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

Peter Munch and Katharina Kormann and Martin Kronbichler

October 29, 2021

Abstract

This work presents the efficient, matrix-free finite-element library `hyper.deal` for solving partial differential equations in two up 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 obtained with 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.

1 Introduction

Three-dimensional problems are today solved in great detail up to supercomputer scale with codes relying on domain decomposition. With the increase in computational power and the advance in algorithms, also the solution of higher-dimensional problems comes within the realm of the possible. With this contribution, we target 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 the modeling of 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 the application of phase space are, e.g., cosmic microwave background radiation or magnetic reconnection in the earth's magnetosphere.

In phase space applications, there are two distinct sets of variables, configuration space and velocity space, with the distribution function nonlinearly coupled to additional equations in the configuration 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 unstructured meshes as is commonly provided by finite-element method (FEM) libraries for grids in up to three dimensions. In the fusion community, the focus has shifted from simulations of the core to simulations of the edge and scrape-off layer, where the geometry of the problem gets more involved and density profiles get less homogeneous.

With the library `hyper.deal`, we are targeting phase-space simulation with complex geometry requirements in either dimension: Two separate possibly complex meshes describing configuration space and velocity space are defined based on the capabilities of a "low-dimensional" finite-element library, in our case the library `deal.II` [4, 5], and combined by taking their tensor product on the fly. The equations we have in mind are advection-dominated, which is why we mostly focus on the discretization of the advection equation in this paper.

The presented concept is not limited to the description of the phase space. Possible applications of a high-dimensional FEM library, like `hyper.deal`, could for instance include three-dimensional problems that involve a low-dimensional parameter space or low-dimensional Fokker–Planck-type equations, e.g., the Black–Scholes equation for option pricing in mathematical finance.

1.1 Related work

While the solution of partial differential equations (PDE) on complex domains in up to three dimensions is a well-studied problem, PDEs 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]. In [37], Helmholtz equations on adaptive structured grids up to $d = 4$ are studied. Higher-dimensional problems are also solved based on sparse grids [17] or low-rank tensors [6]. However, these techniques require a certain low-rank structure of the solution. In the plasma community, some specialized codes exist that target gyrokinetic or kinetic equations. The `Gekyll` code [18] is a discontinuous Galerkin solver for plasma physics applications, and a fully kinetic version for Cartesian grids was presented in [22]. The authors use higher-order serendipity elements to reduce the number of degrees of freedom.

The specifics of domain decomposition in higher dimensions have only been studied recently. In [27], a six-dimensional domain decomposition for a semi-Lagrangian solver on a Cartesian grid of the Vlasov–Poisson system was investigated and the high demands on memory transfer between neighboring processes due to the increased surface-to-volume ratio with increasing dimensionality were highlighted. The parallelization of a similar algorithm has also been addressed in [42]. However, the domain decomposition is limited to configuration space in that work, which poses a strong limit to the scalability of the implementation.

The most widespread algorithm for solving problems in high-dimensional phase space is a solution based on the particle-in-cell method. While this method scales very well to high dimensions and features automatic adaptivity in velocity space, it suffers from inherent noise. Grid-based codes, as proposed in this work, may provide a promising alternative, as discussed, e.g., in [15].

1.2 Our contribution

This work shows a way to extend a low-dimensional general-purpose high-order matrix-free FEM library to high dimensions. 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. In particular, we show how to create a high-dimensional triangulation by taking the tensor product of possibly unstructured partitioned triangulations from a low-dimensional FEM library. This way to construct a triangulation enables us to re-use information related to the low-dimensional finite-element space, quadrature, and mapping and to combine the information on the fly.

We evaluate cell and face integrals with a matrix-free approach, using highly-optimized sum-factorization kernels: This involves loading the portion of the solution vector describing the element unknowns and computing derived quantities such as values or gradients at the quadrature points. We have investigated different sequences to loop over cells and faces and show the advantage of looping over all cells and processing in direct succession all $2d$ faces of a cell in an element-centric manner (ECL). In regard to parallelization, we discuss the usage of explicit SIMD vectorization over multiple elements, domain decomposition based on the domain decomposition of the low-dimensional triangulations, and shared-memory parallelization using `MPI-3.0` features, all taking hardware characteristics into account.

The concepts described 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` [4] to high dimensions. Analyses of both the node-level performance and strong/weak scaling conducted for this library confirm the suitability of the proposed concepts for solving partial differential equations in high dimensions.

The remainder of this work is organized as follows. In Section 2, we introduce the model problem equation and discretize it with a skew-symmetric discontinuous Galerkin approach. Section 3 introduces the concept of a tensor product of partitioned low-dimensional meshes and details its implementation for phase space. In Section 4, we describe a shared-memory vector based on `MPI-3.0`, which keeps the memory overhead due to ghost regions to a minimum without the requirement to add a second parallelization concept. 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 bench-

mark problem from plasma physics. Finally, Section 7 summarizes our conclusions and points to further research directions.

2 Discretization with discontinuous Galerkin methods

2.1 Model problem

For the analysis of our algorithms and implementations, we consider the advection equation on the d-dimensional domain Ω:

$$\frac{\partial f}{\partial t} + \nabla \cdot (\vec{a}(t, \vec{x}) f) = 0 \qquad \text{on} \qquad \Omega \times [0, t_{\text{final}}], \tag{1}$$

with $\vec{a}(t, \vec{x})$ being the 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 will discuss the Vlasov–Poisson equation in Section 6, where we combine the advection solver from the library `hyper.deal` and a Poisson solver based on `deal.II`.

2.2 Discontinuous Galerkin discretization of the advection equation

High-order discontinuous Galerkin (DG) methods are attractive methods for solving hyperbolic partial differential equations like (1) due to their high accuracy in terms of dispersion and dissipation, while maintaining geometric flexibility through unstructured grids [21]. The skew-symmetric DG discretization of (1) reads as follows [26]:

$$\left(g, \frac{\partial f}{\partial t}\right)_{\Omega^{(e)}} = \left(g, -\beta(\vec{a} \cdot \nabla f)\right)_{\Omega^{(e)}} + \left(\nabla g, (1-\beta)\vec{a} f\right)_{\Omega^{(e)}} - \left\langle g, \vec{n} \cdot (\vec{a} f)^* - \beta u^-(\vec{n} \cdot \vec{a})\right\rangle_{\Gamma^{(e)}} \tag{2}$$

with the element domain $\Omega^{(e)}$, g the test function, and $(\vec{a} f)^*$ being the numerical flux, like a central ($\alpha = 0$) or an upwind flux ($\alpha = 1$):

$$(\vec{a} f)^* = \frac{1}{2}\left((f^- + f^+)(\vec{n} \cdot \vec{a}) + (f^- - f^+)|\vec{n} \cdot \vec{a}| \cdot \alpha\right). \tag{3}$$

The factor β controls the formulation of the flux: $\beta = \frac{1}{2}$ represents the skew-symmetric version, whereas $\beta = 0$ corresponds to the conservative DG method, see also [26]. 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, i.e., $\int_\Omega \mathrm{d}\Omega = \int_{\Omega^{(e)}} |\mathcal{J}| \, \mathrm{d}\Omega^{(e)}$ and $\nabla = \mathcal{J}^{-T} \nabla_{\vec{\xi}}$, 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 use a tensor product of 1D nodal polynomials with nodes in the Gauss–Lobatto points. These nodal polynomials are chosen to ensure minimal data access on faces [39]. 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 for interpolation (collocation setup [13]). The resulting semi-discrete system has the following form:

$$\mathcal{M} \frac{\partial \vec{f}}{\partial t} = \mathcal{A}(\vec{f}, t) \qquad \leftrightarrow \qquad \frac{\partial \vec{f}}{\partial t} = \mathcal{M}^{-1} \mathcal{A}(\vec{f}, t), \tag{4}$$

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 with blocks equal to the number of unknowns per element in the case of the consistent Gauss quadrature. For 2D and 3D high-order DG methods, efficient matrix-free operator evaluations for the individual operators $\mathcal{A}$ [29, 30] and

Table 1: Estimated working set of different stages of a matrix-free evaluation of the advection operator for ECL with shape functions of polynomial degree k and n_q quadrature points in each direction: (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	$>$	$\max\left((k+1)^d,\, n_q^d\right)$
(2)	derived quantities	$>$	$(d+1)\cdot n_q^d$
(3)	flux computation	$\gg$	$(k+1)^d + 2\cdot d\cdot(k+1)^{d-1}$

Table 2: Estimated minimal memory consumption for d-dimensional advection simulations with N degrees of freedom on p processes, $n_q = k+1$, 3 vectors (one with ghost values), and pre-computed mapping data.

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

$\mathcal{M}^{-1}$ [32] as well as for the merged operator $\mathcal{M}^{-1}\mathcal{A}$ are known in the context of fluid mechanics [28], structural mechanics [11], and acoustic wave propagation [39]. Discontinuous Galerkin methods and matrix-free operator evaluation kernels are part of many low-dimensional general-purpose FEM libraries nowadays [3, 35, 24, 40, 5].

As the conclusion of this subsection, Table 1 gives a rough estimate of the working sets for the matrix-free evaluation of the advection operator at different stages during cell and face integrals. Table 2 shows the estimated minimal memory consumption of a complete simulation, considering vectors including ghost values and precomputed mapping information. The number of ghost degrees of freedom, computed under the assumption of a partitioning of the domain into cubes, also gives an estimate for the amount of data to be communicated.

2.3 Challenges

The finite-element formulations are dimension-agnostic. However, we face the following major challenges tied to the lack of libraries designed for high dimensions. For example, the library `deal.II` and its back end for handling distributed triangulations, `p4est` [10], are limited to dimensions up to three. These libraries can not easily be extended for high dimensions due to the following specific difficulties:

1. **Significant memory overhead due to ghost values and mapping**: In high dimensions, solution vectors (at least 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, necessary 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 of the scalar solution 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. Even if one would decide not to pre-compute the mapping information, one would need to store the coordinate of each vertex $\mathcal{O}(d)$, what might also already require too much memory, and to solve for the Jacobian $\mathcal{J}\in\mathbb{R}^{d\times d}$ at each quadrature point, which is a $\mathcal{O}(d^3)$ operation.

2. **Increased ghost-value exchange due to increased surface-to-volume ratio**: The communication amount scales with $2\cdot d\cdot N^{(d-1)/d}\cdot p^{1/d}$ for a hypercube-shaped partition (cf. Table 2). According to [30], the MPI ghost-value exchange already leads to a noticeable share of time in

purely MPI-parallelized applications (30% for 3D Laplacian) in comparison to the highly efficient matrix-free operator evaluations if computations are performed on a single compute node for 3D problems. 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.

3. **Decreased efficiency of the operator evaluation due to working sets exceeding the cache capacities**: The working set of a cell with shape functions of polynomial degree k and n_q quadrature points in each direction 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. This can lead to a significant drop in performance once the data has to be streamed from the slow main memory.

This work shows how these problems can be mitigated by certain design choices: 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 possible to reuse much of the infrastructure available in a low-dimensional library, such as `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 particularly in high dimensions. 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. Therefore, we propose a novel shared-memory implementation of finite-element-type vectors, which 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. To reduce the working set with cross-element vectorization, we also allow to use narrower SIMD registers containing data from fewer elements than the given instruction-set extensions allow. For example, we use `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

The main idea is to combine two meshes of a low-dimensional general-purpose finite-element library to solve problems in up to six dimensions. We therefore work with a computational domain defined as a tensor product of two domains $\Omega := \Omega_{\vec{x}} \otimes \Omega_{\vec{v}}$. Since our sample application is an advection equation in phase space, separating the meshes in configuration space and in velocity space is natural. As a consequence, we use the indices $\vec{x}$ and $\vec{v}$ for the two parts of the dimensions. The boundary of the high-dimensional domain is then described by $\Gamma := (\Gamma_{\vec{x}} \otimes \Omega_{\vec{v}}) \cup (\Omega_{\vec{x}} \otimes \Gamma_{\vec{v}})$.

The concept of obtaining higher-dimensional triangulations by taking the tensor product of low-dimensional triangulations is generic and could in principle be built upon any general-purpose FEM library, such as MFEM [3], DUNE [12], FEniCS [2], or Firedrake [9, 40]. Our description is, however, specialized to the implementation in `hyper.deal` that is constructed on top of the `deal.II` library. Also, we use some of the naming conventions from the `deal.II` project. In Subsection 3.6, we list requirements a FEM library needs to fulfill to be extensible.

For a tensor-product domain, the discretized advection equation (2) can be reformulated and simplified for the phase space. We can exploit the fact that the Jacobian matrix $\mathcal{J}$ is a block-diagonal matrix in phase space,

$$\mathcal{J} = \begin{pmatrix} \mathcal{J}_{\vec{x}} & 0 \\ 0 & \mathcal{J}_{\vec{v}} \end{pmatrix}, \tag{5}$$

with the blocks being the respective matrices of the $\vec{x}$-space and the $\vec{v}$-space. Hence, the inverse $\mathcal{J}^{-1}$ is the inverse of each of its blocks. 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 holds for $\vec{x}$-space faces: Due to $\vec{n}^\top = (\vec{n}_x^\top, 0)$ for $\vec{x}$-space faces, $\vec{n} \cdot \vec{a} = \vec{n}_{\vec{x}} \cdot \vec{a}_{\vec{x}}$. Analogously, $\vec{n} \cdot \vec{a} = \vec{n}_{\vec{v}} \cdot \vec{a}_{\vec{v}}$ is true for $\vec{v}$-space faces. Hence, the specialization

Algorithm 1: Element-centric loop

```
   /* loop over all cells (cell pairs)                                          */
1  foreach (c_x, c_v) ∈ C_x⃗ × C_v⃗ do
2      process_cell(c_x, c_v)
       /* loop over all x-faces (face-cell pairs)                               */
3      for 0 ≤ i < 2 · d_x do
4          process_face(face(c_x, i), c_v);   /* face(c, i) returns the i-th face of the cell c */
       /* loop over all v-faces (cell-face pairs)                               */
5      for 0 ≤ i < 2 · d_v do
6          process_face(c_x, face(c_v, i))
```

Algorithm 2: Face-centric loop (boundary faces not shown)

```
   /* loop over all cells (cell pairs)                                          */
1  foreach c ∈ C_x⃗ × C_v⃗ do
2      process_cell(c)
   /* loop over all x-faces (face-cell pairs)                                   */
3  foreach f ∈ I_x⃗ × C_v⃗ do
4      process_face(f)
   /* loop over all v-faces (cell-face pairs)                                   */
5  foreach f ∈ C_x⃗ × I_v⃗ do
6      process_face(f)
```

of (2) solved by `hyper.deal` is:

$$\begin{aligned}\left(g, \frac{\partial f}{\partial t}\right)_{\Omega^{(e)}} &= \left(g, -\beta \begin{pmatrix} \vec{a}_{\vec{x}}\mathcal{J}_{\vec{x}}^{-1} \\ \vec{a}_{\vec{v}}\mathcal{J}_{\vec{v}}^{-1} \end{pmatrix} \nabla_\xi f\right)_{\Omega^{(e)}} + \left(\nabla_\xi g, \begin{pmatrix} \mathcal{J}_{\vec{x}}^{-1}\vec{a}_{\vec{x}} \\ \mathcal{J}_{\vec{v}}^{-1}\vec{a}_{\vec{v}} \end{pmatrix} (1-\beta) f\right)_{\Omega^{(e)}} \\ &+ \left(g,\, \vec{n}_{\vec{x}} \cdot (\vec{a}_{\vec{x}} f)^* - \beta f^- (\vec{n}_{\vec{x}} \cdot \vec{a}_{\vec{x}})\right)_{\Gamma_{\vec{x}}^{(e)} \otimes \Omega_{\vec{v}}^{(e)}} + \left(g,\, \vec{n}_{\vec{v}} \cdot (\vec{a}_{\vec{v}} f)^* - \beta f^- (\vec{n}_{\vec{v}} \cdot \vec{a}_{\vec{v}})\right)_{\Omega_{\vec{x}}^{(e)} \otimes \Gamma_{\vec{v}}^{(e)}}.\end{aligned} \tag{6}$$

We proceed with explaining practical implementation details of our proposed approach.

3.1 Triangulation

Naturally, both domains $\Omega_{\vec{x}}$ and $\Omega_{\vec{v}}$ can be meshed separately. As a consequence, the final triangulation results from the tensor product of the two triangulations $\mathcal{T}_{\vec{x}}$ and $\mathcal{T}_{\vec{v}}$, which might come from a low-dimensional library (visualized in Figure 1),

$$\mathcal{T} := \mathcal{T}_{\vec{x}} \otimes \mathcal{T}_{\vec{v}}. \tag{7}$$

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

$$\mathcal{C} := \mathcal{C}_{\vec{x}} \otimes \mathcal{C}_{\vec{v}}, \qquad \mathcal{I} := (\mathcal{I}_{\vec{x}} \otimes \mathcal{C}_{\vec{v}}) \cup (\mathcal{C}_{\vec{x}} \otimes \mathcal{I}_{\vec{v}}), \qquad \mathcal{B} := (\mathcal{B}_{\vec{x}} \otimes \mathcal{C}_{\vec{v}}) \cup (\mathcal{C}_{\vec{x}} \otimes \mathcal{B}_{\vec{v}}), \tag{8}$$

where $\mathcal{C}_{\vec{x}/\vec{v}}$, $\mathcal{I}_{\vec{x}/\vec{v}}$, and $\mathcal{B}_{\vec{x}/\vec{v}}$ are the collection of cells, inner faces, and boundary faces of the low-dimensional triangulations $\mathcal{T}_x$ or $\mathcal{T}_v$. With this concept, we never need to explicitly construct the high-dimensional triangulation $\mathcal{T}$, but can extract the relevant information on the fly. We simply loop over all possible cell-cell and cell-face pairs in the form of nested loops. Algorithms 1–2 show two possible ways to loop over all high-dimensional cells and faces. While "element-centric loops" (ECL) loop over all cells and process all $2d$ faces of a cell in direct succession, involving only the test functions of the respective cell (i.e., visit interior faces twice), "face-centric loops" (FCL) visit all faces only once in a separate loop (with test functions from both sides of an interior face).

The way we create the high-dimensional triangulation restricts the possibilities of mesh refinement to the two spaces separately from each other. We defer investigations on how to allow local refinement on a part of the tensor product to future work.

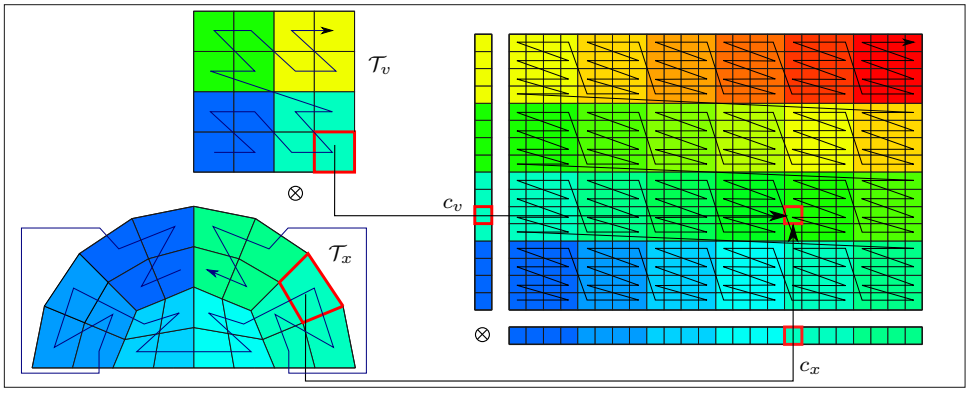


Figure 1: On-the-fly mesh generation of a high-dimensional distributed triangulation (right) by taking the tensor product of two low-dimensional triangulations (left) from a low-dimensional library (for a hypothetical setup of 24 partitions and of a 6×4 partitioning of the 4D space). Cells are ordered lexicographically within a process (as are the processes themselves), leading to the depicted global enumeration of the cells.

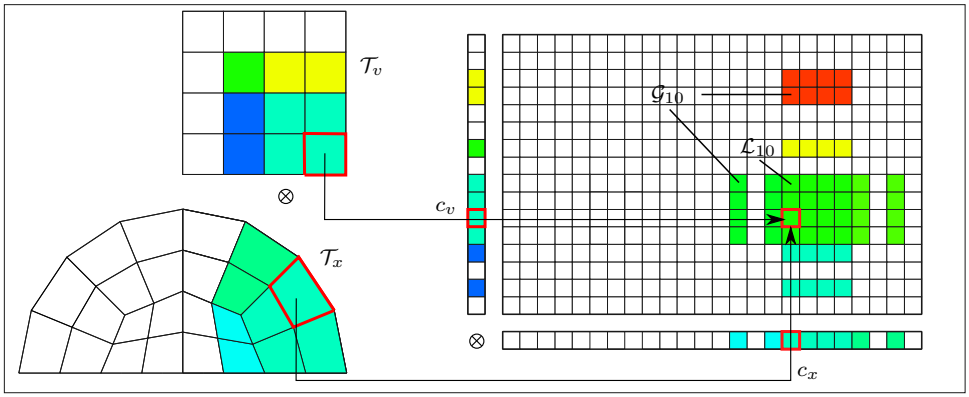


Figure 2: Local view of an arbitrary process for the hypothetical setup of Figure 1 (here: process 10). Local cells $\mathcal{L}_{10}$ result as tensor product of local cells from each low-dimensional triangulation and ghost cells $\mathcal{G}_{10}$ as tensor product of one local and one ghost cell.

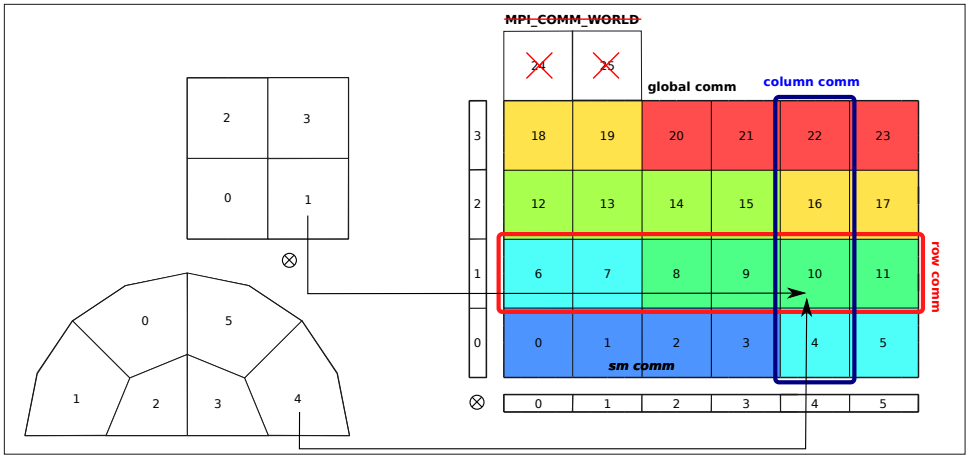


Figure 3: Four MPI communicators (constructed from `MPI_COMM_WORLD` with 26 processes and shared-memory domains of size 4) used in `hyper.deal` for the hypothetical setup of Figure 1: `global_comm` collects all non-empty partitions; `row_/column_comm` collects processes owning the same partitions of $\mathcal{T}_v/\mathcal{T}_x$; `sm_comm` collects processes on the same shared-memory domain.

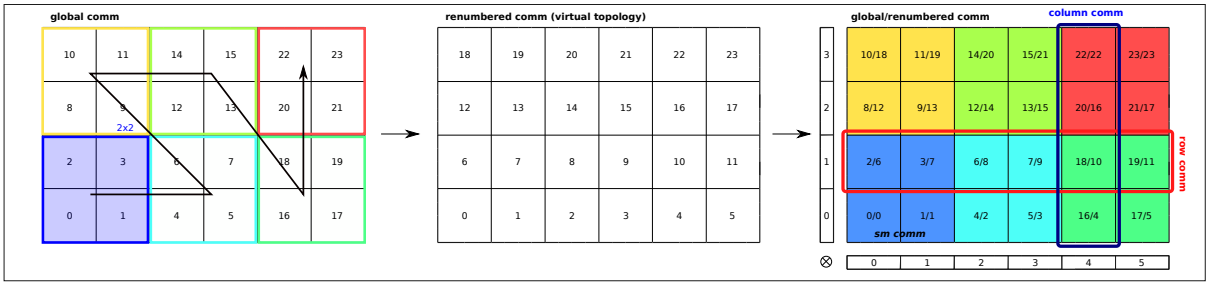


Figure 4: Renumbering of ranks in the global communicator (via `MPI_Comm_split`): For a hypothetical setup of 26 ranks in `MPI_COMM_WORLD` and a 6×4 partition, processes are grouped in 2×2 blocks, which are ordered along a z-curve. Based on this new global communicator, the partitioning, as described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

3.2 Domain decomposition

We split up the low-dimensional triangulations $\mathcal{T}_{\vec{x}}$ and $\mathcal{T}_{\vec{v}}$ independently into $p_{\vec{x}}$ and $p_{\vec{v}}$ partitions with

$$\mathcal{T}_{\vec{x}} = \biguplus_{0 \le i < p_{\vec{x}}} \mathcal{T}_{\vec{x}}^i \qquad \text{and} \qquad \mathcal{T}_{\vec{v}} = \biguplus_{0 \le j < p_{\vec{v}}} \mathcal{T}_{\vec{v}}^j \tag{9}$$

and a halo of ghost cells. Ghost faces are shared by locally-owned cells and ghost cells. We will use the terms "ghost cell" and "ghost face" interchangeably, although we only allocate memory for the degrees of freedom on the faces.

The partition of the phase space belonging to rank $r(i,j)$ is constructed by $\mathcal{T}^{r(i,j)} := \mathcal{T}_{\vec{x}}^i \otimes \mathcal{T}_{\vec{v}}^j$, where ranks are enumerated lexicographically according to $r(i,j) := j \cdot p_{\vec{x}} + i$ with the $\vec{x}$-rank i being the fastest running index. As a consequence, a quasi-Cartesian partitioning is obtained (see Figure 1). 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 3). This is achieved by a subcommunicator of `MPI_COMM_WORLD`, which we will call `global_comm`(unicator) in the following.

The following relationship holds:

$$\mathcal{T}_{\vec{x}} \otimes \mathcal{T}_{\vec{v}}^j = \biguplus_{f(i,j)/n_{\vec{x}} = j} \mathcal{T}^{f(i,j)} \qquad \text{and} \qquad \mathcal{T}_{\vec{x}}^i \otimes \mathcal{T}_{\vec{v}} = \biguplus_{f(i,j)\%n_{\vec{x}} = i} \mathcal{T}^{f(i,j)} \tag{10}$$

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. This is important in mathematical operations like $\int \mathrm{d}\Omega_{\vec{x}}$ and $\int \mathrm{d}\Omega_{\vec{v}}$, hence we make the subsets of processes available via the MPI communicators `column_comm` and `row_comm` similarly as in distributed matrix-matrix multiplication implementations [43].

We enumerate cells within a subdomain lexicographically to get a global numbering, as depicted in Figure 1. 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.

Placing ranks according to $\lfloor r(i,j)/p_{node} \rfloor$ 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 3). This results in a bad shape of the union of subpartitions belonging to the same compute node, leading to decreased benefit of shared memory. In order to improve the placing of subpartitions onto the compute nodes without having to change the function r, we block within a Cartesian virtual topology (see Figure 4), 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, it should be emphasized that the presented partitioning approach delivers good results regarding communication amount and communication pattern if the low-dimensional triangulations $\mathcal{T}_{\vec{x}}$ and $\mathcal{T}_{\vec{v}}$ have been partitioned well by the underlying low-dimensional library. Depending on the given mesh, a space-filling curve approach [44, 10, 7] or a graph-based approach [23, 20] might be

beneficial for this. Nevertheless, it should be noted that, generally, PDEs with surface data exchange become increasingly heavier on communication as the dimension increases, including a significant amount of inter-node communication even for an optimal communication layout.

3.3 Elements, degrees of freedom, quadrature, and mapping

On each element of the mesh, we use d-dimensional tensor-product shape functions of polynomial degree k, based on Gauss-Lobatto support points,

$$\mathcal{P}_k^d = \underbrace{\mathcal{P}_k^1 \otimes ... \otimes \mathcal{P}_k^1}_{\times d}. \tag{11}$$

The unknowns are discontinuous across cells with $(k+1)^d$ unknowns per cell for a scalar field f. The total number of degrees of freedom (DoF) for $|\mathcal{C}|$ cells is

$$N = |\mathcal{C}| \cdot (k+1)^d. \tag{12}$$

In DG, the unknowns are coupled via fluxes. This necessitates the access to the degrees of freedom of neighboring cells. The dependency region for computing all contributions of a cell with a nodal basis and nodes on the faces is

$$\underbrace{(k+1)^d}_{\text{cell}} + \underbrace{2 \cdot d \cdot (k+1)^{d-1}}_{\text{faces}}. \tag{13}$$

This expression includes all unknowns of the cell and the unknowns residing on faces of the $2 \cdot d$ neighboring cells, as shown in Figure 5. The dependency region influences how much data should be cached and how much data has to be communicated.

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

$$\mathcal{Q}_{n_q}^d = \underbrace{\mathcal{Q}_{n_q}^1 \otimes ... \otimes \mathcal{Q}_{n_q}^1}_{\times d} \tag{14}$$

with the evaluation points given as $\vec{x}_q^d = (x_{q_1}^1, ..., x_{q_d}^1)^T$ and the quadrature weights as $w_q^d = w_{q_1}^1 \cdot ... \cdot w_{q_d}^1$. We support the Gauss–Legendre family of quadrature rules (see Figure 5), which are exact for polynomials of degree $2n_q - 1$ and require an interpolation operation from the Gauss–Lobatto to the Gauss–Legendre points. As an alternative, we support the Gauss–Lobatto family of quadrature rules, which allow a collocation setup, i.e, do not require a basis change. However, they are only exact for polynomials of degree $2n_q - 3$.

To be able to evaluate (6), the Jacobian matrices $\mathcal{J}_{\vec{x}} \in \mathbb{R}^{d_{\vec{x}} \times d_{\vec{x}}}$ and $\mathcal{J}_{\vec{v}} \in \mathbb{R}^{d_{\vec{v}} \times d_{\vec{v}}}$ and their determinants are needed at each cell quadrature point. At face quadrature points, we require the Jacobian determinant and the face normals $\vec{n}_{\vec{x}} \in \mathbb{R}^{d_{\vec{x}}}$ and $\vec{n}_{\vec{v}} \in \mathbb{R}^{d_{\vec{v}}}$. If these quantities are precomputed once during initialization, this leads, as shown in Table 3, to an additional memory consumption per unknown, which is significantly less than if the tensor-product structure would not be exploited.

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.

3.4 Matrix-free operator evaluation

Cell and face integrals in (6) can be efficiently evaluated via the effect of the operator on element vectors on the fly. For example, for cell integrals the following five steps are performed:

1. gather $(k+1)^d$ cell-local values f_i,
2. interpolate values and gradients to quadrature points f_q (if no collocation setup is used),

Table 3: Comparison of the memory consumption (in doubles) of the mapping data (per quadrature point) due to the Jacobian matrices 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:	$\frac{(k+1)^{d_{\vec{x}}} \cdot d_{\vec{x}}^2 + (k+1)^{d_{\vec{v}}} \cdot d_{\vec{v}}^2}{(k+1)^{d_{\vec{x}}+d_{\vec{v}}}}$	$(d_{\vec{x}} + d_{\vec{v}})^2$	$0.28 \ll 36$

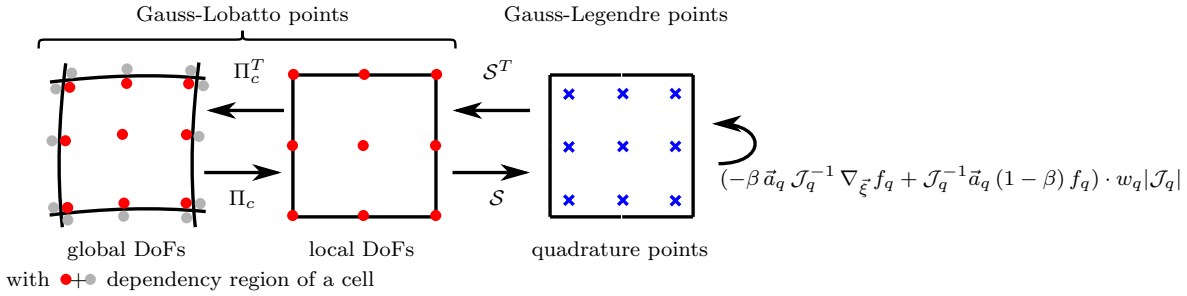


Figure 5: Visualization of the 5 steps of a matrix-free cell-integral evaluation for polynomial degree $k=2$ and number of quadrature points $n_q=3$.

3. perform the operations $(-\beta\,\vec{a}_q\,\mathcal{J}_q^{-1}\,\nabla_{\vec{\xi}}\,f_q + \mathcal{J}_q^{-1}\vec{a}_q\,(1-\beta)\,f_q)\cdot w_q|\mathcal{J}_q|$ at each quadrature point,
4. test with the value and gradient of the shape functions, and
5. write back the local contributions into the global vector.

The five steps are visualized in Figure 5.

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 $k+1=n_q$) [30]. This operation is known as sum factorization and has its origin in the spectral-element community [13, 34, 36]. Similarly, the testing with the gradient of the shape functions can be performed efficiently with $2d$ sweeps [30].

In the case of FCL, we perform the same five steps as in the case of cell integrals on the faces in a separate loop. In contrast, ECL allows to perform some optimizations, although we visit each face twice and, as a consequence, have to evaluate fluxes between neighboring cells twice for each side of an interior face. For example, values already interpolated to the cell quadrature points can be interpolated to a face with a single 1D sweep. Furthermore, entries in the solution vector are written back to main memory exactly once in the case of ECL. This makes the algorithm more cache-friendly. Moreover, no synchronization between threads is needed while accessing the solution vector in the case of ECL, since each entry of the solution vector is accessed only exactly once. This makes ECL particularly suitable for shared-memory parallelization, a key ingredient for the reduction of communication. A further advantage of ECL is that the application of a cell-wise implementation of the inverse mass matrix [32] can be merged into the application of the advection operator, avoiding another access to the global solution vector. For a more detailed discussion on ECL, see [30].

Algorithm 3 shows the pseudocode of a possible matrix-free merged advection and inverse-mass-matrix operator evaluation in the context of ECL. Lines 3, 7, 9, 13, 15, 19 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 further reduce the working set by loop blocking [30].

Since we loop over cell-cell and cell-face pairs, 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 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 3: Element-centric loop for arbitrary operators and DG integration of a cell batch for advection operator evaluation for vectorization over elements

`/* loop over all cell pairs */`

1 **foreach** $c := (c_{\vec{x}}, c_{\vec{v}}) \in \mathcal{C}_{\vec{x}} \times \mathcal{C}_{\vec{v}}$ **do**

`/* step 1: gather values (Array of structs (AoS) → struct of arrays (SoA)) */`

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

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

3 - interpolate local vector values $\vec{u}^{(c)}$ onto quadrature points, $u_h^c(\vec{\xi}_q) = \sum_i \phi_i u_i^{(c)}$ and compute gradients $\nabla_{\vec{x}} u_h^c(\vec{\xi}_q)$ in reference coordinate system

4 **foreach** *quadrature index* $q = (q_{\vec{x}}, q_{\vec{v}})$ **do**

5 - compute convective-term contribution
$\vec{b}_i = -\beta \vec{a}_{\vec{x}} \left(\hat{x}^{(c_{\vec{x}})}(\vec{\xi}_{q_{\vec{x}}}),\ \hat{x}^{(c_{\vec{v}})}(\vec{\xi}_{q_{\vec{v}}}) \right) \mathcal{J}_{(c_{\vec{x}})}^{-1} \nabla_{\vec{\xi}} u_h^{(c)}(\vec{\xi}_q) \underbrace{|\mathcal{J}_{q_{\vec{x}}}||\mathcal{J}_{q_{\vec{v}}}| w_{q_{\vec{x}}} w_{q_{\vec{v}}}}_{\text{“}|\mathcal{J}_{(q)}|w_q\text{”}}$

6 - prepare integrand on each quadrature point by computing
$\vec{t}_q = (1-\beta) \mathcal{J}_{(c_{\vec{x}})}^{-1} \vec{a}_{\vec{x}} \left(\hat{x}^{(c_{\vec{x}})}(\vec{\xi}_{q_{\vec{x}}}),\ \hat{x}^{(c_{\vec{v}})}(\vec{\xi}_{q_{\vec{v}}}) \right) u_h^{(c)}(\vec{\xi}_q) \underbrace{|\mathcal{J}_{q_{\vec{x}}}||\mathcal{J}_{q_{\vec{v}}}| w_{q_{\vec{x}}} w_{q_{\vec{v}}}}_{\text{“}|\mathcal{J}_{(q)}|w_q\text{”}}$; `/* buffer` $\vec{b}$ `*/`

7 - evaluate local integrals by quadrature $b_i = b_i + \left(\nabla_{\vec{x}} \phi_i^{co}, \vec{c}_{\vec{x}} u_h^{(c)} \right)_{\Omega_{(e)}} \approx b_i + \sum_q \nabla_{\vec{x}} \phi_i^{co}(\vec{\xi}_q) \cdot \vec{t}_q$

8 **end**

9 repeat lines 3-8 for v space

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

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

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

12 - if interior face, gather values from neighbor Ω_{e^+} of current face

13 - interpolate u^+ onto face quadrature points

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

15 - evaluate local integrals related to cell c by quadrature and add into cell contribution b_i

16 **end**

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

17 **foreach** *quadrature index* $q = (q_{\vec{x}}, q_{\vec{v}})$ **do**

18 - prepare integrand at each quadrature point by computing $t_q = b_i w_{q_{\vec{x}}}^{-1} w_{q_{\vec{v}}}^{-1}$

19 - transformation from collocation space to the Gauss-Lobatto space used for vector storage:
$y_i^{(c)} = \sum_q \tilde{\phi}_{iq} \cdot \vec{t}_q$ with $\tilde{\phi}_{iq} = \mathcal{V}_{iq}^{-1}$ with $\mathcal{V}_{iq} = \phi_i(\vec{\xi}_q)$

20 **end**

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

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

22 **end**

3.5 Implementation of operator evaluations with hyper.deal

The library `hyper.deal` provides classes that are built around `deal.II` classes and contain inter alia utility functions needed in Algorithm 3. 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 classes in `hyper.deal` and classes in `deal.II` 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) include functions to read and write cell-/face-local values from a global vector as well as operations at the quadrature points. As an example, Figure 6 shows the implementation of the `hyperdeal::FEEvaluation::submit_gradient()` method, which uses two instances of the `deal.II` class with the same name, one for $\vec{x}$- and one for $\vec{v}$-space, for the evaluation of $f(\vec{u}) = \mathcal{J}_{c,q}^{-1} |\mathcal{J}_q| w_q \vec{u}$ for $\vec{u} \in \mathbb{R}^d$.

We process a batch of v_{len} cells or faces in a “vectorization over elements” fashion. We do not operate

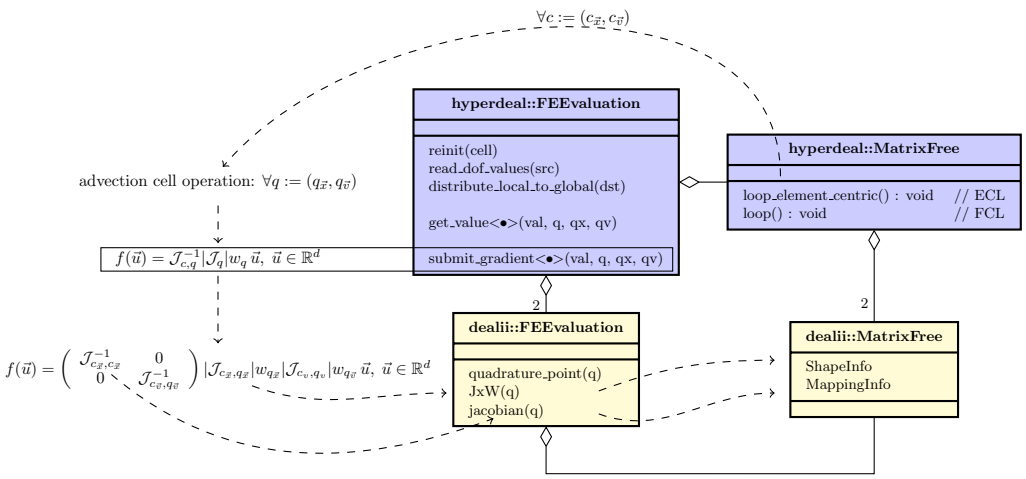


Figure 6: Class diagram of a part of the matrix-free infrastructure of `hyper.deal`. It presents how classes from `hyper.deal` (namespace `hyperdeal`— highlighted in blue) and from `deal.II` (namespace `dealii`— highlighted in yellow) relate to each other. Only the `hyper.deal` methods are shown that are relevant for the evaluation of one term of the advection cell integral and the `deal.II` methods that are used in those. The methods `read_dof_values()` and `distribute_local_to_global()` are, on the one hand, responsible for gathering from and scattering to the solution vector as well as, on the other hand, for the transformation from an array-of-struct format (AoS) to a struct-of-array format (SoA), as needed by the integration routines, and for the transformation back.

directly on the primitive types `double`/`float` but on structs built around intrinsic instructions[1], with each vector lane dedicated to a separate cell of mesh. The maximal number of vector lanes depends on the given hardware; with AVX-512 instruction-set extension, as most modern Intel-based processors have, 8 doubles (i.e., 8 cells in the context of our application) can be processed by a single instruction.

Currently, we vectorize only over elements in $\vec{x}$-space, whereas the $\vec{v}$-space is not vectorized. The reason is that the Vlasov–Maxwell and the Vlasov–Poisson models contain heavy operations on the full phase space and on the $\vec{x}$-space, respectively, which benefit from vectorization over $\vec{x}$. As a consequence, the data structures are already laid out correctly for an efficient matrix-free solution of the lower-dimensional problem.

3.6 Requirements to a low-dimensional finite-element library

As a conclusion of this section, we list the requirements on a low-dimensional finite-element library for the proposed concept:

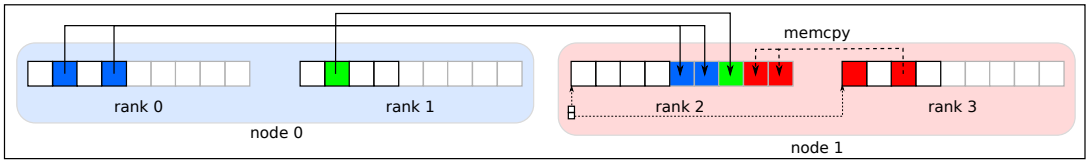
1. The low-dimensional triangulation $\mathcal{T}$ can be partitioned among a user-defined group of processes. Besides locally owned cells $\mathcal{L}_i$, each process needs a halo of ghost cells.

2. 1D scalar (discontinuous) Lagrange shape functions $\mathcal{P}_k^1$ of polynomial degree k, based on Gauss–Lobatto support points, and 1D quadrature rules with n_q points are accessible. Dimension-independent interpolation kernels based on sum factorization are available.

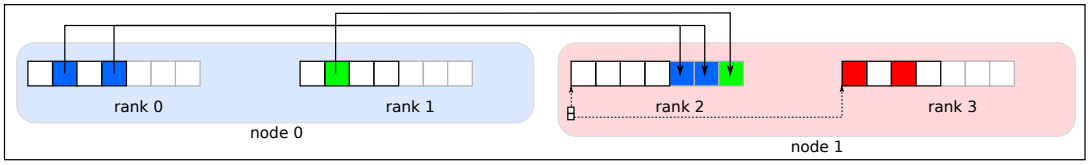
3. The library has access to the mapping data, like $\mathcal{J}^{-T}$ and $|\mathcal{J}| \times w$ both on cells and faces as well as $\vec{n}$ on faces. It is not relevant whether these quantities are pre-computed or recomputed on the fly.

4. It is beneficial if the library has the option to work on a batch of cells where the size of the batch can be set arbitrarily. This implies that the needed data is already provided in a vectorized fashion, which allows to skip an additional reshuffling step.

[1]For this purpose, `deal.II` provides the class struct `dealii::VectorizedArray<Number, v_len>`, where `Number` denotes the underlying primitive type and `v_len` the number of lanes. The information is automatically translated to the right instruction-set extension [29].



(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 7a, 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 7: 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.

Note: The library `hyper.deal` uses data structures and functions of `deal.II` directly, making it impossible to switch the backend library at this point. By introducing an intermediate layer, this problem might be circumvented.

4 Parallelization by shared-memory MPI

The parallelization of the library `hyper.deal` is purely MPI-based. 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 with one rank used per core is that many data structures 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 [29]. Depending on the MPI implementation, these operations will be replaced by efficient alternatives, avoiding additional copying and unexpected message buffers 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, with the downside of manually annotating and parallelizing all relevant loops. We propose a different approach to exploit shared memory. It is based on the observation that the major time and memory benefit of using shared memory in a purely MPI-parallelized FEM application comes 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 [30, 31]. 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 with an otherwise unchanged vector interface.

4.1 Shared-memory vector

The `MPI` function `MPI_WIN_ALLOCATE_SHARED()` allocates contiguous and non-contiguous memory that is shared among all processes on the same compute node. To query the beginning of the local array of each process, `MPI` provides the function `MPI_WIN_SHARED_QUERY()`.

These functions form the basis of the new shared-memory modus of the vector class `dealii::LinearAlgebra::distributed::Vector` and provide memory for the locally-owned unknowns and for unknowns of ghost faces that are not owned by any process on the same compute node. Furthermore, pointers to the arrays of the other processes are included so that with a basic preprocessing step the address of each cell residing on the same compute node can be determined. Appendix A provides further implementation details of the allocation/deallocation process of the shared memory in the vector class.

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.

An interpretation layer also provides access to the values of the degrees of freedom of the local and the ghost cells: It returns pointers either to buffers or to the shared memory, depending on the cell type (shared or remote). 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 at a single place and only a few functions querying values from the vector (e.g., `FEEvaluation::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 7a), 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 7b) 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.

4.2 Overlapping communication and computation

Finally, we discuss an appropriate integration of the new shared-memory vector into ECL-based operator-evaluation algorithms (see Algorithm 3). For this purpose, we categorize cells owned by a process into the following subpartitions $\mathcal{S}_i$:

1. cells with only locally-owned neighbors,
2. cells with locally-owned neighbors or neighbors shared within the same shared-memory domain,
3. remaining cells, i.e., cells with at least one remote neighbor.

Since $\mathcal{S}_1$ does not depend on any data possessed by other processes, the communication for updating the ghost values and the computation can be overlapped. We furthermore split up the communication into two steps, one for shared data and one for remote data exchange.

The shared data exchange step consists of the notification of relevant processes on the same shared-memory domain (start), on the one hand, and of the waiting until the needed data of relevant processes on the same shared-memory domain is ready (finish), on the other hand. In the case of the `buffering` mode, the latter substep also comprises the copying of the data into buffers.

By merging $\mathcal{S}_2$ and $\mathcal{S}_3$, one recovers the standard overlapping communication and computation that does not exploit shared memory. By merging all three subpartitions, the overlapping can be completely turned off.

One could skip $\mathcal{S}_3$ and instead loop over all faces with remote neighbors in a second loop. However, this would involve a second write access to the solution vector during the face integrals what we intentionally try to avoid.

5 Performance analysis for 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

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

`hyper.deal` for high orders and high dimensions. Both node-level performance and parallel scalability are shown, including 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 point and to prevent algorithms explicitly designed for affine meshes, we deform the Cartesian meshes slightly. The velocity $\vec{a}$ in (1) is set constant and uniform over the whole domain.

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 [41, 38]. The following metrics are used to quantify the quality of the implementations:

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

$$\text{throughput} = \frac{\text{processed DoFs}}{\text{time}} \overset{\text{Eqn. (12)}}{=} \frac{|\mathcal{C}|\cdot(k+1)^d}{\text{time}} \tag{15}$$

  (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 (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, which, for a fixed discretization, corresponds to increasing the throughput. 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.

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

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 2.5 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)	2.7 GHz (2.7 GHz)
SIMD width	512 bit
arithmetic peak (`dgemm` performance)	4147 GFLOP/s (3318 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`

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) 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. The parallel network is organized into islands of 792 compute nodes each. The maximum network bandwidth per node within an island is 100GBit/s=12.5GB/s[2] 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 into blocks of the size of $48{=}8{\times}6$. All processes in these blocks share their values via the shared-memory vector. The cells in low-dimensional triangulations are enumerated along a Morton curve, which is equally distributed among the processes. We use the highest ISA extension AVX-512 so that 8 cells are processed at once. The Jacobian matrices and their determinants are precomputed for $\vec{x}$- and $\vec{v}$-space and combined on the fly. The quadrature is based on the Gauss-Legendre formula with $n_q = k + 1$. The skew factor β is set to 0.5 such that the gradients of the solution have to be computed at the cell quadrature points and the values have to be tested by the gradient of the test functions. In the following, we refer to this configuration as "default configuration".

5.2 Cell-local computation

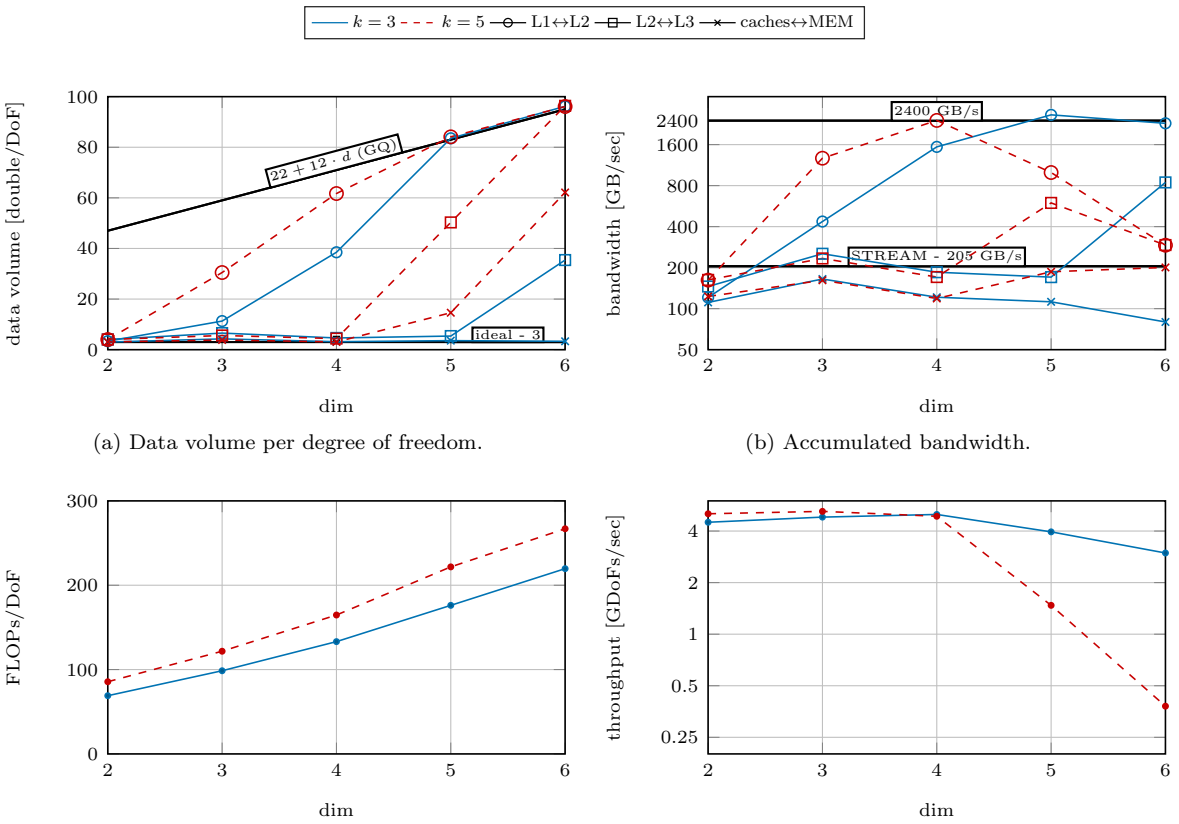
This subsection analyzes the cell-local computation in the element-centric evaluation of the advection operator (see Algorithm 3) as a means to assess the caching efficiency of temporary arrays in sum factorization with respect to the increasing number of sweeps and working-set sizes.

5.2.1 Cell integrals

We first consider all steps in Algorithm 3 related to the cell integrals (lines 2-9, 17-21), 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 d data sweeps for reading and d for writing), the gradients at the quadrature points are computed ($2d$), the gradients obtained are multiplied with the velocities (d), the values obtained are multiplied with the velocities at the quadrature points ($2d$) and tested by the value and the gradient of the collocation functions ($3d$), the inverse mass matrix is applied ($2d$), and finally the results are written back to the global vector. A total of $12d$ 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.

[2] `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.

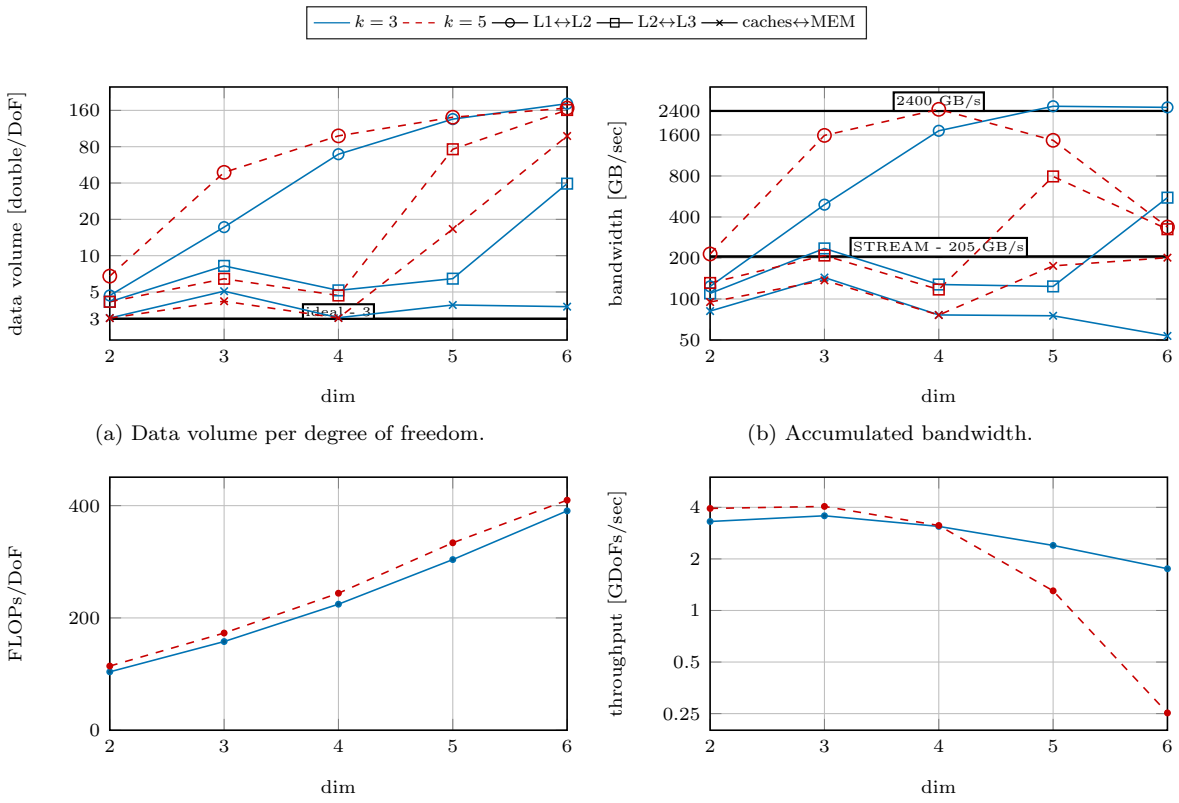
Figure 8: Node-level analysis of the cell integrals of the advection operator in terms of data transfer, bandwith, arithmetic operations, and throughput in two to six dimensions.

The theoretical considerations made above are supported by the measurement results in Figure 8, 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 8a, 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 the L2 cache again during every sweep, resulting in a data volume traffic between the L1 and L2 caches that linearly increases with the number of sweeps. The constant offset of 22 double/DoF is mainly related to the access to the global solution vector and to the mapping data. However, data has to be loaded from the L2 cache also 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 the main memory have to be accessed during the sweeps. While this operation is negligible for $k = 3$ (see Figure 8d), it is performance-limiting in the case of $k = 5$: For $k = 5$ / $d = 5$, the bandwidth to the L1 cache is limited by the access to the L2 cache (see Figure 8b); 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. This comes along with a significant performance drop.

Figure 8c also shows the number of floating-point operations performed per degree of freedom, which increases linearly with the dimension d—with higher polynomial degrees requiring more work. It is clear that also the arithmetic intensity will increase linearly as long as the data stays in the cache (see also Subsection 5.3).



(a) Data volume per degree of freedom.

(b) Accumulated bandwidth.

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

(d) Throughput.

Figure 9: Node-level analysis of the evaluation of the cell and face intergrals of the advection operator, ignoring loads from neighboring cells.

5.2.2 Local cell and face integrals

In this subsection, we consider all computation steps in Algorithm 3, but ignore the data access to neighboring cells (line 12). 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 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 during the flux computation.

Figure 9 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 8, the following observations can be made: As expected, the data volume transferred between the cache levels (Figure 8a and 9a) and the number of floating-point operations approximately double (Figure 8c and 9c). 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 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 9d). In the case of $k = 3$, we can also observe a drop of performance in high dimensions. Given that the memory transfer between the L1 and L2 caches reaches 2,400 GB/s or around 22 bytes/cycle, we suspect that the data transfer between the L1 and L2 caches is the main limit in this case. The memory transfer between the L2 and L3 caches is about 550GB/s. This value is significantly less than that is observed for the cell-integral-only run, resulting in the drop of the overall performance by 40% for high dimensions.

5.3 Full advection operator

This subsection considers the application of the full advection operator, as shown in Algorithm 3, including the access to neighboring cells during the computation of the numerical flux. Figure 10 presents the results of parameter studies of the dimension $2 \leq d \leq 6$ for different polynomial degrees $2 \leq k \leq 5$.

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 size is a suitable indicator of the overall performance of the operator evaluation. 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 10: 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 so that 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.

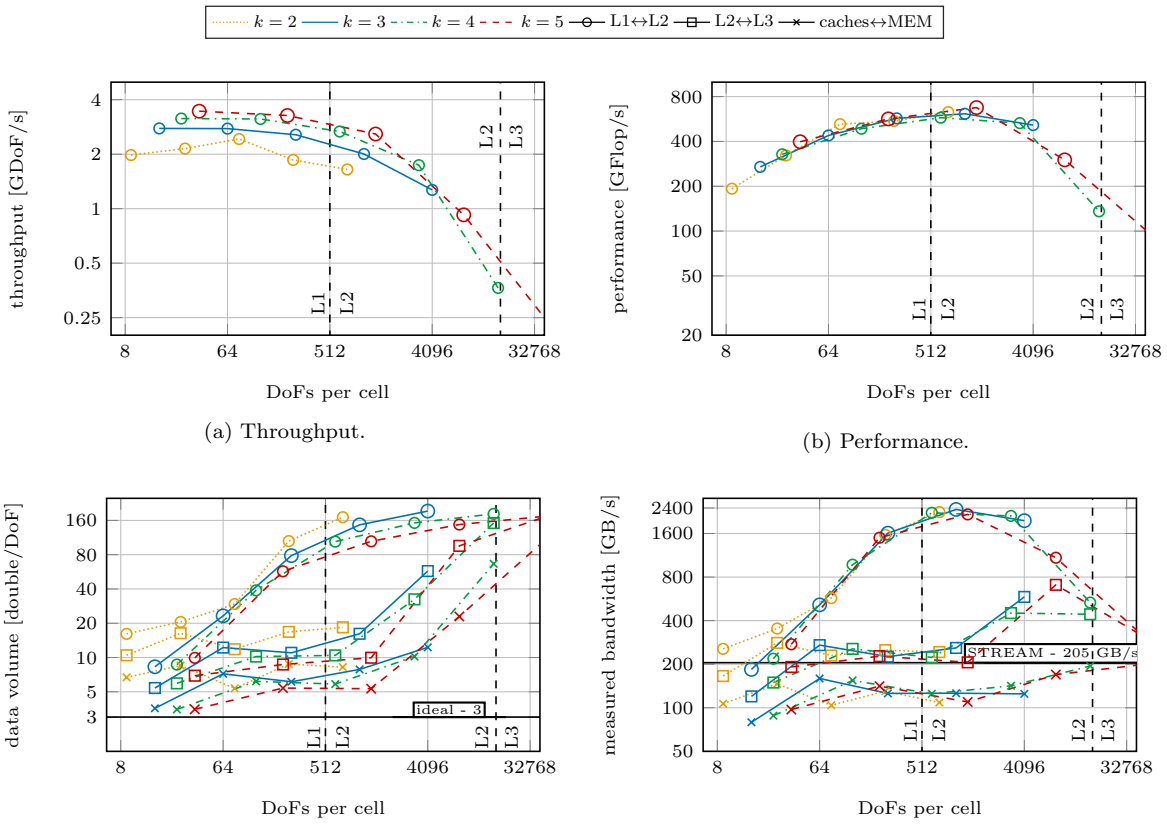
Comparing these findings with the results presented in Subsection 5.2.1 and 5.2.2, an averaged performance drop of 30% and 20%, respectively, can be observed (see Table 6 and 7). Looking at the results for high dimensions, it becomes clear that processing the faces is more expensive than loading the actual values from the neighbors.

Figure 11 shows a roofline model [45] for $k = 3$ and $k = 5$. In this model, the measured performance is plotted over the measured arithmetic intensity

$$\left(\text{measured arithmetic intensity}\right)_i = \frac{\text{measured performance}}{\left(\text{measured bandwidth}\right)_i}, \tag{16}$$

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, a high performance can only be reached if the caches (L1 and L2) are utilized well. Once the working sets get too large, the 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.

We point out that by selecting the skew-symmetric parameter $\beta = 0.5$, we need to compute both the values and the gradients at the quadrature points as well as to test by the value and the gradient



(a) Throughput.

(b) Performance.

(c) Data volume per degree of freedom.

(d) Accumulated bandwidth.

Figure 10: Node-level analysis of the evaluation of the full advection operator in terms of data transfer, bandwidth, arithmetic operations, and throughput in two to six dimensions. Dashed vertical lines show the limits of the L1 and L2 caches for a hypothetical fully associative cache with optimal replacement policy.

Table 6: Relative performance due to flux computation (ratio Fig. 9-8).

k/d	2	3	4	5	6
3	73%	74%	62%	61%	59%
5	78%	78%	64%	88%	66%

Table 7: Relative performance due to access to the values of neighboring cells (ratio Fig. 10-9).

k/d	2	3	4	5	6
3	84%	78%	83%	84%	72%
5	88%	81%	82%	71%	88%

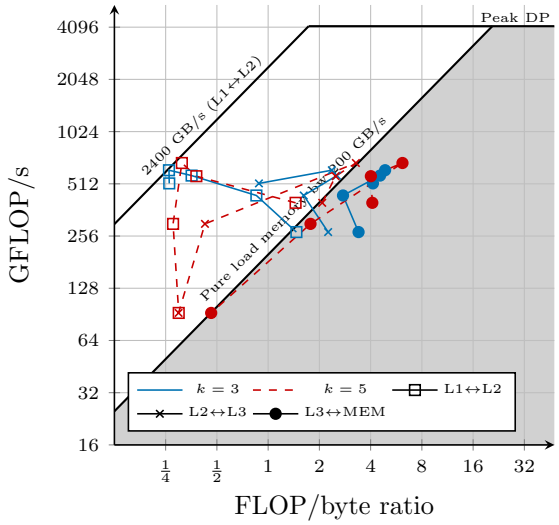


Figure 11: Roofline model of the application of the full advection operator for $k = 3/5$.

Table 8: Evaluation of the shared-memory vector for different configurations on a single compute node with 48 processes. In the simulation, there were accumulated 2.1G locally-owned degrees of freedom and 1.6G ghost degrees of freedom for a 6D problem with $k = 3$. Communication and computation has been overlapped in all cases.

	`MPI-2.0`	`MPI-3.0` w. buffering	`MPI-3.0` w.o. buffering
throughput [GDoFs/sec]	1.02	1.23	1.33

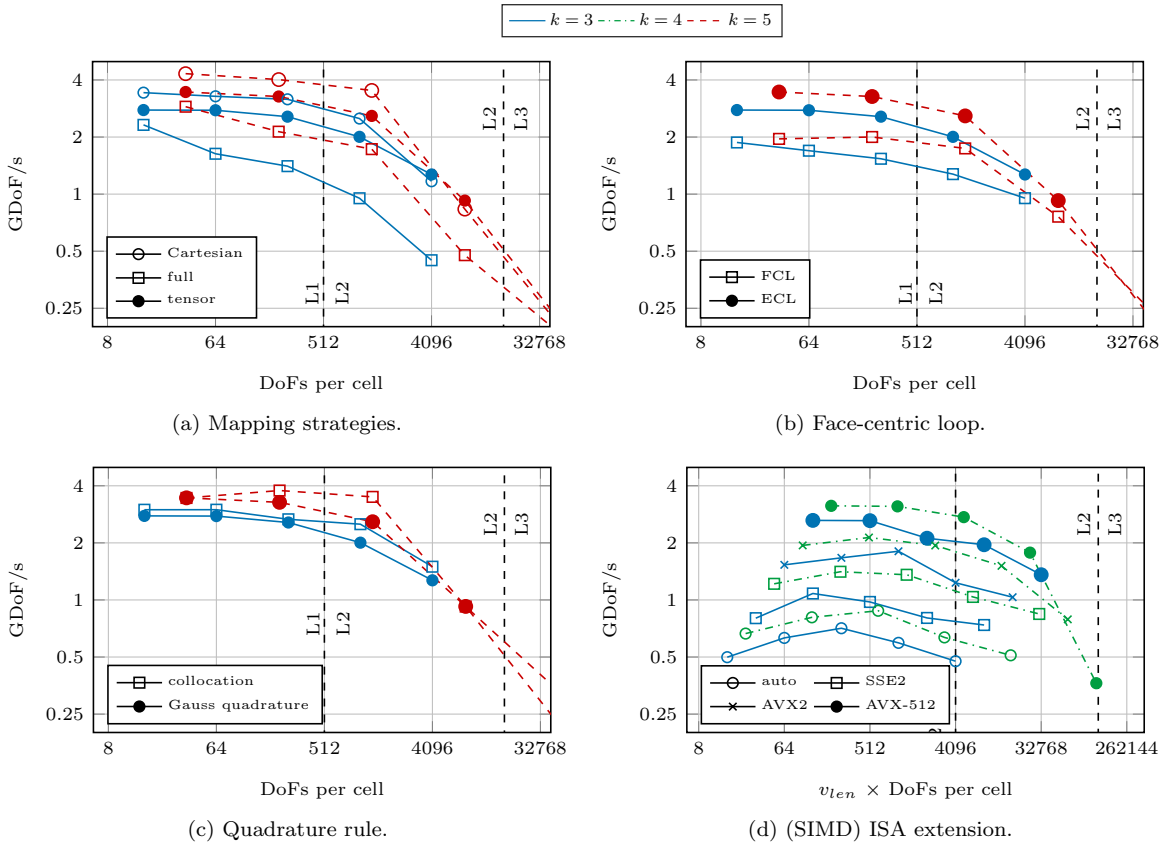
of the test functions. Instead, by using a conservative formulation ($\beta = 0.0$) or a convective formulation ($\beta = 1.0$), it is possible to skip either the computation of the gradients or testing by the gradients, which leads to fewer sum-factorization sweeps and potentially decreases the cache misses. Experiments indeed confirm this and show a speed-up of approx. 14% for both $\beta = 0.0$ and $\beta = 1.0$.

We conclude this subsection by discussing the performance benefit of using the shared-memory vector introduced in Section 4. We consider three different modi of the vector: 1) exploitation of no shared-memory features (by setting the shared-memory group size to 1 so that the implementation falls back to pure `MPI-2.0` features), 2) `buffered mode`, and 3) `non-buffered mode`. The results are summarized in Table 8. Exploiting the shared memory explicitly is beneficial in cases 2 and 3. While in the case of buffering we observe a speed-up of 20%, we even see a speed-up of 30% when not buffering.

5.4 Alternative implementations

In the following subsection, we compare the performance of the default configuration of the library `hyper.deal` (tensor product of mappings, ECL, Gauss quadrature, vectorization over elements with the 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. The upper limit of the performance of the tensor-product approach is given by the consideration of the tensor product of two Cartesian grids, which leads to the same constant 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 12a 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. 13%. 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



(a) Mapping strategies.

(b) Face-centric loop.

(c) Quadrature rule.

(d) (SIMD) ISA extension.

Figure 12: Node-level analysis of the application of the full advection operator: 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 $k = 4$.

factors is essentially for free, compared to storing the full Jacobian matrices.

Figure 12b compares ECL with FCL and shows the clear advantage of the former. We have neither implemented any advanced blocking schemes for ECL or FCL nor are we processing cell and face integrals in alternating order, as it is done in `deal.II` [30], to potentially increase cache efficiency. The fact that ECL still shows a better performance demonstrates the natural cache-friendly property of ECL. The benefit of ECL decreases for high dimensions and high polynomial degrees due to the increased number of sweeps, which is related to the repeated evaluation of the flux terms. Nevertheless, we propose to use ECL for high-dimensional 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 12c shows a performance drop of 11% on average due to these basis changes as long as the data that should be interpolated remains in the 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 12d. 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 and AVX2 show the best performance.

In this subsection, we have demonstrated that the chosen default configuration of the library `hyper.deal`

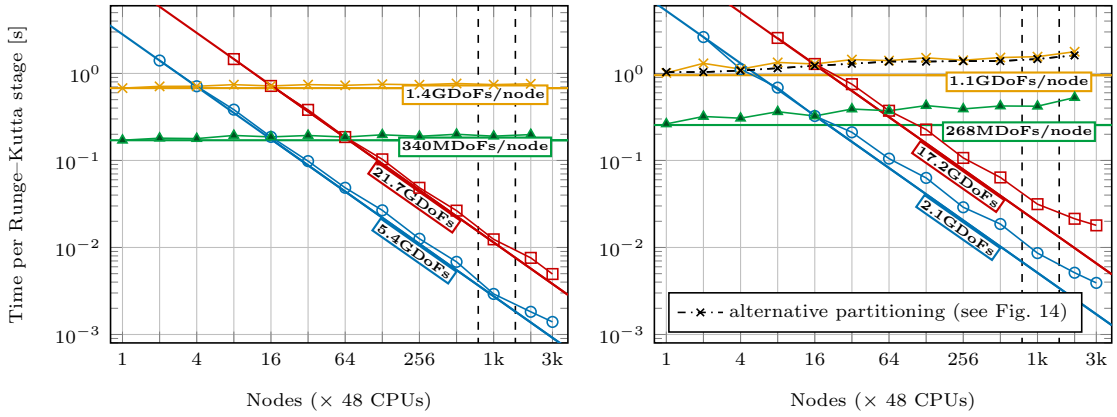
Table 9: Strong and weak scaling configurations: left) $d = 4$ / $k = 5$; right) $d = 6$ / $k = 3$

DoFs	configuration
5.4GDoFs	$384^2 \cdot 192^2$
21.7GDoFs	384^4
268MDoFs/node	$192^2 \cdot 96^2$
1.1GDoFs/node	192^4

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

Table 10: 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



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

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

Figure 13: Strong and weak scaling of one Runge–Kutta step with the advection operator as right-hand side. Each line corresponds to a weak scaling experiment with the given number of DoFs per node or to a strong scaling experiment with the given number of DoFs, as specified in Table 9.

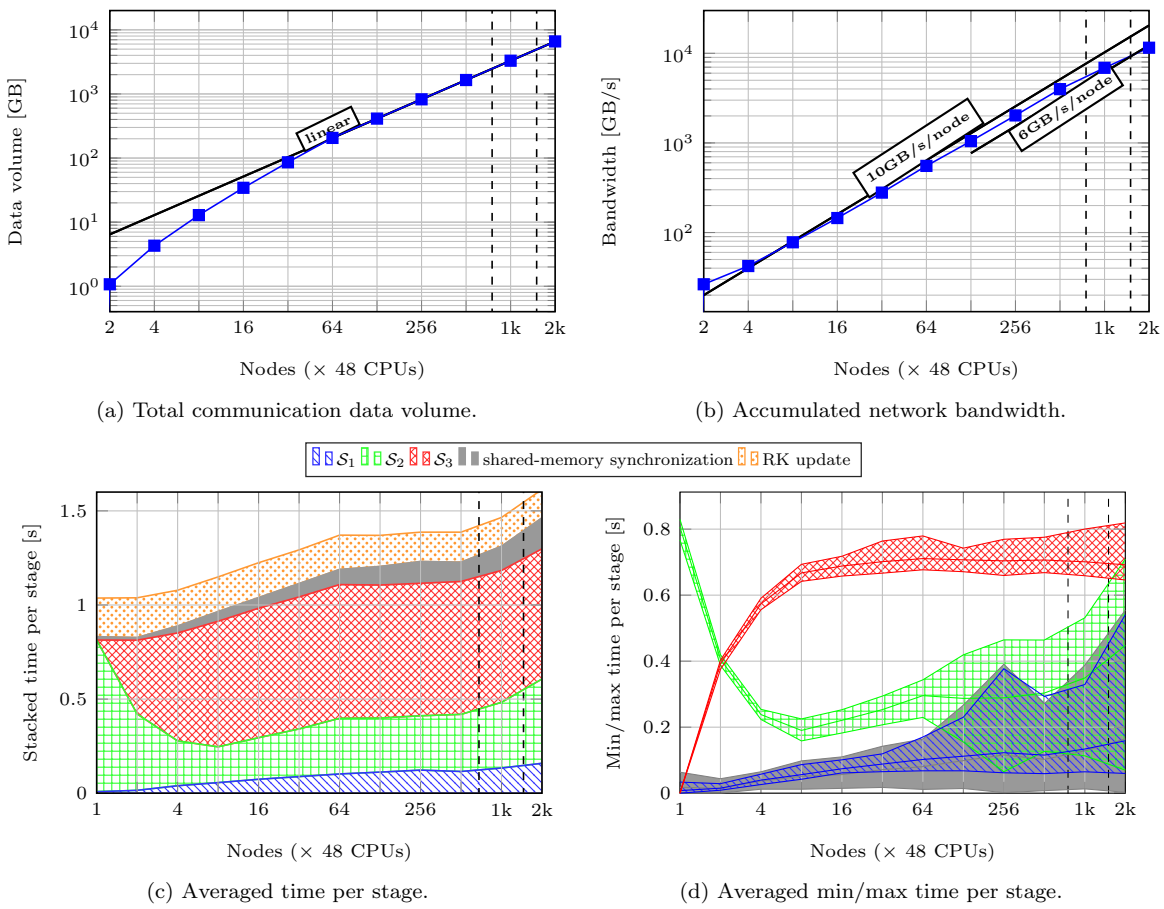
has a competitive throughput compared to less memory-expensive and computationally demanding algorithms, which are numerically inferior.

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 [25]. From these three vectors, only one (auxiliary) vector has ghost values.

Figure 13 shows strong and weak scaling results of runs on SuperMUC-NG with up to 3,072 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 9). Table 10 shows the considered process decomposition $p = p_{\vec{x}} \cdot p_{\vec{v}}$.

For the "$d = 4$ / $k = 5$" configuration, we observe excellent weak-scaling behavior with parallel efficiencies of 89% and 86% for up to 2,048 nodes on the large and the small setup, respectively. We get more than 75/80% efficiency for strong scaling up to the increase in the number of nodes by a factor of 256. For the "$d = 6$ / $k = 3$" configuration, we see parallel efficiencies of 49/57% 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 Figure 15, the ghost data to remote nodes amounts to 29% of the solution vector in 6D and only 5% in 4D.



(a) Total communication data volume.

(b) Accumulated network bandwidth.

(c) Averaged time per stage.

(d) Averaged min/max time per stage.

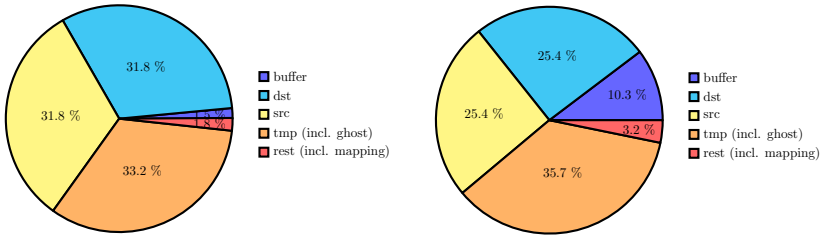
Figure 14: Details on the weak scaling of a single Runge–Kutta stage of the solution of the advection equation.

Finally, we analyze the drop in parallel efficiency of the weak-scaling runs of the 6D 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 and a total number of degrees of freedom of 1.1GDoFs). 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 direction, we also double the number of processes in that direction, keeping the number of processes in the other direction constant. 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 communication amount of the considered setup increases linearly with the number of processes, as presented in Figure 14a. Measurements in Figure 14b show that the network can handle this increase: The data can be sent with a constant network bandwidth of 10GB/s per node, which is close to the theoretical 100Gbit/s as long as the job stays on an island due to the fat-tree network topology. Once the job stretches over multiple islands due to the pruned-tree network architecture, we observe a bandwidth of 6GB/s, which is related to the fact that only a small ratio of the messages crosses island boundaries.

Figure 14c and 14d show the time spent in different sections (in the following referred to as steps) of the advection operator. We consider the following five steps:

1. Start the shared-memory communication and the remote communication by calling `MPI_Irecv` as



(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 15: Approximate memory-consumption distribution of high-dimensional advection application for 48×1024 cores.

well as pack and send (via `MPI_Isend`) messages to each neighboring process residing on remote compute nodes. Furthermore, process $\mathcal{S}_1$ in the overlap of the communication and the computation strategy.

2. Finish shared-memory communication and process $\mathcal{S}_2$.
3. Finish the remote ghost-value update by waiting (with `MPI_Waitall`) until all messages have been sent and received as well as process $\mathcal{S}_3$.
4. Synchronize the shared-memory processes, which is needed to prevent race conditions due to the reusage of the source vector during the subsequent Runge–Kutta update steps.
5. 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.

On a single node, most of the time is spent for $\mathcal{S}_2$, since all data is available in the same shared-memory domain. As the number of nodes increases, the time spent for $\mathcal{S}_3$ needing more data from remote processes increases. The time spent for $\mathcal{S}_1$ is comparatively small, which can be attributed to the fact that in 6D nearly all cells have neighbors owned by other processes.

The overall runtime increases with increasing number of nodes. In particular, the time increases for low node numbers as new periodic neighbors are added. For high node numbers, slower communication to other islands is required.

The minimum and the maximum time spent for each step differ significantly, which can be attributed to the non-trivial communication pattern and the varying size of $\mathcal{S}_2$ and $\mathcal{S}_3$. Overall, however, this imbalance caused by the MPI communication is not performance-hindering, as has been 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.

5.6 Memory consumption

Figure 15b shows the approximated memory consumption for a large-scale simulation from Subsection 5.5 (1024 nodes, $d = 6$ / $k = 3$, $1.1 \cdot 10^{12}$ DoFs). A total of 34.6PB main memory from available 98PB is used. The largest amount of memory is occupied by the three solution vectors (each. 25.4%). The two buffers for MPI communication occupy each 10.3%. 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 (3.2%) 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 15a.

5.7 Largest simulations

In order to demonstrate the large-scale suitability of our code, we have performed simulations on 3,072 compute nodes, the largest possible configuration available on the SuperMUC-NG system. We have

conducted a 6D simulation with a curved mesh, which contains $128^6 = 4.4 \cdot 10^{12}$ degrees of freedom ($\approx$ 1.4GDoFs/node) with a partitioning as specified in Table 10. In this case, we reached a total throughput of $1.1 \cdot 10^{12}$ DoFs/s. This means that each Runge-Kutta stage is processed in 3.7s and a complete time step consisting of 5 Runge-Kutta stages takes 19.3s. As a reference, the largest 4D simulation run processed $1,536^4 = 5.56 \cdot 10^{12}$ degrees of freedom with $1.9 \cdot 10^{12}$ DoFs/s.

6 Application: Vlasov–Poisson

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

$$\frac{\partial}{\partial t} f(t, \vec{x}, \vec{v}) + \vec{v} \cdot \nabla_{\vec{x}} f(t, \vec{x}, \vec{v}) - \vec{E}(t, \vec{x}) \cdot \nabla_{\vec{v}} f(t, \vec{x}, \vec{v}) = 0, \tag{17}$$

is considered, where $f(t, \vec{x}, \vec{v})$ denotes the probability density of a particle as a function of the phase space and $\vec{E}$ the electric field. If we define the gradient operator as $\nabla^\top := (\nabla_{\vec{x}}^\top,\ \nabla_{\vec{v}}^\top)$ and $\vec{x} := (\vec{x}, \vec{v})^T$, (17) can be rewritten as

$$\frac{\partial\, f(t, \vec{x}, \vec{v})}{\partial\, t} + \begin{pmatrix} \vec{v} \\ -\vec{E}(t, \vec{x}) \end{pmatrix} \cdot \nabla f(t, \vec{x}, \vec{v}) = 0. \tag{18a}$$

The electric field is obtained from the Poisson problem

$$\rho(t, \vec{x}) = 1 - \int f(t, \vec{x}, \vec{v})\, \mathrm{d}v, \qquad -\nabla_{\vec{x}}^2 \phi(t, x) = \rho(t, \vec{x}), \qquad \vec{E}(t, \vec{x}) = -\nabla_{\vec{x}} \phi(t, \vec{x}). \tag{18b}$$

For the time propagation, we use a low-storage Runge–Kutta method of order 4 with 5 stages [25]. 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.

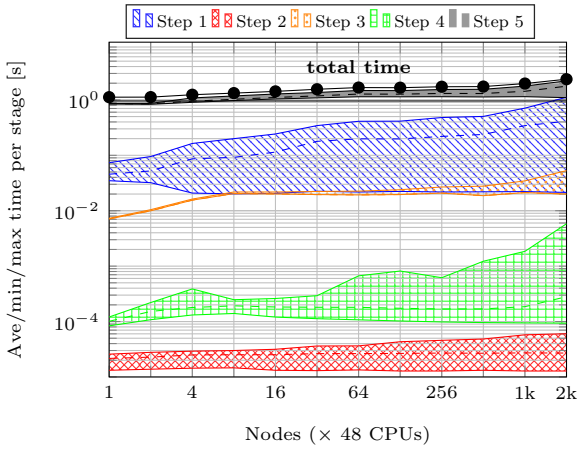
Step (5) is a $d_{\vec{x}} + d_{\vec{v}}$-dimensional problem, and Steps (2)–(4) are $d_{\vec{x}}$-dimensional problems. Step (1) reduces information from the phase space to the configuration space.

6.1 Implementation details

The advection step (Step (5)) relies on the advection 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 evaluation of $\vec{v}$ at a quadrature point can be queried from a low-dimensional FEM library in $\vec{v}$-space. 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 variable velocity field.

For the solution of the Poisson problem

$$(\nabla_{\vec{x}} \psi, \nabla_{\vec{x}} \phi)_{\vec{x}} = (\psi,\ \rho)_{\vec{x}}, \tag{19}$$



#nodes	total time [s]
1	1.12
2	1.12
4	1.24
8	1.33
16	1.42
32	1.56
64	1.67
128	1.66
256	1.73
512	1.74
1024	1.97
2048	2.36

Figure 16: Weak scaling of a single Runge–Kutta stage of the solution of the Vlasov–Poisson equations on a hypercube. The average (dashed), averaged minimum and averaged maximum runtime of each step are indicated. Additionally, the total runtime is shown with bullets and in the table.

with ψ denoting the test functions, we utilize a matrix-free geometric multigrid solver from `deal.II`, which uses a Chebyshev smoother with polynomial degree 5 [1] and has settings similar to [33, 14]. The Poisson problem is solved up to a relative tolerance of 10^{-4} 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 for 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 [27].

6.2 Weak scaling

We perform a weak-scaling experiment for the 6D Vlasov–Poisson system, 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 as in Subsection 5.5.

Figure 16 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 the 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 one process and is thus the same in every experiment. The total amount of data sent/received is therefore proportional to the total number of processes, while the number of reduction steps is only $\mathcal{O}(\log(p_{\vec{v}}))$. 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. We also note that the process grid is designed to optimize the communication of the advection step so that communication patterns of other steps might be suboptimal due to shared-memory blocking, see Figure 4.

Steps 2 to 4 are three-dimensional problems, which 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.

6.3 Tensor product of a torus and a sphere

We conclude this section by presenting timings for simulations conducted on the tensor product of a torus and a sphere, a prototype of a complex geometry needed for the simulation of a Tokamak.

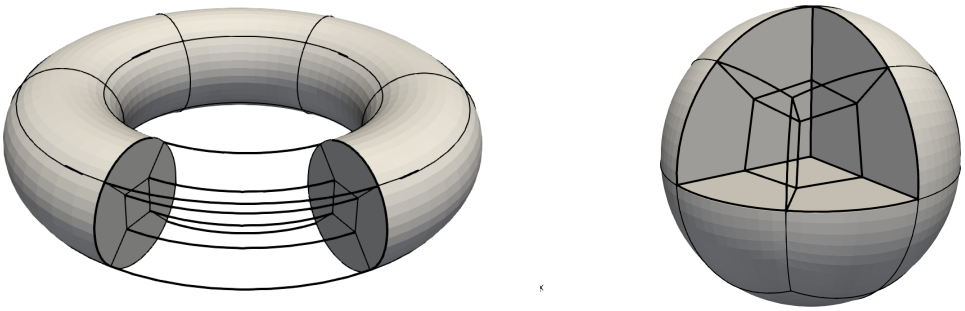


Figure 17: Coarse grid of the "torus $\otimes$ sphere"-simulation. The torus consists of 30 and the sphere of 32 cells. Curved surface descriptions are used to derive a high-order mapping of the curved surfaces.

Table 11: Weak scaling of the problem size of 1.00e+09 DoFs per node.

#nodes	**#degrees of freedom**			**total time per stage [s]**
	x	**v**	**x $\otimes$ v**	
2	122.9k	16.4k	2.0G	1.23
16	122.9k	131.1k	16.1G	1.53
128	983.0k	131.1k	128.8G	1.67
1024	983.0k	1.0M	1.0T	2.03

The torus has a major radius of 6.2 and a minor radius of 2.0; the sphere has a radius of 5.0. The coarse grid of both 3D geometries is shown in Figure 17. The curved surfaces are described by analytical manifolds according to the model described in [19] and extended into the interior of the computational domain with transfinite interpolation [16]. The analytical geometry representation is queried for the position of mesh vertices during mesh refinement and for auxiliary points of polynomial mappings to enable a high-order curvilinear geometry description. The final mesh is obtained by uniform global refinement of each 3D geometry.

We apply a homogeneous Dirichlet boundary condition on all surfaces in phase space and a homogeneous Neumann boundary condition in the case of the Poisson problem.

The timings of a weak scaling experiment are presented in Table 11. The fact that the timings are comparable with those of a high-dimensional hypercube with periodic boundaries, as presented in Figure 16, verifies that the proposed algorithms are indeed generic and efficient for complex unstructured meshes and for more complex boundary conditions.

7 Summary and outlook

We have presented the finite-element library `hyper.deal`, which 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. The proposed algorithms are aligned with the architecture of current pre-exascale machines with high FLOP-per-byte ratios and are expected to also run efficiently on projected exascale machines.

We have compared the node-level performance of the default configuration of `hyper.deal` with alternative algorithms, which are specialized for Cartesian and affine meshes or use collocation integration schemes. Even though the proposed algorithms are not primarily limited by the raw memory bandwidth, our studies reveal that loading less mapping data is most beneficial to improve the performance, and,

to a lesser extent, reducing the number of sum-factorization sweeps is beneficial, too. To utilize these advantages on a broader set of configurations, we plan to look into computing the low-dimensional mapping information on the fly and study 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.

All simulations have been run on uniformly refined meshes. In future work, we will target the extension of the presented algorithms to adaptively refined meshes. Generic low-dimensional finite-element implementations perform interpolations across hanging nodes, involving a tensor product of interpolation matrices on half the reference interval for a 2:1 mesh ratio plus some changes to the neighbor data access. One could even go beyond the use of a single mesh by combining the meshes for different refinement level pairs ("slices") of the phase space, requiring that each "slice" is treated on its own with the presented tensor-product approach and "slices" are glued together by special-purpose coupling operators. While the implementation is not trivial, we believe that it only involves changes in the vector access and possibly in the MPI partitioning of the low-dimensional meshes, with the general interface of the library remaining unmodified.

The high degree of optimization of our implementation together with the features offered by `deal.II` regarding meshes of complex geometry and refinement paves the way to exploring the physics of fusion plasmas with this novel library also on future exascale machines.

A APPENDIX

The following code snippets give implementation details on the new shared-memory modus of the vector class `dealii::LinearAlgebra::distributed::Vector`, 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 the simplified allocation routines of the vector class for the value type `T` and the size `_local_size+_ghost_size`:

```
MPI_Win          win;         // window
T *              data_this;   // pointer to locally-owned data
std::vector<T *> data_others; // pointers to shared data

// 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_this, &win);

// get pointers to the shared data owned by the processes is same sm domain
data_others.resize(size_sm);
for (int i = 0, int disp_unit, MPI_Aint ssize; i < size_sm; i++)
  MPI_Win_shared_query(win, i, &ssize, &disp_unit, &data_others[i]);

Assert(data_this==data_others[rank_sm]);
```

Once the data is not needed anymore, the window has to be freed, which also frees the locally-owned data:

```
MPI_Win_free(&win)
```

Acknowledgments

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.

References

[1] M. Adams, M. Brezina, J. Hu, and R. Tuminaro. Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *Journal of Computational Physics*, 188:593–610, 2003.

[2] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100):9–23, 2015.

[3] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini. MFEM: A modular finite element methods library. *Comput. Math. Appl.*, 81:42–74, 2021.

[4] D. Arndt, W. Bangerth, B. Blais, T. C. Clevenger, M. Fehling, A. V. Grayver, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, R. Rastak, I. Thomas, B. Turcksin, Z. Wang, and D. Wells. The `deal.II` library, version 9.2. *Journal of Numerical Mathematics*, 28(3):131–146, 2020.

[5] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells. The deal.ii finite element library: Design, features, and insights. *Comput. Math. Appl.*, 81:407–422, 2021.

[6] M. Bachmayr, R. Schneider, and A. Uschmajew. Tensor networks and hierarchical tensors for the solution of high-dimensional partial differential equations. *Found. Comput. Math.*, 16(6):1423–1472, 2016.

[7] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Transactions on Mathematical Software*, 38(2), 2011.

[8] P. Bastian, C. Engwer, J. Fahlke, M. Geveler, D. Göddeke, O. Iliev, O. Ippisch, R. Milk, J. Mohring, S. Müthing, M. Ohlberger, D. Ribbrock, and S. Turek. Hardware-based efficiency advances in the EXA-DUNE project. In H.-J. Bungartz, P. Neumann, and W. E. Nagel, editors, *Software for Exascale Computing – SPPEXA 2013–2015*, pages 3–23, Cham, 2016. Springer International Publishing.

[9] G.-T. Bercea, A. T. T. McRae, D. A. Ham, L. Mitchell, F. Rathgeber, L. Nardi, F. Luporini, and P. H. J. Kelly. A structure-exploiting numbering algorithm for finite elements on extruded meshes, and its performance evaluation in firedrake. *Geosci. Model Dev.*, 9(10):3803–3815, 2016.

[10] C. Burstedde, L. C. Wilcox, and O. Ghattas. p4est : Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM J. Sci. Comput.*, 33(3):1103–1133, 2011.

[11] D. Davydov, J.-P. Pelteret, D. Arndt, M. Kronbichler, and P. Steinmann. A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid. *International Journal for Numerical Methods in Engineering*, 121(13):2874–2895, 2020.

[12] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive scientific computing: Abstraction principles and the dune-fem module. *Computing*, 90:165–196, 2010.

[13] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*, volume 9. Cambridge University Press, Cambridge, 2002.

[14] N. Fehn, P. Munch, W. A. Wall, and M. Kronbichler. Hybrid multigrid methods for high-order discontinuous galerkin discretizations. *Journal of Computational Physics*, 415:109538, 2020.

[15] F. Filbet and E. Sonnendrücker. Comparison of Eulerian Vlasov solvers. *Comput. Phys. Communic.*, 150(3):247–266, 2003.

[16] W. J. Gordon and L. C. Thiel. Transfinite mappings and their application to grid generation. *Applied Mathematics and Computation*, 10:171–233, 1982.

[17] W. Guo and Y. Cheng. A sparse grid discontinuous galerkin method for high-dimensional transport equations and its application to kinetic simulations. *SIAM J. Sci. Comput.*, 38(6):A3381–A3409, 2016.

[18] A. Hakim, G. Hammett, E. L. Shi, and N. Mandell. Discontinuous galerkin schemes for a class of hamiltonian evolution equations with applications to plasma fluid and kinetic problems. *arXiv*, 1908.01814, 2019.

[19] L. Heltai, W. Bangerth, M. Kronbichler, and A. Mola. Using exact geometry information in finite element computations. *arXiv preprint arXiv:1910.09824*, 2019.

[20] M. A. Heroux, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, K. S. Stanley, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, and R. P. Pawlowski. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, 2005.

[21] J. S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Springer, New York, 2008.

[22] J. Juno, A. Hakim, J. TenBarge, E. L. Shi, and W. Dorland. Discontinuous galerkin algorithms for fully kinetic plasmas. *Journal of Computational Physics*, 353:110–147, 2018.

[23] G. Karypis and V. Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, September 1998.

[24] D. Kempf, R. Heß, S. Müthing, and P. Bastian. Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures. *ACM Transactions on Mathematical Software*, 47(1), 2021.

[25] C. A. Kennedy, M. H. Carpenter, and R. M. Lewis. Low-storage, explicit runge–kutta schemes for the compressible navier–stokes equations. *Applied Numerical Mathematics*, 35(3):177–219, 2000.

[26] D. A. Kopriva and G. J. Gassner. An energy stable discontinuous galerkin spectral element discretization for variable coefficient advection problems. *SIAM J. Sci. Comput.*, 36(4):A2076–A2099, 2014.

[27] K. Kormann, K. Reuter, and M. Rampp. A massively parallel semi-lagrangian solver for the six-dimensional vlasov–poisson equation. *The International Journal of High Performance Computing Applications*, 33(5):924–947, 2019.

[28] B. Krank, N. Fehn, W. A. Wall, and M. Kronbichler. A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. *Journal of Computational Physics*, 348:634–659, 2017.

[29] M. Kronbichler and K. Kormann. A generic interface for parallel cell-based finite element operator application. *Computers & Fluids*, 63:135–147, 2012.

[30] M. Kronbichler and K. Kormann. Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM Transactions on Mathematical Software*, 45(3), 2019.

[31] M. Kronbichler, K. Kormann, N. Fehn, P. Munch, and J. Witte. A hermite-like basis for faster matrix-free evaluation of interior penalty discontinuous galerkin operators. *arXiv preprint arXiv:1907.08492*, 2019.

[32] M. Kronbichler, S. Schoeder, C. Müller, and W. A. Wall. Comparison of implicit and explicit hybridizable discontinuous galerkin methods for the acoustic wave equation. *International Journal for Numerical Methods in Engineering*, 106:712–739, 2016.

[33] M. Kronbichler and W. A. Wall. A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers. *SIAM J. Sci. Comput.*, 40(5):A3423–A3448, 2018.

[34] J. M. Melenk, K. Gerdes, and C. Schwab. Fully discrete hp-finite elements: fast quadrature. *Computer Methods in Applied Mechanics and Engineering*, 190(32):4339–4364, 2001.

[35] S. Müthing, M. Piatkowski, and P. Bastian. High-performance implementation of matrix-free high-order discontinuous galerkin methods. *arXiv preprint arXiv:1711.10885*, 2017.

[36] S. A. Orszag. Spectral methods for problems in complex geometries. *Journal of Computational Physics*, 37(1):70–92, 1980.

[37] B. Reps and T. Weinzierl. Complex additive geometric multilevel solvers for Helmholtz equations on spacetrees. *ACM Transactions on Mathematical Software*, 44(1):1–36, 2017.

[38] T. Roehl, J. Treibig, G. Hager, and G. Wellein. Overhead Analysis of Performance Counter Measurements. In *2014 43rd International Conference on Parallel Processing Workshops*, volume 2015-May, pages 176–185, Minneapolis, Minnesota, USA, 2014. IEEE.

[39] S. Schoeder, K. Kormann, W. A. Wall, and M. Kronbichler. Efficient explicit time stepping of high order discontinuous Galerkin schemes for waves. *SIAM J. Sci. Comput.*, 40(6):C803–C826, 2018.

[40] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. Kelly. A study of vectorization for matrix-free finite element methods. *Int. J. High Perf. Comput. Appl.*, 34(6):629–644, 2020.

[41] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In W.-C. Lee, editor, *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216, Piscataway, NJ, 2010. IEEE.

[42] T. Umeda, K. Fukazawa, Y. Nariyuki, and T. Ogino. A scalable full-electromagnetic vlasov solver for cross-scale coupling in space plasma. *IEEE T. Plasma Sci.*, 40(5):1421–1428, 2012.

[43] R. A. Van De Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency–Pract. Ex.*, 9(4):255–274, 1997.

[44] T. Weinzierl. The peano software—parallel, automaton-based, dynamically adaptive grid traversals. *ACM Transactions on Mathematical Software*, 45(2), 2019.

[45] S. Williams, A. Waterman, and D. Patterson. Roofline : An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65, 2009.