hyper.deal: An efficient, matrix-free finite-element library for high-dimensional partial differential equations*

Peter Munch[†‡] Katharina Kormann[§] Martin Kronbichler[†]

February 20, 2020

Abstract

This work presents the efficient, matrix-free finite-element library `hyper.deal` for solving partial differential equations in two to six dimensions with high-order discontinuous Galerkin methods. It builds upon the low-dimensional finite-element library `deal.II` to create complex low-dimensional meshes and to operate on them individually. These meshes are combined via a tensor product on the fly and the library provides new special-purpose highly optimized matrix-free functions exploiting domain decomposition as well as shared memory via `MPI-3.0` features. Both node-level performance analyses and strong/weak-scaling studies on up to 147,456 CPU cores confirm the efficiency of the implementation. Results of the library `hyper.deal` are reported for high-dimensional advection problems and for the solution of the Vlasov–Poisson equation in up to 6D phase space.

Key words. matrix-free operator evaluation, discontinuous Galerkin methods, high-dimensional, high-order, Vlasov–Poisson equation, MPI-3.0 shared memory.

1 Introduction

Three-dimensional problems are today simulated in great detail based on domain decomposition codes up to supercomputer scale. With the increase in computational power, it becomes feasible to simulate also higher-dimensional problems. With this contribution, we target the case of moderately high-dimensional ($\leq$6D) problems with complex geometry requirements in some of the dimensions. Our primary target are kinetic equations that describe the evolution of a distribution function in phase space. Such Boltzmann-type equations are for instance used in magnetic confinement fusion where the evolution of a plasma is described by a distribution function that evolves according to the Vlasov equation coupled to a system of Maxwell's equations for its self-consistent fields. Other areas of application of phase space are, e.g., cosmic microwave background radiation or magnetic reconnection in the earth's magnetosphere.

In these applications, there are two distinct sets of variables, configuration and velocity space. Especially in configuration space, the domain can be geometrically complex (e.g., torus-like shapes as in the case of tokamak and stellarator fusion reactors), necessitating a flexible description of the mesh as is commonly provided by finite-element method (FEM) libraries for grids in up to three dimensions. Another typical feature of kinetic equations is the coupling of high-dimensional problems in phase space to lower-dimensional problems in the spatial variables only. With the library

*This work was supported by the German Research Foundation (DFG) under the project "High-order discontinuous Galerkin for the exa-scale" (ExaDG) within the priority program "Software for Exascale Computing" (SPPEXA), grant agreement no. KO5206/1-1 and KR4661/2-1. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (LRZ, `www.lrz.de`) through project id pr83te.

[†]Institute for Computational Mechanics, Technical University of Munich, Boltzmannstr. 15, 85748 Garching, Germany ({`munch,kronbichler`}`@lnm.mw.tum.de`).

[‡]Institute of Materials Research, Materials Mechanics, Helmholtz-Zentrum Geesthacht, Max-Planck-Str. 1, 21502 Geesthacht, Germany (`peter.muench@hzg.de`).

[§]Max Planck Institute for Plasma Physics, Boltzmannstr. 2, 85748 Garching, Germany, and Department of Mathematics, Technical University of Munich, Boltzmannstr. 3, 85748 Garching, Germany, (`katharina.kormann@ipp.mpg.de`).

`hyper.deal`, we are targeting these applications: For a phase-space simulation, two separate—possibly complex—meshes describing configuration and velocity space are defined based on the capabilities of a base low-dimensional finite-element library—in our case the library `deal.II` [3; 5]—and combined by taking their tensor product on the fly. The equations that we have in mind are advection-dominated, which is why we mostly focus on the discretization of the advection equation in this paper.

The general concept presented is not limited to the description of the phase space. Possible applications of a high-dimensional FEM library could for instance include three-dimensional problems that involve a low-dimensional parameter space or low-dimensional Fokker–Planck-type equations.

1.1 Previous work

While the decomposition of complex domains in up to three dimensions is a well-studied problem, partial differential equations on complex domains in dimensions higher than three are not tackled by generic FEM libraries to date. Some libraries have started to extend their capabilities to higher dimensions on structured grids. An example is the `YaspGrid` module of the finite-element library `DUNE`, which implements structured grids in arbitrary dimensions [8]. Also, higher-dimensional problems are solved based on sparse grids [15], reduced basis methods, or low-rank tensors [6]. On the other hand, in the plasma community some specialized codes exist that target gyrokinetic or kinetic equations. The `Gekyll` code [16] is a discontinuous Galerkin-method solver for plasma physics applications, and a fully kinetic version for Cartesian grids was presented in [18]. The authors use Serendipity elements to reduce the number of degrees of freedom. On the other hand, such a reduced basis does not form a tensor product and is therefore not amenable to the algorithms presented in this work. The specifics of domain decomposition in higher dimensions have only been studied recently. In [22], a six-dimensional domain decomposition for a semi-Lagrangian solver on a Cartesian grid of the Vlasov–Poisson system was studied, highlighting the high demands on memory transfer between neighboring processes due to the increased surface-to-volume ratio with increasing dimensionality. The parallelization of a similar algorithm has also been addressed in [34]. However, the domain decomposition is limited to configuration space in that work, which poses a strong limit to the scalability of the implementation.

1.2 Model problem

The Vlasov equation is given as

$$\frac{\partial}{\partial t} f_s(t,\vec{x},\vec{v}) + \vec{v}\cdot\nabla_{\vec{x}} f_s(t,\vec{x},\vec{v}) + \frac{q_s}{m_s}\left(\vec{E}(t,\vec{x}) + \vec{v}\times\vec{B}(t,\vec{x})\right)\cdot\nabla_{\vec{v}} f_s(t,\vec{x},\vec{v}) = 0, \tag{1}$$

where $f_s(t,\vec{x},\vec{v})$ denotes the probability density of a particle species s with charge q_s and mass m_s as a function of the phase space, $\vec{E}$ the electric field, and $\vec{B}$ the magnetic field. The phase space is defined as the tensor product of configuration space, $\vec{x}\in\Omega_{\vec{x}}\subset\mathbb{R}^{d_{\vec{x}}}$, and velocity space, $\vec{v}\in\mathbb{R}^{d_{\vec{v}}}$. This equation is coupled either to the Maxwell or to the Poisson equation (see Section 6) for the self-consistent fields. If we define the gradient operator as $\nabla^\top := (\nabla_{\vec{x}}^\top,\ \nabla_{\vec{v}}^\top)$, Equation (1) can be rewritten as

$$\frac{\partial}{\partial t} f_s(t,\vec{x},\vec{v}) + \nabla\cdot\left(\begin{pmatrix} \vec{v} \\ \frac{q_s}{m_s}\left(\vec{E}(t,\vec{x}) + \vec{v}\times\vec{B}(t,\vec{x})\right) \end{pmatrix} f_s(t,\vec{x},\vec{v})\right) = 0. \tag{2}$$

This is a non-linear, high-dimensional ($d = d_{\vec{x}} + d_{\vec{v}}$), hyperbolic partial differential equation. To simplify the presentation in the following, we will consider the advection equation defined on the arbitrary d-dimensional domain Ω, denoting the independent variable again by $\vec{x}$,

$$\frac{\partial f}{\partial t} + \nabla\cdot(\vec{a}(f,\,t,\,\vec{x})f) = 0 \qquad \text{on} \qquad \Omega\times[0,t_{\text{final}}], \tag{3}$$

with $\vec{a}(f,\,t,\,\vec{x})$ being the solution-, time-, and space-dependent advection coefficient. The system is closed by an initial condition $f(0,\vec{x})$ and suitable boundary conditions. In the following, we concentrate on periodic boundary conditions. We return to the Vlasov–Poisson equation in Section 6 where we combine an advection solver from the library `hyper.deal` and a Poisson solver based on `deal.II`.

1.3 Discontinuous Galerkin discretization of the advection equation

High-order discontinuous Galerkin methods are attractive methods for solving hyperbolic partial differential equations like Equation (3) due to their high accuracy in terms of dispersion and dissipation, while maintaining geometric flexibility through unstructured grids. The discontinuous Galerkin (DG) discretization of these equations can be derived by the following steps: The partial differential equation is multiplied with the test function g, integrated over the element domain $\Omega^{(e)}$ with $\biguplus_e \Omega^{(e)} = \Omega$, and the divergence theorem is applied:

$$\left(g, \frac{\partial f}{\partial t}\right)_{\Omega^{(e)}} - \left(\nabla g, \vec{a} f\right)_{\Omega^{(e)}} + \left\langle g,\, \vec{n} \cdot (\vec{a} f)^* \right\rangle_{\Gamma^{(e)}} = 0, \tag{4}$$

with $(\vec{a} f)^*$ being the numerical flux, like a central or an upwind flux. Integration $\int_\Omega \mathrm{d}\Omega$ and derivation ∇ are not performed in the real space but in the reference space $\Omega_0^{(e)}$ and $\Gamma_0^{(e)}$ and require a mapping to the reference coordinates:

$$\left(g, |\mathcal{J}| \frac{\partial f}{\partial t}\right)_{\Omega_0^{(e)}} - \left(\mathcal{J}^{-T} \nabla_{\vec{\xi}} g,\ |\mathcal{J}| \vec{a} f\right)_{\Omega_0^{(e)}} + \left\langle g,\, |\mathcal{J}|\, \vec{n} \cdot (\vec{a} f)^* \right\rangle_{\Gamma_0^{(e)}} = 0, \tag{5}$$

where $\mathcal{J}$ is the Jacobian matrix of the mapping from reference to real space and $|\mathcal{J}|$ its determinant.

To discretize this equation in space, we consider nodal polynomials with nodes in the Gauss–Lobatto points. The integrals are evaluated numerically by weighted sums. We consider both the usual Gauss(–Legendre) quadrature rules and the integration directly in the Gauss–Lobatto points without the need of interpolation (collocation setup [13]). The resulting semi-discrete system has the following form:

$$\mathcal{M} \frac{\partial \vec{f}}{\partial t} = \mathcal{A}(\vec{f},\, t) \qquad \leftrightarrow \qquad \frac{\partial \vec{f}}{\partial t} = \mathcal{M}^{-1} \mathcal{A}(\vec{f},\, t), \tag{6}$$

where $\vec{f}$ is the vector containing the coefficients for the polynomial approximation of f, $\mathcal{M}$ the mass matrix, and $\mathcal{A}$ the discrete advection operator. This system of ordinary differential equations can be solved with classical time integration schemes such as explicit Runge–Kutta methods. They require the right-hand side $\mathcal{M}^{-1}\mathcal{A}(\vec{f},\, t)$ to be evaluated efficiently. The particular structure of the mass matrix $\mathcal{M}$ should be noted: It is diagonal in the collocation case and block-diagonal in the case of the normal Gauss quadrature, which leads to a simple inversion of the mass matrix [27].

For 2D and 3D high-order DG methods, efficient matrix-free operator evaluations [24; 25] for individual operators $\mathcal{M}^{-1}$, $\mathcal{A}$ as well as for the merged operator $\mathcal{M}^{-1}\mathcal{A}$ are common in the context of fluid mechanics [23], structural mechanics [11], and acoustic wave propagation [32]. These methods do not assemble matrices, but evaluate the effect of the (linear or non-linear) operator on element vectors on the fly, often exploiting the tensor-product structure of the shape functions on hexahedron meshes. Since generally the access to main memory is the major limiting factor on modern CPU hardware, the fact that no matrix has to be loaded for matrix-free operator evaluation directly translates into a significant performance improvement. Discontinuous Galerkin methods and matrix-free operator evaluation kernels are part of many low-dimensional general-purpose FEM libraries nowadays.

1.4 Our contribution

This work shows that an extension of low-dimensional general-purpose high-order matrix-free FEM libraries to high dimensions is possible. We discuss how to cope with possible performance deteriorations due to the "curse of dimensionality" and present special-purpose concepts exploiting the structure of the phase space, using domain decomposition and shared memory, all taking hardware characteristics into account.

All concepts that we describe in this work have been implemented and are available under the LGPL 3.0 license as the library `hyper.deal` hosted at `https://github.com/hyperdeal/hyperdeal`. It extends—as an example—the open-source FEM library `deal.II` [3] to high dimensions. Both node-level performance and strong/weak-scaling analyses, conducted for this library,

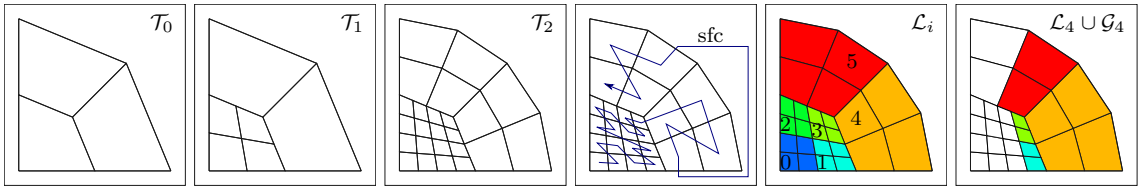


Figure 1: Generation of a low-dimensional mesh in the base library `deal.II`: a) coarse grid; b-c) refinement of a coarse grid; d) enumeration of cells along a space-filling curve (sfc); e) partitioning of the space-filling curve; f) local view of process 4 (showing only active local cells and active ghost cells)

confirm the suitability of the described concepts for solving partial differential equations in high dimensions.

The remainder of this work is organized as follows. In Section 2, we present the concepts of a generic modern low-dimensional finite-element library; special focus is put on mesh generation, parallelization, and efficient matrix-free operator evaluation as well as on why these concepts are only partly applicable for higher-dimensional problems. Sections 3-4 give an introduction into the concept of a tensor product of partitioned low-dimensional meshes and details its implementation for phase space. Section 5 presents performance results for the advection equation, confirming the efficiency of the design decisions made during the implementation. Section 6 explains how `hyper.deal` can efficiently be combined with a `deal.II`-based Poisson solver and shows scaling results for a benchmark problem from plasma physics. Finally, Section 7 summarizes our conclusions and points to further research directions.

2 Matrix-free finite-element algorithms

In the following, we describe building blocks commonly used in modern general-purpose FEM libraries with a special focus on triangulations, parallelization, and efficient matrix-free methods.

The concept of this work is generic and could in principle be built upon any general-purpose FEM library such as MFEM [4], DUNE [12], FEniCS [2], or Firedrake [9]. Our description is, however, specialized to the implementation in `hyper.deal` that is constructed on top of the `deal.II` library and uses some of naming conventions in that project.

2.1 Triangulation

A triangulation $\mathcal{T}$ covers the computational domain Ω and is defined by a set of point coordinates $\mathcal{P}$ as well as by a set of cells $\mathcal{C}$. Each cell $c \in \mathcal{C}$ consists of a fixed number of vertices. Since we consider unstructured quad/hex-only meshes, cells may only consist of 2, 4, or 8 vertices.

The triangulation $\mathcal{T}$ can be the result of an external mesh generator or of the repeated refinement of cells in a coarse mesh $\mathcal{T}_0$ in an octree manner (see Figures 1a-c). The latter way to create the actual triangulation fits into the concept of adaptive mesh refinement and geometric multigrid methods (used in Section 6 for the Poisson problem), since the refinement levels can also be used as multigrid levels. The cells in the mesh are continuously enumerated, which leads to a space-filling curve (see also Figure 1d).

2.2 Parallelization: partitioning

Distributed memory parallelization of finite-element codes is generally implemented based on a domain decomposition via the Message Passing Interface (MPI). For this purpose, the space-filling curve of cells mentioned above can be partitioned—uniformly—among all processes within an MPI communicator (generally MPI_COMM_WORLD) so that a process i is only responsible for a subset of (locally owned) cells $\biguplus_i \mathcal{L}_i = \mathcal{C}$.

For the management of large-scale distributed meshes, libraries based on space-filling curves like `p4est` [7; 10], which uses the Morton order/z-curves, or `Peano` [35], which uses Peano curves,

and graph-partitioning algorithms as implemented in `Metis` [19] and `Zoltan` [17] are utilized.

Besides locally owned cells $\mathcal{L}_i$, each process has a halo of ghost cells $\mathcal{G}_i \subseteq \mathcal{C}$ with $\forall i : \mathcal{L}_i \cap \mathcal{G}_i = \emptyset$. Regularly, the degrees of freedom in these ghost cells have to be exchanged between neighbors, e.g., to evaluate fluxes in DG methods, requiring communication (see also Subsection 2.3).

Cells owned by a process can be processed in parallel by threads. They can exploit shared memory, however, they have to be synchronized to prevent race conditions. For this purpose, the libraries `TBB` and `OpenMP` are integrated in the `deal.II` library. In Subsection 4, we present a novel hybrid parallelization scheme, which exploits the shared-memory capabilities of MPI-3.0 with the advantage that no threads have to be created explicitly.

As the last type of parallelization, explicit vectorization for SIMD (single instruction stream, multiple data streams) can be used. A detailed explanation of SIMD in context of matrix-free methods follows in Subsection 2.6.

2.3 Elements and degrees of freedom

Unknowns are assigned for all cells $c \in \mathcal{C}$. We use d-dimensional scalar, discontinuous tensor-product shape functions of polynomial degree k, based on Gauss-Lobatto support points (see also Figure 2):

$$\mathcal{P}_k^d = \underbrace{\mathcal{P}_k^1 \otimes ... \otimes \mathcal{P}_k^1}_{\times d}. \tag{7}$$

In DG, unknowns are not shared between cells so that each cell holds $(k+1)^d$ unknowns. The total number of degrees of freedom (DoF) is:

$$N = |\mathcal{C}| \cdot (k+1)^d. \tag{8}$$

The unknowns are coupled via fluxes of DG schemes (see also Section 1). To be able to compute the fluxes, the degrees of freedom of neighboring cells have to be accessed. The dependency region for computing all contributions of a cell is in the case of advection:

$$\underbrace{(k+1)^d}_{\text{cell}} + \underbrace{2 \cdot d \cdot (k+1)^{d-1}}_{\text{faces}}, \tag{9}$$

i.e., the union of all unknowns of the cell and the unknowns residing on faces of the $2 \cdot d$ neighboring cells due to the nodal polynomials, which are combined with nodes at the boundary (see Figure 2).

2.4 Parallelization: communication

Due to the selection of nodal polynomials with nodes at the element faces, the amount of data (in doubles) to be transferred during ghost-value exchanges can be estimated as follows

$$|\mathcal{L}_i'|^{\frac{d-1}{d}} \leq \frac{(\texttt{data amount})_i}{2 \cdot d \cdot (k+1)^{d-1}} \leq |\mathcal{L}_i|, \quad i = 0, \ldots, p-1, \tag{10}$$

where p is the number of partitions. The lower bound is given by the data to be transferred on an idealized hypercube partition with $|\mathcal{L}_i'| \approx \hat{N}/(k+1)^d$ cells, where $\hat{N} \approx N/p$ is the averaged number of locally owned degrees of freedom of process i, and the upper bound is defined by the case that all locally owned cells are unconnected. The total amount of data to be transferred,

$$2 \cdot N^{\frac{d-1}{d}} \cdot p^{\frac{1}{d}} \leq \sum_i (\texttt{data amount})_i, \tag{11}$$

can be minimized by well-shaped—hypercube-like—partitions and by subdomains containing a large amount of cells. The latter approach competes with the domain-decomposition parallelization approach of a given mesh size. However, the amount of data exchange becomes smaller when using shared memory (see also Section 4).

The partitioning does not only influence the data volume to be transferred but also the memory overhead resulting from the need to allocate memory for at least two buffers of a size bounded

by Equation (11) for sending and receiving the data. Generally, one buffer is part of the actual distributed solution vector, which makes the data access to the degrees of freedom of neighboring cells simpler during operator evaluations. At least on one side of the transfer, data has to be packed and unpacked before or after each communication step, leading to the requirement of the second buffer [25].

2.5 Quadrature

Similar to the shape functions, arbitrary dimensional quadrature rules are expressed as a tensor product of 1D quadrature rules (with n_q points). For the cell integral, we get

$$\mathcal{Q}_{n_q}^d = \underbrace{\mathcal{Q}_{n_q}^1 \otimes ... \otimes \mathcal{Q}_{n_q}^1}_{\times d} \tag{12}$$

with the evaluation points given as $\vec{x}_{n_q}^d = x_{n_q}^1 \otimes ... \otimes x_{n_q}^1$ and the quadrature weights as $w_{n_q}^d = w_{n_q}^1 \cdot ... \cdot w_{n_q}^1$. For the face integrals of the faces $2f$ and $2f+1$, we have

$$\mathcal{Q}_{n_q}^{d-1} = \underbrace{\mathcal{Q}_{n_q}^1 \otimes \cdots \otimes \mathcal{Q}_{n_q}^1}_{\times(f-1)} \otimes \mathcal{Q}_f^1 \otimes \underbrace{\mathcal{Q}_{n_q}^1 \otimes \cdots \otimes \mathcal{Q}_{n_q}^1}_{\times(d-f)} \quad \texttt{with} \quad \mathcal{Q}_f^1 \in \{0,1\}. \tag{13}$$

A widespread approach is the Gauss–Legendre family of quadrature rules (see also Figure 2), which are exact for polynomials of degree $2n_q - 1$. Most efficient implementations of these rules perform a non-trivial interpolation operation from the Gauss–Lobatto to the Gauss–Legendre points. Subsection 2.6 discusses an efficient approach to perform this basis change and to compute gradients at the quadrature points.

An alternative is the Gauss–Lobatto family of quadrature rules, which do not require a basis change, however, are only exact for polynomials of degree $2n_q - 3$. We will consider the Gauss–Legendre family of quadrature rules to quantify the overhead resulting from the basis change.

To be able to evaluate Equation (5), the Jacobian matrix $\mathcal{J}_q \in \mathbb{R}^{d\times d}$, which can be derived from the element geometry, and its determinant $|\mathcal{J}_q| \in \mathbb{R}$ are needed at each cell quadrature point. At face quadrature points, the Jacobian determinant and the face normal $\vec{n}_q \in \mathbb{R}^d$ are needed. These quantities are generally precomputed once during initialization. This leads to an additional memory consumption per final unknown:

$$\frac{n^d \cdot (d^2+1) + 2 \cdot d \cdot n^{d-1}(d+1)}{(k+1)^d} = \mathcal{O}(d^2). \tag{14}$$

For affine meshes, only one set of mapping quantities has to be precomputed and cached, since they are the same for all quadrature points. As we are considering complex non–affine meshes in this paper, we will use simulations with these optimization techniques specific for affine or Cartesian grid only to quantify the quality of our implementation.

2.6 Matrix-free operator evaluation

Let us now turn to the cell integral of the advection operator $\mathcal{A}$ (see Equation 5) as an example:

$$-\left(\mathcal{J}^{-T}\nabla_{\vec{\xi}} v,\ |\mathcal{J}|\vec{a}f\right)_{\Omega_0^{(e)}} \approx -\sum_q \left(\nabla_{\vec{\xi}} v_q,\ \mathcal{J}_q^{-1} w_q\ |\mathcal{J}_q|\vec{a}_q f_q\right)_{\Omega_0^{(e)}}. \tag{15}$$

As visualized in Figure 2, the steps of the evaluation of cell integrals are as follows: 1) gather $(k+1)^d$ cell-local values f_i, 2) interpolate values to quadrature points f_q, 3) perform the operations $\mathcal{J}_q^{-1} w_q\ |\mathcal{J}_q|\vec{a}_q\, f_q$ at each quadrature point, 4) test with the gradient of the shape functions, and 5) write back the local contributions into the global vector.

The most efficient implementations of the basis change (from Gauss–Lobatto to Gauss–Legendre points) perform a sequence of d 1D interpolation sweeps, utilizing the tensor-product form of the shape functions in an even-odd decomposition fashion with $(3 + 2 \cdot \lfloor((k-1)\cdot(k+1)/2)\rfloor/(k-1))$ FLOPs/DoF (for $k+1 = n_q$) [25]. This operation is known as sum factorization and has its origin

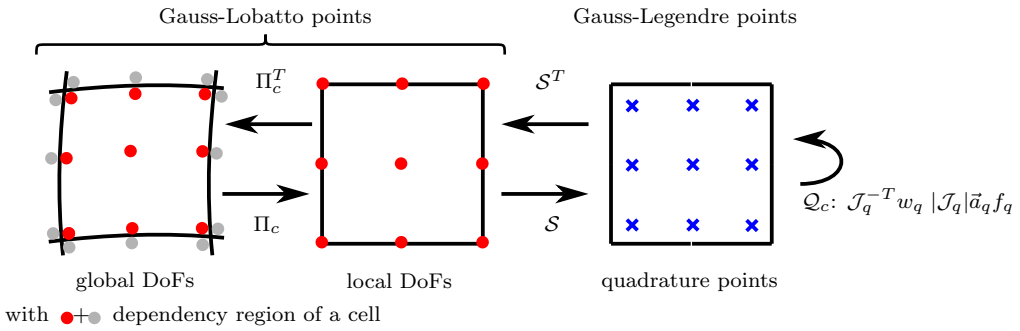


Figure 2: Visualization of the 5 steps of a matrix-free cell-integral evaluation for polynomial degree $k = 2$ and number of quadrature points $n_q = 3$.

in the spectral-element community [13; 29; 30]. Similarly, the testing with the gradient of the shape functions can be performed efficiently with $2d$ sweeps [25].

The same five steps can be performed for all faces $f \in \mathcal{F}$ to compute the flux between two neighboring cells once (with the difference that steps 1–2 and 4–5 are performed for both sides of a face) in a separate loop known as "face-centric loop" (FCL). However, recently the benefits of processing a cell and in direct succession processing all its $2d$ faces (i.e., visiting all faces twice), as in the case of "element-centric loops" (ECL), have become clear for modern CPU processor architecture in the literature [25], although this kind of loop implies that fluxes have to be computed twice (for each side of an interior face). The reasons for the advantage of ECL are twofold: On the one hand, entries in the solution vector are written exactly only once back to main memory in the case of ECL, while in the case of FCL at least once—despite of cache-efficient scheduling of cell and face loops—due to cache capacity misses. On the other hand, since each entry of the solution vector is accessed only exactly once, no synchronization between threads is needed while accessing the solution vector in the case of ECL. This absence of race conditions during writing into the destination vector makes ECL particularly suitable for shared-memory parallelization. Since the exploitation of shared memory is a key ingredient for the reduction of the communication and as a consequence of the memory consumption (as already discussed in Subsection 2.4), we believe that ECL is a suitable approach to enable computations with larger problem sizes despite of the possible increase in computations.

One should also note that although fluxes are computed twice in the case of ECL, this does not automatically translate into doubling of the computation, since values already interpolated to the cell quadrature points can be interpolated to a face with a single 1D sweep.

It is common practice to process a batch of v_{len} cells by one instruction, known as vectorization over elements. This kind of processing can be accomplished by not operating directly on the primitive types `double`/`float` but on structs built around intrinsic instructions, with each vector lane dedicated to a separate cell of mesh. Depending on the given hardware, up to v_{len}=2 (SSE2), =4 (AVX), and =8 doubles (AVX-512 instruction-set extension—as most modern Intel-based processors have) can be processed by a single instruction. Although the working-set size increases by a factor of v_{len} [20], this vectorization strategy showed a better performance than alternative vectorization strategies in 2D and 3D (for their description see [25]) and the auto-vectorization by the compiler.

As the conclusion of this section, Table 1 gives a rough estimate of the working sets for the matrix-free evaluation of the advection operator at different stages of the algorithm. Table 2 shows the estimated minimal memory consumption of the complete simulation. The number of ghost degrees of freedom also gives an estimate for the amount of data to be communicated.

2.7 Potential problems in higher dimensions

The descriptions in Sections 1 and 2 regarding finite-element methods have been general for d-dimensional space Ω. As a consequence, all algorithms also work in high dimensions. However, we face two major challenges: The first one is the lack of libraries designed for high dimensions. For example, the libraries `deal.II` and `p4est` are limited to dimensions up to three. If we would

Table 1: Estimated working set of different stages of a matrix-free evaluation of the advection operator for ECL and vectorization over elements: (1) includes both the source and the destination element vector, (2) includes the buffers needed during testing and face evaluation, (3) includes the degrees of freedom of neighboring cells, needed during flux computation.

	stage		**working set**
(1)	sum factorization	$>$	$v_{len} \cdot \max\left((k+1)^d, n_q^d\right)$
(2)	derived quantities	$>$	$(d+1) \cdot v_{len} \cdot n_q^d$
(3)	flux computation	$\gg$	$(2 \cdot d + 1) \cdot v_{len} \cdot n_q^d$

Table 2: Estimated minimal memory consumption for d-dimensional simulations with N degrees of freedom on p processes.

	reason		**times**		**amount**
(1)	ghost DoFs (+buffer)		2	$\cdot$	$2 \cdot d \cdot N^{\frac{d-1}{d}} \cdot p^{\frac{1}{d}}$
(2)	vector	$+$	2	$\cdot$	N
(3)	mapping	$+$			$d^2 \cdot N$

"naïvely" extend these libraries for high dimensions, the second problem would be that some algorithms do not scale to higher dimensions. The following specific difficulties would arise:

(1) **Significant memory overhead due to ghost values and mapping**: In high dimensions, solution vectors (2–3 are needed, depending on the selected time discretization scheme) are huge ($\mathcal{O}(N_{1D}^d)$), with N_{1D} the number of degrees of freedom in each direction, which are needed to achieve the required resolution. Also the ghost values and the mapping have significant memory requirements in high dimensions: The evaluation of the advection cell integral on complex geometries needs among other things the Jacobian matrix of size $\mathcal{O}(d^2)$ at each quadrature point. If precomputed, this implies an at least 36-fold memory consumption as the actual vector in 6D. For high- dimensional problems, this is not feasible, as only little memory would remain for the actual solution vectors and only problems with significantly smaller resolutions could be solved.

(2) **Increased ghost-value exchange due to increased surface-to-volume ratio**: The communication amount scales—according to Equation (11)—with $2 \cdot d \cdot N^{(d-1)/d} \cdot p^{1/d}$. According to [25], the MPI ghost-value exchange already leads to a noticeable share of time in purely MPI-parallelized applications (30% for Laplacian) in comparison to the highly efficient matrix-free operator evaluations if computations are performed on a single compute node. For high dimensions, the situation is even worse: an estimation with $d = 6$, $N = 10^{12}$, $p = 1024 \cdot 48$ (1024 compute nodes with 48 processes each) gives that the size of the ghost values is at least 72% of the size of the actual solution vector. The usage of shared memory is inevitable to decrease the memory consumption and the time spent in communication: For the given example, the size of ghost values could be halved to 37% if all 48 processes on a compute node shared their locally owned values.

(3) **Decreased efficiency of the operator evaluation due to exponential increasing size of working sets**: The working set of a cell is at least $\mathcal{O}(\max((k+1)^d, n_q^d))$ (cf. Table 1) so that for high order and/or dimension the data eventually drops out of the cache during each sum-factorization sweep of one cell.

While this work cannot solve all these difficulties, we will show how it is possible to mitigate them: We address problem ((1)) by restricting ourselves to the tensor product of two grids in 1-3D. This reduces the size of the mapping data and makes it, moreover, possible to reuse much of the infrastructure available in the baseline library `deal.II`. We will describe in the next section how such a tensor product can be formed. Problem ((2)) demonstrates that it is essential to exploit shared-memory parallelism in particular in high dimensions. In Section 4, we propose a novel shared-memory implementation that is based on MPI-3.0. To mitigate problem ((3)), we try to minimize the number of cache misses due to increased working set sizes by reorganizing the loops. In this respect, we will demonstrate the benefit of vectorizing over fewer elements than the given

instruction-set extensions allow. We accomplish this by explicitly using narrower instructions, such as `AVX2` or `SSE2`, instead of `AVX-512` or by working directly with `double`s and relying on auto-optimization of the compiler. We defer the investigation of explicit vectorization within elements to future work.

3 hyper.deal: a tensor product of two meshes

In this section, we explain how a tensor product of two meshes can be used to solve problems in up to six dimensions. In our sample application of an advection equation in phase space, separating the meshes in configuration space as well as in velocity space is natural, which is why we use the indices $\vec{x}$ and $\vec{v}$ for the two parts of the dimensions.

In the following, we assume that our computational domain is given as a tensor product of two domains $\Omega := \Omega_{\vec{x}} \otimes \Omega_{\vec{v}}$. The boundary of the high-dimensional domain is then described by $\Gamma := (\Gamma_{\vec{x}} \otimes \Omega_{\vec{v}}) \cup (\Omega_{\vec{x}} \otimes \Gamma_{\vec{v}})$. Let us reformulate and simplify the discretized advection equation (see Equation. (5)) for the phase space, exploiting the fact that the Jacobian matrix $\mathcal{J}$ is a block-diagonal matrix in phase space,

$$\mathcal{J} = \begin{pmatrix} \mathcal{J}_{\vec{x}} & 0 \\ 0 & \mathcal{J}_{\vec{v}} \end{pmatrix}, \tag{16}$$

with the blocks being the respective matrices of the $\vec{x}$-space and the $\vec{v}$-space. As a consequence, the inverse $\mathcal{J}^{-1}$ is the inverse of each of its blocks and its determinant $|\mathcal{J}|$ is the product of the determinants of each of its blocks. Exploiting the block-diagonal structure of $\mathcal{J}$, some operations in Equation (5) can be simplified:

$$\mathcal{J}^{-1}(\vec{a}f) = \begin{pmatrix} \mathcal{J}_{\vec{x}}^{-1} & 0 \\ 0 & \mathcal{J}_{\vec{v}}^{-1} \end{pmatrix} \begin{pmatrix} \vec{a}_{\vec{x}} \\ \vec{a}_{\vec{v}} \end{pmatrix} f = \begin{pmatrix} \mathcal{J}_{\vec{x}}^{-1}\vec{a}_{\vec{x}} \\ \mathcal{J}_{\vec{v}}^{-1}\vec{a}_{\vec{v}} \end{pmatrix} f. \tag{17}$$

Furthermore, face integrals over the faces in phase space $\Gamma^{(e)} = (\Gamma_{\vec{x}}^{(e)} \otimes \Omega_{\vec{v}}^{(e)}) \cup (\Omega_{\vec{x}}^{(e)} \otimes \Gamma_{\vec{v}}^{(e)})$ can be split into integration over $\vec{x}$-space faces $\Gamma_{\vec{x}}^{(e)} \otimes \Omega_{\vec{v}}^{(e)}$ and integration over $\vec{v}$-space faces $\Omega_{\vec{x}}^{(e)} \otimes \Gamma_{\vec{v}}^{(e)}$. The following relation is true for $\vec{x}$-space faces: Due to $\vec{n}^\top = (\vec{n}^\top, 0)$ for $\vec{x}$-space faces, $\vec{n}\cdot\vec{a} = \vec{n}_{\vec{x}}\cdot\vec{a}_{\vec{x}}$; and the same is true for $\vec{v}$-space faces $\vec{n}\cdot\vec{a} = \vec{n}_{\vec{v}}\cdot\vec{a}_{\vec{v}}$. Finally, Equation (5), which can be solved efficiently with `hyper.deal`, is:

$$\begin{aligned} &\left(g, \frac{\partial f}{\partial t}\right)_{\Omega^{(e)}} - \left(\nabla_\xi g, \begin{pmatrix} \mathcal{J}_{\vec{x}}^{-1}\vec{a}_{\vec{x}} \\ \mathcal{J}_{\vec{v}}^{-1}\vec{a}_{\vec{v}} \end{pmatrix} f\right)_{\Omega^{(e)}} \\ &\qquad = \left(g, \vec{n}_{\vec{x}} \cdot (\vec{a}_{\vec{x}} f)^*\right)_{\Gamma_{\vec{x}}^{(e)} \otimes \Omega_{\vec{v}}^{(e)}} + \left(g, \vec{n}_{\vec{v}} \cdot (\vec{a}_{\vec{v}} f)^*\right)_{\Omega_{\vec{x}}^{(e)} \otimes \Gamma_{\vec{v}}^{(e)}}. \end{aligned} \tag{18}$$

3.1 Triangulation

Naturally, both domains $\Omega_{\vec{x}}$ and $\Omega_{\vec{v}}$ can be meshed separately. Note, however, that this restricts the possibilities of mesh refinement to the two spaces separately from each other. As a consequence, the final triangulation results from the tensor product of the two triangulations $\mathcal{T}_{\vec{x}}$ and $\mathcal{T}_{\vec{v}}$ (visualized in Figure 3),

$$\mathcal{T} := \mathcal{T}_{\vec{x}} \otimes \mathcal{T}_{\vec{v}}. \tag{19}$$

In this context, cells $\mathcal{C}$, inner faces $\mathcal{I}$, and boundary faces $\mathcal{B}$ are defined as

$$\mathcal{C} := \mathcal{C}_{\vec{x}} \otimes \mathcal{C}_{\vec{v}}, \qquad \mathcal{I} := (\mathcal{I}_{\vec{x}} \otimes \mathcal{C}_{\vec{v}}) \cup (\mathcal{C}_{\vec{x}} \otimes \mathcal{I}_{\vec{v}}), \qquad \mathcal{B} := (\mathcal{B}_{\vec{x}} \otimes \mathcal{C}_{\vec{v}}) \cup (\mathcal{C}_{\vec{x}} \otimes \mathcal{B}_{\vec{v}}). \tag{20}$$

The cells in both low-dimensional triangulations are enumerated independently along space-filling curves. The enumeration of the cells of $\mathcal{T}$ can be chosen arbitrarily. However, in Subsection 3.2 we propose an order that simplifies the initial setup.

Note: By simplifying the $d_{\vec{x}}$-dimensional triangulation $\mathcal{T}_{\vec{x}}$ and the $d_{\vec{v}}$-dimensional triangulation $\mathcal{T}_{\vec{v}}$ to a 1D space-filling curve in each case and by taking the tensor product of these curves, we get—depending on the point of view—a 2D Cartesian grid or a matrix. As a result, we use well-known

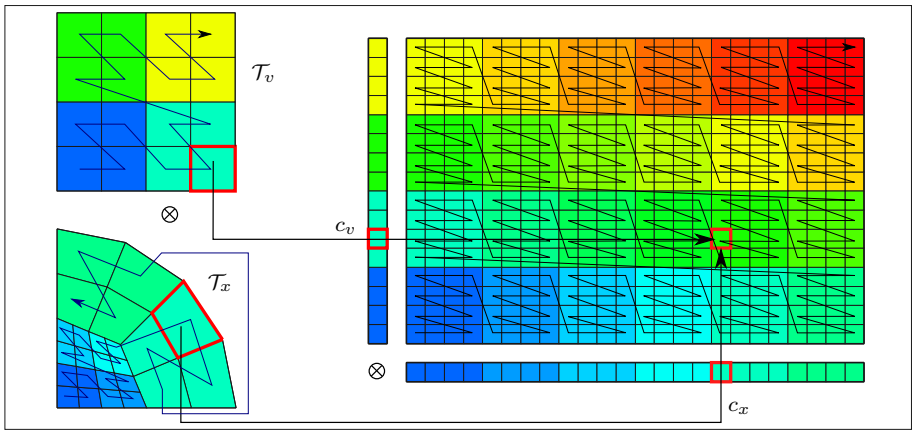


Figure 3: On-the-fly mesh generation of a distributed triangulation in `hyper.deal` by taking the tensor product of two low-dimensional triangulations from the base library `deal.II`. Cells are ordered lexicographically within a process (as are the processes themselves), leading to the depicted space-filling curve.

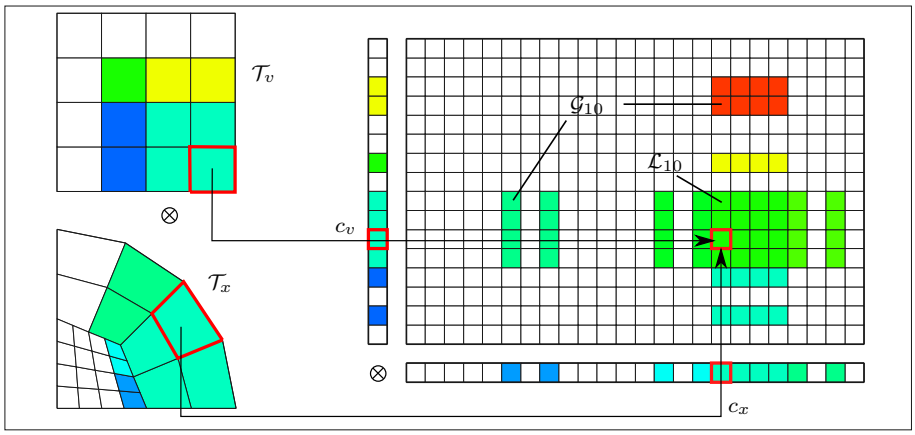


Figure 4: Local view of an arbitrary process: local cells and ghost cells

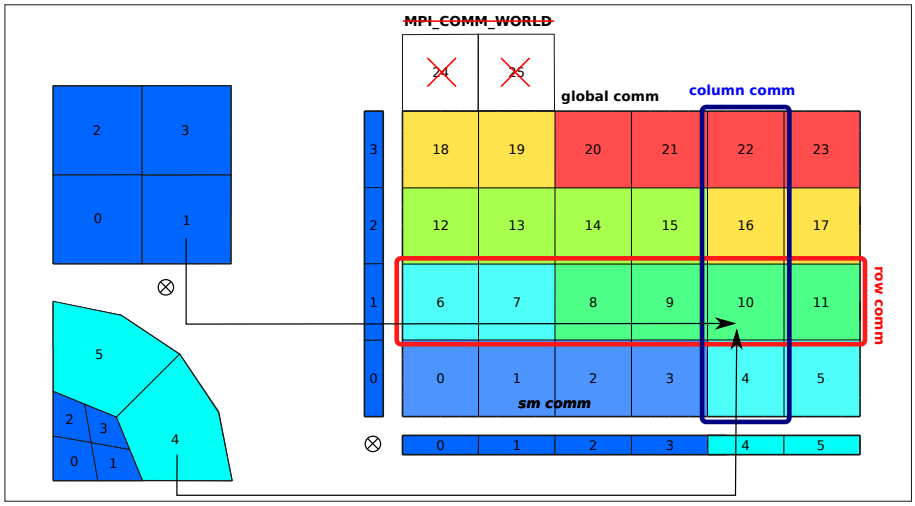


Figure 5: Four MPI communicators used in `hyper.deal` (`global`, `row`, `column`, `sm` comm) for a hypothetical setup of 26 ranks in `MPI_COMM_WORLD` and of a 6×4 partition of the 4D space.

concepts from these fields of research in the library `hyper.deal`*. However, it should be emphasized that, the analogy is not immediate. While the neighborship of a cell on a 2D grid is clear, this is not the case for the high-dimensional triangulation that is depicted as a 2D grid: The number of neighbors is significantly larger, and neighboring cells might be disjunct in this case (see Figure 4). What is true for individual cells is also true for partitions so that the communication patterns are (structured but) significantly more complicated and communication over the boundaries of nodes is unavoidable.*

3.2 Parallelization: partitioning

Since we are using a domain decomposition from the base library `deal.II` for $\mathcal{T}_{\vec{x}}$ and $\mathcal{T}_{\vec{v}}$, we assume these meshes are already split up independently into $p_{\vec{x}}$ and $p_{\vec{v}}$ subpartitions with

$$\mathcal{T}_{\vec{x}} = \biguplus_{0 \leq i < p_{\vec{x}}} \mathcal{T}_{\vec{x}}^i \qquad \text{and} \qquad \mathcal{T}_{\vec{v}} = \biguplus_{0 \leq j < p_{\vec{v}}} \mathcal{T}_{\vec{v}}^j. \tag{21}$$

The subpartition of the phase-space domain belonging to rank $f(i,j)$ is constructed by $\mathcal{T}^{f(i,j)} := \mathcal{T}_{\vec{x}}^i \otimes \mathcal{T}_{\vec{v}}^j$, where ranks are enumerated lexicographically according to $f(i,j) := j \cdot p_{\vec{x}} + i$ with the $\vec{x}$-rank i being the fastest running index. As a consequence, a quasi-checkerboard partitioning is obtained (see Figure 3). Note that it might be advantageous in some cases not to use the full number of processes in order to get a more symmetric decomposition (cf. Figure 5). To take this into account, we are not working on `MPI_COMM_WORLD` but on a subcommunicator, which we will call *global comm*(unicator) in the following.

Please note the following relationship:

$$\mathcal{T}_{\vec{x}} \otimes \mathcal{T}_{\vec{v}}^j = \biguplus_{f(i,j)/n_{\vec{x}} = j} \mathcal{T}^{f(i,j)} \qquad \text{and} \qquad \mathcal{T}_{\vec{x}}^i \otimes \mathcal{T}_{\vec{v}} = \biguplus_{f(i,j)\%n_{\vec{x}} = i} \mathcal{T}^{f(i,j)} \tag{22}$$

so that parallel reduction to distributed $\Omega_{\vec{x}}$-space and to distributed $\Omega_{\vec{v}}$-space becomes a collective communication of subsets of processes. Due to the importance of parallel reductions in mathematical operations like $\int \mathrm{d}\Omega_{\vec{x}}$ and $\int \mathrm{d}\Omega_{\vec{v}}$, we make the subsets of processes available via the MPI communicators *column comm* and *row comm*.

We enumerate cells within a subdomain lexicographically so that we get a space-filling curve as depicted in Figure 3. This enables us to determine a globally unique cell ID of locally owned cells and of ghost cells by querying the low-dimensional triangulation cells for their IDs and ranks without the need for communication.

As a final remark, it should be emphasized that the presented partitioning approach delivers good results if the low-dimensional triangulations $\mathcal{T}_{\vec{x}}$ and $\mathcal{T}_{\vec{v}}$ have already been partitioned well. Depending on the given mesh, a space-filling curve approach as supported by `p4est` or a graph-based approach as supported by `METIS` and `Zoltan` might be beneficial.

In Subsection 2.7, we have discussed the importance of the usage of shared memory for solving high-dimensional problems. Placing ranks according to $\lfloor f(i,j)/p_{node} \rfloor$ onto the same compute node (with p_{node} being the number of processes per node) leads to striped partitioning (see the colors of the blocks in Figure 5). This results in a suboptimal shape of the union of subpartitions belonging to the same compute node, leading to decreased benefit of the usage of shared memory. In order to improve the placing of subpartitions onto the compute nodes without having to change the function f, we are operating on a virtual topology, as will be presented in Section 4.

3.3 Elements, degrees of freedom, and quadrature

Since we are considering shape functions that are derived by the tensor product of 1D-shape functions, i.e., $\mathcal{P}_k^{d_{\vec{x}}} = \underbrace{\mathcal{P}_k^1 \otimes \cdots \otimes \mathcal{P}_k^1}_{\times d_{\vec{x}}}$ and $\mathcal{P}_k^{d_{\vec{v}}} = \underbrace{\mathcal{P}_k^1 \otimes \cdots \otimes \mathcal{P}_k^1}_{\times d_{\vec{v}}}$, the extension to higher dimensions is trivial

$$\mathcal{P}_k^{d_{\vec{x}}+d_{\vec{v}}} = \underbrace{\mathcal{P}_k^1 \otimes \cdots \otimes \mathcal{P}_k^1}_{\times d_{\vec{x}}} \otimes \underbrace{\mathcal{P}_k^1 \otimes \cdots \otimes \mathcal{P}_k^1}_{\times d_{\vec{v}}} = \mathcal{P}_k^{d_{\vec{x}}} \otimes \mathcal{P}_k^{d_{\vec{v}}}. \tag{23}$$

Table 3: Comparison of the memory consumption (in doubles) of the mapping data (per quadrature point) if the phase-space structure is exploited ($J_{\vec{x}}$ and $J_{\vec{v}}$) and if the phase-space structure is not exploited (J).

	$J_{\vec{x}}$ and $J_{\vec{v}}$	J	**example:** $k=3$, $d=6$
Jacobian:	$\frac{(k+1)^{d_{\vec{x}}} \cdot d_{\vec{x}}^2 + (k+1)^{d_{\vec{v}}} \cdot d_{\vec{v}}^2}{(k+1)^{d_{\vec{x}}+d_{\vec{v}}}}$	$(d_{\vec{x}} + d_{\vec{v}})^2$	$0.28 \ll 36$

This relationship is also true for the quadrature formulas so that a basis change can be performed—as usual—with $d_{\vec{x}} + d_{\vec{v}}$ sum-factorization sweeps.

As a consequence, only the mapping data from lower-dimensional spaces (e.g., Jacobian matrix and its determinant) have to be precomputed and can be reused, which leads to a significantly reduced memory consumption even for complex geometries as shown in Table 3.

3.4 Matrix-free operator evaluation

The extension of operator evaluations with sum factorization to higher dimensions is also straightforward. Instead of looping over all cells of a triangulation, it requires an iteration over all possible pairs of cells from both low-dimensional triangulations $(c_{\vec{x}}, c_{\vec{v}}) \in \mathcal{C}_{\vec{x}} \times \mathcal{C}_{\vec{v}}$ in the case of ECL. In addition to cell pairs, FCL iterates also over cell-face and cell-boundary-face pairs as expressed by Equation (20). This comes in handy, since the mapping information of the cells $c_{\vec{x}}$ and $c_{\vec{v}}$ as well as of the faces $f_{\vec{x}}$ and $f_{\vec{v}}$ can be queried from the low-dimensional library independently and is only combined on the fly. The separate cell IDs $c_{\vec{x}}$ and $c_{\vec{v}}$ only have to be combined when accessing the solution vector, which is the only data structure set up for the whole high-dimensional space.

Algorithm 1–2 show the pseudocode of a possible matrix-free advection operator evaluation. As an example, Algorithm 1 contains an element-centric loop (ECL) iterating over all cell pairs and calling the function that should be evaluated on the cell pairs. Up to here, the algorithm is independent of the equation to be solved; the equation comes into play (apart from the specific ghost-value update) by the called function, which might be the advection operator in Algorithm 2.

Furthermore, we note that lines 2–4, 8, 10, 12 in Algorithm 2 are evaluated with sum factorization. To reduce the working set, we do not compute all $(d_{\vec{x}} + d_{\vec{v}})$-derivatives at once, but first we compute the contributions from $\vec{x}$-space and then the contributions from $\vec{v}$-space. One could reduce the working set even more by loop blocking [25], but we deal with a generic variant here.

3.5 Implementation of operator evaluations with hyper.deal

The library `hyper.deal` provides classes that contain inter alia utility functions needed in Algorithm 2 and are built around `deal.II` classes. To enable a smooth start for users already familiar with `deal.II`, we have chosen the same class and function names living in the namespace `hyperdeal`. The relationship between some `hyper.deal` classes and some `deal.II` classes is visualized in the UML diagram in Figure 6. The class `hyperdeal::MatrixFree` is responsible for looping over cells (and faces) as well as for storing precomputed information related to shape functions and precomputed quantities at the quadrature points. The classes `hyperdeal::FEEvaluation` and `hyperdeal::FEFaceEvaluation` (not shown) contain utility functions for the current cells and faces: These utility functions include functions to read and write cell-/face-local values from a global vector as well as also operations at the quadrature points. As an example, Figure 6 shows the implementation of the `hyperdeal::FEEvaluation::submit_gradient()` method, which uses for the evaluation of $f(\vec{u}) = \mathcal{J}_{c,q}^{-1} |\mathcal{J}_q| w_q \, \vec{u}, \ \vec{u} \in \mathbb{R}^d$, two instances of the `deal.II` class with the same name—one for $\vec{x}$- and one for $\vec{v}$-space.

We use "vectorization over elements" from `deal.II` as vectorization strategy. To be precise, we vectorize only over elements in $\vec{x}$-space, whereas $\vec{v}$-space is not vectorized. Note that in the Vlasov–Maxwell or Vlasov–Poisson model, where parts of the code operate on the full phase space and other parts on the $\vec{x}$-space only, it is important to vectorize over $\vec{x}$. Then, the data structures are already laid out correctly for an efficient matrix-free solution of the lower-dimensional problem.

Algorithm 1: Element-centric loop for arbitrary operators

`update_ghost_values`: Import vector values of $\vec{u}$ from MPI processes that are adjacent to locally owned cells

`/* loop over all cell pairs */`

foreach $(e_{\vec{x}}, e_{\vec{v}}) \in \mathcal{C}_{\vec{x}} \times \mathcal{C}_{\vec{v}}$ **do**

  process_cell($e_{\vec{x}}$, $e_{\vec{v}}$) ; `/* e.g., advection cell operator in Algorithm 2 */`

Algorithm 2: DG integration of a cell batch for advection operator evaluation for ECL and for vectorization over elements

`/* step 1: gather values (AoS → SoA) */`

gather local vector values $u_i^{(e)}$ on the cell from global input vector $\vec{u}$

`/* step 2: apply advection cell contributions */`

interpolate local vector values $\vec{u}^{(e)}$ onto quadrature points, $u_h^e(\vec{\xi}_q) = \sum_i \phi_i u_i^{(e)}$

for each quadrature index $q = (q_{\vec{x}}, q_{\vec{v}})$, prepare integrand on each quadrature point by computing $\vec{t}_q = \mathcal{J}_{(e_{\vec{x}})}^{-1} \vec{c}_{\vec{x}} \left(\hat{x}^{(e_{\vec{x}})}(\vec{\xi}_{q_{\vec{x}}}),\ \hat{x}^{(e_{\vec{v}})}(\vec{\xi}_{q_{\vec{v}}}) \right) u_h^{(e)}(\vec{\xi}_q) \underbrace{|\mathcal{J}_{q_{\vec{x}}}||\mathcal{J}_{q_{\vec{v}}}| w_{q_{\vec{x}}} w_{q_{\vec{v}}}}_{\text{"}|\mathcal{J}_{(q)}| w_q\text{"}}$ and evaluate local integrals by quadrature $b_i = \left(\nabla_{\vec{x}} \phi_i^{co}, \vec{c}_{\vec{x}} u_h^{(e)} \right)_{\Omega_{(e)}} \approx \sum_q \nabla_{\vec{x}} \phi_i^{co}(\vec{\xi}_q) \cdot \vec{t}_q$; `/* buffer` $\vec{b}$ `*/`

for each quadrature index $q = (q_{\vec{x}}, q_{\vec{v}})$, prepare integrand on each quadrature point by computing $\vec{t}_q = \mathcal{J}_{(e_{\vec{v}})}^{-1} \vec{c}_{\vec{v}} \left(\hat{x}^{(e_{\vec{x}})}(\vec{\xi}_{q_{\vec{x}}}),\ \hat{x}^{(e_{\vec{v}})}(\vec{\xi}_{q_{\vec{v}}}) \right) u_h^{(e)}(\vec{\xi}_q) \underbrace{|\mathcal{J}_{q_{\vec{x}}}||\mathcal{J}_{q_{\vec{v}}}| w_{q_{\vec{x}}} w_{q_{\vec{v}}}}_{\text{"}|\mathcal{J}_{(q)}| w_q\text{"}}$ and evaluate local integrals by quadrature $b_i = b_i + \left(\nabla_{\vec{v}} \phi_i^{co}, \vec{c}_{\vec{x}} u_h^{(e)} \right)_{\Omega_{(e)}} \approx b_i + \sum_q \nabla_{\vec{v}} \phi_i^{co}(\vec{\xi}_q) \cdot \vec{t}_q$

`/* step 3: apply advection face contributions (loop over all` $2d$ `faces of` Ω_e`) */`

foreach $f \in \mathcal{F}_{(e)}$ **do**

  interpolate values from cell array $\vec{u}^{(e)}$ to quadrature points of face

  if not on boundary, gather values from neighbor Ω_{e^+} of current face

  interpolate u^+ onto face quadrature points

  compute numerical flux and multiply by quadrature weights ; `/* not shown here */`

  evaluate local integrals related to cell e by quadrature and add into cell contribution b_i

end

`/* step 4: apply inverse mass matrix */`

for each quadrature index $q = (q_{\vec{x}}, q_{\vec{v}})$, prepare integrand on each quadrature point by computing $t_q = b_i w_{q_{\vec{x}}}^{-1} w_{q_{\vec{v}}}^{-1}$ and evaluate local integrals by quadrature $y_i^{(e)} = \sum_q \tilde{\phi}_{iq} \cdot \vec{t}_q$ with $\tilde{\phi}_{iq} = \mathcal{V}_{iq}^{-1}$ with $\mathcal{V}_{iq} = \phi_i(\vec{\xi}_q)$

`/* step 5: scatter values (SoA → AoS) */`

set all contributions of cell, $\vec{y}^{(e)}$, into global result vector $\vec{y}$

4 Parallelization by shared-memory MPI

The parallelization of the library `hyper.deal` is purely MPI-based with one MPI process per physical core. Each MPI process possesses its own subpartition and has a halo of ghost faces (see Figure 1 and 4). MPI allows to program for distributed systems, which is crucial for solving high-dimensional problems due to their immense memory requirements. A downside of a purely MPI-based code is that many data structures (incl. ghost values) are created and updated multiple times on the same compute node, although they could be shared. In FEM codes, it is widespread that ghost values filled by standard `MPI_(I)Send`/`MPI_(I)Recv` reside in an additional section of the solution vector. Depending on the MPI implementation, these operations will be replaced by efficient alternatives (e.g., based on `memcpy`—if the calling peers are on the same compute node. Nevertheless, the allocation of additional memory—if main memory is scarce—might be unacceptable.

Adding shared-memory libraries like `TBB` and `OpenMP` to an existing `MPI` program would allow to use shared memory, however, this comes with an overhead for the programmer, since all parallelizable code sections have to be found and transformed according to the library used, including the difficulty when some third-party numerical library like an iterative solver package only relies

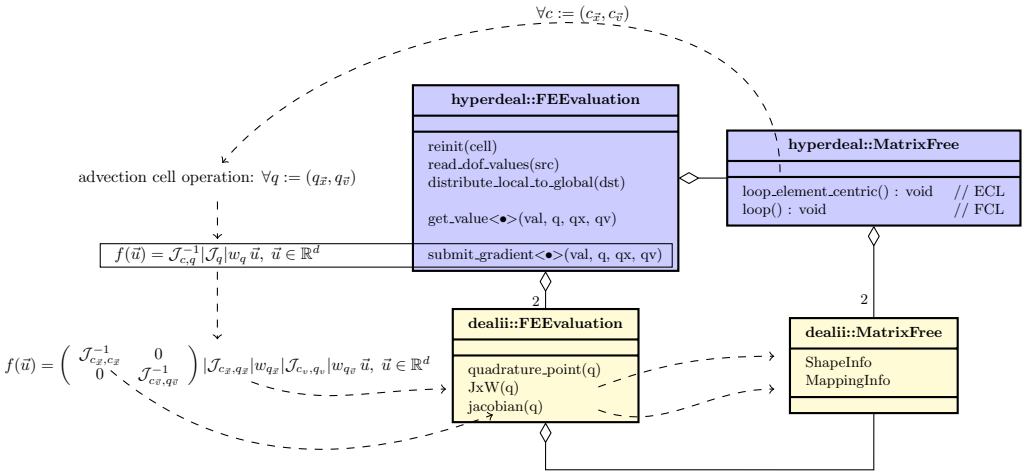


Figure 6: Class diagram of part of the matrix-free infrastructure of `hyper.deal`. It presents how classes from `hyper.deal` (namespace `hyperdeal`— highlighted blue) and from `deal.II` (namespace `dealii`— highlighted yellow) relate to each other. Only the `hyper.deal` methods are shown that are relevant for the evaluation of the advection operator and the `deal.II` methods that are used in those.

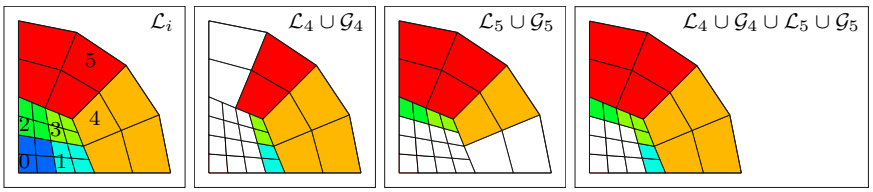
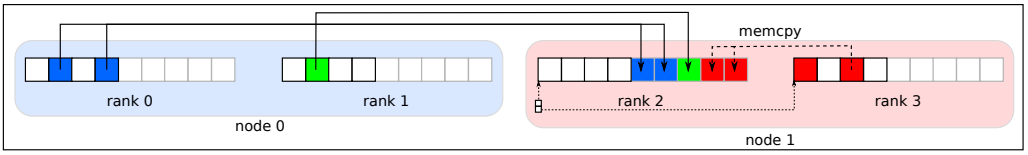
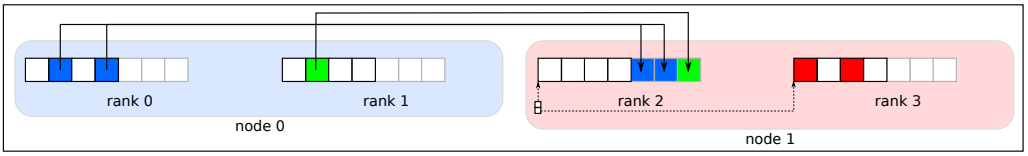


Figure 7: Shared-memory domain: a) partitioned distributed triangulation; b-c) locally relevant cells of ranks 4 and 5; d) locally relevant cells of shared-memory domain (containing ranks 4 and 5) with a shared ghost layer and without ghost cells between the processes in the same shared-memory domain.



(a) A hybrid approach using MPI-3.0 shared-memory features (`buffered mode`). Ghost values are updated via send/recv between nodes or explicitly via memory copy within the node. The similarity to a standard `MPI` implementation is clear with the difference that memcpy is called directly by the program, making packing/unpacking of data superfluous.



(b) A hybrid approach using MPI-3.0 shared-memory features (`non-buffered mode`) similar to 8a, with the difference that only ghost values that live in different shared-memory domains are updated. Ghost values living in the same shared-memory domain are directly accessed only when needed. The similarity to a thread-based implementation is clear with the differences that vectors are non-contiguous, requiring an indirect access to values owned by other processes, and that each process may manage its own ghost values and send/receive its own medium-sized messages needed to fully utilize the network controller.

Figure 8: Two hybrid ghost-value-update approaches for a hypothetical setup with 2 nodes, each with two cores. Only the communication pattern of rank 2 is considered.

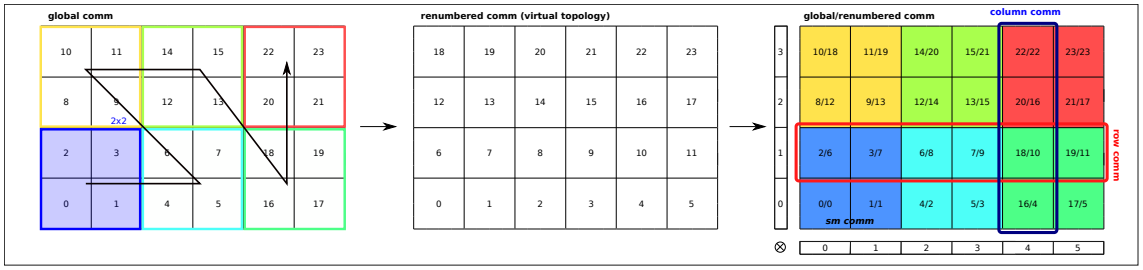


Figure 9: Renumbering of ranks in the global communicator (via `MPI_Comm_split`): For a hypothetical setup of 26 ranks in `MPI_COMM_WORLD` and of a 6×4 partition, processes are grouped in 2×2 blocks and these blocks are ordered along a z-curve. Based on this new global communicator, the partitioning as described in Subsection 3.2 is applied. A checkerboard partitioning with a better data locality for the shared-memory domains is obtained.

on `MPI`.

Considering a purely MPI-parallelized FEM application, one can identify that the major time and memory benefit of using shared memory would come from accessing the part of the solution vector owned by the processes on the same compute node without the need to make explicit copies and buffering them. This is why we propose a new vector class that uses `MPI-3.0` features to allocate shared memory and provides controlled access to it, while retaining the same vector interface and adding only a few new methods.

`MPI` provides the function `MPI_WIN_ALLOCATE_SHARED` to allocate non-contiguous memory that is shared among all processes in the subcommunicator containing all MPI processes on the same compute node. To query the beginning of the local array of each process, it provides the function `MPI_WIN_SHARED_QUERY`.

These two functions enable us to allocate memory needed by each process for its locally owned cells and for the ghost faces that are not owned by any process on the same compute node as well as to get hold of the beginning of the arrays of the other processes. Appendix A provides further implementation details of the allocation/deallocation process of the shared memory in the vector class. With basic preprocessing steps, the address of each cell residing on the same compute node can be determined so that the processes have access to all degrees of freedom owned by that compute node (see Figure 7). A natural way to access the solution vector is by specifying vector entry indices and the cell ID for degrees of freedom owned by a cell or by specifying a pair of a cell ID and a face number ($< 2d$) for degrees of freedom owned by faces.

In `hyper.deal`, the class `dealii::LinearAlgebra::SharedMPI::Vector` manages the shared memory and also provides access to the values of the degrees of freedom of the local and the ghost cells: It returns either pointers to buffers or the shared memory, depending on the cell type (see Figure 8b). In this way, the user of the vector class gets the illusion of a pure MPI program, since the new vector has to be added only at a single place and only a few functions querying values from the vector (e.g., `read_dof_values` and `distribute_local_to_global` in Figure 6)—oblivious to the user— have to be specialized.

We provide two operation modes:

- In the `buffered mode` (see Figure 8a), memory is allocated also for ghost values owned by the same compute node; these ghost values are updated directly via `memcpy`, without an intermediate step via `MPI`. This mode is necessary if ghost values are modified, as it takes place in face-centric loops. It promises some performance benefit, since data packing/unpacking can be skipped.

- The `non-buffered mode` (see Figure 8b) does not allocate any redundant memory for ghost values owned by the same compute node. This mode works perfectly with ECL, since it is by design free of race conditions. Due to the harmony of ECL and the `non-buffered mode` of the shared-memory vector, we rely on this vector mode in the rest of this paper.

In order to make this type of hybrid parallelization approach successful, the union of the subpartitions of all processes on a compute node should build a well-formed subpartition on its

own with an improved surface-to-volume ratio. We achieve this via blocking within a Cartesian virtual topology (see also Figure 9), e.g., by 48 process blocks with 8 processes in $\vec{x}$-space and with 6 processes in $\vec{v}$-space. Compute nodes are ordered along a z-curve.

As a final remark, we emphasize that a vector built around `MPI-3.0` shared-memory features is not limited to high dimensions, but can be used also in lower dimensions; in particular, if not only a single "ghost layer" has to be exchanged (as in the case of an advection operator) but multiple "ghost layers" (as in the case of a Laplace operator discretized with the interior penalty method [26]).

5 Application: high-dimensional scalar transport

In the following section, we show results of the solution of a high-dimensional scalar transport problem. These results confirm the suitability of the underlying concepts and the implementation of the library `hyper.deal` for high orders and high dimensions. Both node-level performance and parallel performance results are shown, including the findings of strong and weak scaling analyses with up to 147,456 processes on 3,072 compute nodes.

5.1 Experimental setup and performance metrics

The setup of the simulations is as follows. We consider the computational domains $\Omega_{\vec{x}}=[0,1]^{d_{\vec{x}}}$ and $\Omega_{\vec{v}}=[0,1]^{d_{\vec{v}}}$ with the following decomposition of the dimensions $d = d_{\vec{x}} + d_{\vec{v}}$: $2 = 1 + 1$, $3 = 2 + 1$, $4 = 2 + 2$, $5 = 3 + 2$, $6 = 3 + 3$. The computational domains are initially meshed separately with subdivided $d_{\vec{x}}/d_{\vec{v}}$-dimensional hyperrectangles– with $(2^{l_1}, \dots, 2^{l_{d_{\vec{x}}}}) \in \mathbb{N}^{d_{\vec{x}}}$ and $(2^{l_{d_{\vec{x}}+1}}, \dots, 2^{l_{d_{\vec{x}}+d_{\vec{v}}}}) \in \mathbb{N}^{d_{\vec{v}}}$ hexahedral elements in each direction and with a difference in the mesh size of at most two, i.e., meshed for 4D from the mesh sequence (l_1, l_2, l_3, l_4): $(1,1,1,1)$, $(2,1,1,1)$, $(2,2,1,1)$, $(2,2,2,1)$, $(2,2,2,2)$, $(3,2,2,2)$. The number of elements is selected for each "dimension d / polynomial degree k" configuration in such a way that the solution vectors do not fit into the cache. To obtain unique Jacobian matrices at each quadrature points ($\mathcal{J} = \mathcal{J}(\vec{x})$) and to prevent the usage of algorithms explicitly designed for affine meshes, we deform the Cartesian mesh slightly. The velocity $\vec{a}$ in Equation (3) is set constant and uniform over the whole domain ($\vec{a} \neq \vec{a}(t, \vec{x})$).

The measurement data have been gathered either with user-defined timers or with the help of the script `likwid-mpirun` from the `LIKWID suite` and with suitable in-code `LIKWID API` annotations [31; 33]. The following metrics are used to quantify the quality of the implementations:

- **throughput**: processed degrees of freedom per time unit

$$\text{throughput} = \frac{\text{processed DoFs}}{\text{time}} \overset{\text{Eqn. (8)}}{=} \frac{|\mathcal{C}| \cdot (k+1)^d}{\text{time}} \tag{24}$$

  (In Subsections 5.2–5.4, we consider the throughput for the application of the advection operator, while a single Runge–Kutta stage, i.e., the evaluation of the advection operator plus vector updates, is considered in Subsection 5.5.);

- **performance**: maximum number of floating-point operations per second;

- **data volume**: the amount of data transferred within the memory hierarchy (Here we consider the transfer between the L1-, L2-, and L3-caches as well as the main memory.);

- **bandwidth**: data volume transferred between the levels in the memory hierarchy per time.

Our main objective is to decrease the time-to-solution and to increase the throughput (at constant error, not considered in this paper). The measured quantities "performance", "data volume", and "bandwidth" are useful, since they show how well the given hardware is utilized and how much additional work or memory transfer is performed compared to the theoretical requirements of the mathematical algorithm.

The focus of this study is on the performance of `hyper.deal` for high-order and high-dimensional problems that have a large working set $(k+1)^d$. The evaluation of this expression for $2 \leq k \leq 5$ and $2 \leq d \leq 6$ is presented in Table 4. The k-d configurations with working-set size of $v_{len} \cdot (k+1)^d < L_1$,

Table 4: Degrees of freedom per cell: $(k+1)^d$. The k-d configurations with working-set size $v_{len} \cdot (k+1)^d < L_1$ are highlighted in italics and configurations with working-set size $L_1 \leq v_{len} \cdot (k+1)^d < L_2$ in bold. Hardware characteristics (v_{len}, L_1, L_2) are taken from Table 5.

k/d	2	3	4	5	6
2	*9*	*27*	*81*	*243*	**729**
3	*16*	*64*	*256*	**1024**	**4096**
4	*25*	*125*	**625**	**3125**	**15625**
5	*36*	*216*	**1296**	**7776**	46656

Table 5: Specification of the hardware system used for evaluation with turbo mode enabled. Memory bandwidth is according to the STREAM triad benchmark (optimized variant without read for ownership transfer involving two reads and one write), and GFLOP/s are based on the theoretical maximum at the AVX-512 frequency. The `dgemm` performance is measured for $m = n = k = 12{,}000$ with Intel MKL 18.0.2. We measured a frequency of 2.3 GHz with AVX-512 dense code for the current experiments. The empirical machine balance is computed as the ratio of measured `dgemm` performance and STREAM bandwidth from RAM memory.

	Intel Skylake Xeon Platinum 8174
cores	2×24
frequency base (max AVX-512 frequency)	2.3 GHz (2.3 GHz)
SIMD width	512 bit
arithmetic peak (`dgemm` performance)	4147 GFLOP/s (2920 GFLOP/s)
memory interface	DDR4-2666, 12 channels
STREAM memory bandwidth	205 GB/s
empirical machine balance	14.3 FLOP/Byte
L1-/L2-/L3-/MEM size	32kB/1MB/66MB (shared)/96GB(shared)
compiler + compiler flags	`g++`, version 9.1.0, `-O3 -funroll-loops -march=skylake-avx512`

fitting into L1 cache (highlighted in italics), are expected to show good performance; the k-d configurations with working-set size of $L_1 \leq v_{len} \cdot (k+1)^d < L_2$ (highlighted in bold) are expected to be performance-critical, since each sum-factorization sweep might drop out of the cache. These latter configurations are, however, the most relevant with regard to high-order and high-dimensional problems.

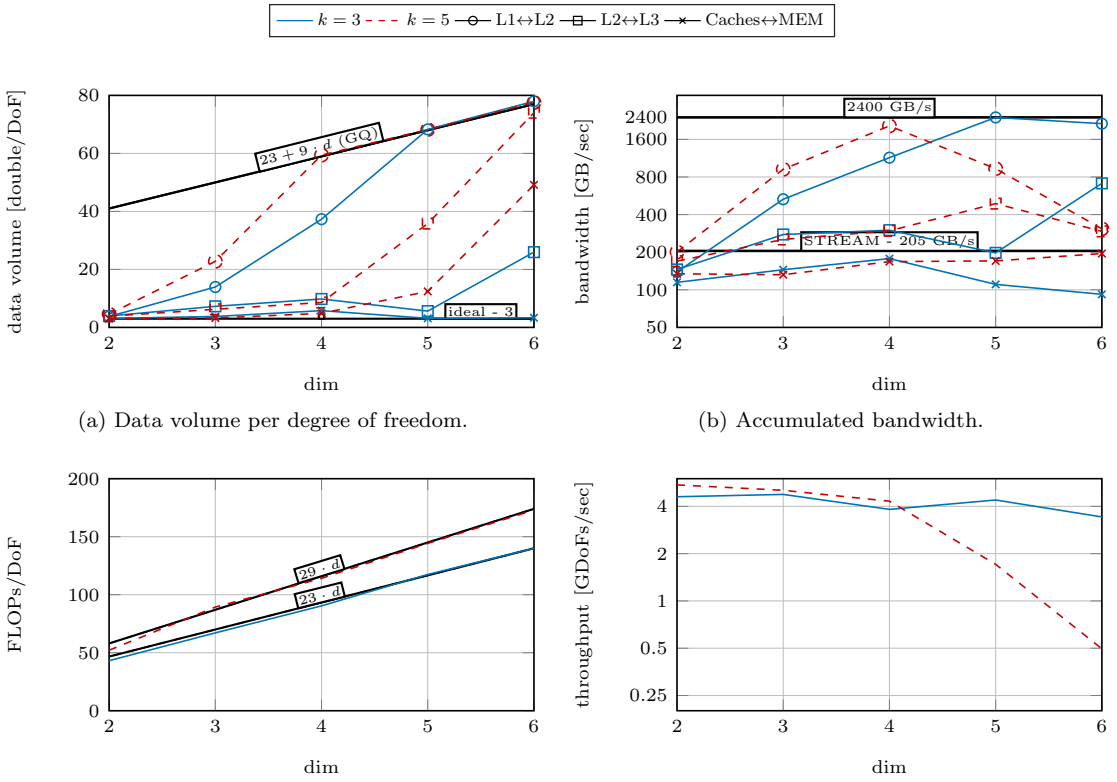
All performance measurements have been conducted on the SuperMUC-NG supercomputer. Its compute nodes have 2 sockets (each with 24 cores of Intel Xeon Skylake), a measured bandwidth of 205GB/s to main memory, and the AVX-512 ISA extension so that 8 doubles can be processed per instruction. A detailed specification of the hardware is given in Table 5. An island consists of 792 compute nodes. The maximum network bandwidth per node within an island is 100GBit/s=12.5GB/s[1] due to the fat-tree network topology. Islands are connected via a pruned tree network architecture (pruning factor 1:4).

The library `hyper.deal` has been configured in the following way: All processes of a node are grouped, and they build blocks of the size of 48=8×6. All processes in these blocks share their values via the shared-memory vector. We use the highest ISA extension AVX-512 so that 8 cells are processed at once. Jacobian matrix and its determinant are precomputed for $\vec{x}$- and $\vec{v}$-space and combined on the fly. The quadrature is based on the Gauss-Lagrange formula with $n_q = k+1$. In the following, we refer to this configuration as "default configuration".

5.2 Cell-local computation

This subsection takes an in-depth look at the cell-local computation in the element-centric evaluation of the advection operator (see Algorithm 2). Cell-local computations do not access non-cached values of neighboring cells, making it easier to reason about the number of sweeps and the working-set size (see the first two working-set sizes in Table 1), which increase with the dimension.

[1] `https://doku.lrz.de/display/PUBLIC/SuperMUC-NG`



(a) Data volume per degree of freedom.

(b) Accumulated bandwidth.

(c) Floating-point operations per degree of freedom.

(d) Throughput.

Figure 10: Node-level analysis of the cell integrals of the advection operator

5.2.1 Cell integrals

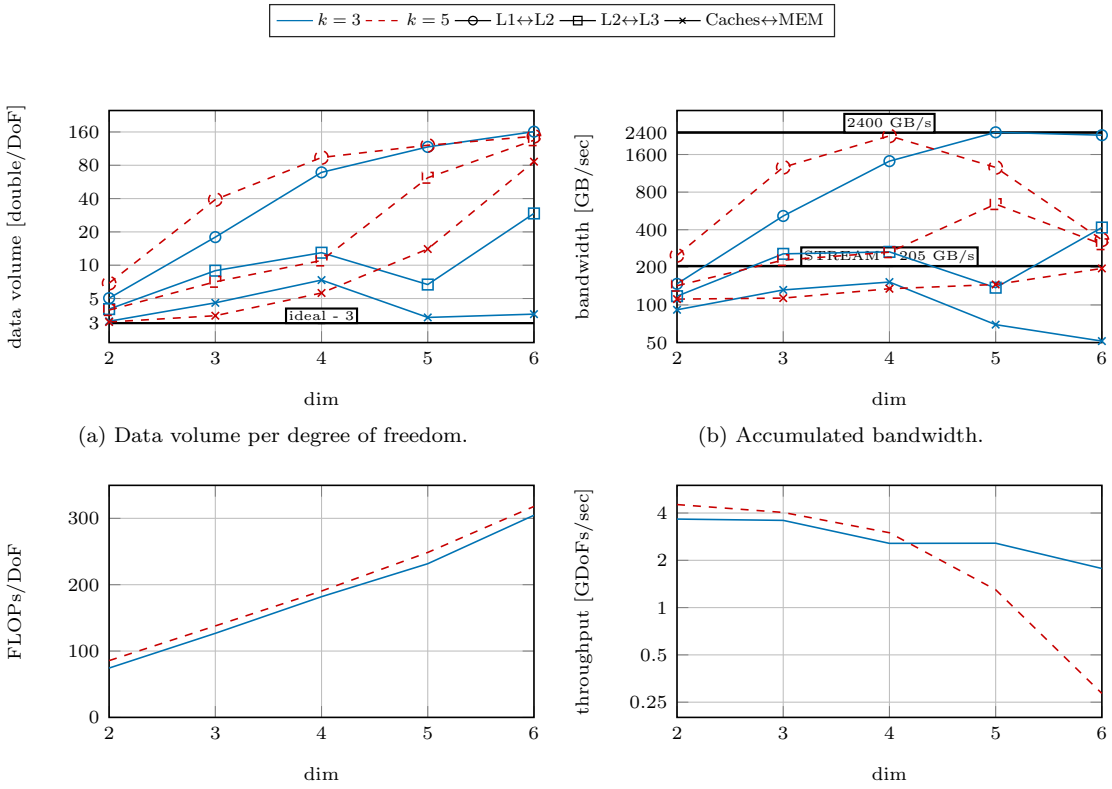
We first consider all steps in Algorithm 2 related to the cell integrals (lines 1-4, 12, 13), skipping the loops over faces and ignoring the flux computation.

During the cell integrals, values are read from the global vector, a basis change to the Gauss–Legendre quadrature points is performed (with $\approx d$ data sweeps for reading and $\approx d$ for writing), the values obtained are multiplied with the velocities at the quadrature points ($\approx 2d$) and tested by the gradient of the collocation functions ($\approx 3d$), the inverse mass matrix is applied ($\approx 2d$), and finally the results are written back to the global vector. A total of $\approx 9d$ data sweeps are necessary if reading and writing are counted separately. The working set of sum factorization is $v_{len} \cdot (k+1)^{d_1+d_2}$, and the working set of the intermediate values is $v_{len} \cdot \max(d_1, d_2) \cdot (k+1)^{d_1+d_2}$. A comparison with hardware statistics shows that the working set of sum factorization exceeds the size of the L1 cache for configurations $k = 3$ / $d = 5$ and $k = 5$ / $d = 4$ so that every data sweep has to fetch the data from the L2 cache.

The theoretical considerations made above are supported by the measurement results in Figure 10, which shows the data traffic between the memory hierarchy levels (data volume per DoF and bandwidth), the floating-point operations per DoF, and the throughput for $k = 3$ and $k = 5$ for $2 \leq d \leq 6$.

In Figure 10a, one can observe that with increasing dimension the data traffic between the memory hierarchy levels increases as the data volume and the corresponding bandwidth increase. Beginning from the configurations mentioned above ($k = 3$ / $d = 5$ and $k = 5$ / $d = 4$), the data has to be fetched from and written back to L2 cache again during every sweep, resulting in a data volume traffic between L1 and L2 caches that linearly increases with the number of sweeps. The constant offset of 23 double/DoF is mainly related to the access to the global solution vector. However, data also has to be loaded from L2 cache for smaller working-set sizes than the ones of the k-d configurations mentioned above; the main reason for this is that the intermediate values do not fit into the cache any more.

For even higher dimensions, the L3 cache and main memory have to be accessed during the



(a) Data volume per degree of freedom.

(b) Accumulated bandwidth.

(c) Floating-point operations per degree of freedom.

(d) Throughput.

Figure 11: Full operator evaluation without loading values from neighboring cells

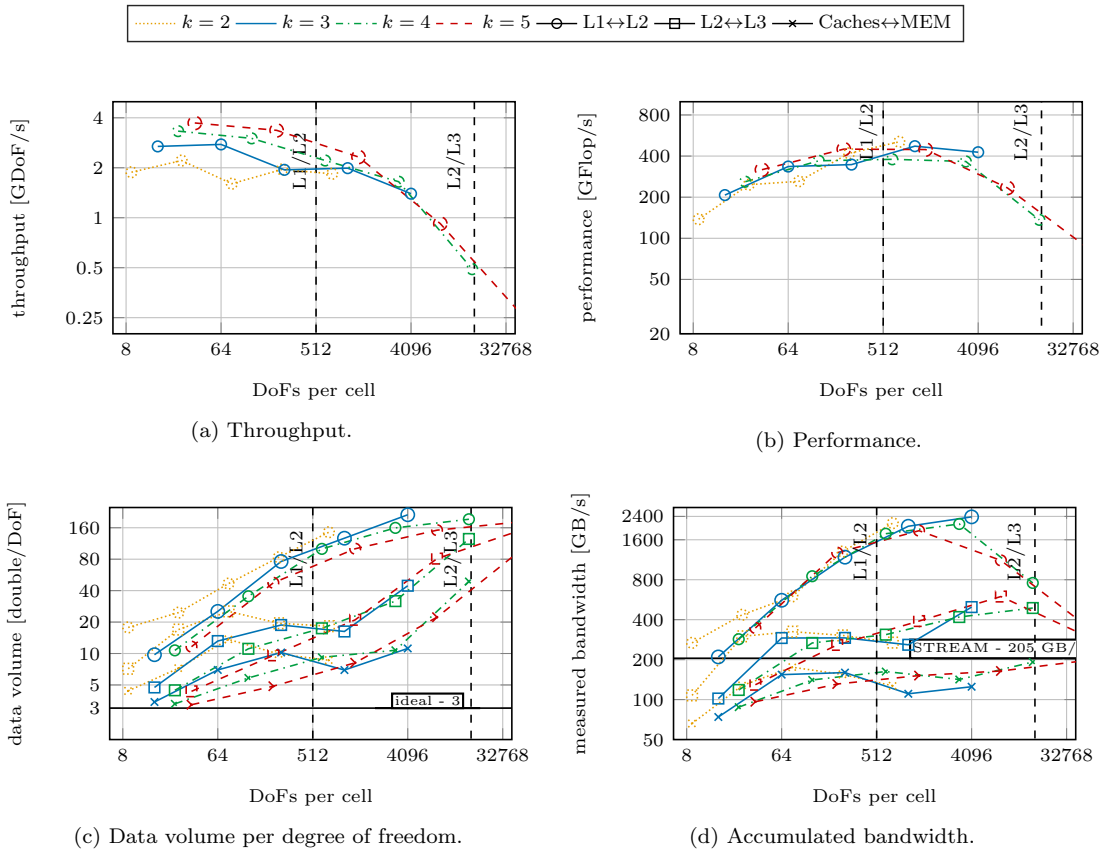
sweeps. While this operation is negligible for $k = 3$ (see Figure 10d), it is performance-limiting in the case of $k = 5$: For $k = 5$ / $d = 5$, the bandwidth to L1 cache is limited by the access to L2 cache (see Figure 10b); for $k = 5$ / $d = 6$, it is even limited by the main memory. In the latter case, the caches are hardly utilized any more and the data has to be fetched from/written back to main memory during every sweep, leading to a bandwidth close to the values measured for the STREAM benchmark, however, also resulting in a significant performance drop.

Figure 10c also shows the number of floating-point operations performed per degree of freedom. It increases linearly with the dimension d—with higher polynomial degrees requiring more work. It is clear that the arithmetic intensity will also increase linearly as long as the data stays in the cache (see also Subsection 5.3).

5.2.2 Local cell and face integrals:

In this subsection, we consider all computation steps in Algorithm 2, but ignore the data access to neighboring cells (line 7). This means that face values from neighboring cells are not gathered and face buffers for exterior values are left unchanged. In this way, we are able to demonstrate the effects of increased working sets (of both face buffers) and of the increased number of sweeps. Additional $\geq 2d$ sweeps have to be performed for interpolating values from the cell quadrature points to the quadrature points of the $2d$ faces (as well as for interpolating the values of the neighboring cells onto the quadrature points and for the flux computation).

Figure 11 shows the data traffic between the memory hierarchy levels (data volume per DoF and bandwidth), the floating-point operations per DoF, and the throughput for $k = 3$ and $k = 5$ for $2 \leq d \leq 6$. In comparison to the results of the experiments that only consider the cell integrals in Figure 10, the following observations can be made: As expected, the data volume transferred between the cache levels (Figure 10a and 11a) and the number of floating-point operations approximately double (Figure 10c and 11c). However, the configurations at which the traffic to the next cache level increases have not changed, indicating that the increase in working set is not limiting



(a) Throughput.

(b) Performance.

(c) Data volume per degree of freedom.

(d) Accumulated bandwidth.

Figure 12: Node-level analysis of the application of the full advection operator.

the performance here.

The doubling of the data volume to be transferred for $k = 5$ and high dimensions naturally leads to half the throughput (see Figure 11d). In the case of $k = 3$, we can also observe a drop of performance in high dimensions. This has another cause: The memory transfer between L1 and L2 caches is up to 2,400GB/s and between L2 and L3 caches about 400GB/s. The latter value is the half of that observed for the cell-integral-only run, indicating that the data for the computation is mainly delivered form the L2 cache. This fact leads to an under-utilization of the L3 cache and of the main-memory bandwidth, resulting in the drop of the overall performance by 50% for high dimensions.

5.3 Full advection operator

This subsection considers the application of the full advection operator as shown in Algorithm 2, including the access to neighboring cells during the computation of the numerical flux. Figure 12 presents the results of parameter studies of the dimension $2 \leq d \leq 6$ for different polynomial degrees $2 \leq k \leq 5$.

Please note that the number of degrees of freedom per cell, $(k+1)^d$, is utilized as x-axis. This is done because the working set of this size is a suitable indicator of the overall performance of the operator evaluation (see Subsection 5.2). Also results taken from parameter studies of the polynomial degree k are comparable to results obtained from parameter studies of the dimension d.

The following observations can be made in Figure 12: For working sets that fit into the L1 cache, a higher polynomial degree leads to a higher throughput. For working sets exceeding the size of the L1 cache and only fitting into the L2 cache, the throughput drops. In this region, the curves overlap and we conclude that the throughput is indeed a function of the working-set size $\approx (k+1)^d$ and independent of the polynomial degree k and the dimension d individually.

Table 6: Relative performance due to flux computation (ratio Fig 12-10).

k/d	2	3	4	5	6
3	58%	58%	51%	45%	41%
4	66%	60%	53%	41%	54%
5	68%	66%	54%	54%	52%

Table 7: Relative performance due to access to the values of neighboring cells (ratio Fig 12-11).

k/d	2	3	4	5	6
3	73%	77%	76%	77%	79%
4	79%	75%	77%	67%	80%
5	82%	83%	78%	71%	91%

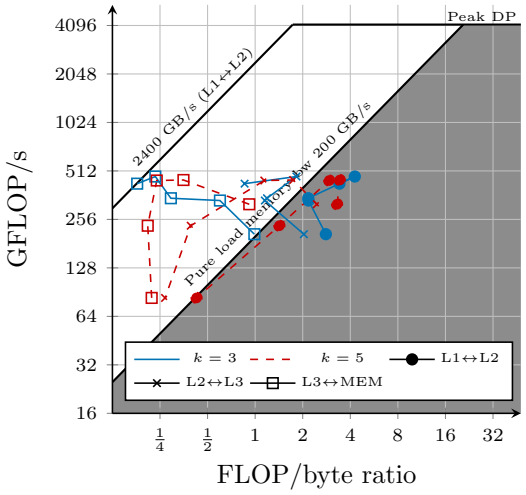


Figure 13: Roofline model for $k = 3/5$.

Comparing these findings with the results presented in Subsection 5.2.1 and 5.2.2, an averaged performance drop of 54% and 23%, respectively, can be observed (see Table 6 and 7). Looking at the results for high dimensions, it becomes clear that processing the faces is more expensive than loading the actual values from the neighbors.

Figure 13 shows a Roofline model [36] for $k = 3$ and $k = 5$. In this model, the measured performance is plotted over the measured arithmetic intensity

$$(\text{measured arithmetic intensity})_i = \frac{\text{measured performance}}{(\text{measured bandwidth})_i}, \tag{25}$$

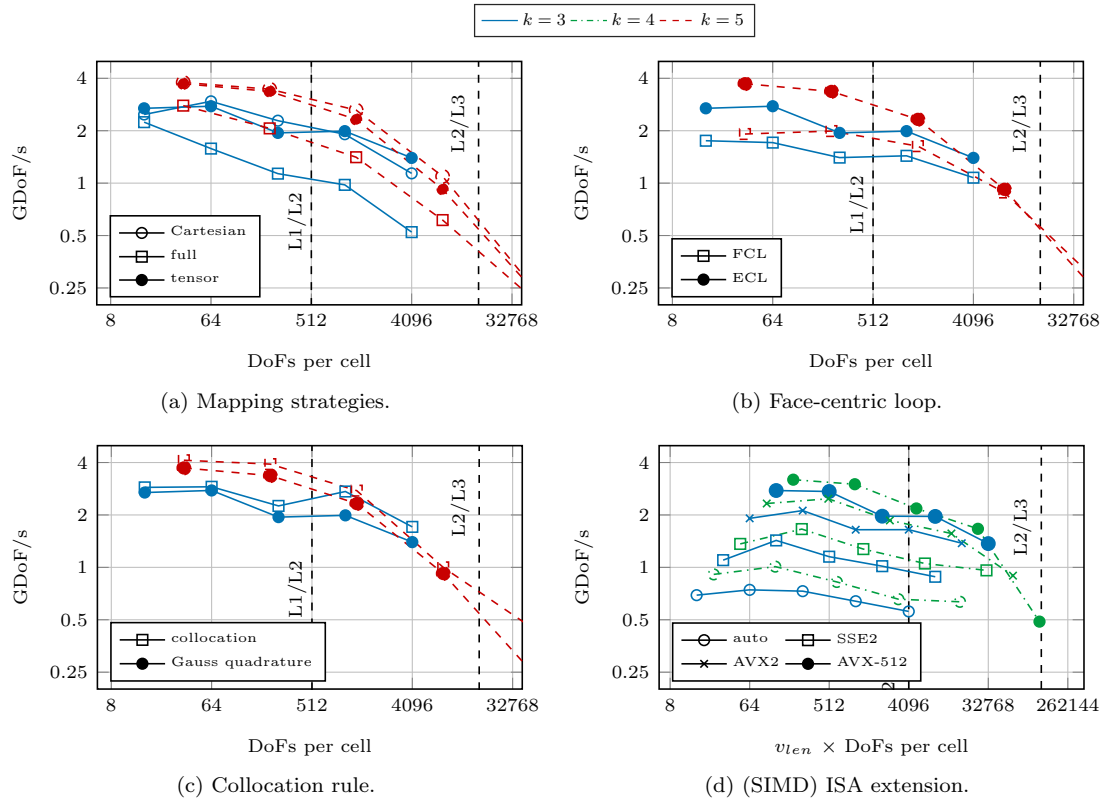
with $i \in \{\text{L1} \leftrightarrow \text{L2}, \text{L2} \leftrightarrow \text{L3}, \text{Caches} \leftrightarrow \text{MEM}\}$. We can compute the arithmetic intensity of each level of memory hierarchy as we measure the necessary bandwidth with `LIKWID`. The diagram confirms the observation made above: A high arithmetic intensity and consequently high performance can only be reached if the caches (L1 and L2) are utilized well. As the working sets get too large, L1 and L2 caches are under-utilized, the arithmetic intensity on the other levels drops and new hard (bandwidth) ceilings limit the maximal possible performance.

5.4 Alternative implementations

In the following subsection, we compare the performance of the default setup of the library `hyper.deal` (tensor product of mappings, ECL, Gauss quadrature, vectorization over elements with highest ISA extension for vectorization–see also Subsection 5.1) with the performance of alternative algorithms and/or configurations.

The library `hyper.deal` has been developed to be able to compute efficiently on complex geometries both in geometric and velocity space. This is achieved by combining the Jacobians of the mapping of the individual spaces on the fly. The upper limit of the performance of this approach is given by the consideration of the tensor product of two Cartesian grids, which leads to the same constant and diagonal Jacobian matrix at all quadrature points. As a lower limit, one can consider the case that each quadrature point has a unique Jacobian of size $d \times d$. Figure 14a shows that the behavior of the default tensor-product setup is similar to that of a pure Cartesian grid simulation with only a small averaged performance penalty of approx. 4%. This observation matches our expectations expressed in Subsection 3.3 and means that in high dimensions the evaluation of curved meshes in the tensor-product factors is essentially for free, compared to storing the full Jacobian matrices.

In the default configuration of `hyper.deal`, we have been using element-centric loops (ECL). They show significantly better performance than face-centric loops (FCL) as shown in Figure 14b. We have not implemented any advanced blocking schemes—as it is available in `deal.II`—for neither ECL nor FCL. The fact that ECL nevertheless shows a better performance demonstrates the natural cache-friendly property of ECL. Furthermore, ECL is well-suited for shared-memory



(a) Mapping strategies.

(b) Face-centric loop.

(c) Collocation rule.

(d) (SIMD) ISA extension.

Figure 14: a-c)Comparison of the performance of different algorithms and the default configuration for $k = 3$ and $k = 5$. d) Comparison of the performance of different (SIMD) ISA extensions and the auto-vectorization for $k = 3$ and $k = 4$.

Table 8: Partitioning the $\vec{x}$- and $\vec{v}$-triangulations on up to 3,072 nodes with 48 cores

nodes	1	2	4	8	16	32	64	128	256	512	1,024	2,048	3,072
$p_{\vec{x}}$	8	12	16	24	32	48	64	96	128	192	256	384	384
$p_{\vec{v}}$	6	8	12	16	24	32	48	64	96	128	192	256	384

Table 9: Strong and weak scaling configurations: left) $d = 4$ / $k = 5$; right) $d = 6$ / $k = 3$

DoFs	configuration	DoFs	configuration
5.4GDoFs	$384^2 \cdot 192^2$	2.1GDoFs	$32^5 \cdot 64^1$
21.7GDoFs	384^4	17.2GDoFs	$32^2 \cdot 64^4$
268MDoFs/node	$192^2 \cdot 96^2$	268MDoFs/node	$16^2 \cdot 32^4$
1.1GDoFs/node	192^4	1.1GDoFs/node	32^6

computation and a single communication step is sufficient. The benefit of ECL decreases for high dimensions due to the increased number of sweeps, related to the repeated evaluation of the flux terms. Nevertheless, we propose to use ECL for high-dimension problems because of its suitability for shared-memory computations that reduce the allocated memory.

We favor the Gauss–Legendre quadrature method over the collocation methods due to its higher numerical accuracy. This benefit comes at the price of a basis change from the Gauss–Lobatto points to the Gauss quadrature points and vice versa. Figure 14c shows a performance drop of on average 15% due to these basis changes as long as the data that should be interpolated remains in cache.

In the library `hyper.deal`, we currently only support "vectorization over elements". As a default, the highest instruction-set extension is selected, i.e., the maximum number of cells is processed at once by a core. Since the number of lanes to be used is templated, the user can reduce the number of elements that are processed at once, as it is demonstrated in Figure 14d. It can be observed that, in general, the usage of higher instruction-set extensions leads to a better throughput. However, once the working set of a cell batch exceeds the size of the cache, a performance drop can be observed. The performance drop leads to the fact that in 6D with cubic elements the throughputs of AVX-512 and of AVX2 are comparable and in 6D with quartic elements SSE2 shows the best performance.

In this subsection, we have demonstrated that the chosen default configuration of the library `hyper.deal` has a competitive throughput compared to less memory-expensive and computationally demanding algorithms, which are numerically inferior.

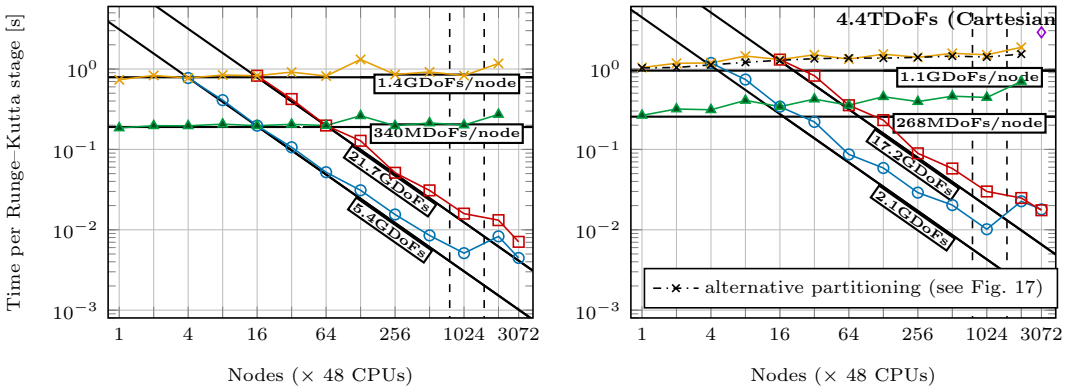
5.5 Strong and weak scaling

In this subsection, we examine the parallel efficiency of the library `hyper.deal`. For this study, we consider the advection operator embedded into a low-storage Runge–Kutta scheme of order 4 with 5 stages, which uses two auxiliary vectors besides the solution vector [21]. From these three vectors, only one (auxiliary) vector is ghosted.

Figure 15 shows strong and weak scaling results of runs on SuperMUC-NG with up to 3072 nodes with a total of 147,456 cores. We consider two configurations: "$d = 4$ / $k = 5$", an easy configuration, and "$d = 6$ / $k = 3$", a demanding configuration. As examples, we present for each configuration two strong and two weak scaling curves (see Table 8). Table 9 shows the considered process decomposition $p = p_{\vec{x}} \cdot p_{\vec{v}}$, which has not been optimized for the given mesh configurations.

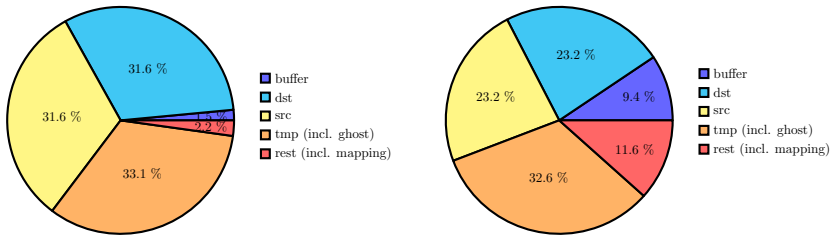
For the "$d = 4$ / $k = 5$" configuration, we observe excellent weak-scaling behavior with parallel efficiencies of 92% and 89% for up to 2,048 nodes on the large and small setup, respectively. We get more than 70/80% efficiency for strong scaling up to the increase in the number of nodes by a factor of 128/256. For the "$d = 6$ / $k = 3$" configuration, we see parallel efficiencies of 38/56% for weak scaling. These values are lower than the ones in the 4D case, however, they are still very good in the light of the immense communication amount in the 6D case: As shown in Fig 16, the ghost data amount to 40% of the solution vector in 6D and only 5% in 4D.

The largest simulation with curved mesh, which contains $128^5 \cdot 64^1 = 2.2 \cdot 10^{12}$ degrees of freedom, reaches a total throughput of 1.2PDoFs/s(=$1.2 \cdot 10^{12}$DoFs/s) on 2048 compute nodes and 1.7PDoFs/s on 3072 compute nodes. This means that each Runge-Kutta stage is processed



(a) Configuration "$d = 4$ / $k = 5$". (b) Configuration "$d = 6$ / $k = 3$".

Figure 15: Strong and weak scaling of one Runge–Kutta step with the advection operator as right-hand side with up to 147,456 processes on 3,072 compute nodes.



(a) $d = 4$ / $k = 5$ with $1.4 \cdot 10^{12}$ DoFs. (b) $d = 6$ / $k = 3$ with $1.1 \cdot 10^{12}$ DoFs.

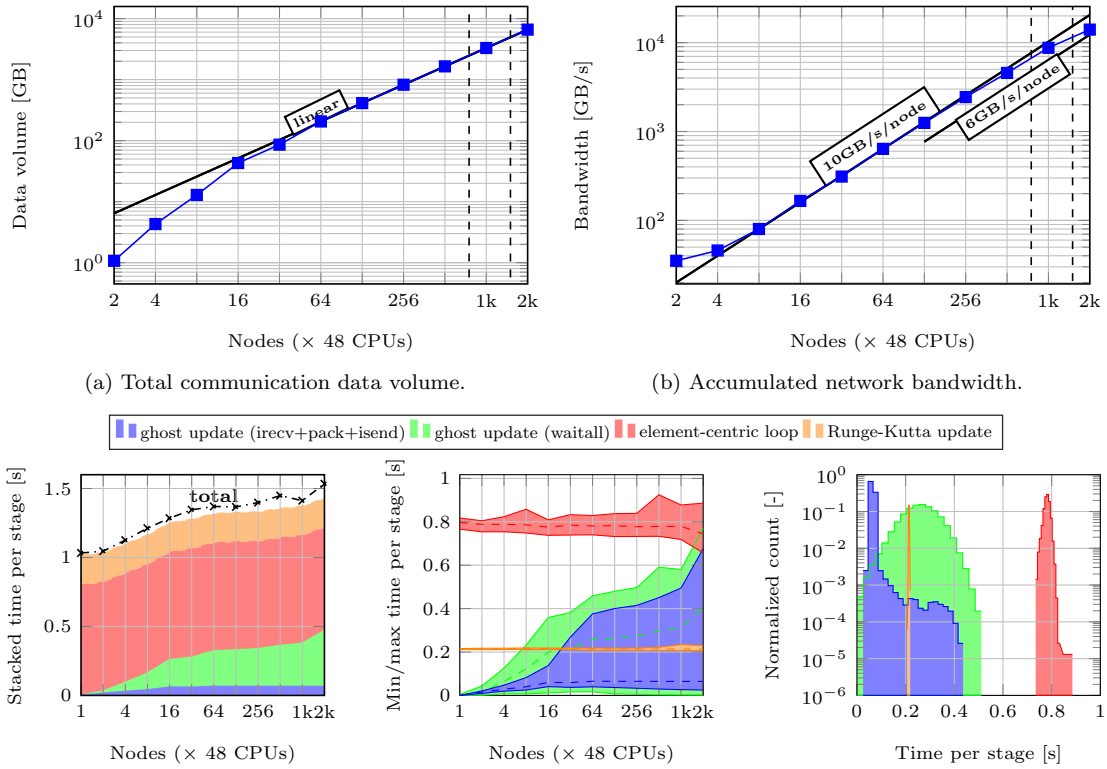
Figure 16: Memory consumption for 48×1024 cores.

in 1.8s/1.3s, and a complete time step consisting of 5 Runge-Kutta stages takes 9.2s/6.4s.

Figure 16b shows the approximated memory consumption for a large-scale simulation (1024 nodes, $d = 6$ / $k = 3$, $1.1 \cdot 10^{12}$ DoFs). A total of 34.4PB main memory from available 98PB is used. The largest amount of memory is occupied by the three solution vectors (each. 23.2%). The two buffers for MPI communication occupy each 9.4%. One of the buffers is attributed to the ghost-value section of the vector called *tmp*. The remaining data structures, which include inter alia the mapping data, occupy only a small share (11.6%) of the main memory, illustrating the benefit of the tensor-product approach employed by the library `hyper.deal` in constructing a memory-efficient algorithm for arbitrary complex geometries for high dimensions. As reference, the memory consumption for a 4D simulation is shown in Figure 16a. Obviously, the additional memory consumption due to the precomputed mapping slightly reduces the largest possible problem size fitting to a certain number of nodes. For example, the largest 4.4 trillion DoFs simulation of Figure 15 only fits into memory on 3072 compute nodes for a Cartesian mesh.

Finally, we discuss the drop in parallel efficiency of the weak-scaling runs of the large-scale simulations. For this, we have slightly modified the setup: We start from a configuration of 8^6 cells with $k = 3$ on one node (32 degrees of freedom in each direction). When doubling the number of processes, we double the number of cells in one direction, starting from direction 1 to direction 6 (and starting over at direction 1). Each time, we double the number of cells along a $\vec{x}$-direction, we also double the number of processes $p_{\vec{x}}$, keeping $p_{\vec{v}}$ constant and vice versa. In this way, the number of DoFs per process along $\vec{x}$ and $\vec{v}$ as well as the number of ghost degrees of freedom per process remain constant once all cells at the boundary have periodic neighbors residing on other nodes (number of nodes $\geq 2^{d_{\vec{x}}+d_{\vec{v}}}$). As a consequence, the computational work load and the communication amount of each node are constant. The total time per Runge–Kutta stage is shown in Figure 15 with dashed lines.

The total communication amount of the considered setup increases linearly with the number of processes as shown in Figure 17a. Measurements in Figure 17b show that the network can handle this increase: The data can be sent with a constant network bandwidth of 10GB/s per node, which



(a) Total communication data volume.

(b) Accumulated network bandwidth.

(c) Averaged runtime distribution.

(d) Averaged minimum and maximum runtime.

(e) Histograms of step runtime (64 nodes).

Figure 17: Weak scaling of a single Runge–Kutta stage of the solution of the advection equation.

is close to the theoretical 100Gbit/s, as long as the job stays on an island due to the fat-tree network topology and with a smaller bandwidth once the job stretches over multiple islands due to the pruned tree network architecture. In the latter case, we observed a bandwidth of 6GB/s, which is related to the fact that only a small ratio of the messages crosses island boundaries.

Figure 17c and 17d show the time spent in different sections (in the following referred to as steps) of the advection operator. We consider the following four steps:

(1) Start a ghost-value update by calling `MPI_Irecv` as well as pack and send (vie `MPI_Isend`) messages to each neighboring process residing on remote compute nodes.

(2) Finish the ghost-value update by waiting (with `MPI_Waitall`) until all messages have been sent and received.

(3) Evaluate the right-hand-side term of the advection equation by performing an element-centric loop on locally owned cells.

(4) Perform the remaining Runge–Kutta update steps.

Besides averaged times, the minimum and maximum times encountered on any process are shown for each step. The times have been averaged over all Runge–Kutta stages.

Not surprisingly, the imbalance in the element-centric loop and in the Runge–Kutta update is small, which can be explained with the access to the shared recourses (mainly RAM). Furthermore, the ratio of communication in the overall run time increases with increasing number of nodes—in particular, the time increases for low numbers of nodes, while new periodic neighbors are still added and for high node numbers when communication to other islands is required.

What is surprising is that the minimum and the maximum time spent in updating the ghost values differ significantly: The difference increases even with increasing number of nodes. The detailed analysis of the histograms of the times on different processors presented in Figure 17e shows that communications are processed according to a Gaussian distribution, resulting in a few processes finishing much earlier and in a few ones much later. Overall, however, this imbalance caused by the MPI communication is not performance-hindering as was demonstrated by the fact that a significant portion of the network bandwidth is used. However, one should keep this imbalance in mind and not attribute it accidentally to other sections of the code.

6 Application: Vlasov–Poisson

We now study the Vlasov–Poisson equations as an example of an application scenario for our library. We consider the Vlasov equation for electrons in a neutralizing background in the absence of magnetic fields,

$$\frac{\partial f}{\partial t} + \begin{pmatrix} \vec{v} \\ -\vec{E}(t,\vec{x}) \end{pmatrix} \cdot \nabla f = 0, \tag{26a}$$

where the electric field can be obtained from the Poisson problem

$$\rho(t,\vec{x}) = 1 - \int f(t,\vec{x},\vec{v})\,\mathrm{d}v, \qquad -\nabla_{\vec{x}}^2 \phi(t,x) = \rho(t,\vec{x}), \qquad \vec{E}(t,\vec{x}) = -\nabla_{\vec{x}}\phi(t,\vec{x}). \tag{26b}$$

For the solution of this problem, we use the low-storage Runge–Kutta method of order 4 with 5 stages. Each stage contains the following five steps for evaluating the right-hand side:

(1) Compute the degrees of freedom of the charge density via integration over the velocity space.

(2) Compute the right-hand side for the Poisson equation.

(3) Solve the Poisson equation for ϕ.

(4) Compute $\vec{E}$ from ϕ.

(5) Apply the advection operator.

Step ((5)) is a $d_{\vec{x}} + d_{\vec{v}}$-dimensional problem, and Steps (2)–(4) are $d_{\vec{x}}$-dimensional problems. Step (1) reduces information from the phase space to the configuration space.

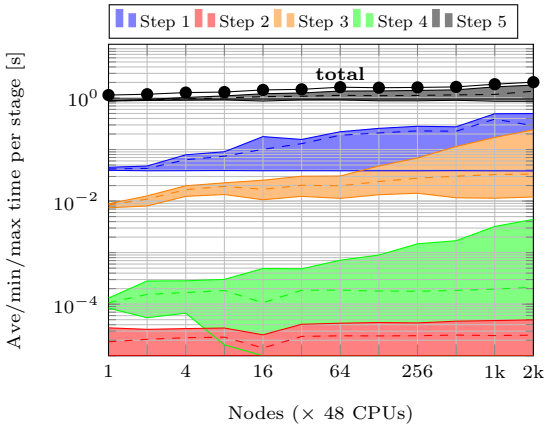


Figure 18: Weak scaling of a single Runge–Kutta stage of the solution of the Vlasov–Poisson equations. The average (dashed), averaged minimum and averaged maximum runtime of each step are indicated.

6.1 Implementation details

The advection step (Step ((5))) relies on the operator analyzed in Section 5. The constant velocity field function $\vec{a}$ is replaced by the function $\vec{a}(t,\vec{x},\vec{v})^\top = \left(\vec{v}^\top,\ -\vec{E}(t,\vec{x})^\top\right)$. The following two points should be noted: Firstly, $\vec{v}$ evaluated at a quadrature point corresponds exactly to a quadrature point in the $\vec{v}$- domain and can hence be queried from a low-dimensional FEM library. Similarly, $\vec{E}$ is independent of the velocity $\vec{v}$ and can be precomputed once at each Runge–Kutta stage for all quadrature points in the $\vec{x}$-space. Exploiting these relations, we never compute the $d_{\vec{x}} + d_{\vec{v}}$-dimensional velocity field, but compose the $d_{\vec{x}}$- and $d_{\vec{v}}$-information on the fly, just as we did in the case of the mapping. Since the data to be loaded per quadrature point is negligible (see also the reasoning regarding the Jacobian matrices in Subsection 3.3), the throughput of the advection operator is weakly effected by the need to evaluate the velocity field at every quadrature point.

For the solution of the Poisson problem

$$(\nabla_{\vec{x}}\psi, \nabla_{\vec{x}}\phi)_{\vec{x}} = (\psi,\ \rho)_{\vec{x}}, \tag{27}$$

we utilize a matrix-free geometric multigrid solver [14; 28] from `deal.II`, which uses a Chebyshev smoother [1]. The Poisson problem is solved by each process group with a constant velocity grid (see `row_comm` in Subsection 3.2). The result of this is that the solution ϕ is available on each process without the need of an additional broadcast step.

The integration of f over the velocity space is implemented via an `MPI_Allreduce` operation over all processes with the same $\vec{x}$ grid partition (i.e., `colomn_comm`) so that the resulting ρ is available on all processes.

We have verified our implementation with a simulation of the Landau damping problem as in [22].

6.2 Weak scaling

We perform a weak-scaling experiment for the 6D Vlasov equation, starting from a configuration of 8^6 cells with $k = 3$ on one node. When doubling the number of processes, we double the number of cells in one direction, starting from direction 1 to direction 6 (and starting over at direction 1). Each time, we double the number of cells along a $\vec{x}$-direction, we also double the number of processes $p_{\vec{x}}$ keeping $p_{\vec{v}}$ constant and vice versa. In this way, the number of DoFs per process along $\vec{x}$ and $\vec{v}$ remains constant.

Figure 18 shows the scaling of Steps 1-5 of one Runge–Kutta stage. We can see that the total computing time is dominated by the 6D-advection step, which we have analyzed earlier.

Step 1, which reduces f to ρ, becomes increasingly important as the problem size and parallelism increase. In this step, an all-reduction is performed over process groups with constant $p_{\vec{x}}$ coordinate

in the process grid (called `comm_column` in Subsection 3.2). The amount of data sent by each process corresponds to the number of DoFs in $\vec{x}$-direction of this process and is thus the same in every experiment. The total amount of data sent/received is thus proportional to the total number of processes, while the size of the subcommunicators—and hence the number of reduction steps—only depends on $p_{\vec{v}}$ ($\log(p_{\vec{v}})$). The scaling experiment shows that the time needed by step 1 generally increases with the number of nodes and that this step has the worst scaling behavior. This can be explained by the fact that this step contains a global communication within the subcommunicators, while the other steps only contain nearest-neighbor communication. We also note that the process grid is designed to optimize the communication scheme of the advection step. In step 1, this results in the fact that each subcommunicator might involve all the nodes.

The steps 2 to 4 are three-dimensional problems that are mostly negligible, only the Poisson solver (step 3) has some impact on the total computing time. Let us note that the 3D parts are solved $p_{\vec{v}}$ times on all subcommunicators (called `comm_row` in Subsection 3.2) with constant $p_{\vec{v}}$ coordinate. Between the first and the forth data point as well as between the seventh and the tenth data point, the size of the Poisson problem increases, which results in a slight increase of the computing time. Between data point 4 and 7 as well as 10 and 11, the problem size stays constant and only the number of process groups performing these operations increases, which is why the CPU time stays approximately constant between these steps.

7 Summary and outlook

We have presented the finite-element library `hyper.deal`. It efficiently solves high-dimensional partial differential equations on complex geometries with high-order discontinuous Galerkin methods. It constructs a high-dimensional triangulation via the tensor product of distributed triangulations with dimensions up to three from the low-dimensional FEM library `deal.II` and solves the given partial differential equation with sum-factorization-based matrix-free operator evaluation. To reduce the memory consumption and the communication overhead, we use a new vector type, which is built around the shared-memory features from MPI-3.0.

We have compared the node-level performance with alternative algorithms, which are specialized for Cartesian and affine meshes or use collocation integration schemes. These studies revealed that loading less mapping data and reducing the number of sweeps is beneficial to improve the performance. We plan to look into the benefits of computing the low-dimensional mapping information on the fly and the benefits of increasing the cache locality during sum-factorization sweeps by a suitable hierarchical cache-oblivious blocking strategy.

Furthermore, we have studied the reduction of the working set of "vectorization over elements" by processing fewer cells in parallel as SIMD lanes would allow. Since we observed the benefit of this approach for 6D and polynomial orders higher than three, we intend to investigate "vectorization within an element" as an alternative vectorization approach in the future.

In this work, we have addressed neither overlapping communication and computation nor graph-based partitioning of the mesh. Both points will supposedly mitigate the scaling limits due to the huge amount of communication that we have encountered especially in 6D.

A APPENDIX

The following code snippets give implementation details on the new vector class, which is built around MPI-3.0 features (see Section 4). A new MPI communicator `comm_sm`, which consists of processes from the communicator `comm` that have access to the same shared memory, can be created via:

```
MPI_Comm_split_type(comm, MPI_COMM_TYPE_SHARED, rank, MPI_INFO_NULL, &comm_sm);
```

We recommend to create this communicator only once globally during setup and pass it to the vector. The following code snippet shows a simplified vector class. It is a container for the (local and ghosted) data and for the partitioner class as well as it provides linear algebra operations (not shown here).

```
class Vector
{
  Vector(std::shared_ptr<Partitioner> partitioner)
    : data_this((T *)malloc(0))
  {
    this->partitioner = partitioner;
    this->partitioner->initialize_dof_vector(data_this, data_others, win);
  }

  void
  clear() { MPI_Win_free(&win); /*free window*/ }

  void
  update_ghost_values_start() const {
    partitioner->update_ghost_values_start(this->data_this, this->data_others);
  }

  void
  update_ghost_values_finish() const {
    partitioner->update_ghost_values_finish(this->data_this, this->data_others);
  }

  std::shared_ptr<Partitioner> partitioner;

  MPI_Win                  win;
  mutable T *              data_this;
  mutable std::vector<T *> data_others;
};
```

The partitioner class stores the information on which cells the current process owns and which ghost faces it needs. Based on this information, it is able to determine which cells/faces are shared and to precompute communication patterns with other compute nodes. As a consequence, the partitioner class is able to allocate the memory of the vector class.

```
class Partitioner
{
  void
  reinit(const vector<unsigned int> &                    local_cells,
         const vector<pair<unsigned int, unsigned int>> &ghost_faces) {
    // based on the IDs of local cells and ghost faces set up the access to the
    // shared memory and the communication patterns to other compute nodes
    // as well as allocate buffer
  }

  void
  initialize_dof_vector(T *& data, vector<T *> &data_others, MPI_Win & win) {
    // configure shared memory
    MPI_Info info;
    MPI_Info_create(&info);
    MPI_Info_set(info, "alloc_shared_noncontig", "true");

    // allocate shared memory
    MPI_Win_allocate_shared((_local_size + _ghost_size) * sizeof(T), sizeof(T),
                            info, comm_sm, data, &win);
```

```
    // get pointers to the
    data_others.resize(size_sm);
    for (int i = 0; i < size_sm; i++)
      {
        int      disp_unit;
        MPI_Aint ssize;
        MPI_Win_shared_query(win, i, &ssize, &disp_unit, &data_others[i]);
      }

    data_this = data_others[rank_sm];
  }

  void
  update_ghost_values_start(T *&data_this, vector<T *> &data_others) const {
    MPI_Barrier(comm_sm); // sync processes in same shared memory communicator
    // start communication with other nodes and fill buffer (not shown)
  }

  void
  update_ghost_values_finish(T *&data_this, vector<T *> &data_others) const {
    // finish communication with other nodes(not shown)
    MPI_Barrier(comm_sm); // sync processes in same shared memory communicator
  }

  const MPI_Comm &comm; // global communicator
  const MPI_Comm &comm_sm; // shared memory communicator
  int rank_sm, size_sm;
};
```

The partitioner class only has to be created once and can be shared by multiple vectors. In our simulations, it was shared by three vectors, with only one vector being ghosted.

References

[1] Mark Adams, Marian Brezina, Jonathan Hu, and Ray Tuminaro. 2003. Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *Journal of Computational Physics* 188 (2003), 593–610. DOI:http://dx.doi.org/10.1016/S0021-9991(03)00194-3

[2] Martin S. Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. 2015. The FEniCS Project Version 1.5. *Archive of Numerical Software* 3, 100 (2015), 9–23. DOI:http://dx.doi.org/10.11588/ans.2015.100.20553

[3] Giovanni Alzetta, Daniel Arndt, Wolfgang Bangerth, Vishal Boddu, Benjamin Brands, Denis Davydov, Rene Gassmoeller, Timo Heister, Luca Heltai, Katharina Kormann, Martin Kronbichler, Matthias Maier, Jean-Paul Pelteret, Bruno Turcksin, and David Wells. 2018. The deal.II Library, version 9.0. *Journal of Numerical Mathematics* 26, 4 (2018), 173–183. DOI:http://dx.doi.org/10.1515/jnma-2018-0054

[4] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Johann Dahm, Veselin Dobrev, Yohann Dudouit, Aaron Fisher, Tzanio Kolev, Will Pazner, Mark Stowell, Vladimir Tomov, Johann Dahm, David Medina, and Stefano Zampini. 2020. MFEM: A Modular Finite Element Library. *arXiv* 1911.09220 (2020).

[5] Daniel Arndt, Wolfgang Bangerth, Denis Davydov, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Jean-Paul Pelteret, Bruno Turcksin, and David Wells. 2020. The deal.II finite element library: Design, features, and insights. *Computers and Mathematics with Applications* submitted (2020).

[6] Markus Bachmayr, Reinhold Schneider, and André Uschmajew. 2016. Tensor Networks and Hierarchical Tensors for the Solution of High-Dimensional Partial Differential Equations. *Foundations of Computational Mathematics* 16, 6 (01 Dec 2016), 1423–1472. DOI:http://dx.doi.org/10.1007/s10208-016-9317-9

[7] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. 2011. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Software* 38, 2 (dec 2011), 1–28. DOI:http://dx.doi.org/10.1145/2049673.2049678

[8] Peter Bastian, Christian Engwer, Jorrit Fahlke, Markus Geveler, Dominik Göddeke, Oleg Iliev, Olaf Ippisch, René Milk, Jan Mohring, Steffen Müthing, Mario Ohlberger, Dirk Ribbrock, and Stefan Turek. 2016. Hardware-Based Efficiency Advances in the EXA-DUNE Project. In *Software for Exascale Computing – SPPEXA 2013–2015*, Hans-Joachim Bungartz, Peter Neumann, and Wolfgang E. Nagel (Eds.). Springer International Publishing, Cham, 3–23.

[9] Gheorghe-Teodor Bercea, Andrew T.T. McRae, David A. Ham, Lawrence Mitchell, Florian Rathgeber, Luigi Nardi, Fabio Luporini, and Paul H.J. Kelly. 2016. A structure-exploiting numbering algorithm for finite elements on extruded meshes, and its performance evaluation in Firedrake. *arXiv* 1604.05937 (2016).

[10] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. 2011. p4est : Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing* 33, 3 (jan 2011), 1103–1133. DOI:http://dx.doi.org/10.1137/100791634

[11] Denis Davydov, Jean-Paul Pelteret, Daniel Arndt, Martin Kronbichler, and Paul Steinmann. 2020. A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid. *Internat. J. Numer. Methods Engrg.* in press (2020). DOI:http://dx.doi.org/10.1002/nme.6336

[12] Andreas Dedner, Robert Klöfkorn, Martin Nolte, and Mario Ohlberger. 2010. A generic interface for parallel and adaptive scientific computing: Abstraction principles and the DUNE-FEM module. *Computing* 90 (11 2010), 165–196. DOI:http://dx.doi.org/10.1007/s00607-010-0110-3

[13] Michel O. Deville, Paul F. Fischer, and Ernest H. Mund. 2002. *High-Order Methods for Incompressible Fluid Flow*. Vol. 9. Cambridge University Press, Cambridge.

[14] Niklas Fehn, Peter Munch, Wolfgang A. Wall, and Martin Kronbichler. 2019. Hybrid multigrid methods for high-order discontinuous Galerkin discretizations. *ArXiv* abs/1910.01900 (2019).

[15] Wei Guo and Yingda Cheng. 2016. A Sparse Grid Discontinuous Galerkin Method for High-Dimensional Transport Equations and Its Application to Kinetic Simulations. *SIAM Journal on Scientific Computing* 38, 6 (2016), A3381–A3409. DOI:http://dx.doi.org/10.1137/16M1060017

[16] Ammar Hakim, Greg Hammett, Eric L. Shi, and Noah Mandell. 2019. Discontinuous Galerkin schemes for a class of Hamiltonian evolution equations with applications to plasma fluid and kinetic problems. *arXiv* 1908.01814 (2019).

[17] Michael A Heroux, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M Willenbring, Alan Williams, Kendall S. Stanley, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, and Roger P. Pawlowski. 2005. An overview of the Trilinos project. *ACM Trans. Math. Software* 31, 3 (sep 2005), 397–423. DOI:http://dx.doi.org/10.1145/1089014.1089021

[18] James Juno, Ammar Hakim, Jason TenBarge, Eric L. Shi, and William Dorland. 2018. Discontinuous Galerkin algorithms for fully kinetic plasmas. *J. Comput. Phys.* 353 (2018), 110–147. DOI:http://dx.doi.org/10.1016/j.jcp.2017.10.009

[19] George Karypis and Vipin Kumar. 1998. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.*

[20] Dominic Kempf, René Hess, Steffen Müthing, and Peter Bastian. 2018. Automatic Code Generation for High-Performance Discontinuous Galerkin Methods on Modern Architectures arXiv : 1812 . 08075v1 [math . NA] 19 Dec 2018. (2018).

[21] Christopher A. Kennedy, Mark H. Carpenter, and R. Michael Lewis. 2000. Low-storage, explicit Runge–Kutta schemes for the compressible Navier–Stokes equations. *Applied Numerical Mathematics* 35, 3 (2000), 177–219. DOI:http://dx.doi.org/10.1016/s0168-9274(99)00141-5

[22] Katharina Kormann, Klaus Reuter, and Markus Rampp. 2019. A massively parallel semi-Lagrangian solver for the six-dimensional Vlasov–Poisson equation. *The International Journal of High Performance Computing Applications* 33, 5 (2019), 924–947. DOI:http://dx.doi.org/10.1177/1094342019834644

[23] Benjamin Krank, Niklas Fehn, Wolfgang A. Wall, and Martin Kronbichler. 2017. A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. *J. Comput. Phys.* 348, 1 November (nov 2017), 634–659. DOI:http://dx.doi.org/10.1016/j.jcp.2017.07.039

[24] Martin Kronbichler and Katharina Kormann. 2012. A generic interface for parallel cell-based finite element operator application. *Computers and Fluids* 63 (2012), 135–147. DOI:http://dx.doi.org/10.1016/j.compfluid.2012.04.012

[25] Martin Kronbichler and Katharina Kormann. 2019. Fast Matrix-Free Evaluation of Discontinuous Galerkin Finite Element Operators. *ACM Trans. Math. Softw.* 45, 3, Article Article 29 (Aug. 2019), 40 pages. DOI:http://dx.doi.org/10.1145/3325864

[26] Martin Kronbichler, Katharina Kormann, Niklas Fehn, Peter Munch, and Julius Witte. 2019. A Hermite-like basis for faster matrix-free evaluation of interior penalty discontinuous Galerkin operators. *arXiv preprint arXiv:1907.08492* (2019).

[27] Martin Kronbichler, Svenja Schoeder, Christopher Müller, and Wolfgang A. Wall. 2016. Comparison of implicit and explicit hybridizable discontinuous Galerkin methods for the acoustic wave equation. *Internat. J. Numer. Methods Engrg.* 106 (2016), 712–739. DOI:http://dx.doi.org/10.1002/nme.5137

[28] Martin Kronbichler and Wolfgang A Wall. 2018. A Performance Comparison of Continuous and Discontinuous Galerkin Methods with Fast Multigrid Solvers. *SIAM Journal on Scientific Computing* 40, 5 (2018), A3423–A3448. DOI:http://dx.doi.org/10.1137/16M110455X

[29] J. Markus Melenk, Klaus Gerdes, and Christoph Schwab. 2001. Fully discrete hp-finite elements: fast quadrature. *Computer Methods in Applied Mechanics and Engineering* 190, 32 (2001), 4339–4364. DOI:http://dx.doi.org/10.1016/S0045-7825(00)00322-4

[30] Steven A. Orszag. 1980. Spectral methods for problems in complex geometries. *J. Comput. Phys.* 37, 1 (aug 1980), 70–92. DOI:http://dx.doi.org/10.1016/0021-9991(80)90005-4

[31] Thomas Roehl, Jan Treibig, Georg Hager, and Gerhard Wellein. 2014. Overhead Analysis of Performance Counter Measurements. In *2014 43rd International Conference on Parallel Processing Workshops*, Vol. 2015-May. IEEE, Minneapolis, Minnesota, USA, 176–185. DOI:http://dx.doi.org/10.1109/ICPPW.2014.34

[32] Svenja Schoeder, Katharina Kormann, Wolfgang A. Wall, and Martin Kronbichler. 2018. Efficient explicit time stepping of high order discontinuous Galerkin schemes for waves. *SIAM Journal on Scientific Computing* 40, 6 (2018), C803–C826. DOI:http://dx.doi.org/10.1137/18M1185399

[33] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *2010 39th International Conference on Parallel Processing Workshops*, Wang-Chien Lee (Ed.). IEEE, Piscataway, NJ, 207–216. DOI:http://dx.doi.org/10.1109/ICPPW.2010.38

[34] Takayuki Umeda, Keiichiro Fukazawa, Yasuhiro Nariyuki, and Tatsuki Ogino. 2012. A Scalable Full-Electromagnetic Vlasov Solver for Cross-Scale Coupling in Space Plasma. *IEEE Transactions on Plasma Science* 40, 5 (May 2012), 1421–1428. DOI:http://dx.doi.org/10.1109/TPS.2012.2188141

[35] Tobias Weinzierl. 2019. The Peano software—parallel, automaton-based, dynamically adaptive grid traversals. *ACM Transactions on Mathematical Software (TOMS)* 45, 2 (2019), 1–41. DOI:http://dx.doi.org/10.1145/3319797

[36] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline : An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65. DOI:http://dx.doi.org/10.1145/1498765.1498785