

ExaDG: High-Order Discontinuous Galerkin for the Exa-Scale



Daniel Arndt, Niklas Fehn, Guido Kanschat, Katharina Kormann, Martin Kronbichler, Peter Munch, Wolfgang A. Wall, and Julius Witte

Abstract This text presents contributions to efficient high-order finite element solvers in the context of the project ExaDG, part of the DFG priority program 1648 *Software for Exascale Computing* (SPPEXA). The main algorithmic components are the matrix-free evaluation of finite element and discontinuous Galerkin operators with sum factorization to reach a high node-level performance and parallel scalability, a massively parallel multigrid framework, and efficient multigrid smoothers. The algorithms have been applied in a computational fluid dynamics context. The software contributions of the project have led to a speedup by a factor 3 – 4 depending on the hardware. Our implementations are available via the deal.II finite element library.

D. Arndt
Oak Ridge National Laboratory, Oak Ridge, TN, USA
e-mail: arndtd@ornl.gov

N. Fehn · M. Kronbichler (✉) · W. A. Wall
Technical University of Munich, Garching, Germany
e-mail: fehnl@lm.mw.tum.de; kronbichler@lm.mw.tum.de; wall@lm.mw.tum.de

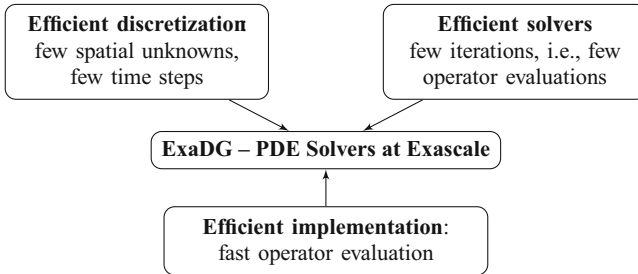
K. Kormann
Max Planck Institute for Plasma Physics, Garching, Germany
Technical University Munich, Garching, Germany
e-mail: katharina.kormann@ipp.mpg.de; katharina.kormann@tum.de

P. Munch
Technical University of Munich, Garching, Germany
Helmholtz-Zentrum Geesthacht, Geesthacht, Germany
e-mail: munch@lm.mw.tum.de; peter.muench@hzg.de

G. Kanschat · J. Witte
Heidelberg University, Heidelberg, Germany
e-mail: kanschat@uni-heidelberg.de; julius.witte@iwr.uni-heidelberg.de

1 Introduction

Exa-scale performance of numerical algorithms is determined by two factors, node-level performance and distributed-memory scalability to thousands of nodes over an Infiniband-type fabric. Additionally, the final application efficiency in terms of time-to-solution is strongly influenced by the choice of numerical methods, where a high sequential efficiency is essential. The project ExaDG aims to bring together these three pillars to create an algorithmic framework for the next generation of solvers for partial differential equations (PDEs). The guiding principles of the project are as follows:



If we define the overall goal to be a minimum of computational cost to reach a predefined accuracy, this aim can be split into three components, namely the efficiency of the discretization in terms of the number of degrees of freedom (DoFs) and time steps, the efficiency of the solvers in terms of iteration counts, and the efficiency of the implementation [22]:

$$\begin{aligned}
 E &= \frac{\text{accuracy}}{\text{computational cost}} \\
 &= \underbrace{\frac{\text{accuracy}}{\text{DoFs} \cdot \text{timesteps}}}_{\text{discretization}} \cdot \underbrace{\frac{1}{\text{iterations}}}_{\text{solvers/preconditioners}} \cdot \underbrace{\frac{\text{DoFs} \cdot \text{timesteps} \cdot \text{iterations}}{\text{computational cost}}}_{\text{implementation}}.
 \end{aligned} \tag{1}$$

We define computational cost as the product of compute resources (cores, nodes) times the wall time resulting in the metric of CPUh, the typical currency of supercomputing facilities.

Regarding the first metric, the type of discretizations in space and time are often the first decision to be made. Numerical schemes that involve as few unknowns and as few time steps as possible to reach the desired accuracy will be more efficient. This goal can be reached by using higher order methods which have a higher resolution capability, especially for problems with a large range of scales and some regularity in the solution [19]. However several possibilities and profound knowledge regarding the performance capability of potential algorithms on modern

hardware are still required to select those algorithms and implementations that are optimal with respect to the guiding metric of accuracy versus time-to-solution. High-order (dis-)continuous finite element methods are the basic building block of the ExaDG project due to their generality and geometric flexibility.

Regarding the second metric, solvers are deemed efficient if they keep the number of iterations minimal. We emphasize that “iterations” are defined in a low-level way as the number of operator evaluations, which is also accurate when nesting several iterative schemes within each other. Note that we assume that large-scale systems must be addressed by iterative solvers; in a finite element context sparse direct solvers are not scalable due to fill-in and complex dependencies during factorizations. One class of efficient solvers of particular interest to ExaDG are multigrid methods with suitable smoothers, which have developed to be the gold standard of solvers for elliptic and parabolic differential equations over the last decades. Here, the concept of iterations would accumulate several matrix-vector products within a multigrid cycle that in turn is applied in an outer Krylov subspace solver. Due to the grid transfer and the coarse grid solver, such methods are inherently challenging for highly parallel environments. As part of our efforts in ExaDG, we have developed an efficient yet flexible implementation in the deal.II finite element library [1, 15].

Third, the evaluation of discretized operators and smoothers remains the key component determining computational efficiency of a PDE solver. The relevant metric in this context is the throughput measured as the number of degrees of freedom (unknowns) processed per second (DoFs/s). An important contribution of our efforts is to both tune the implementation of a specific algorithm, but more importantly to also adapt algorithms towards a higher throughput. This means that an algorithm is preferred if it increases the DoFs/s metric, even if it leads to lower arithmetic performance in GFlop/s or lower memory throughput in GB/s. Operator evaluation in PDE solvers only involves communication with the nearest neighbors in terms of a domain decomposition of the mesh, which makes the node-level performance the primary concern in this regard. Since iterative solvers only require the action of the matrix on a vector (and a preconditioner), they are amenable to matrix-free evaluation where the final matrix entries are neither computed nor stored globally in memory in some generic sparse matrix format (e.g., compressed row storage). While matrix-free methods were historically often considered because they lower main memory requirements and allow to fit larger problems in memory [8], their popularity is currently increasing because they need to move less memory: Sparse matrix-vector products are limited by the memory bandwidth on all major computing platforms, so a matrix-free alternative promises to deliver a (much) higher performance.

The outline of this article is as follows. We begin with an introduction of matrix-free algorithms and a presentation of node-level performance results in Sect. 2. In Sect. 3, we describe optimizations of the conjugate gradient method for efficient memory access and communication. Next, we detail our multigrid developments, focusing on performance numbers and the massively parallel setup in Sect. 4 and on the development of better smoothers in Sect. 5. Application results in the field

of computational fluid dynamics are presented in Sect. 6, where the efficiency and parallel scalability of our discontinuous Galerkin incompressible turbulent flow solver are shown. An extension of the kernels to up to 6D PDEs is briefly presented in Sect. 7. We conclude with an outlook in Sect. 8.

2 Node-Level Performance Through Matrix-Free Implementation

An intuitive example of a matrix-free implementation is a finite difference method implemented by its stencil rather than an assembled sparse matrix [33]. For finite element discretizations with sufficient structure of the underlying mesh and low-order shape functions, a small number of stencils allows to represent the operator of a large-scale problem [8]. Such methods are used in the German exascale project TerraNeo, utilizing the regular data structures in hierarchical hybrid grids and embedded into a highly scalable multigrid solver for Stokes systems [31, 32]. By suitable interpolations, the stencils can be extended from the affine coarse grid assumption to also treat smoothly deformed geometries and variable coefficients [7].

For higher-order methods, finite element discretizations lead to fat stencils, making the direct evaluation inefficient even when done through stencils. An alternative matrix-free scheme used in ExaDG is to not compute the explicit DoF coupling and instead turn to integrals underlying the finite element scheme. As an example, we consider the constant-coefficient Laplacian

$$-\nabla^2 u = f \quad \text{in } \Omega, \quad u = g \quad \text{on } \partial\Omega, \quad (2)$$

whose weak form in a finite-dimensional setting is

$$(\nabla\varphi_i, \nabla u_h)_{\Omega_h} = (\varphi_i, f)_{\Omega_h}, \quad (3)$$

where $u_h(x) = \sum_{j=1:n} \varphi_j(x) u_j$ is the finite element interpolant of the solution with n degrees of freedom, φ_i denotes the test functions with $i = 1, \dots, n$, f is some right hand side, and Ω_h is the finite element representation of the computational domain Ω . The left-hand side of this equation represents a finite element operator, mapping a vector of coefficients $u = [u_i]_i$ to an output vector $v = [v_i]_i$ by evaluating the weak form for all test functions φ_i separately. A matrix-free implementation is obtained by evaluating the element-wise integrals

$$\begin{aligned} [(\nabla\varphi_i, \nabla u_h)_{\Omega_h}]_{i=1:n} &= \sum_K \int_{\hat{K}} (\mathcal{J}_K^\top \hat{\nabla}\varphi_i)^\top \left(\mathcal{J}_K^\top \sum_{j=1}^{n_{\text{dof,ele}}} \hat{\nabla}\varphi_j u_j^{(K)} \right) \det(\mathcal{J}_K) \, d\hat{x} \\ &\approx \sum_K I_K^\top \left[\sum_{q=1}^{n_q} (\hat{\nabla}\varphi_{i_K}(\hat{x}_q))^\top \underbrace{\mathcal{J}_K^{-1} \mathcal{J}_K^\top \det(\mathcal{J}_K) w_q}_{\text{physics at quadrature point}} \sum_{j=1}^{n_{\text{dof,ele}}} \hat{\nabla}\varphi_j(\hat{x}_q) u_j^{(K)} \right]_{i_K=1:n_{\text{dof,ele}}} \end{aligned} \quad (4)$$

by quadrature on n_q points per cell K . Here, K denotes the elements in the mesh, \hat{x} the coordinates of the reference element $\hat{K} = (0, 1)^d$, \mathcal{J}_K denotes the Jacobian of the mapping from the reference to the real cell, and w_q the quadrature weight. The operator I_K denotes the index mapping from $n_{\text{dof,ele}}$ element-local to global unknowns and defines the element-related unknowns $u^{(K)} = I_K u$.

On element K , the formulation of Eq. (4) consists of two nested sums over the elemental unknowns $u_j^{(K)}$, $j \in n_{\text{dof,ele}}$, and the quadrature points q . The result is tested against all test functions φ_{i_K} on the reference element, which are related to the global test functions φ_i through I_K . Since the metric terms do not depend on the shape function indices i_K and j , and the sum over j does not depend on i_K , the summations in the equation can be broken up into (1) an $dn_q \times n_{\text{dof,ele}}$ matrix operation to evaluate the reference element derivative of $u^{(K)}$ at the quadrature points, (2) the application of metric terms as well as other physics terms at n_q quadrature points, and (3) an $n_{\text{dof,ele}} \times dn_q$ matrix operation to test by all $n_{\text{dof,ele}}$ test functions and perform the summation over the quadrature points. The separation of point-wise physics evaluation at quadrature points is a common abstraction in integration-based matrix-free methods [29, 41, 49, 50].

For high-order finite element methods, the naive evaluation would involve all shape functions at all quadrature points, which is of complexity $O(k^{2d})$ for polynomials of degree k in d dimensions per element, or $O(k^d)$ per unknown, similarly to the fat stencil of the final matrix.

At this point, the structure in the reference-cell shape function and quadrature points can be utilized to lower the computational complexity. If the multi-dimensional shape functions are the tensor product of 1D shape functions, and if the quadrature formula is a tensor product of 1D formulas, the so-called sum-factorization algorithm can be used to group common factors along the various dimensions and break down the work into one-dimensional interpolations. Figure 1 visualizes the process of computing the interpolation of nodal values, visualized by black disks, to the values at the quadrature points. Rather than using a naive interpolation of cost $2(k+1)^{2d}$ operations, it can be done in $2d(k+1)^{d+1}$ operations instead. In matrix-vector notation, the interpolation of the gradient with respect to

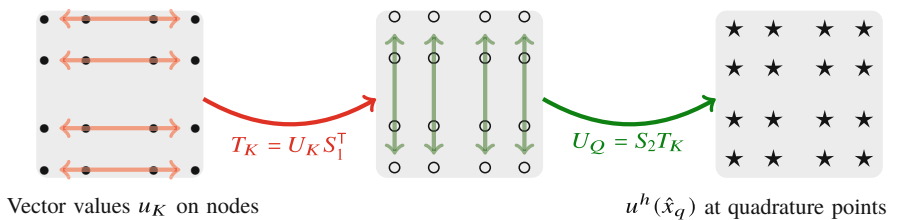


Fig. 1 Illustration of sum factorization for interpolation from node values on the left to the values in quadrature points (right)

\hat{x} , evaluated at quadrature points, can be written as

$$\begin{bmatrix} \partial \mathbf{u}_h / \partial \hat{x}_1 \\ \partial \mathbf{u}_h / \partial \hat{x}_2 \\ \vdots \\ \partial \mathbf{u}_h / \partial \hat{x}_d \end{bmatrix} = \begin{bmatrix} I \otimes \dots \otimes I \otimes D_1 \\ I \otimes \dots \otimes D_2 \otimes I \\ \vdots \\ D_d \otimes I \otimes \dots \otimes I \end{bmatrix} [S_d \otimes \dots \otimes S_2 \otimes S_1] \mathbf{u}^{(K)}. \quad (5)$$

Here, S_1, \dots, S_d denote the $n_q^{\text{1D}} \times (k + 1)$ interpolation matrices from the nodal values to the quadrature points, obtained by evaluating the 1D basis at all 1D quadrature points, and D_1, \dots, D_d the $n_q^{\text{1D}} \times n_q^{\text{1D}}$ matrices of the derivatives of the Lagrangian basis in quadrature points. In this form, the multiplication by Kronecker matrices is implemented by small matrix-matrix multiplications.

Sum factorization was initially developed in the context of spectral element methods by Orszag [61], see [19] for an overview of the developments. In [12], sum factorization was compared against assembled matrices with the goal to find the best evaluation strategy among assembled matrices and matrix-free schemes. For hexahedral elements considered in this work, the memory consumption and arithmetic complexity indicate that this is the case already for quadratic basis functions [11, 49], with a growing gap for higher polynomial degrees.

2.1 Implementation of Sum Factorization in the deal.II Library

As part of the ExaDG project, we have developed efficient implementations in the deal.II finite element library [1, 4] with the following main features, see [50] for a detailed performance analysis:

- support for both continuous [49] and discontinuous finite elements on uniform and adaptively refined meshes with hanging nodes and deformed elements,
- support for arbitrary polynomial expansions on quadrilateral and hexahedral element shapes as well as tensor product quadrature rules,
- minimization of arithmetic operations by using available symmetries, such as the even-odd decomposition [69] and a switch between the collocation derivative (5) for $n_q^{\text{1D}} \approx k + 1$ quadrature points or an alternative variant based on derivatives of the original polynomials as used in [49] and discussed in [29],
- flexible implementation of operations at quadrature points,
- vectorization across several elements to optimally use SIMD units (AVX, AVX-512, AltiVec) of modern processors,
- applicability to modern multi-core CPUs as well as GPUs [51, 57],
- data access optimizations such as element-based loops for DG elements [50, 56],
- and MPI implementation with tight data exchange as well as MPI-only and shared-memory models [43, 48, 54].

The concept of matrix-free evaluation with sum factorization has been widely adopted by now, like in the deal.II [1], DUNE [5, 40, 60], Firedrake [63], mfem [2], Nek5000 [28] or Nektar++ [13] projects. These fast evaluation techniques are directly applicable to explicit time stepping schemes, as we have demonstrated for wave propagation in [42, 53, 65–68] and the compressible Navier–Stokes equations [24]. The proposed developments make matrix-free evaluation of high-order DG operators reach a throughput in unknowns per second almost as high as for optimized 5-wide finite difference stencils in a CFD context [75], despite delivering much higher accuracy.

2.2 Efficiency of Matrix-Free Implementation

In Fig. 2, we give an overview of the achieved performance with our framework applied to the discontinuous Galerkin interior penalty (IPDG) discretization of the 3D Laplacian on an affine geometry. The most advanced implementation presented in [50] is used, namely a cell-based loop with a Hermite-like basis for minimal data access [56]. The figure lists the throughput, which is measured by recording the run time of the matrix-vector product in an experiment with around 50 million DoFs (too large to fit into caches), and reporting the normalized quantity DoFs/s obtained by dividing the number of DoFs by the measured run time. The code is run on a single node of six dual-socket HPC systems from the last decade with a shared-memory parallelization with OpenMP, threads pinned to logical cores with the close affinity rule, and using streaming stores to avoid the read-for-ownership data transfer [33] on the result vector. As systems, we consider a 2×8 core AMD Opteron 6128 system from 2010, a 2×8 core Intel Xeon 2680 Sandy Bridge from 2012 (as used in the SuperMUC phase 1 installation in Garching, Germany), a 2×8 core Intel Xeon 2630 v3 (Haswell) representing a medium-core count chip from

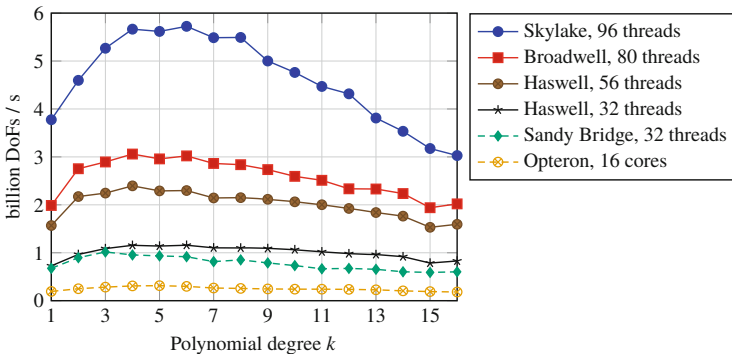


Fig. 2 Throughput of matrix-free evaluation of the IPDG discretization of the 3D Laplacian on an affine grid

2014, a 2×14 core Intel Xeon 2697 v3 (Haswell) representing a high-core count chip of the same generation (as used in the SuperMUC phase 2 installation), a 2×20 core Intel Xeon 2698 v4 (Broadwell) system from 2016, and a 2×24 core Intel Xeon Platinum 8174 from 2017, labeled ‘Skylake’ in the remainder of this work, and installed in the SuperMUC-NG supercomputer. The chips are operated at 2.0 GHz, 2.7 GHz, 2.4 GHz, 2.6 GHz, 2.2 GHz, and 2.3 GHz, respectively, and all run with fully populated memory interfaces. The Intel machines are run with 2-way hyperthreading, e.g. with 96 threads for the Xeon Platinum Skylake.

The throughput results in Fig. 2 demonstrate the advancements of hardware during the last decade. In particular the increased width of vectorization, from 2 to 4 doubles with Sandy Bridge and from 4 to 8 doubles with Skylake, are clearly visible. Furthermore, the comparison between Sandy Bridge and the smaller Haswell system reveals the benefit of fused multiply-add (FMA) instructions and higher L1 cache bandwidth of the latter: For low polynomial degrees with a modest number of FMA instructions, Sandy Bridge with its higher frequency can approximately deliver the same performance as Haswell. As the polynomial degree is increased, the arithmetic work is increasingly dominated by FMAs in the sum factorization sweeps similar to (5) as shown in [50], and Haswell pulls ahead. Finally, while we observe a throughput of up to 5.7 billion DoFs/s on Skylake (with up to 1.35 TFlop/s for $k = 8$), we observe a relatively strong decrease of performance for polynomial degrees $k \geq 13$: This is because the vectorization across elements leads to an excessive size of the temporary data within sum factorization—here, a different vectorization strategy could lead to better results. However, we consider the polynomial degrees $3 \leq k \leq 8$ most interesting for practical simulations, where almost constant throughput in terms of DoFs/s is reached. This somewhat surprising result, given the expected $O(1/k)$ complexity of throughput for sum factorization, is because face integrals and memory access with an $O(1)$ complexity are dominant. Compared to our initial implementation in 2015, which achieved a throughput of 0.32 billion DoFs/s on Sandy Bridge with degree $k = 3$, the progress in software technologies allowed us to reach 1.02 billion DoFs/s on the same system. For Intel Skylake, where memory access is more important, the software progress of our project is more than $4\times$.

Figure 3 shows the throughput normalized by the number of cores for polynomial degree $k = 4$ over the different hardware generations. For operator evaluation with discontinuous elements and face integrals, approximately 200 floating point operations per unknown are involved with our optimized implementations [50]. At the same time, we must access at least 16 byte (read one double, write one double) plus some neighbors that are not cached, so the arithmetic intensity is around 8–12 Flop/Byte, close to the machine balance of the Skylake Xeon. This means that both memory bandwidth and arithmetic performance are relevant for performance (on one Skylake node, we measured memory throughput of around 160 GB/s, compared to the STREAM limit of 205 GB/s). Likewise, continuous elements evaluated on an affine mesh have seen a considerable increase in throughput per core (arithmetic intensity of 7 Flop/Byte). However, the improvement has been much more modest for continuous elements evaluated on curved elements. In this setting, separate

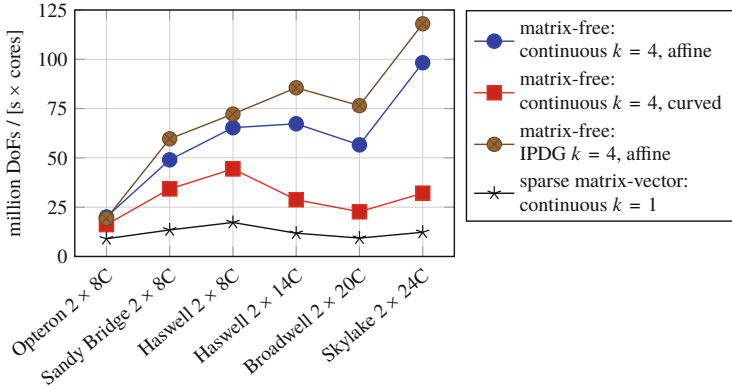


Fig. 3 Evolution of throughput of matrix-vector product per core (computed on a fully populated node and divided by the number of cores) with matrix-free evaluation with $k = 4$ versus a sparse matrix-vector product with continuous linear elements on various hardware

metric terms for all quadrature points and all elements are needed (as opposed to a single term per element in the affine mesh case), reducing the arithmetic intensity to around 1.2 Flop/Byte.¹

Figure 3 also contains the evolution of performance of a sparse matrix-vector product for tri-linear continuous finite elements. The performance is much lower due to the aforementioned memory bandwidth limit, and has hardly improved per core on Skylake over the dated Opteron architecture. This illustrates the effect of the so-called memory wall. We emphasize that the sparse matrix-vector product for $k = 1$ is more than three times slower than even the matrix-free evaluation for $k = 4$ on curved elements. Hence, high-order methods with matrix-free implementations are faster per unknown on newer hardware, *in addition* to their higher accuracy.

3 Performance-Optimized Conjugate Gradient Methods

The developments of matrix-free implementations presented in the previous section result in a throughput for evaluation of the IPDG operator in Fig. 2 of up to 5.7 billion DoFs/s on Skylake. This is equivalent in time to the mere access of 4.5 doubles per DoF (either reading or writing). In other words, our developments have made the operator evaluation so fast that the matrix-vector product may no longer be the dominant operation in algorithms like the conjugate gradient (CG) method preconditioned by the diagonal, or Chebyshev smoothers. These algorithms

¹The merged final coefficient tensor $\mathcal{J}_K^{-1} \mathcal{J}_K^{-T} \det(\mathcal{J}_K) w_q$ is used for the present results, i.e., 6 doubles per quadrature point [29, 51].

involve access to between 6 and 18 vectors for vector updates, the application of the diagonal preconditioner, and inner products. For optimal application performance it is therefore necessary to look into the access to vectors. As proposed in our work [51, 56, 65], merging the vector operations can improve throughput by up to a factor of two, and in particular for the DG case with cell-based loops which allow for a single pass through data [48, 56]. Fusion of different steps of a scheme has also been proposed for explicit time integrators in [14].

For the assessment of optimization opportunities on the algorithm level that goes beyond the matrix-vector product, we consider a high-order finite element benchmark problem suggested by the US exascale initiative “Center for Efficient Exascale Discretization” (CEED). The benchmark involves a continuous finite element discretization of the Laplacian (3), using matrix-free operator evaluation within a conjugate gradient solver preconditioned by the matrix diagonal. In this study, we consider the case BP5 [29], see also <https://ceed.exascaleproject.org/bps/>, which integrates the weak form (4) of polynomial degree k using a Gauss–Lobatto quadrature formula with $n_q^{\text{1D}} = k + 1$ quadrature points on a cube with deformed elements. While this integration is not exact, it is the typical spectral element setup with an identity interpolation matrix $S_i = I$ in Eq. (5).

Figure 4 lists the contributors to run time for the plain conjugate gradient method preconditioned by the point-Jacobi method as a function of the problem size for the polynomial degree $k = 6$. Here, the metric terms \mathcal{J}_K are computed on the fly from a tri-linear representation of the geometry. Three different performance regimes can be distinguished in the graph: To the left, there is not enough parallelism given the domain decomposition on 48 MPI ranks and batches of 8 elements due to vectorization—indeed, at least 85,000 DoFs are needed to saturate all cores and SIMD lanes. Furthermore, the synchronization barriers due to the inner products in the conjugate gradient method also lead to a slowdown. As the problem size and parallelism increase, the run times decrease significantly and reach a minimum for a problem size around one million DoFs. Here, all data involved in the algorithm

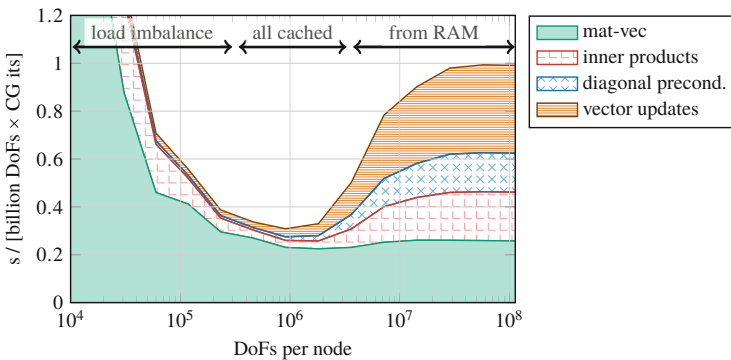


Fig. 4 Breakdown of times per CG iteration in CEED benchmark problem BP5 [29] for the plain conjugate gradient method with $k = 6$ on one node of dual-socket Intel Skylake

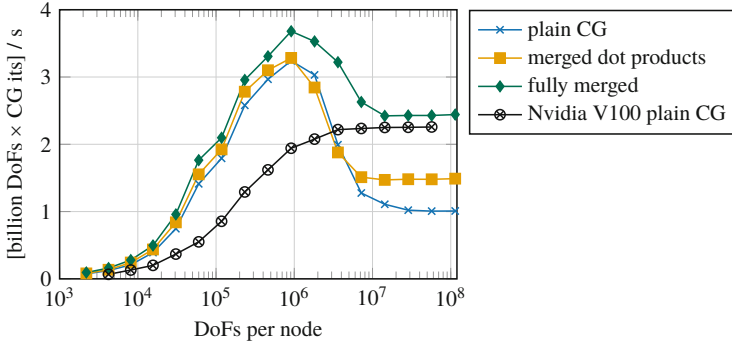


Fig. 5 Study of merged vector operations for conjugate gradient solver for the CEED benchmark problem BP5 [29] on one node of dual-socket Intel Skylake for $k = 6$

fits into the approximately 110 MB of L2+L3 cache on the processors. As the size is further increased, caches are exhausted and most data must be fetched from slow main memory. As a consequence, the run time of the solver increases significantly, and the vector updates, the diagonal preconditioner, and the inner products take a significant share. Note that all vector operations use the hardware optimally with a memory throughput of 205 GB/s.

In order to improve performance, we have therefore developed conjugate gradient implementations with merged vector operations by loop fusion. Figure 5 compares three variants of the conjugate gradient solver: the plain conjugate gradient method runs all vector operations through high-level vector interfaces with separate loops for addition and inner products. In the “merged dot products”, we have merged the dot product $p^T A p$ following the matrix-vector product into the loop over the elements, and merged the vector updates to the residual and solution with the dot product for $r^T P^{-1} r$. Here, r denotes the residual vector, p the search direction of the conjugate gradient method, A the matrix operator (represented in a matrix-free way), and P^{-1} the diagonal preconditioner. However, the improvements with this algorithm are relatively modest.

Much more performance can be gained by creating a conjugate gradient variant we call “fully merged”: Here, each CG iteration performs a single loop through all vector entries and ideally reads 5 vectors (solution, residual, search direction, temporary vector to hold the matrix-vector product, and diagonal of preconditioner) and writes four (solution, residual, search direction, temporary vector). All vector updates of the previous CG iteration are scheduled before the matrix-vector product and all inner products are scheduled after the matrix-vector product. The vector operations are interleaved with the loop over elements, ensuring that dependencies due to the access pattern of the loop and the MPI communication are fulfilled (this leads to slightly more access in practice). This approach applies the preconditioner several times with partial sums to construct the inner products with a single MPI_Allreduce, trading some local computations for the decreased memory

access. Of course, fusing the preconditioner into the loop assumes that it is both cheap to apply and does not involve long-range coupling between the DoFs. The results in Fig. 5 show that performance in the saturated limit, i.e., for large sizes beyond 10^7 DoFs, is 2.5 times faster than with the plain CG iteration. Interestingly, this also improves performance for the sizes fitting into caches, which is due to less synchronization and reducing access to the slower L3 cache.

To put the performance of the fully merged case on Intel Skylake into perspective, we compare with executing the plain CG method on an Nvidia V100 GPU using the implementation from [51, 57]: even though the GPU runs with around 700 GB/s of memory throughput, the performance is higher on Intel Skylake with only 200 GB/s from RAM memory because the merged loops significantly increase data locality. Furthermore, on the GPU we do not compute the metric terms on the fly, but load a precomputed tensor $\mathcal{J}_K^{-1} \mathcal{J}_K^T \det(\mathcal{J}_K) w_q$ which is faster due to reduced register pressure, see also the analysis for BP5 in [71]. We also note that the GPU results with our implementation are faster than an implementation with the OCCA library described in [29] with up to 0.6 billion DoFs/s on a V100 of the Summit supercomputer. The reason is that our implementation uses a continuous finite element storage that does not duplicate the unknowns at shared vertices, edges and faces, which reduces the memory access by about a factor of two. Furthermore, the results from [29] involve a separate gather/scatter step with additional memory transfer to enforce continuity, while this is part of the operator evaluation within a single loop in our code.

Figure 6 lists the achieved throughput with a fully merged conjugate gradient solver for polynomial degrees $k = 2, \dots, 8$, the most interesting regime for our solvers. We use a tri-linear representation of geometry and compute the geometric

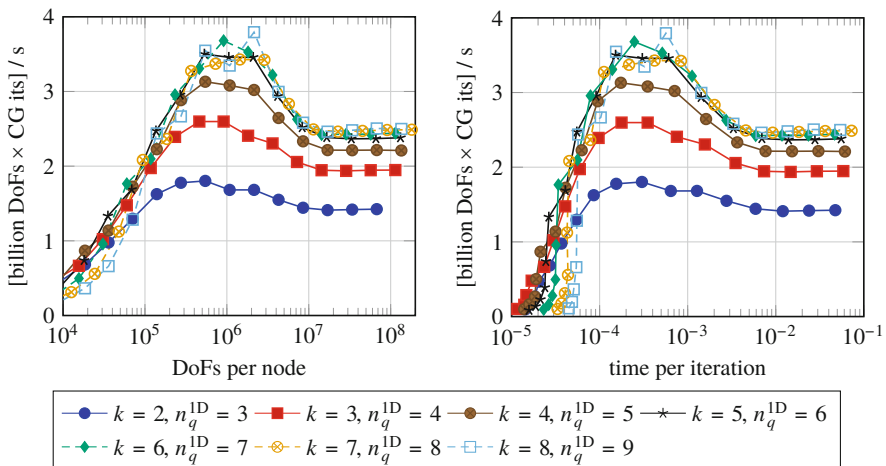


Fig. 6 Throughput of the CEED benchmark problem BP5 [29] on one node of dual-socket Intel Skylake for $k = 2, \dots, 8$ with fully merged conjugate gradient solver

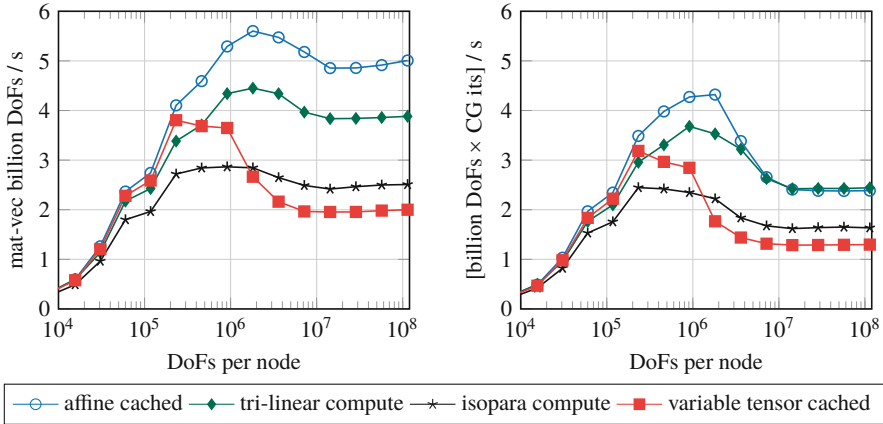


Fig. 7 Study of geometry representation for the CEED benchmark problem BP5 [29] on one node of dual-socket Intel Skylake with $k = 6$ for matrix-vector product only (left) and with fully merged vector operations in the conjugate gradient solver (right)

factors on the fly. Throughput is somewhat lower for quadratic and cubic elements because the geometry data located in the vertices is still noticeable.

The results in Fig. 3 motivate the analysis of the representation of the geometry in the matrix-vector product, with results presented in Fig. 7. The figure lists both the throughput of the matrix-vector product in the left panel and the throughput of the complete CG iteration with merged vector operations. Highest performance is obtained for the affine mesh case where our implementation can compress the memory access of the Jacobian. While this case is excluded from the CEED BP5 specification that requires a deformed geometry [29], it is an interesting baseline to compare against. Using separate tensors for each quadrature point, “variable tensor cached”, is equally fast as the affine case as long as data fits into caches. However, performance drops once the big geometric arrays must be fetched from main memory. For the case the geometry is computed on the fly from a tri-linear representation of the mesh, i.e., the vertices, the matrix-vector product is slower than the affine variant. For the conjugate gradient solver, however, we observe that the two reach essentially the same performance for five million and more DoFs, as they are both limited by the memory bandwidth from vector access. The “tri-linear compute” case involves a higher Flop/s rate with almost 700 GFlop/s, as compared to the throughput of 330 GFlop/s for the affine mesh case. This means that the merged vector operations allow us to fit additional computations behind the unavoidable memory transfer *without* affecting application performance. Finally, an isoparametric representation of the geometry (labeled “isopara compute” in Fig. 7) can also be computed on the fly by sum factorization from a k th degree polynomial [50]. While this case is obviously slower than the precomputed variable-tensor case from caches, it leverages higher performance when data must be fetched from main memory.

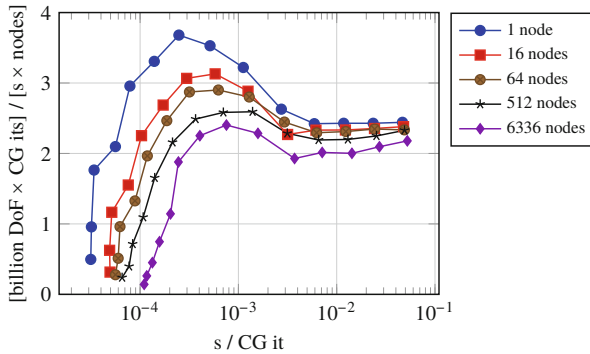


Fig. 8 Throughput of the CEED benchmark problem BP5 [29] with $k = 6$ on up to the full SuperMUC-NG machine. Throughput normalized per node

The tri-linear and isoparametric cases are not equivalent, as only the latter represents higher order curved boundaries. Intermediate polynomial degrees for the geometry are conceivable, which would land between the two in terms of application throughput. To combine the higher performance of the former, we plan to investigate the tradeoffs in more detail in the future, e.g. by using a k -degree representation on a single layer of elements at the boundary and a tri-quadratic representation in the domain's interior.

Finally, Fig. 8 shows the weak scaling of the BP5 benchmark problem up to the full size of the SuperMUC-NG machine with 6336 nodes and 304,128 cores. The data is normalized by reporting the number of DoF per node, so ideal weak scaling would correspond to coinciding lines. While the saturated performance is scaling well, giving a sustained performance of up to 4.4 PFlop/s,² most of the in-cache performance advantage is lost due to the communication latency over MPI, see also [62] for limits with MPI in PDE solvers. Defining the strong scaling limit as the point where throughput reduces to 80% of saturated performance [29], it is reached for wall times of $56\ \mu\text{s}$ on 1 node. On 512 nodes, the strong scaling limit is already around $180\ \mu\text{s}$, whereas it is $245\ \mu\text{s}$ on the full SuperMUC-NG machine. Note that even though most optimizations presented in this section have addressed the node-level performance, we have also considered the strong scaling in our work—indeed, the strong scaling on SuperMUC-NG is excellent with a limit around 5 times lower than the BlueGene-Q results presented in [29].

²The LINPACK performance of SuperMUC-NG according to the top500 list is 19.4 PFlop/s. Considering that we use an iterative solver for PDE with optimization of throughput, this is an extremely good value.

4 Geometric Multigrid Methods in Distributed Environments

Multigrid methods are efficient solvers for the linear systems arising from the discretization of elliptic problems, see [30] for a recent efficiency evaluation and [35] for a projection of elliptic solver performance to the exascale setting. They apply simple iterative schemes called smoothers on a hierarchy of coarser problem representations. On each level of the hierarchy, the smoothers address the high-frequency content of the solution by smoothening the error. On a sufficiently coarse level with a small number of unknowns, a direct solver can be applied. The multigrid algorithm can be realized by a V-cycle as illustrated in Fig. 9 or some related cycle (W-cycle or F-cycle). In the matrix-free high-order finite element context, variants of the Chebyshev iteration around a simple additive scheme, such as point-Jacobi or approximate block-Jacobi with some rank- d approximation of the cell matrix, are state of the art. The results in this section are based on this selection. Overlapping Schwarz schemes are a new development detailed in Sect. 5 below.

In terms of finding the coarser representations for the multigrid hierarchy, high-order finite element and discontinuous Galerkin methods permit a range of options. The hierarchy can both be constructed by coarser meshes (h -multigrid), by lowering the polynomial degree (p -multigrid), by a discontinuous-continuous transfer as well as algebraically based on the matrix entries only (algebraic multigrid). The latter do not fit into a matrix-free context, since they explicitly rely on a sparse matrix and also often are not robust enough as the degree increases. As it has been shown by the work [70], scalability to the largest supercomputers is much more favorable if knowledge about coarsening by a mesh can be provided. In other words, geometric multigrid is to be preferred over algebraic multigrid in case there is such structure in the problem.

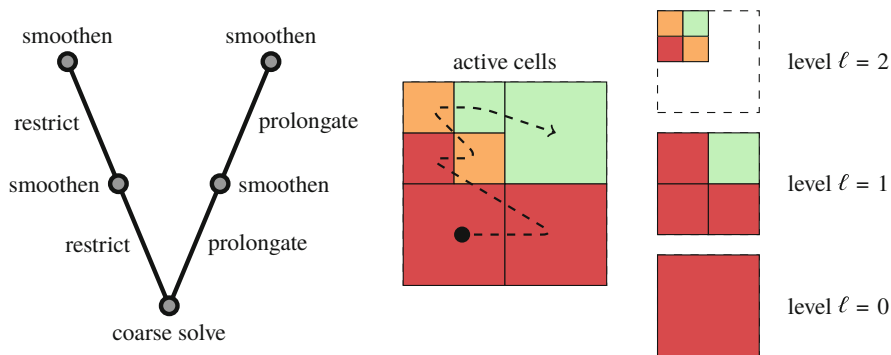


Fig. 9 Illustration of multigrid V-cycle with smoothing on each level and restriction/prolongation between the levels (left) and exemplary partitioning of a grid with adaptive refinement partitioned among 3 processors. The partitioning of the active cells is shown in the mid panel and on the various multigrid levels on the right panel

For these reasons, we have developed a comprehensive geometric multigrid framework with deal.II. In [27], a hybrid multigrid solver with all possibilities of h -, p -, and algebraic coarsening has been combined in a flexible framework, with the possibility to perform an additional c -transfer from discontinuous to continuous function spaces for the DG case. In terms of the h -MG method on adaptive meshes, the deal.II library implements the local smoothing algorithm [9, 36, 37] where smoothing is done level by level. Our work [15] developed a communication-efficient coarsening strategy for this setup, at the cost of a load imbalance for smoothing on the multigrid levels with adaptively refined meshes. The tradeoffs in this choice and the associated costs have been quantified by a performance model in [15].

Figure 10 shows the results of two strong scaling experiments of the multigrid V-cycle with the h -multigrid infrastructure of the deal.II library. The uniform grid and a typical adaptively refined case are compared for the same problem size of 137 million and 46 billion DoFs, respectively, see [15] for details on the experiment. Differences in run time are primarily due to the load imbalance for the level operations. The results demonstrate optimal parallel scaling of both the uniform and adaptively refined cases down to around 10^{-2} s, with a slightly better strong scaling of the adaptive case due to the slower baseline. This performance barrier—typical for strong scaling of multigrid schemes in general—can be explained by the specific type of global communication in this algorithm: from the fine mesh level with many unknowns distributed among a large number of cores, we transfer residuals to coarser meshes with restriction operators until the coarse grid solver is either run on a single core or with few cores in a tightly coupled manner. Then, the coarse-grid corrections are broadcast during prolongation, involving all processors again. The communication pattern of a multigrid V-cycle thus relates to a tree-based

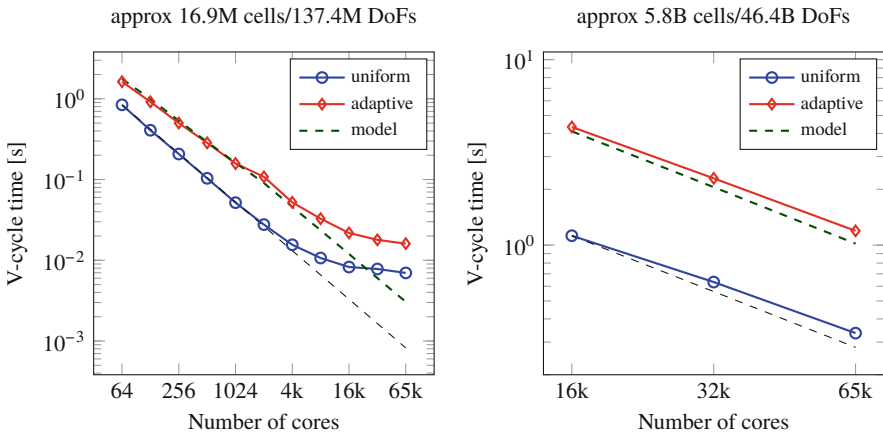


Fig. 10 Strong scaling of geometric multigrid V-cycle for 3D Laplacian on uniform and adaptively refined mesh using continuous Q_2 elements with matrix-free evaluation on up to 4096 nodes (64k cores) of 2×8 core Intel Sandy Bridge (SuperMUC phase 1). Adapted from [15]

implementation of `MPI_Allreduce`, with the difference that the communication tree is induced by the grid and substantial operations, namely smoothing and level transfer, are intermixed with the communication. In this particular case, nine matrix-vector products with nearest-neighbor communication are performed per level (eight in the smoother and one for the residual before restriction). In addition, two vertical nearest-neighbor exchange operations are done in restriction and prolongation. A typical matrix-vector product with up to 26 neighbors takes around 10^{-4} s on the chosen Intel Sandy Bridge system when run on a few thousands of nodes [52]. When done on seven levels plus the coarse mesh for the uniformly refined 137 million DoFs case, the expected saturated limit of around 8 ms is exactly seen in the figure. On the newer SuperMUC-NG machine, a latency barrier per V-cycle of around 2–4 ms per V-cycle has been measured, depending on the number of matrix-vector products for the level smoothers. This limit is attractive compared to alternative solvers for elliptic problems such as the fast multipole method or the fast Fourier transform [30, 35].

Multigrid schemes are at the heart of incompressible flow solvers through the pressure Poisson equation, as detailed in Sect. 6 below. Applications of matrix-free geometric multigrid to continuum mechanics were presented in [18] and to electronic calculations with sparse multivectors in [16, 17].

As an example of the large-scale suitability of the developed multigrid framework, Fig. 11 shows two scaling experiments on the SuperMUC-NG supercomputer with up to 304,128 cores of Intel Skylake. Black dashed lines denote ideal strong scaling along a line and weak scaling with a factor of 8 between the lines. The computational domain is a cube meshed by hexahedral elements, using the affine mesh code path for matrix-free algorithms discussed in Sect. 2. A consistent Gaussian quadrature with $n_q^{1D} = k + 1$ points is chosen. We run a conjugate gradient solver to a relative tolerance of 10^{-3} compared to the initial unpreconditioned residual. This setup is motivated by applications where a very good initial guess is already

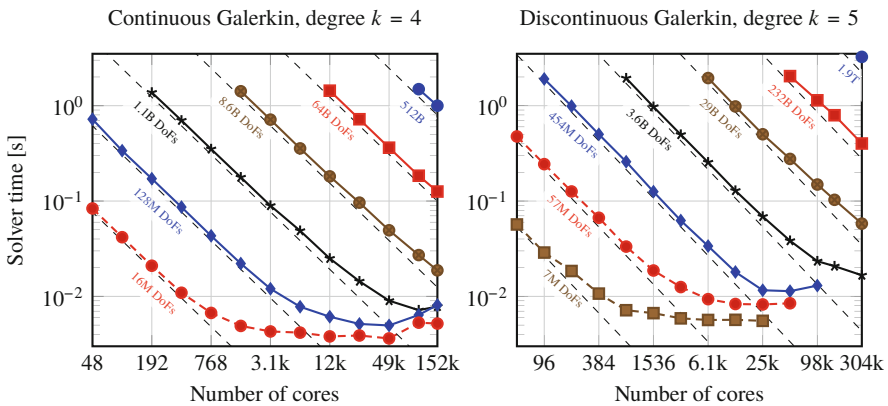


Fig. 11 Multigrid strong scaling analysis for tolerance 10^{-3} with 2 CG iterations

available, e.g. by extrapolation of solutions from the old time step [22, 45], and only a correction is needed. More accurate solves are obtained by tighter tolerances or by full multigrid setups [51]. The multigrid V-cycle is run in single precision to increase throughput, together with a double precision correction through the outer CG solver. This setup has been shown in [51] to increase throughput by around $1.8\times$ without affecting the multigrid convergence.

In the left panel of Fig. 11, we present results for a continuous Galerkin discretization with a polynomial degree $k = 4$. A pure geometric coarsening down to a single mesh element is used. A Chebyshev iteration of degree five based on the matrix diagonal, i.e., point Jacobi, is used on all levels for pre- and post-smoothing. The maximal eigenvalue $\tilde{\lambda}_{\max}$ is estimated by 15 iterations of a conjugate gradient solver and the Chebyshev parameters are set to smoothen in a range $[0.06\tilde{\lambda}_{\max}, 1.2\tilde{\lambda}_{\max}]$. As a coarse solver, we use a Chebyshev iteration with the degree set to reduce the residual by 10^3 in terms of the Chebyshev a-priori error estimate [74]. We observe ideal weak scaling and strong scaling to around 10^{-2} s. More importantly, the absolute run time is excellent: For instance, the 8.6 billion DoF case on 1536 cores is solved in 1.4 s, i.e., 4.0 million DoFs are solved per core per second.

The right panel of Fig. 11 shows the result for multigrid applied to an IPDG discretization with $k = 5$. Here, we use a transfer from the discontinuous space to the associated continuous finite element space with $k = 5$ on the finest mesh level (see [3] for the theoretical background and [27] for the multigrid context) and then progress by h -coarsening to a single element. On the DG level, we use a Chebyshev smoother around a block-Jacobi method, with the block-Jacobi problems inverted by the fast diagonalization method [58]. On all continuous finite element levels, a Chebyshev iteration around the point-Jacobi method is used. The degree of the Chebyshev polynomial is six. This solver setup achieves a multigrid convergence rate of about 0.025, i.e., reduces the residual by 3 orders of magnitude with two V-cycles. If used in a full multigrid setting [51], a single V-cycle on the finest level would suffice to solve the problem to discretization accuracy. Merged vector operations with a Hermite-like basis for the Chebyshev iteration are used according to [56]. The final application performance of the largest computation on 1.9 trillion DoFs is 5.9 PFlop/s, with 5.6 PFlop/s done in single precision and 0.27 PFlop/s in double precision. The limiting factor is mostly memory transfer, however, with an application throughput of around 175 GB/s per node (the STREAM limit of one node is 205 GB/s).

5 Fast Tensor Product Schwarz Smoothers

In Sect. 4, we have discussed a scalable implementation of geometric multigrid methods, obtaining an efficient solver in the sense of cost per iteration. It employs the matrix-free operator implementation from Sect. 2 in order to reduce the computational cost for residuals and grid transfer. The missing building block for our cost

model in Eq. (1) is an efficient implementation (in terms of computational cost per DoF) of an efficient smoother (in terms of number of multigrid iterations).

The main challenge consists of finding preconditioners whose cost is similar to operator evaluation. So far, we have discussed Chebyshev smoothers, which can be implemented matrix-free in a straight-forward fashion. Alas, their performance is not robust for higher order elements. Likewise, from an arithmetic cost point of view sparse matrices can be competitive at most for moderate polynomial degrees $k = 2, 3$ [55] or when done via auxiliary spaces of linear elements on a subdivided grid using some matrix-based preconditioner. However, Fig. 3 shows that even sparse matrices for linear elements are up to 10 times slower than the matrix-free operator evaluation. It seems that only the two SPPEXA projects ExaDUNE and ExaDG have addressed this question in [6, 76]. While [6] focuses on iterative solution of cell problems for multigrid smoothing, we consider domain decomposition based smoothers in the form of multilevel additive and multiplicative Schwarz methods based on low-rank tensor approximations. They consist of a subdivision of the mesh on each level into small subdomains consisting either of a single cell, or of the patch of cells sharing a common vertex. On each of these subdomains, local finite element problems are solved. Comparing with operator application, these smoothers share the structural property of evaluation of local operators on mesh cells or on a patch of cells. They differ by the fact that the smoothers involve local inverses instead of local forward operators, and that these local inverses in general are not amenable to a tensor decomposition like sum factorization. There is one exception though, namely separable differential operators. In d dimensions these can be written in the form

$$L = I_d \otimes \cdots \otimes I_2 \otimes L_1 + \cdots + L_d \otimes I_{d-1} \otimes \cdots \otimes I_1, \quad (6)$$

where L_k are one-dimensional differential operators and I_k are identity operators in directions $k = 1, \dots, d$. This representation transfers to finite element operators with tensor product shape functions in a straight-forward way, reading I_k as one-dimensional mass matrices M_k .

Due to [58], the inverse of L can be represented as the product

$$L^{-1} = Q \Lambda^{-1} Q^T, \quad (7)$$

with the diagonal matrix $\Lambda = I_d \otimes \cdots \otimes I_2 \otimes \Lambda_1 + \cdots + \Lambda_d \otimes I_{d-1} \otimes \cdots \otimes I_1$, where I_k denote identity matrices, and a rank-1 decomposition $Q = Q_d \otimes \cdots \otimes Q_1$. The tensor factors are obtained by solving d generalized eigenvalue problems

$$\begin{aligned} \Lambda_k &= Q_k^T L_k Q_k, \\ I_k &= Q_k^T M_k Q_k, \quad k = 1, \dots, d. \end{aligned} \quad (8)$$

Thus, the computational effort for computing the inverse has been reduced from $O(k^{3d})$ to $O(dk^3)$ and for the application of local solvers from $O(k^{2d})$ to $O(dk^{d+1})$

by exploiting sum factorization. Based on this technique, we have implemented a geometric multigrid method in [76] based on earlier work in [37–39].

5.1 The Laplacian on Cartesian Meshes

In order to test our concept and to obtain a performance baseline for more complicated cases, we first attend to the case where the decomposition described above can be applied in a straightforward way, namely the additive Schwarz method with subdomains equal to mesh cells. As Table 1 shows, it yields an efficient preconditioner with less than 25 conjugate gradient steps for a gain of accuracy of 10^8 . While it is uniform in the mesh level, it is not uniform in the polynomial degree due to the increasing penalty parameter of the interior penalty method. The computational effort for a smoothing step based on local solvers in the form (7) is below the effort for a matrix-free operator application for polynomial degrees between 3 and 15 in three dimensions because it only involves operations on cells. The setup time for computing Q and Λ is even less. Thus, in the context of the performance analysis of the conjugate gradient method in Sect. 2, it barely adds to the cost per iteration step, but reduces the number of matrix-vector products when comparing to the accumulated numbers within a Chebyshev/point Jacobi method, and almost independently of polynomial degree.

In view of application to incompressible flow, we also study vertex patches as typical subdomains for smoothing. First, we observe that a regular vertex patch with 2^d cells attached to a vertex inherits the low-rank tensor product structure from its cells, possibly after renumbering due to changes in orientation. Thus, we can apply the same method as on a single cell, resulting effectively in a factor 2^d in the complexity estimates above. Patches around vertices with irregular topology like 3 or 5 cells in two dimensions do not possess a tensor product structure. Fortunately,

Table 1 Fractional CG iterations, preconditioned by h -MG with additive Schwarz smoother on cells

Levels	Convergence steps			
	$k = 3$	$k = 4$	$k = 7$	$k = 10$
2D				
7	14.5	14.3	18.8	20.9
8	14.5	14.3	18.8	20.9
9	14.5	14.3	18.8	20.9
10	14.5	14.3	18.8	20.9
3D				
3	16.7	16.8	22.0	24.5
4	17.1	17.0	22.0	24.5
5	17.2	17.0	22.1	24.6
6	17.1	17.0	22.1	24.7

Relative solver tolerance of 10^{-8} and relaxation parameter $\hat{\omega} = 0.7$

on meshes obtained by refinement of a coarse mesh, they are all determined by irregularities of the coarse mesh and thus small in number.

Vertex patches lead to overlapping decompositions with overlap of at least 4 and 8 in two and three dimensions, respectively. From the analysis of Schwarz methods, it becomes clear that a multiplicative method is required for highest multigrid convergence rates. In order to parallelize such a smoother and to avoid race conditions, mesh cells are colorized, that is, they are separated into “colors” such that patches of the same “color” do not share any common face or cell. As a consequence, the multiplicative method coincides with an additive method within each color, such that we can execute the local solvers in parallel within each color, and the colors sequentially. Typical convergence results for the Laplacian are reported in Table 2, suggesting that this scheme is almost a direct solver.

The vertex patch has 4 and 8 times as many unknowns as a single mesh cell in two and three dimensions, respectively. Thus, the effort for a smoothing step with 16 colors and the optimizations described above turns out to be about 20 to 24 times the effort of a matrix-free operator application, measured over polynomial degrees from 3 to 15. This seems excessive at first glance, but it must be kept in mind that the Chebyshev smoother of degree 6 used in Fig. 11 also involves 12 matrix-vector products for pre- and post-smoothing. Furthermore, the current scheme comes with a reduction of the number of steps by a factor 10 compared to the additive cell smoother for the Laplacian, which makes it almost competitive [76]. Finally, the iteration counts are independent of the polynomial degree, making the scheme attractive for higher degrees. Moreover, we point out that this smoother also allows for the solution of a Stokes problem in four iteration steps [39].

Table 2 Fractional GMRES iterations, preconditioned by h -MG with multiplicative Schwarz smoothers on vertex patches

Levels	Convergence steps				Colors	Levels	Convergence steps				Colors
2D	$k = 3$	$k = 4$	$k = 7$	$k = 10$		2D	$k = 3$	$k = 4$	$k = 7$	$k = 10$	
7	2.5	2.5	2.1	2.1	8	7	2.9	2.9	2.6	2.5	17
8	2.5	2.5	2.1	2.0	8	8	2.9	2.9	2.6	2.5	17
9	2.5	2.4	2.1	2.0	8	9	2.9	2.9	2.6	2.5	17
10	2.5	2.4	2.0	2.0	8	10	2.9	2.9	2.6	2.4	17
3D	$k = 3$	$k = 4$	$k = 7$	$k = 10$		3D	$k = 3$	$k = 4$	$k = 7$	$k = 10$	
3	2.4	2.5	2.1	1.8	16	3	2.6	2.7	2.4	2.4	35
4	2.4	2.5	2.1	1.9	16	4	2.8	2.8	2.5	2.4	49
5	2.4	2.5	2.1	1.9	16	5	2.8	2.8	2.5	2.4	51
6	2.4	2.5	2.1	1.9	16	6	2.8	2.8	2.5	2.4	52

Based on minimal coloring (left) and graph coloring (right) with a relative solver tolerance of 10^{-8}

5.2 General Geometry

As soon as the mesh cells are not Cartesian anymore, the special structure of separable operators in (6) is lost and the inverse cannot be computed according to (7). In this case, we have two options: solving the local problems iteratively, as in [6], or approximately. A possible approximation which recovers the situation of the previous subsection consists of replacing a non-Cartesian mesh cell by an approximating (hyper-)rectangle, then inverting the separable differential operator on the rectangle (omitting the prefix hyper from here on).

Such a surrogate rectangle can be obtained from the following procedure: first, we compute the arc length of all edges. From these, we obtain the length of the rectangle in each of its natural directions by averaging over all parallel edges (in a topological sense). Thus, the geometry of the rectangle is determined up to its position and orientation in space. Given the fact that the Laplacian is invariant under translation and rotation, these do not matter and we can choose a rectangle centered at the origin with edges parallel to the coordinate directions. Different differential operators may require different approximations here.

The convergence theory of Schwarz methods allows for inexact local solvers as long as they are spectrally equivalent. Naturally, the deviation from exactness enters into the convergence speed of the method. Additionally, inexact local solvers can amplify the solution, such that a smaller relaxation parameter may be necessary. This is exhibited in Table 3, where we compare the efficiency of multigrid with exact local solvers and the method with surrogate rectangles as described above. We see that a reduction of the relaxation parameter $\hat{\omega} = 0.7$ for exact local solvers to $\hat{\omega} = 0.49$ is necessary for robust convergence. We point out though, that while the inexact methods need more iteration steps, they are much faster than exact inverses, since they use the Kronecker representation (7) of the approximate inverse. For instance, the setup cost is 3000 times higher, with a growing gap for higher polynomial degrees.

Table 3 Fractional CG iterations with additive cell-based Schwarz smoothers, exact as well as inexact local solution with varying damping factors $\hat{\omega}$

Levels	Convergence steps to 10^{-8}					
	<i>exact</i> ($\hat{\omega} = 0.7$)	$\hat{\omega} = 0.35$	$\hat{\omega} = 0.42$	$\hat{\omega} = 0.49$	$\hat{\omega} = 0.56$	$\hat{\omega} = 0.63$
2D						
4	17.8	28.4	24.8	24.3	30.8	>100
5	17.3	27.1	23.9	23.8	40.7	>100
6	17.2	26.8	23.7	23.9	58.1	>100
3D						
2	20.6	31.8	28.5	25.8	25.0	28.5
3	20.6	33.3	29.1	26.5	27.4	74.8
4	20.6	32.4	28.6	26.6	47.0	>100

Two pre- and post-smoothing steps are used, respectively, and the polynomial degree is $k = 4$

5.3 Linear Elasticity

In order to provide an outlook on how to apply this concept to more general problems, we consider linear elasticity, namely the Lamé-Navier equations, with the bilinear form

$$a(u, v) = 2\mu(\varepsilon(u), \varepsilon(v)) + \lambda(\nabla \cdot u, \nabla \cdot v). \quad (9)$$

Here, $\varepsilon(u) = \frac{1}{2}(\nabla u + \nabla u^\top)$ is the strain tensor of the displacement field u and (\cdot, \cdot) denote the appropriate DG discretization with interior penalty terms.

Consider a Cartesian vertex patch, that is, a patch with all faces aligned with the coordinate planes and with tensor product shape functions on each cell. As before, let M_k be the one-dimensional mass matrix in direction k and L_k the matrix representing the Laplacian including all face terms introduced by the interior penalty formulation. Furthermore, let G_k be the matrix associated to the first derivative, again including the DG interface terms which arise in products of the form $G_k^\top \otimes G_l$. With these notions and the three-dimensional Laplacian

$$L = M_3 \otimes M_2 \otimes L_1 + M_3 \otimes L_2 \otimes M_1 + L_3 \otimes M_2 \otimes M_1, \quad (10)$$

we can write the bilinear form $a(\cdot, \cdot)$ on the patch in matrix form

$$\begin{aligned} A_p = \mu & \begin{bmatrix} L + M_3 \otimes M_2 \otimes L_1 & M_3 \otimes G_2^\top \otimes G_1 & G_3^\top \otimes M_2 \otimes G_1 \\ M_3 \otimes G_2 \otimes G_1^\top & L + M_3 \otimes L_2 \otimes M_1 & G_3^\top \otimes G_2 \otimes M_1 \\ G_3 \otimes M_2 \otimes G_1^\top & G_3 \otimes G_2^\top \otimes M_1 & L + L_3 \otimes M_2 \otimes M_1 \end{bmatrix} \\ & + \lambda \begin{bmatrix} M_3 \otimes M_2 \otimes L_1 & M_3 \otimes G_2 \otimes G_1^\top & G_3 \otimes M_2 \otimes G_1^\top \\ M_3 \otimes G_2^\top \otimes G_1 & M_3 \otimes L_2 \otimes M_1 & G_3 \otimes G_2^\top \otimes M_1 \\ G_3^\top \otimes M_2 \otimes G_1 & G_3^\top \otimes G_2 \otimes M_1 & L_3 \otimes M_2 \otimes M_1 \end{bmatrix} \end{aligned} \quad (11)$$

Clearly, this matrix lacks the simple structure of Kronecker products we employed in the previous subsections. Nevertheless, we have Korn's inequality [10], and thus the block diagonal of the left matrix is spectrally equivalent to the matrix itself. Consequently, we expect that

$$\tilde{A}_p = \mu \begin{bmatrix} L + M_3 \otimes M_2 \otimes L_1 & & \\ & L + M_3 \otimes L_2 \otimes M_1 & \\ & & L + L_3 \otimes M_2 \otimes M_1 \end{bmatrix}, \quad (12)$$

which has the desired Kronecker product structure, is a good local solver. Indeed, Table 4 confirms this expectation. Iteration counts remain almost constant over a

Table 4 Solver performance depending on level and polynomial degree k

Levels	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$
3	–	–	–	–	–	–	–	4.0	4.1
4	–	–	–	3.7	3.9	3.9	4.0	4.1	4.1
5	–	3.7	3.7	3.7	3.8	3.9	3.9	3.9	3.9
6	5.1	3.7	3.8	3.6	3.8	3.9	3.9	3.9	3.9
7	5.2	3.8	3.9	3.7	3.7	3.8	3.7	3.8	3.8
8	5.5	3.9	3.9	3.8	3.8	3.7	3.8	–	–
9	5.4	3.9	4.0	–	–	–	–	–	–

CG iterations to reduce the residual by 10^8 preconditioned by h -MG with multiplicative vertex patch smoother and approximate local solvers \tilde{A}_p^{-1} . Only levels with 10^4 to 10^7 degrees of freedom are shown. $\mu = 1$, $\lambda = 1$ and the coarse grid consists of 2×2 cells

Table 5 Solver performance depending on Lamé parameters μ and λ

Levels	(μ, λ)					
	(100, 1)	(10, 1)	(1, 1)	(1, 5)	(1, 10)	(1, 25)
6	3.4	3.4	3.6	6.3	19.5	>200
7	3.6	3.6	3.7	6.2	19.9	>200
8	3.7	3.7	3.8	6.0	20.2	>200
9	3.8	3.8	3.9	5.9	20.2	>200
10	3.8	3.8	3.9	5.9	20.3	>200
11	3.8	3.8	3.9	5.8	19.9	>200

CG iterations to reduce the residual by 10^8 preconditioned by h -MG with block-diagonal smoother. Shape functions of degree $k = 4$ are used. The coarse grid consists of 2×2 cells

wide range of mesh levels and polynomial degrees. Comparing to Table 2, we lose less than a factor two, typically requiring 4 steps instead of 3.

While Korn's inequality helped us with the left matrix in (11), the matrix corresponding to the “grad-div” term in the Lamé–Navier equations has a nontrivial kernel and thus its inverse cannot be approximated by a block diagonal. We confirm this in Table 5. After augmenting \tilde{A}_p by the diagonal terms of the grad-div matrix, we vary μ and λ . As expected, iteration counts increase when $\lambda \gg \mu$ to the point, where the method becomes infeasible.

The case $\lambda \gg \mu$ corresponds to an almost incompressible material. Thus, this behavior has to be addressed from two sides. First, the discretization must be suitable [34]. Then, the local solvers must be able to reduce the divergence sufficiently. Here, we have to find ways to implement a smoother like in [39] in an efficient way. Its structure prevents us from utilizing the tensor product techniques, namely the fast diagonalization method, used so far.

As an outlook, we describe a solution approach for two dimensions, which has been developed in a recent bachelor's thesis [64]. The diagonal blocks of the matrix A_p are

$$A_1 = (2\mu + \lambda) M_2 \otimes L_1 + \mu L_2 \otimes M_1, \quad A_2 = \mu M_2 \otimes L_1 + (2\mu + \lambda) L_2 \otimes M_1. \quad (13)$$

Both A_1 and A_2 admit a fast diagonalization, for instance

$$A_2^{-1} = (Q_2 \otimes Q_1)(I_2 \otimes \Lambda_1 + \Lambda_2 \otimes I_1)^{-1}(Q_2 \otimes Q_1)^T. \quad (14)$$

Given the off-diagonal block $B = \mu G_2^T \otimes G_1 + \lambda G_2 \otimes G_1^T$, the Schur complement of A_p is

$$S = A_1 - B^T A_2^{-1} B. \quad (15)$$

While this is not a sum of Kronecker products, Kronecker singular value decomposition (KSVD), see [72, 73], can be utilized to construct an approximation of the Schur complement which is fast diagonalizable. We proceed as follows:

- A.1** compute the fast diagonalizations of A_1 and A_2
- A.2** compute the rank- ρ_Λ KSVD of the inverse diagonal matrix in Eq. 14

$$(I \otimes \Lambda^{(1)} + \Lambda^{(2)} \otimes I)^{-1} \approx \sum_{i=1}^{\rho_\Lambda} C_i \otimes D_i \quad (16)$$

- A.3** compute the rank-2 KSVD

$$\widehat{S} := E_1 \otimes F_1 + E_2 \otimes F_2 \approx \widetilde{S} \quad (17)$$

of the approximate Schur complement

$$\widetilde{S} := A_1 - B^T \left[\sum_{i=1}^{\rho_\Lambda} Q_2 C_i^{-1} Q_2^T \otimes Q_1 D_i^{-1} Q_1^T \right] B \quad (18)$$

- A.4** compute the fast diagonalization of \widehat{S} .

Then, Gaussian block elimination provides an approximate inverse

$$A_p^{-1} \approx \begin{bmatrix} I & -A_1^{-1} B \\ 0 & I \end{bmatrix} \begin{bmatrix} A_1^{-1} & 0 \\ 0 & \widehat{S}^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -B^T A_1^{-1} & I \end{bmatrix}. \quad (19)$$

Implementation and evaluation of these smoothers are still work in progress, but the thesis [64] suggests fast and robust convergence at least in a finite difference context.

The take-home message from this section is that an efficient approximate solution of the local problems in Schwarz smoothers is possible using low-rank tensor representations and can be achieved with effort similar to a matrix-free operator application in the best case. Finding such low-rank representations is nevertheless highly dependent on the differential equation and geometry. Further investigation will be directed in particular at dealing with the grad-div operator.

6 High-performance Simulations of Incompressible Flows

Computational fluid dynamics (CFD) simulations of turbulent flows at large Reynolds number, e.g., $\text{Re} > 10^6$, are among those problems that typically require a huge amount of computational resources in order to resolve the turbulent flow structures in space and time, and have been addressed as an application by the ExaDG project. The underlying model problem is given by the incompressible Navier–Stokes equations

$$\frac{\partial u}{\partial t} + \nabla \cdot (u \otimes u) - \nu \nabla^2 u + \nabla p = f , \quad (20)$$

$$\nabla \cdot u = 0 . \quad (21)$$

Scale-resolving simulations for engineering applications typically involve beyond $\mathcal{O}(10^{10} - 10^{11})$ unknowns (DoFs) and $\mathcal{O}(10^5 - 10^7)$ time steps. High-performance implementations for this type of problem are therefore of paramount importance for the CFD community. It is important to stress that implementing a given algorithm optimally for a given hardware, i.e., an implementation that performs close to the hardware limits, is only one step to achieve the goal of providing efficient flow solvers for engineering problems as emphasized in the introduction. While the previous sections discussed the second and third term in Eq. (1), namely the performance of matrix-free evaluation routines and fast multigrid solvers for high-order discretizations, we now also include discretization aspects into the discussion. The implementation makes use of the fast matrix-free evaluation routines and multigrid solvers discussed in previous sections.

We use a method of lines approach with high-order DG discretizations in space and splitting methods with BDF time integration. Splitting methods separate the solution of the incompressible Navier–Stokes equations into sub-problems such as a Poisson equation for the pressure and a (convection–)diffusion equation for the velocity and are among the most efficient solvers currently known. In a first contribution [45], we highlighted that previous discretization methods lack robustness, on the one hand in the limit of small time step sizes, and on the other

hand in under-resolved scenarios where the spatial discretization only resolves the largest scales of the flow. The stability problem for small time step sizes has been addressed in detail in [21] where we found that a proper DG discretization of velocity-pressure coupling terms is essential to achieve robustness at small time steps. In [23], we presented the first high-order DG incompressible flow solver that is robust in the under-resolved regime and that relies completely on efficient matrix-free evaluation routines. The developed discretization approach is attractive as it provides a generic solver for turbulent flow simulations that is robust and accurate without the use of explicit turbulence models. Such a technique is known as implicit large-eddy simulation in the literature and has the advantage that it does not require turbulence model parameters. While this property of high-order DG discretizations is already known from discontinuous Galerkin discretizations of the compressible Navier–Stokes equations, the work [23] has been the first demonstrating this property for DG discretizations of the incompressible Navier–Stokes equations. The key ingredient for a robust high-order, L_2 -conforming DG discretization for incompressible flows turns out to be the use of consistent stabilization terms that enforce the divergence-free constraint and inter-element mass conservation in a weak sense. These requirements can also be included into the finite element function spaces by using so-called $H(\text{div})$ -conforming (normal-continuous) discretizations that are exactly (pointwise) divergence-free by using Raviart–Thomas elements. As investigated in detail in [26], such an approach has indeed very similar discretization properties when compared with the stabilized L_2 -conforming approach in practically relevant, under-resolved application scenarios. The model has been extended to moving meshes in [20].

A detailed performance analysis has been undertaken in [22] where we discuss the incompressible flow solver w.r.t. its efficiency according to Eq.(1). Based on this efficiency model, we have then compared matrix-free solvers based on incompressible and compressible Navier–Stokes formulations in [24] for under-resolved turbulent incompressible flows. The compressible solver uses explicit time integration and therefore only requires one operator evaluation in every Runge–Kutta stage as opposed to the incompressible solver involving the solution of linear system of equations such as a pressure Poisson equation within every time step. Simple explicit solvers are often considered efficient due to better parallel scalability since implicit Krylov solvers with multigrid preconditioning involve global communication. However, our work shows a significant performance advantage of the incompressible formulation over the compressible one on the node-level for sufficient workload. Albeit speed-up factors are higher, it is difficult to achieve a performance advantage for the algorithmically simple, explicit-in-time compressible solver in the strong-scaling limit in terms of absolute run time. In our experience, the potential to outperform an implicit solver at some point in the strong-scaling limit has not materialized. We see it as a future challenge to devise optimal PDE solvers providing good performance over a wide range of problems and hardware platforms due to this high degree of interdisciplinarity.

We have applied this solver framework to conduct direct numerical simulations of turbulent channel flow in [45], the first direct numerical simulation of the turbulent

flow over a periodic hill at $\text{Re} \approx 10^4$ in [46], and to large-eddy simulation of the FDA benchmark nozzle problem in [25]. Furthermore, we have developed multiscale wall modeling approaches that allow to use the proposed highly efficient schemes also for industrial cases with even higher Reynolds numbers than what is feasible for wall-resolved large eddy simulation [47].

Here, we show performance results obtained on SuperMUC-NG with Intel Skylake CPUs. We study the three-dimensional Taylor–Green vortex problem as a standard benchmark to assess the accuracy and computational efficiency of incompressible turbulent flow solvers. Regarding discretization accuracy and from a physical point of view, the quantity of interest is the kinetic energy dissipation rate shown in Fig. 12 as a function of time $0 \leq t \leq T = 20$ for increasing Reynolds numbers $\text{Re} = 100, 200, 400, 800, 1600, 3000, 10,000, \infty$. The first direct numerical simulation for the $\text{Re} = 1600$ case with a high-order DG scheme of the incompressible Navier–Stokes equations with a resolution of 1024^3 and polynomial degrees $k = 3, 7$ has been shown in [22]. Here, we show results for effective resolutions up to 3072^3 (corresponding to $0.99 \cdot 10^{11}$ DoFs) for the highest Reynolds number cases. Despite these fine resolutions, grid-converged results are achieved only up to $\text{Re} = 3000$. The inviscid problem ($\text{Re} = \infty$) is most challenging, and the results in Fig. 12 suggest that even finer resolutions are required for grid-convergence, a goal that might be achievable in the foreseeable future. The largest problem with $0.99 \cdot 10^{11}$ DoFs involved $6.6 \cdot 10^4$ time steps and required 11.4 h of wall time on 152,064 cores. In terms of degrees of freedom solved per time step per core, this results in a throughput of 1.05 MDoFs/s/core.

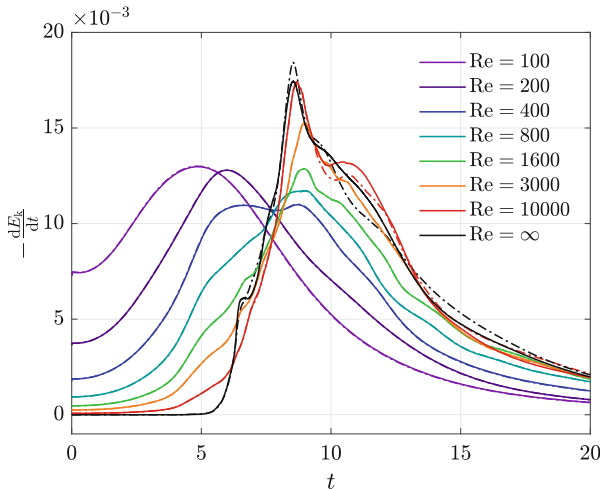


Fig. 12 Taylor–Green vortex: Kinetic energy dissipation rates for two different problem sizes (fine mesh as solid line and coarse mesh as dashed-dotted line) for each Re number: The polynomial degree is $k = 3$ and the effective resolutions $N_{\text{eff}} = (N_{\text{ele,1d}}(k + 1))^3$ considered are $N_{\text{eff}} = 64^3, 128^3$ for $\text{Re} = 100$, $N_{\text{eff}} = 128^3, 256^3$ for $\text{Re} = 200, 400$, $N_{\text{eff}} = 256^3, 512^3$ for $\text{Re} = 800$, $N_{\text{eff}} = 1024^3, 2048^3$ for $\text{Re} = 1600$, and $N_{\text{eff}} = 2048^3, 3072^3$ for $\text{Re} = 3000, 10000, \infty$

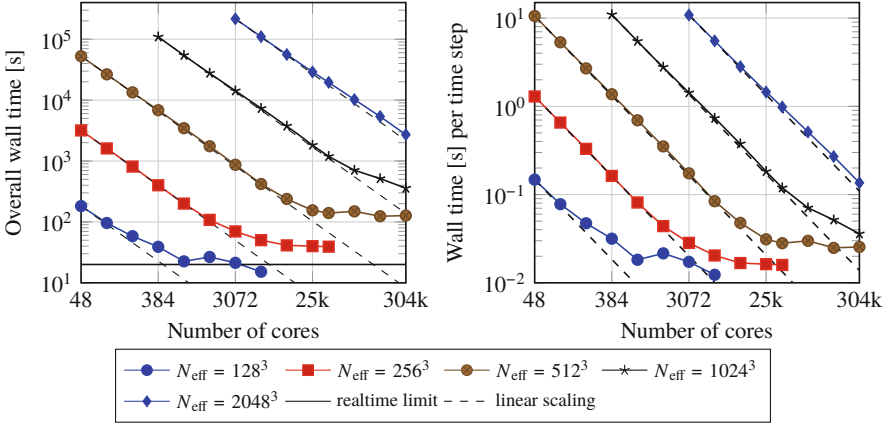


Fig. 13 Scaling analysis for incompressible flow solver on 3D Taylor–Green vortex with polynomial degree $k = 3$ at $\text{Re} = 1600$ and spatial resolutions of 128^3 , 256^3 , 512^3 , 1024^3 , 2048^3

Figure 13 shows strong scaling results for the TGV problem at $\text{Re} = 1600$ for effective resolutions of 128^3 , 256^3 , 1024^3 , 2048^3 and polynomial degree $k = 3$. We assess strong scalability in terms of absolute run times for the whole application (including mesh-generation, setup of data structures, solvers, preconditioners, and postprocessing) rather than normalized speed-up factors as the aim of strong scalability is not only reducing but also minimizing time-to-solution, i.e., demonstrating strong-scalability of a code with poor serial performance is meaningless. The results in Fig. 13 reveal that we are able to perform the TGV simulations in realtime ($t_{\text{wall}} \leq T = 20\text{s}$) for spatial resolutions up to 128^3 . These numbers can be considered outstanding and we are not aware of other high-order DG solvers achieving this performance, see also the discussions in [22, 24]. The minimum wall time in the strong-scaling limit increases on finer meshes due to more time steps (the time step size is restricted according to the CFL condition, $\Delta t \sim 1/h$, for the mixed explicit–implicit splitting solver used here). For this reason, we also show strong scalability in terms of the wall time per time step, to allow extrapolations of how many time steps can be solved within a given wall time limit which is the typical use case for large-eddy and direct numerical simulations of turbulent flows. In this metric, the curves level off around 0.02 – 0.03 s of wall time per time step, independently of the spatial resolution. The SuperMUC-NG machine with $3 \cdot 10^5$ cores is too small to show the strong scaling limit for the largest problem size with 2048^3 resolution considered here. A parallel efficiency of 80.6% is achieved with a speed-up factor of 79.8 when scaling from 3072 cores to 304,128 cores.

7 hyper.deal: Extending the Matrix-Free Kernels to Higher Dimensions

The matrix-free kernels developed within the ExaDG project have been implemented in a recursive manner which enables compilation with arbitrary spatial dimension. In order to be compatible with the mesh infrastructure of deal.II which is restricted to dimensions up to 3, we have developed schemes working on a tensor product of two deal.II meshes. This allows extension to 2+2, 2+3, and 3+3 dimensions. The corresponding framework is currently under development as the deal.II-extension `hyper.deal` [59].

The major application that we have in mind are kinetic problems in phase space where we use the tensor product of a spatial and a velocity mesh. However, other applications might arise such as parameter-dependent flow problems. Table 6 gives an overview of computational times on a six-dimensional Vlasov–Poisson problem, which involves an advection in the 6D space of the particle density in x and v space and the solution of a 3D Poisson equation for finding the electric potential that in turn specifies the electric field that transports the density field (cf. [44] for the same application tackled with a semi-Lagrangian solver).

Figure 14 lists the throughput of the matrix-free evaluation of cell integrals for the multi-dimensional advection in three to six spatial dimensions for polynomial degrees $k = 2, 3, 4, 5$ for AVX2 and AVX-512 vectorization over elements, respectively, without any application-specific tuning at this stage. While throughput is very good in 3D and 4D as well as 5D up to $k = 4$, performance drops significantly in 6D because the local arrays in sum factorization exhaust caches,

Table 6 Contributions to run time on 6D Vlasov–Poisson system on 320 cores with 8.6 billion spatial DoFs over 42 time steps

Category	6D advect total (of which MPI exchange)	integrate v	3D Poisson + electric field
time [s]	560 (130)	13.0	35.9

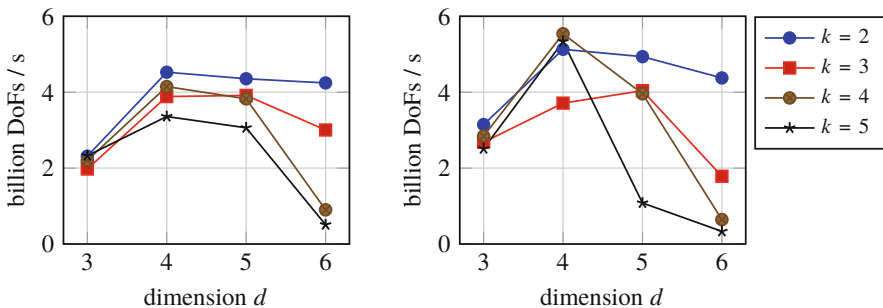


Fig. 14 Throughput of cell term for advection as a function of the spatial dimension on Intel Skylake with 4-wide vectorization (left) and 8-wide vectorization (right)

especially with AVX-512. Vectorization strategies within an element [50] are currently under development.

8 Outlook

Our work in the ExaDG project presented in this text has resulted in a highly competitive finite element framework. We have demonstrated excellent performance both for the pure operator evaluation, demonstrated e.g. by the CEED benchmark problems, as well as on an application level in computational fluid dynamics. We plan to engage in benchmarking also in the future to establish best-practices for the high-order finite element community. Furthermore, the evolving hardware landscape requires a continued effort, with increasing pressure to additional performance improvements on throughput architectures such as GPUs and FPGAs. In addition, we plan to extend our hybrid *hp*-multigrid framework to also handle *hp*-adaptive meshes. Finally, while the results from the Schwarz-based multigrid smoothers are very promising from a mathematical point of view, further steps are necessary to make them perform optimally on massively parallel hardware, and it is not yet clear how an optimal implementation compares in time-to-solution against the simpler Chebyshev-based ingredients we have considered on the large scale so far.

References

1. Alzetta, G., Arndt, D., Bangerth, W., Boddu, V., Brands, B., Davydov, D., Gassmoeller, R., Heister, T., Heltai, L., Kormann, K., Kronbichler, M., Maier, M., Pelteret, J.P., Turcksin, B., Wells, D.: The deal.II library, version 9.0. *J. Numer. Math.* **26**(4), 173–184 (2018). <https://doi.org/10.1515/jnma-2018-0054>
2. Anderson, R., Barker, A., Bramwell, J., Cerveny, J., Dahm, J., Dobrev, V., Dudouit, Y., Fisher, A., Kolev, T., Stowell, M., Tomov, V.: MFEM: modular finite element methods (2019). mfem.org
3. Antonietti, P.F., Sarti, M., Verani, M., Zikatanov, L.T.: A uniform additive Schwarz preconditioner for high-order discontinuous Galerkin approximations of elliptic problems. *J. Sci. Comput.* **70**(2), 608–630 (2017). <https://doi.org/10.1007/s10915-016-0259-9>
4. Arndt, D., Bangerth, W., Davydov, D., Heister, T., Heltai, L., Kronbichler, M., Maier, M., Pelteret, J.-P., Turcksin, B., Wells, D.: The deal.II finite element library: Design, features, and insights. *Comput. Math. Appl.* (2020). <https://doi.org/10.1016/j.camwa.2020.02.022>
5. Bastian, P., Engwer, C., Fahlke, J., Geveler, M., Göddeke, D., Iliev, O., Ippisch, O., Milk, R., Möring, J., Müthing, S., Ohlberger, M., Ribbrock, D., Turek, S.: Hardware-based efficiency advances in the EXA-DUNE project. In: Bungartz, H.J., Neumann, P., Nagel, W.E. (eds.) *Software for Exascale computing—SPPEXA 2013-2015*, pp. 3–23. Springer, Cham (2016)
6. Bastian, P., Müller, E.H., Müthing, S., Piatkowski, M.: Matrix-free multigrid block-preconditioners for higher order discontinuous Galerkin discretisations. *J. Comput. Phys.* **394**, 417–439 (2019). <https://doi.org/10.1016/j.jcp.2019.06.001>
7. Bauer, S., Drzisga, D., Mohr, M., Rude, U., Waluga, C., Wohlmuth, B.: A stencil scaling approach for accelerating matrix-free finite element implementations. *SIAM J. Sci. Comput.* **40**(6), C748–C778 (2018)

8. Bergen, B., Hülsemann, F., Rüde, U.: Is 1.7×10^{10} unknowns the largest finite element system that can be solved today? In: Proceeding of ACM/IEEE Conference Supercomputing (SC'05), pp. 5:1–5:14 (2005). <https://doi.org/10.1109/SC.2005.38>
9. Brandt, A.: Multi-level adaptive solutions to boundary-value problems. *Math. Comput.* **31**, 333–390 (1977). <https://doi.org/10.1090/S0025-5718-1977-0431719-X>
10. Brenner, S.C.: Korn's inequalities for piecewise H^1 vector fields. *Math. Comput.* **73**(247), 1067–1087 (2004)
11. Brown, J.: Efficient nonlinear solvers for nodal high-order finite elements in 3D. *J. Sci. Comput.* **45**(1–3), 48–63 (2010)
12. Cantwell, C.D., Sherwin, S.J., Kirby, R.M., Kelly, P.H.J.: Form h to p efficiently: Selecting the optimal spectral/ hp discretisation in three dimensions. *Math. Model. Nat. Phenom.* **6**, 84–96 (2011)
13. Cantwell, C.D., Moxey, D., Comerford, A., Bolis, A., Rocco, G., Mengaldo, G., De Grazia, D., Yakovlev, S., Lombard, J.E., Ekelschot, D., Jordi, B., Xu, H., Mohamied, Y., Eskilsson, C., Nelson, B., Vos, P., Biotto, C., Kirby, R.M., Sherwin, S.J.: Nektar++: An open-source spectral/ hp element framework. *Comput. Phys. Comm.* **192**, 205–219 (2015). <https://doi.org/10.1016/j.cpc.2015.02.008>
14. Charrier, D.E., Hazelwood, B., Tutlyaeva, E., Bader, M., Dumbser, M., Kudryavtsev, A., Moskovsky, A., Weinzierl, T.: Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver. *Int. J. High Perf. Comput. Appl.* **33**(5), 973–986 (2019). <https://doi.org/10.1177/1094342019842645>
15. Clevenger, T.C., Heister, T., Kanschä, G., Kronbichler, M.: A flexible, parallel, adaptive geometric multigrid method for FEM. Technical report, arXiv:1904.03317 (2019)
16. Davydov, D., Kronbichler, M.: Algorithms and data structures for matrix-free finite element operators with MPI-parallel sparse multi-vectors. *ACM Trans. Parallel Comput.* (2020). <https://doi.org/10.1145/3399736>
17. Davydov, D., Heister, T., Kronbichler, M., Steinmann, P.: Matrix-free locally adaptive finite element solution of density-functional theory with nonorthogonal orbitals and multigrid preconditioning. *Phys. Status Solidi B: Basic Solid State Phys.* **255**(9), 1800069 (2018). <https://doi.org/10.1002/pssb.201800069>
18. Davydov, D., Pelteret, J.P., Arndt, D., Kronbichler, M., Steinmann, P.: A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid. *Int. J. Numer. Meth. Eng.* (2020). <https://doi.org/10.1002/nme.6336>
19. Deville, M.O., Fischer, P.F., Mund, E.H.: High-order Methods for Incompressible Fluid Flow, vol. 9. Cambridge University, Cambridge (2002)
20. Fehn, N., Heinz, J., Wall, W.A., Kronbichler, M.: High-order arbitrary Lagrangian-Eulerian discontinuous Galerkin methods for the incompressible Navier-Stokes equations. Technical report, arXiv:2003.07166 (2020).
21. Fehn, N., Wall, W.A., Kronbichler, M.: On the stability of projection methods for the incompressible Navier–Stokes equations based on high-order discontinuous Galerkin discretizations. *J. Comput. Phys.* **351**, 392–421 (2017). <https://doi.org/10.1016/j.jcp.2017.09.031>
22. Fehn, N., Wall, W.A., Kronbichler, M.: Efficiency of high-performance discontinuous Galerkin spectral element methods for under-resolved turbulent incompressible flows. *Int. J. Numer. Meth. Fluids* **88**(1), 32–54 (2018). <https://doi.org/10.1002/flid.4511>
23. Fehn, N., Wall, W.A., Kronbichler, M.: Robust and efficient discontinuous Galerkin methods for under-resolved turbulent incompressible flows. *J. Comput. Phys.* **372**, 667–693 (2018). <https://doi.org/10.1016/j.jcp.2018.06.037>
24. Fehn, N., Wall, W.A., Kronbichler, M.: A matrix-free high-order discontinuous Galerkin compressible Navier–Stokes solver: a performance comparison of compressible and incompressible formulations for turbulent incompressible flows. *Int. J. Numer. Meth. Fluids* **89**(3), 71–102 (2019). <https://doi.org/10.1002/flid.4683>

25. Fehn, N., Wall, W.A., Kronbichler, M.: Modern discontinuous Galerkin methods for the simulation of transitional and turbulent flows in biomedical engineering: a comprehensive LES study of the FDA benchmark nozzle model. *Int. J. Numer. Meth. Biomed. Eng.* **35**(12), e3228 (2019). <https://doi.org/10.1002/cnm.3228>
26. Fehn, N., Kronbichler, M., Lehrenfeld, C., Lube, G., Schroeder, P.W.: High-order DG solvers for under-resolved turbulent incompressible flows: a comparison of L^2 and $H(\text{div})$ methods. *Int. J. Numer. Meth. Fluids* **91**(11), 533–556 (2019). <https://doi.org/10.1002/flid.4763>
27. Fehn, N., Munch, P., Wall, W.A., Kronbichler, M.: Hybrid multigrid methods for high-order discontinuous Galerkin discretizations. *J. Comput. Phys.* (2020). <https://doi.org/10.1016/j.jcp.2020.109538>
28. Fischer, P., Kerkemeier, S., Peplinski, A., Shaver, D., Tomboulides, A., Min, M., Obabko, A., Merzari, E.: Nek5000 Web page (2019). <https://nek5000.mcs.anl.gov>
29. Fischer, P., Min, M., Rathnayake, T., Dutta, S., Kolev, T., Dobrev, V., Camier, J.S., Kronbichler, M., Warburton, T., Świrydowicz, K., Brown, J.: Scalability of high-performance PDE solvers. *Int. J. High Perf. Comput. Appl.* (2020). <https://doi.org/10.1177/1094342020915762>
30. Gholami, A., Malhotra, D., Sundar, H., Biros, G.: FFT, FMM, or multigrid? A comparative study of state-of-the-art Poisson solvers for uniform and nonuniform grids in the unit cube. *SIAM J. Sci. Comput.* **38**(3), C280–C306 (2016). <https://doi.org/10.1137/15M1010798>
31. Gmeiner, B., Rüde, U., Stengel, H., Waluga, C., Wohlmuth, B.: Towards textbook efficiency for parallel multigrid. *Numer. Math.-Theory Me. Appl.* **8**(1), 22–46 (2015)
32. Gmeiner, B., Huber, M., John, L., Rüde, U., Wohlmuth, B.: A quantitative performance study for Stokes solvers at the extreme scale. *J. Comput. Sci.* **17**, 509–521 (2016). <https://doi.org/10.1016/j.jocs.2016.06.006>. <http://www.sciencedirect.com/science/article/pii/S187750316301077>. Recent Advances in Parallel Techniques for Scientific Computing
33. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Boca Raton (2011)
34. Hansbo, P., Larson, M.G.: Discontinuous Galerkin methods for incompressible and nearly incompressible elasticity by Nitsche’s method. *Comput. Methods Appl. Mech. Eng.* **191**, 1895–1908 (2002)
35. Ibeid, H., Olson, L., Gropp, W.: FFT, FMM, and multigrid on the road to exascale: performance challenges and opportunities. *J. Parallel Distrib. Comput.* **136**, 63–74 (2020). <https://doi.org/10.1016/j.jpdc.2019.09.014>
36. Janssen, B., Kanschat, G.: Adaptive multilevel methods with local smoothing for H^1 - and H^{curl} -conforming high order finite element methods. *SIAM J. Sci. Comput.* **33**(4), 2095–2114 (2011). <https://doi.org/10.1137/090778523>
37. Kanschat, G.: Multi-level methods for discontinuous Galerkin FEM on locally refined meshes. *Comput. Struct.* **82**(28), 2437–2445 (2004). <https://doi.org/10.1016/j.compstruc.2004.04.015>
38. Kanschat, G.: Robust smoothers for high order discontinuous Galerkin discretizations of advection-diffusion problems. *J. Comput. Appl. Math.* **218**, 53–60 (2008). <https://doi.org/10.1016/j.cam.2007.04.032>
39. Kanschat, G., Mao, Y.: Multigrid methods for \mathbf{H}^{div} -conforming discontinuous Galerkin methods for the Stokes equations. *J. Numer. Math.* **23**(1), 51–66 (2015). <https://doi.org/10.1515/jnma-2015-0005>
40. Kempf, D., Hess, R., Müthing, S., Bastian, P.: Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures. Technical report, arXiv:1812.08075 (2018)
41. Knepley, M.G., Brown, J., Rupp, K., Smith, B.F.: Achieving high performance with unified residual evaluation. Technical report, arXiv:1309.1204 (2013)
42. Kormann, K.: A time-space adaptive method for the Schrödinger equation. *Commun. Comput. Phys.* **20**(1), 60–85 (2016). <https://doi.org/10.4208/cicp.101214.021015a>

43. Kormann, K., Kronbichler, M.: Parallel finite element operator application: graph partitioning and coloring. In: Proceeding of 7th IEEE International Conference eScience, pp. 332–339 (2011). <https://10.1109/eScience.2011.53>
44. Kormann, K., Reuter, K., Rampp, M.: A massively parallel semi-Lagrangian solver for the six-dimensional Vlasov–Poisson equation. *Int. J. High Perform. Comput. Appl.* **33**(5), 924–947 (2019). <https://doi.org/10.1177/1094342019834644>
45. Krank, B., Fehn, N., Wall, W.A., Kronbichler, M.: A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. *J. Comput. Phys.* **348**, 634–659 (2017). <https://doi.org/10.1016/j.jcp.2017.07.039>
46. Krank, B., Kronbichler, M., Wall, W.A.: Direct numerical simulation of flow over periodic hills up to $Re_h = 10,595$. *Flow Turbulence Combust.* **101**, 521–551 (2018). <https://doi.org/10.1007/s10494-018-9941-3>
47. Krank, B., Kronbichler, M., Wall, W.A.: A multiscale approach to hybrid RANS/LES wall modeling within a high-order discontinuous Galerkin scheme using function enrichment. *Int. J. Numer. Meth. Fluids* **90**, 81–113 (2019). <https://doi.org/10.1002/fld.4712>
48. Kronbichler, M., Allalen, M.: Efficient high-order discontinuous Galerkin finite elements with matrix-free implementations. In: Bungartz, H.J., Kranzlmüller, D., Weinberg, V., Weismüller, J., Wohlgemuth, V. (eds.) *Advances and New Trends in Environmental Informatics*, pp. 89–110. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-99654-7_7
49. Kronbichler, M., Kormann, K.: A generic interface for parallel cell-based finite element operator application. *Comput. Fluids* **63**, 135–147 (2012). <https://doi.org/10.1016/j.compfluid.2012.04.012>
50. Kronbichler, M., Kormann, K.: Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM Trans. Math. Softw.* **45**(3), 29:1–29:40 (2019). <https://doi.org/10.1145/3325864>
51. Kronbichler, M., Ljungkvist, K.: Multigrid for matrix-free high-order finite element computations on graphics processors. *ACM Trans. Parallel Comput.* **6**(1), 2:1–2:32 (2019). <https://doi.org/10.1145/3322813>
52. Kronbichler, M., Wall, W.A.: A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers. *SIAM J. Sci. Comput.* **40**(5), A3423–A3448 (2018). <https://doi.org/10.1137/16M110455X>
53. Kronbichler, M., Schoeder, S., Müller, C., Wall, W.A.: Comparison of implicit and explicit hybridizable discontinuous Galerkin methods for the acoustic wave equation. *Int. J. Numer. Meth. Eng.* **106**(9), 712–739 (2016). <https://doi.org/10.1002/nme.5137>
54. Kronbichler, M., Kormann, K., Pasichnyk, I., Allalen, M.: Fast matrix-free discontinuous Galerkin kernels on modern computer architectures. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D.E. (eds.) *ISC High Performance 2017*, LNCS 10266, pp. 237–255 (2017). <https://doi.org/10.1007/978-3-319-58667-013>
55. Kronbichler, M., Diagne, A., Holmgren, H.: A fast massively parallel two-phase flow solver for microfluidic chip simulation. *Int. J. High Perf. Comput. Appl.* **32**(2), 266–287 (2018). <https://doi.org/10.1177/1094342016671790>
56. Kronbichler, M., Kormann, K., Fehn, N., Munch, P., Witte, J.: A Hermite-like basis for faster matrix-free evaluation of interior penalty discontinuous Galerkin operators. Technical report, arXiv:1907.08492 (2019)
57. Ljungkvist, K.: Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes. In: *Proceedings of the 25th High Performance Computing Symposium, HPC '17*, pp. 1:1–1:12. Society for Computer Simulation International, San Diego (2017). <http://dl.acm.org/citation.cfm?id=3108096.3108097>
58. Lynch, R.E., Rice, J.R., Thomas, D.H.: Direct solution of partial difference equations by tensor product methods. *Numer. Math.* **6**, 185–199 (1964). <https://doi.org/10.1007/BF01386067>
59. Munch, P., Kormann, K., Kronbichler, M.: hyper.deal: An efficient, matrix-free finite-element library for high-dimensional partial differential equations. Technical report, arXiv:2002.08110 (2020)

60. Müthing, S., Piatkowski, M., Bastian, P.: High-performance implementation of matrix-free high-order discontinuous Galerkin methods. Technical report, arXiv:1711.10885 (2017)
61. Orszag, S.A.: Spectral methods for problems in complex geometries. *J. Comput. Phys.* **37**, 70–92 (1980)
62. Raffenetti, K., Amer, A., Oden, L., Archer, C., Bland, W., Fujita, H., Guo, Y., Janjusic, T., Durnov, D., Blocksome, M., Si, M., Seo, S., Langer, A., Zheng, G., Takagi, M., Coffman, P., Jose, J., Sur, S., Sannikov, A., Oblomov, S., Chuvelev, M., Hatanaka, M., Zhao, X., Fischer, P., Rathnayake, T., Otten, M., Min, M., Balaji, P.: Why is MPI so slow?: Analyzing the fundamental limits in implementing MPI-3.1. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, pp. 62:1–62:12. ACM, New York (2017). <https://doi.org/10.1145/3126908.3126963>
63. Rathgeber, F., Ham, D.A., Mitchell, L., Lange, M., Luporini, F., McRae, A.T.T., Bercea, G.T., Markall, G.R., Kelly, P.H.J.: Firedrake: automating the finite element method by composing abstractions. *ACM Trans. Math. Soft.* **43**(3), 24:1–24:27 (2017). <https://doi.org/10.1145/2998441>
64. Schmidt, S.: Fast, tensor-based solution of problems involving incompressibility, Bachelor thesis. Heidelberg University, Heidelberg (2019)
65. Schoeder, S., Kormann, K., Wall, W.A., Kronbichler, M.: Efficient explicit time stepping of high order discontinuous Galerkin schemes for waves. *SIAM J. Sci. Comput.* **40**(6), C803–C826 (2018). <https://doi.org/10.1137/18M1185399>
66. Schoeder, S., Kronbichler, M., Wall, W.: Arbitrary high-order explicit hybridizable discontinuous Galerkin methods for the acoustic wave equation. *J. Sci. Comput.* **76**, 969–1006 (2018). <https://doi.org/10.1007/s10915-018-0649-2>
67. Schoeder, S., Sticko, S., Kreiss, G., Kronbichler, M.: High-order cut discontinuous Galerkin methods with local time stepping for acoustics. *Int. J. Numer. Meth. Eng.* (2020). <https://doi.org/10.1002/nme.6343>
68. Schoeder, S., Wall, W.A., Kronbichler, M.: ExWave: A high performance discontinuous Galerkin solver for the acoustic wave equation. *Soft. X* **9**, 49–54 (2019). <https://doi.org/10.1016/j.softx.2019.01.001>
69. Solomonoff, A.: A fast algorithm for spectral differentiation. *J. Comput. Phys.* **98**(1), 174–177 (1992). [https://doi.org/10.1016/0021-9991\(92\)90182-X](https://doi.org/10.1016/0021-9991(92)90182-X)
70. Sundar, H., Biros, G., Burstedde, C., Rudi, J., Ghattas, O., Stadler, G.: Parallel geometric-algebraic multigrid on unstructured forests of octrees. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 43. IEEE Computer Society, Silver Spring (2012)
71. Świrydowicz, K., Chalmers, N., Karakus, A., Warburton, T.: Acceleration of tensor-product operations for high-order finite element methods. *Int. J. High Perf. Comput. Appl.* **33**(4), 735–757 (2019). <https://doi.org/10.1177/1094342018816368>
72. Van Loan, C.F.: The ubiquitous Kronecker product. *J. Comput. Appl. Math.* **123**(1–2), 85–100 (2000)
73. Van Loan, C.F., Pitsianis, N.: Approximation with Kronecker products. In: *Linear Algebra for Large Scale and Real-time Applications*, pp. 293–314. Springer, Berlin (1993)
74. Varga, R.S.: *Matrix Iterative Analysis*, 2nd edn. Springer, Berlin (2009)
75. Wichmann, K.R., Kronbichler, M., Löhner, R., Wall, W.A.: Practical applicability of optimizations and performance models to complex stencil-based loop kernels in CFD. *Int. J. High Perf. Comput. Appl.* **33**(4), 602–618 (2019). <https://doi.org/10.1177/1094342018774126>
76. Witte, J., Arndt, D., Kanschä, G.: Fast tensor product Schwarz smoothers for high-order discontinuous Galerkin methods. Technical report, arXiv:1910.11239 (2019)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

