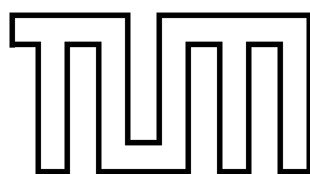


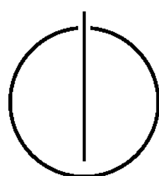
FAKULTÄT FÜR INFORMATIK

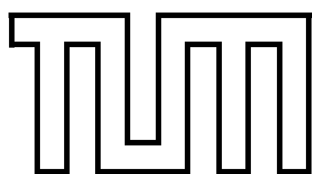
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

A Hardware-aware ADER-DG Method for Hyperbolic Partial Differential Equations

Pablo Gómez





FAKULTÄT FÜR INFORMATIK

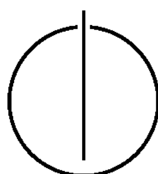
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

A Hardware-aware ADER-DG Method for Hyperbolic Partial Differential Equations

Eine Hardware-nahe ADER-DG Methode für Hyperbolische Partielle Differentialgleichungen

Author:	Pablo Gómez
Supervisor:	Univ.-Prof. Dr. Michael Bader
Advisor:	Dipl.-Math Alexander Breuer
Date:	October 14, 2015



Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.
I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, den 14. Oktober 2015 Pablo Gómez

Acknowledgements

At this point I want to express my gratitude to the persons, who assisted me in understanding the mathematics behind this thesis and supported me in its creation. Also I would like to thank the members of the Leibniz Rechenzentrum and the chair of scientific computing for the opportunity to work on the CoolMUC2 supercomputer and the possibility to do my thesis on a topic that I found both interesting and demanding.
My thanks go especially to my thesis advisor Alexander Breuer for his support, endurance and guidance during the creation of this thesis. Whenever the challenges of the topic threatened to overwhelm me, I could count on him.
Furthermore I am very thankful for the support of my friend Max, who assisted me in eliminitating grammatical and spelling errors, and in finding an appropiate phrasing.
Finally, my grateful thanks are extended to my mother for her continuous support throughout my study.

Abstract

This thesis covers the implementation of a hardware-aware arbitrary high order derivative discontinuous galerkin (ADER-DG) method for hyperbolic partial differential equations. This method has been of a particular interest recently, as it is able to obtain a high order solution in space and time. A specific focus of this thesis is on the optimization of such a method on a current architecture. While the implemention is only used to solve a three-dimensional advection problem, it is also applicable to other hyperbolic problems like, e.g., the elastic wave equation.
A brief introduction to the mathematics of the approach is given, followed by a thorough analysis of the characteristics of the method. The implementation is optimized for running on *Intel Xeon E5-2697 v3* CPUs. The matrix-matrix multiplications in the method are performed using efficient library implementations and a hybrid OpenMP/MPI implementation is presented. Overall it is shown that the ADER-DG method is suitable for performance optimization on supercomputers and can be used to attain a high order of accuracy. Results for single and multi node performance on such an architecture are presented. In particular, convergence order, FLOPS and scaling are investigated. On a single node 50% of the theoretical peak performance and on 64 nodes up to 38% theoretical peak performance are reached.

Contents

Acknowledgements **iv**

Abstract **v**

A Hardware-aware ADER-DG Method for Hyperbolic Partial Differential Equations **1**

1 Introduction **1**

2 Hyperbolic Partial Differential Equations and Discontinuous Galerkin Schemes **3**
2.1 Hyperbolic Partial Differential Equations 3
2.2 Arbitrary High Order Discontinuous Galerkin 4
2.3 Fully Discrete ADER-DG Scheme 7

3 Problem Characteristics and Optimization **11**
3.1 Problem Characteristics 11
3.2 Matrix Structure 11
3.3 Single Core Optimization 12
3.4 Hybrid MPI/OpenMP Parallelization 14

4 Results **19**
4.1 Setup 19
4.2 Single-Node Performance 20
4.2.1 Kernel Implementation 21
4.2.2 OpenMP Implementation 23
4.3 Multi-Node Performance 25

5 Conclusion and Outlook **29**

Bibliography **29**

1 Introduction

Numerical methods converging with high order have been of particular interest in recent years. For both linear and nonlinear systems, increasingly accurate models can be attained using such methods. For hyperbolic problems in particular, the popular combination of continuous or discontinuous Galerkin methods with Runge-Kutta methods for time discretization has been established. Unfortunately though, the computational cost of Runge-Kutta methods for higher orders is negatively affected by the Butcher barrier [9].
One particular scheme that provides high order in both time and space without being affected by such constraints, is the arbitrary high order derivative discontinuous Galerkin (ADER-DG) scheme used in this thesis. Based on the standard discontinuous Galerkin (DG) schemes [14], it relies on the arbitrary high order derivative (ADER) approach first presented by Toro et al. in [19]. This has lead to a multitude of different schemes and applications, e.g. [12][16][11].
Originally, the Cauchy-Kovalewski procedure is used in the ADER-DG scheme. It provides a way to replace time derivatives in the scheme with spatial derivatives. Recent research has been focused on using a so-called element-local space-time Galerkin predictor that is based on the weak differential form of the equation instead of the strong form used in the Cauchy-Kovalewski procedure [12].
Research for ADER-DG approaches on tetrahedons using a Cauchy-Kovalewski procedure can be found in [6][8][7]. In [6], an efficient implementation on the Intel's *Sandy Bridge* architecture reaching up to 50% of achievable peak performance is presented and in [8] petascale performance on the SuperMUC supercomputer with an ADER-DG approach is achieved. In [7], energy and time-to-solution oriented optimization of a high order ADER-DG implementation is performed.
This thesis' focus lies in the application of an ADER-DG scheme using a predictor based on the weak differential form to hyperbolic partial differential equations (PDE) in three dimensions, spatially discretized into cuboids. For simplicity's sake, an approach for the advection equation is constructed. This approach is presented in detail in the following section. Application to, e.g., the elastic wave equation and other hyperbolic problems would be possible too.
Afterwards a detailed analysis of the characteristics of the scheme is performed to explore the possibilities in optimizing a hardware-aware implementation in Section 3. Starting on one computation node only, we present results comparing the suitability of different, efficient implementations of matrix-matrix multiplications and then move on to a multi-node setting. Therein, relying on the Message Parsing Interface (MPI) and the Open Multiprocessing (OpenMP) API, we use a hybrid MPI/OpenMP implementation similar to [13].

In Section 4 we present results produced with the above-mentioned implementation. Section 5, finally, takes a look at possible directions to pursue in the future and gives a brief conclusion drawn from the results.

2 Hyperbolic Partial Differential Equations and Discontinuous Galerkin Schemes

In this section hyperbolic PDEs in three dimensions are introduced and we show the mathematical ideas behind the ADER-DG scheme used in this thesis. A more detailed presentation of hyperbolic problems can be found in [17]. The mathematics of the ADER-DG approach are shown in more detail in [16] [19].

2.1 Hyperbolic Partial Differential Equations

As shown in [17, 421 f.], assuming no source term, a three dimensional conservation law can be written as

$$\frac{\partial q}{\partial t} + \frac{\partial f(q)}{\partial x} + \frac{\partial g(q)}{\partial y} + \frac{\partial h(q)}{\partial z} = 0, \tag{2.1}$$

with $q(x, y, z, t) \in \mathbb{R}^{N^{\text{Var}}}$ describing the N^{Var} conserved quantities and $f(q), g(q)$ and $h(q)$ being the flux functions in x-,y- and z-direction respectively. Examples of hyperbolic PDEs are, e.g., the advection equation, the shallow water equations or the elastic wave equation. For the advection equation the above equation simplifies to

$$\frac{\partial q}{\partial t} + A\frac{\partial q}{\partial x} + B\frac{\partial q}{\partial y} + C\frac{\partial q}{\partial z} = 0, \tag{2.2}$$

where A, B and C are in $\mathbb{R}^{N^{\text{Var}} \times N^{\text{Var}}}$ and constant. There is a multitude of numerical schemes to deal with these types of equations. One popular way are finite volume methods which are presented in great detail in [17].

2.2 Arbitrary High Order Discontinuous Galerkin

Discontinuous Galerkin methods have been a focus of detailed research in the last years. With the demand and need for high order methods such methods found a variety of applications, not only to hyperbolic problems, but also to elliptic and parabolic ones. A more detailed introduction to DG methods can be found in [14].
For the ADER-DG scheme we start by introducing the reference element $T^e := T_s^e \times T_t^e := [0,1]^4$ with $T_s^e := [\xi_0, \xi_1] \times [\eta_0, \eta_1] \times [\zeta_0, \zeta_1]$ and $T_t^e = [\tau_0, \tau_1]$ to discretize the spatial and temporal domain into N^E elements of size $\Delta x \times \Delta y \times \Delta z \times \Delta t$. Reference coordinates for each element $T_{i,j,k}^e$ are

$$x = x_{i-\frac{1}{2}} + \xi \Delta x, \quad y = y_{j-\frac{1}{2}} + \eta \Delta y, \quad z = z_{k-\frac{1}{2}} + \zeta \Delta z, \tag{2.3}$$

where $x_{i-\frac{1}{2}}$, $y_{j-\frac{1}{2}}$ and $z_{k-\frac{1}{2}}$ are the left boundaries of $T_{i,j,k}^e$ in x, y and z direction respectively. Similar to [12] for each spatial dimension, we approximate the numerical solution u in the reference element T^e using modal basis functions $\psi_{\mathbf{q}}(\xi, \eta, \zeta)$ of degree N with coefficients $\hat{u}_{\mathbf{q}}$, where $\mathbf{q} = (p, q, r)$ is a multi-index with $0 < p, q, r < N+1$. They are constructed using the tensor product of one dimensional basis functions $\psi_k(\xi), \psi_k(\eta), \psi_k(\zeta), 1 \leq k \leq N+1$ at supporting Gauss-Legendre points in the interval [0; 1] as

$$\psi_{\mathbf{p}}(\xi, \eta, \zeta) = \psi_p(\xi)\psi_q(\eta)\psi_r(\zeta). \tag{2.4}$$

In particular we use the shifted Legendre polynomials

$$P_n(x) = (-1)^n \sum_{k=0}^{n} \binom{n}{k} \binom{n+k}{k} (-x)^k, \tag{2.5}$$

so that the numerical solution is approximated by

$$u = \sum_{1}^{\mathbf{q}} \psi_{\mathbf{q}}(\xi, \eta, \zeta) \hat{u}_{\mathbf{q}}. \tag{2.6}$$

Following [12], we can insert time into the approximation and define space-time basis functions $\theta_{\mathbf{p}}(\xi, \eta, \zeta, \tau)$ with coefficients $\hat{q}_{\mathbf{p}}$, which are polynomials of degree N and where $\mathbf{p} = (p, q, r, s)$ is a multi-index with $1 \leq p, q, r, s \leq N+1$. Additionally using the basis function $\psi_k(\tau)$ for the time, they are constructed by the tensor-product of the space basis functions ψ_k, so that

$$\theta_{\mathbf{p}}(\xi, \eta, \zeta, \tau) = \psi_p(\xi)\psi_q(\eta)\psi_r(\zeta)\psi_s(\tau). \tag{2.7}$$

This allows us to rewrite (2.1) as

$$\frac{\partial u}{\partial \tau} + \frac{\partial f^*}{\partial \xi} + \frac{\partial g^*}{\partial \eta} + \frac{\partial h^*}{\partial \zeta} = 0 \tag{2.8}$$

where

$$f^* = \frac{\Delta t}{\Delta x} f, \quad g^* = \frac{\Delta t}{\Delta y} g, \quad h^* = \frac{\Delta t}{\Delta z} h. \tag{2.9}$$

We can then integrate (2.8) over the space-time control volume T^e and multiply it by the test functions, in this case the $\theta_{\mathbf{p}}$, which yields the weak form

$$\int_{T^e} \theta_{\mathbf{p}} \left(\frac{\partial u}{\partial \tau} + \frac{\partial f^*}{\partial \xi} + \frac{\partial g^*}{\partial \eta} + \frac{\partial h^*}{\partial \zeta} \right) dT^e = 0. \tag{2.10}$$

As in [12], Integration by parts then results in

$$\begin{gathered} \int_{T_s^e} \theta_{\mathbf{p}}(\xi,\eta,\zeta,1) u(\xi,\eta,\zeta,1) dT_s^e - \int_{T_s^e} \theta_{\mathbf{p}}(\xi,\eta,\zeta,0) u(\xi,\eta,\zeta,0) dT_s^e - \\ \int_{T^e} \left(\frac{\partial \theta_{\mathbf{p}}}{\partial \tau} \right) u dT^e + \int_{T^e} \theta_{\mathbf{p}} \frac{\partial f^*}{\partial \xi} dT^e + \int_{T^e} \theta_{\mathbf{p}} \frac{\partial g^*}{\partial \eta} dT^e + \int_{T^e} \theta_{\mathbf{p}} \frac{\partial h^*}{\partial \zeta} dT^e = 0, \end{gathered} \tag{2.11}$$

Furthermore, we can approximate the discrete space-time solution q in the same manner as u, so that

$$q = \sum_{1}^{\mathbf{p}} \theta_{\mathbf{p}}(\xi,\eta,\zeta,\tau) \hat{q}_{\mathbf{p}}. \tag{2.12}$$

Given our modal basis functions, we can thereby approximate the fluxes in the advection equation, so that

$$\hat{f}_{\mathbf{p}} = A \hat{q}_{\mathbf{p}}, \quad \hat{g}_{\mathbf{p}} = B \hat{q}_{\mathbf{p}}, \quad \hat{h}_{\mathbf{p}} = C \hat{q}_{\mathbf{p}}, \tag{2.13}$$

$$\hat{f}^* = \frac{\Delta t}{\Delta x} \hat{f}, \hat{g}^* = \frac{\Delta t}{\Delta y} \hat{g}, \hat{h}^* = \frac{\Delta t}{\Delta z} \hat{h}. \tag{2.14}$$

Inserting into equation (2.11) , it becomes

$$\begin{aligned}
&\int_{T_s^e} \theta_{\mathbf{p}_2}(\xi,\eta,\zeta,1)\theta_{\mathbf{p}}(\xi,\eta,\zeta,1)\hat{q}_{\mathbf{p}} dT_s^e - \int_{T_s^e} \theta_{\mathbf{p}}(\xi,\eta,\zeta,0)\psi_{\mathbf{q}}\hat{u}_{\mathbf{q}} dT_s^e - \\
&\int_{T^e} \left(\frac{\partial \theta_{\mathbf{p}_2}}{\partial \tau}\right) \theta_{\mathbf{p}}\hat{q}_{\mathbf{p}} dT^e + \\
&\int_{T^e} \theta_{\mathbf{p}_2} \frac{\partial \theta_{\mathbf{p}}}{\partial \xi} \hat{f}^*_{\mathbf{p}} dT^e + \int_{T^e} \theta_{\mathbf{p}_2} \frac{\partial \theta_{\mathbf{p}}}{\partial \eta} \hat{g}^*_{\mathbf{p}} dT^e + \int_{T^e} \theta_{\mathbf{p}_2} \frac{\partial \theta_{\mathbf{p}}}{\partial \zeta} \hat{h}^*_{\mathbf{p}} dT^e = 0,
\end{aligned} \tag{2.15}$$

Thereby we can then derive the following matrices

$$K^{\xi}_{\mathbf{p}_2\mathbf{p}} = \int_0^1\int_0^1\int_0^1\int_0^1 \frac{\partial \theta_{\mathbf{p}_2}}{\partial \xi}\,\theta_{\mathbf{p}}\,d\xi d\eta d\zeta d\tau, \tag{2.16}$$

$$K^{\eta}_{\mathbf{p}_2\mathbf{p}} = \int_0^1\int_0^1\int_0^1\int_0^1 \frac{\partial \theta_{\mathbf{p}_2}}{\partial \eta}\,\theta_{\mathbf{p}}\,d\xi d\eta d\zeta d\tau, \tag{2.17}$$

$$K^{\zeta}_{\mathbf{p}_2\mathbf{p}} = \int_0^1\int_0^1\int_0^1\int_0^1 \frac{\partial \theta_{\mathbf{p}_2}}{\partial \zeta}\,\theta_{\mathbf{p}}\,d\xi d\eta d\zeta d\tau, \tag{2.18}$$

$$K^{1}_{\mathbf{p}_2\mathbf{p}} = \int_0^1\int_0^1\int_0^1 \theta_{\mathbf{p}_2}(\xi,\eta,\zeta,1)\,\theta_{\mathbf{p}}(\xi,\eta,\zeta,1)\,d\xi d\eta d\zeta - \int_0^1\int_0^1\int_0^1 \frac{\partial \theta_{\mathbf{p}_2}}{\partial \tau}\,\theta_{\mathbf{p}}\,d\xi d\eta d\zeta d\tau, \tag{2.19}$$

$$F^{0}_{\mathbf{p}\mathbf{q}} = \int_0^1\int_0^1\int_0^1 \theta_{\mathbf{p}}(\xi,\eta,\zeta,0)\,\psi_{\mathbf{q}}\,d\xi d\eta d\zeta, \tag{2.20}$$

so that (2.15) can be written as

$$K^{1}_{\mathbf{p}_2\mathbf{p}}\hat{q}_{\mathbf{p}} + K^{\xi}_{\mathbf{p}_2\mathbf{p}}\hat{f}^*_{\mathbf{p}} + K^{\eta}_{\mathbf{p}_2\mathbf{p}}\hat{g}^*_{\mathbf{p}} + K^{\zeta}_{\mathbf{p}_2\mathbf{p}}\hat{h}^*_{\mathbf{p}} - F^{0}_{\mathbf{p}\mathbf{q}}\hat{u}_{\mathbf{q}} = 0. \tag{2.21}$$

This results in an iterative scheme

$$\hat{q}^{n+1}_{\mathbf{p}} = (K^1)^{-1}_{\mathbf{p}_2\mathbf{p}}\left(F^0_{\mathbf{p}_2\mathbf{q}}\hat{u}_{\mathbf{q}} - K^{\xi}_{\mathbf{p}_2\mathbf{p}}\hat{f}^*_{\mathbf{p}} - K^{\eta}_{\mathbf{p}_2\mathbf{p}}\hat{g}^*_{\mathbf{p}} - K^{\zeta}_{\mathbf{p}_2\mathbf{p}}\hat{h}^*_{\mathbf{p}}\right), \tag{2.22}$$

where we use $\hat{u}_{1,1,1}$ as initial guess. (2.22) will also be referred to as the prediction step.

2.3 Fully Discrete ADER-DG Scheme

As in [16], to attain a fully discrete scheme, we can integrate the flux terms in (2.15) by parts, which yields

$$\begin{aligned}\int_{T_s^e}\theta_{\mathbf{p}_2}(\xi,\eta,\zeta,1)\theta_{\mathbf{p}}(\xi,\eta,\zeta,1)\hat{q}_{\mathbf{p}}dT_s^e-\int_{T_s^e}\theta_{\mathbf{p}}(\xi,\eta,\zeta,0)\psi_{\mathbf{q}}\hat{u}_{\mathbf{q}}dT_s^e-\\ \int_{T^e}\left(\frac{\partial\theta_{\mathbf{p}_2}}{\partial\tau}\right)\theta_{\mathbf{p}}\hat{q}_{\mathbf{p}}dT^e+\\ \int_{T_t^e}\int_{\partial T_s^e}\theta_{\mathbf{p}}\psi_{\mathbf{q}}F d\partial T_s^e dT_t^e-\int_{T^e}\theta_{\mathbf{p}}\frac{\partial\psi_{\mathbf{q}}}{\partial\xi}\hat{f}_{\mathbf{p}}^* dT^e+\\ \int_{T_t^e}\int_{\partial T_s^e}\theta_{\mathbf{p}}\psi_{\mathbf{q}}G d\partial T_s^e dT_t^e-\int_{T^e}\theta_{\mathbf{p}}\frac{\partial\psi_{\mathbf{q}}}{\partial\eta}\hat{g}_{\mathbf{p}}^* dT^e+\\ \int_{T_t^e}\int_{\partial T_s^e}\theta_{\mathbf{p}}\psi_{\mathbf{q}}H d\partial T_s^e dT_t^e-\int_{T^e}\theta_{\mathbf{p}}\frac{\partial\psi_{\mathbf{q}}}{\partial\zeta}\hat{h}_{\mathbf{p}}^* dT^e=0,\end{aligned} \tag{2.23}$$

where F, G and H are numerical fluxes. As numerical flux function the Local-Lax-Friedrichs-Flux

$$\mathcal{F}(q)=\frac{1}{2}\left(f(q^-)+f(q^+)+s_{\max}(q^+-q^-)\right), \tag{2.24}$$

is employed, where q^- and q^+ are the quantities at the respective boundary interface ∂T_s and $s_{\max}$ is the absolute value of the signal velocity wave speed.
As before we can then define matrices

$$\tilde{K}_{\mathbf{pq}}^{\xi}=\int_0^1\int_0^1\int_0^1\frac{\partial\psi_{\mathbf{q}}}{\partial\xi}\,\theta_{\mathbf{p}}\,d\xi d\eta d\zeta d\tau, \tag{2.25}$$

$$\tilde{K}_{\mathbf{pq}}^{\eta}=\int_0^1\int_0^1\int_0^1\frac{\partial\psi_{\mathbf{q}}}{\partial\eta}\,\theta_{\mathbf{p}}\,d\xi d\eta d\zeta d\tau, \tag{2.26}$$

$$\tilde{K}_{\mathbf{pq}}^{\zeta}=\int_0^1\int_0^1\int_0^1\frac{\partial\psi_{\mathbf{q}}}{\partial\zeta}\,\theta_{\mathbf{p}}\,d\xi d\eta d\zeta d\tau, \tag{2.27}$$

$$\tilde{M}_{\mathbf{q}\mathbf{q}_2}=\int_0^1\int_0^1\int_0^1\psi_{\mathbf{q}}\,\psi_{\mathbf{q}_2}\,d\xi d\eta d\zeta, \tag{2.28}$$

and, as in [16], describe the element boundary-related terms in ξ-direction as

$$F_{\mathbf{pq}}^{+,+}=\int_0^1\int_0^1\int_0^1\psi_{\mathbf{q}}(1,\eta,\zeta)\,\theta_{\mathbf{p}}(1,\eta,\zeta,\tau)\,d\eta d\zeta d\tau, \tag{2.29}$$

$$F_{\mathbf{pq}}^{+,-}=\int_0^1\int_0^1\int_0^1\psi_{\mathbf{q}}(1,\eta,\zeta)\,\theta_{\mathbf{p}}(0,\eta,\zeta,\tau)\,d\eta d\zeta d\tau, \tag{2.30}$$

$$F_{\mathbf{pq}}^{-,+}=\int_0^1\int_0^1\int_0^1\psi_{\mathbf{q}}(0,\eta,\zeta)\,\theta_{\mathbf{p}}(1,\eta,\zeta,\tau)\,d\eta d\zeta d\tau, \tag{2.31}$$

$$F_{\mathbf{pq}}^{-,-}=\int_0^1\int_0^1\int_0^1\psi_{\mathbf{q}}(0,\eta,\zeta)\,\theta_{\mathbf{p}}(0,\eta,\zeta,\tau)\,d\eta d\zeta d\tau. \tag{2.32}$$

In η-direction and ζ-direction the matrices are analogously

$$G^{+,+}_{\mathbf{pq}} = \int_0^1 \int_0^1 \int_0^1 \psi_{\mathbf{q}}(\xi, 1, \zeta)\, \theta_{\mathbf{p}}(\xi, 1, \zeta, \tau)\, d\xi d\zeta d\tau, \tag{2.33}$$

$$G^{+,-}_{\mathbf{pq}} = \int_0^1 \int_0^1 \int_0^1 \psi_{\mathbf{q}}(\xi, 1, \zeta)\, \theta_{\mathbf{p}}(\xi, 0, \zeta, \tau)\, d\xi d\zeta d\tau, \tag{2.34}$$

$$G^{-,+}_{\mathbf{pq}} = \int_0^1 \int_0^1 \int_0^1 \psi_{\mathbf{q}}(\xi, 0, \zeta)\, \theta_{\mathbf{p}}(\xi, 1, \zeta, \tau)\, d\xi d\zeta d\tau, \tag{2.35}$$

$$G^{-,-}_{\mathbf{pq}} = \int_0^1 \int_0^1 \int_0^1 \psi_{\mathbf{q}}(\xi, 0, \zeta)\, \theta_{\mathbf{p}}(\xi, 0, \zeta, \tau)\, d\xi d\zeta d\tau, \tag{2.36}$$

$$H^{+,+}_{\mathbf{pq}} = \int_0^1 \int_0^1 \int_0^1 \psi_{\mathbf{q}}(\xi, \eta, 1)\, \theta_{\mathbf{p}}(\xi, \eta, 1, \tau)\, d\xi d\eta d\tau, \tag{2.37}$$

$$H^{+,-}_{\mathbf{pq}} = \int_0^1 \int_0^1 \int_0^1 \psi_{\mathbf{q}}(\xi, \eta, 1)\, \theta_{\mathbf{p}}(\xi, \eta, 0, \tau)\, d\xi d\eta d\tau, \tag{2.38}$$

$$H^{-,+}_{\mathbf{pq}} = \int_0^1 \int_0^1 \int_0^1 \psi_{\mathbf{q}}(\xi, \eta, 0)\, \theta_{\mathbf{p}}(\xi, \eta, 1, \tau)\, d\xi d\eta d\tau, \tag{2.39}$$

$$H^{-,-}_{\mathbf{pq}} = \int_0^1 \int_0^1 \int_0^1 \psi_{\mathbf{q}}(\xi, \eta, 0)\, \theta_{\mathbf{p}}(\xi, \eta, 0, \tau)\, d\xi d\eta d\tau. \tag{2.40}$$

To reduce the notational burden, we can now define

$$\begin{aligned}
f^+_{\mathbf{q}} &= \frac{1}{2} F^{+,+}_{\mathbf{pq}} (\hat{f}_{\mathbf{p}} + s_{\max} \hat{q}^{n+1}_{\mathbf{p}}) + \frac{1}{2} F^{+,-}_{\mathbf{pq}} (\hat{f}_{\mathbf{p}} - s_{\max} \hat{q}^{n+1}_{\mathbf{p}}), \\
f^-_{\mathbf{q}} &= \frac{1}{2} F^{-,+}_{\mathbf{pq}} (\hat{f}_{\mathbf{p}} + s_{\max} \hat{q}^{n+1}_{\mathbf{p}}) + \frac{1}{2} F^{-,-}_{\mathbf{pq}} (\hat{f}_{\mathbf{p}} - s_{\max} \hat{q}^{n+1}_{\mathbf{p}}), \\
g^+_{\mathbf{q}} &= \frac{1}{2} G^{+,+}_{\mathbf{pq}} (\hat{g}_{\mathbf{p}} + s_{\max} \hat{q}^{n+1}_{\mathbf{p}}) + \frac{1}{2} G^{+,-}_{\mathbf{pq}} (\hat{g}_{\mathbf{p}} - s_{\max} \hat{q}^{n+1}_{\mathbf{p}}), \\
g^-_{\mathbf{q}} &= \frac{1}{2} G^{-,+}_{\mathbf{pq}} (\hat{g}_{\mathbf{p}} + s_{\max} \hat{q}^{n+1}_{\mathbf{p}}) + \frac{1}{2} G^{-,-}_{\mathbf{pq}} (\hat{g}_{\mathbf{p}} - s_{\max} \hat{q}^{n+1}_{\mathbf{p}}), \\
h^+_{\mathbf{q}} &= \frac{1}{2} H^{+,+}_{\mathbf{pq}} (\hat{h}_{\mathbf{p}} + s_{\max} \hat{q}^{n+1}_{\mathbf{p}}) + \frac{1}{2} H^{+,-}_{\mathbf{pq}} (\hat{h}_{\mathbf{p}} - s_{\max} \hat{q}^{n+1}_{\mathbf{p}}), \\
h^-_{\mathbf{q}} &= \frac{1}{2} H^{-,+}_{\mathbf{pq}} (\hat{h}_{\mathbf{p}} + s_{\max} \hat{q}^{n+1}_{\mathbf{p}}) + \frac{1}{2} H^{-,-}_{\mathbf{pq}} (\hat{h}_{\mathbf{p}} - s_{\max} \hat{q}^{n+1}_{\mathbf{p}}),
\end{aligned}$$

so that finally, the resulting update step for each iteration in the scheme becomes

$$\begin{aligned}
\hat{u}^{n+1}_{\mathbf{q}} =& \hat{u}^n_{\mathbf{q}} - \\
& \tilde{M}^{-1} \left(\frac{\Delta t}{\Delta x} (f^+_{\mathbf{q}} - f^-_{\mathbf{q}}) - \tilde{K}^{\xi}_{\mathbf{pq}} \hat{f}^*_{\mathbf{p}} \right) + \\
& \tilde{M}^{-1} \left(\frac{\Delta t}{\Delta y} (g^+_{\mathbf{q}} - g^-_{\mathbf{q}}) - \tilde{K}^{\eta}_{\mathbf{pq}} \hat{g}^*_{\mathbf{p}} \right) + \\
& \tilde{M}^{-1} \left(\frac{\Delta t}{\Delta z} (h^+_{\mathbf{q}} - h^-_{\mathbf{q}}) - \tilde{K}^{\zeta}_{\mathbf{pq}} \hat{h}^*_{\mathbf{p}} \right).
\end{aligned} \tag{2.41}$$

Note that for $N = 0$, the presented method simplifies to a finite volume method. Periodic boundary conditions were employed and sufficiently smooth initial conditions used to avoid the problems arising with discontinuities in the initial condition. Alternatively, those could be addressed using limiters as shown in [20].

3 Problem Characteristics and Optimization

This section details the computational characteristics of the ADER-DG solver. We analyze the sparsity patterns in the matrices and describe ways to optimize single core up to multi-node performance. We refer to [8] for similar investigations with a spatial discretization employing tetrahedons.

3.1 Problem Characteristics

The ADER-DG approach has several features that have to be considered to perform an efficient optimization. As the computationally expensive part of the solver can be written as matrix-matrix multiplications, we focus on performing these efficiently.
The first thing to consider is that due to the nature of the ADER-DG approach, the computation of the prediction and update step for each element requires the same matrices. In case of the prediction step those matrices are $K^{\xi}, K^{\eta}, K^{\zeta}, F^0$ and K^{-1}. For the update step they are $K_*^{\xi}, K_*^{\eta}, K_*^{\zeta}, M_*^{-1}$ and the matrices in equations (2.24) to (2.39).
Furthermore these matrices are independent of the quantities, parameters of the PDE and time. Therefore, they can be precomputed and reused. In case of the advection equation the matrices A, B and C are also constant, resulting in a constant time step Δt.
The prediction step in particular is well-suited for optimization as it is completely element-local. As most of the matrices in equations (2.22) are of size $(N+1)^4 \times (N+1)^4$ the prediction step is likely to require most of the computational effort for bigger N , i.e. higher order. Note that this is caused by the approach taken here and is not necessarily the case for a Cauchy-Kovalewski-based prediction step. It is also possible to reduce the number of degrees of freedom in time as shown in [10].

3.2 Matrix Structure

Another important observation is that all matrices involved are sparse matrices. While the implementation does not consider this due to time-constraints of this thesis, this is still a relevant feature of the approach. Sparsity patterns for the matrices for nodal basis functions are shown in [18].
Note that the involved matrices are sparse in lower dimensional cases as well. In the Figures 3.1 and 3.2 the sparsity patterns of the matrices involved in each iteration are depicted.
Ideally a sparse matrix-matrix multiplication kernel can be implemented to address this

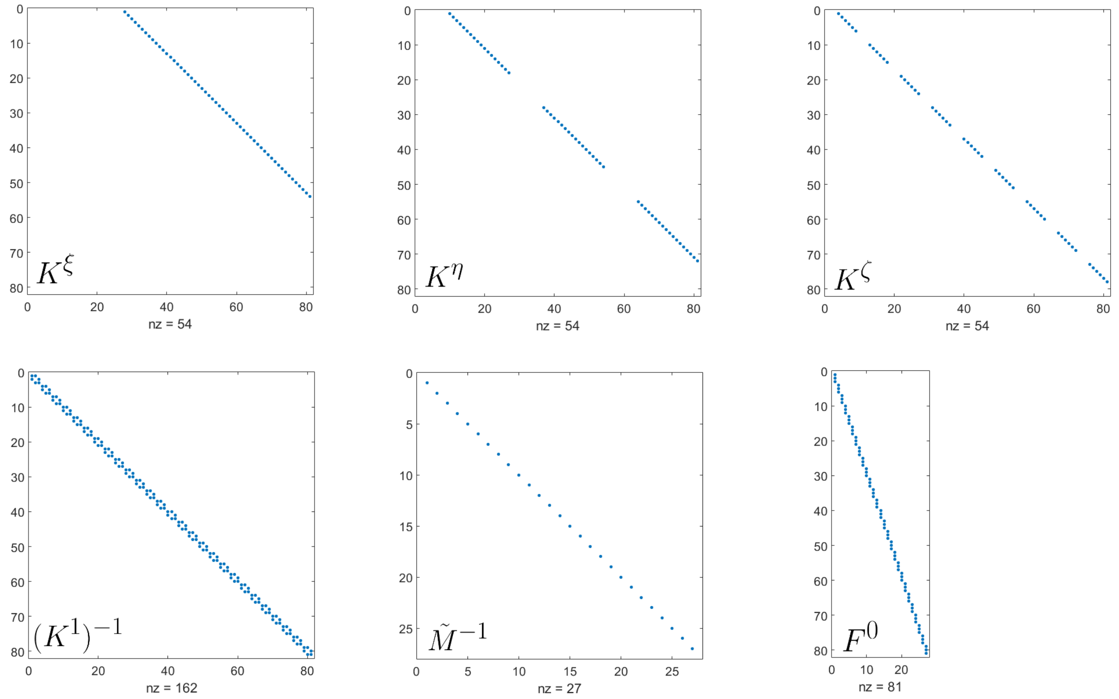


Figure 3.1: Sparsity patterns for the matrices $K^\xi, K^\eta, K^\zeta, (K^1)^{-1}, \tilde{M}^{-1}$ and F^0 for $N = 2$. Similar sparsity patterns occur for all tested N.

and additionally improve performance as in [6]. The memory limitations of the chosen computing system however will most likely not be affected by this to a large degree as the main impact on memory results from the size of $\hat{q}$, $\hat{f}$, $\hat{g}$, $\hat{h}$, $\hat{f}^*$, $\hat{g}^*$ and $\hat{h}^*$ as they contain $N^E \times (N+1)^4 \times N^{\text{Var}}$ entries.
To consider the impact of the sparsity patterns, Section 4 compares zero and non-zero entries of the involved matrices for all tested N.

3.3 Single Core Optimization

Considering the factors mentioned previously, standard procedures can be used to optimize the single core performance. We focus on the optimization for *Intel Xeon E5-2697 v3* CPUs supporting the *Advanced Vector Extensions* 2 (AVX2). General guidelines for this type of optimization can be found in [5].
First off, all arrays used to represent the matrices and variables in the implementation are 64-byte aligned in contiguous memory to allow for the efficient use of AVX2 intrinsics and the available memory bandwith. The contiguous aligned access helps to attain an optimal

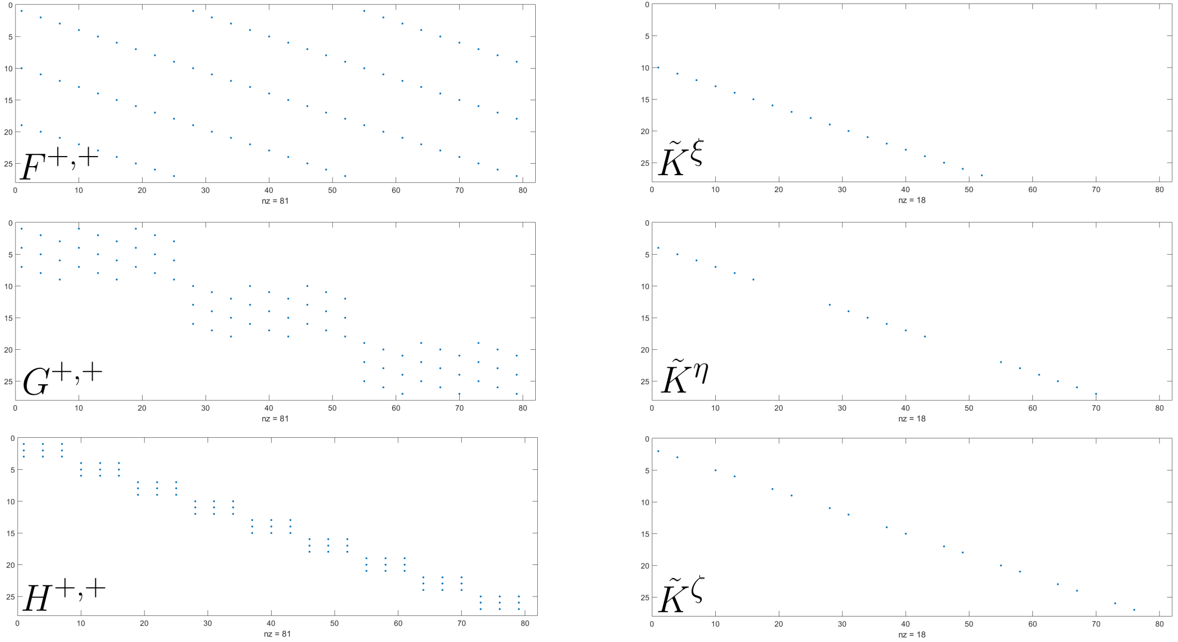


Figure 3.2: Sparsity patterns for the matrices $\tilde{K}^{\xi}, \tilde{K}^{\eta}, \tilde{K}^{\zeta}, F^{+,+}, G^{+,+}$ and $H^{+,+}$ for $N = 2$. Similar sparsity patterns occur for all tested N.

memory bandwith. It is also beneficial for the aforementioned intrinsics, a set of instructions that allows the automated use of the corresponding AVX2 registers and of fused multiply-add (FMA) operations. Thus, the so called *single instruction, multiple data* (SIMD) instructions on current Intel architectures can be efficiently utilized. All executables were compiled using the *Intel C++ Compiler 15.0*. To benefit from compiler-based optimization techniques, the flags for aggressive optimization $-O3$ and the *Interprocedural Optimization* (IPO) were set. Furthermore, the keyword *restrict* was used were applicable. Finally, AVX2 instructions support was enabled using the flag *-xCORE-AVX2*.
The activation of aggressive optimizations using $-O3$ allows the compiler to perform actions such as loop transformations or collapsing *if*-statements. Most importantly it enables auto-vectorization on the Intel compiler. However, the $-O3$ flag sometimes impairs the performance rather than aid it depending on the implementation.
Similar to the optimization with $-O3$, the IPO is automatic and gives the compiler additional information about things like, e.g., loop trip counts, data alignment or data dependencies. This way it is possible to auto-vectorize more loops or automatically inline functions. More information on the usage of this feature can be found in [2].
The *restrict* keyword, finally, asserts that memory referenced by a pointer is not accessed in any other way than in the current code scope. This disables runtime checks for aliasing and therefore needs to be used carefully.
We employ different kernels for the prediction step and the update step to allow for distinct

performance analysis of them. For comparison we tested multiple matrix-matrix multiplication implementations. One version using only the auto-vectorization features as baseline, one employing the *Intel Math Kernel Library's 11.2* (MKL) DGEMM implementation [3] and one utilizing LIBXSMM [4], a library optimized for small matrix-matrix multiplications. In case of LIBXSMM, this required including a file containing inlined assembly code, whereas the MKL was statically linked. Note that the baseline version does not feature specific optimizations like cache-blocking. Regarding the LIBXSMM kernels, one has to keep in mind that the library is optimized for matrix-matrix multiplications with small problem sizes, which limits the performance for higher order scenarios. Conversely the MKL is optimized for large problem sizes, i.e. it is not the best fit for the lower order scenarios.

3.4 Hybrid MPI/OpenMP Parallelization

In regards to parallelization a hybrid MPI/OpenMP parallelization with overlapping communication and computation was implemented. We use one rank for every node and one rank runs a number of OpenMP threads. Furthermore, asynchronous communication between the MPI ranks is employed to minimize waiting times and allow for minimal overhead. On each rank we pick one thread on one core that is dedicated to handling the rank's communication. A similiar approach was taken in [13].
We divide the spatial domain in equally sized subblocks in one, two, or three dimensions depending on the number of ranks. Each of these subblocks is divided into an inner, copy, and ghost layer as depicted in Figure 3.3, but in three dimensions. Therein the ghost layer is equivalent to the adjacent blocks' copy layer and analogously the copy layer to the adjacent blocks' ghost layer. The reason for this lies in the need to access neighbouring elements in the computation of the update step (2.41). This requires either sending the prediction step on the copy layer to adjacent blocks or computing the prediction on the ghost layer too.
In regards to the OpenMP parallelization, for both the prediction and update step, we split the rank's domain into as many subdomains as we have computation threads, i.e. threads apart from the one dedicated to communication. We pin the computation threads to all cores apart from the communication core in a such a fashion that without hyper-threading every core is running at most one thread and with hyper-threading, all cores are running at most two threads.
The communication thread persistently checks if the ongoing communication has been completed and, when it is appropiate, posts the necessary sends and receives for each rank. The most critical part here is to make sure that no computations are performed on the data that is still being sent and that no computation begins before all necessary data has been received. While this is certainly more complex and difficult to implement than a synchronous communication, where communication and computation do not overlap, it is advantageous in this case as most of the necessary computations do not rely on data from

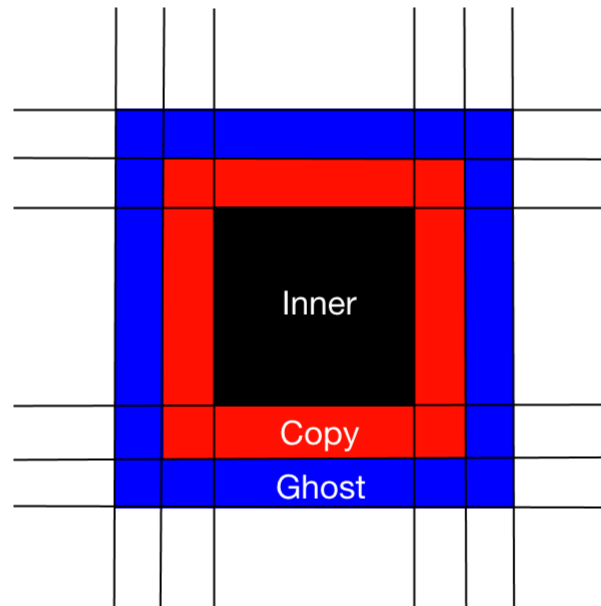


Figure 3.3: Two-dimensional depiction of the domain of one rank. The inner layer comprises all elements, which only require the inner and copy layer for the computation of the update step. The update step on the copy layer requires access to elements from all three layers. The ghost layer contains elements from the neighbouring ranks' copy layer. Each rank has up to six neighbours, depending on the total number of ranks.

other ranks. One disadvantage, however, is that the core handling the communication does not perform any computations and we sacrifice its computational power. On modern architectures this is increasingly sustainable as the trend for a higher number of cores per node persists.

Considering the view of one rank, one rank could perform an iteration, i.e. the computation of the prediction and update step on its inner and copy layer, as shown in Figure 3.4. In this approach it is necessary to send $4N^{\text{Var}}(N+1)^4$ values for each element in the copy layer. After posting the necessary receive for its ghost layer (step I), each rank first needs to be sure that there is no send on his part still ongoing, as in the next step each one computes the prediction step of his copy layer (step II). The copy layer now needs to be sent to the adjacent ranks (step III). Meanwhile, the prediction and update step of the inner layer is computed (steps IV and V). Following that, if the receive on the ghost layer has been completed the update step on the copy layer can be performed (step VI).

The advantage of this approach is that the time between receiving and sending can be used to perform the two computationally most expensive steps, the computation of the prediction and update step on the inner layer.

Alternatively, it is possible to proceed as shown in Figure 3.5. This approach relies on sending only $N^{\text{Var}}(N+1)^3$ values for each element. As in the first approach a receive for the

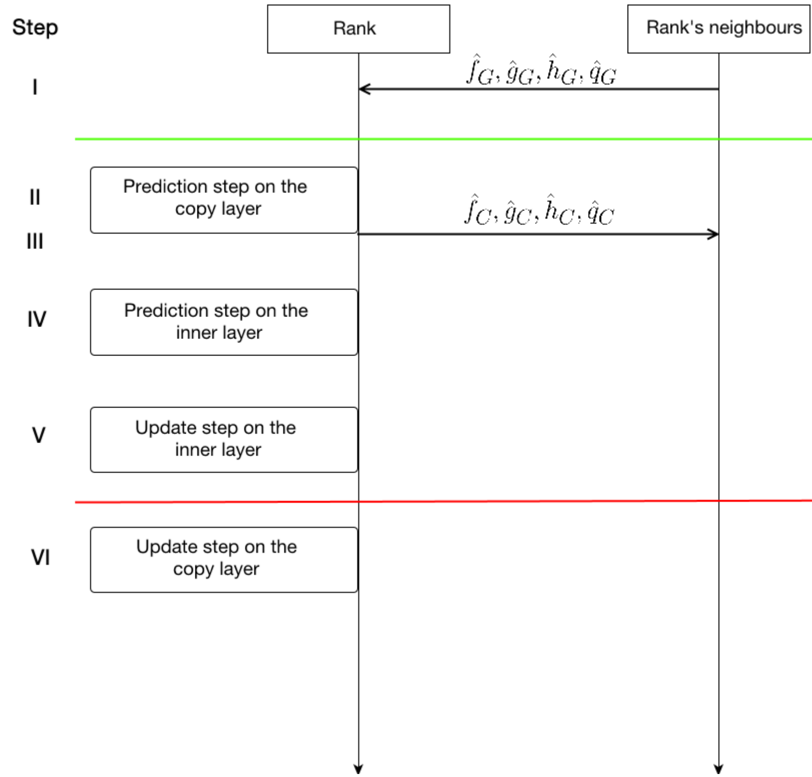


Figure 3.4: A visualization of the steps necessary to perform one iteration on one rank, if the prediction step on the ghost layer is performed on the neighbours. The green line indicates a MPI barrier for outgoing communication, the red one for incoming communication.Subscripts C and G denote elements in the copy and ghost layer respectively. One rank computes the prediction step on its copy layer and sends the results to its neighbours. Similarly it receives the prediction step on the ghost layer from its neighbours and can then perform the update step on its copy layer. The inner layer is independent of the ghost layer and prediction and update step on it can be computed asynchronously.

ghost layer is posted first (step I). After that the prediction step on the inner and copy layer is computed (step II). When the aforementioned receive completes, we now also compute the prediction step on the ghost layer (step III). If no sends are still ongoing, the update step on the copy layer is then computed and sent to the rank's neighbours (steps IV and V). Finally the update step on the inner layer is computed (step VI).
Similarly to the previous approach, during communication we compute the update step and prediction step on the inner layer and, additionally, the prediction step on the copy layer. Compared to the previous approach we also have significantly less values that need to be sent, especially for higher N. However, the computations of adjacent ranks overlap as each rank also computes the prediction step on its ghost layer.
We implemented the second approach, given that it provides us more communication-

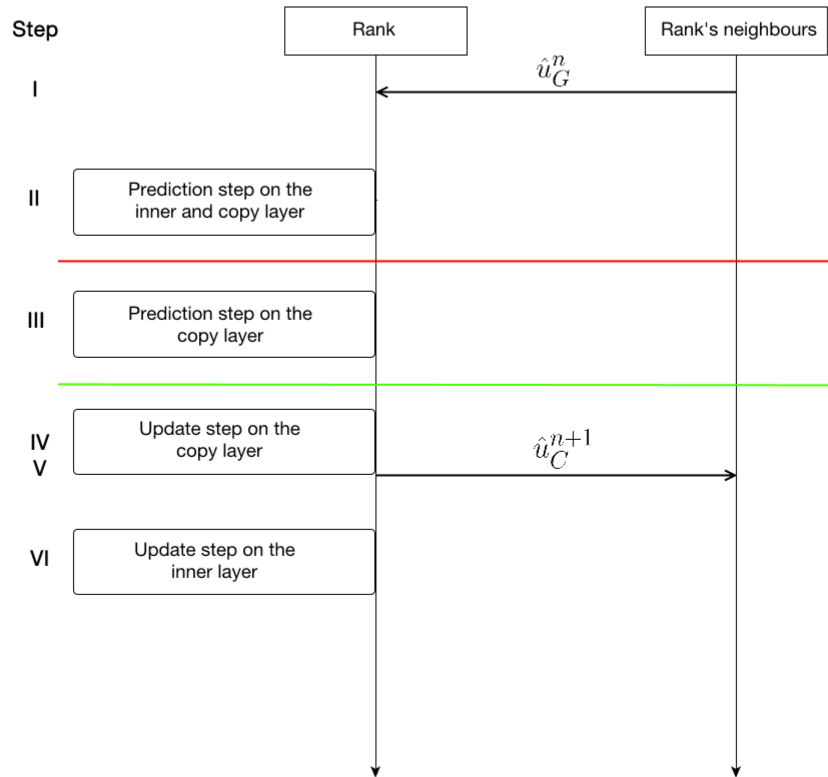


Figure 3.5: A visualization of the steps necessary to perform one iteration on one rank, if the prediction step on the ghost layer is performed on the rank. The green line indicates a MPI barrier for outgoing communication, the red one for incoming communication.Subscripts C and G denote elements in the copy and ghost layer respectively. One rank computes the prediction step on its copy and inner layer. It receives the neighbours $\hat{u}_G$ and then performs prediction step on the ghost layer. The inner layer is independent of the ghost layer and prediction and update step on it can be computed asynchronously. Note that before the start of the first iteration, the rank receives the initial $\hat{u}_G$ from its neighbours, which is not depicted in the image.

independent computations while needing less communication overall. Although, if we compare the computational overhead created by performing the prediction step on the ghost layer too, we perform the prediction on $6(N^i)^2$ elements twice, where N^i is the amount of elements per dimension. This is neglible compared to the $(N^i)^3$ elements on the inner and copy layer.

4 Results

This section features the results of different performance and error measurements of the implementation. After an introduction to the basic setup for all these measurements single-node and multi-node performance measurements are given.

4.1 Setup

All measurements were performed on the *Leibniz Rechenzentrum's CoolMUC2* [1] cluster that consists of 252 nodes, each featuring dual socket 14-core *Intel Xeon E5-2697 v3* CPUs, 64 GByte memory per node and an Infiniband FDR network connection. The maximum frequency of one core is set to 2600 Mhz. All measurements were performed with double-precision.
Test scenarios were chosen so that each rank's memory is sufficiently utilized and that a reasonable number of iterations is manageable. Results were averaged over multiple iterations. Test cases ranging from $N = 1$ to $N = 4$ using $N^i \in 150, 100, 60, 40$ elements per dimension respectively with a time step $\Delta t = \frac{1}{8}\frac{1}{(2N+1)}$. The A, B and C matrices were chosen to be diagonal matrices with only 1's on the diagonal, so that due to the periodic boundary conditions and the domain $[0; 1]^3$ at $t = 1$ the initial condition should be restored. As a sufficiently smooth initial condition is needed, we choose a three-dimensional Gaussian function at $t = 0$. As the matrices and basis functions from Section 2 are independent of the initial condition they were precomputed with a Matlab implementation. The initial conditions were created using the same code and below error measurements were taken in Matlab as the basis functions were not available in the C++ implementation.
As error function we choose a relative $L2$ error with

$$e_{\mathrm{L2}} = \frac{\sum_i (\tilde{u}_i - u_i)^2}{\sum_i \tilde{u}_i^2}, \tag{4.1}$$

where the $\tilde{u}_i$ are the exact solutions at the supporting points of all elements and the u_i the numerical ones at $t = 1$ respectively.
Figure 4.1 shows an error plot for the Matlab implementation. Unfortunately, the implementation does not provide stable results as one can see in the plot. One reason for this might be that due to the performance limitations of the Matlab implementation, only small test cases could be performed. E.g., the order of convergence for $N = 0$ for $N^i = 10$ and $N^i = 20$ is only 0.60, whereas for $N^i = 20$ and $N^i = 40$ it is 0.96, indicating that bigger test

[1] https://www.lrz.de/services/compute/linux-cluster/

cases might clarify the order of convergence. The order of convergence and corresponding tests are shown in Table 4.1.
The C++ implementation is identical to the Matlab implementation in regards to the im-

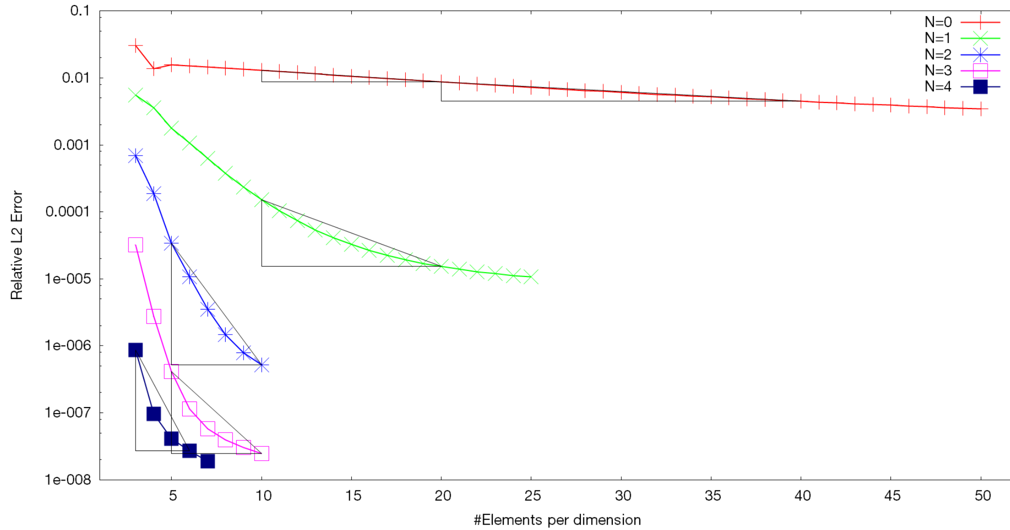


Figure 4.1: Convergence results ranging from $N = 0$ to $N = 4$ and $N^i = 3$ to $N^i = 40$ for a Matlab implementation. Black triangles depict the data points chosen to compute the order of convergence shown in 4.1. Bigger test cases were not feasible with the Matlab implementation.

plemented scheme, so that above measurements can represent both. All following results will be referring to the C++ implementation.
In case of the multi-node measurements, each rank contains a domain of size $[0;1]^3$ and initialized as it is described above to allow for simple weak-scaling tests. On each rank N^i elements per dimension comprise the inner and copy layer, whereas $6N^i$ comprise the ghost layer.
Note that with higher N due to the sparsity of the matrices involved, the reported hardware FLOPs are increasingly zero FLOPs. For $N = 1$ about 91% of the matrix entries in the matrices in (2.41) and (2.22) are zeroes. For $N = 2, N = 3$ and $N = 4$ about 97.7%, 99.08% and 99.57% are zero entries respectively.

4.2 Single-Node Performance

This section presents all results performed on only one node. The strong scaling performance measurements were all taken on one specific node of the cluster to avoid node-related performance variance. Results regarding OpenMP strong scaling, performance and memory bandwidth are given. With *Intel Turbo Boost* disabled, the theoretical peak performance of one core of the cluster is 41.6 GFLOPS and of one node therefore 1164.8

Table 4.1: Order of convergence for different N, corresponding to the values in Figure 4.1. Expected orders of convergence for N would be $N+1$. For $N=0$, $N=3$ and $N=4$ values are as expected.

N	N_1^i	N_2^i	e^1_{L2}	e^2_{L2}	Order of convergence
0	10	20	0.013057	0.0086972	0.60
0	20	40	0.0086972	0.0044869	0.96
1	10	20	0.00015316	$1.5272e-05$	3.33
2	5	10	$3.4006e-05$	$5.2299e-07$	6.02
3	5	10	$4.1187e-07$	$2.4835e-08$	4.05
4	3	6	$8.7062e-07$	$2.7041e-08$	5.01

GFLOPS. However, due to higher power requirements of AVX2 instructions, clock speeds are lower in practice as shown in [1].
Regarding all results, note that one of the 28 cores is the dedicated communication core. This could have been disabled for single-node tests, but to allow for easy comparison with the multi-node tests it was not.

4.2.1 Kernel Implementation

In Figure 4.2 we compare the different kernel implementations, i.e. auto-vectorized, LIBXSMM or MKL-based ones for different N. As expected the auto-vectorized version can not keep up with the other two. At $N=1$ and $N=2$ the LIBXSMM kernels are the fastest ones and for $N=3$ and $N=4$ the MKL kernels. Overall performance increases with higher N, which coincides with the results from [7]. A comparison of the time spent in the update and prediction kernel using the respectively fastest version is depicted in Figure 4.3 for different N. For $N=1$ the update step is more time-consuming as the difference in size between the matrices used in it compared to the bigger matrices in the prediction step is relatively small, e.g. the $\tilde{K}^\xi, \tilde{K}^\eta, \tilde{K}^\zeta$ as well as the flux matrices are of size 8×16, whereas the matrices in the prediction step are mostly of size 16×16. However, for $N=4$ this difference is decisive, as the prediction step matrices are now of size 625×625 where as the smaller ones in the update step are of size 125×625. Therefore, with higher N, the prediction step is increasingly computationally expensive.
For $N=1$ and using LIBXSMM kernels the achieved memory bandwidth on one node with 55 threads in the prediction step kernel was 41.25 GiB/s and 36.35 GiB/s in the update step kernel. Values for higher N decrease, indicating that our problem is compute- and not memory-bound as a memory bandwidth of 104 GiB/s was attained performing the STREAM [15] benchmark on one node.

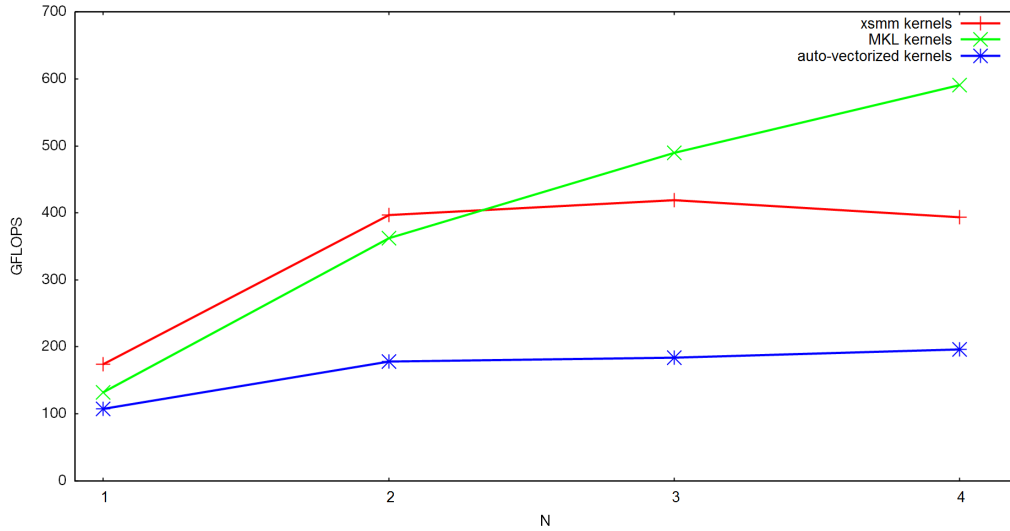


Figure 4.2: Comparison of the performance of the different implementations for N ranging from 1 to 4. For $N = 1$ and $N = 2$ the LIBXSMM kernels perform best at 174 and 391 GFLOPS respectively. For $N = 3$ and $N = 4$ the MKL kernels are faster at 490 and 591 GFLOPS respectively. The auto-vectorized version was slower than the library-based ones in all cases.

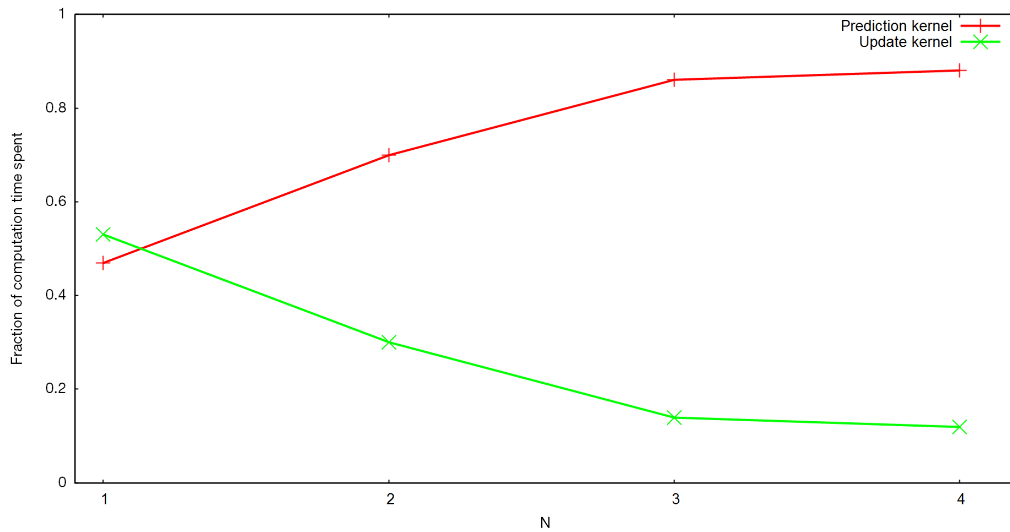


Figure 4.3: A depiction of the time spent computing the prediction and update step using the respectively fastest kernel implementation for cases with N ranging from 1 to 4. For $N = 1$ the update kernel requires more computation time, whereas with increasing N the prediction kernel becomes more time-consuming.

4.2.2 OpenMP Implementation

In Figure 4.4 performance measurements ranging from 2 to 55 OpenMP threads are shown,

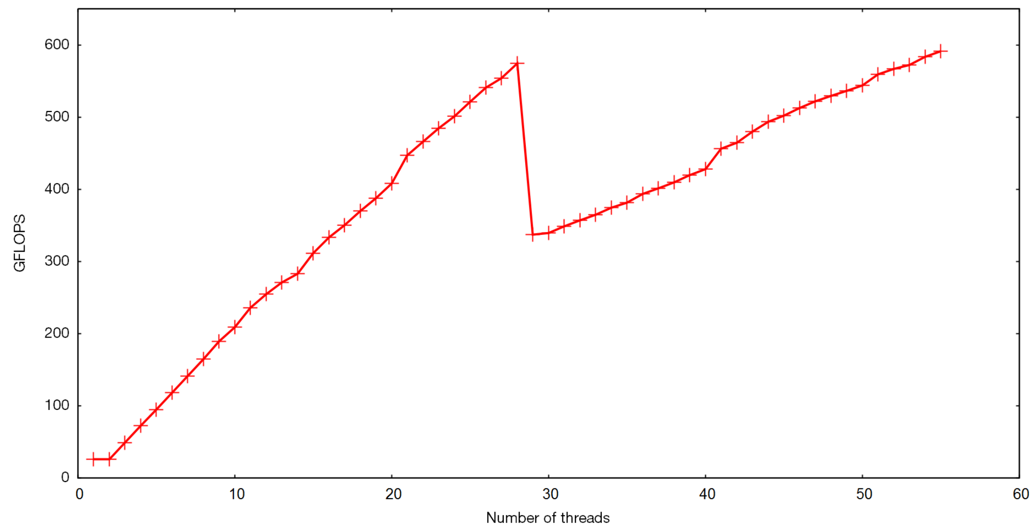


Figure 4.4: Performance measurements for $N = 4$ using the MKL kernels and 2 to 55 threads. For more than 28 threads hyper-threading is used,wgucg allows for a peak performance of 591 GFLOPS at 55 threads. Using two threads - one dedicated to communication and thereby not contributing on one node - a performance of 26 GFLOPS is achieved.

where one thread is a dedicated communication thread not contributing to the computation. This run was performed using the MKL kernels, $N = 4$, and $N^i = 40$. Thus the largest problem size tested is used for the OpenMP tests. The reason for this is that the highest order cases are most interesting and computationally demanding.
A single-core performance of 26 GFLOPS is achieved, i.e. 62% of the theoretical peak performance. For $N = 4$ the matrix sizes range from 125×125 to 625×625. It is discernible that for higher N performance might yet increase, as the MKL performs best in cases involving larger matrices. For 55 OpenMP threads a peak performance of 591 GFLOPS is reached, i.e. 50% of the theoretical peak performance.
Note that the drop in performance at 29 threads is caused by the fact that at this point hyper-threading is used on one core. The implementation uses static scheduling, therefore the performance is negatively affected, because the cores only running one thread are quicker than the one core running two and have to wait for it. This trend continues until 55 threads are reached, where every core, aside from the communication core, is running two threads.
Figure 4.5 shows the corresponding strong scaling. For 28 threads 22.02 times the performance of one computing thread is achieved, whereas for 55 threads it is 22.66. One might expect close to theoretically perfect scaling, but multiple factors affect the perfor-

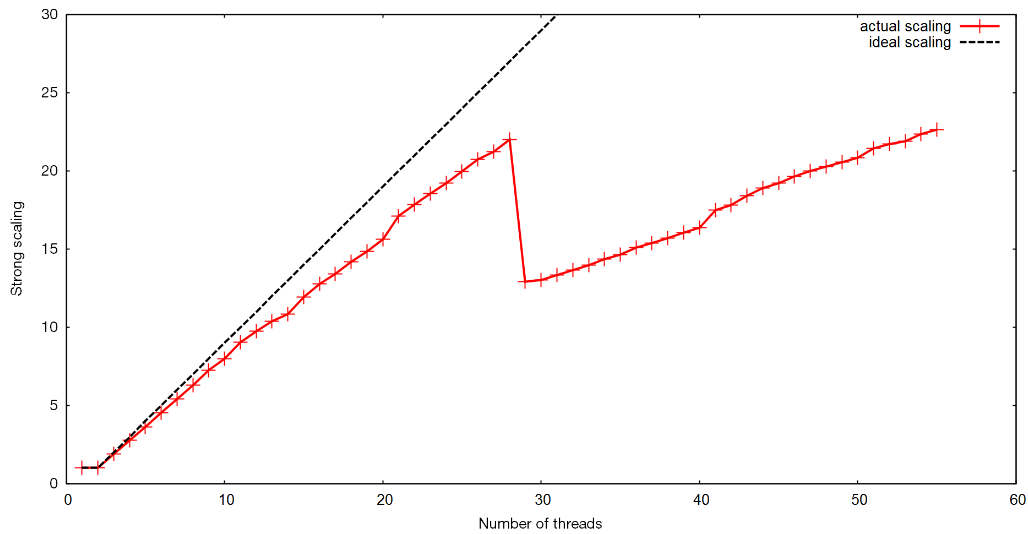


Figure 4.5: Strong scaling measurements for $N = 4$ using the MKL kernels and 2 to 55 threads. The dashed line indicates the values for perfect scaling. 22.02 times and 22.66 times the performance of one computing thread is achieved with 28 and 55 threads respectively.

mance. First-off the higher power consumption for AVX2 instructions mentioned before negatively affects the performance. In [1] a reduction of up to 10% of the core frequency for this CPU model is observed. Furthermore, the division of the computation, especially of the six faces in the ghost and copy layer, between the threads becomes more difficult as less elements are concerned.

4.3 Multi-Node Performance

In this section, the results using multiple nodes are presented. Unfortunately, it was not feasible to use the same nodes for each computational run, so some variance between results has to expected. Building on the single-node results, test cases with $N = 1$ and $N = 2$ were performed using the LIBXSMM kernels and $N = 3$ and $N = 4$ ones with the MKL-based kernels. Performance and scaling results for different rank counts are given and time spent in the computational kernels versus overall runtime is shown. In particular, weak scaling was analyzed, so that for, e.g., four ranks the domain size was four times as big.
In Figures 4.6,4.7,4.8 and 4.9 the performance and scaling measurements for $N \in [1; 4]$ are depicted. For $N = 1$ and $N = 2$ performance is limited by the single-node performance.

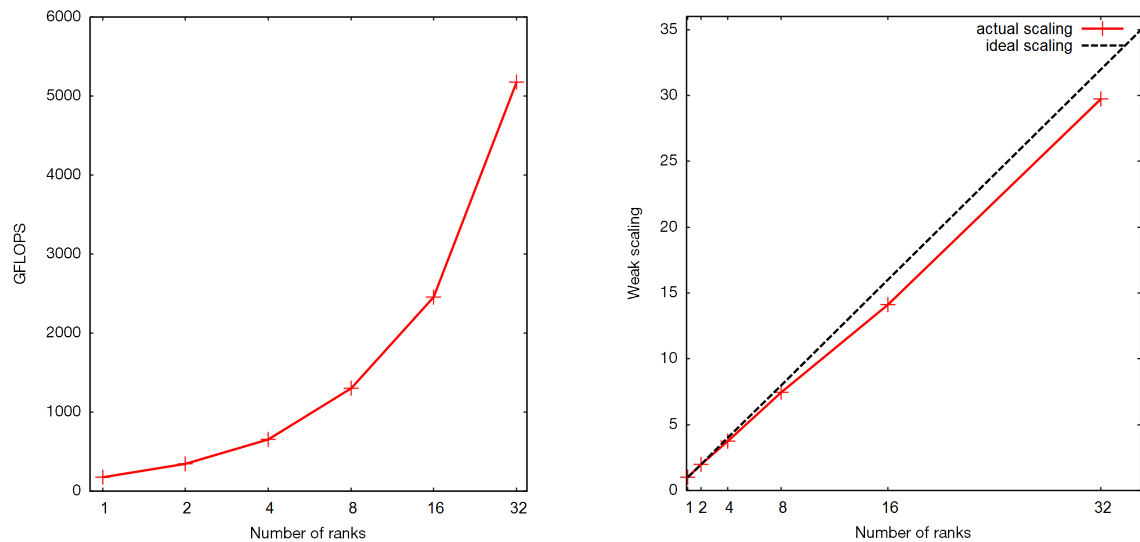


Figure 4.6: Performance results for $N = 1$ on 1 to 32 nodes using the LIBXSMM kernels on the left. A peak performance of 5175 GFLOPS on 32 nodes is achieved. Weak scaling results on the right. 29.76 times the speed of one rank is achieved on 32 ranks. The dashed line indicates the values for perfect scaling.

The maximum performance reached is 5175 GFLOPS and 10389 GFLOPS respectively. The implementation reached 29.76 and 26.16 times the speed of one rank on 32 ranks respectively. For $N = 3$ a peak performance of 23.5 TFLOPS on 64 nodes is reached, i.e. 32% of the theoretical peak performance. In the $N = 4$ run on 64 nodes 28.2 TFLOPS are achieved, i.e. 38% of the theoretical peak performance. For $N = 3$ and $N = 4$ the speedup was 47.94 and 49.72 with 64 ranks compared to one respectively. Unfortunately, for higher N the scaling is worse than for, e.g., $N = 1$. This is odd, since the computational cost increases by a higher order than the amount of data that has to be sent for bigger N, as the size of $\hat{u}$ depends on the factor $(N + 1)^3$, whereas many of the matrix-matrix multiplications in

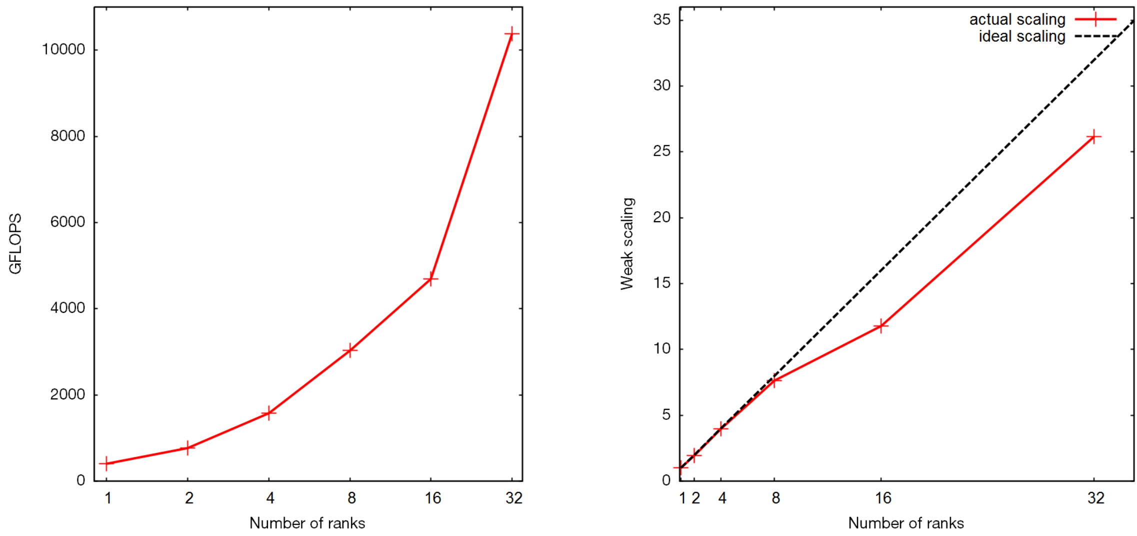


Figure 4.7: Performance results for $N = 2$ on 1 to 32 nodes using the LIBXSMM kernels on the left. A peak performance of 10389 GFLOPS on 32 nodes is achieved. Weak scaling results on the right. 26.16 times the speed of one rank is achieved on 32 ranks. The dashed line indicates the values for perfect scaling.

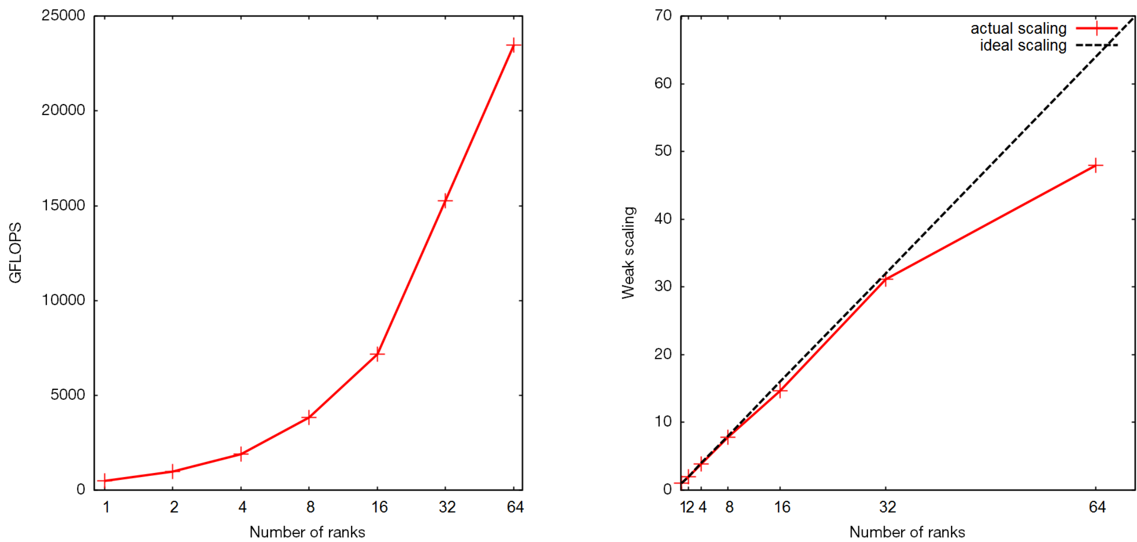


Figure 4.8: Performance results for $N = 3$ on 1 to 64 nodes using the MKL kernels on the left. A peak performance of 23478 GFLOPS on 64 nodes is achieved. Weak scaling results on the right. 47.94 times the speed of one rank is achieved on 64 ranks. The dashed line indicates the values for perfect scaling.

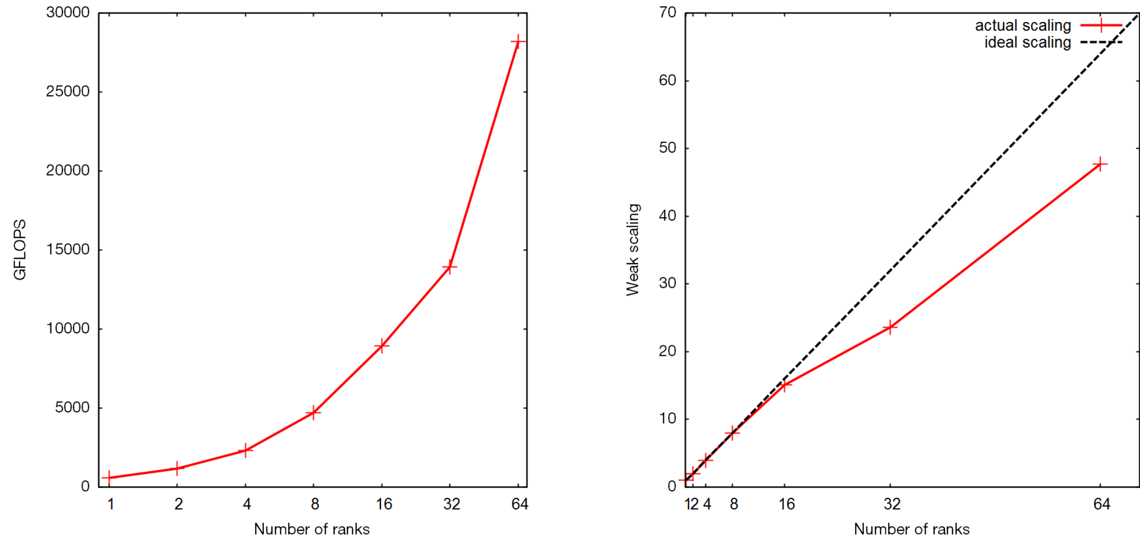


Figure 4.9: Performance results for $N = 4$ on 1 to 64 nodes using the MKL kernels on the left. A peak performance of 28214 GFLOPS on 64 nodes is achieved. Weak scaling results on the right. 47.72 times the speed of one rank is achieved on 64 ranks. The dashed line indicates the values for perfect scaling.

(2.22) involve matrices of size $(N+1)^4 \times (N+1)^4$. One reason might be the actual topology of the selected nodes in the runs or differences in the computational speed achieved by specific nodes. A better load balancing approach might be necessary in that case.
In Figure 4.10 the fraction of the time spent in the kernels compared with the overall runtime per rank is depicted for $N = 1$. A version implementing a simple synchronous communication using the *MPI_Sendrecv* command was tested too. The synchronous version ran with 54 threads so that both versions had the same amount of threads available for the compute kernels. The overall runtimes for the asynchronous implementation were 15.82, 16.70 and 15.87 seconds for 8, 16 and 32 ranks respectively, while the synchronous one required 15.03, 19.44 and 18.30 seconds. Note that the synchronous version spends a greater fraction of time outside of the kernels for a greater number of ranks, while the asynchronous version does not. As more ranks are involved more communication is required and as each communication has to be completed before the computation can continue in the synchronous case. The performance is negatively affected by this. Overall the asynchronous approach provides better scalability, even if it sacrifices $\frac{1}{28}$ of the performance due to the dedicated communication core.

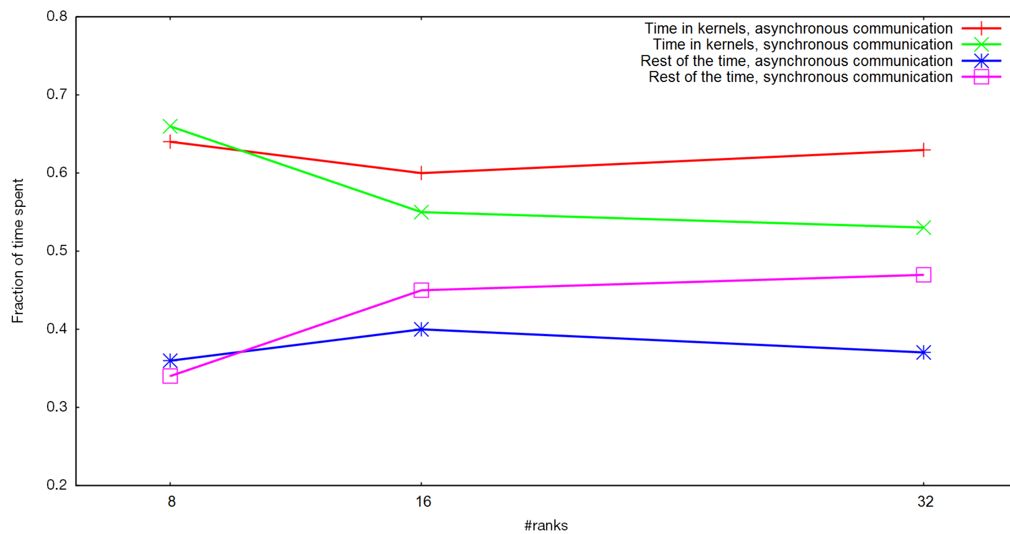


Figure 4.10: A comparison of the fraction of the time spent in the kernels and the overall runtime for an implementation using the synchronous *MPI_Sendrecv* command and the asynchronous implementation. Both implementations utilized 54 threads in the kernels. The overall runtimes for the asynchronous implementation were 15.82, 16.70 and 15.87 seconds for 8, 16 and 32 ranks respectively, while the synchronous one required 15.03, 19.44 and 18.30 seconds. For an increasing number of nodes the synchronous version spends less time in the computation kernels, whereas no significant change occurs in the asynchronous implementation. Overall runtimes are comparable, especially as the slightly slower synchronous implementation could utilize two more threads.

5 Conclusion and Outlook

Overall the results of the implementation coincide with [6][8][7] in that ADER-DG approaches are well-suited to achieve good performance on current supercomputer architectures. The pitfalls of Runge Kutta methods for higher orders do not apply. Furthermore the element-local features of the ADER-DG approach allow an easy parallelization. The fact that the involved matrices are identical for all elements saves memory and simplifies the implementation.
On a single core 62% of the theoretical peak performance are achieved, on one node 50%. This decline is partially attributable to the higher power demands of AVX2 instructions, but also points to some issues in the OpenMP parallelization. We reach 38% of the theoretical peak performance on 64 nodes for $N = 4$. The decline in this case might require further investigation. A comparison between a synchronous and asynchronous communication approach showed better scaling results for the asynchronous one.
There is a number of things relevant to future research. First-off the implementation or use of sparse matrix-matrix multiplication kernels should be considered to drive runtime-oriented optimization and to allow for realistic runtime measurements. While a variety of libraries exists to perform such operations, it might be advantageous to implement kernels specifically designed with the structure of the sparsity patterns from Section 3 in mind.
It is necessary to confirm the order of convergence with bigger test cases. Also, there is still room for optimizations in the implementation. The scaling issues with the OpenMP implementation should be addressed as well as the similar problems with the MPI implementation.
Furthermore the extension to more complex problems like the elastic wave equation or the Euler equations is desirable to furthermore confirm the practical usability of this method and to be able to use it to compute solutions to problems that are not analytically solvable. One problem could be in the applicability of the modal basis functions to nonlinear problems. However, nodal basis functions as in [16][12] could be used here as well. Finally the implementation of limiters, as in [20], to deal with more complex initial conditions and the use of a more flexible discretization like, e.g., tetrahedons and adaptive element sizes are of interest to be able to model complex geometries.

Bibliography

[1] Detailed specifications of the intel® xeon e5-2600v3 haswell-ep processors. https://www.microway.com/knowledge-center-articles/detailed-specifications-intel-xeon-e5-2600v3-haswell-ep-processors/. Accessed: 2015-10-12.

[2] A guide to auto-vectorization with intel® C++ compiler. https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers. Accessed: 2015-10-12.

[3] Intel® math kernel library. https://software.intel.com/en-us/intel-mkl. Accessed: 2015-10-12.

[4] Libxsmm. https://github.com/hfp/libxsmm. Accessed: 2015-10-12.

[5] Step by step performance optimization with intel® C++ compiler. https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler. Accessed: 2015-10-12.

[6] Alexander Breuer, Alexander Heinecke, Michael Bader, and Christian Pelties. Accelerating seissol by generating vectorized code for sparse matrix operators. In *Parallel Computing - Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 347–356. IOS Press, 2014.

[7] Alexander Breuer, Alexander Heinecke, Leonhard Rannabauer, and Michael Bader. High-order ader-dg minimizes energy- and time-to-solution of seissol. In *High Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 340–357. Springer International Publishing, 2015.

[8] Alexander Breuer, Alexander Heinecke, Sebastian Rettenberger, Michael Bader, Alice-Agnes Gabriel, and Christian Pelties. Sustained petascale performance of seismic simulations with seissol on supermuc. In *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 1–18. Springer International Publishing, 2014.

[9] J. C. Butcher. On fifth and sixth order explicit runge-kutta methods: order conditions and order barriers. *Canadian Applied Mathematics Quarterly*, 17(3), 2009.

[10] Michael Dumbser, Dinshaw S. Balsara, Eleuterio F. Toro, and Claus-Dieter Munz. A unified framework for the construction of one-step finite volume and discontinuous galerkin schemes on unstructured meshes. *Journal of Computational Physics*, 227(18):8209 – 8253, 2008.

[11] Michael Dumbser and Olindo Zanotti. Very high order pnpm schemes on unstructured meshes for the resistive relativistic mhd equations. *J. Comput. Phys.*, 228(18):6991–7006, October 2009.

[12] Michael Dumbser, Olindo Zanotti, Arturo Hidalgo, and Dinshaw S. Balsara. Aderweno finite volume schemes with spacetime adaptive mesh refinement. *Journal of Computational Physics*, 248:257 – 286, 2013.

[13] Schubert G. Schoenemeyer T. Wellein G. Hager, G. Prospects for truly asynchronous communication with pure mpi and hybrid mpi/openmp on current supercomputing platforms. *Cray Users Group Conference 2011, Fairbanks, AK, USA*, 2011.

[14] J.S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Texts in Applied Mathematics. Springer New York, 2007.

[15] Ph.D. John D. McCalpin. Stream benchmark. `https://www.cs.virginia.edu/stream/`. Accessed: 2015-10-12.

[16] Martin Käser and Michael Dumbser. An arbitrary high-order discontinuous galerkin method for elastic waves on unstructured meshes i. the two-dimensional isotropic case with external source terms. *Geophysical Journal International*, 166(2):855–877, 2006.

[17] R.J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002.

[18] Leonhard Andreas Rannabauer. Runge-kutta and ader discontinuous galerkin schemes for hyperbolic partial differential equations. Bachelor's Thesis, Technische Universität München, 2014.

[19] V.A. Titarev and E.F. Toro. Ader: Arbitrary high order godunov approach. *Journal of Scientific Computing*, 17(1-4):609–618, 2002.

[20] O. Zanotti, F. Fambri, M. Dumbser, and A. Hidalgo. Space-time adaptive ADER discontinuous Galerkin finite element schemes with a posteriori sub-cell finite volume limiting. *ArXiv e-prints*, November 2014.