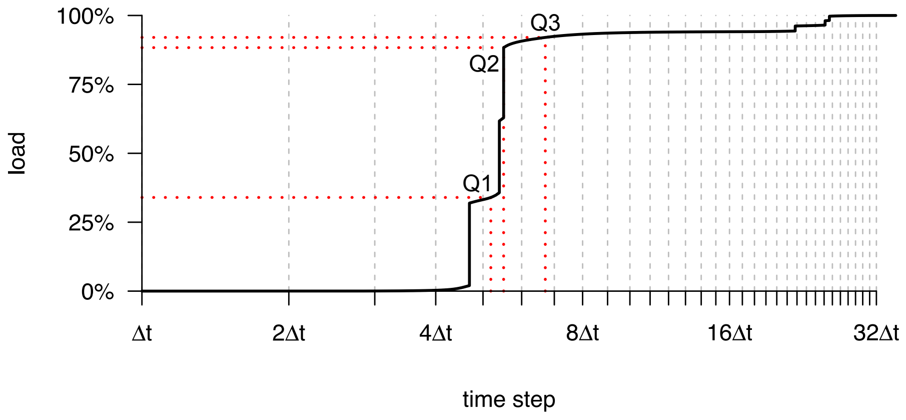


Figure 42: Distribution of the computational load in the Mount Merapi setup assuming a theoretical, perfectly performing, per-element LTS algorithm. The solid line shows the summed load in dependency of the time step. Red dotted lines show the location of the 25 % quartile (Q1), the 50 % quartile (Q2) and the 75 % quartile (Q3). The shown density is limited to time steps in $[\Delta t, 32\Delta t[$.

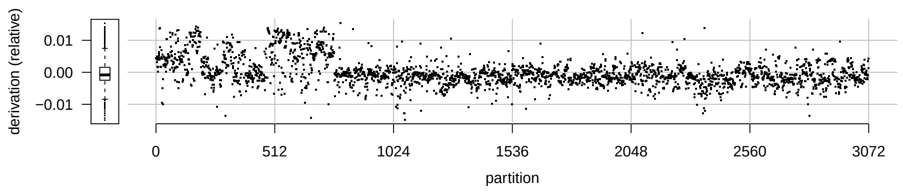
tions of the quartiles show. Assuming that per-element time stepping is possible at GTS performance, the maximum possible LTS speedup is 6.4 ×.

With respect to a suitable clustering, we see that rate-2 clustering is nearly optimal. In this case cluster $C_3 = [4\Delta t, 8\Delta t[$ carries 76.2 % of all elements and by far most of the overall load. Cluster $C_5 = [16\Delta t, 32\Delta t[$ still contains 22.2 % of all elements with a comparable small load. As a result the maximum possible speedup for rate-2 clustering is 4.8 × with respect to GTS. This is more than 75 % of the theoretical per-element LTS speedup.

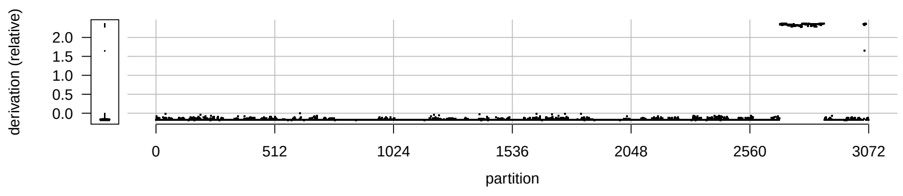
For rate-3 clustering cluster $C_2 = [3\Delta t, 9\Delta t[$ contains 76.8 % of the elements. However in comparison to rate-2 clustering, we underestimate the time step by an additional Δt for the majority of elements in C_2. The maximum possible speedup for rate-3 clustering is 3.6 × corresponding to 56.7 % of the theoretical per-element LTS speedup.

For the memory requirements of local time stepping, we have to consider the non-constant, per-element time data and the load balancing. While the influence of the time data is distributed among the partitions, the load balancing proves to be crucial. Fig. 43 shows the derivation from the mean number of elements per partition (#elements /mean − 1). Here METIS generated 3,072 partitions (see Ch. 3.3). The derivations for global time stepping in Fig. 43a reflect the load imbalances of the partitioning. We see that the obtained quality is high in general with load imbalances within ±2%.

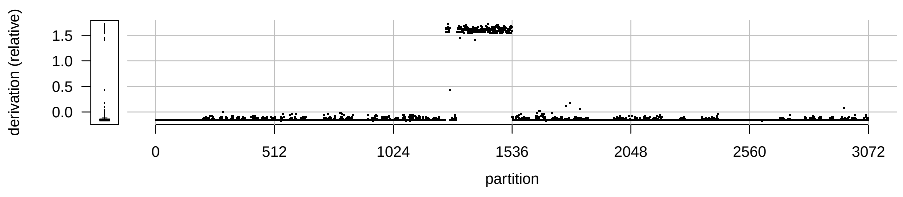
Fig. 43b shows the derivations for rate-2 clustering and Fig. 43c for rate-3 clustering. Now the majority of partitions contains less than the average number of elements. This is a result of the two large clusters containing almost all elements in both clusterings.



(a) Global time stepping.



(b) Rate-2 clustered LTS.



(c) Rate-3 clustered LTS

Figure 43: Relative number of elements per partition in the Mount Merapi setup. Shown is the relative derivation from the mean number of elements per partition (#elements/mean − 1) for GTS and clustered LTS. On the left boxplots are shown, while the right plots show the derivation in dependency of the partition.

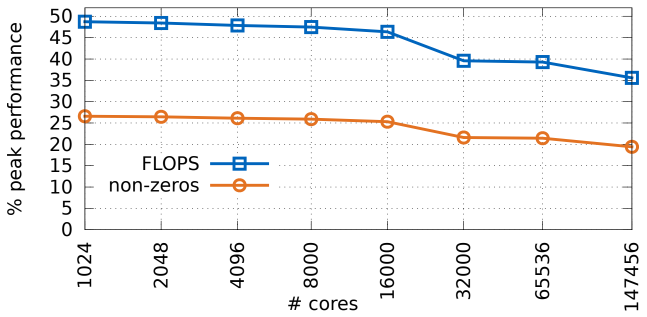


Figure 44: Strong scaling peak efficiencies using an intermediate version of SeisSol for the Mount Merapi setup. Source: [9]

The partition dependent derivations of rate-2 clustering in Fig. 43b show that METIS schedules C_5, the cluster covering the dense, outer block in rate-2 clustering, mainly to partitions between 2560 and 3072. The result is an increased memory consumption of ~$2.5\times$ compared to GTS for nodes computing these partitions.

For rate-3 clustering in Fig. 43c C_3, the cluster covering the dense, outer block, is scheduled between partitions 1024 and 1536. This results in an increased memory consumption of ~$1.5\times$.

The next chapter shows, that an increase of $1.5-2.5\times$ is easily amortized by our computational core's strong scaling performance. However, these factors might increase for larger meshes and become problematic for larger node counts, e.g. the announced 50,000+ nodes of Aurora[1]. In this case future versions of our core could account for memory usage in the partitioning or use low-bandwidth memory on future Intel Xeon Phi generations for partitions with many elements.

11.3 STRONG SCALING

INTERMEDIATE VERSION The authors of [9] use the Mount Merapi setup to study the strong scaling behavior of our computational core. The homogeneous functionality of the core in the version of [9] is very similar to the version of [31] discussed as part of the weak scaling in Ch. 9.2.

Fig. 44 shows the obtained peak efficiencies of the Mount Merapi setup on SuperMUC-1. We see similar peak efficiencies to those of the 1992 Landers strong scaling in Fig. 34. The efficiencies lower after 1000 nodes (16,000 cores). The descent is more severe than in the 1992 Landers runs of [31] due to the lower number of total elements.

The runs of [9] sustained 35.6 % of SuperMUC-1's theoretical peak performance when running on all 9,216 nodes (147,456 cores). This corresponds to 1.13 PFLOPS in hardware with a parallel efficiency of 73.0 % using 64 nodes (1,024 cores) as baseline. The per-node memory

1 https://web.archive.org/web/20150907104816/http://aurora.alcf.anl.gov/

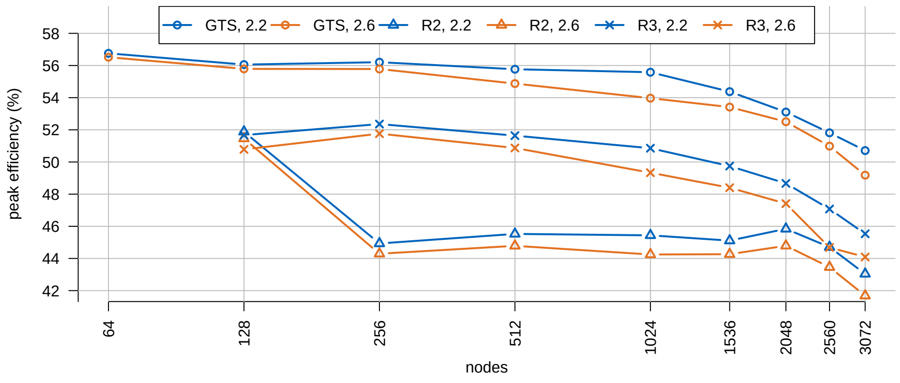


Figure 45: Strong scaling peak efficiencies of the final computational core for the Mount Merapi setup. Shown are the peak efficiencies of GTS, rate-2 (R2) and rate-3 (R3) clustered LTS using HSW-MUC's guaranteed frequency of 2.2 GHz in blue and using the base frequency of 2.6 GHz in orange.

consumption is 142.8 MiB for our computational core using all 9,216 nodes.

FINAL VERSION With the global time stepping results of [9] as reference, we now study the strong scaling performance of our final computational core on SuperMUC-2. Again we expect to benefit from the high single-node performance on SuperMUC-2 and our asynchronous communication scheme. However, in contrast to the 1992 Landers setting of Ch. 10.2, we are now able to exploit the full local time stepping functionality of our core. As discussed in Ch. 11.2, the maximum possible speedup is 4.8 × for rate-2 and 3.6 × for rate-3 clustering. Considering the observed GTS speedups at scale in comparison to [9, 31], these maximum possible speedups are outstanding. Thus, in terms of time-to-solution, we expect to benefit mostly from clustered LTS.

Fig. 45 shows the peak efficiencies for the Merapi strong scaling on SuperMUC-1. All runs were performed at 2.2 GHz and 2.6 GHz, used HTT and left the last core for our communication thread (see Ch. 8.3). The strong scaling for global time stepping starts at 64 nodes and that for local time stepping at 128 nodes. As discussed in Ch. 11.2, we have to respect the higher memory requirements of partitions holding elements with large time steps in LTS, while the memory requirements of GTS are almost constant.

At 128 nodes the GTS runs are able maintain the high peak efficiencies of the weak scaling study in Ch. 9. In comparison to the strong scaling presented in [9] this is an increase of ~5 % in terms of the peak efficiency. The rate-2 and rate-3 clustered local time stepping show slightly lower peak efficiencies, but still reach more than 50 % of hardware peak efficiency. Taking the higher value of the local time stepping operations into account, this corresponds to a speedup of

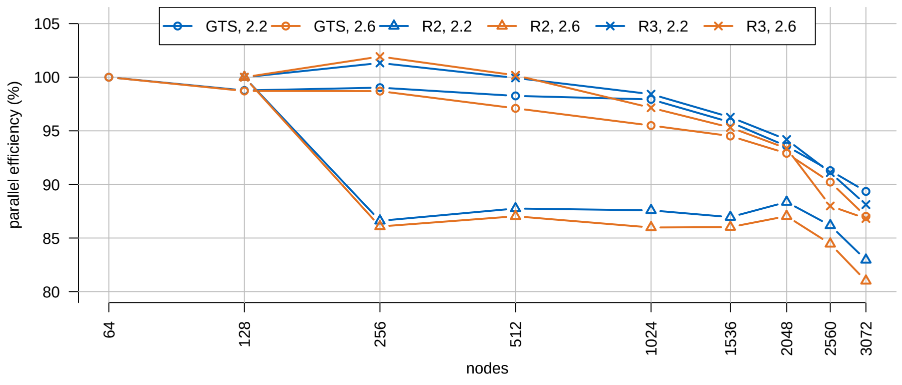


Figure 46: Strong scaling parallel efficiencies of the final computational core for the Mount Merapi setup. Shown are the parallel efficiencies of GTS, rate-2 (R2) and rate-3 (R3) clustered LTS using HSW-MUC's guaranteed frequency of 2.2 GHz in blue and using the base frequency of 2.6 GHz in orange.

4.5 × for rate-2 clustering at 2.2 GHz and 2.6 GHz. Rate-3 clustering sustains a speedup of 3.3 × at both frequencies.

Fig. 46 shows the obtained parallel efficiencies of the strong scaling study. We see that global time stepping maintains more than 87 % in all runs, while rate-2 clustering stays above 81 % and rate-3 clustering above 86 %. For rate-2 clustering we observe a drop when going from 128 to 256 nodes. This is an indicator that our simple partitioning approach should be improved as discussed in Ch. 3.3.

All configurations sustained petascale performance when utilizing all 3,072 nodes of SuperMUC-2. Global time stepping reached 1.5 PFLOPS in hardware at 2.2 GHz and 1.8 PFLOPS at 2.6 GHz. This is equivalent to 50.7 % of theoretical peak performance together with a parallel efficiency of 89.3 % at 2.2 GHz. At 2.6 GHz the GTS runs reached 49.1 % peak efficiency and 87.0 % parallel efficiency at 3,072 nodes.

Rate-2 clustered local time stepping sustained 1.3 PFLOPS with a peak efficiency of 43.0 % and a parallel efficiency of 83.0 % at 2.2 GHz. At 2.6 GHz 1.5 PFLOPS with a peak efficiency of 41.7 % and a parallel efficiency of 81.0 % were reached. In terms of time-to-solution, rate-2 clustered LTS outperformed the GTS runs by 4.1 × using all 3,072 nodes of SuperMUC-2. This is 84.9 % of the theoretical rate-2 speedup and 64.3 % of the theoretical peak speedup for per-element time stepping.

Rate-3 clustering reached 1.4 PFLOPS at 2.2 GHz. This corresponds to a peak efficiency of 45.5 % and a parallel efficiency of 88.1 %. At 2.6 GHz a performance of 1.6 PFLOPS was reached in hardware. Here we observe a peak efficiency of 44.1 % and a parallel efficiency of 86.8 %. The obtained speedup for rate-3 clustering is 3.2 × compared to GTS. In this case the runs reached 89.8 % of the theoretical peak speedup of rate-3 clustering and 50.9 % of the theoretical peak speedup for per-element time stepping.

Concerning SuperMUC-2's reported 2.8 PFLOPS in the Linpack benchmark, we are able to reach 62.5 % of the HPL performance using GTS, 53.0 % with rate-2 and 56.1 % with rate-3 clustering.

11.4 PRODUCTION CHARACTER

The authors of [9] present a run under production conditions, which simulated the first 5 s of the Mount Merapi setup using GTS. This computation took 3 hours and 7.5 minutes including the initialization and sustained a performance of 1.1 PFLOPS in hardware on all 9,216 nodes of the SuperMUC-1 system.

To show the applicability of all of our computational core's features under production conditions, we now discuss a similar simulation on all 3,072 nodes of SuperMUC-2. The repeated simulation used rate-2 clustering and reached a simulation time of 10 s. Additional to the seismic receivers of [9], the simulation wrote the wave-field every 0.2 s to SuperMUC-2's GSS. The total size of the output was reduced to 349 GiB by writing the data for the constant basis only. Note that receiver output is element-local and performed independently in every time stepping cluster. However wave-field output requires synchronization of all clusters in time.

The simulation took a total of 1 hour, 6 minutes and 38 seconds on all 3,072 nodes of SuperMUC-2. This duration includes all overhead, such as the MPI startup or I/O. In average the run sustained 1.3 PFLOPS in hardware with a hardware to non-zero ratio of 1.9. This is equivalent to speedup of $5.6 \times$ with respect to [9] and equivalent to 46.2 % of SuperMUC-2's HPL performance under production conditions using local time stepping and including I/O.

Fig. 47 shows the time series and frequency spectrum of GTS and rate-2 clustered LTS for σ_{zz} with respect to a randomly picked receiver inside the volcano. The same output is presented for the 5 simulated seconds of the production charactrer run in [9]. We see that the rate-2 results are almost identical to the global time stepping results and that frequencies beyond 20 Hz are resolved.

Additionally Fig. 48 and Fig. 49 show the obtained wave field of the petascale, rate-2 clustered LTS simulation. Shown is a grid with extends $[-2.5 \text{ km}, 2.5 \text{ km}] \times [-2.5 \text{ km}, 2.5 \text{ km}] \times [0, 2.5 \text{ km}]$ for orientation and contours of different velocity magnitudes.

The seismic waves propagate regularly from the seismic source in the first snapshot after 1.0 s in Fig. 48a. After 1.8 s the seismic waves hit the surface in Fig. 48b. The result is highly scattered wave propagation visible in figures 49a and 49b.

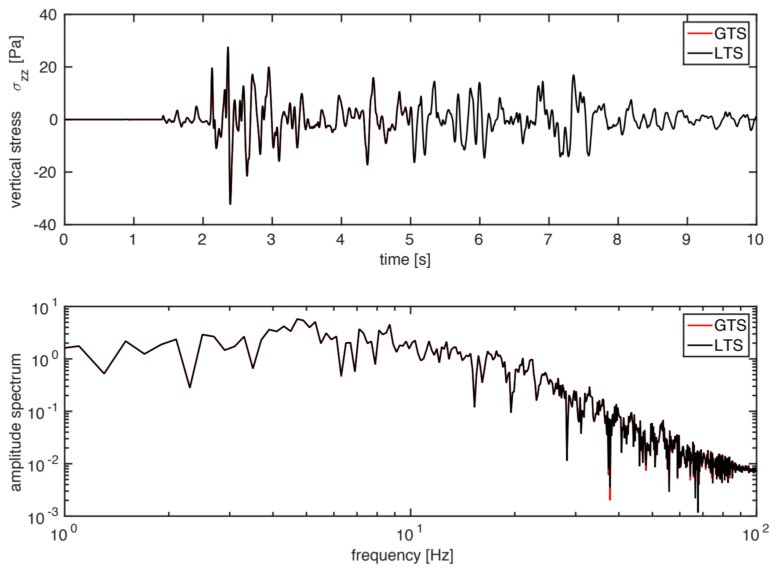


Figure 47: Time series and frequency spectrum of a random receiver using GTS and LTS for the Mount merapi setup. Shown is σ_{zz} for 10 simulated seconds and rate-2 clustering.

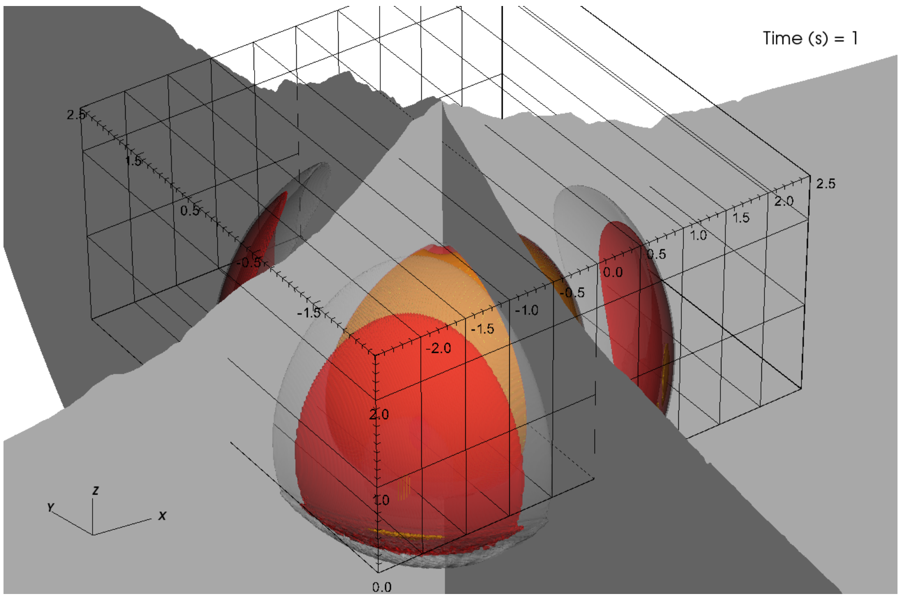
SUMMARY

This final chapter studied all of our computational core's functionality in a setup simulating seismic wave propagation in Mount Merapi. The setup uses sixth order of convergence in the wave propagation component and a total of 99,831,401 elements for spatial discretization.

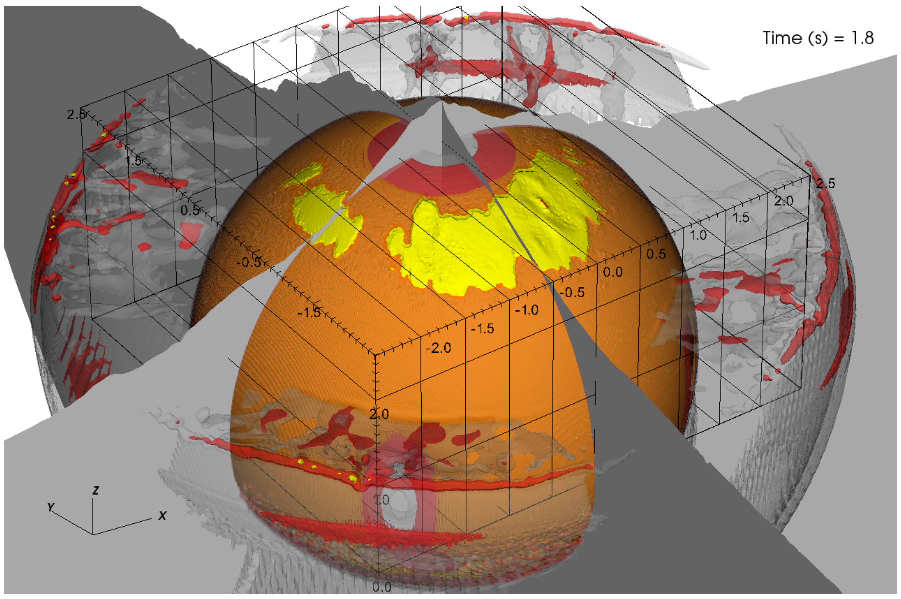
We started our evaluation in Ch. 11.2 by studying possible LTS speedups for the given setup. We quantified the theoretical peak speedup for all LTS schemes over GTS as 6.4 × and between 3.6 ×-4.8 × for our two clustering strategies. Additionally, we studied the time step density and load. Here, we identified two large homogeneous time stepping regions, but also many loosely scattered sliver elements in vicinity of geometric features.

The next chapter, Ch. 11.3, presented a strong scaling study of the Mount Merapi setup on SuperMUC-2. Here, we outperformed the SuperMUC-1 results of our computational core in the version of [9]. The GTS speedups are a result of further improvements of our performance engineering, especially the asynchronous communication scheme. The LTS speedups stem from algorithmic improvements.

All GTS and LTS configurations on SuperMUC-2 reached petascale performance on all 3,072 nodes. Further, with 128 nodes as baseline, all runs sustained parallel efficiencies above 80 %. At 2.6 GHz we sustained 1.8 PFLOPS for GTS and 1.3-1.4 PFLOPS for LTS, in other words 53.0 %-62.5 % of SuperMUC-2's HPL performance.

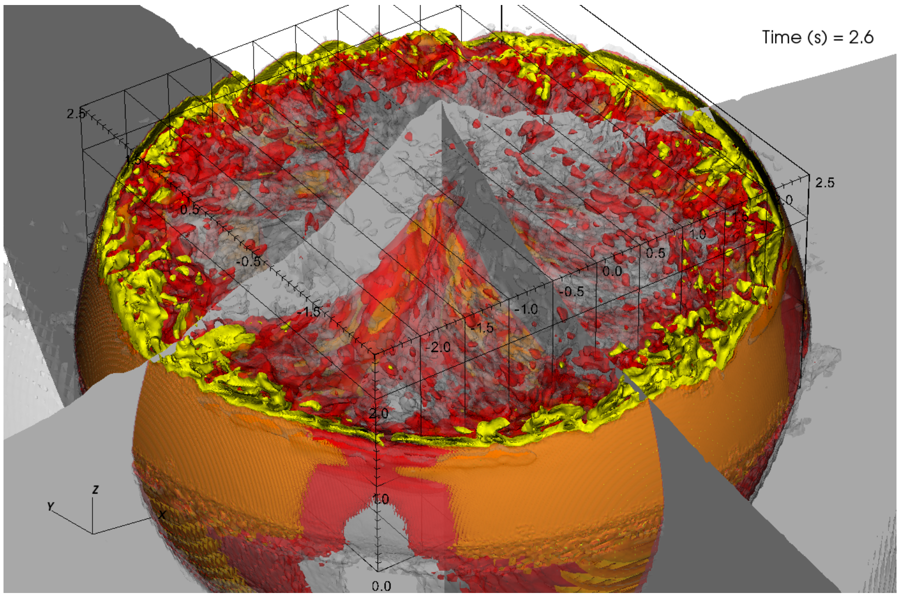


(a) Wave field after 1.0 s of simulated time.

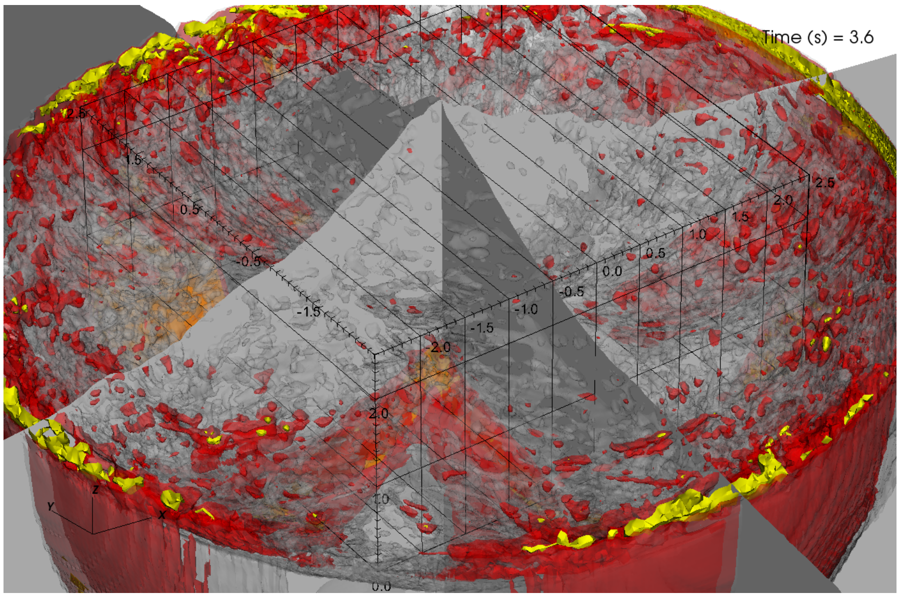


(b) Wave field after 1.8 s of simulated time.

Figure 48: Wave field of the full-machine, production character Merapi run on SuperMUC-2 after 1.0 s and 1.8 s of simulated time. Shown are contour plots for different velocity magnitudes.



(a) Wave field after 2.6 s of simulated time.



(b) Wave field after 3.6 s of simulated time.

Figure 49: Wave field of the full-machine, production character Merapi run on SuperMUC-2 after 2.6 s and 3.6 s of simulated time. Shown are contour plots for different velocity magnitudes.

Most importantly the use of local time stepping greatly increased the value of the floating point operations. Here, we obtained a 3.2 $\times$-4.1 $\times$ speedup with respect to GTS on all 3,072 nodes of SuperMUC-2.

We finalized this chapter by a repetition of the production character run presented in [9]. However, in comparison to [9], we used all of SuperMUC-2 instead of SuperMUC-1, wrote wave-field output and used our clustered LTS scheme. This run sustained 1.3 PFLOPS in hardware and sustained a speedup of 5.6 $\times$ over the SuperMUC-1 run of [9].

Summarizing the small-scale runs of Ch. 8, the weak scaling of Ch. 9, the multiphysics scaling and production runs in Ch. 10, and the large-scale LTS wave propagation runs in this chapter, we systematically evaluated a broad range of performance characteristics. Our computational core sustains high efficiencies in the two important metrics of sustained peak efficiency and sustained parallel efficiency. Especially the results of the final run under production conditions, simulating seismic wave propagation in Mount Merapi, is outstanding. The authors of [9] already used an intermediate version of our computational core and reported speedups exceeding 5 $\times$ over the initial GTS version of SeisSol. Considering our LTS results on SuperMUC-2, this speedup sums to over 25 $\times$ and summarizes the scientific outcome of this thesis.

12 CONCLUSIONS AND OUTLOOK

CONCLUSIONS

This thesis presented a new computational core for the earthquake simulation package SeisSol. The core replaced SeisSol's entire time marching procedure with a novel, clustered LTS scheme and features a streamlined layout of data structures and asynchronous, hybrid parallelization. Within this thesis we systematically addressed all made design decisions from an algorithmic perspective in Pt. i and an engineering perspective in Pt. ii. Finally Pt. iii evaluated our core in different setups up to full machine utilization. Note that most of the discussed developments were released[1][2] as open-source software under the BSD 3-Clause license.

Our algorithmic considerations in Pt. i covered constraints on LTS relations of neighboring elements in Ch. 3.1 and multi-partition clustering, including a normalization step for homogeneity, in Ch. 3.3. The following chapters 3.4 and 3.5 of Pt. i discussed the time management and scheduling. This led to a asynchronous, clustered LTS scheme with prioritization of critical work.

After the derivation of our clustered LTS scheme, Pt. ii switched to an engineering perspective for the high performance implementation of the scheme. First, Ch. 4 introduced our targeted, homogeneous and heterogeneous supercomputers SuperMUC-1, SuperMUC-2, Stampede and Tianhe-2. Next, Ch. 5 derived the layout for the data structures of our computational core. Our considerations included alignment for different vector-instruction sets, hardware prefetching and in-place communication. With the data structures and the algorithm at hand, Ch. 6 presented a hybrid, asynchronous parallelization. This parallelization uses the send-receive-model of MPI and features a dedicated communication thread for MPI-progression. Pt. ii's final chapter, Ch. 7, introduced our low-level ADER-DG kernels for the time, volume and surface integrators. Inside the ADER-DG kernels we use custom matrix kernels and an auto-tuning approach for the corresponding sparse-dense decisions.

The final part of this thesis, Pt. iii, evaluated the performance of our computational core using different setups. In Pt. iii's first chapter, Ch. 8, we discussed the single-node GTS performance obtained in [10] and extended these results with small-scale GTS and LTS results on SuperMUC-2. The discussion covered three generations of

1 https://github.com/SeisSol/SeisSol
2 https://github.com/TUM-I5/seissol_kernels

Intel Xeon CPUs and the first-generation Intel Xeon Phi coprocessor. While the use of high-order simulations is compelling from a mathematical convergence standpoint, our results also show that we enter the compute-bound regime with high orders of convergence. We saw that our computational core is able to handle lowering machine balances, increased data locality requirements, and that our core is able to exploit mature vector instructions, such as 256-bit and 512-bit fused multiply-add instructions, when using high orders of convergence. The obtained hardware peak efficiencies of ~60 % on the CPUs and ~40 % on the Intel Xeon Phi coprocessor prove that our computational core fully unleashes the computational power of state-of-art architectures.

Before studying the obtained LTS performance, we discussed theoretical speedups of possible LTS strategies compared to GTS. For the given setup, we identified a theoretical maximum of $4.0\times$ for every LTS implementation. Our simulations showed that our core is able to maintain GTS peak efficiencies when running LTS. Additionally, the results showed a speedup of $2.6\times$ using a single node with LTS and excellent strong scalability up to eight nodes with a parallel efficiency of more than 90 %.

Ch. 9's setup studied our computational core's weak scaling performance. Here, we discussed the results obtained in [31] for SuperMUC-1, Stampede and Tianhe-2 and extended them with a scaling on SuperMUC-2. The largest configuration on Tianhe-2 used a total of 24,576 Intel Xeon Phi coprocessors, which is equivalent to 1.4 Mio. cores, and obtained 8.6 PFLOPS in hardware. The used $\sim 5.0 \cdot 10^{12}$ degrees of freedom show that our computational core is ready to push the resolution of earthquake simulations to the limits.

Pt. iii's third chapter, Ch. 10, applied our computational core to a simulation of the 1992 Landers earthquake. Outstanding is the discussed production run performed on SuperMUC-2, which outperformed the results of [31]. Here we sustained 1.4 PFLOPS in hardware for a non-stop computation lasting 1 hour and 52 minutes. This performance is equivalent to 50.5 % of SuperMUC-2's HPL-performance and includes the writing of wave-field output and checkpoints.

The final chapter, Ch. 11, of Pt. iii evaluated our computational core's performance using high-order GTS and LTS configurations in machine-size SuperMUC-2 configurations. Here, we studied the performance of seismic wave propagation in the volcano Mount Merapi. All GTS and LTS configurations sustained petascale performance on the 3,072 nodes of SuperMUC-2 and reached strong-scaling parallel efficiencies above 80 % with 64 nodes (GTS) and 128 nodes (LTS) as baseline.

Ch. 11 closed with the presentation of a high-order LTS run under productions conditions on SuperMUC-2. This run took over 1 hour and 6 minutes, wrote wave-field output, and sustained 1.3 PFLOPS in hardware. Compared to the equivalent GTS run on SuperMUC-1, presented in [9], this translates to a 5.6 $\times$ speedup in time-to-solution. However, the authors of [9] already used the computational core, presented in this thesis, in an intermediate version and observed speedups above 5 $\times$ with respect to the initial GTS version of SeisSol. The combined speedup is above 25 $\times$ and summarizes the scientific outcome of this thesis.

SCIENTIFIC IMPACT The computational core, presented in this thesis, significantly contributed to the work shown in [10, 31, 9, 8]. The collaborative paper [9] received the *PRACE-ISC Award 2014* for high-order, petascale simulations of seismic wave propagation with SeisSol. The work presented in [31] was nominated as finalist for the *2014 ACM Gordon Bell Prize* and used SeisSol for the simulation of dynamic rupture earthquake simulations on homogeneous and heterogeneous supercomputers. Additionally, my doctoral research project was honored with an *ACM-IEEE CS George Michael Memorial HPC Fellowship* in 2014 and selected for the 2015 *Doctoral Showcase Program* of the International Conference for High Performance Computing, Networking, Storage and Analysis.

OUTLOOK

As discussed in Ch. 1, the modeling, algorithmic and engineering demands of dynamic rupture earthquake simulations are immense. Future extensions of the presented work have to include latest model extensions, tackle algorithms featuring the requested geometric complexities and material heterogeneities, and concurrently use million of cores. All of these challenges are typically tightly coupled and design decisions taken anywhere in the simulation pipeline might influence all remaining steps. In the following we discuss SeisSol's most important, future algorithmic and engineering challenges.

ALGORITHM With respect to numerics our current approach uses planar faces for the representation of the tetrahedral elements. This approach could be improved by the use of high-order elements to maintain the order of convergence along dynamic rupture interfaces and for accurate representation of the surface topography. Additionally, an increased spatial resolution of the fault system and surface topography together with p-adaptivity [24] could be used alternatively to or in conjunction with high-order elements.

Similar, our constant material parameters inside the elements might conflict with the high-order convergence. A possible solution is the use of high-order material representations inside the elements [14].

In terms of our parallel algorithm, we introduced in Ch. 3.3 a very simple approach with weighted vertices in the dual-graph for LTS partitioning. This approached successfully achieved high efficiencies up to full-machine utilization. Anyhow, future extension could further increase scalability with weights for the edges summarizing non-constant communication frequencies and volumes, or feature an auto-tuning for the LTS partitioning in the offline-phase.

While the presented computational core supports coupling to dynamic rupture propagation in GTS configurations, support for LTS configurations is pending. Tackling this support in near-future has the potential of enabling new dynamic rupture physics. Aside from homogeneous time step variations, the meshing of complex fault systems, such as dipping faults with shallow angles, tends to generate extreme sliver elements. These elements effectively prevent simulations using global time stepping. Ch. 10.1 discussed different approaches for the usage of dynamic rupture in LTS configurations. Any of the approaches could be integrated with moderate effort into our computational core. However, thorough testing and verification of the resulting scheme is imperative. Aside from comparison against different time stepping approaches in SeisSol itself, the benchmarks of the *SCEC/ USGS Spontaneous Rupture Code Verification Project* [29] qualify for this task.

HPC From an engineering perspective SeisSol's new computational core, presented in this thesis, is prepared for next-generation Intel Xeon CPUs and socketed, next-generation Intel Xeon Phis. Here, the final tuning has to be accomplished, once the respective architectures are available. This includes, for example, explicit usage of near and far memory on next-generation Intel Xeon Phis for optimal performance.

The presented static shared memory parallelization of the computational loops in Ch. 6 and the static partitioning of the computational domain for distributed memory parallelization in Ch. 3.3 reflects the uniform load of our computational core. However, the performance of a single core might be the weakest link of our static partitioning. In other words, a single faulty core has the potential to slow down SeisSol entirely. In fact, we observed first performance variations stemming from non-constant node performance in Ch. 7.4 and Pt. iii. Here, dynamic scheduling and dynamic repartitioning could help to mitigate the problem. Another promising solution of the issue could involve backup-nodes, which would serve as replacement for low-performing nodes or in the case of hard faults. Such an approach could be combined with systematic oversubscription and shared backup-nodes for multiple jobs running in parallel.

HAZARD ASSESSMENT The ever increasing power of supercomputers, with several, announced 100+ PFLOPS machines for the next few years (e.g. [61]), allows to think big in terms of seismic hazard assessment. While current approaches use simplified models for the forward simulations (e.g. [19, 27]), at some point it might become feasible to use dynamic earthquake rupture simulations for all or a selected number of forward simulations.

This would result in a broadened focus of the topics discussed in this thesis and bring our co-design of algorithms and performance engineering to the next level. From an algorithmic viewpoint, we would embed single runs of our computational core in a larger simulation environment covering the high-dimensional space of uncertainties.

Here, we could exploit similarities of the uncertainty configurations and solve multiple forward simulations concurrently. For example, different locations of the nucleation patch or different prescribed stress fields result in almost identical hardware operations performed on the respective machine. Thus, single-node or single-core parallelization could be interpreted in terms of concurrent forward runs. This means that we could use the same mesh and constant data for multiple, concurrent forward runs and, for example, perform vector instructions across problem boundaries. This would significantly increase the ratio of solution-related data, allow for perfect vectorization without zero-padding, and pave the ground for high performance implementations of advanced numerics, such as high-order elements or heterogeneous materials within the elements.

BIBLIOGRAPHY

[1] *Software Optimization Guide for AMD Family 15h Processors*. Advanced Micro Devices, 2012.

[2] *AMD64 Architecture Programmer's Manual, Volume 4: 128-Bit and 256-Bit Media Instructions*. Advanced Micro Devices, 2013.

[3] M. Barall and R. A. Harris. Metrics for Comparing Dynamic Earthquake Rupture Simulations. *Seismological Research Letters*, 2014.

[4] M. Benjemaa, N. Glinsky-Olivier, V. Cruz-Atienza, and J. Virieux. 3-D dynamic rupture simulations by a finite volume method. *Geophysical Journal International*, 2009.

[5] J. Bielak, O. Ghattas, and E. Kim. Parallel Octree-Based Finite Element Method for Large-Scale Earthquake Ground Motion Simulation. *Computer Modeling in Engineering and Sciences*, 2005.

[6] J. Bielak, R. W. Graves, K. B. Olsen, R. Taborda, L. Ramírez-Guzmán, S. M. Day, G. P. Ely, D. Roten, T. H. Jordan, P. J. Maechling, et al. The ShakeOut earthquake scenario: Verification of three simulation sets. *Geophysical Journal International*, 2010.

[7] R. F. Boisvert, R. Pozo, and K. A. Remington. The Matrix Market Exchange Formats: Initial Design. *National Institute of Standards and Technology Internal Report*, 1996.

[8] A. Breuer, A. Heinecke, M. Bader, and C. Pelties. Accelerating SeisSol by Generating Vectorized Code for Sparse Matrix Operators. In *Parallel Computing - Accelerating Computational Science and Engineering (CSE)*, Advances in Parallel Computing. IOS Press, 2013.

[9] A. Breuer, A. Heinecke, S. Rettenberger, M. Bader, A.-A. Gabriel, and C. Pelties. Sustained Petascale Performance of Seismic Simulations with SeisSol on SuperMUC. In *Supercomputing*, Lecture Notes in Computer Science. Springer, 2014.

[10] A. Breuer, A. Heinecke, L. Rannabauer, and M. Bader. High-Order ADER-DG Minimizes Energy- and Time-to-Solution of SeisSol. In *High Performance Computing*, Lecture Notes in Computer Science. Springer, 2015.

[11] I. N. Bronstein, J. Hromkovic, B. Luderer, H.-R. Schwarz, J. Blath, A. Schied, S. Dempe, G. Wanka, S. Gottwald, E. Zeidler, et al. *Taschenbuch der Mathematik*. Springer, 2012.

[12] L. Carrington, D. Komatitsch, M. Laurenzano, M. M. Tikir, D. Michéa, N. Le Goff, A. Snavely, and J. Tromp. High-Frequency Simulations of Global Seismic Wave Propagation Using SPECFEM3D_GLOBE on 62K Processors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[13] C. Castro, M. Käser, and E. Toro. Space–time adaptive numerical methods for geophysical applications. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 2009.

[14] C. E. Castro, M. Käser, and G. B. Brietzke. Seismic waves in heterogeneous material: subcell resolution of the discontinuous Galerkin method. *Geophysical Journal International*, 2010.

[15] M. Christen, O. Schenk, and Y. Cui. PATUS for Convenient High-Performance Stencils: Evaluation in Earthquake Simulations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.

[16] B. Cockburn, C.-W. Shu, and G. E. Karniadakis. *Discontinuous Galerkin Methods: Theory, Computation and Applications*. Springer, 2000.

[17] V. Cruz-Atienza and J. Virieux. Dynamic rupture simulation of non-planar faults with a finite-difference approach. *Geophysical Journal International*, 2004.

[18] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, et al. Scalable Earthquake Simulation on Petascale Supercomputers. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010.

[19] Y. Cui, E. Poyraz, K. B. Olsen, J. Zhou, K. Withers, S. Callaghan, J. Larkin, C. Guest, D. Choi, A. Chourasia, et al. Physics-based Seismic Hazard Analysis on Petascale Heterogeneous Supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013.

[20] S. M. Day, J. Bielak, D. Dreger, R. Graves, S. Larsen, K. Olsen, and A. Pitarka. Tests of 3D Elastodynamic Codes: Final Report for Lifelines Project 1A02. *Pacific Earthquake Engineering Research Center*, 2003.

[21] J. de la Puente, M. Käser, M. Dumbser, and H. Igel. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes - IV. Anisotropy. *Geophysical Journal International*, 2007.

[22] J. de la Puente, J.-P. Ampuero, and M. Käser. Dynamic Rupture Modeling on Unstructured Meshes Using a Discontinuous Galerkin Method. *Journal of Geophysical Research: Solid Earth*, 2009.

[23] M. Dumbser and M. Käser. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes - II. The three-dimensional isotropic case. *Geophysical Journal International*, 2006.

[24] M. Dumbser, M. Käser, and E. F. Toro. An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes - V. Local time stepping and p-adaptivity. *Geophysical Journal International*, 2007.

[25] G. Gassner, F. Lörcher, and C.-D. Munz. A discontinuous Galerkin scheme based on a space-time expansion II. Viscous flow equations in multi dimensions. *Journal of Scientific Computing*, 2008.

[26] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The SCALASCA performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 2010.

[27] R. Graves, T. H. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, et al. CyberShake: A Physics-Based Seismic Hazard Model for Southern California. *Pure and Applied Geophysics*, 2011.

[28] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.

[29] R. Harris, M. Barall, R. Archuleta, E. Dunham, B. Aagaard, J. Ampuero, H. Bhat, V. Cruz-Atienza, L. Dalguer, P. Dawson, et al. The SCEC/USGS Dynamic Earthquake Rupture Code Verification Exercise. *Seismological Research Letters*, 2009.

[30] R. A. Harris and S. M. Day. Effects of a Low-Velocity Zone on a Dynamic Rupture. *Bulletin of the Seismological Society of America*, 1997.

[31] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A.-A. Gabriel, C. Pelties, A. Bode, W. Barth, X.-K. Liao, K. Vaidyanathan, M. Smelyanskiy, and P. Dubey. Petascale High Order Dynamic Rupture Earthquake Simulations on Heterogeneous Supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014.

[32] T. Ichimura, K. Fujita, S. Tanaka, M. Hori, M. M. L. Lakshman, Y. Shizawa, and H. Kobayashi. Physics-based urban earthquake

simulation enhanced by 10.7 BlnDOF $\times$ 30 K time-step unstructured FE non-linear seismic wave simulation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2014.

[33] *Intel Xeon Processor E5-2600 Product Family Uncore Performance Monitoring Guide*. Intel, 2012.

[34] *Intel Architecture Instruction Set Extensions Programming Reference*. Intel, 2013.

[35] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel, 2013.

[36] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, September 2014.

[37] *Optimizing Performance with Intel Advanced Vector Extensions*. Intel, September 2014.

[38] *Intel Xeon Phi Coprocessor System Software Developers Guide*. Intel, March 2014.

[39] G. Karniadakis and S. Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford University Press, 2013.

[40] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on scientific Computing*, 1998.

[41] M. Käser, M. Dumbser, J. De La Puente, and H. Igel. An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes - III. Viscoelastic attenuation. *Geophysical Journal International*, 2007.

[42] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, et al. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*. Springer, 2012.

[43] M. Kohler, H. Magistrale, and R. Clayton. Mantle Heterogeneities and the SCEC Reference Three-Dimensional Seismic Velocity Model Version 3. *Bulletin of the Seismological Society of America*, 2003.

[44] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, 2010.

[45] J. E. Kozdon, E. M. Dunham, and J. Nordström. Simulation of Dynamic Earthquake Ruptures in Complex Geometries Using High-Order Finite Difference Methods. *Journal of Scientific Computing*, 2013.

[46] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.

[47] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.

[48] O. O'Reilly, J. Nordström, J. E. Kozdon, and E. M. Dunham. Simulation of Earthquake Rupture Dynamics in Complex Geometries Using Coupled Finite Difference and Finite Volume Methods. *Communications in Computational Physics*, 2015.

[49] C. Pelties, J. Puente, J.-P. Ampuero, G. B. Brietzke, and M. Käser. Three-dimensional dynamic rupture simulation with a high-order discontinuous Galerkin method on unstructured tetrahedral meshes. *Journal of Geophysical Research: Solid Earth*, 2012.

[50] A. Plesch, J. H. Shaw, C. Benson, W. A. Bryant, S. Carena, M. Cooke, J. Dolan, G. Fuis, E. Gath, L. Grant, et al. Community Fault Model (CFM) for Southern California. *Bulletin of the Seismological Society of America*, 2007.

[51] R. Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.

[52] M. Rietmann, P. Messmer, T. Nissen-Meyer, D. Peter, P. Basini, D. Komatitsch, O. Schenk, J. Tromp, L. Boschi, and D. Giardini. Forward and Adjoint Simulations of Seismic Wave Propagation on Emerging Large-Scale GPU Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.

[53] M. Rietmann, D. Peter, O. Schenk, B. Uçar, and M. Grote. Load-Balanced Local Time Stepping for Large-Scale Wave Propagation. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015.

[54] M. Schlegel, O. Knoth, M. Arnold, and R. Wolke. Multirate Runge-Kutta schemes for advection equations. *Journal of Computational and Applied Mathematics*, 2009.

[55] B. Seny, J. Lambrechts, R. Comblen, V. Legat, and J.-F. Remacle. Multirate Time Stepping for Accelerating Explicit Discontinuous Galerkin Computations with Application to Geophysical Flows. *International Journal for Numerical Methods in Fluids*, 2013.

[56] B. Seny, J. Lambrechts, T. Toulorge, V. Legat, and J.-F. Remacle. An efficient parallel implementation of explicit multirate Runge-Kutta schemes for discontinuous Galerkin computations. *Journal of Computational Physics*, 2014.

[57] *2015 Science Collaboration Plan*. Southern California Earthquake Center, 2014.

[58] J. Tago, V. M. Cruz-Atienza, J. Virieux, V. Etienne, and F. J. Sánchez-Sesma. A 3D hp-adaptive discontinuous Galerkin method for modeling earthquake dynamics. *Journal of Geophysical Research: Solid Earth*, 2012.

[59] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*. Springer, 2009.

[60] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron. From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE, 2006.

[61] *Fact Sheet: Collaboration of Oak Ridge, Argonne, and Livermore (CORAL)*. U.S. Department Of Energy, National Nuclear Security Administration, December 2014.

[62] S. Wenk, C. Pelties, H. Igel, and M. Käser. Regional wave propagation using the discontinuous Galerkin method. *Journal of Geophysical Research: Solid Earth*, 2013.

[63] L. C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas. A high-order discontinuous Galerkin method for wave propagation through coupled elastic-acoustic media. *Journal of Computational Physics*, 2010.

[64] A. R. Winters and D. A. Kopriva. High-Order Local Time Stepping on Moving DG Spectral Element Meshes. *Journal of Scientific Computing*, 2014.