

Matrix-free finite-element computations at extreme scale and for challenging applications

*Kumulative Dissertation
zur Erlangung des akademischen Grades*

Dr. rer. nat.

*eingereicht an der
Mathematisch-Naturwissenschaftlich-Technischen Fakultät
der Universität Augsburg*

von

Peter Münch

Augsburg, August 2023



Erstgutachter: Prof. Dr. Martin Kronbichler

Zweitgutachter: Prof. Dr. Daniel Peterseim

Drittgutachter: Prof. Jed Brown, Ph.D.

Tag der mündlichen Prüfung: 14.11.2023

Abstract

For numerical computations based on finite element methods (FEM), it is common practice to assemble the system matrix related to the discretized system and to pass this matrix to an iterative solver. However, the assembly step can be costly and the matrix might become locally dense, e.g., in the context of high-order, high-dimensional, or strongly coupled multicomponent FEM, leading to high costs when applying the matrix due to limited bandwidth on modern CPU- and GPU-based hardware. Matrix-free algorithms are a means of accelerating FEM computations on HPC systems, by applying the effect of the system matrix without assembling it. Despite convincing arguments for matrix-free computations as a means of improving performance, their usage still tends to be an exception at the time of writing of this thesis, not least because they have not yet proven their applicability in all areas of computational science, e.g., solid mechanics.

In this thesis, we further develop a state-of-the-art matrix-free framework for high-order FEM computations with focus on the preconditioning and adopt it in novel application fields. In the context of high-order FEM, we develop means of improving cache efficiency by interleaving cell loops with vector updates, which we use to increase the throughput of preconditioned conjugate gradient methods and of block smoothers based on additive Schwarz methods; we also propose an algorithm for the fast application of hanging-node constraints in 3D for up to 137 refinement configurations. We develop efficient geometric and polynomial multigrid solvers with optimized transfer operators, whose performance is experimentally investigated in detail in the context of locally refined meshes, indicating the superiority of global-coarsening algorithms. We apply the developed solvers in the context of novel stage-parallel implicit Runge–Kutta methods and demonstrate the benefit of stage-parallel solvers in decreasing the time to solution at the scaling limit. Novel challenging application fields of matrix-free computations include high-dimensional computational plasma physics, solid-state-sintering simulations with a high and dynamically changing number of strongly coupled components, and coupled multiphysics problems with evaluation and integration at arbitrary points. In the context of these fields, we detail computational challenges, propose modified versions of the standard matrix-free algorithms for high-performance computing, and discuss preconditioning-related topics.

The efficiency of the derived algorithms on the node level and at extreme scales is demonstrated experimentally on SuperMUC-NG, one of Germany's leading supercomputers, with up to 150k processes and by solving systems of up to 5×10^{12} unknowns. Such problem sizes would not be conceivable for equivalent matrix-based algorithms. The major achievements of this thesis allow to run larger simulations faster and more efficiently, enabling progress and new possibilities for a range of application fields in computational science.

Acknowledgments

This PhD thesis has been written at several places. I started at the Institute for Computational Mechanics at the Technical University of Munich before I switched to the Helmholtz-Zentrum Geesthacht (HZG). During my time there, I worked, Covid-related, mostly from home in Hallbergmoos and HZG was renamed into Helmholtz-Zentrum Hereon. At the end of 2021 and beginning of 2022, I spent in total half a year as a guest researcher at the Division of Scientific Computing of the Department of Information Technology at Uppsala University, Sweden. In summer 2022, I joined the newly founded Chair for High-Performance Scientific Computing at the Institute of Mathematics of the University of Augsburg. Throughout my PhD studies, I met, collaborated with and benefited from many excellent scientists and worked on many different, interesting, and challenging topics with them. Listing all names is beyond the scope of these acknowledgments. I thank all of them.

In particular, I would like to thank my scientific supervisor, Prof. Martin Kronbichler, and my supervisor at Helmholtz-Zentrum Hereon, Prof. Christian Cyron. Martin introduced me into matrix-free methods and the `deal.II` finite-element library as well as helped with fruitful discussions to improve the quality of my work. Christian always showed great interest in my work and gave me full support. Furthermore, I would like to thank my external colleagues: Ivo Dravins, Vladimir Ivannikov, and Magdalena Schreter-Fleischhacker. Working with you on interdisciplinary projects was fun, and I have learned a lot from you.

I would like to acknowledge the collaboration with the `deal.II` community, particularly, my colleagues in the principal-developer team. It is a special honor to be part of this team and to be able to discuss challenging scientific issues at any time. Without the possibility to rely on your previous work in the library, many results of this thesis would not have been possible or would have been much harder to achieve. `deal.II` as a tool to connect researchers with different scientific background is truly inspiring.

Finally, I would like to thank my family for supporting my work in the last years.

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

- I M. Kronbichler, D. Sashko, and P. Munch, “Enhancing data locality of the conjugate gradient method for high-order matrix-free finite-element implementations”, *The International Journal of High Performance Computing Applications*, July 2022.
- II P. Munch, K. Ljungkvist, and M. Kronbichler, “Efficient application of hanging-node constraints for matrix-free high-order FEM computations on CPU and GPU”, in *High Performance Computing. ISC High Performance 2022*, (A. L. Varbanescu, A. Bhatele, P. Luszczek, B. Marc, eds.), pp. 133-152, Cham: Springer, 2022.
- III P. Munch, T. Heister, L. Prieto Saavedra, and M. Kronbichler, “Efficient distributed matrix-free multigrid methods on locally refined meshes for FEM computations”, *ACM Transactions on Parallel Computing (TOPC)*, vol. 10, no 1, pp.3:1-38, 2023.
- IV P. Munch and M. Kronbichler, “Cache-optimized and low-overhead implementations of additive Schwarz methods for high-order FEM multigrid computations”, *The International Journal of High Performance Computing Applications*, December 2023.
- V P. Munch, I. Dravins, M. Kronbichler, and M. Neytcheva, “Stage-parallel fully implicit Runge-Kutta implementations with optimal multilevel preconditioners at the scaling limit”, *SIAM Journal on Scientific Computing (SISC)*, pp. S71–S96, 2023.
- VI P. Munch, K. Kormann, and M. Kronbichler, “hyper.deal: an efficient, matrix-free finite-element library for high-dimensional partial differential equations”, *ACM Transactions on Mathematical Software*, vol. 47, no. 4, pp. 1–34, 2021.
- VII P. Munch, V. Ivannikov, C. Cyron, and M. Kronbichler. “On the construction of an efficient finite-element solver for phase-field simulations of many-particle solid-state-sintering processes”, *Computational Materials Science*, vol. 231, pp. 112589:1–21, 2024.

Reprints were made with permission from the publishers.

Contents

| | | |
|-------|---|----|
| 1 | Motivation | 1 |
| 1.1 | Background | 1 |
| 1.2 | Matrix-free methods for finite-element computations | 1 |
| 1.3 | Historical overview, state of the art & challenges | 3 |
| 1.4 | Aim of this thesis | 6 |
| 2 | Algorithmic aspects of efficient matrix-free computations | 8 |
| 2.1 | Derivation of the matrix-free algorithm | 8 |
| 2.2 | Cell loop & vector updates | 14 |
| 2.3 | Vector access & application of constraints | 15 |
| 2.4 | Evaluation & application of test function | 17 |
| 2.5 | Quadrature operation | 22 |
| 2.6 | Discontinuous Galerkin methods | 23 |
| 2.7 | Arbitrary shapes: simplex and mixed meshes | 24 |
| 2.8 | Local refinement | 25 |
| 2.9 | Example implementation & interface | 26 |
| 2.10 | Performance aspects | 32 |
| 2.11 | Variations: interleaving on cell level | 34 |
| 2.12 | Preconditioning | 35 |
| 3 | Accomplishments | 39 |
| 3.1 | Application to high-order FEM | 39 |
| 3.1.1 | Increasing the data locality | 39 |
| 3.1.2 | Application of hanging-node constraints | 41 |
| 3.1.3 | Multigrid: global coarsening for locally refined meshes | 42 |
| 3.1.4 | Multigrid: efficient block smoothers | 45 |
| 3.1.5 | Stage-parallel implicit Runge–Kutta methods | 47 |
| 3.2 | Application to high-dimensional FEM | 49 |
| 3.2.1 | Motivation: computational plasma physics | 49 |
| 3.2.2 | Software and performance aspects | 50 |
| 3.3 | Application to multicomponent FEM | 53 |
| 3.3.1 | Motivation: solid-state sintering | 53 |
| 3.3.2 | Software and performance aspects | 55 |
| 3.3.3 | Nonlinear solver and preconditioning | 56 |
| 3.4 | Coupling via non-matching grids | 56 |
| 3.4.1 | Motivation: multiphysics applications | 57 |
| 3.4.2 | Distributed search | 61 |

| | | |
|-------|--|----|
| 3.4.3 | Efficient operator evaluation | 63 |
| 3.4.4 | Software | 63 |
| 3.4.5 | Black-box coupling via preCICE | 65 |
| 4 | Additional publications and software | 67 |
| 4.1 | List of papers | 67 |
| 4.2 | List of publicly available software | 68 |
| 5 | Conclusions & Outlook | 70 |
| | References | 72 |

1. Motivation

1.1 Background

This thesis follows a long tradition in numerical analysis in that new mathematical techniques were developed in response to the inability to solve linear systems or to the requirements by novel computer hardware. Linear systems arising from partial differential equations (PDE) were first rather small and were solved by hand, using Gaussian elimination. The advent of computers allowed to solve larger systems but required, e.g., the development of pivoting strategies [13] and the backward stability analysis for the examination of the stability of numerical methods. The need to solve even larger systems, enabled by advances in computer hardware, required the development of algorithms whose computational complexity does not increase as $\mathcal{O}(N^3)$ with the number of unknowns. This requirement first led to development of reordering strategies [14] and banded direct linear solvers and, after that, to development (or revival) of iterative solvers like the conjugate gradient method [15] or the generalized minimal residual method (GMRES) [16]. This necessitated the examination of the convergence behavior based on the matrix properties and the development of preconditioners, i.e., good and cheap approximations of the inverse of the system matrix, like multigrid, to make these methods more efficient. Similarly, the introduction of parallel computers led to a shift towards simpler but easy-to-parallelize algorithms. The matrix-free evaluation of linear operators arising, e.g., in the context of finite element methods, the focus of this thesis, is motivated by the observation that, on today's processors, the machine balance increases, i.e., one can do much more computations while accessing the slow memory system. These methods perform redundant computations, e.g., evaluation of integrals, at high arithmetic intensity, however, are tailored to the given problem and do not allow the application of different well-established numerical algorithms. This requires the development of alternative variants by careful mathematical analysis and hardware-portable software design that allows for straightforward usage.

1.2 Matrix-free methods for finite-element computations

The established approach of solving partial differential equations with the finite element method (FEM) consists of two steps: In the first one, a system matrix and a right-hand-side vector are assembled, making up a system of linear equations, which needs to be solved in a second step. For the latter, direct

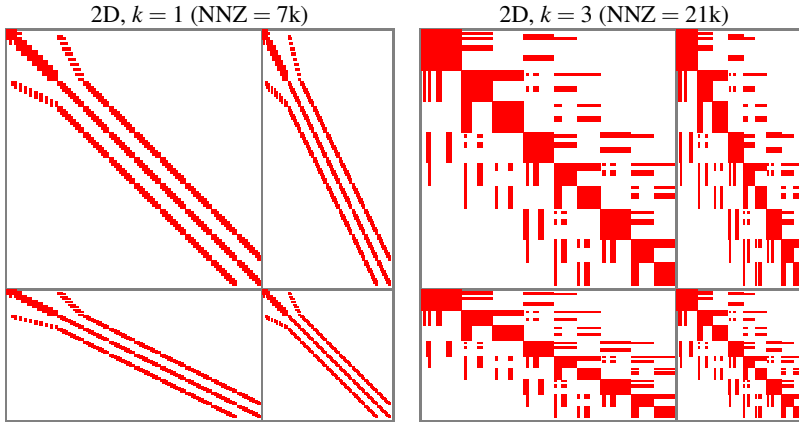


Figure 1.1. Sparsity patterns with identical number of rows and components $(2+1)$ for different polynomial degrees k . Such a sparsity pattern might occur in the context of the solution of stabilized incompressible Navier–Stokes equations.

matrix-based:

- ① $A = \sum_{K \in \{\text{cells}\}} R_K^T A_K R_K$
- ② $x = A^{-1}b \rightarrow$ direct or iterative solver

matrix-free (\rightarrow matvec for iterative solver):

$$v = Au = \sum_{K \in \{\text{cells}\}} R_K^T A_K R_K v$$

... with preconditioner $P^{-1} = A^{-1}$.

Figure 1.2. Matrix-based vs. matrix-free FEM

solvers and iterative solvers using preconditioners from well-established and publicly available linear-algebra packages can be used.

This two-step *matrix-based* approach is feasible as long as the intermediate quantity—the system matrix—is not the bottleneck from memory and performance points of view. A (locally) dense sparse matrix with a high number of non-zeroes (NNZ, see Fig. 1.1) implies not only a high memory consumption but also high costs for setup and application, since each entry needs to be loaded from the slow main memory (random access memory; RAM). Examples where computing and storing the matrix are inherently expensive are the high-order computational fluid dynamics (CFD) or other nonlinear problems, in which the Jacobian matrix is only reused a few times.

This issue motivates the development of alternative approaches that do not rely on matrices, i.e., are *matrix-free* by nature. In the following, we consider only methods that return—up to machine precision—the same results as the equivalent matrix-based algorithms. Such matrix-free operator evaluations

are usually applied inside iterative solvers, which do not need the actual matrix but only the effect of the matrix-vector product, and loop over geometric entities of the mesh (e.g., vertices or cells; Fig. 1.2), while applying stencils or element stiffness matrices on a vector directly. Exploiting the structure of the dense local element stiffness matrix allows, e.g., to derive algorithms that are memory-efficient and use modern processors optimally.

Matrix-free computations imply that one can not or does not want to compute the system matrix for initialization of the linear solver and the preconditioner like incomplete LU factorization (ILU) and algebraic multigrid (AMG). This does not mean that—given a matrix-free operator—one can not reconstruct its matrix representation, however, in doing so one would give up the initial intention of matrix-free algorithms. Not surprisingly, the development of efficient matrix-free operator evaluations comes hand in hand with the development of efficient iterative solvers, in particular, of *preconditioners* based, e.g., on domain decomposition methods (DDM) or multigrid methods (MG). They can be evaluated in a matrix-free way and only need the mesh or reduced information on the system matrix, like its diagonal, which is straightforward to obtain.

The term “matrix-free” implies that the standard algorithm uses matrices, as is often the case in FEM, and circumventing the usage of matrices is an innovation. However, the algorithms labeled as “matrix-free” are also applicable to compute right-hand-side vectors or nonlinear residuals, for which one would not assemble matrices. This is not surprising, since the matrix-vector product is mapped to a residual evaluation, for which the optimizations are performed.

1.3 Historical overview, state of the art & challenges

The type of matrix-free methods that we study originates in the spectral-element community [17] in the 1970s. Spectral element methods (SEM), which consider high-order continuous finite elements, became popular in the context of (incompressible) computational fluid dynamics in the 1990s and with those also matrix-free computations, e.g., for solving the pressure Poisson problem, which normally consumes the largest portion of the simulation time. This work was pioneered by Paul Fischer and the NEK5000 project [18–22].

In the early 2010s, matrix-free approaches have been adopted by the finite-element community and generalized from a software perspective for the evaluation of more general operators. These works include the publications by Brown et al. [23, 24], Kronbichler and Kormann [25], and Sherwin et al. [26–28]. In the last few years, there has been considerable interest in matrix-free computations due to their compliance with current computer-hardware trends. In particular, the rise in the usage of general-purpose GPUs, which

have higher bandwidth and higher performance compared to CPUs and consequently promise higher throughput for the same arithmetic intensity, led to a growing interest in matrix-free methods [29–31]. By now, matrix-free computations have been used in the context of different applications of not only CFD [32–35] but also other branches of science (e.g., computational mechanics [36, 37], phase-field simulations [38], density functional theory [39–41], cardiac electrophysiology [42], fluid-structure interaction [43, 44], acoustics [45, 46], geosciences [47, 48]) and are supported by a number of open-source libraries (e.g., `deal.II` [25, 49], MFEM [50], DUNE [51], Firedrake [52]). A special library is `libCEED` [53], which is solely focused on matrix-free operator evaluations. More details on this will be given below.

Despite their conceptual simplicity and convincing advantages as a means of improving performance, matrix-free finite-element computations are, at the time of writing this thesis, not standard yet. This is related to the fact that matrix-free computations are only viable if efficient representations of the element stiffness matrix, which do not introduce excessive redundant computations, are available and a suitable preconditioner can be constructed so that the overall costs are not higher than the ones in the matrix-based case. For simple operators, like a Poisson operator on meshes only consisting of tensor-product-shaped cells, this is the case, as presented multiple times in the literature [25, 54, 55]. However, application scientists and other users of FEM libraries usually want to solve more challenging and novel PDEs without dealing with implementation details of the underlying libraries, like representation of the element stiffness matrix or construction and configuration of the preconditioner. Instead, they wish to concentrate on physical modeling or other challenging scientific aspects of their research field, relying on the high quality of the used software package. Working in large, interdisciplinary teams consisting of mathematicians, computer scientists, and engineers would allow to tackle all of these issues at once and to develop efficient special-purpose solvers. However, such teams are currently rather an exception in the academic and research world. To nevertheless allow any domain expert to take full advantage of matrix-free computations without too many obstacles, it is crucial to provide easy-to-use implementations with a clear *application programming interface (API)*. Such prototypical implementations are provided by open-source software packages where a team of mathematicians and computer scientists can concentrate on maintaining and improving the implementation via bug fixes or performance optimizations as well as adding features depending on the needs.

Current research efforts are spearheaded by developers of open-source libraries, each with a slightly different focus, and are addressing the change of this situation by targeting the following issues:

- ① user-friendly interface: Providing and improving an API are crucial. Different libraries pursue different strategies in this regard to improve the usability. The library `deal.II`, for example, provides helper classes that

can be used directly by users, while Dune and Firedrake rely on code generation [56, 57] from the Unified Form Language (UFL). The latter approach allows to write a single code that can be optionally converted to a matrix-based or a matrix-free code.

- ② off-the-shelf building blocks & preconditioners: The development of matrix-free *off-the-shelf* building blocks is crucial, too. These include widely used preconditioners and operators (e.g., Poisson or mass-matrix operators), which justify a dedicated holistic performance optimization. Providing preconditioners that work in all situations is practically impossible, however, simple preconditioners can be composed by users to more complex ones, e.g., in the context of block preconditioners.
- ③ performance & portability: Matrix-free methods need to be (significantly) faster than their matrix-based equivalents to convince users to switch to these in existing code bases. This requirement and the fact that matrix-free operators perform operations over and over (in each iteration, in contrast to the matrix-based approach, where much of the complex evaluations happens only once during the assembly loop) require and justify extensive performance optimizations, which are normally tailored to a given hardware. To allow nevertheless hardware portability, special considerations need to be taken. These might influence the choices regarding API, ▷ issue ①. For instance, users of `deal.II` work on data structures built around the most advanced instruction-set architecture of the given CPU-based hardware and `libCEED` only allows to write code that can be dispatched on both CPU- and GPU-based hardware.
- ④ missing features: Many widely used features of FEM libraries are still slow or not ported/portable in the context of matrix-free computations. Such features are the support of simplex/mixed meshes crucial to simulate complex geometries as well as $H(\text{div})$ and $H(\text{curl})$ elements (e.g., Raviart–Thomas and Nedelec elements). The support of these features is necessitated by many applications, and the lack of it is a major obstacle for the usage of matrix-free algorithms in these application areas, e.g., in electromagnetics.
- ⑤ real-world applications: Developers of matrix-free algorithms are generally closely affiliated with user codes that extensively use the matrix-free algorithms and are involved in porting of existing or new applications together with domain experts, e.g., in the context of challenging multiphysics coupled problems. Such collaborations allow to refine the interface, ▷ issue ①, and to identify missing building blocks, ▷ issue ②. The focus on the development and porting of missing features, ▷ issue ④, might receive more attention. Furthermore, algorithms that can not be ported to matrix-free algorithms can be identified, which allows to derive alternative algorithms that can be expressed suitably for matrix-free computations by using existing building blocks. The success of porting challenging applications is motivation for other projects to do the

same, and lessons learned there can be directly adopted to new application fields.

The Center for Efficient Exascale Discretizations (CEED [30, 53, 58, 59]¹), a co-design center within the U.S. Department of Energy (DOE), consisting of researchers from, i.a., MFEM, NEK5000/nekRS, and PETSc, works on the standardization of the matrix-free interface with special focus on hardware portability, ▷ issues ① and ③. A unified interface seems to be appealing, particularly, considering the success of BLAS (Basic Linear Algebra Subprograms), which allows to easily switch backends provided and optimized by hardware vendors. This development, however, might be too early, since many of the issues listed above are still open research. Furthermore, it is not clear how much interest such a unified interface will generate outside the FEM community, justifying the decoupling of FEM and matrix-free operator evaluations, which possibly does not allow certain domain-specific optimizations.

1.4 Aim of this thesis

The overall aim of this thesis is the development of efficient algorithms and their implementations for matrix-free computation in the context of FEM at extreme scale in regard to problem size and hardware usage. The work is motivated by the challenges of concrete applications: in particular, rather classical application cases of high-order computational fluid dynamics and novel application areas of high-dimensional computational plasma physics and solid-state sintering with phase-field simulations needing high and dynamically changing number of components. Furthermore, matrix-free algorithms for multiphysics applications, e.g., fluid-structure interaction (FSI) and two-phase flow, are investigated, requiring the adaption of the standard matrix-free algorithm to a non-matching-grid setting. Within the matrix-free challenges listed in Section 1.3, the focus of this thesis lies on the following issues:

- We port challenging real-world applications (▷ issue ⑤; e.g., in computational plasma physics, solid-state sintering, fluid-structure interaction, and two-phase flow) to the context of matrix-free computations, for which new algorithms are developed.
- For this, we develop off-the-shelf components and preconditioners, ▷ issue ②, with emphasis on multigrid and locally refined meshes. The implementations are made publicly available via the open-source general-purpose finite-element library deal.II.
- We perform node-level and large-scale performance optimizations, ▷ issue ③, and demonstrate the efficiency of the developed algorithms with up to 150k processes on 3072 compute nodes on SuperMUC-NG, Germany's former largest supercomputer.

¹<https://ceed.exascaleproject.org/>

While not being the focus, we also contributed to the issues ① and ④. In this thesis, we concentrate on CPU-based systems due to the current hardware landscape in Germany. However, some work has been also conducted on GPU systems, on which we will comment in relevant parts of the thesis. We would like to note that, while the focus of this thesis is on the matrix-free development, we were also involved in other aspects of FEM development, motivated by the fact that once a finite-element operator has been optimized by matrix-free algorithms, other parts of the code might become new bottlenecks.

We concentrate on cell-based matrix-free operator evaluations. Interested readers are referred to, e.g. [48] for details on stencil-based variants, which have similarities to matrix-free finite difference methods [60].

The structure of the introductory part of this thesis is as follows. Section 2 gives an introduction into efficient matrix-free operator evaluations, which form the basis of this thesis. In Section 3, we summarize the major accomplishments of this work, the application and extension of the basic matrix-free algorithm in the context of preconditioning, computational plasma physics, solid-state sintering, and non-matching computations. Section 4 lists additional publications written in the scope of this thesis and all major software contributions developed in this time. Section 5 concludes this work by providing an outlook on future research directions.

2. Algorithmic aspects of efficient matrix-free computations

This section discusses the software aspects of efficient matrix-free finite-element computations. For this purpose, we consider different types of PDEs and finite-element discretizations, which we first introduce.

Note: The following section introduces concepts and implementation details on matrix-free operator evaluation from the literature, e.g., [34, 61], that are relevant for this thesis. We use some of the naming conventions from the deal.II project and present its matrix-free implementation as a possible high-performance implementation. It was written by Martin Kronbichler, Katharina Kormann, and others and is described in detail in [25, 49]. The author of this thesis built upon and benefited from the implementations and contributed to them in the course of working on the main contribution described in Section 3.

2.1 Derivation of the matrix-free algorithm

Poisson equation & Laplace operator

The Poisson equation is given as

$$-\Delta u = f \quad \text{in } \Omega,$$

where u is the solution variable and f is a source term. The physical domain Ω is bounded by $\partial\Omega = \Gamma$, which is partitioned into the Dirichlet part Γ_D , where $u = g_D$, and the Neumann part Γ_N , where $-\mathbf{n} \cdot \nabla u = g_N$ is prescribed. The vector \mathbf{n} denotes the outer unit normal vector on the boundary Γ . For discretization, we assume a tessellation \mathcal{T}_h of the computational domain Ω_h into cells Ω_K . The bilinear forms that are associated with volume and face integrals are denoted by

$$(v, u)_\Omega = \int_\Omega v \odot u \, d\Omega \quad \text{and} \quad \langle v, u \rangle_\Gamma = \int_\Gamma v \odot u \, d\Gamma,$$

where the operator \odot symbolizes inner products.

In the context of continuous Galerkin formulation, the final weak form is to find a function $u_h \in V_{h, g_D}^{\text{CG}}$ such that

$$(\nabla v_h, \nabla u_h)_{\Omega_h} = (v_h, f)_{\Omega_h} - \langle v_h, g_N \rangle_{\Gamma_N} \quad \forall v_h \in V_{h, 0_D}^{\text{CG}}. \quad (2.1)$$

We assume a polynomial approximation of the solution of elements from the space

$$V_{h,g_D}^{\text{CG}} = \{u_h \in H^1(\Omega) : u_h|_{\Omega_K} \in \mathcal{Q}_k(\Omega_K) \forall \Omega_K \in \mathcal{T}_h \wedge u_h|_{\Gamma_D} = g_D\}.$$

H^1 is the space of square integrable functions with square integrable derivatives. $\mathcal{Q}_k(\Omega_K)$ donates the spaces spanned by degree- k -polynomials on Ω_K , where $u_h^K(x) = \sum_{i=1}^{N_{\text{DoFs}}} \phi_i(x) u_{K,i}$ donates the local FE interpolation associated to the unknowns (degrees of freedoms; DoFs) restricted to the cell K and to the set of corresponding basis functions ϕ_i . The solution space satisfies the Dirichlet boundary condition g_D on Γ_D . On each cell, the left-hand side gives rise to an element stiffness matrix A_K and the right-hand side to an element vector b_K :

$$A_{K,ij} = \underbrace{(\nabla\phi_i, \nabla\phi_j)_{\Omega_K}}_* \quad \text{and} \quad b_{K,i} = (\phi_i, f)_{\Omega_K} - \underbrace{(\phi_i, g_N)_{\Gamma_K \cap \Gamma_N}}_{\ddagger}. \quad (2.2)$$

In standard (matrix-based) finite-element computations, the local quantities A_K and b_K are assembled via a local-to-global mapping (restriction operator; R_K) into the global stiffness matrix $A = \sum_K R_K^T A_K R_K$ and the global load vector $b = \sum_K R_K^T b_K$, resulting in a linear system $Ax = b$. The underbraced terms are discussed below in more detail.

In the context of symmetric interior-penalty discontinuous Galerkin (DG) formulation [55, 62], the final weak form, for a single cell, is, e.g., to find a function $u_h \in V_h^{\text{DG}}$ such that

$$\begin{aligned} & \underbrace{(\nabla v_h, \nabla u_h)_{\Omega_K}}_* - \underbrace{\langle \nabla v_h, (u_h^- - \{\{u_h\}\}) \mathbf{n} \rangle_{\Gamma_K \setminus (\Gamma_D \cup \Gamma_N)}}_{\diamond} \\ & \quad - \underbrace{\langle v_h, \{\{ \nabla u_h \} \} \cdot \mathbf{n} - \tau \llbracket u_h \rrbracket \cdot \mathbf{n} \rangle_{\Gamma_K \setminus (\Gamma_D \cup \Gamma_N)}}_{\diamond \text{ (cont.)}} \\ & \quad - \underbrace{\langle \nabla v_h, u_h \mathbf{n} \rangle_{\Gamma_K \cap \Gamma_D} - \langle v_h, \nabla u_h \cdot \mathbf{n} - 2\tau u_h \rangle_{\Gamma_K \cap \Gamma_D}}_{\dagger} \\ & = (v_h, f)_{\Omega_K} - \langle \nabla v_h, g_D \mathbf{n} \rangle_{\Gamma_K \cap \Gamma_D} + \langle v_h, 2\tau g_D \rangle_{\Gamma_K \cap \Gamma_D} \quad (2.3) \\ & \quad - \underbrace{\langle v_h, g_N \rangle_{\Gamma_K \cap \Gamma_N}}_{\ddagger} \quad \forall v_h \in V_h^{\text{DG}}, \end{aligned}$$

with the trial space

$$V_h^{\text{DG}} = \{u_h \in L_2(\Omega) : u_h|_{\Omega_K} \in \mathcal{Q}_k(\Omega_K) \forall \Omega_K \in \mathcal{T}_h\}.$$

L_2 is the space of square integrable functions. The average operator is given as $\{\{u\}\} = (u^- + u^+)/2$, and the jump operator is given as $\llbracket u \rrbracket = u^- \mathbf{n}^- + u^+ \mathbf{n}^+$. The superscript \square^+ donates exterior information from the neighbor cell. The

face terms are split into inner-face terms $\Gamma_e \setminus (\Gamma_D \cup \Gamma_N)$ and boundary-face terms (Γ_D, Γ_N) . Inhomogenous boundary conditions are applied weakly.

Now, let us discuss three types of terms in (2.2) and (2.3). Term * corresponds to a *volume integral*. Replacing this integral by a quadrature rule with N_q quadrature points and introducing a mapping $\mathbf{x} \leftarrow \chi_K^{-1}(\hat{\mathbf{x}})$, in which the element (\mathbf{x}) is the image of a reference element ($\hat{\mathbf{x}}$), we get the following definition of the element stiffness matrix:

$$(A_K)_{ij} = \sum_{0 \leq q < N_q} |J_K(\hat{\mathbf{x}}_q)| w_q \left(J_K^{-T}(\hat{\mathbf{x}}_q) \hat{\nabla} \hat{\phi}_i(\hat{\mathbf{x}}_q) \right)^T \left(J_K^{-T}(\hat{\mathbf{x}}_q) \hat{\nabla} \hat{\phi}_j(\hat{\mathbf{x}}_q) \right),$$

where $\hat{\mathbf{x}}_q$ donates the position of the quadrature point q in reference space, $\hat{\phi}_i(\hat{\mathbf{x}})$ is defined on the reference cell, $\hat{\nabla}$ is the gradient on the reference cell, $(J_K^{-T}(\hat{\mathbf{x}}))_{ij} = \partial \mathbf{x}_i / \partial \hat{\mathbf{x}}_j = \partial (\chi_K^{-1}(\hat{\mathbf{x}}))_i / \partial \hat{\mathbf{x}}_j$, $1 \leq i, j \leq \text{dimension } d$, is the Jacobian, $|J_K(\hat{\mathbf{x}}_q)|$ is the determinant of the Jacobian, and w_q is the quadrature weight. After some reformulations, the application of this matrix to a vector $v = A_K u_K$ reads as

$$v_{K,i} = \underbrace{\sum_{0 \leq q < N_q} \hat{\nabla} \hat{\phi}_i(\hat{\mathbf{x}}_q)}_{\textcircled{3}} \cdot \underbrace{[|J_K(\hat{\mathbf{x}}_q)| w_q J_K^{-1}(\hat{\mathbf{x}}_q) J_K^{-T}(\hat{\mathbf{x}}_q)]}_{\textcircled{2}} \underbrace{\sum_{0 \leq j < N_{\text{DoFs}}} \hat{\nabla} \hat{\phi}_j(\hat{\mathbf{x}}_q) u_{K,j}}_{\textcircled{1}},$$

which implies that the operation can be decomposed into three steps $\textcircled{1}$ – $\textcircled{3}$. Here, the first term corresponds to the evaluation of the gradient at the quadrature point of the cell, the second term to operations on the quadrature-point level, and the third term to testing in reference space that includes the summation over the quadrature points. In the following, we make use of the two-step notation

$$\begin{aligned} \textcircled{1} \quad & \hat{\nabla} \hat{u}_{K,q} = \hat{\nabla} \hat{\phi}_j(\hat{\mathbf{x}}_q) u_{K,j} \quad \forall q : 0 \leq q < N_q \\ \textcircled{2} + \textcircled{3} \quad & v_{K,i} = \sum_{0 \leq q < N_q} \hat{\nabla} \hat{\phi}_i(\hat{\mathbf{x}}_q) \cdot |J_K(\hat{\mathbf{x}}_q)| w_q J_K^{-1}(\hat{\mathbf{x}}_q) J_K^{-T}(\hat{\mathbf{x}}_q) \hat{\nabla} \hat{u}_{K,q} \end{aligned}$$

as a generalized notation suitable also for the evaluation of (nonlinear) residuals [23].

Term † in (2.3) corresponds to a *boundary-face integral* and can be rewritten using a quadrature rule defined on faces as

$$\begin{aligned} \begin{pmatrix} \hat{u}_{K,q} \\ \hat{\nabla} \hat{u}_{K,q} \end{pmatrix} &= \sum_{0 \leq j < N_{\text{DoFs}}} \begin{pmatrix} \hat{\phi}_j(\hat{\mathbf{x}}_q) \\ \hat{\nabla} \hat{\phi}_j(\hat{\mathbf{x}}_q) \end{pmatrix} u_{K,j} \quad \forall q \\ v_{K,i} &= \sum_{0 \leq q < N_q} \begin{pmatrix} \hat{\phi}_i(\hat{\mathbf{x}}_q) \\ \hat{\nabla} \hat{\phi}_i(\hat{\mathbf{x}}_q) \end{pmatrix} \cdot |J_K(\hat{\mathbf{x}}_q)| w_q \begin{pmatrix} -(J_K^{-T} \hat{\nabla} \hat{u}_{K,q}) \cdot \mathbf{n} + 2\tau \hat{u}_{K,q} \\ -J_K^{-1}(\hat{\mathbf{x}}_q) \mathbf{n}(\hat{\mathbf{x}}_q) \hat{u}_{K,q} \end{pmatrix}, \end{aligned}$$

which again allows a decomposition in three steps. Similarly, term \ddagger in (2.2) and (2.3) is a boundary-face term, too. It can be written as

$$v_i = \sum_{0 \leq q < N_q} \hat{\phi}_i(\hat{\mathbf{x}}_q) \cdot [|J_K(\hat{\mathbf{x}}_q)| w_q J_K^{-1}(\hat{\mathbf{x}}_q) g_N(x_q)]$$

and only needs the evaluation of the function g_N at the real positions of the quadrature points.

Term \diamond in (2.3) corresponds to an *internal-face integral*. Here, the values and gradients of neighboring cells need to be evaluated at quadrature points given by the faces of cell K . For this purpose, the reference position on the face of the neighboring cell $\hat{\mathbf{x}}_q^+$ has to be determined:

$$\hat{\mathbf{x}}_q^+ = \chi(x_q, \Omega_K^+) \quad \text{with} \quad x_q = \chi^{-1}(\hat{\mathbf{x}}_q, \Omega_K). \quad (2.4)$$

Now, the values and gradients can be computed on both sides by

$$\begin{pmatrix} \hat{u}_{K,q} \\ \hat{\nabla} \hat{u}_{K,q} \end{pmatrix} = \sum_{0 \leq j < N_{\text{DoFs}}} \begin{pmatrix} \hat{\phi}_j(\hat{\mathbf{x}}_q) \\ \hat{\nabla} \hat{\phi}_j(\hat{\mathbf{x}}_q) \end{pmatrix} u_{K,j} \quad \forall q, \quad (2.5a)$$

$$\begin{pmatrix} \hat{u}_{K,q}^+ \\ \hat{\nabla} \hat{u}_{K,q}^+ \end{pmatrix} = \sum_{0 \leq j < N_{\text{DoFs}}} \begin{pmatrix} \hat{\phi}_j^+(\hat{\mathbf{x}}_q^+) \\ \hat{\nabla} \hat{\phi}_j^+(\hat{\mathbf{x}}_q^+) \end{pmatrix} u_{K,j}^+ \quad \forall q, \quad (2.5b)$$

which corresponds to the first term of the cell integral and, as a consequence, term \diamond fits well with the decomposition into three steps, allowing to precompute quantities at both sides before proceeding with the loop over quadrature points:

$$v_{K,i} = \sum_{0 \leq q < N_q} \begin{pmatrix} \hat{\phi}_i(\hat{\mathbf{x}}_q) \\ \hat{\nabla} \hat{\phi}_i(\hat{\mathbf{x}}_q) \end{pmatrix} \cdot |J_K(\hat{\mathbf{x}}_q)| w_q \begin{pmatrix} -0.5(J_K^{-T}(\hat{\mathbf{x}}_q) \hat{\nabla} \hat{u}_q + J_K^{-T}(\hat{\mathbf{x}}_q^+) \hat{\nabla} \hat{u}_q^+) \cdot \mathbf{n} + \tau(\hat{u}_q - \hat{u}_q^+) \\ -0.5 J_K^{-1}(\hat{\mathbf{x}}_q - \hat{\mathbf{x}}_q^+) \mathbf{n} \end{pmatrix}. \quad (2.5c)$$

Mass-matrix operator

The weak form of the mass-matrix operator reads as

$$(v_h, u_h)_{\Omega_h} \quad \text{and} \quad (v_h, u_h)_{\Omega_e}$$

for the continuous and the discontinuous Galerkin case, respectively. The action of the element stiffness matrix can be expressed on a cell level as

$$\hat{u}_{K,q} = \sum_{0 \leq j < N_{\text{DoFs}}} \hat{\phi}_j(\hat{\mathbf{x}}_q) u_{K,j} \quad \forall q \quad (2.6a)$$

$$v_{K,i} = (M_K u_K)_i = \sum_{0 \leq q < N_q} \hat{\phi}_i(\hat{\mathbf{x}}_q) \cdot |J_K(\hat{\mathbf{x}}_q)| w_q \hat{u}_{K,q}, \quad (2.6b)$$

which again can be decomposed into three steps.

Note: Due to the importance of the mass matrix for the transfer between two solution spaces (e.g., $V_{h,gD}^{CG}$ or V_h^{DG} in the context of multigrid or non-nested meshes), we provide some basic information here. An L_2 -projection from space 0 to space 1 reads as

$$(v_1, u_1)_\Omega = (v_1, u_0)_\Omega \quad (2.7)$$

and, in matrix notation, $M_{11}u_1 = M_{10}u_0$, where M_{11} needs to be inverted. Choosing a set of quadrature points allows to adopt (2.6) for both the left-hand-side and the right-hand-side term. Alternatively to the L_2 -projection, one can perform a global pointwise evaluation in the case that space 1 is nodal. On a cell level, this operation is described by (2.6a). Contributions from multiple cells need to be weighted, e.g., by the inverse of the valence of each point.

Advection equation

The skew-symmetric DG discretization of the advection equation

$$\partial u / \partial t + \nabla \cdot \mathbf{a}u = 0 \quad \text{and} \quad \Omega \times [0, t_{\text{final}}]$$

reads, for a divergence-free \mathbf{a} , as follows [63]:

$$\begin{aligned} (v, \partial u / \partial t)_{\Omega_K} = & \underbrace{(v, -\beta(\mathbf{a} \cdot \nabla u))_{\Omega_K} + (\nabla v, (1 - \beta)\mathbf{a}u)_{\Omega_K}}_{*} \\ & - \underbrace{\langle v, \mathbf{n} \cdot (\mathbf{a}u)^* - \beta u^- (\mathbf{n} \cdot \mathbf{a}) \rangle_{\Gamma_K}}_{\dagger}, \end{aligned} \quad (2.8)$$

with the element domain Ω_K and $(\mathbf{a}u)^*$ being a numerical flux like a central ($\alpha = 0$) or an upwind flux ($\alpha = 1$):

$$(\mathbf{a}u)^* = \frac{1}{2} ((\mathbf{a}u^- + \mathbf{a}u^+) + \alpha \cdot |\mathbf{n} \cdot \mathbf{a}| \cdot (\mathbf{n}u^- - \mathbf{n}u^+)).$$

The factor β controls the formulation of the flux ($\beta = \frac{1}{2}$: skew-symmetric; $\beta = 0$: conservative).

Summary

The previous examples showed that the application of element stiffness matrices related to cell and face integrals can be decomposed into three steps. For this purpose, we adopt an operator notation:

$$v_K = \mathcal{A}_K u_K = \mathcal{S}_K^T \circ \mathcal{Q}_K \circ \mathcal{S}_K u_K, \quad (2.9)$$

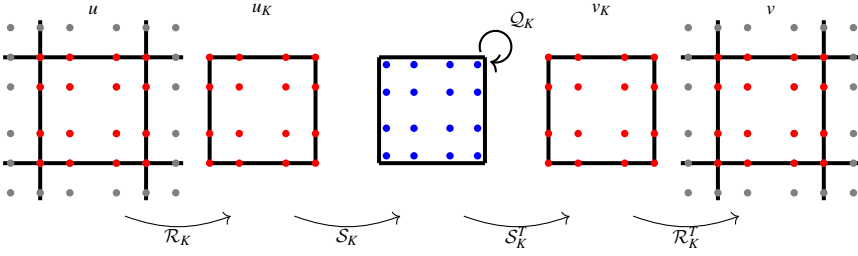


Figure 2.1. Matrix-free operator evaluation without the application of constraints.

where \circ indicates the composition of operators. Here, S_K computes optionally the values, gradients, or Hessians in reference space at the quadrature points. These quantities are used by Q_K . The application of element stiffness matrices is called within a loop over cells/faces:

$$v = Au = \sum_K \mathcal{R}_K^T \circ \mathcal{C}_K^T \circ \mathcal{A}_K \circ \mathcal{C}_K \circ \mathcal{R}_K u, \quad (2.10)$$

for which cell-/face-relevant DoFs need to be gathered with \mathcal{R}_K , constraints have to be resolved with \mathcal{C}_K , and the result needs to be added to the result vector with \mathcal{R}_K^T . Putting everything together results in the formula [25]:

$$v = \mathcal{A}(u) = \sum_K \mathcal{R}_K^T \circ \mathcal{C}_K^T \circ \underbrace{\mathcal{S}_K^T \circ \mathcal{Q}_K \circ \mathcal{S}_K}_{v_K = \mathcal{A}_K(u_K)} \circ \mathcal{C}_K \circ \mathcal{R}_K u. \quad (2.11)$$

Fig. 2.1 visualizes this formula for a single cell. As a generalization to cells/faces, one can consider (2.10) as a loop over arbitrary objects like patches or cell pairs, an observation we use in Paper IV (see Section 3.1.4) and Paper VI (see Section 3.2).

Next, we discuss *building blocks* (abbr. BB) needed for the efficient implementation of (2.11) as well as study recurring *design patterns*. Our ambition is to design an algorithm that is not only general enough for the presented examples but also applicable to novel and challenging cases as those discussed in Section 3. Similarly, building blocks and design choices are discussed in [49] and [64], respectively.

We conclude this subsection with the first BB.

BB1: cell/face loops and integrals

Matrix-free operators loop over cells/faces,

$$v = \mathcal{A}(u) = \sum_K \mathcal{R}_K^T \circ \mathcal{C}_K^T \circ \mathcal{A}_K \circ \mathcal{C}_K \circ \mathcal{R}_K u,$$

gather/scatter cell-relevant unknowns, optionally resolve constraints, and apply the effect of the element stiffness matrix to a local vector u_K . The application of the element stiffness matrix can be decomposed in three phases:

$$v_K = \mathcal{A}_K(u_K) = \mathcal{S}_K^T \circ \mathcal{Q}_K \circ \mathcal{S}_K u_K.$$

2.2 Cell loop & vector updates

Matrix-free operators iterate, according to (2.10), over all (N_c) cells,

$$v = \sum_{0 \leq K < N_c} \mathcal{R}_K^T \circ \mathcal{A}_K \circ \mathcal{R}_K u,$$

and read the cell-relevant DoFs from the source vector u with \mathcal{R}_K and write the result back into the destination vector v with \mathcal{R}_K^T . Here, \mathcal{R}_K and \mathcal{R}_K^T prescribe the *read/write dependency* regions of a cell, which might be the same, e.g., in the context of continuous elements, or distinct, e.g., in the DG case, where one needs to read the DoFs from neighboring cells but only modifies cell-local DoFs.

We would like to note that we operate not on individual cells but on *cell batches* and perform the cell operation (2.9) for each of them in one go in a vectorization-over-cells fashion to increase the throughput, by explicitly exploiting SIMD (single instruction, multiple data) capabilities of modern processors. Without loss of generality, we refer to cell batches simply as cells. We defer the discussion on vectorization and implementation details of \mathcal{R}_K to Sections 2.3 and 2.9.

Overlapping dependency regions of cells imply that neighboring cells should be processed in sequence to use temporal-locality properties of caches. In the following, we assume that cells are well enumerated, e.g., based on a space-filling curve.

Splitting up the index space of cells $[0, N_c[$,

$$v = \sum_r \underbrace{\sum_{N_c^r \leq K < N_c^{r+1}} \mathcal{R}_K^T \circ \mathcal{A}_K \circ \mathcal{R}_K u}_*, \quad (2.12)$$

and processing the resulting cell ranges (term $*$) with a reasonable granularity individually make sense for different reasons:

1. In the case of tasking, ranges with independent dependency regions can be assigned to threads in a way that no race conditions happen, for

which a dependency analysis, e.g., based on graph coloring, has to be performed. In the following, we ignore threading capabilities and refer interested readers to [65, 66].

2. In the case of distributed computations, cell ranges might depend on remote processes or influence them, requiring communication with MPI. Computations on cell ranges without dependency on remote processes can be, however, overlapped with the communication.¹
3. Similarly, vector updates on the source and destination vectors (u and v , respectively) can be performed before and after cell ranges. In this context, vector entries need to be in a valid state once the first cell (range) touches them and they are finalized and, as a result, can be used and modified once the last cell (range) has touched them. We discuss in detail the latter concept and some examples in Paper I (see Section 3.1.1).

BB2: cells, cell batches, cell ranges, and data dependency

The restriction operator \mathcal{R}_K determines the read/write dependency of cells. Processing cells in batches and these in ranges is a good fit to modern SIMD-/cache-based architectures. The dependency regions of cell ranges determine 1) cells that can be processed by threads independently without causing race conditions, 2) cells that do not depend on values from remote processes and can be processed in parallel to communication, and 3) vector ranges, whose update can be interleaved with the cell loops.

2.3 Vector access & application of constraints

The gathering operator \mathcal{R}_K and the scattering operator \mathcal{R}_K^T are Boolean matrices (DoF maps) representing indirect addressing into the source and destination vectors. They store the relevant DoF indices for each cell, implying a memory consumption of $\mathcal{O}((k+1)^d)$ per cell.

Depending on the discretization and the numbering of DoFs, however, not all entries of the Boolean matrix need to be stored and the remaining indices can be reconstructed. For example, in the case of DG, it is enough to store the index of the first DoF of a cell, since the remaining DoFs are generally enumerated contiguously. In the context of FEM, the situation is somewhat more complicated. Nevertheless, one can adopt a similar concept here as well: one stores the first index of each geometric entity of a cell (see Paper I; Section 3.1.1). This implies that one only needs to store $3^3 = 27$ indices for a

¹Note that this implies that ranges need to be assigned to three categories. The first category does not depend on remote processes and is processed, while ghost values are updated. The second one depends on remote processes and optionally influences them. The third category does not depend on remote processes again and can be processed, while computations from different processes are collected.

hexahedron (8 vertices, 12 lines, 6 faces, 1 cell) and can, thereby, improve the data access, since contiguous data can be streamed from the global vector. As a requirement for this, DoFs have to be enumerated contiguously within each geometric entity. Furthermore, DoFs might need to be permuted in a post-processing step in the event that geometric entities are not orientated in the correct way, as is generally the case for unstructured meshes [67].

In the case of MPI-3.0, processes can access not only their own vector entries but also the vector entries owned by processes on the same shared-memory domain. Naturally, indices in the Boolean matrix would correspond to a pair of the rank and the (local) index within that rank. Obviously, this is not a feasible approach on a DoF level, however, can be efficiently used if only the first index (of a cell) has to be stored, to avoid duplicating data during MPI communication. However, working directly on the data of other processes leads to race-condition issues just as in the case of threading, requiring techniques discussed in Paper VI (see Section 3.2).

Optionally, constraints need to be resolved during the cell loop. They relate constrained DoFs dependent on constraining DoFs in form of affine combinations $x_i = C_{ij}x_j + b_i$ with possible inhomogeneity b . This implies that the Boolean matrix might have multiple entries per row and the weights need to be stored. The unity $\mathcal{C}_K \circ \mathcal{R}_K$ implies a compressed-row-storage (CRS) format, which can be partly compressed by only storing unique rows [25]. Although the constraint matrix is generally sparse, it might become locally dense for certain constraint types. For this purpose, the constraint matrix can be decomposed as $\mathcal{C}_K = \mathcal{C}_K^{\text{SP}} \circ \mathcal{C}_K^{\text{GP}}$ and executed as $\mathcal{C}_K^{\text{SP}} \circ (\mathcal{C}_K^{\text{GP}} \circ \mathcal{R}_e)$, with $\mathcal{C}_K^{\text{SP}}$ applying special-purpose (SP) constraints after DoFs have been loaded and general-purpose (GP) constraints have been applied. This decomposition approach enables to switch data structures of $\mathcal{C}_K^{\text{SP}}$ and has been successfully used to efficiently apply hanging-node constraints. For more details, see the discussion on locally refined meshes and on an efficient way of applying hanging-node constraints in Sections 2.8 and 3.1.2 (Paper II).

BB3: restriction and constraints

The restriction operator \mathcal{R}_K is a Boolean matrix mapping local DoF indices to the global ones. Depending on the enumeration, only a few indices need to be stored per cell and the rest can be reconstructed. Constraints require that constraining indices and their weights need to be stored, implying a CRS format. For hanging-node constraints, this matrix can be decomposed, which allows to switch the data structures requiring only an additional single flag per cell.

2.4 Evaluation & application of test function

During evaluation with S_K , quantities, like values or gradients of u_K ,

$$(\hat{u}_K)_q = \sum_{0 \leq j < N_{\text{DoFs}}} \hat{\phi}_j(\hat{\mathbf{x}}_q) u_{K,j}, \quad (2.13a)$$

$$(\hat{\nabla} u_K)_q = \sum_{0 \leq j < N_{\text{DoFs}}} \hat{\nabla} \hat{\phi}_j(\hat{\mathbf{x}}_q) u_{K,j} \quad (2.13b)$$

need to be determined in reference space at the quadrature points. The values and, in each direction, the gradients of the shape functions can be precomputed and tabulated (N and D_x, D_y, D_z , respectively) at the quadrature points:

$$\begin{aligned} \hat{u}_K &= N u_K, \\ \hat{\nabla} \hat{u}_K &= \begin{pmatrix} D_x \\ D_y \\ D_z \end{pmatrix} u_K. \end{aligned}$$

Testing corresponds to the transpose of the evaluation:

$$\begin{aligned} y_K &= N^T \hat{y}_K, \\ y_K &= \begin{pmatrix} D_x \\ D_y \\ D_z \end{pmatrix}^T \begin{pmatrix} \hat{y}_K^x \\ \hat{y}_K^y \\ \hat{y}_K^z \end{pmatrix} = D_x^T \hat{y}_K^x + D_y^T \hat{y}_K^y + D_z^T \hat{y}_K^z. \end{aligned}$$

Due to this similarity, we skip the detailed discussion of testing.

In the following, we concentrate on tensor-product elements and quadrature rules. We discuss computations on non-tensor-product-shaped cells in Section 2.7 and non-tensor-product quadrature rules in Section 3.4.

Fig. 2.2 shows typical support points and evaluation/quadrature points. If the support points and the evaluation points are identical, they are collocated and corresponding interpolation steps become no-ops.

Note: In the following, all depictions are shown in 2D and the corresponding equations in 3D, indicating the dimension-independent nature of the algorithms, as exploited in Paper VI (see Section 3.2) to simulate PDEs up to 6D.

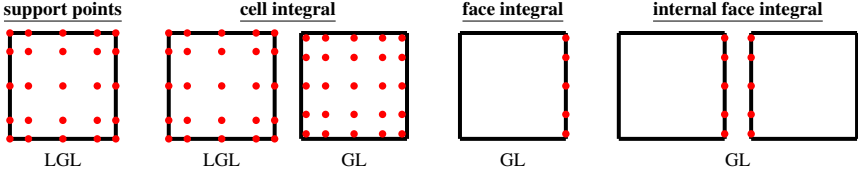


Figure 2.2. Illustration of support points and evaluation points during cell and face integrals on quadrilateral-shaped elements. We consider Legendre–Gauss–Lobatto (LGL) and Gauss–Legendre (GL) points.

Cell integrals: evaluation

For tensor-product elements, (2.13a) reads as

$$\begin{aligned}
 (\hat{u}_K)_q &= \sum_{0 \leq k < N_{\text{DoFs}}^{\text{1D}}} \sum_{0 \leq j < N_{\text{DoFs}}^{\text{1D}}} \sum_{0 \leq i < N_{\text{DoFs}}^{\text{1D}}} \hat{\phi}_k^{\text{1D}}(\hat{\mathbf{x}}_2^q) \hat{\phi}_j^{\text{1D}}(\hat{\mathbf{x}}_1^q) \hat{\phi}_i^{\text{1D}}(\hat{\mathbf{x}}_0^q) u_{K,ijk} \quad (2.14a) \\
 (\hat{u}_K)_q &= \sum_{0 \leq k < N_{\text{DoFs}}^{\text{1D}}} \hat{\phi}_k^{\text{1D}}(\hat{\mathbf{x}}_2^q) \sum_{0 \leq j < N_{\text{DoFs}}^{\text{1D}}} \hat{\phi}_j^{\text{1D}}(\hat{\mathbf{x}}_1^q) \sum_{0 \leq i < N_{\text{DoFs}}^{\text{1D}}} \hat{\phi}_i^{\text{1D}}(\hat{\mathbf{x}}_0^q) u_{K,ijk}, \quad (2.14b)
 \end{aligned}$$

using multiindices (i, j, k) . If the quadrature rule has a tensor-product structure as well, the equation becomes

$$(\hat{u}_K)_{\hat{i}\hat{j}\hat{k}} = \sum_{0 \leq k < N_{\text{DoFs}}^{\text{1D}}} \phi_k^{\text{1D}}(\hat{\mathbf{x}}_2^{\hat{k}}) \sum_{0 \leq j < N_{\text{DoFs}}^{\text{1D}}} \phi_j^{\text{1D}}(\hat{\mathbf{x}}_1^{\hat{j}}) \sum_{0 \leq i < N_{\text{DoFs}}^{\text{1D}}} \phi_i^{\text{1D}}(\hat{\mathbf{x}}_0^{\hat{i}}) u_{K,ijk},$$

using multiindices $(\hat{i}, \hat{j}, \hat{k})$ for the quadrature points. In tensor-product notation, the computation of all values can be written as

$$\hat{u}_K = Nu = (N_z^{\text{1D}} \otimes N_y^{\text{1D}} \otimes N_x^{\text{1D}}) u_K, \quad (2.15)$$

assuming a lexicographic ordering in u_K . This operation can be efficiently implemented via a sequence of 1D basis changes with the total cost $\mathcal{O}(d(k+1)^{d+1})$ [17, 26], since

$$\hat{u} = (N_z^{\text{1D}} \otimes I_y^{\text{1D}} \otimes I_x^{\text{1D}}) (I_z^{\text{1D}} \otimes N_y^{\text{1D}} \otimes I_x^{\text{1D}}) (I_z^{\text{1D}} \otimes I_y^{\text{1D}} \otimes N_x^{\text{1D}}) u.$$

Fig. 2.3 gives a geometric visualization of these basis-change steps. Similarly, the gradients can be computed as

$$\nabla \hat{u}_K = \begin{pmatrix} D_x \\ D_y \\ D_z \end{pmatrix} u_K = \begin{pmatrix} N_z^{\text{1D}} \otimes N_y^{\text{1D}} \otimes D_x^{\text{1D}} \\ N_z^{\text{1D}} \otimes D_y^{\text{1D}} \otimes N_x^{\text{1D}} \\ D_z^{\text{1D}} \otimes N_y^{\text{1D}} \otimes N_x^{\text{1D}} \end{pmatrix} u_K$$

if we exploit the tensor-product structure of the element and the quadrature rule. This indicates that the gradients can be computed in nine 1D basis-change steps. However, if a decomposition $D_{\square}^{\text{1D}} = \tilde{D}_{\square}^{\text{1D}} N_{\square}^{\text{1D}}$ is possible, one

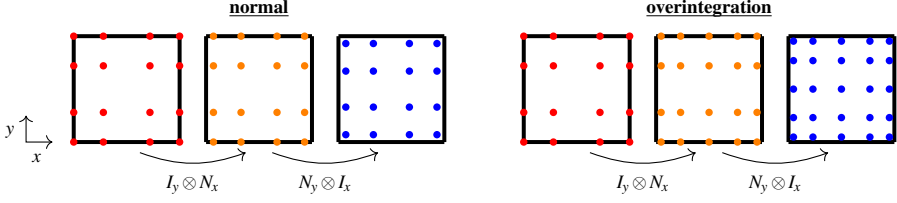


Figure 2.3. Evaluation steps for a cell in 2D.

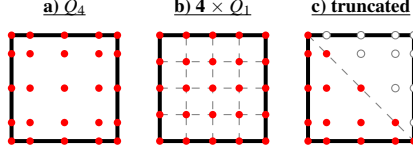


Figure 2.4. 2D element types with tensor-product structure.

can exploit the fact that $(AB) \otimes (CD) = (A \otimes C)(B \otimes D)$ [49, 68],

$$\nabla \hat{u}_K = \begin{pmatrix} I_z^{1D} \otimes I_y^{1D} \otimes \tilde{D}_x^{1D} \\ I_z^{1D} \otimes \tilde{D}_y^{1D} \otimes I_x^{1D} \\ \tilde{D}_z^{1D} \otimes I_y^{1D} \otimes I_x^{1D} \end{pmatrix} \underbrace{(N_z^{1D} \otimes N_y^{1D} \otimes N_x^{1D})}_{\text{Eq. (2.15)}} u_K \quad (2.16a)$$

$$= \begin{pmatrix} I_z^{1D} \otimes I_y^{1D} \otimes \tilde{D}_x^{1D} \\ I_z^{1D} \otimes \tilde{D}_y^{1D} \otimes I_x^{1D} \\ \tilde{D}_z^{1D} \otimes I_y^{1D} \otimes I_x^{1D} \end{pmatrix} \hat{u}_K, \quad (2.16b)$$

to reduce the number of 1D basis-change steps to 6 or to 3 in the case that the values are already evaluated at the quadrature points. This can have two reasons: the support points and the quadrature points are collocated or an evaluation has been performed with (2.15) beforehand.

The previous discussion has exposed the two-level nature of the evaluation operator \mathcal{S}_K . On the one hand, one needs efficient means of 1D basis transformations (2D: $I_y^{1D} \otimes A_x^{1D}$, $A_y^{1D} \otimes I_x^{1D}$; 3D: $I_z^{1D} \otimes I_y^{1D} \otimes A_x^{1D}$, $I_z^{1D} \otimes A_y^{1D} \otimes I_x^{1D}$, $A_z^{1D} \otimes I_y^{1D} \otimes I_x^{1D}$) and, on the other hand, the routines that run these 1D basis transformations in sequence with the right input/output, depending on what quantities are needed at the quadrature points and which shape-function/quadrature pair is given. The 1D basis transformation allows to further reduce the number of operations, depending on the structure of the 1D interpolation matrix, e.g., by an even-odd decomposition [49, 61].

The discussed algorithm works not only for tensor-product elements based on nodal polynomials, e.g., Lagrange polynomials, but also for piecewise linear polynomials defined on subcells (see Fig. 2.4b) and for non-nodal polynomials, e.g., Legendre or Hermite polynomials [1]. An extension to truncated tensor-product spaces (see Fig. 2.4c), e.g., to

$$\left\{ \xi_1^{i_1} \xi_2^{i_2} \xi_3^{i_3} \mid 0 \leq i_1 + i_2 + i_3 \leq k \right\} \subset \left\{ \xi_1^{i_1} \xi_2^{i_2} \xi_3^{i_3} \mid 0 \leq i_1, i_2, i_3 \leq k \right\},$$

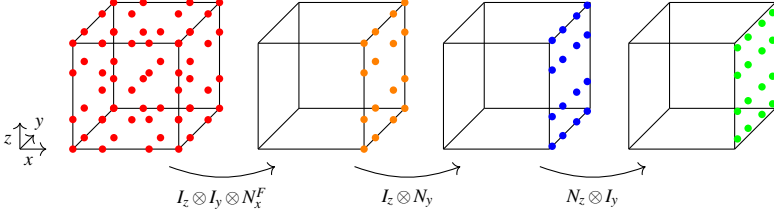


Figure 2.5. Evaluation steps on faces in 3D.

is also possible by zeroing the corresponding entries in the embedding space.

Face integrals: evaluation

For face integrals, values and gradients need to be evaluated at quadrature points positioned on the faces, and, as a result, they only have a tensor-product structure on the face. For example, to evaluate values at a quadrature point of the left or the right face F of a hexahedron, one needs to perform the following steps:

$$(\hat{u}_K^F)_{\hat{j},\hat{k}} = \sum_{0 \leq k < N_{\text{DoFs}}^{\text{1D}}} \hat{\phi}_k^{\text{1D}}(\hat{\mathbf{x}}_2^k) \sum_{0 \leq j < N_{\text{DoFs}}^{\text{1D}}} \hat{\phi}_j^{\text{1D}}(\hat{\mathbf{x}}_1^j) \sum_{0 \leq i < N_{\text{DoFs}}^{\text{1D}}} \hat{\phi}_i^{\text{1D}}(\hat{\mathbf{x}}_0) u_{K,i,j,k},$$

with $\hat{\mathbf{x}}_0 \in \{0, 1\}$. Tabulating the shape functions and writing the equation in tensor-product notation gives:

$$\hat{u}_K^F = (N_z^{\text{1D}} \otimes N_y^{\text{1D}} \otimes N_x^F) u = (N_z^{\text{1D}} \otimes N_y^{\text{1D}}) (I_z^{\text{1D}} \otimes I_y^{\text{1D}} \otimes N_x^F) u_K. \quad (2.17)$$

The latter notation implies that values are projected first onto the face before an in-face interpolation is performed (on a reduced data set). Similarly, one can write for the other directions:

$$\begin{aligned} \hat{u} &= (N_z^{\text{1D}} \otimes N_y^F \otimes N_x^{\text{1D}}) u = (N_z^{\text{1D}} \otimes N_x^{\text{1D}}) (I_z^{\text{1D}} \otimes N_y^F \otimes I_x^{\text{1D}}) u, \\ \hat{u} &= (N_y^F \otimes N_y^{\text{1D}} \otimes N_x^{\text{1D}}) u = (N_y^{\text{1D}} \otimes N_x^{\text{1D}}) (N_z^F \otimes I_y^{\text{1D}} \otimes I_x^{\text{1D}}) u. \end{aligned}$$

For the derivative on the left or right face, one gets:

$$\nabla \hat{u}_K^F = \begin{bmatrix} D_z^{\text{1D}} \otimes N_y^{\text{1D}} \otimes N_x^F \\ N_z^{\text{1D}} \otimes D_y^{\text{1D}} \otimes N_x^F \\ N_z^{\text{1D}} \otimes N_y^{\text{1D}} \otimes D_x^F \end{bmatrix} u_K = \begin{bmatrix} [D_z^{\text{1D}} \otimes N_y^{\text{1D}}] \\ [N_z^{\text{1D}} \otimes D_y^{\text{1D}}] \\ [N_z^{\text{1D}} \otimes N_y^{\text{1D}}] \end{bmatrix} \begin{bmatrix} [I_z^{\text{1D}} \otimes I_y^{\text{1D}} \otimes N_x^F] \\ [I_z^{\text{1D}} \otimes I_y^{\text{1D}} \otimes D_x^F] \end{bmatrix} u_K. \quad (2.18)$$

Again, the number of 1D basis-change steps can be reduced for a nodal basis and the collocation case (not shown). Furthermore, only the DoFs on the faces need to be gathered for computing the values, while, for computing the gradients, all DoFs of a cell. To address the latter issue, we introduced in [1]

a Hermite-like basis and showed that, with this basis, only two layers of DoFs are needed to compute the gradients.

Similarly to (2.16b), there are instances where the values are already computed at the quadrature points of a cell. For this purpose, we use the decomposition $N_x^F = \tilde{N}_x^F N_x^{1D}$:

$$\hat{u}_K^F = (I_z^{1D} \otimes I_y^{1D} \otimes \tilde{N}_x^F) \underbrace{(N_z^{1D} \otimes N_y^{1D} \otimes N_x^{1D})}_{\text{Eq. (2.15)}} u = (I_z^{1D} \otimes I_y^{1D} \otimes \tilde{N}_x^F) \hat{u}.$$

Comparing this equation with (2.17) suggests that interleaving cell and face integrals allows to reduce the number of 1D interpolations and, as a consequence, the amount of computations.

More details on face integrals are given in Section 2.6 for discontinuous Galerkin methods.

Multiple components

The extension of the two-level evaluation algorithms to the context of multicomponent FEM is straightforward if, e.g., vectorial Lagrange elements, where $N_c = I_c \otimes N$ with the identity matrix $I_c = \mathbb{R}^{c \times c}$, are used. Paper VII (see Section 3.3) discusses the application to very high number of components that varies over time.

This approach can be applied with slight modifications in the context of more involved vectorial elements, like H(div) and H(curl) elements, as demonstrated by Niklas Fehn, Katharina Kormann, Martin Kronbichler, and Niklas Wik and documented in [69].

BB4: two-level evaluation

The evaluation operator S_K computes, i.a., values and gradients at the quadrature points in reference space. For this purpose, it can exploit the structures of the finite element and the quadrature rule. For the evaluation of, e.g., values, the following sequence of 1D basis transformations can be used:

$$v_K = Nu = (N_z^{1D} \otimes N_y^{1D} \otimes N_x^{1D}) u_K.$$

Each 1D basis transformation of type $v_K = I_z^{1D} \otimes I_y^{1D} \otimes A_x^{1D} u_K$ can be further optimized.

BB5: face evaluation

In the case of tensor-product elements, it is beneficial to project values and gradients to the faces via

$$v_K = (I_z \otimes I_y \otimes N_x^F) u_K$$

and perform all operations considered in BB4 in face.

2.5 Quadrature operation

The quadrature operation \mathcal{Q}_K takes computed values and gradients and prepares the values and gradients to be integrated and tested. In the case of term $*$ of the advection equation (2.8),

$$(v, -\beta(\mathbf{a} \cdot \nabla u))_{\Omega^{(e)}} + (\nabla v, (1 - \beta)\mathbf{a}u)_{\Omega^{(e)}},$$

this looks like

$$\begin{aligned} \begin{pmatrix} \hat{u}_q \\ \nabla \hat{u}_q \end{pmatrix} &= \underbrace{\sum_{j=0}^{N_{\text{DoFs}}} \begin{pmatrix} \hat{\phi}_j(x_q) \\ \nabla \hat{\phi}_j(x_q) \end{pmatrix}}_{\text{evaluation}} u_{K,j} \quad \forall q \\ v_{K,i} &= \underbrace{\sum_{q=0}^{N_q} \begin{pmatrix} \hat{\phi}_i(x_q) \\ \nabla \hat{\phi}_i(x_q) \end{pmatrix}}_{\text{integration \& testing}} \cdot \underbrace{[|J_K(\hat{\mathbf{x}}_q)|w_q]}_{\text{q-operation}} \begin{pmatrix} -\beta \mathbf{a} \cdot \nabla \hat{u}_q \\ (1 - \beta)\mathbf{a}\hat{u}_q \end{pmatrix}, \end{aligned}$$

with the quadrature operation being the focus of this section. As discussed before, the evaluation operation \mathcal{S}_K only works in reference space and, as a consequence, the derived quantities need to be corrected according to the mapping by \mathcal{Q}_K :

$$\begin{aligned} \begin{pmatrix} \hat{u}_q \\ \hat{\nabla} \hat{u}_q \end{pmatrix} &= \underbrace{\sum_{j=0}^{N_{\text{DoFs}}} \begin{pmatrix} \hat{\phi}_j(\hat{\mathbf{x}}_q) \\ \hat{\nabla} \hat{\phi}_j(\hat{\mathbf{x}}_q) \end{pmatrix}}_{\text{evaluation} \rightarrow \mathcal{S}_K} u_{K,j} \quad \forall q \\ v_{K,i} &= \underbrace{\sum_{q=0}^{N_q} \begin{pmatrix} \hat{\phi}_i(\hat{\mathbf{x}}_q) \\ \hat{\nabla} \hat{\phi}_i(\hat{\mathbf{x}}_q) \end{pmatrix}}_{\text{sum \& testing} \rightarrow \mathcal{S}_K^T} \cdot \underbrace{|J_K(\hat{\mathbf{x}}_q)|w_q}_{\text{q-operation} \rightarrow \mathcal{Q}_K} \begin{pmatrix} -\beta \mathbf{a} \cdot J_K^{-T}(\hat{\mathbf{x}}_q) \hat{\nabla} \hat{u}_q \\ (1 - \beta)J_K^{-1}(\hat{\mathbf{x}}_q)\mathbf{a}\hat{u}_q \end{pmatrix}. \end{aligned}$$

In the case of Lagrange elements, only the gradient has to be multiplied by the Jacobian. Note that we moved the multiplication with the term $|J_K(\hat{\mathbf{x}}_q)|w_q$, corresponding to the quadrature and its weighting, inside the quadrature operation, since \mathcal{S}_K^T implicitly only performs the summation during testing.

The advection velocity \mathbf{a} in the equation above might be a constant in the most basic case. Alternatively, it could be a function of the position of the quadrature point and/or the solution, $\mathbf{a}(x_q, u_q, \nabla u_q, \dots)$, implying that also the real position of the quadrature points needs to be queried. Furthermore, \mathbf{a} might be related to a solution vector, which has to be evaluated at the quadrature points. Such a variable/vector is also referred to as *passive variable* in the literature [53], since it does not change during the current cell loop. Similarly, $J_K(\hat{\mathbf{x}}_q)$, $|J_K(\hat{\mathbf{x}}_q)|$, and $\hat{\mathbf{x}}_q$ can be regarded as passive variables as well. Other examples of passive variables include linearization points in the case of nonlinear solvers and variable-viscosity terms.

Depending on the context, there are different storage formats for such quantities. If the data is related to a solution vector, one can gather the values from a global vector, followed by a basis change to the quadrature points. Alternatively, the data can be stored cell by cell. For this, there are two options: to store the data in the original (discontinuous) basis or to precompute them on the quadrature-point level.

BB6: passive variables

On the quadrature-point level, one needs controlled access to passive variables, whose data structures might differ. The most prominent passive variables are the quantities related to mapping, i.e., the Jacobian, and to the linearization point in the context of a nonlinear setting.

2.6 Discontinuous Galerkin methods

In the context of DG [66, 70–73], one needs to evaluate fluxes at each internal face of a cell, as shown in Section 2.1. Examples are term \diamond in (2.3) and term \dagger in (2.5c). The evaluation points on the positive side of inner faces are given by (2.5b). On unstructured, conformal meshes, this set of evaluation points still has a tensor-product structure, however, its entries might be located on any face of the neighboring cell and might have a different ordering compared to the standard ordering. To be able to use the standard (tensor-product) face evaluation routines (2.17) and (2.18), one needs to determine, in a preprocessing step, the correct *face number* and *orientation* [66], based on the topology of the mesh, and, in a postprocessing step, one needs to correct the computed values/gradients at the determined face according to the orientation.

Processing cell by cell [70], i.e., the cell integral and all face integrals of a cell in one go, seems to be the natural choice. It has multiple advantages. On the one hand, operations of the cell and face integrals can be reused; e.g., for the cell integral, the values have to be computed at the quadrature points, which can be projected simply onto all faces, as indicated by (2.17). On the other hand, the destination vector is written only once, making this approach more cache-efficient and particularly interesting for shared-memory computations due to the lack of race conditions, as discussed in Paper VI (see Section 3.2). A challenge from a software point of view is that cell and face evaluations need to be interleaved.

As an alternative, one can split up the cell and face integrals into separate loops [70]. In this case, it is natural to compute the face integral on both sides

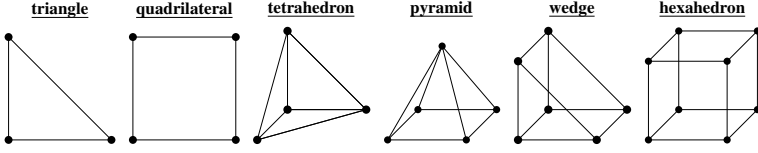


Figure 2.6. 2D and 3D reference cells.

of a face in one go, e.g.,

$$v_i = \sum_{0 \leq q < N_q} \begin{pmatrix} \hat{\phi}_i(\hat{\mathbf{x}}_q) \\ \hat{\nabla} \hat{\phi}_i(\hat{\mathbf{x}}_q) \\ \hat{\phi}_i^+(\hat{\mathbf{x}}_q^+) \\ \hat{\nabla} \hat{\phi}_i^+(\hat{\mathbf{x}}_q^+) \end{pmatrix} \cdot |J_K(\hat{\mathbf{x}}_q)| w_q \begin{pmatrix} -0.5(J_K^{-T}(\hat{\mathbf{x}}_q) \hat{\nabla} \hat{u}_q + J_K^{-T}(\hat{\mathbf{x}}_q^+) \hat{\nabla} \hat{u}_q^+) \cdot \mathbf{n} + \tau(\hat{u}_q - \hat{u}_q^+) \\ -0.5(\hat{u}_q - \hat{u}_q^+) \mathbf{n} \\ +0.5(J_K^{-T}(\hat{\mathbf{x}}_q) \hat{\nabla} \hat{u}_q - J_K^{-T}(\hat{\mathbf{x}}_q^+) \hat{\nabla} \hat{u}_q^+) \cdot \mathbf{n} + \tau(\hat{u}_q - \hat{u}_q^+) \\ -0.5(\hat{u}_q - \hat{u}_q^+) \mathbf{n} \end{pmatrix},$$

extending (2.5c) by testing with the test functions of the cell on the positive side. As the colors indicate, many of the computations on the quadrature-point level can be reused due to the symmetry of the fluxes, allowing to reduce the computational complexity in this case as well. Similarly to the vector-update operations in (2.12), cell loops and (inner/boundary) face loops can be interleaved to efficiently exploit caches of modern hardware.

2.7 Arbitrary shapes: simplex and mixed meshes

For non-tensor-product shape function/quadrature pairs, matrix-free operator evaluation (2.11) is also possible; however, the derivation of efficient evaluation kernels (\mathcal{S}_K) is not straightforward or even not feasible. Such cases include, i.a., cells with the shape of simplices, wedges, or pyramids (see Fig. 2.6). In the worst case, matrices, like N , D_x , D_y , D_z , have to be tabulated and applied directly. The costs of application are $\mathcal{O}(k^{2d})$.

Moxey et al. [74] investigated efficient high-order finite-element evaluation for simplices, pyramids, and wedges. The trick the authors use is 1) to define tensor-product quadrature rules on $[-1, 1]^d$, 2) to map these, via Duffy mapping, to a simplicial domain (which is mapped in the regular way to real coordinates, requiring chaining of Jacobians), and 3) to use polynomials that allow for sum factorization. In the context of tetrahedra, the Duffy mapping [75] from a tetrahedral domain to a hexahedral domain for points with

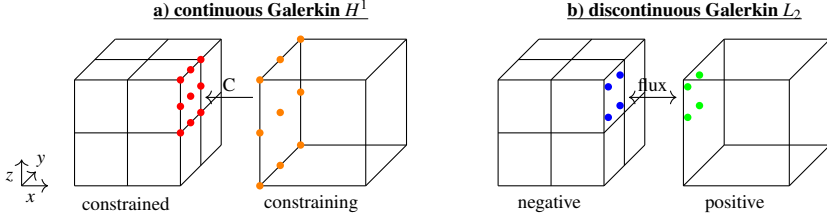


Figure 2.7. Resolving hanging-node constraints in the continuous Galerkin case and interpolation to subfaces in the discontinuous Galerkin case.

$-1 < \eta_1, \eta_2, \eta_3 < 1$ is given as

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

and the polynomial ansatz is

$$\hat{u}(\eta) = \sum_{p=0}^P \phi_p^a(\eta_1(\xi_1, \xi_2, \xi_3)) \sum_{q=0}^{Q-p} \phi_{pq}^b(\eta_2(\xi_2, \xi_3)) \sum_{r=0}^{R-p-q} \phi_{pqr}^c(\eta_3(\xi_3)) \hat{u}_{pqr},$$

allowing sum factorization. The presented results are promising, however, indicate a slowdown by up to a factor of 10 compared to the fully tensor-product case.

Note: At the time of writing this thesis, deal.II had an experimental support for simplices, wedges, and pyramids, using full interpolation matrices limiting the throughput. The implementation was proposed and implemented by the author of this thesis [7, 9].

2.8 Local refinement

In the following, we are considering locally refined meshes in which quadrilateral cells and hexahedral cells are replaced by four or eight children cells, resulting in *hanging nodes*. Depending on whether continuous or discontinuous Galerkin methods are applied, different steps are needed to deal with this issue.

In the case of continuous Galerkin methods, the DoFs constrained on faces or edges need to be computed by resolving hanging-node constraints [76], which corresponds to an interpolation of the DoFs on the coarse side to subfaces/subedges, as shown in Fig. 2.7a). This operation can be performed as an inplace interpolation if constrained DoFs are replaced by constraining ones during gathering. More details on such a special-purpose operator $\mathcal{C}_K^{\text{HN}}$, for which refinement configurations have to be en- and decoded, are given in Paper II (see Section 3.1.2).

In the case of DG, hanging-node constraints do not have to be resolved, but values need to be evaluated on the coarse (positive) side at the quadrature

points prescribed by faces on the fine (negative) side, as shown in Fig. 2.7b). This can be done in two steps, similarly as in (2.17): first, the values are projected to the face and, then, interpolated to the quadrature points on the correct subface.

BB7: local mesh refinement

Locally refined meshes require interpolations to subfaces or subedges. While, in the case of continuous Galerkin methods, DoFs on subfaces or subedges are replaced by the computed ones, these are used directly as values on the positive side for the evaluation of fluxes in the case of DG.

2.9 Example implementation & interface

In the following, we describe the `MatrixFree` module of `deal.II`, version 9.5 [12]. We start with presenting the user interface, continue with discussing implementation aspects, and list its relevant features. Finally, we compare the user interface of `deal.II` to the one of `libCEED`.

User Interface

Listings 2.1, 2.4, and 2.5 present example codes using the matrix-free module of `deal.II`.

In Listing 2.1, the setup of a `MatrixFree` object is shown. It is initialized with a mapping, a set of finite-element spaces (defined by a mesh with a finite element, `dof_handlers`, and the constraints), and a set of quadratures. This information is enough to set up cache-friendly loops over cells and the indices of the restriction operator as well as to precompute the mapping metrics. The `MatrixFree` object can be furthermore configured, e.g., with regard to what types of mapping metric and threading are needed.

The code snippet also shows how a `MatrixFree` object can be used to loop over cells as well as over internal and boundary faces. Users provide callback functions containing the implementation of the integrals to the functions `cell_loop()` or `loop()`. These functions call, on the one hand, the callback function and, on the other hand, are responsible for communication and for optional vector updates. Listing 2.2 presents a naive, sequential implementation of these steps. Listing 2.3 shows a possible implementation of BB2, which interleaves cell loops with face loops, communication, and vector updates.

Listing 2.4 and 2.5 present the callback functions related to cell integrals in the case of a Poisson operator (2.1) and an advection operator (2.8). These callback functions are called on ranges of cell batches to decrease overhead due to function calls and setup. Within the callback function, the user loops over the specified cell batches and applies the helper class `FEEvaluation` for gathering/scattering ($\mathcal{R}_K/\mathcal{R}_K^T$ with application of constraints) and evaluation/testing ($\mathcal{S}_K/\mathcal{S}_K^T$) on cell level as well as for operations on the quadrature-

Listing 2.1. Basic setup and loop capabilities of the MatrixFree object.

```
// setup
MatrixFree<dim, Number> matrix_free;

typename::MatrixFree<dim, Number> setting; // configuration
settings.mapping_update_flags = update_gradients|update_JxW_values;

matrix_free.reinit(mapping, dof_handlers, constraints,
                  quadratures, setting);

// loop over cells
matrix_free.cell_loop(cell_op, dst, src);

// loop over cells and faces
matrix_free.loop(cell_op, face_op, boundary_op, dst, src);
```

Listing 2.2. Naive implementation of MatrixFree::loop(). Integrals, vector updates, and communication are executed in sequence.

```
// pre operation
pre_operation(0, vector_size);

// communication (update ghost values)
src.update_ghost_values();
dst.zero_out_ghost_values();

// loop over cells and faces
cell_integral(matrix_free, dst, src, {0, n_cells});
face_integral(matrix_free, dst, src, {0, n_faces});
boundary_integral(matrix_free, dst, src, {0, n_boundary_faces});

// communication (collect partial results and clean up)
src.zero_out_ghost_values();
dst.compress(VectorOperation::add);

// post operation
post_operation(0, vector_size);
```

point level (\mathcal{Q}_K). In the case of the latter, the user works in real space with FEEvaluation being responsible for correcting by metric data the gradients evaluated in reference space during \mathcal{S}_K . Here, one can exploit the information whether a cell is Cartesian, affine or arbitrarily deformed, to reduce the number of operations.

For the sake of brevity, we do not show the callback function related to face integrals. In this case, FEFaceEvaluation, the face equivalent of FEEvaluation, is used, which can be initialized for negative and positive sides of a face and also works on subfaces.

Listing 2.3. Simplified implementation of the optimized version of `MatrixFree::loop()` in `deal.II`.

```
// pre operation on DoFs to be communicated
funct.cell_loop_pre_range(part_index[part_index.size() - 2]);

funct.vector_update_ghosts_start(); // start ghost update

for (unsigned int part = 0; part < part_index.size() - 2; ++part)
{
    if (part == 1)
        funct.vector_update_ghosts_finish(); // finish ghost update

    for (auto i = part_index[part]; i < part_index[part + 1]; ++i)
    {
        funct.cell_loop_pre_range(i); // pre operation
        funct.cell(i); // cell integral
        funct.face(i); // face integral
        funct.boundary(i); // boundary integral
        funct.cell_loop_post_range(i); // post operation
    }

    if (part == 1)
        funct.vector_compress_start(); // start compression
}

funct.vector_compress_finish(); // finish compression

// post operation on DoFs on remote dependencies
funct.cell_loop_post_range(part_index[part_index.size() - 2]);
```

As a summary, we discuss how the tasks in the matrix-free interface are structured. The `MatrixFree` object extracts relevant information from `deal.II` data structures, stores it internally for efficient gathering/scattering, evaluation of constraints, and application of metric data as well as loops over cells/faces in a parallel and cache-efficient way. The `FEEvaluation` and `FEFaceEvaluation` classes provide structured access to information on the cell and the quadrature-point levels and, in particular, apply the metric data. Fig. 2.8 summarizes the user interface with a UML diagram.

Separation of concerns

Within the `MatrixFree` object, data is stored 1) regarding efficient restriction (incl. constraints and information on tasking and efficient interleaving of integrals with vector updates and communication), 2) regarding tabulated shape values, 3) regarding mapping at the quadrature points, and 4) optionally, regarding face definitions. Each of these concerns is separated in the following classes: `TaskInfo`, `DoFInfo`, `ShapeInfo`, `MappingInfo`, and `FaceInfo`, as

Listing 2.4. Cell integral of Poisson operator (2.1).

```
const auto cell_integral =
  [](const auto &data, auto &dst, const auto &src, const auto range){
    FEEvaluation<dim, degree, n_points_1d, 1> phi(data);

    for (auto cell = range.first; cell < range.second; ++cell)
    {
      phi.reinit(cell);

      // gather from source vector and evaluate gradients
      phi.gather_evaluate(src, EvaluationFlags::gradients);

      // loop over quadrature points
      for (unsigned int q = 0; q < phi.n_q_points; ++q)
      {
        // get evaluated quantities in real space
        const auto gradient = phi.get_gradient(q);

        // preparation for testing
        phi.submit_gradient(gradient, q);
      }

      // integrate and scatter back into destination vector
      phi.integrate_scatter(EvaluationFlags::gradients, dst);
    }
  };
```

shown in Fig. 2.8. The instances of these are stored within `MatrixFree` and are accessed via `FEEvaluation/FEFaceEvaluation`.

The classes `FEEvaluation` and `FEFaceEvaluation` give the impression to be working in real space, by applying the mapping metrics to the reference cell data during the `get` and `submit` functions. The actual evaluation/testing is forwarded to two internal functions that, optionally, call tensor-product kernels in a two-level-evaluation way (BB4).

Vectorization

The matrix-free infrastructure vectorizes over cells and faces to increase the throughput, by explicitly exploiting SIMD (single instruction, multiple data) capabilities of modern processors. The maximum number of cells processed together is determined by the SIMD capabilities of the given hardware. Each lane of a vector register is assigned to one cell/face. This implies, e.g., that, in the case of double precision and AVX2, 4 cells are processed in one go and, in the case of AVX-512, 8 cells.

Listing 2.5. Cell integral of advection operator (2.8).

```
const auto cell_integral =
  [](const auto &data, auto &dst, const auto &src, const auto range){
    FEEvaluation<dim, degree, n_points_1d, 1> phi(data);

    for (auto cell = range.first; cell < range.second; ++cell)
    {
      phi.reinit(cell);

      // gather from source vector and evaluate gradients
      phi.gather_evaluate(
        src, EvaluationFlags::gradients | EvaluationFlags::value);

      // loop over quadrature points
      for (unsigned int q = 0; q < phi.n_q_points; ++q)
      {
        // get evaluated quantities in real space
        const auto value = phi.get_value(q);
        const auto gradient = phi.get_gradient(q);

        // preparation for testing
        phi.submit_value(beta * a * gradient, q);
        phi.submit_gradient((1 - beta) * a * value, q);
      }

      // integrate and scatter back into destination vector
      phi.integrate_scatter(
        EvaluationFlags::gradients | EvaluationFlags::value, dst);
    }
  };
```

To ease usage, users do not work with plain numbers but with the wrapper class `VectorizedArray<Number, N>`², which overloads typical arithmetic functions and passes the function calls to appropriate intrinsics calls.

Working on individual lanes is rarely needed but can be done by accessing individual vector entries. Alternatively, masked operations or categorization (see below) are possible to allow diverging code paths.

Templating

The matrix-free infrastructure of `deal.II` is heavily templated. In particular, the degrees and the numbers of quadrature points are template arguments to guarantee fixed loop bounds during basis changes [49]. Dealing with these template arguments in user codes might be a burden if they are not known at compilation time or many different template arguments of these are needed,

²This class is conceptually similar to `std::simd`, which will be introduced as part of the C++23 standard [7]. Preliminary experiments showed that the given wrapper class in `deal.II` can be replaced by `std::simd` in future without any performance degradation.

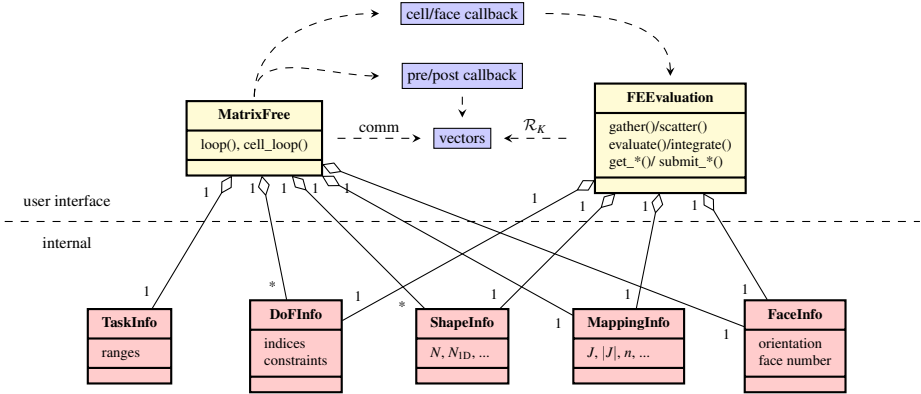


Figure 2.8. Relation of public and internal classes within the matrix-free infrastructure of deal.II. The MatrixFree object loops over cells/faces, performs communication, and executes the callback functions for integrals and vector updates. It also stores data, which can be accessed by users during integrals via FEEvaluation and FEFaceEvaluation (not shown).

e.g., in the instance of p -multigrid or hp -adaptivity. Therefore, deal.II provides factory classes that internally convert, with a minimal overhead, non-templated degrees and numbers of quadrature points to templated ones and dispatch the work to the right precompiled functions.

Furthermore, the dimension and the number of components are templated. These two arguments determine the return type of functions of the classes FEEvaluation and FEFaceEvaluation and, as a consequence, allow object-orientated programming with function overloads for specialized tensor operations. Inside the library, the data is stored with plain numbers so that the template arguments are not passed too deep into the library, which would make the compilation too expensive.

Categorization: mixed meshes & hp -adaptivity

The infrastructure supports, in addition to hypercube meshes, both simplex and mixed meshes. In the latter case, $\tilde{S}_K^T \circ Q_K \circ S_K$ differs depending on the shape of the cell due to different numbers of DoFs, different numbers of quadrature points, and different tabulated shape functions in S_K . In these instances, cells are categorized according to their shape. Only cells with the same category are assigned to the same cell batch, allowing easy vectorization.

Similarly, the cell operations $\tilde{S}_K^T \circ Q_K \circ S_K$ might differ in the case of hp -adaptivity [77] even for the same cell shapes but for different degrees. The categorization approach discussed for mixed meshes also works in this context.

The categorization capabilities can be also applied for user categories if different operations should be performed on different cells. For very dynamic

systems, it is not feasible to setup the internal data structures after each recategorization. In this case, user cell batches can be created from existing ones, by modifying the gathering/scattering operations. Furthermore, it is possible to activate or deactivate cells, for which the cell operations would result in no-ops.

Comparison with libCEED

We conclude this section with a comparison of deal.II’s matrix-free framework with the C-based library libCEED [53]³. Although both have similar functionalities, they take, conceptually, very different approaches. One of the focuses of libCEED is the hardware portability, a topic we will ignore in the following.

The main task of libCEED is:

$$\mathbf{v} = \mathcal{A}(\mathbf{x}) = \sum_K \overbrace{\mathcal{R}_K^T \circ \mathcal{S}_K^T}^{\text{config.}} \circ \overbrace{\mathcal{Q}_K}^{\text{user}} \circ \overbrace{\mathcal{S}_K \circ \mathcal{R}_K}^{\text{config.}} \mathbf{x}$$

to 1) loop over objects K , 2) provide read/write access to arbitrary number of (active/passive) vectors via \mathcal{R}^T , 3) perform evaluation/testing, which can be configured at runtime, and 4) call user functions that are called on cell level and are supposed to iterate over the quadrature points. libCEED has no native support for parallelization; however, it allows to start matrix-free loops asynchronously. Furthermore, providing mapping data is not of concern of this library and needs to be treated as a passive variable by the user. Applying constraints lies also in the responsibility of the user.

Conceptually, it is very interesting that libCEED is not looping over cells but over arbitrary objects defined by user-provided index sets in \mathcal{R}_K . This conceptual separation is quite useful, given that one wants to loop over a subset of cells or over patches. For such purposes, one needs to use, at the time of writing this thesis, internal data structures directly in deal.II.

In summary, libCEED misses certain FEM infrastructure (constraints, mapping, parallelization) and operates on a much lower abstraction level than deal.II. Users do not provide to libCEED a discretized system, which is used to set up internal data structures, like DoFInfo and ShapeInfo in deal.II, but they themselves set up the “internal” data structures and provide these to libCEED.

2.10 Performance aspects

Matrix-free computations replace access to slow main memory by redundant computations on cached data. As a consequence, it is crucial to efficiently use

³<https://libceed.org>

the compute units by accessing data close in the memory hierarchy. Different stages of (2.11) related to cells and faces have different requirements for active working sets. For instance, \mathcal{R}_K accesses data in global vectors. The data might have been already accessed by neighboring cells, e.g., since they are, in the continuous case, shared or they have been needed for the computation of the fluxes. Appropriate cell orders and DoF enumerations increase the probability that the data is still in the caches. \mathcal{S}_K operates on the loaded data of the size $\mathcal{O}(c(k+1)^d)$, with c being the number of components, and computes derived quantities in reference space of the size $\mathcal{O}(c(d+1)p^d)$, with p being the number of quadrature points in each direction and under the assumption that both values and gradients are needed. In the tensor-product case, the derived quantities are computed by performing 1D basis changes, which correspond to multiple *sweeps* through the buffered data of the size $\mathcal{O}(\max(k+1, p)^d)$. During \mathcal{Q}_K , the derived quantities and, additionally, passive variables are accessed. Obviously, with increasing degree k , increasing number of quadrature points p (overintegration [35]), increasing dimension d , as discussed in Paper VI (see Section 3.2), and increasing number of components c , as considered in Paper VII (see Section 3.3), the caching of data on the higher levels of the caches becomes challenging so that data needs to be loaded from slower parts of the memory. In the worst case, each sweep of the 1D kernels drops out of cache. Vectorization over elements increases the pressure on the caches, since not data of one cell but data of multiple cells need to be cached.

Developments of matrix-free computations mainly focus on the optimization of the evaluation step \mathcal{S}_K , which has the complexity $\mathcal{O}(d(k+1)^{d+1})$ for tensor-product elements, implying an arithmetic intensity of $\mathcal{O}(k)$. Less emphasis is laid on the operation on the quadrature-point level \mathcal{Q}_K due to a favorable computation complexity. For Poisson operators on Cartesian meshes, the number of floating-point operations is rather low and, apart from the Jacobian, no passive variables need to be accessed, while, in other instances, this is not the case. Davydov et al. [36] investigated, for example, computational solid mechanics with hyperelastic materials and looked into the trade-off of recomputing information in comparison to computing wisely chosen information once and storing it as passive variable, which is accessed during cell loops. The results indicate that a compromise between them gave the best performance by computing intermediate quantities, which allowed to reduce not only the memory access but also the number of computations. We made similar observations in Paper VII (see Section 3.3), where the immense costs of the work on the quadrature-point level required a thorough optimization by directly applying material kernels to vectors (in a matrix-free way, also on the quadrature-point level). Such optimizations are very much problem- and hardware-specific and might be not possible in certain cases.

Communication for updating ghost values and collecting partial results also influences the performance of matrix-free cell loops. However, since communication is needed independently of whether operators are evaluated in a

matrix-based or matrix-free way, we put less emphasis on this topic in the following. Communication can be overlapped with computation, and the amount of data to be communicated can be reduced by using shared memory (see Sections 2.2 and 2.3) and exploiting the characteristics of the element type [49].

2.11 Variations: interleaving on cell level

Some of the potential performance issues listed above can be addressed by variations of the matrix-free implementation discussed till now, by explicitly exploiting its structure and dependencies on the cell level. In the following, we drop the subscript K for the sake of simplicity.

In (2.11), gathering and evaluation as well as integration and scattering are performed in sequence, respectively. However, in the case of tensor-product elements, it is possible to merge gathering with the interpolation in x -direction:

$$S \circ G = S_z \circ S_y \circ S_x \circ G = S_z \circ S_y \circ (S_x \circ G).$$

In this case, the basis change along the line can be performed once its values have been read:

$$\tilde{v}(:, j, k) = N_x u_e(:, j, k).$$

This allows to overlap slow memory access with computations. However, in the presence of constraints, particularly hanging-node constraints, this optimization becomes nontrivial, potentially requiring the tagging of cells.

In (2.11), evaluation, loop over the quadrature points, and integration are performed in sequence. This results in multiple sweeps over data and, potentially, in the generation of non-negligible intermediate data. However, an interleaving is possible. For a mass-matrix operator (2.6),

$$v_e = \mathcal{M}_e(u_e) = S_x^T \circ S_y^T \circ \underbrace{S_z^T \circ Q \circ S_z \circ S_y \circ S_x}_{z\text{-line}} u_e,$$

the underbraced term can be performed line by line [78]:

$$\tilde{v}(i, j, :) = N_z^T Q(i, j, :) N_z \tilde{u}(i, j, :).$$

In the case of a Poisson operator (2.1), one can not interleave on lines but on planes (Algorithm 3 in [49]):

$$\begin{aligned} v_e = \mathcal{L}_e(u_e) &= S_x^T \circ S_y^T \circ S_z^T \circ \begin{pmatrix} \mathcal{D}_x \\ \mathcal{D}_y \\ \mathcal{D}_z \end{pmatrix}^T \circ Q \circ \begin{pmatrix} \mathcal{D}_x \\ \mathcal{D}_y \\ \mathcal{D}_z \end{pmatrix} \circ S_z \circ S_y \circ S_x \\ &= S_x^T \circ S_y^T \circ \begin{pmatrix} \mathcal{S}_z \\ \mathcal{D}_z \end{pmatrix}^T \circ \underbrace{\begin{pmatrix} \begin{pmatrix} \mathcal{D}_x \\ \mathcal{D}_y \\ \mathcal{I}_z \end{pmatrix} \\ \begin{pmatrix} \mathcal{D}_x \\ \mathcal{D}_y \\ \mathcal{I}_z \end{pmatrix} \end{pmatrix}^T \circ Q \circ \begin{pmatrix} \begin{pmatrix} \mathcal{D}_x \\ \mathcal{D}_y \\ \mathcal{I}_z \end{pmatrix} \\ \begin{pmatrix} \mathcal{D}_x \\ \mathcal{D}_y \\ \mathcal{I}_z \end{pmatrix} \end{pmatrix} \circ \begin{pmatrix} \mathcal{S}_z \\ \mathcal{D}_z \end{pmatrix} \circ S_y \circ S_x. \end{aligned}$$

This optimization is also applicable for a Helmholtz operator.

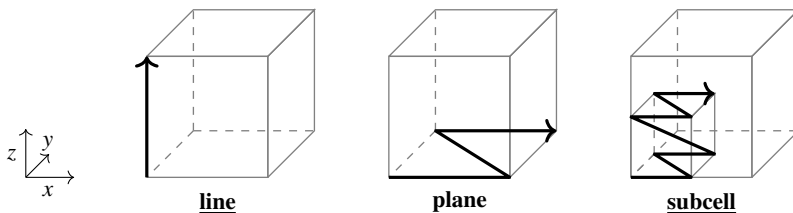


Figure 2.9. Different non-standard ways to loop over quadrature points in 3D.

In the case of piecewise shape functions on subcells, the latter can be processed independently, resulting in a natural 3D blocking. Within each cell, optimizations presented above can be used depending on the operator.

As a summary, Fig. 2.9 shows how evaluation, loop over quadrature points, and interpolation can be interleaved. It is clear that one, in none of the cases, loops over all quadrature points in one go and the sequence of looping is not guaranteed to be lexicographic. This implies a user interface that only specifies the operation at a quadrature point, which is called for arbitrary quadrature-point indices. One should note that we have presented optimizations for commonly used operators above. However, one can construct other examples with arbitrary number of passive variables (e.g., position vectors for mapping on the fly), which might also need a basis change. Writing variants for all possible versions by hand is not feasible and, therefore, might require code generation.

BB8: quadrature-point-centric operations

Depending on the weak form, \mathcal{R}_K , \mathcal{S}_K , and \mathcal{Q}_K need to be interleaved differently to obtain optimal performance. This might involve that quadrature points are processed in different orders, favoring a quadrature-point-centric view,

$$(v, \nabla v, \dots) \leftarrow \mathcal{Q}_K(q, u, \nabla u, \dots),$$

with an appropriate callback function.

2.12 Preconditioning

There are different ways of constructing a preconditioner $P^{-1} \approx A^{-1}$ for matrix-free operators. For this purpose, one can exploit 1) the fact that matrix representation can be generated from matrix-free operators and 2) the fact that one has access to the underlying mesh. The development of preconditioners for matrix-free operators is an active research area, and we discuss our contributions to this topic in Section 3.

Algebraic approaches

The explicit representation of the linear operator in the form of a matrix of a matrix-free operator \mathcal{A} can be obtained column by column by the application of unit vectors e_i : $A(:, i) = \mathcal{A}e_i$. To reduce the costs, one can exploit the fact that the sparsity pattern of the matrix is sparse to compute multiple columns of the matrix that have distinct non-zero row entries. Alternatively, one can compute the matrices on the cell level, $A_K(:, i) = \mathcal{A}_K e_i$, and assemble the results in the standard FEM way. This approach is applicable to generate blocks (e.g., the block diagonal in the case of DG) or the diagonal of the linear operator. Given these matrix representations, widely used algebraic preconditioners can be set up, most notably, algebraic multigrid (AMG), incomplete LU factorization (ILU), Jacobi and Chebyshev preconditioners.

Furthermore, additive Schwarz methods (ASM) are possible. They have the structure

$$v = P^{-1}u = \sum R_K^T W_K \hat{A}_K^{-1} R_K u \quad \text{with} \quad \hat{A}_K = R_K A R_K^T \quad (2.19)$$

and the weighting matrix W_K . One issue with this definition is that it relies on the assembly of the matrix A , which leads to prohibitive costs in the context of higher-order methods. Constructing low-rank tensor approximations to A_i^{-1} is an alternative. These can be constructed without computing the actual element stiffness matrices but by applying \mathcal{A}_K [79] or geometrically (see below). Furthermore, element blocks can be inverted iteratively, using, e.g., Krylov solvers [80] with preconditioners like diagonal or low-rank tensor approximations.

BB9: matrix representation for preconditioning

For setup of algebraic preconditioners, efficient kernels allow to construct the matrix representation or its blocks and diagonal, based on \mathcal{A} or \mathcal{A}_K and unit vectors.

Geometric approaches

The mesh can be explicitly used to rediscretize the system and generate a preconditioner. For example, geometric multigrid rediscretizes the operator on multiple refinement levels and combines the level results multiplicatively via intergrid transfer operators.

In the context of high-order FEM, it is possible to create multigrid levels by decreasing the polynomial degree of the used polynomials (p -multigrid). Alternatively, it is possible to construct a new mesh based on the support points of the higher-order elements and use linear elements. The resulting matrix is sparse and spectrally equivalent to the original system and is, as a consequence, an effective preconditioner. This approach is referred to as SEM-FEM in the literature [81, 82].

In the context of ASM, low-rank tensor approximation of patch matrices \hat{A}_K can be derived from an auxiliary operator defined on an approximate Cartesian

mesh [81, 83, 84]. For such meshes, the Laplacian A_i^{cart} can be expressed by the tensor product of 1D stiffness and mass matrices in 2D:

$$A_i^{\text{cart}} = K_1 \otimes M_0 + M_1 \otimes K_0.$$

The fast-diagonalization method [85] gives an explicit formula for the inverse as

$$(A_i^{\text{cart}})^{-1} = (T_1 \otimes T_0)(\Lambda_1 \otimes I + I \otimes \Lambda_0)^{-1}(T_1^T \otimes T_0^T), \quad (2.20)$$

with T_i and Λ_i , being the (orthonormal) eigenvectors and the diagonal matrix of eigenvalues, obtained from the generalized eigendecomposition $K_i T_i = \Lambda_i M_i T_i$.

Block preconditioner

The basic preconditioners listed above can be composed to more complex block preconditioners, most notably, to block-Jacobi preconditioners, Schur-complement-based approaches,

$$A = \begin{bmatrix} K & C^T \\ C & 0 \end{bmatrix}, \quad P = \begin{bmatrix} K & C^T \\ 0 & -S \end{bmatrix}, \quad (2.21)$$

with the Schur complement $S = CK^{-1}C^T$, and PRESB-based approaches [86],

$$A = \begin{bmatrix} K & -M \\ M & K \end{bmatrix}, \quad (2.22a)$$

$$P = \begin{bmatrix} K & -M \\ M & K + 2M \end{bmatrix} = \begin{bmatrix} I & -I \\ 0 & I \end{bmatrix} \begin{bmatrix} K + M & 0 \\ M & K + M \end{bmatrix} \begin{bmatrix} I & I \\ 0 & I \end{bmatrix}. \quad (2.22b)$$

Jacobian-free Newton–Krylov methods

A nonlinear equation of the form

$$f(x) = 0$$

is solved with the Newton method by the update iteration

$$x_{k+1} = x_k - J(x_k)^{-1}f(x_k).$$

In practice, the Jacobian matrix J is not inverted exactly but can be approximated by a preconditioned Krylov solver with a few iterations [23]. To avoid the costly assembly of the Jacobian matrix, one can evaluate its effect in a matrix-free way or approximate it on a vector by a finite difference,

$$J'(u)a \approx \frac{1}{\varepsilon}(F(u + \varepsilon a) - F(u)), \quad (2.23)$$

with $\varepsilon(u, a)$ being the perturbation factor [87–89]. Note that the Krylov solver still needs a preconditioner, $P^{-1} \approx J^{-1}$ or $P^{-1} \approx (J')^{-1}$. It might be based on the strategies discussed at the beginning of this section, for which matrix representation would be needed and, as a consequence, the setup might be costly. However, the preconditioners can be reused for multiple nonlinear iterations so that the costs might amortize.

Please be aware that Jacobian-free methods are naturally matrix-free methods, however, they potentially lead to different results compared to the matrix-based approach and are not robust for all configurations. As noted earlier, (nonlinear) residuals can be efficiently evaluated with the algorithms discussed in the context of matrix-free algorithms so that optimizations for the latter are also beneficial for Jacobian-free methods.

3. Accomplishments

This section presents the main accomplishments of this thesis. We start with discussing the scientific advances achieved in the context of high-order FEM. We continue with the discussion of novel high-dimensional and high-component applications. We conclude this section with presenting an adoption of the matrix-free operator evaluation in the context of non-matching grids.

3.1 Application to high-order FEM

3.1.1 Increasing the data locality

This subsection is based on Paper I published jointly with Martin Kronbichler and Dmytro Sashko. The applications were developed with Ivo Dravis (PDE optimization), Sebastian Proell (explicit time stepping; [11]), and Vladimir Ivannikov (JFNK for sintering applications; Paper VII).

We propose a software infrastructure that allows to increase data locality by performing vector updates before and after DoFs of cells have been touched during cell loops according to BB2. A possible use case is the zeroing of the destination vector, which is needed to be able to simply add cell contributions during the cell loop. If users want to efficiently interleave other (custom) vector-update operations with cell loops, algorithmic adjustments might be needed on the user side.

The idea of interleaving cell loops and vector updates is based on the fact that an entry in the source vector u and the destination vector v needs to be correctly initialized once before the *first* cell tries to operate on it. An entry in v is finalized, i.e., ready for further usage, once the *last* cell has added to it. At this stage, the entry in u is not needed any more and its value can be overridden for the next usage. Doing vector updates close to the cell loop is attractive, since the data is already in cache and does not have to be fetched again from main memory. However, tracking the state of each entry (first and last access) is not feasible and would lead to excessively high organizational overhead. Instead, we propose to track the state of ranges (with sizes of a multiple of the cache-line size). Due to good cache locality during the cell loop, we automatically get also good cache locality during the vector updates, with DoFs staying in the cache from the first access till the last usage, which allows to load large parts of the source and the destination vector only once.

Theoretical reasonings revealed that DoFs shared by multiple cells, particularly the corner DoFs, might have rather long life spans between their first and last usage (*long “liveliness”*). In experiments, this results in corresponding cache lines being evicted from cache in the meantime, even though only one entry prevents the finalization. We propose DoF-renumbering schemes, which enumerate the DoFs of different geometric entities according to the touch count separately, resolving the fragmentation issue and leading to longer continuous ranges that can be updated. We schedule the vector updates—referred to pre- or post-operation—and interrupt the cell loop as long as the accumulated data of the cells’ work still fits into the fast L2 cache, to maximize the L2 hit rate during vector updates.

In different projects, we explored and proposed reformulations of algorithms to be able to interleave cell loops and vector updates. We summarize these in the following. Use cases in the context of additive Schwarz methods as well as relaxation and Chebyshev iterations are shown in Paper IV (see Section 3.1.4).

Preconditioned conjugate gradient methods

We applied the proposed pre-/post-algorithm to accelerate a conjugate gradient method [90] that is preconditioned by a diagonal matrix. The resulting reformulated algorithm is presented as Algorithm 5 in Paper I.

Explicit time stepping

A forward Euler time stepper can be written as

$$T_{k+1} \leftarrow T_k + \frac{1}{\Delta t} M^{-1} f(T_k).$$

If one replaces the mass matrix M by a lumped version I_M , one can rewrite the algorithm with the pre-operation $T_{k+1} \leftarrow 0$, the right-hand-side evaluation into T_{k+1} , and the post-operation $T_{k+1} \leftarrow T_k + \frac{1}{\Delta t} I_M^{-1} T_{k+1}$.

PDE optimization

This example comes from the context of PDE-constrained optimizations. We solve a Poisson problem, imposing box constraints on the state enforced by the Moreau–Yosida framework, box constraints on the control and L_1 -regularization on the control. The resulting block system (ignoring the right-hand-side vector) reads as

$$\begin{bmatrix} v_s \\ v_c \end{bmatrix} = \begin{bmatrix} I_M & -I_E K \\ K I_E & I_B I_M \end{bmatrix} \begin{bmatrix} u_s \\ u_c \end{bmatrix},$$

with K being the regular stiffness matrix and I_M , I_E , and I_B being different diagonal or lumped matrices. For more details on this system, see Section 4.5

in [91]. The block system can be decomposed as follows:

$$\begin{bmatrix} v_s \\ v_c \end{bmatrix} = \underbrace{\begin{bmatrix} -I_E & 0 \\ 0 & I \end{bmatrix}}_{\text{post}} \underbrace{\begin{bmatrix} 0 & K \\ K & 0 \end{bmatrix}}_{\text{matvec}} \underbrace{\begin{bmatrix} I_E & 0 \\ 0 & I \end{bmatrix}}_{\text{pre}} \begin{bmatrix} u_s \\ u_c \end{bmatrix} + \underbrace{\begin{bmatrix} I_M & 0 \\ 0 & I_B I_M \end{bmatrix}}_{\text{post}} \begin{bmatrix} u_s \\ u_c \end{bmatrix}.$$

The resulting pre-/post-operations are indicated directly in the equation. The necessity to perform the vector scaling $I_E u_s$ in the pre-operation, whose result is used as input of the matrix-vector product with K , implies that a temporal vector is needed.

Jacobian-free Newton–Krylov method (JFNK)

The Jacobian-free Newton method (2.23) can be efficiently implemented with the pre-/post-algorithm by perturbing the linearization point in a pre-operation and taking the finite difference in a post-operation, as demonstrated in Paper VII.

3.1.2 Application of hanging-node constraints

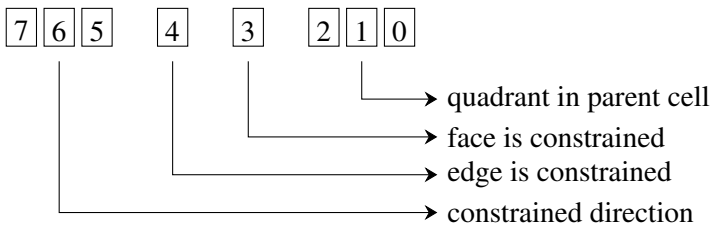
This section is based on Paper II published jointly with Karl Ljungkvist and Martin Kronbichler.

The need to apply hanging-node constraints in the case of continuous elements in order to guarantee continuity, i.e., H^1 conformity, leads to locally dense blocks in the constraint matrix (C_K), whose application can become overly expensive compared to efficient cell integrals. Adopting ideas from [21, 29, 34, 92], one can split up the application of constraints in a general-purpose and a hanging-node part:

$$C_K \circ R_K = C_K^{\text{HN}} \circ C_K^{\text{GP}} \circ R_K = C_K^{\text{HN}} \circ (C_K^{\text{GP}} \circ R_K).$$

Here, C_K^{HN} corresponds to an in-place interpolation based on the refinement configuration in the case that the constrained DoFs are replaced with the constraining ones during gathering.

A major challenge is that there are 137 possible refinement configurations in 3D. We propose a method to encode these in 8 bits (1 byte) in such a way that it is easy to decode by basic bit operations. The bit format is as follows:



The first entry (3 bits) specifies the quadrant within the parent cell, the second and third entry determine whether faces or edges are constrained, respectively, and the last entry (3 bits) specifies the direction of the constrained entities. The fact that both a face and an edge are constrained implies that a face and its orthogonal edge are constrained. The decoded information can be used to schedule a minimal number of in-face and in-line interpolation steps.

The second challenge we tackle is that, in the context of vectorization over cells, cells might have different refinement configurations requiring different steps to resolve the constraints. We have investigated strategies to solve this problem. The two most promising approaches in this regard are:

1. Process each cell individually. This approach leads to the lowest number of scalar operations and, in our experiments, turned out to be the fastest for low-order elements.
2. Perform an in-place interpolation on all geometric entities (4 lines and 4 quadrilaterals in each direction in 3D) and enable or disable the effect on certain cells by masks. Setting up the masks introduces noticeable overhead so that this approach is only beneficial for higher orders or multiple components.

Note: In our publication, we concentrate on CPU-based hardware, however, demonstrate experimentally the benefits of using the algorithms on GPU-based hardware as well.

3.1.3 Multigrid: global coarsening for locally refined meshes

This section is based on Paper III published jointly with Timo Heister, Laura Prieto Saavedra, and Martin Kronbichler.

We propose an efficient and generic implementation of multigrid algorithms for locally refined meshes for distributed compute systems, which can handle both local-smoothing [93–107] and global-coarsening approaches [103, 108]. The local-smoothing algorithm that we consider uses the refinement levels as multigrid levels, resulting in levels that do not cover the whole computational domain. In contrast, global-coarsening algorithms operate on the whole computational domain on all refinement levels. In the case of geometric global coarsening, the levels are obtained by combining children cells and, in the case of polynomial coarsening [23, 109–138], they are obtained by globally reducing the polynomial order of the underlying solution space. Independently of the way the levels are constructed, residuals need to be restricted and solution defects have to be prolonged.

Adopting the algorithms for pointwise interpolation from Section 2.1, we develop a prolongation operator that injects the solution from the support points of a coarse grid into the support points of a fine grid via looping over

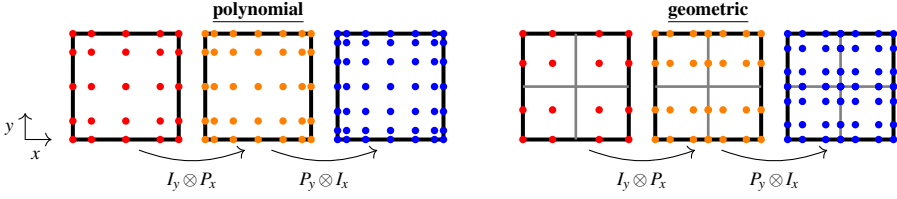


Figure 3.1. Evaluation steps of polynomial and geometric prolongation.

coarse cells in a matrix-free way:

$$x^{(f)} = \mathcal{P}_K^{(f,c)} = \sum_K \mathcal{S}_K^{(f)} \circ \mathcal{W}_K^{(f)} \circ \mathcal{P}_K^{(f,c)} \circ \mathcal{C}_K^{(c)} \circ \mathcal{R}_K^{(c)} x^{(c)}. \quad (3.1)$$

Here, $\mathcal{C}_K^{(c)} \circ \mathcal{R}_K^{(c)}$ reads the DoFs of a coarse cell and resolves the constraints. Important constraint types—in the case of global coarsening—include hanging-node constraints. They need to be evaluated on all levels, which is in contrast to local smoothing, where the hanging-node constraints do not have to be applied within the multigrid cycle, since the multigrid levels are constructed—per definition—so that they do not have hanging nodes but instead artificial Dirichlet boundaries. The need for evaluating hanging-node constraints in the new global-coarsening implementation of deal.II was the motivation to optimize the implementation of their application, which is documented in Paper II (see Section 3.1.2).

$\mathcal{P}_K^{(f,c)}$ prolongates the solution from a coarse cell to the fine cell(s). We use an L_2 -projection. The projection matrix differs in the context of geometric and polynomial coarsening. In the first case, the projection is defined between a parent cell and its children cells, while, in the case of p-multigrid, the projection is defined between a low- and a high-order element. For global coarsening, it might also be that the cells are not refined (anymore), implying an identity matrix as projection matrix. For tensor-product elements, the resulting projection matrices also have a tensor-product structure ($P_c = P_z \otimes P_y \otimes P_x$) so that the tensor-product kernels from Section 2.4 can be easily adopted, as indicated in Fig. 3.1.

The weighting operator $\mathcal{W}_K^{(f)}$ applies the inverse of the multiplicity of each DoF, and $\mathcal{S}_K^{(f)}$, finally, adds the result to the global (fine) vector. We choose the restriction operator as the transpose of the prolongation operator. Both prolongation (3.1) and restriction can be implemented efficiently in a matrix-free fashion. In the deal.II implementation, we directly use the internal building blocks ($\mathcal{C}_K^{(c)} \circ \mathcal{R}_K^{(c)}$ with advanced hanging-node kernels, $\mathcal{S}_K^{(f)}$, and tensor-product kernels in $\mathcal{P}_K^{(f,c)}$) to construct a light-weight transfer operator. We vectorize over coarse cells, for which we categorize them so that only cells with the same cell-local prolongation matrices are processed together.

The developed transfer operator has been integrated in a parallel multigrid setting. In parallel multigrid, levels can be partitioned differently among processes. The partitioning has an effect not only on the load balance during smoothing on a level (*horizontal communication*) but also on the costs of inter-grid transfer (*vertical communication*). For instance, repartitioning the mesh on each level implies low load imbalance but potentially high costs during transfer. In contrast, the first-child policy implies low transfer costs but potentially high load imbalances. In Paper III, we investigate experimentally—for the solution of a basic Poisson problem—the influence of partitioning on the behavior of global coarsening and compare this behavior with that of local smoothing (with first-child policy). We can not observe a clear advantage of the choice of the partitioning of the levels, since it is very much mesh-dependent. However, we observe, throughout the experiments, a significant advantage of global coarsening over local smoothing [47, 101] in a parallel setting, even though the former performs more work and is, thus, more expensive in serial. The main reason for this is that the work can be reduced more significantly between each multigrid-level pair. Similar observations are also made while solving a variable viscosity Stokes problem with a Schur-complement approach (2.21). For this purpose, we integrated the developed infrastructure into ASPECT [139, 140]¹, an advanced deal.II-based code to simulate the convection in Earth’s mantle.

Note: The proposed concept allows a simple extension not only to DG but also to an auxiliary-space idea where DG discretization is only applied on the finest level and continuous discretization is used on coarser multigrid levels. In the context of DG, the evaluation of hanging-node constraints is not needed during transfer and is replaced by the evaluation of fluxes on subfaces during smoothing, and weighting is redundant. Similarly, the transfer from continuous to discontinuous space can be performed by cell-local prolongation and weighting, based on identity matrices. In [8], we extend the hybrid multigrid solver in ExaDG [2, 3] for locally refined meshes, using this approach, and solve incompressible-flow problems on complex lung geometries.

Note: The proposed algorithms also work in the context of hp-adaptive FEM by extending the cell-local prolongation so that it can handle arbitrary coarse-fine degree pairs. Preliminary results have been released in tutorial step-75 of deal.II [141], which solves a Poisson problem on a reentrant-corner geometry, using the matrix-free, the distributed hp [77, 142], and the multigrid infrastructures of this library. The results are promising and indicate a faster and more robust solver than AMG. This work was done in collaboration with Marc Fehling.

¹<https://github.com/geodynamics/aspect>

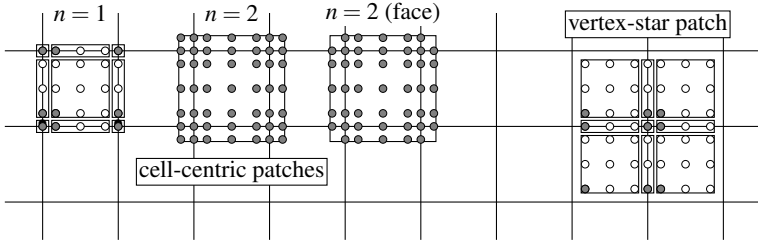


Figure 3.2. Visualization of cell-centric and vertex-star patches, including index storage in 2D for $p = 4$, with filled circles indicating stored information; the rest is deduced by contiguous storage. Adopted from Paper IV.

3.1.4 Multigrid: efficient block smoothers

This section is based on Paper IV authored jointly with Martin Kronbichler.

In Paper III (see Section 3.1.3), we discussed how the construction of the multigrid levels influences the overall performance, in particular, the resulting load imbalances and communication costs. We used rather basic smoothers in form of Chebyshev iterations around a point-Jacobi preconditioner. In Paper IV, we concentrate on the efficient implementation of block smoothers and propose cache-efficient and low-overhead versions of additive Schwarz preconditioners built around patches, which we optionally accelerate by relaxation sweeps or Chebyshev iterations.

According to (2.19), additive Schwarz methods (ASM) add several contributions to unknowns in the overlapping regions. In order to cope with this issue, they need to weight, e.g., by the inverse of the valence of unknowns. Weights can be applied globally or locally. Weighting after the application of the local inverse results in a non-symmetric preconditioner even if the underlying operator is symmetric. This fact motivated the development and investigation of alternative ways for weighting, e.g., the splitting into parts before and after the local inverse.

We considered cell-centric patches with overlap $n = 1$ and $n = 2$ as well as vertex-star patches mainly on structured meshes (see Fig. 3.2). Patches on unstructured meshes may involve arbitrary number of cells, making the development of efficient patch smoothers challenging [143]. A remedy for cell-centric patches is to only consider face neighbors [144]. We propose cache-optimized and low-overhead implementations of ASM in the context of matrix-free computations. In the following, we summarize the main developments and observations. We vectorize over patches, apply the tensor-product kernels from Section 2.4, and treat the eigenvalues and eigenvectors as passive variables.

- The matrix-free cell-loop algorithm (with overlapping of communication and computation and interleaving with vector updates; BB2) is also applicable to the ASM case. In contrast to the cell loop in the standard

matrix-free algorithm, we loop over patches (defined around cells or vertices) with their own sets of DoFs, i.e., restriction matrices R_K . They influence when DoFs are touched for the first and the last time and, as a consequence, when pre-/post-operations are executed. Furthermore, the number of DoFs that need to be communicated during vector updates is influenced. We would like to note that cell-centric patches with overlap $n = 1$ have the same DoF sets as regular elements so that data structures that have been set up, e.g., for the operator A , can be reused.

- Since the matrix-free algorithm with pre-/post-operation is applicable, it is a natural choice to perform the global weighting during a pre-/post-vector update interleaved with a cell loop. In the context of cell-centric patches with overlap $n = 1$ and vertex-star patches, one can, however, adopt, as an alternative, a strategy we also use for prolongation (see Section 3.1.3). In these cases, the weight of each geometric entity a patch is made up from is stored. For hypercubes, only 3^d weights need to be stored in a compressed way, which is independent of the polynomial degree k . For cell-centric patches with larger overlap ($n > 1$), this strategy is not applicable. Our experiments show that interleaving vector updates with cell loops gives the best throughput in this case. In the context of cell-centric patches with overlap $n = 1$ and vertex-star patches, the local application of the compressed weights gives the highest throughput and the interleaving strategy with pre-/post-operation is close to the best option.
- Just like the weights, the indices in R_K can also be compressed, in the case of cell-centric patches with overlap $n = 1$ and vertex-star patches, by only storing the first index of each 3^d geometric entity and a subsequent reorientation of DoFs. For cell-centric patches with $n > 1$, such a compression is not trivial.

We accelerate the developed ASM implementation and a point-Jacobi preconditioner by a relaxation scheme or a Chebyshev iteration. Both involve a sequence of residual computations ($r = b - Ax$), preconditioner applications, and vector updates. In order to improve the throughput by using the cache and reducing the access to the main memory, we propose to exploit the structures of A (cell loop) and preconditioner P (either diagonal or cell loop) and perform necessary vector updates during cell/patch loops in the form of pre-/post-operations (see Section 3.1.1). Novel, reformulated versions of the relaxation scheme and Chebyshev iterations are presented in Paper IV.

We include the developed and highly optimized smoothers in the context of p -multigrid and conduct extensive studies on the multigrid parameters, e.g., the number of smoothing steps and the type of coarsening of the polynomial degree k . In addition, we consider 4th-kind Chebyshev polynomials and one-side smoothing, which have been recently developed and investigated in related studies [54, 145]. Our results indicate that highly optimized ASM preconditioners can outperform highly optimized point-Jacobi preconditioners

even for rather simple configurations, since, generally, less smoothing steps accumulated are needed to obtain the same tolerances. In addition, we compared the non-symmetric and symmetric types of weighting. For the former, we obtain—in accordance with the literature [146]—less iterations, but we need to apply GMRES with expensive orthogonalization steps and an additional preconditioner application. In our experiments, we could not identify a clear winner. As a concluding remark, we would like to state that finding the optimal set of parameters for multigrid (smoothers) is a nontrivial task due to the large search space and an auto-tuning step—as proposed by [81]—is unavoidable in large production runs.

3.1.5 Stage-parallel implicit Runge–Kutta methods

This section is based on Paper V published jointly with Ivo Dravins, Martin Kronbichler, and Maya Neytcheva.

The development of stage-parallel Runge–Kutta methods [147] and parallel-in-time integration methods is currently particularly popular, since they potentially allow to shift the scaling limit to lower values on future large-scale compute systems. This is done by performing operations of different stages or time steps in parallel instead of in sequence. However, potential additional operations and communication counteract the possible benefits so that these methods might become actually slower than the sequential counterpart if they are not implemented thoroughly. As a consequence, they are not widely used and are rather experimental, as is the case at the time of writing of this thesis.

In Paper V, we develop and investigate multiple stage-parallel implementations of implicit Runge–Kutta methods. These methods advance the solution to the next time step by a linear combination of Q intermediate stage-function values:

$$u_{n+1} = u_n + \tau \sum_{1 \leq q \leq Q} b_q \mathbf{k}_q \quad \text{with} \quad \mathbf{k}_i = f \left(t_n + c_i \tau, u_n + \tau \sum_{1 \leq j \leq Q} a_{ij} \mathbf{k}_j \right),$$

using the Butcher tableau $\begin{array}{c|c} \mathbf{c}_Q & A_Q \\ \hline & \mathbf{b}_Q^T \end{array}$. For a linear system of equations, the intermediate stage-function values can be obtained, expressed using Kronecker products, via:

$$\underbrace{(A_Q^{-1} \otimes M + \tau I_Q \otimes K)}_{\mathcal{A}} \mathbf{k} = (A_Q^{-1} \otimes I_n) \bar{\mathbf{g}} - (A_Q^{-1} \otimes K)(\mathbf{e}_Q \otimes \mathbf{u}_0),$$

with M the mass matrix, K the stiffness matrix, $\bar{\mathbf{g}}$ the right-hand-side function evaluated at different stages, and \mathbf{u}_0 the solution from the last time step. Following Butcher [148], one can construct the spectral decomposition of

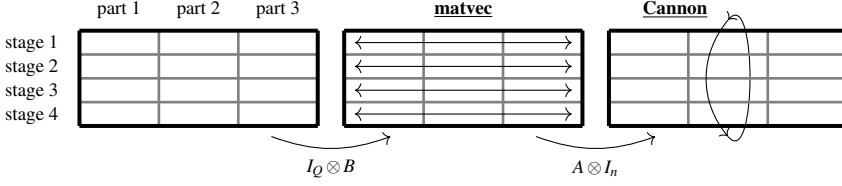


Figure 3.3. Implementation of $\mathbf{v} = (A \otimes B)\mathbf{u} = (A \otimes I)(I \otimes B)\mathbf{u}$ with $A \in \mathbb{R}^{Q \times Q}$, $B \in \mathbb{R}^{n \times n}$ in the context of stage-parallel Runge–Kutta methods with distributed stage solutions. In the case that stages are assigned to different process groups, $\mathbf{v} = (A \otimes I)\mathbf{u}$ is efficiently implemented via Cannon’s algorithm [155].

$A_Q^{-1} = S\Lambda S^{-1}$, with potential complex eigenvectors S and eigenvalues Λ , and use this to transform the matrix:

$$A = (A_Q^{-1} \otimes M + \tau I_Q \otimes K) = (S \otimes I_n) \underline{(\Lambda \otimes M + \tau I_Q \otimes K)} (S^{-1} \otimes I_n). \quad (3.2)$$

The system matrix A can be either explicitly factorized,

$$A^{-1} = (S \otimes I_n) \underline{(\Lambda \otimes M + \tau I_Q \otimes K)^{-1}} (S^{-1} \otimes I_n), \quad (3.3)$$

requiring complex arithmetic, or solved iteratively, e.g., by GMRES with the help of a real-valued preconditioner [149–151],

$$P^{-1} = (\tilde{S} \otimes I_n) \underline{(\tilde{\Lambda} \otimes M + \tau I_Q \otimes K)^{-1}} (\tilde{S}^{-1} \otimes I_n), \quad (3.4)$$

where \tilde{S} and $\tilde{\Lambda}$ are obtained by the spectral decomposition $L = \tilde{S}\tilde{\Lambda}\tilde{S}^{-1}$ of the lower triangular part L of A_Q^{-1} [152]. Other ways to solve (3.2) in both the linear and the nonlinear setting are discussed in [153, 154].

In (3.2), (3.3), and (3.4), the inner terms are block-diagonal so that the blocks can be applied or solved independently once a basis change has been performed. Different tensor-product operations, particularly the following ones, which are also shown in Fig. 3.3, can be identified in the equations:

- $\mathbf{v} = (I_Q \otimes C)\mathbf{u} \leftrightarrow \mathbf{v}_i = C\mathbf{u}_i$ with $1 \leq i \leq Q$, $C \in \mathbb{R}^{n \times n}$
- $\mathbf{v} = (D \otimes I_n)\mathbf{u} \leftrightarrow \mathbf{v}_i = \sum_{1 \leq j \leq Q} D_{ij}\mathbf{u}_j$ with $1 \leq i \leq Q$, $D \in \mathbb{R}^{Q \times Q}$
- $\mathbf{v} = A \otimes B\mathbf{u} = (D \otimes I)(I \otimes C)\mathbf{u}$ with $C \in \mathbb{R}^{n \times n}$, $D \in \mathbb{R}^{Q \times Q}$.

We formalize these operations for distributed compute systems where stages are assigned to process groups and investigate experimentally potential benefits of stage parallelism by applying the derived algorithm to a heat equation and comparing the results to theoretical performance models. In this context, we apply $(\Lambda_i M + \tau K)$ via a *matrix-free cell loop* and approximate $(\tilde{\Lambda}_i M + \tau K)^{-1}$ via a single V-cycle of the geometric multigrid solver presented in Paper III (see Section 3.1.3).

We investigate two ways to run matrix-free loops corresponding to independent blocks: 1) run matrix-free loops by independent process groups or

2) merge the matrix-free loops and process the cell integrals in a batched way (multicomponent FEM; see also Sections 2.4 and 3.3). In our experiments, the first approach achieves the best performance and, indeed, can shift the scaling limit, which we showed experimentally on up to 150k MPI processes. However, we believe that the batched approach is competitive for deformed meshes, where metric terms need to be loaded.

Note: In (3.3), the eigenvalues/eigenvectors are complex and, as a result, the blocks are complex as well:

$$\lambda_q M + \tau K = (\Re(\lambda_q) + i \cdot \Im(\lambda_q))M + \tau K = \underbrace{(\Re(\lambda_q)M + \tau K)}_{K'_q} + i \cdot \underbrace{\Im(\lambda_q)M}_{M'_q},$$

which can be written as a two-by-two block-matrix system:

$$\begin{bmatrix} K'_q & -M'_q \\ M'_q & K_q \end{bmatrix} \begin{bmatrix} \Re(u_q) \\ \Im(u_q) \end{bmatrix} = \begin{bmatrix} \Re(v_q) \\ \Im(v_q) \end{bmatrix}.$$

This system is implemented with a single cell loop, and cell integrals are efficiently expressed in the form of two-component FEM. As an appropriate block preconditioner, we use PRESB (2.22).

3.2 Application to high-dimensional FEM

This section is based on Paper VI published jointly with Katharina Kormann and Martin Kronbichler.

We show the applicability of matrix-free operator evaluations to higher dimensional PDEs. For us, higher dimension is more than the number of dimensions that regular general-purpose finite-element libraries support, which is typically up to 3D. The work is motivated by computational plasma physics, where higher-dimensional advection-type equations need to be solved.

3.2.1 Motivation: computational plasma physics

The main objective of computational plasma physics is the description of the evolution of a plasma and its interaction in magnetic fields. A field of application is the fusion-energy research, in which the plasma in fusion reactors (e.g., tokamak and stellarator) is investigated. Besides the particle model [156], where the motion of each particle (with mass m_i , charge q_i , position \mathbf{x}_i , and velocity \mathbf{v}_i) in an electric field \mathbf{E} and a magnetic field \mathbf{B} is described by an N-body problem of the form

$$\frac{\partial^2 \mathbf{x}_i}{\partial t^2} = \frac{q_i}{m_i} (\mathbf{E}(t, \mathbf{x}_i) + \mathbf{v}_i \times \mathbf{B}(t, \mathbf{x}_i)),$$

and fluid models [157], in which the Navier–Stokes equations are coupled to a system of Maxwell’s equations (magnetohydrodynamics; MHD), *kinetic models* are common. They are described by a distribution function $f(t, \mathbf{x}, \mathbf{v})$, which evolves according to the Vlasov equation coupled to a system of Maxwell’s equations.

The Vlasov equation with a single particle species with charge q and mass m is

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{x}} f + \mathbf{a}(t, f, \mathbf{x}, \mathbf{v}) \cdot \nabla_{\mathbf{v}} f = 0, \quad (3.5)$$

which contains derivations in \mathbf{x} -space and phase (\mathbf{v} -)space and is coupled to the Maxwell’s equations for the self-consistent fields:

$$\mathbf{a}(t, f, \mathbf{x}, \mathbf{v}) = \frac{q}{m} (\mathbf{E}(t, \mathbf{x}) + \mathbf{v} \times \mathbf{B}(t, \mathbf{x})).$$

In the following, we assume $q/m = -1$ and that the magnetic field can be neglected:

$$\mathbf{a}(t, f, \mathbf{x}, \mathbf{v}) = -\mathbf{E}(t, \mathbf{x}).$$

The electric field can be obtained, e.g., by solving a Poisson equation:

$$\begin{aligned} \mathbf{E}(t, \mathbf{x}) &= -\nabla_{\mathbf{x}} \phi(t, \mathbf{x}), \\ -\nabla_{\mathbf{x}}^2 \phi(t, \mathbf{x}) &= \rho(t, \mathbf{x}), \\ \rho(t, \mathbf{x}) &= 1 - \int f(t, \mathbf{x}, \mathbf{v}) \, d\mathbf{v}, \end{aligned}$$

which gives together with (3.5) the Vlasov–Poisson equations. They form a nonlinear, high-dimensional system of PDEs. The discretization suffer from *curse of dimensionality*, which is the reason why different numerical schemes tailored to these equations have been derived: sparse-grid methods [158], tensor compression [159], and semi-Lagrangian methods [160]. In Paper VI, we investigate an approach based on a discontinuous Galerkin discretization. Fig. 3.4 shows a possible solution of the Vlasov–Poisson equations for a two-stream instability.

3.2.2 Software and performance aspects

For the sake of simplicity and without loss of generality, we consider—instead of the nonlinear Vlasov equation (3.5)—a high-dimensional advection equation with the DG discretization (2.8), as a prototype. To simplify the following discussion and to keep the similarity to the actual problem, we define:

$$\mathbf{a} := \begin{pmatrix} \mathbf{a}_x \\ \mathbf{a}_v \end{pmatrix} \quad \text{and} \quad \nabla := \begin{pmatrix} \nabla_x \\ \nabla_v \end{pmatrix}.$$

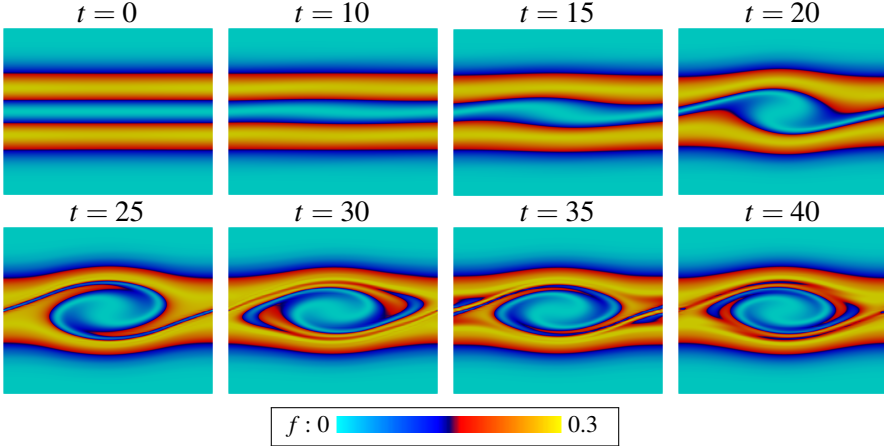


Figure 3.4. Solution of the Vlasov–Poisson equations for a 1D+1D two-stream instability in phase space.

It is obvious that a matrix-free operator evaluation is also applicable to higher dimensions. However, there is a lack of finite-element libraries that can tackle higher dimensions, i.e., due to an increased complexity in the description of the connectivity of cells in such meshes.

Instead of developing a completely new library or extending an existing one for a niche application, we propose an approach that allows to reuse existing building blocks from lower dimensional libraries. For this purpose, we limit ourselves to tensor-product meshes of the form $\Omega := \Omega_{\mathbf{x}} \otimes \Omega_{\mathbf{v}}$, which is motivated by the application at hand, where the geometry and the phase space are independent. A natural consequence is to discretize $\Omega_{\mathbf{x}}$ and $\Omega_{\mathbf{v}}$ independently such that the cells are given as $\Omega_K := \Omega_{K_{\mathbf{x}}} \otimes \Omega_{K_{\mathbf{v}}}$ and faces accordingly, implying a block-diagonal structure of the Jacobi matrices. Exploiting this, Eq. (2.8) finally becomes:

$$\begin{aligned} \left(g, \frac{\partial f}{\partial t} \right)_{\Omega_{K_{\mathbf{x}}} \otimes \Omega_{K_{\mathbf{v}}}} &= \left(\hat{\nabla} g, \begin{pmatrix} J_{\mathbf{x}}^{-1} \mathbf{a}_{\mathbf{x}} \\ J_{\mathbf{v}}^{-1} \mathbf{a}_{\mathbf{v}} \end{pmatrix} f \right)_{\Omega_{K_{\mathbf{x}}} \otimes \Omega_{K_{\mathbf{v}}}} \\ &+ \left(g, \mathbf{n}_{\mathbf{x}} \cdot (\mathbf{a}_{\mathbf{x}} f)^* \right)_{\Gamma_{K_{\mathbf{x}}} \otimes \Omega_{K_{\mathbf{v}}}} + \left(g, \mathbf{n}_{\mathbf{v}} \cdot (\mathbf{a}_{\mathbf{v}} f)^* \right)_{\Omega_{K_{\mathbf{x}}} \otimes \Gamma_{K_{\mathbf{v}}}}. \end{aligned}$$

The inspection of this equation reveals that one needs building blocks providing implementations of the following tasks:

1. loop over pairs of lower-dimensional cells ($\Omega_{K_{\mathbf{x}}} \otimes \Omega_{K_{\mathbf{v}}}$) and face/cell pairs ($\Gamma_{K_{\mathbf{x}}} \otimes \Omega_{K_{\mathbf{v}}}$ and $\Omega_{K_{\mathbf{x}}} \otimes \Gamma_{K_{\mathbf{v}}}$);
2. access data in global vectors in high and low dimensions in a controlled way², i.e., dealing with ghost-value updates and compression as well as with race conditions;

²Remember that $\mathbf{a}_{\mathbf{v}} = -\mathbf{E}(t, \mathbf{x})$ in the Vlasov-Poisson equations.

3. perform basis change from the support points to the quadrature points and vice versa during cell/face integrals;
4. access mapping data at quadrature-point pairs $(J_x, J_v, \mathbf{n}_x, \mathbf{n}_v)$.

The mapping data (task 4) can be easily queried, e.g., from two lower-dimensional mesh objects. The evaluation and tensor-product kernels (see Sections 2.4, 2.6, and 2.11) are applicable also to higher dimensions (task 3), since

$$N = N_v \otimes N_x = \underbrace{N_{1D} \otimes \cdots \otimes N_{1D}}_{\times d_x} \otimes \underbrace{N_{1D} \otimes \cdots \otimes N_{1D}}_{\times d_v} = \underbrace{N_{1D} \otimes \cdots \otimes N_{1D}}_{\times d_x + d_v}.$$

For task 1, the algorithm implied by (2.12) can be simply extended such that it loops over pairs of cells and faces. The data access to the global vector (task 2) is straightforward given that we use a DG discretization: all one needs is to convert a pair of cell indices to a constant offset into the vector. In order to be able to perform operations in x - and v -space independently, we work on different MPI communicators.

In conclusion, we propose—for high-dimensional applications—to decouple the work done by the matrix-free object (see Section 2.9): 1) looping over cells and vector access are performed with an object tailored for higher dimensions and 2) mapping is queried from two low-dimensional (matrix-free) objects. Please note that many low-level building blocks from a low-dimensional library can also be reused for task 1.

In Paper VI, we have shown that the proposed concept can be efficiently implemented and applied to solve problems of computational plasma physics. However, it is needless to say that the curse of dimensionality makes the computations expensive and leads to other corresponding problems, which can be partly reduced but not completely resolved by thorough performance optimizations. Such issues include (see also Section 2.10):

- unavoidable high computational costs $\mathcal{O}(dk^{d+1})$ during cell integrals;
- memory consumption that increases with $\mathcal{O}(p^d)$, with p being the number of DoFs in each direction; we adopt a low-storage Runge–Kutta method [161] to reduce the memory consumption;
- challenging, rather dense, long-range communications and an increased amount of ghost DoFs due to ghost faces with $(k+1)^{d-1}$ DoFs; for the latter issue, shared-memory capabilities of MPI-3.0 are used (see Section 2.3) and the number of ghosted vectors is reduced to a single one;
- increased working-set size $\mathcal{O}(dk^d)$ during cell/face integrals particularly if work is vectorized over cells; as a remedy, special-purpose operators are provided where evaluation/integration and loop over quadrature points are interleaved.

As a reference, the computational complexity of matrix-based algorithms is $\mathcal{O}((k+1)^{2d})$; and for 6D: $\mathcal{O}((k+1)^{12})$ in contrast to $\mathcal{O}(6(k+1)^7)$ using sum factorization. For special matrices, the tensor-product structure (mass matrix:

$M = M_{\mathbf{v}} \otimes M_{\mathbf{x}}$; stiffness matrix: $K = M_{\mathbf{v}} \otimes K_{\mathbf{x}} + K_{\mathbf{v}} \otimes M_{\mathbf{x}}$) can be exploited to reduce the complexity [162]. However, the velocity field in the Vlasov-Poisson equations is not separable, not allowing this technique.

Note: We discussed a DG implementation. However, the high-dimensional matrix-free algorithm is also applicable for FEM [162], as shown in a prototype on GitHub³. Here, no interface fluxes have to be evaluated, but indices need to be stored explicitly, making the access to global vectors and the updates of ghost values more expensive.

Note: The algorithms discussed here are interesting and applicable for lower-dimensional PDEs, too. In many instances, 3D meshes are generated by the extrusion of 2D meshes, resulting in tensor-product meshes again. Parallel-in-time integration algorithms also feature tensor-product structures; see also the discussion on a stage-parallel implicit Runge–Kutta implementation in Paper V (see Section 3.1.5), which reuses, e.g., the partitioning and shared-memory infrastructure developed for solving high-dimensional PDEs in the scope of this thesis.

3.3 Application to multicomponent FEM

This section is based on Paper VII, a work done jointly with Vladimir Ivanikov, Christian Cyron, and Martin Kronbichler.

Solving systems with multiple unknowns (components) is widespread in the context of FEM, e.g., in solid mechanics (d), incompressible fluid mechanics ($d + 1$), and compressible fluid mechanics ($d + 2$). Considering more components are not that common and, as a result, implementations for such use cases are not optimized in general-purpose libraries. Examples for such niche application areas are the density functional theory (DFT) [39–41] and phase-field applications like solid-state sintering [38, 163–165]. We investigate the latter in Paper VII.

3.3.1 Motivation: solid-state sintering

The classical formulation of modeling solid-state-sintering processes of N particles proposed by Wang [166] is based on a system of Cahn–Hilliard and

³<https://github.com/hyperdeal/hyperdeal>

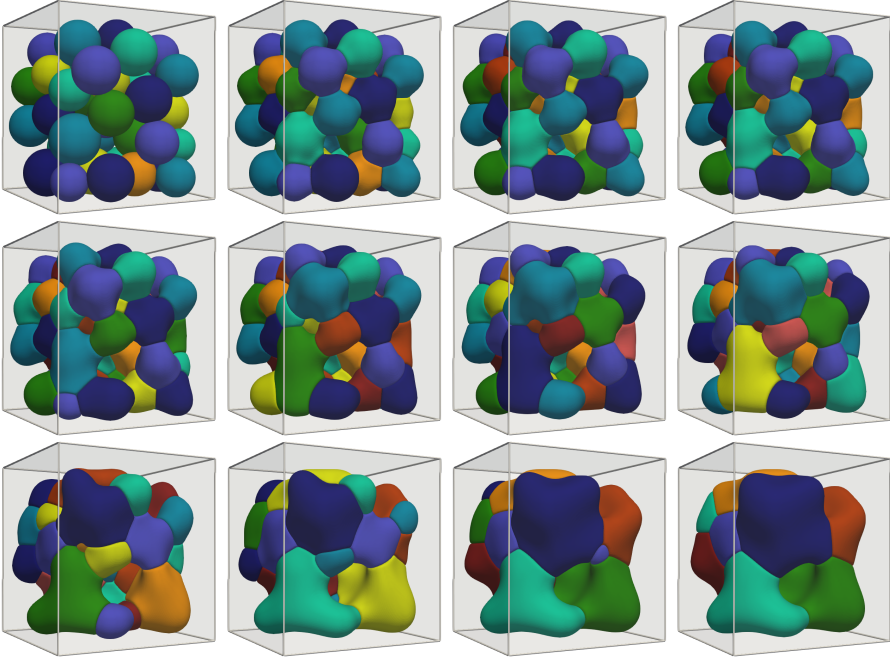


Figure 3.5. Isocontours of a 49-particle simulation at different times. Colors indicate the order parameters.

Allen–Cahn equations:

$$\frac{\partial c}{\partial t} = \nabla \cdot [M(c, \eta_i) \nabla \mu], \quad (3.6a)$$

$$\mu = \frac{\partial f(c, \eta_i)}{\partial c} - \kappa_c \nabla^2 c, \quad (3.6b)$$

$$\frac{\partial \eta_i}{\partial t} = -L \left[\frac{\partial f(c, \eta_i)}{\partial \eta_i} - \kappa_p \nabla^2 \eta_i \right] \quad \text{for } 1 \leq i \leq N. \quad (3.6c)$$

Here, the microstructure evolution is described by a conserved variable c and a set of non-conserved unknowns η_i . $\mu = \delta F / \delta c$ is the chemical potential. Variable c can be interpreted as the molar fraction of the given material and has a magnitude of 1 inside particles and 0 in voids. The unknown η_i describes the position of particle i within the domain such that $\eta_i = 1$ inside the i -th particle and $\eta_i = 0$ everywhere else. Due to the local support of η_i , it is common in the literature [167–169] to collect non-neighboring particles in groups called *order parameters* and describe all particles in such a group by a single η_i . In practical simulations, numbers of order parameters between 8–12 are common, leading to 10–14 components in total. $f(c, \eta_i)$ is a polynomial related to the free energy of the system and $M(c, \eta_i)$ is the mobility, leading to a strong coupling of the unknowns.

Fig. 3.5 shows, as an example, the isocontour of a 49-particle packing over time. The colors indicate the order parameters. It is clear that particles switch order parameters during simulations when they become too close to each other due to topological changes. Particles can even disappear. These special features can lead to the situation that the number of order parameters, i.e., the number of components, changes in the course of the simulation, a situation that is not common in conventional FEM simulations.

The strong coupling between the components results in memory consumption that increases quadratically with the number of components and is inherently expensive, considering that 10–14 components are used in representative simulations. This requires the investigation of alternatives like the evaluation of the Jacobian based on the matrix-free algorithm or the approximation via finite difference (JFNK; see Section 2.12).

3.3.2 Software and performance aspects

The extension of matrix-free operator evaluations to multiple components is straightforward, as shown in Section 2.4. To cope with the dynamically changing number of components, we suggest to convert the number to a constant expression and to precompile the relevant cell integral for all possible numbers of components. This results in Algorithm 1 in Paper VII. However, experiments revealed that, with increasing number of components, the throughput decreases. Interleaving evaluation and integration steps, as described in Section 2.11, reduces this effect, but keeping the number of components low, e.g., via grain tracking [169], is crucial.

Alternatively to Algorithm 1, Algorithm 2 in Paper VII proposes to exploit the fact that grains (η_i) have local support, in order to process only grains locally relevant on the cell. This implies that, on each cell, different numbers of grains need to be processed, resulting in different numbers of components, each of which needs to be translated to constant expressions, and different blocks of block vectors need to be accessed. In an optimal implementation, only DoFs of relevant grains (within a cell batch) are loaded and, only for these, memory is allocated. However, this implies the use of gather/scatter operators that can work on sparse block vectors. In [39], such data structures were investigated in the context of DFT. Although we did not adopt this optimization but worked on full block vectors, we observed significant speedup during operator evaluations despite unavoidable overhead related to communication and redundant vector update steps. These results are motivations for future work to investigate sparse block vectors in more detail and to tackle issues related to these types of vectors, like preconditioning.

3.3.3 Nonlinear solver and preconditioning

We discretize (3.6) with BDF2 in time and solve the resulting nonlinear problem with a Newton solver. The Jacobian is evaluated approximately in a matrix-free manner, and, then, the linear system is solved iteratively, using GMRES with a preconditioner. We adopt a block-Jacobi preconditioner based on ILU: the Cahn-Hilliard block is treated as a single block and we apply a single ILU instance for each Allen-Cahn block. This preconditioner is, although matrix-based, rather cheap and robust, outperforming the alternative matrix-free preconditioners we tried. This highlights that the choice of the preconditioner is very problem-specific and an efficient matrix-free operator evaluation does not automatically imply the need to use matrix-free preconditioners.

3.4 Coupling via non-matching grids

The following section is based on material mostly not published yet. The implementation was developed jointly with Martin Kronbichler. The applications were carried out together with Niklas Fehn (fluid-structure interaction), Maximilian Bergbauer, Simon Sticker (CutDG/CutFEM), Johannes Heinz (Nitsche-type mortar methods for acoustic equations), Magdalena Schreter–Fleischhacker (sharp-interface methods for two-phase flow), and Marco Feder (non-conformal multigrid).

In the following, we show that the matrix-free operator-evaluation strategies presented in Section 2 are also applicable in the context of non-matching grids, where solutions need to be evaluated at arbitrary points. We start with presenting application cases that need efficient implementations of such an operator. This allows us to collect requirements and to formalize an algorithm tailored for such use cases. We conclude this section by discussing challenges and efficient implementations as well as by comparing this algorithm with a black-box coupling approach provided by coupling libraries like preCICE [170, 171].

Note: Similar work has been also done in the spectral-element codes Nek5000/nekRS, where the fluid solution of two Nek5000 instances [172, 173] as well as particles with fluid [174] can be coupled. In contrast, we aim to provide a generalized concept that can be used in any FEM-based solvers.

Note: The literature on numerical solution of coupled problems is extensive and involves the development of many aspects like quasi-Newton methods. Our aim is not to discuss all ingredients but to concentrate on one of them: data mapping. For this, we show how to adopt matrix-free algorithms. Publications on data mapping include [175–182].

3.4.1 Motivation: multiphysics applications

In the following, we list application projects which the author has been involved in and used the developed matrix-free non-matching infrastructure.

Point-to-point interpolation and Nitsche-type mortaring

Let us assume we want to solve the advection equation (2.8) with DG on Ω . Two non-overlapping parts (Ω_1, Ω_2) of the domain are discretized independently, resulting potentially in a nonconformal interface ($\Gamma_I = \Omega_1 \cap \Omega_2$). During the flux computations in term \dagger of (2.8), the values⁴ are needed at the quadrature points from both sides of a face. However, in contrast to conforming meshes, quadrature points on the same negative side might belong to different cells on the positive side. Since the solution, in the FEM context, is defined by cell-local polynomials and, in the case of DG, by discontinuous shape functions, contributions coming from different cells imply kinks and discontinuities, potentially leading to aliasing effects during the face integral if a simple *point-to-point* evaluation is performed.

One way to tackle these aliasing effects is to perform the integrals on an intersected mesh, resulting, for each face, in a set of mortars (*Nitsche-type mortaring*). On each of them, a quadrature rule can be defined with the result that one needs to perform a non-standard (non-tensor-product) quadrature on such interfaces.

In [10], we consider the point-to-point and the Nitsche-type-mortaring evaluation for the conservative formulation of acoustic equations discretized with DG and show the benefits of the latter for suppressing artificial modes. In addition, we investigate overlapping meshes, which require intersections between faces and cells. For generating the intersections, we use the library CGAL [183]. This study is a preparatory work for simulating aeroacoustic problems where the *near field* (mesh), which moves with a rotating geometry and is described by an arbitrary Lagrangian-Eulerian formulation (ALE) [184], is coupled to a static *far field* (mesh).

Note: Due to the fact that integrals are only computed on the refined side, the DG implementation for locally refined meshes in deal.II (see Section 2.8) is implicitly using a mortar approach.

ALE formulation for fluid-structure interaction

The point-to-point approach described above has been applied to solve fluid-structure-interaction problems with ALE [44]. In this context, the fluid field is

⁴Gradients are accessed in the Poisson case as well.

described by

$$\left. \frac{\partial \mathbf{u}^F}{\partial t} \right|_{\chi} + ((\mathbf{u}^F - \mathbf{u}^M) \cdot \nabla) \mathbf{u}^F - \nabla \cdot F_v(\mathbf{u}^F) + \nabla p^F = \mathbf{f}^F, \quad (3.7a)$$

$$\nabla \cdot \mathbf{u}^F = 0 \quad (3.7b)$$

on Ω^F with the fluid velocity \mathbf{u}^F and mesh-motion velocity \mathbf{u}^M as well as appropriate initial and boundary conditions [185]. The solid field is described, in Lagrangian form, by

$$\rho_0 \frac{d^2 \mathbf{d}^S}{dt^2} - \nabla_0 \cdot \mathbb{P} = \mathbf{b}_0^S \quad (3.8)$$

on Ω^S with appropriate initial and boundary conditions as well as with \mathbb{P} being the first Piola–Kirchhoff stress tensor, d being the displacement and b_0^S the body forces. The coupling conditions at $\Gamma^I = \Omega^F \cap \Omega^S$ are

$$\mathbf{u}^F = \frac{d\mathbf{d}^S}{dt}, \quad (3.9a)$$

$$\boldsymbol{\sigma}^F \cdot \mathbf{n}^F + \boldsymbol{\sigma}^S \cdot \mathbf{n}^S = 0, \quad (3.9b)$$

with the Cauchy stress tensor $\boldsymbol{\sigma}$ and the output normal vector $\mathbf{n}^F = -\mathbf{n}^S$. Furthermore, the mesh displacement d^M and, as a result, the mesh velocity u^M are obtained by extending the displacement on the solid surface into the fluid domain.

During coupling, the deformation—as Dirichlet boundary condition of the fluid field and the mesh motion—and the stresses—as Neumann boundary condition of the solid field—have to be exchanged on the interface. The ALE formulation implies that the evaluation points do not change as long as the mesh is not remeshed so that the setup has to be performed only once. The latter is needed once the mesh quality has deteriorated significantly and involves a remapping of the solution and setting up the evaluation points for coupling again.

Immersed boundaries: CutFEM, CutDG, & XFEM

The methods CutFEM [186], CutDG [187, 188], and XFEM [189] solve PDEs on an arbitrarily shaped domain Ω by embedding it into a structured computational domain Ω_h . For this purpose, the PDEs are modified to weakly enforce boundary conditions at arbitrary positions (cuts) and integrals need to be evaluated on parts of cells.

In the context of CutFEM, the weak form of the Poisson problem⁵ is to find a function $u_h \in V_{\Omega}^h$ such that

$$a_h(u_h, v_h) = L_h(v_h), \quad \forall v_h \in V_{\Omega}^h, \quad (3.10a)$$

⁵adopted from https://www.dealii.org/developer/doxygen/deal.II/step_85.html

where

$$a_h(u_h, v_h) = (\nabla u_h, \nabla v_h)_\Omega - (\partial_n u_h, v_h)_\Gamma - (u_h, \partial_n v_h)_\Gamma + \left(\frac{\gamma_D}{h} u_h, v_h \right)_\Gamma, \quad (3.10b)$$

$$L_h(v_h) = (f, v)_\Omega + \left(u_D, \frac{\gamma_D}{h} v_h - \partial_n v_h \right)_\Gamma. \quad (3.10c)$$

Please note that volume integrals are only performed on cells $(\Omega_K \cap \Omega)$ and intersections $(\Omega_K \cap \Gamma)$, requiring non-standard quadrature rules. Important tasks are to identify cells that are cut, to determine the position of the cut within the cell, and to compute adequate quadrature rules. The first two tasks can be accomplished, e.g., based on level-set fields [186] or on intersection of two non-matching grids [190]. High-order quadrature rules can be generated, e.g., by the algorithm described in [191]. Given the weights and the reference positions of the quadrature points, the shape functions and the operators of the form (3.10) can be evaluated in a matrix-free way. Similar reasoning is also true in the CutDG and XFEM cases. All three approaches can be used to solve multiphysics problems like two-phase flow [192, 193] or fluid-structure interaction [194].

Coupling with particles

The motion of a massless particle can be described by the following ordinary differential equation:

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{u}(\mathbf{x}_i),$$

where $\mathbf{u}(\mathbf{x}_i)$ is the current velocity, e.g., from a finite-element solution evaluated at the current position \mathbf{x}_i of the particle. Independently of the time-integration scheme, the background velocity field needs to be evaluated efficiently at arbitrary points, which change after each time step. However, in many applications, particles do not move far in comparison to the underlying mesh; this information can be used to accelerate the setup [195].

For details on more involved coupling of particles with flow, we refer interested readers to [196]. Here, particles that are described by the discrete element method (DEM) [6] are coupled two-way to an incompressible CFD solver [197] to simulate cases relevant for chemical and process techniques (e.g., particle sedimentation, fluidized bed). The author of this thesis has been involved in the development of two tutorials covering the particle infrastructure of deal.II (step-19⁶, step-68⁷).

⁶https://www.dealii.org/developer/doxygen/deal.II/step_19.html

⁷https://www.dealii.org/developer/doxygen/deal.II/step_68.html

Sharp-interface methods/front tracking

Let us assume that the non-dimensional form of the compressible Navier-Stokes equations [198]

$$\begin{aligned}\rho^* \mathbf{u}_t + \rho^* \mathbf{u} \cdot \nabla \mathbf{u} &= -\nabla p + \frac{1}{Re} \nabla \cdot (2\mu^* \nabla^s \mathbf{u}) + \mathbf{F} + \mathbf{F}_S, \\ \nabla \cdot \mathbf{u} &= 0\end{aligned}$$

with two phases

$$\rho^* = \begin{cases} 1 & \text{in phase 1} \\ \frac{\rho_2}{\rho_1} & \text{in phase 2} \end{cases} \quad \mu^* = \begin{cases} 1 & \text{in phase 1} \\ \frac{\mu_2}{\mu_1} & \text{in phase 2.} \end{cases}$$

is given. Here, \mathbf{F}_S is the surface-tension force

$$\mathbf{F}_S = \frac{1}{We} \kappa \mathbf{n} \delta_\Gamma.$$

In a finite-element context, the actual surface-tension force is not needed in a strong form but in a weak sense $(\mathbf{v}, \mathbf{F}_S)_\Omega$. In numerical simulations, this term is either approximated using a smooth ansatz [198–200],

$$(\mathbf{v}, \mathbf{F}_S)_\Omega \approx \left(\mathbf{v}, \frac{1}{We} \kappa \mathbf{n} \nabla H \right)_\Omega,$$

replacing the Dirac delta function by a smooth function ∇H related to a level-set field, or evaluated sharply (sharp-interface method; SIM):

$$(\mathbf{v}, \mathbf{F}_S)_\Omega = \left(\mathbf{v}, \frac{1}{We} \kappa \mathbf{n} \right)_\Gamma \approx \sum_q \left(\mathbf{v}(\mathbf{x}_q), \frac{1}{We} \kappa(\mathbf{x}_q) \mathbf{n}(\mathbf{x}_q) |J(\mathbf{x}_q)| w_q \right). \quad (3.11)$$

In the case that a level-set description (ϕ) is available, the quadrature can be generated as above in the CutFEM/CutDG case. The normal field $\mathbf{n} = \nabla \phi / |\nabla \phi|$ and the curvature field $\kappa = -\nabla \mathbf{n}$ might have been computed from the level-set field ϕ via an L_2 -projection. Alternatively, the interface could be optionally described by a codim-1 mesh via a front-tracking approach [201, 202]. In the latter case, J , \mathbf{n} , and κ would be determined geometrically on the surface mesh, based on the current deformation of the mesh, for which a Lagrangian approach—as in the case of the particle motion—can be used. J , \mathbf{n} , and κ would need to be transferred to the background mesh.

Closest point projection, extrapolation, & geometric reinitialization

Let us assume that a signed distance function ϕ and a normal field \mathbf{n} are given. In such a setting, finding the closest point \mathbf{x}^* on the zero contour of the level-set field can be explicitly expressed as

$$\mathbf{x}^* = \mathbf{x} - \mathbf{n}(\mathbf{x}) \phi(\mathbf{x})$$

for a given point \mathbf{x} . Due to numerical errors and further constraints (e.g., orthogonality), this might not be enough and a fix-point iteration including geometric corrections might be necessary to find the correct point.

Once the point \mathbf{x}^* has been identified, we can project values from the interface to a narrow band and determine the distance $|\mathbf{x}^* - \mathbf{x}|$ to perform a geometric reinitialization [203].

Multigrid with non-nested levels

In many industrial applications, problems are solved with low-order finite elements on fine meshes generated by external mesh-generation tools. In this context, geometric multigrid and polynomial multigrid are not applicable, but AMG and non-nested multigrid methods [204, 205] are a natural choice. Non-nested multigrid is conceptually very similar to global coarsening, since both perform smoothing globally. Non-nestedness, however, implies that support points of the fine cells are arbitrarily positioned in regard to the coarse cells and parallel partitions. The interpolation is globally given by the pointwise algorithm from Section 2.1 and the L_2 -projection by (2.7), where the quadrature points might come from an intersected mesh. The complexity, in the parallel setting, is that the evaluation happens on the coarse mesh and the results need to be communicated to the fine mesh before one can proceed with weighting (in the injection case) or integration and testing (in the L_2 -projection case).

Summary & requirements

The requirements of the applications listed above can be summarized as follows:

- Evaluation points might have been generated on a cell or have to be sorted into cells via a search algorithm.
- Evaluation points might be structured within the donating mesh but unstructured within the evaluation mesh, not allowing sum factorization.
- Evaluation points might be owned by cells on remote processes, requiring communication during both setup and evaluation.
- At the evaluation points, either the solution is only evaluated or an integration is also performed, implying the need for mapping particularly to determine $|J(\mathbf{x}_q)|_{w_q}$.

In the following, we discuss general-purpose implementations that meet these requirements.

3.4.2 Distributed search

Many of the application cases listed above specify a set of points at which the solution should be evaluated. For this purpose, the cell K and the reference position $\hat{\mathbf{x}}$ within the cell need to be determined for a given point \mathbf{x} . The search is made more complicated by the fact that the mesh might be partitioned between processes and points might fall into a cell of any process.

In order to find the cell in a parallel setting, one can take a two-level-search approach. First, all processes that might possess the point are determined (*coarse search*). For this purpose, e.g., a distributed R-tree based on bounding boxes around locally owned domains could be used. Once the processes have been determined and the points have been sent to them as a request, one can start to find the cells among locally owned cells (*fine search*). In order to accelerate this search, an R-tree that is built around the vertices of the mesh can be used. One of the cells connected to the closest vertex is the cell of interest.⁸

Once the cell K containing point x has been found, one can find the reference position, in the general case, by performing the minimization:

$$\min_{\hat{\mathbf{x}}} (|\hat{\phi}_K(\hat{\mathbf{x}}) - \mathbf{x}|) \quad \text{with} \quad \hat{\mathbf{x}} \in [0, 1]^d.$$

For certain types of mapping (e.g., Cartesian mapping), the roots can be found explicitly.

Note: The search algorithm discussed above assumes no explicit knowledge of the position of points regarding cells. However, this is not always the case. For example, in time-dependent problems (e.g., DEM, SIM, moving meshes), particles/support points move in space with a specific velocity and are in the next time step still in the proximity of the last position. If the Courant–Friedrichs–Lewy (CFL) condition is fulfilled, i.e., $u\Delta t/\Delta x \leq 1$, they can even only be located in directly neighboring cells. In [195], algorithms are presented that allow to obtain a good initial guess, which neighboring cell is the relevant one. [207] discusses how to determine cells and reference positions for sliding interfaces, needed, e.g., for the simulation of rotating machines like turbines.

Note: Alternatively to the distributed R-tree based on bounding boxes around local domains, one can use other approaches for distributed search. One of them is, e.g., based on a coarse Cartesian mesh with Cartesian partitioning. The partition and the corresponding cell can be found trivially. In such a context, cells are referred to as bins. Example applications can be found in [208, 209]. Alternatively, there are techniques that allow to determine the owners of points on locally-refined meshes based on forest-of-trees approaches and space-filling-curve partitioning [210, 211].

⁸The coarse search determines, for each point, a list of processes that might own it. The subsequent fine search by each process determines whether the processes actually own these points. The sequence of request (“Does the process own the point?”) and answer (“Yes.”/“No.”) can be realized efficiently, based on consensus-based algorithms for dynamic sparse communications [206].

3.4.3 Efficient operator evaluation

Once all points within a cell, including their reference position, have been determined, the actual evaluation can be generally performed, according to (2.13a), via

$$\hat{u}_{K,q} = \sum_{0 \leq i < N_{\text{DoFs}}} \hat{\phi}_i(\hat{\mathbf{x}}_0^q, \hat{\mathbf{x}}_1^q, \hat{\mathbf{x}}_2^q) u_{K,j},$$

or, for tensor-product elements according to (2.14b), via

$$\hat{u}_{K,q} = \sum_{0 \leq k < N_{\text{DoFs}}^{\text{1D}}} \hat{\phi}_k(\hat{\mathbf{x}}_2^q) \sum_{0 \leq j < N_{\text{DoFs}}^{\text{1D}}} \hat{\phi}_j(\hat{\mathbf{x}}_1^q) \sum_{0 \leq i < N_{\text{DoFs}}^{\text{1D}}} \hat{\phi}_i(\hat{\mathbf{x}}_0^q) u_{K,ijk},$$

which allows sum factorization [172]. The shape functions evaluated at the quadrature points can be tabulated. In tensor-product notation, this implies

$$\hat{u}_{K,q} = \sum N_{q,i} u_{K,i} = N_z^q (I_z \otimes N_y^q) (I_z \otimes I_y \otimes N_x^q) u_K, \quad (3.12)$$

with $N_{\square}^q \in \mathbb{R}^{1 \times N_{\text{DoFs}}^{\text{1D}}}$ containing the tabulated values for point q in a given direction. This implies the computational complexity $\mathcal{O}((k+1)^d)$ for each evaluation point. In the case that the solution has to be evaluated at $(k+1)^d$ evaluation points, the computational complexity $\mathcal{O}(k^{2d}) \gg \mathcal{O}(dk^{d+1})$ is quite high, compared to the sum-factorization case. Computations can be accelerated, e.g., by vectorization over points. This implies loop unrolling by processing multiple rows of N at once. Similar reasoning also applies to computation of gradients and to testing, as needed, e.g., by SIM (3.11), CutFEM (3.10), and CutDG.

Note: The need to evaluate the solution sequentially at quadrature points implies that a pointwise view—as discussed in Section 2.11—can be adopted naturally, where evaluation, operation at the quadrature point and integration are performed in one go before continuing with the next point.

3.4.4 Software

Similarly to the regular case (Section 2 in general and Section 2.9 in detail), one can derive, in the non-nested case, classes that have similar tasks. In particular, one needs: 1) to iterate over cells that own points at which evaluation has to be performed and 2) to perform the actual evaluation on a cell level. Optionally, 3) processes have to exchange solutions on the point level and 4) one needs $|J(x_q)| w_q$.

Listing 3.1 shows, as an example, a deal.II code that sharply computes the right-hand-side vector of the Navier–Stokes equations due to surface tension according to (3.11). The interface is tracked explicitly by a distributed

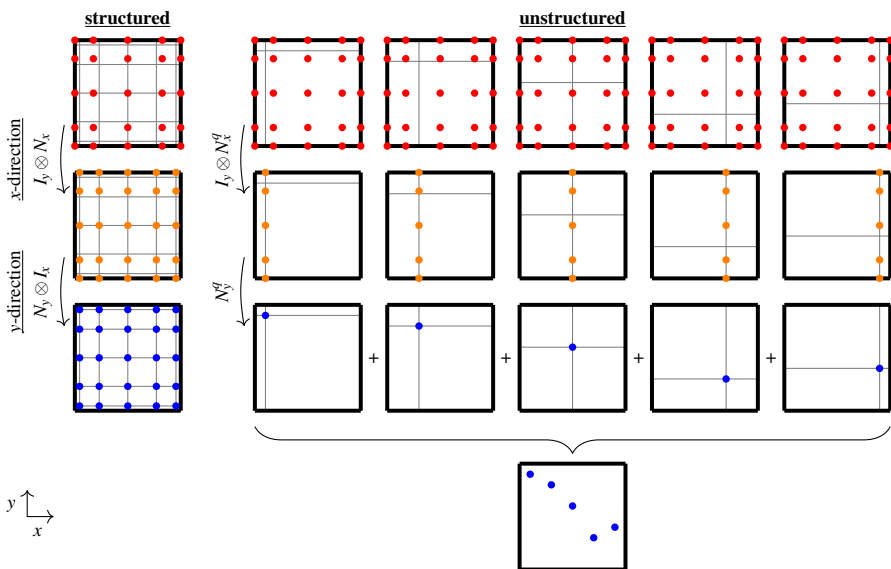


Figure 3.6. Interpolation from Gauss–Lobatto points to Gauss–Legendre points (left) and to a set of 5 arbitrary points (right).

codim-1 mesh, which predefines the quadrature points including the weights. The code assumes that the data structures already have been initialized, the term $\frac{1}{W_e} \kappa(\mathbf{x}_q) \vec{n}(\mathbf{x}_q) |J(\mathbf{x}_q)| w_q$ has been evaluated locally on the quadrature-point level of the codim-1 mesh, and the values are stored in `forces_codim`. The two relevant classes in the listing are `RemotePointEvaluation` and `FEPointEvaluation`. On an abstract level, the tasks of the class `RemotePointEvaluation` are similar to those of `MatrixFree`: it is responsible for the cell loop and the vector communication (exchange of `forces_codim` \rightarrow `forces_background`). It also provides access to information on the evaluation-point level within the cell, like reference positions, which we directly use in the example code to initialize `FEPointEvaluation`. This class is the unstructured equivalent of `FEEvaluation` and used here only for testing. During its initialization, the shape functions are tabulated. For tensor-product elements, only 1D shape functions are tabulated according to (3.12). The operation is finalized by adding the cell-local results to the global vector, for which we use basic `deal.II` infrastructure.

Note: The matrix-free non-matching infrastructure of `deal.II` is—at the time of writing this thesis—still a work in progress, and it is not yet as mature as the implementation of the structured one. A complication is the distinct difference between the application cases, which allows to reuse the infrastructure but does not allow to provide a unified workflow. Which building blocks need to be exposed to the users and what the optimal interface is, is a current research emphasis.

Listing 3.1. Implementation highlighting the interaction of RemotePoint-Evaluation and FEPointEvaluation. As an example, the testing of forces (on a background mesh, given a non-matching codim-1 mesh) is shown.

```

using value_type = Tensor<1, dim, Number>; // data type: forces

// vector containing the value of forces evaluated at the quadrature
// points on codim-1 mesh (filling of forces_codim: not shown)
std::vector<value_type> forces_codim;

// operation on background mesh
const auto evaluation_function = [&](
    const auto &forces_background /*sorted according to points*/,
    const auto &cell_data /*data needed on point level*/) {
    FEPointEvaluation<dim, dim, dim, Number> evaluator(
        mapping, fe, update_values); // evaluator class

    for (unsigned int i = 0; i < cell_data.cells.size(); ++i)
    {
        // initialize evaluator with the current cell and the
        // reference positions
        typename DoFHandler<dim>::active_cell_iterator cell = {
            tria, cell_data.cells[i].first,
            cell_data.cells[i].second, &dof_handler_coarse};

        const ArrayView<const Point<dim>> unit_points(
            cell_data.reference_point_values.data() + q_begin,
            q_end - q_begin);

        evaluator.reinit(cell, unit_points);

        // submit values
        for (const auto q : evaluator.quadrature_point_indices())
            evaluator.submit_value(forces_background[q + begin], q);

        // test and sum into global vector
        solution_values.resize(fe.n_dofs_per_cell());
        evaluator.integrate(solution_values, EvaluationFlags::values);
        cell->distribute_local_to_global(solution_values, dst);
    }
};

rpe.template process_and_evaluate<value_type>(forces_codim, buffer,
                                             evaluation_function);

```

3.4.5 Black-box coupling via preCICE

So far, this section has proposed a general-purpose framework for computations on non-matching grids with unstructured evaluation points. For this purpose, we exploited the following characteristics of the structure of elements: 1) a point within a cell is only influenced by DoFs of the cell, which results in a natural blocking, and 2) the tensor-product structure (of the underlying

shape functions) allows for optimizations on the cell level. This approach has many advantages, e.g., higher-order interpolation is easily possible [175] and intersections can be created. Such an approach is called *white-box coupling* in the literature [175] sometimes.

As an alternative to such a domain-specific approach, it is also possible to rely purely on point positions and point values in a *black-box* fashion. For this, nearest-neighbor and radial-basis-function-based mapping [177] are applicable [175]. An extension of the nearest-neighbor approach is the nearest-projection mapping, where the nearest mesh element is used.

These approaches are applicable to any method (e.g., DEM, FEM, finite difference methods, finite volume methods) and are used, e.g., in the open-source library `preCICE` [170, 171] to couple different executables, which potentially discretize different PDEs with different numerical approaches. Similarly to the point-to-point interpolation discussed above, these approaches might suffer from aliasing effects, which can be solved, e.g., by subsampling. For this, [175] proposed the increase of the evaluation points by a factor of three in each direction. Please note that `preCICE` is not limited to the task of data mapping but also provides many useful other features, e.g., quasi-Newton methods and wavefront relaxations [212, 213].

There exists an interface (wrapper) that allows to use `preCICE` with `deal.II` applications.⁹ The task of it, in particular, is the conversion of data structures (e.g., vector of points to vector of doubles). David Schneider has, jointly with the author of this thesis, integrated `preCICE` into the fluid and the structure modules of `ExaDG`. This allows to couple these modules to external solvers like `OpenFOAM`, as demonstrated in [214]. Furthermore, `preCICE` can be used as an alternative to the native `ExaDG` FSI coupling implementation, which is based on the above-described modules and couples the fluid and the structure modules of `ExaDG` directly via `deal.II` data structures.

We conclude this section with the comment that the above-described data-mapping modules are extendible to the nearest-neighbor approach by generalizing the search algorithm (see Section 3.4.2).

⁹<https://github.com/precice/dealii-adapter>

4. Additional publications and software

4.1 List of papers

Beside the main papers I–VII, the author of this thesis has contributed to following 12 publications (sorted according to the submission date):

- [1] M. Kronbichler, K. Kormann, N. Fehn, P. Munch, and J. Witte, “A Hermite-like basis for faster matrix-free evaluation of interior penalty discontinuous Galerkin operators,” *arXiv:1907.08492*, 2019.
- [2] D. Arndt, N. Fehn, G. Kanschat, K. Kormann, M. Kronbichler, P. Munch, W. A. Wall, and J. Witte, “ExaDG: High-order discontinuous Galerkin for the exa-scale,” in *Software for Exascale Computing - SPPEXA 2016-2019. Lecture Notes in Computational Science and Engineering* (H.-J. Bungartz, S. Reiz, B. Uekermann, P. Neumann, and W. E. Nagel, eds.), vol. 136, pp. 189–224, Cham: Springer, 2020.
- [3] N. Fehn, P. Munch, W. A. Wall, and M. Kronbichler, “Hybrid multigrid methods for high-order discontinuous Galerkin discretizations,” *Journal of Computational Physics*, vol. 415, p. 109538, 2020.
- [4] D. Arndt, W. Bangerth, B. Blais, T. C. Clevenger, M. Fehling, A. V. Grayver, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, R. Rastak, I. Thomas, B. Turcksin, Z. Wang, and D. Wells, “The deal.II library, version 9.2,” *Journal of Numerical Mathematics*, vol. 28, no. 3, pp. 131–146, 2020.
- [5] N. Fehn, M. Kronbichler, P. Munch, and W. A. Wall, “Numerical evidence of anomalous energy dissipation in incompressible Euler flows: Towards grid-converged results for the inviscid Taylor–Green problem,” *Journal of Fluid Mechanics*, vol. 932, pp. A40:1–37, 2022.
- [6] S. Golshan, P. Munch, R. Gassmüller, M. Kronbichler, and B. Blais, “Lethe-DEM: an open-source parallel discrete element solver with load balancing,” *Computational Particle Mechanics*, vol. 10, no. 1, pp. 77–96, 2023.
- [7] D. Arndt, W. Bangerth, B. Blais, M. Fehling, R. Gassmüller, T. Heister, L. Heltai, U. Köcher, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, S. Proell, K. Simon, B. Turcksin, D. Wells, and J. Zhang, “The deal.II library, version 9.3,” *Journal of Numerical Mathematics*, vol. 29, no. 3, pp. 171–186, 2021.
- [8] M. Kronbichler, N. Fehn, P. Munch, M. Bergbauer, K.-R. Wichmann, C. Geitner, M. Allalen, M. Schulz, and W. A. Wall, “A next-generation

discontinuous Galerkin fluid dynamics solver with application to high-resolution lung airflow simulations,” in *SC’ 21: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (B. R. de Supinski, M. Hall, and T. Gamblin, eds.), (St. Louis, MO), pp. 21:1–15, Association for Computing Machinery, 2021.

- [9] D. Arndt, W. Bangerth, M. Feder, M. Fehling, R. Gassmüller, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, S. Sticko, B. Turcksin, and D. Wells, “The deal.II library, version 9.4,” *Journal of Numerical Mathematics*, vol. 30, no. 3, pp. 231–246, 2022.
- [10] J. Heinz, P. Munch, and M. Kaltenbacher, “High-order non-conforming discontinuous Galerkin methods for the acoustic conservation equations,” *International Journal for Numerical Methods in Engineering*, vol. 124, no. 9, pp. 2034–2049, 2022.
- [11] S. D. Proell, P. Munch, W. A. Wall, and C. Meier, “A highly efficient computational framework for fast scan-resolved simulations of metal additive manufacturing processes on the scale of real parts,” *arXiv:2302.05164*, 2023.
- [12] D. Arndt, W. Bangerth, M. Bergbauer, M. Feder, M. Fehling, J. Heinz, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, B. Turcksin, D. Wells, and S. Zampini, “The deal.II library, version 9.5,” *Journal of Numerical Mathematics*, 2023, submitted. <https://dealii.org/deal95-preprint.pdf>.

4.2 List of publicly available software

The author of this thesis made his work available to the public by releasing the benchmark codes used in the papers on GitHub and by contributing to open-source projects. In particular, the author worked on the open-source finite-element library `deal.II` with the following major contributions:

- simplex and mixed-mesh support including matrix-free evaluation [7, 9],
- global-coarsening multigrid (Paper III, Paper V) and additions to linear-algebra computations (Paper IV),
- additions to the matrix-free infrastructure (Paper I, Paper II, Paper VI),
- dynamic-sparse communication patterns based on consensus-based algorithms [4, 9],
- additions to the non-matching support [9, 12], and
- distributed vectors using MPI-3.0 features to access data on the same compute node directly (Paper I, Paper V, Paper VI).

The author was appointed to one of the principal developers of the library in 2020. In this role, he was responsible for organizing user workshops, reviewing patches, and helping users online. The work is documented in the yearly `deal.II` release papers [4, 7, 9, 12].

Furthermore, the author contributed to the following deal.II-based application codes directly or indirectly by providing or extending features in deal.II:

- `adaflo` [198]¹: a two-phase flow solver consisting of a variable-coefficient incompressible Navier–Stokes solver and level-set and phase-field solvers for capturing interface motion. The author has restructured that code in such a way that each module can be used by external packages to construct user-tailored multiphase solvers, as needed by the project `MeltPoolDG`, a multiphysics code targeting the simulation of melt-pool processes relevant for additive manufacturing. Furthermore, he added experimental support for simplex meshes and sharp-interface methods.
- `ExaDG` [2, 33]²: a multiphysics framework with the focus on the solution of incompressible Navier–Stokes equations with high-order discontinuous Galerkin methods. The author contributed, e.g., the global coarsening multigrid infrastructure (Paper III), many of the matrix-based algorithms for hybrid multigrid, as well as distributed non-matching search and communication algorithms.
- `HPSint`³: a phase-field-based sintering code targeting the simulation of thousands of grains on supercomputers. The author worked on high-performance implementations of the nonlinear operators, the derivation of efficient linear and nonlinear solvers, as well as efficient communication patterns for grain tracking (Paper VII).
- `hyper.deal`⁴: a library providing modules that allow to efficiently simulate PDEs for higher dimensions, up to 6D. The author developed appropriate interfaces between low- and higher-dimensional libraries, performed node-level and parallel optimizations to allow for simulations with $5 \cdot 10^{12}$ unknowns on half of SuperMUC-NG (Paper VI).

¹<https://github.com/kronbichler/adaflo>

²<https://github.com/exadg/exadg>

³<https://github.com/hpsint/hpsint>

⁴<https://github.com/hyperdeal/hyperdeal>

5. Conclusions & Outlook

This thesis follows a long tradition in numerical analysis whereby new mathematical techniques are developed in order to solve linear systems of increasing sizes on newly arising computer hardware. Within this topic, we concentrate on the development of algorithms for matrix-free finite-element computations on modern CPU-based hardware, at extreme scale, and for challenging applications. After analyzing the general matrix-free algorithm from a software perspective and identifying common building blocks, we discussed performance optimizations for high-order finite-element computations and multigrid methods as well as presented novel use cases of matrix-free computations.

The performance optimizations included software patterns for improving cache locality by interleaving compute-heavy cell loops with memory-bound vector updates, which we used to accelerate a preconditioned conjugate gradient solver. We derived an algorithm that allows us to efficiently resolve hanging-node constraints related to 137 possible refinement configurations during matrix-free cell loops in 3D by decoupling the hanging-node constraints from the remaining constraints and performing inplace interpolations.

Matrix-free algorithms were applied for the construction of fast transfer operators in the context of h - and p -multigrid. For locally refined meshes, we investigated how the choice of constructing multigrid levels influences the performance of the transfer operator and the multigrid algorithm in total, pointing out the advantages of global-coarsening algorithms in a parallel setting. We developed, for the multigrid algorithm, cache-efficient block preconditioners based on additive Schwarz methods, fast diagonalization methods, and Chebyshev iterations. Furthermore, we applied the multigrid solver in the context of a stage-parallel implicit Runge–Kutta implementation, for which we demonstrated the possibility of shifting the scaling limit compared to stage-serial implementations.

In addition, we used matrix-free algorithms in the context of new applications, to which we needed to adjust them. Firstly, we solved the six-dimensional Vlasov–Poisson equation. The adequate extension of the matrix-free algorithm is rendered possible by looping over cell pairs in phase space. Tensor-product cell-evaluation routines are straightforward to apply in any dimension; however, large working sets of the intermediate quantities on the cell level require interleaving of evaluation and quadrature routines. Excellent performance was achieved for up to $5 \cdot 10^{12}$ DoFs. Secondly, we simulated many-particle solid-state sintering processes, using a phase-field description. In such a context, the number of components is typically higher than ten and varies

over time. We presented holistic optimizations, which include the extension of the matrix-free algorithm and the development of efficient preconditioners. Thirdly, we adopted the matrix-free algorithm in the context of non-matching grids, where fast distributed search routines and fast evaluation routines at arbitrary points are needed. The application of the developed algorithms to solving coupled multiphysics problems, such as fluid-structure interaction and two-phase flow, is promising.

Matrix-free algorithms are not new in the field of the spectral element and finite element methods, and their capability of improving performance of the solution process has been demonstrated several times in the literature including publications by the author of this thesis. Nevertheless, the application of matrix-free algorithms is not yet common practice. There are multiple reasons for this. For instance, the way of expressing matrix-free operators differs from the traditional assembly loop taught in basic finite-element courses. Furthermore, the fact that no matrix is assembled does not allow for the application of general-purpose linear-algebra packages but requires the development of problem-specific preconditioners. Moreover, some features that are commonly used in the context of traditional finite element methods have not yet been ported to the matrix-free context. Others are not even portable. For the latter, alternative methods need to be derived.

Current research efforts by the developers of major open-source finite-element libraries try to address this issue by providing user-friendly interfaces as well as optimized off-the-shelf building blocks and preconditioners. However, we believe that convincing users to consider matrix-free algorithms in their scientific work as a toolbox for numerical analysis depends on success stories in novel application areas, proving that these algorithms are fast and applicable to challenging problems. Our previous and ongoing work on computations in plasma physics, material science, and multiphysics points in that direction.

References

- [1] M. Kronbichler, K. Kormann, N. Fehn, P. Munch, and J. Witte, “A Hermite-like basis for faster matrix-free evaluation of interior penalty discontinuous Galerkin operators,” *arXiv preprint arXiv:1907.08492*, 2019.
- [2] D. Arndt, N. Fehn, G. Kanschat, K. Kormann, M. Kronbichler, P. Munch, W. A. Wall, and J. Witte, “ExaDG: High-order discontinuous Galerkin for the exa-scale,” in *Software for Exascale Computing - SPPEXA 2016-2019. Lecture Notes in Computational Science and Engineering* (H.-J. Bungartz, S. Reiz, B. Uekermann, P. Neumann, and W. E. Nagel, eds.), vol. 136, pp. 189–224, Cham: Springer, 2020.
- [3] N. Fehn, P. Munch, W. A. Wall, and M. Kronbichler, “Hybrid multigrid methods for high-order discontinuous Galerkin discretizations,” *Journal of Computational Physics*, vol. 415, p. 109538, 2020.
- [4] D. Arndt, W. Bangerth, B. Blais, T. C. Clevenger, M. Fehling, A. V. Grayver, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, R. Rastak, I. Thomas, B. Turcksin, Z. Wang, and D. Wells, “The deal.II library, version 9.2,” *Journal of Numerical Mathematics*, vol. 28, no. 3, pp. 131–146, 2020.
- [5] N. Fehn, M. Kronbichler, P. Munch, and W. A. Wall, “Numerical evidence of anomalous energy dissipation in incompressible Euler flows: Towards grid-converged results for the inviscid Taylor–Green problem,” *Journal of Fluid Mechanics*, vol. 932, pp. A40:1–37, 2022.
- [6] S. Golshan, P. Munch, R. Gassmüller, M. Kronbichler, and B. Blais, “Lethe-DEM: An open-source parallel discrete element solver with load balancing,” *Computational Particle Mechanics*, vol. 10, no. 1, pp. 77–96, 2023.
- [7] D. Arndt, W. Bangerth, B. Blais, M. Fehling, R. Gassmüller, T. Heister, L. Heltai, U. Köcher, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, S. Proell, K. Simon, B. Turcksin, D. Wells, and J. Zhang, “The deal.II library, version 9.3,” *Journal of Numerical Mathematics*, vol. 29, no. 3, pp. 171–186, 2021.
- [8] M. Kronbichler, N. Fehn, P. Munch, M. Bergbauer, K.-R. Wichmann, C. Geitner, M. Allalen, M. Schulz, and W. A. Wall, “A next-generation discontinuous Galerkin fluid dynamics solver with application to high-resolution lung airflow simulations,” in *SC’ 21: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (B. R. de Supinski, M. Hall, and T. Gamblin, eds.), (St. Louis, MO), pp. 21:1–15, Association for Computing Machinery, 2021.
- [9] D. Arndt, W. Bangerth, M. Feder, M. Fehling, R. Gassmüller, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, S. Sticker, B. Turcksin, and D. Wells, “The deal.II library, version 9.4,” *Journal of Numerical Mathematics*, vol. 30, no. 3, pp. 231–246, 2022.

- [10] J. Heinz, P. Munch, and M. Kaltenbacher, “High-order non-conforming discontinuous Galerkin methods for the acoustic conservation equations,” *International Journal for Numerical Methods in Engineering*, vol. 124, no. 9, pp. 2034–2049, 2022.
- [11] S. D. Proell, P. Munch, W. A. Wall, and C. Meier, “A highly efficient computational framework for fast scan-resolved simulations of metal additive manufacturing processes on the scale of real parts,” *arXiv preprint arXiv:2302.05164*, 2023.
- [12] D. Arndt, W. Bangerth, M. Bergbauer, M. Feder, M. Fehling, J. Heinz, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, B. Turcksin, D. Wells, and S. Zampini, “The deal.II library, version 9.5,” *Journal of Numerical Mathematics*, vol. 31, no. 3, pp. 231–246, 2023.
- [13] J. H. Wilkinson, “Error analysis of direct methods of matrix inversion,” *Journal of the ACM (JACM)*, vol. 8, no. 3, pp. 281–330, 1961.
- [14] E. Cuthill and J. McKee, “Reducing the bandwidth of sparse symmetric matrices,” in *Proceedings of the 1969 24th national conference* (S. L. Pollack, ed.), pp. 157–172, Association for Computing Machinery, 1969.
- [15] M. R. Hestenes, E. Stiefel, *et al.*, “Methods of conjugate gradients for solving linear systems,” *Journal of research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952.
- [16] Y. Saad and M. H. Schultz, “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems,” *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.
- [17] S. A. Orszag, “Spectral methods for problems in complex geometries,” *Journal of Computational Physics*, vol. 37, no. 1, pp. 70–92, 1980.
- [18] P. F. Fischer, “Analysis and application of a parallel spectral element method for the solution of the Navier-Stokes equations,” *Computer Methods in Applied Mechanics and Engineering*, vol. 80, no. 1-3, pp. 483–491, 1990.
- [19] P. F. Fischer and A. T. Patera, “Parallel spectral element solution of the Stokes problem,” *Journal of Computational Physics*, vol. 92, no. 2, pp. 380–421, 1991.
- [20] P. F. Fischer, L.-W. Ho, G. E. Karniadakis, E. M. Rønquist, and A. T. Patera, “Recent advances in parallel spectral element simulation of unsteady incompressible flows,” *Computers & Structures*, vol. 30, no. 1-2, pp. 217–231, 1988.
- [21] P. F. Fischer, G. W. Kruse, and F. Loth, “Spectral element methods for transitional flows in complex geometries,” *Journal of scientific computing*, vol. 17, pp. 81–98, 2002.
- [22] H. M. Tufo and P. F. Fischer, “Terascale spectral element algorithms and implementations,” in *SC’99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing* (C. Pancake, ed.), (Portland, OR), pp. 68:1–14, Association for Computing Machinery, 1999.
- [23] J. Brown, “Efficient nonlinear solvers for nodal high-order finite elements in 3D,” *Journal of Scientific Computing*, vol. 45, pp. 48–63, 2010.
- [24] M. G. Knepley, J. Brown, K. Rupp, and B. F. Smith, “Achieving high performance with unified residual evaluation,” *arXiv preprint arXiv:1309.1204*, 2013.

- [25] M. Kronbichler and K. Kormann, “A generic interface for parallel cell-based finite element operator application,” *Computers & Fluids*, vol. 63, pp. 135–147, 2012.
- [26] P. E. Vos, S. J. Sherwin, and R. M. Kirby, “From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations,” *Journal of Computational Physics*, vol. 229, no. 13, pp. 5161–5181, 2010.
- [27] C. D. Cantwell, S. J. Sherwin, R. M. Kirby, and P. H. Kelly, “From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements,” *Computers & Fluids*, vol. 43, no. 1, pp. 23–28, 2011.
- [28] C. D. Cantwell, S. J. Sherwin, R. M. Kirby, and P. H. Kelly, “From h to p efficiently: Selecting the optimal spectral/hp discretisation in three dimensions,” *Mathematical Modelling of Natural Phenomena*, vol. 6, no. 3, pp. 84–96, 2011.
- [29] K. Ljungkvist, “Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes,” in *HPC '17: Proceedings of the 25th High Performance Computing Symposium* (L. Polok, M. Sosonkina, W. I. Thacker, and J. Weinbub, eds.), (Virginia Beach, VA), pp. 1:1–12, Society for Computer Simulation International, 2017.
- [30] T. Kolev, P. Fischer, M. Min, J. Dongarra, J. Brown, V. Dobrev, T. Warburton, S. Tomov, M. S. Shephard, *et al.*, “Efficient exascale discretizations: High-order finite element methods,” *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 527–552, 2021.
- [31] N. Chalmers, A. Mishra, D. McDougall, and T. Warburton, “HipBone: A performance-portable graphics processing unit-accelerated C++ version of the NekBone benchmark,” *The International Journal of High Performance Computing Applications*, 2023.
- [32] G. Orlando, A. Della Rocca, P. F. Barbante, L. Bonaventura, and N. Parolini, “An efficient and accurate implicit DG solver for the incompressible Navier-Stokes equations,” *International Journal for Numerical Methods in Fluids*, vol. 94, pp. 1484–1516, 2021.
- [33] B. Krank, N. Fehn, W. A. Wall, and M. Kronbichler, “A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow,” *Journal of Computational Physics*, vol. 348, pp. 634–659, 2017.
- [34] M. O. Deville, P. F. Fischer, and E. H. Mund, *High-order methods for incompressible fluid flow*. Cambridge: Cambridge University Press, 2002.
- [35] N. Fehn, W. A. Wall, and M. Kronbichler, “A matrix-free high-order discontinuous Galerkin compressible Navier-Stokes solver: A performance comparison of compressible and incompressible formulations for turbulent incompressible flows,” *International Journal for Numerical Methods in Fluids*, vol. 89, no. 3, pp. 71–102, 2019.
- [36] D. Davydov, J.-P. Pelteret, D. Arndt, M. Kronbichler, and P. Steinmann, “A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid,” *International Journal for Numerical Methods in Engineering*, vol. 121, no. 13, pp. 2874–2895, 2020.

- [37] J. Brown, V. Barra, N. Beams, L. Ghaffari, M. Knepley, W. Moses, R. Shakeri, K. Stengel, J. L. Thompson, and J. Zhang, “Performance portable solid mechanics via matrix-free p -multigrid,” *arXiv preprint arXiv:2204.01722*, 2022.
- [38] S. DeWitt, S. Rudraraju, D. Montiel, W. B. Andrews, and K. Thornton, “PRISMS-PF: A general framework for phase-field modeling with a matrix-free finite element method,” *npj Computational Materials*, vol. 6, no. 1, pp. 29:1–12, 2020.
- [39] D. Davydov and M. Kronbichler, “Algorithms and data structures for matrix-free finite element operators with MPI-parallel sparse multi-vectors,” *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 3, pp. 1–30, 2020.
- [40] D. Davydov, T. Heister, M. Kronbichler, and P. Steinmann, “Matrix-free locally adaptive finite element solution of density-functional theory with nonorthogonal orbitals and multigrid preconditioning,” *Physica status solidi (b)*, vol. 255, no. 9, pp. 1800069:1–12, 2018.
- [41] N. Kodali, G. Panigrahi, D. Panda, and P. Motamarri, “Fast hardware-aware matrix-free computations of higher-order finite-element discretized matrix multi-vector products,” *arXiv preprint arXiv:2208.07129*, 2022.
- [42] P. C. Africa, M. Salvador, P. Gervasio, L. Dede, and A. Quarteroni, “A matrix-free high-order solver for the numerical solution of cardiac electrophysiology,” *Journal of Computational Physics*, vol. 478, p. 111984, 2023.
- [43] M. Wichrowski, *Fluid-structure interaction problems: velocity-based formulation and monolithic computational methods*. PhD thesis, Polish Academy of Sciences, 2021.
- [44] N. Fehn, *Robust and efficient discontinuous Galerkin methods for incompressible flows*. PhD thesis, Technische Universität München, 2021.
- [45] S. Schoeder, K. Kormann, W. A. Wall, and M. Kronbichler, “Efficient explicit time stepping of high order discontinuous Galerkin schemes for waves,” *SIAM Journal on Scientific Computing*, vol. 40, no. 6, pp. C803–C826, 2018.
- [46] S. Schoeder, W. A. Wall, and M. Kronbichler, “ExWave: A high performance discontinuous Galerkin solver for the acoustic wave equation,” *SoftwareX*, vol. 9, pp. 49–54, 2019.
- [47] T. C. Clevenger and T. Heister, “Comparison between algebraic and matrix-free geometric multigrid for a Stokes problem on adaptive meshes with variable viscosity,” *Numerical Linear Algebra with Applications*, vol. 28, no. 5, pp. e2375:1–13, 2021.
- [48] N. Kohl, D. Thönnies, D. Drzisga, D. Bartuschat, and U. Rüdte, “The HyTeG finite-element software framework for scalable multigrid solvers,” *International Journal of Parallel, Emergent and Distributed Systems*, vol. 34, no. 5, pp. 477–496, 2019.
- [49] M. Kronbichler and K. Kormann, “Fast matrix-free evaluation of discontinuous Galerkin finite element operators,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 3, pp. 1–40, 2019.
- [50] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cervený, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, *et al.*, “MFEM: A modular finite

- element methods library,” *Computers & Mathematics with Applications*, vol. 81, pp. 42–74, 2021.
- [51] S. Müthing, M. Piatkowski, and P. Bastian, “High-performance implementation of matrix-free high-order discontinuous Galerkin methods,” *arXiv preprint arXiv:1711.10885*, 2017.
- [52] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. Kelly, “A study of vectorization for matrix-free finite element methods,” *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 629–644, 2020.
- [53] J. Brown, A. Abdelfattah, V. Barra, N. Beams, J.-S. Camier, V. Dobrev, Y. Dudouit, L. Ghaffari, T. Kolev, D. Medina, *et al.*, “libCEED: Fast algebra for high-order element-based discretizations,” *Journal of Open Source Software*, vol. 6, no. 63, p. 2945, 2021.
- [54] M. Phillips and P. Fischer, “Optimal Chebyshev smoothers and one-sided v-cycles,” *arXiv preprint arXiv:2210.03179*, 2022.
- [55] M. Kronbichler and W. A. Wall, “A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers,” *SIAM Journal on Scientific Computing*, vol. 40, no. 5, pp. A3423–A3448, 2018.
- [56] D. Kempf, R. Heß, S. Müthing, and P. Bastian, “Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 47, no. 1, pp. 1–31, 2020.
- [57] A. Klöckner, “Loo.py: transformation-based code generation for GPUs and CPUs,” in *ARRAY ’14: Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, (Edinburgh), pp. 82–87, Association for Computing Machinery, 2014.
- [58] P. Fischer, M. Min, T. Rathnayake, S. Dutta, T. Kolev, V. Dobrev, J.-S. Camier, M. Kronbichler, T. Warburton, K. Świrydowicz, *et al.*, “Scalability of high-performance pde solvers,” *The International Journal of High Performance Computing Applications*, vol. 34, no. 5, pp. 562–586, 2020.
- [59] A. Abdelfattah, V. Barra, N. Beams, R. Bleile, J. Brown, J.-S. Camier, R. Carson, N. Chalmers, V. Dobrev, Y. Dudouit, *et al.*, “GPU algorithms for efficient exascale discretizations,” *Parallel Computing*, vol. 108, pp. 102841:1–10, 2021.
- [60] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.
- [61] D. A. Kopriva, *Implementing spectral methods for partial differential equations: Algorithms for scientists and engineers*. Berlin: Springer Science & Business Media, 2009.
- [62] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini, “Unified analysis of discontinuous Galerkin methods for elliptic problems,” *SIAM journal on numerical analysis*, vol. 39, no. 5, pp. 1749–1779, 2002.
- [63] D. A. Kopriva and G. J. Gassner, “An energy stable discontinuous Galerkin spectral element discretization for variable coefficient advection problems,” *SIAM Journal on Scientific Computing*, vol. 36, no. 4, pp. A2076–A2099, 2014.

- [64] I. Huismann, J. Stiller, and J. Froehlich, “Efficient high-order spectral element discretizations for building block operators of CFD,” *Computers & Fluids*, vol. 197, p. 104386, 2020.
- [65] K. Kormann and M. Kronbichler, “Parallel finite element operator application: Graph partitioning and coloring,” in *2011 IEEE Seventh International Conference on eScience*, (Stockholm), pp. 332–339, IEEE Computer Society, 2011.
- [66] M. Kronbichler, K. Kormann, I. Pasichnyk, and M. Allalen, “Fast matrix-free discontinuous Galerkin kernels on modern computer architectures,” in *High Performance Computing: 32nd International Conference, ISC High Performance 2017. Lecture Notes in Computer Science* (J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, eds.), vol. 10266, pp. 237–255, Cham: Springer, 2017.
- [67] M. W. Scroggs, J. S. Dokken, C. N. Richardson, and G. N. Wells, “Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 48, no. 2, pp. 1–23, 2022.
- [68] J. R. Magnus and H. Neudecker, *Matrix differential calculus with applications in statistics and econometrics*. John Wiley & Sons, 2019.
- [69] N. Wik, “High-performance implementation of h (div)-conforming elements for incompressible flows,” *Master thesis, Uppsala University*, 2022.
- [70] M. Kronbichler and M. Allalen, “Efficient high-order discontinuous Galerkin finite elements with matrix-free implementations,” in *Advances and New Trends in Environmental Informatics: Managing Disruption, Big Data and Open Science* (H.-J. Bungartz, D. Kranzlmüller, V. Weinberg, J. Weismüller, and V. Wohlgemuth, eds.), pp. 89–110, Cham: Springer, 2018.
- [71] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven, “Nodal discontinuous Galerkin methods on graphics processors,” *Journal of Computational Physics*, vol. 228, no. 21, pp. 7863–7882, 2009.
- [72] J. S. Hesthaven and T. Warburton, *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. New York, NY: Springer Science & Business Media, 2007.
- [73] F. Hindenlang, G. J. Gassner, C. Altmann, A. Beck, M. Staudenmaier, and C.-D. Munz, “Explicit discontinuous Galerkin methods for unsteady problems,” *Computers & Fluids*, vol. 61, pp. 86–93, 2012.
- [74] D. Moxey, R. Amici, and M. Kirby, “Efficient matrix-free high-order finite element evaluation for simplicial elements,” *SIAM Journal on Scientific Computing*, vol. 42, no. 3, pp. C97–C123, 2020.
- [75] G. Karniadakis and S. Sherwin, *Spectral/hp element methods for computational fluid dynamics*. Oxford: Oxford University Press, 2005.
- [76] M. S. Shephard, “Linear multipoint constraints applied via transformation as part of a direct stiffness assembly process,” *International Journal for Numerical Methods in Engineering*, vol. 20, no. 11, pp. 2107–2112, 1984.
- [77] W. Bangerth and O. Kayser-Herold, “Data structures and requirements for hp finite element software,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 36, no. 1, pp. 1–31, 2009.

- [78] M. Min, J. Brown, V. Dobrev, P. Fischer, T. Kolev, D. Medina, E. Merzari, A. Obabko, S. Parker, R. Rahaman, S. Tomov, *et al.*, “ECP milestone report initial integration of CEED software in ECP applications WBS 1.2. 5.3. 04, milestone CEED-MS8,” 2017.
- [79] W. Pazner and P.-O. Persson, “Approximate tensor-product preconditioners for very high order discontinuous Galerkin methods,” *Journal of Computational Physics*, vol. 354, pp. 344–369, 2018.
- [80] P. Bastian, E. H. Müller, S. Müthing, and M. Piatkowski, “Matrix-free multigrid block-preconditioners for higher order discontinuous Galerkin discretisations,” *Journal of Computational Physics*, vol. 394, pp. 417–439, 2019.
- [81] M. Phillips, S. Kerkemeier, and P. Fischer, “Tuning spectral element preconditioners for parallel scalability on GPUs,” in *Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing* (X. Li and K. Teranishi, eds.), (Seattle, WA), pp. 37–48, SIAM, 2022.
- [82] H. Sundar, G. Stadler, and G. Biros, “Comparison of multigrid algorithms for high-order continuous finite element discretizations,” *Numerical Linear Algebra with Applications*, vol. 22, no. 4, pp. 664–680, 2015.
- [83] J. Witte, D. Arndt, and G. Kanschat, “Fast tensor product Schwarz smoothers for high-order discontinuous Galerkin methods,” *Computational Methods in Applied Mathematics*, vol. 21, no. 3, pp. 709–728, 2021.
- [84] W. Couzy, “Spectral element discretization of the unsteady Navier-Stokes equations and its iterative solution on parallel computers,” *Thèse no. 1380, EPFL*, 1995.
- [85] R. E. Lynch, J. R. Rice, and D. H. Thomas, “Direct solution of partial difference equations by tensor product methods,” *Numerische Mathematik*, vol. 6, no. 1, pp. 185–199, 1964.
- [86] O. Axelsson, M. Pourbagher, and D. K. Salkuyeh, “Robust iteration methods for complex systems with an indefinite matrix term,” *arXiv preprint arXiv:2110.00537*, 2021.
- [87] M. Pernice and H. F. Walker, “NITSOL: A Newton iterative solver for nonlinear systems,” *SIAM Journal on Scientific Computing*, vol. 19, no. 1, pp. 302–318, 1998.
- [88] P. N. Brown and Y. Saad, “Hybrid Krylov methods for nonlinear systems of equations,” *SIAM Journal on Scientific and Statistical Computing*, vol. 11, no. 3, pp. 450–481, 1990.
- [89] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, *et al.*, *PETSc users manual*. Argonne National Laboratory, 2019.
- [90] A. T. Chronopoulos and C. W. Gear, “s-step iterative methods for symmetric linear systems,” *Journal of Computational and Applied Mathematics*, vol. 25, no. 2, pp. 153–168, 1989.
- [91] I. Dravins, *Preconditioning for block matrices with square blocks*. PhD thesis, Acta Universitatis Upsaliensis, 2023.
- [92] M. Kronbichler and K. Ljungkvist, “Multigrid for matrix-free high-order finite element computations on graphics processors,” *ACM Transactions on Parallel Computing (TOPC)*, vol. 6, no. 1, pp. 1–32, 2019.

- [93] A. Brandt, “Multi-level adaptive solutions to boundary-value problems,” *Mathematics of computation*, vol. 31, no. 138, pp. 333–390, 1977.
- [94] P. Bastian and C. Wieners, “Multigrid methods on adaptively refined grids,” *Computing in Science & Engineering*, vol. 8, no. 6, pp. 44–54, 2006.
- [95] S. F. McCormick, *Multilevel adaptive methods for partial differential equations*. Philadelphia, PA: SIAM, 1989.
- [96] M. Storti, N. M. Nigro, and S. Idelsohn, “Multigrid methods and adaptive refinement techniques in elliptic problems by finite element methods,” *Computer methods in applied mechanics and engineering*, vol. 93, no. 1, pp. 13–30, 1991.
- [97] S. Lopez and R. Casciaro, “Algorithmic aspects of adaptive multigrid finite element analysis,” *International journal for numerical methods in engineering*, vol. 40, no. 5, pp. 919–936, 1997.
- [98] A. Schüller, *Portable Parallelization of Industrial Aerodynamic Applications (POPINDA): Results of a BMBF Project*. Wiesbaden: Vieweg+Teubner Verlag, 2013.
- [99] G. Kanschat, “Multilevel methods for discontinuous Galerkin FEM on locally refined meshes,” *Computers & structures*, vol. 82, no. 28, pp. 2437–2445, 2004.
- [100] J.-C. Jouhaud, M. Montagnac, and L. P. Tournette, “A multigrid adaptive mesh refinement strategy for 3D aerodynamic design,” *International journal for numerical methods in fluids*, vol. 47, no. 5, pp. 367–385, 2005.
- [101] T. C. Clevenger, T. Heister, G. Kanschat, and M. Kronbichler, “A flexible, parallel, adaptive geometric multigrid method for FEM,” *ACM Transactions on Mathematical Software*, vol. 47, no. 1, pp. 7:1–27, 2020.
- [102] B. Janssen and G. Kanschat, “Adaptive multilevel methods with local smoothing for h^1 - and h^{curl} -conforming high order finite element methods,” *SIAM Journal on Scientific Computing*, vol. 33, no. 4, pp. 2095–2114, 2011.
- [103] R. Becker and M. Braack, “Multigrid techniques for finite elements on locally refined meshes,” *Numerical linear algebra with applications*, vol. 7, no. 6, pp. 363–379, 2000.
- [104] O. Iliev and D. Stoyanov, “Multigrid-adaptive local refinement solver for incompressible flows,” in *International Conference on Large-Scale Scientific Computing* (S. Margenov, J. Wasniewski, and P. Yalamov, eds.), (Berlin), pp. 361–368, Springer, 2001.
- [105] H. Wu and Z. Chen, “Uniform convergence of multigrid v-cycle on adaptively refined finite element meshes for second order elliptic problems,” *Science in China Series A: Mathematics*, vol. 49, no. 10, pp. 1405–1429, 2006.
- [106] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale, “AMReX: A framework for block-structured adaptive mesh refinement,” *Journal of Open Source Software*, vol. 4, no. 37, pp. 1–4, 2019.
- [107] M. Adams, P. Colella, D. T. Graves, J. N. Johnson, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. Sternberg, and B. Van Straalen, “Chombo software package for AMR applications design document,” *Lawrence Berkeley National Laboratory Technical Report*

LBNL-6616E, 2014.

- [108] R. Becker, M. Braack, and T. Richter, "Parallel multigrid on locally refined meshes," in *Reactive Flows, Diffusion and Transport* (W. Jäger, R. Rannacher, and J. Warnatz, eds.), pp. 77–92, Berlin: Springer, 2007.
- [109] E. M. Rønquist and A. T. Patera, "Spectral element multigrid. I. Formulation and numerical results," *Journal of Scientific Computing*, vol. 2, no. 4, pp. 389–406, 1987.
- [110] Y. Maday and R. Munoz, "Spectral element multigrid. II. Theoretical justification," *Journal of Scientific Computing*, vol. 3, no. 4, pp. 323–353, 1988.
- [111] N. Hu and I. N. Katz, "Multi-p methods: Iterative algorithms for the p-version of the finite element analysis," *SIAM Journal on Scientific Computing*, vol. 16, no. 6, pp. 1308–1332, 1995.
- [112] N. Hu, X.-Z. Guo, and I. N. Katz, "Multi-p preconditioners," *SIAM Journal on Scientific Computing*, vol. 18, no. 6, pp. 1676–1697, 1997.
- [113] X.-Z. Guo and I. N. Katz, "Performance enhancement of the multi-p preconditioner," *Computers & Mathematics with Applications*, vol. 36, no. 4, pp. 1–8, 1998.
- [114] X.-Z. Guo and I. N. Katz, "A parallel multi-p method," *Computers & Mathematics with Applications*, vol. 39, no. 9-10, pp. 115–123, 2000.
- [115] H. L. Atkins and B. T. Helenbrook, "Numerical evaluation of p-multigrid method for the solution of discontinuous Galerkin discretizations of diffusive equations," in *17th AIAA Computational Fluid Dynamics Conference*, (Toronto, Ontario Canada), pp. 5110:1–12, American Institute of Aeronautics and Astronautics, 2005.
- [116] B. T. Helenbrook, D. Mavriplis, and H. L. Atkins, "Analysis of "p"-multigrid for continuous and discontinuous finite element discretizations," in *16th AIAA Computational Fluid Dynamics Conference*, (Orlando, Florida), pp. 3989:1–6, American Institute of Aeronautics and Astronautics, 2003.
- [117] B. T. Helenbrook and H. L. Atkins, "Application of p-multigrid to discontinuous Galerkin formulations of the Poisson equation," *AIAA Journal*, vol. 44, no. 3, pp. 566–575, 2006.
- [118] B. T. Helenbrook and H. L. Atkins, "Solving discontinuous Galerkin formulations of Poisson's equation using geometric and p multigrid," *AIAA Journal*, vol. 46, no. 4, pp. 894–902, 2008.
- [119] B. T. Helenbrook and B. S. Mascarenhas, "Analysis of implicit time-advancing p-multigrid schemes for discontinuous Galerkin discretizations of the Euler equations," in *46th AIAA Fluid Dynamics Conference*, (Washington, D.C.), pp. 3494:1–22, American Institute of Aeronautics and Astronautics, 2016.
- [120] B. S. Mascarenhas, B. T. Helenbrook, and H. L. Atkins, "Application of p-multigrid to discontinuous Galerkin formulations of the Euler equations," *AIAA Journal*, vol. 47, no. 5, pp. 1200–1208, 2009.
- [121] B. S. Mascarenhas, B. T. Helenbrook, and H. L. Atkins, "Coupling p-multigrid to geometric multigrid for discontinuous Galerkin formulations of the convection-diffusion equation," *Journal of Computational Physics*, vol. 229, no. 10, pp. 3664–3674, 2010.

- [122] H. Sundar, G. Stadler, and G. Biros, “Comparison of multigrid algorithms for high-order continuous finite element discretizations,” *Numerical Linear Algebra with Applications*, vol. 22, no. 4, pp. 664–680, 2015.
- [123] J. Stiller, “Robust multigrid for high-order discontinuous Galerkin methods: A fast Poisson solver suitable for high-aspect ratio Cartesian grids,” *Journal of Computational Physics*, vol. 327, pp. 317–336, 2016.
- [124] J. Stiller, “Robust multigrid for Cartesian interior penalty DG formulations of the Poisson equation in 3D,” in *Spectral and High Order Methods for Partial Differential Equations ICOSAHOM 2016. Lecture Notes in Computational Science and Engineering* (M. L. Bittencourt, N. A. Dumont, and J. S. Hesthaven, eds.), vol. 119, pp. 189–201, Cham: Springer, 2017.
- [125] B. O’Malley, J. Kópházi, R. P. Smedley-Stevenson, and M. D. Eaton, “P-multigrid expansion of hybrid multilevel solvers for discontinuous Galerkin finite element discrete ordinate (DG-FEM-S_N) diffusion synthetic acceleration (DSA) of radiation transport algorithms,” *Progress in Nuclear Energy*, vol. 98, pp. 177–186, 2017.
- [126] F. Bassi and S. Rebay, “Numerical solution of the Euler equations with a multiorder discontinuous finite element method,” in *Computational Fluid Dynamics 2002* (S. Armfield, ed.), pp. 199–204, Berlin, Heidelberg: Springer, 2003.
- [127] K. Hillewaert, J.-F. Remacle, N. Cheveaugeon, P.-E. Bernard, and P. Geuzaine, “Analysis of a hybrid p-multigrid method for the discontinuous Galerkin discretisation of the Euler equations,” in *Proceedings of the European Conference on Computational Fluid Dynamics* (P. Wesseling, E. Oñate, and J. Périaux, eds.), (Egmond aan Zee, Netherlands), ECCOMAS, 2006.
- [128] C. C. Liang, R. Kannan, and Z. Wang, “A p-multigrid spectral difference method with explicit and implicit smoothers on unstructured triangular grids,” *Computers & Fluids*, vol. 38, no. 2, pp. 254–265, 2009.
- [129] H. Luo, J. D. Baum, and R. Löhner, “Fast p-multigrid discontinuous Galerkin method for compressible flows at all speeds,” *AIAA Journal*, vol. 46, no. 3, pp. 635–652, 2008.
- [130] H. Luo, J. D. Baum, and R. Löhner, “A p-multigrid discontinuous Galerkin method for the Euler equations on unstructured grids,” *Journal of Computational Physics*, vol. 211, no. 2, pp. 767–783, 2006.
- [131] D. Darmofal and K. Fidkowski, “Development of a higher-order solver for aerodynamic applications,” in *42nd AIAA Aerospace Sciences Meeting and Exhibit*, (Reston, VA), pp. 0436:1–11, American Institute of Aeronautics and Astronautics, 2004.
- [132] F. Bassi, A. Ghidoni, S. Rebay, and P. Tesini, “High-order accurate p-multigrid discontinuous Galerkin solution of the Euler equations,” *International journal for numerical methods in fluids*, vol. 60, no. 8, pp. 847–865, 2009.
- [133] C. R. Nastase and D. J. Mavriplis, “High-order discontinuous Galerkin methods using an hp-multigrid approach,” *Journal of Computational Physics*, vol. 213, no. 1, pp. 330–357, 2006.
- [134] S. Premasuthan, C. Liang, A. Jameson, and Z. Wang, “A p-multigrid spectral difference method for viscous compressible flow using 2D quadrilateral meshes,” in *47th AIAA Aerospace Sciences Meeting including The New*

- Horizons Forum and Aerospace Exposition*, (Orlando, FL), pp. 950:1–18, American Institute of Aeronautics and Astronautics, 2009.
- [135] K. J. Fidkowski, T. A. Oliver, J. Lu, and D. L. Darmofal, “p-multigrid solution of high-order discontinuous Galerkin discretizations of the compressible Navier-Stokes equations,” *Journal of Computational Physics*, vol. 207, no. 1, pp. 92–113, 2005.
- [136] A. Ghidoni, A. Colombo, F. Bassi, and S. Rebay, “Efficient p-multigrid discontinuous Galerkin solver for complex viscous flows on stretched grids,” *International Journal for Numerical Methods in Fluids*, vol. 75, no. 2, pp. 134–154, 2014.
- [137] K. Shahbazi, D. J. Mavriplis, and N. K. Burgess, “Multigrid algorithms for high-order discontinuous Galerkin discretizations of the compressible Navier-Stokes equations,” *Journal of Computational Physics*, vol. 228, no. 21, pp. 7917–7940, 2009.
- [138] Z. Jiang, C. Yan, J. Yu, and W. Yuan, “Practical aspects of p-multigrid discontinuous Galerkin solver for steady and unsteady RANS simulations,” *International Journal for Numerical Methods in Fluids*, vol. 78, no. 11, pp. 670–690, 2015.
- [139] M. Kronbichler, T. Heister, and W. Bangerth, “High accuracy mantle convection simulation through modern numerical methods,” *Geophysical Journal International*, vol. 191, no. 1, pp. 12–29, 2012.
- [140] T. Heister, J. Dannberg, R. Gassmüller, and W. Bangerth, “High accuracy mantle convection simulation through modern numerical methods. II: Realistic models and problems,” *Geophysical Journal International*, vol. 210, no. 2, pp. 833–851, 2017.
- [141] M. Fehling, P. Munch, and W. Bangerth, “The deal.II tutorial step-75: parallel hp-adaptive multigrid methods for the Laplace equation,” June 2022. https://www.dealii.org/current/doxygen/deal.II/step_75.html.
- [142] M. Fehling and W. Bangerth, “Algorithms for parallel generic hp-adaptive finite element software,” *ACM Transactions on Mathematical Software (TOMS)*, 2023.
- [143] P. D. Brubeck and P. E. Farrell, “A scalable and robust vertex-star relaxation for high-order FEM,” *SIAM Journal on Scientific Computing*, vol. 44, no. 5, pp. A2991–A3017, 2022.
- [144] P. F. Fischer, H. M. Tufu, and N. Miller, “An overlapping Schwarz method for spectral element simulation of three-dimensional incompressible flows,” in *Parallel Solution of Partial Differential Equations* (P. Bjørstad and M. Luskin, eds.), pp. 159–180, New York: Springer, 2000.
- [145] J. Lottes, “Optimal polynomial smoothers for multigrid v-cycles,” *Numerical Linear Algebra with Applications*, pp. e2518:1–27, 2023.
- [146] J. W. Lottes and P. F. Fischer, “Hybrid multigrid/Schwarz algorithms for the spectral element method,” *Journal of Scientific Computing*, vol. 24, pp. 45–78, 2005.
- [147] W. Pazner and P.-O. Persson, “Stage-parallel fully implicit Runge–Kutta solvers for discontinuous Galerkin fluid simulations,” *Journal of Computational Physics*, vol. 335, pp. 700–717, 2017.

- [148] J. C. Butcher, “On the implementation of implicit Runge–Kutta methods,” *BIT Numerical Mathematics*, vol. 16, no. 3, pp. 237–240, 1976.
- [149] O. Axelsson and M. Neytcheva, “Numerical solution methods for implicit Runge–Kutta methods of arbitrarily high order,” in *ALGORITMY 2020: 21st Conference on Scientific Computing* (P. Frolkovič, K. Mikula, and D. Ševčovič, eds.), (Vysoké Tatry-Podbanské, Slovakia), pp. 11–20, SPEKTRUM STU, 2020.
- [150] O. Axelsson, I. Dravins, and M. Neytcheva, “Stage-parallel preconditioners for implicit Runge–Kutta methods of arbitrary high order. Linear problems,” 2022. <http://www.it.uu.se/research/publications/reports/2022-004/2022-004-nc.pdf>.
- [151] M. Masud Rana, V. E. Howle, K. Long, A. Meek, and W. Milestone, “A new block preconditioner for implicit Runge–Kutta methods for parabolic PDE problems,” *SIAM Journal on Scientific Computing*, vol. 43, no. 5, pp. S475–S495, 2021.
- [152] O. Axelsson, “Global integration of differential equations through Lobatto quadrature,” *BIT – Nordisk Tidskrift for Informationsbehandling*, vol. 4, pp. 69–86, 1964.
- [153] B. S. Southworth, O. Krzysik, W. Pazner, and H. D. Sterck, “Fast solution of fully implicit Runge–Kutta and discontinuous Galerkin in time for numerical PDEs, part I: the linear setting,” *SIAM Journal on Scientific Computing*, vol. 44, no. 1, pp. A416–A443, 2022.
- [154] B. S. Southworth, O. Krzysik, and W. Pazner, “Fast solution of fully implicit Runge–Kutta and discontinuous Galerkin in time for numerical PDEs, part II: Nonlinearities and DAEs,” *SIAM Journal on Scientific Computing*, vol. 44, no. 2, pp. A636–A663, 2022.
- [155] L. E. Cannon, *A cellular computer to implement the Kalman filter algorithm*. Bozeman (MT): Montana State University, 1969.
- [156] F. Hariri, T.-M. Tran, A. Jocksch, E. Lanti, J. Progsch, P. Messmer, S. Brunner, C. Gheller, and L. Villard, “A portable platform for accelerated PIC codes and its application to GPUs using OpenACC,” *Computer Physics Communications*, vol. 207, pp. 69–82, 2016.
- [157] T. A. Dao and M. Nazarov, “A high-order residual-based viscosity finite element method for the ideal MHD equations,” *Journal of Scientific Computing*, vol. 92, no. 3, pp. 77:1–24, 2022.
- [158] T. Pollinger, J. Rentrop, D. Pflüger, and K. Kormann, “A mass-conserving sparse grid combination technique with biorthogonal hierarchical basis functions for kinetic simulations,” *arXiv preprint arXiv:2209.14064*, 2022.
- [159] K. Kormann, “A semi-Lagrangian Vlasov solver in tensor train format,” *SIAM Journal on Scientific Computing*, vol. 37, no. 4, pp. B613–B632, 2015.
- [160] K. Kormann, K. Reuter, and M. Rampp, “A massively parallel semi-Lagrangian solver for the six-dimensional Vlasov–Poisson equation,” *The International Journal of High Performance Computing Applications*, vol. 33, no. 5, pp. 924–947, 2019.
- [161] C. A. Kennedy, M. H. Carpenter, and R. M. Lewis, “Low-storage, explicit Runge–Kutta schemes for the compressible Navier–Stokes equations,” *Applied numerical mathematics*, vol. 35, no. 3, pp. 177–219, 2000.

- [162] M. Loveland, E. Valseth, M. Lukac, and C. Dawson, “Extending FEniCS to work in higher dimensions using tensor product finite elements,” *Journal of Computational Science*, vol. 64, p. 101831, 2022.
- [163] I. Greenquist, M. R. Tonks, L. K. Aagesen, and Y. Zhang, “Development of a microstructural grand potential-based sintering model,” *Computational Materials Science*, vol. 172, p. 109288, 2020.
- [164] S. Ghosh, C. K. Newman, and M. M. Francois, “Tusas: A fully implicit parallel approach for coupled phase-field equations,” *Journal of Computational Physics*, vol. 448, p. 110734, 2022.
- [165] V. Ivannikov, F. Thomsen, T. Ebel, and R. Willumeit-Römer, “Capturing shrinkage and neck growth with phase field simulations of the solid state sintering,” *Modelling and Simulation in Materials Science and Engineering*, vol. 29, no. 7, pp. 075008:1–18, 2021.
- [166] Y. U. Wang, “Computer modeling and simulation of solid-state sintering: A phase field approach,” *Acta Materialia*, vol. 54, no. 4, pp. 953–961, 2006.
- [167] C. Krill III and L.-Q. Chen, “Computer simulation of 3-d grain growth using a phase-field model,” *Acta Materialia*, vol. 50, no. 12, pp. 3059–3075, 2002.
- [168] S. Vedantam and B. S. V. Patnaik, “Efficient numerical algorithm for multiphase field simulations,” *Physical Review E*, vol. 73, no. 1, pp. 016703:1–8, 2006.
- [169] C. J. Permann, M. R. Tonks, B. Fromm, and D. R. Gaston, “Order parameter re-mapping algorithm for 3D phase field model of grain growth using FEM,” *Computational Materials Science*, vol. 115, pp. 18–25, 2016.
- [170] H.-J. Bungartz, F. Lindner, B. Gatzhammer, M. Mehl, K. Scheufele, A. Shukaev, and B. Uekermann, “preCICE – a fully parallel library for multi-physics surface coupling,” *Computers & Fluids*, vol. 141, pp. 250–258, 2016.
- [171] G. Chourdakis, K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H.-J. Bungartz, L. C. Yau, I. Desai, *et al.*, “preCICE v2: A sustainable and user-friendly coupling library,” *arXiv preprint arXiv:2109.14470*, 2021.
- [172] K. Mittal, S. Dutta, and P. Fischer, “Nonconforming Schwarz-spectral element methods for incompressible flow,” *Computers & Fluids*, vol. 191, p. 104237, 2019.
- [173] K. Mittal, S. Dutta, and P. Fischer, “Direct numerical simulation of rotating ellipsoidal particles using moving nonconforming Schwarz-spectral element method,” *Computers & Fluids*, vol. 205, p. 104556, 2020.
- [174] S. Dutta, M. W. V. Moer, P. Fischer, and M. H. Garcia, “Visualization of the Bulle-effect at river bifurcations,” in *PEARC ’18: Proceedings of the Practice and Experience on Advanced Research Computing* (S. Sanielevici, ed.), (Pittsburgh, PA), pp. 107:1–4, 2018.
- [175] F. Lindner, A. Totounferoush, M. Mehl, B. Uekermann, N. E. Pour, V. Krupp, S. Roller, T. Reimann, D. C. Sternel, R. Egawa, *et al.*, “ExaFSA: Parallel fluid-structure-acoustic simulation,” *Software for Exascale Computing*, vol. 136, pp. 271–300, 2020.
- [176] A. de Boer, A. H. van Zuijlen, and H. Bijl, “Comparison of conservative and consistent approaches for the coupling of non-matching meshes,” *Computer*


- Methods in Applied Mechanics and Engineering*, vol. 197, no. 49-50, pp. 4284–4297, 2008.
- [177] F. Lindner, M. Mehl, and B. Uekermann, “Radial basis function interpolation for black-box multi-physics simulations,” in *Coupled problems VII: proceedings of the VII International Conference on Coupled Problems in Science and Engineering* (M. Papadrakakis, ed.), (Rhodes Island, Greece), pp. 50–61, International Center for Numerical Methods in Engineering (CIMNE), 2017.
- [178] M. R. Ross, C. A. Felippa, K. Park, and M. A. Sprague, “Treatment of acoustic fluid–structure interaction by localized Lagrange multipliers: Formulation,” *Computer methods in applied mechanics and engineering*, vol. 197, no. 33-40, pp. 3057–3079, 2008.
- [179] M. R. Ross, M. A. Sprague, C. A. Felippa, and K. Park, “Treatment of acoustic fluid–structure interaction by localized Lagrange multipliers and comparison to alternative interface-coupling methods,” *Computer Methods in Applied Mechanics and Engineering*, vol. 198, no. 9-12, pp. 986–1005, 2009.
- [180] F. Lindner, *Data transfer in partitioned multi-physics simulations: interpolation & communication*. PhD thesis, Universität Stuttgart, 2019.
- [181] V. Krupp, K. Masilamani, H. Klimach, and S. Roller, “Efficient coupling of fluid and acoustic interaction on massive parallel systems,” in *Sustained Simulation Performance 2016: Proceedings of the Joint Workshop on Sustained Simulation Performance, University of Stuttgart (HLRS) and Tohoku University, 2016* (M. M. Resch, W. Bez, E. Focht, N. Patel, and H. Kobayashi, eds.), pp. 61–81, Cham: Springer, 2016.
- [182] S. Roller, J. Bernsdorf, H. Klimach, M. Hasert, D. Harlacher, M. Cakircali, S. Zimny, K. Masilamani, L. Didinger, and J. Zudrop, “An adaptable simulation framework based on a linearized octree,” in *High Performance Computing on Vector Systems 2011* (M. Resch, X. Wang, W. Bez, E. Focht, H. Kobayashi, and S. Roller, eds.), pp. 93–105, Berlin: Springer, 2012.
- [183] A. Fabri and S. Pion, “CGAL: The computational geometry algorithms library,” in *GIS '09: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (O. Wolfson, D. Agrawal, and C.-T. Lu, eds.), (Seattle, WA), pp. 538–539, 2009.
- [184] J. Heinz, N. Fehn, and M. Kaltenbacher, “High-order discontinuous Galerkin methods for the acoustic conservation equations on moving meshes,” in *Fortschritte der Akustik - DAGA 2023: 49. Jahrestagung für Akustik* (O. v. Estorff and S. Lippert, eds.), (Hamburg), pp. 1074–1077, Deutsche Gesellschaft für Akustik e.V, 2023.
- [185] N. Fehn, J. Heinz, W. A. Wall, and M. Kronbichler, “High-order arbitrary Lagrangian–Eulerian discontinuous Galerkin methods for the incompressible Navier–Stokes equations,” *Journal of Computational Physics*, vol. 430, p. 110040, 2021.
- [186] E. Burman, S. Claus, P. Hansbo, M. G. Larson, and A. Massing, “CutFEM: discretizing geometry and partial differential equations,” *International Journal for Numerical Methods in Engineering*, vol. 104, no. 7, pp. 472–501, 2015.
- [187] C. Gürkan, S. Sticko, and A. Massing, “Stabilized CutDG methods for advection-reaction problems,” *SIAM Journal on Scientific Computing*, vol. 42,

- no. 5, pp. A2620–A2654, 2020.
- [188] T. B. Ulfby, A. Massing, and S. Sticks, “Stabilized cut discontinuous Galerkin methods for advection-reaction problems on surfaces,” *Computer Methods in Applied Mechanics and Engineering*, vol. 413, pp. 116109:1–31, 2023.
- [189] N. Moës, J. Dolbow, and T. Belytschko, “A finite element method for crack growth without remeshing,” *International journal for numerical methods in engineering*, vol. 46, no. 1, pp. 131–150, 1999.
- [190] A. Massing, M. G. Larson, and A. Logg, “Efficient implementation of finite element methods on nonmatching and overlapping meshes in three dimensions,” *SIAM Journal on Scientific Computing*, vol. 35, no. 1, pp. C23–C47, 2013.
- [191] R. I. Saye, “High-order quadrature methods for implicitly defined surfaces and volumes in hyperrectangles,” *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. A993–A1019, 2015.
- [192] S. Claus and P. Kerfriden, “A CutFEM method for two-phase flow problems,” *Computer Methods in Applied Mechanics and Engineering*, vol. 348, pp. 185–206, 2019.
- [193] B. Schott, U. Rasthofer, V. Gravemeier, and W. A. Wall, “A face-oriented stabilized nitsche-type extended variational multiscale method for incompressible two-phase flow,” *International Journal for Numerical Methods in Engineering*, vol. 104, no. 7, pp. 721–748, 2015.
- [194] B. Schott, C. Ager, and W. A. Wall, “Monolithic cut finite element-based approaches for fluid-structure interaction,” *International Journal for Numerical Methods in Engineering*, vol. 119, no. 8, pp. 757–796, 2019.
- [195] R. Gassmüller, H. Lokavarapu, E. Heien, E. G. Puckett, and W. Bangerth, “Flexible and scalable particle-in-cell methods with adaptive mesh refinement for geodynamic computations,” *Geochemistry, Geophysics, Geosystems*, vol. 19, no. 9, pp. 3596–3604, 2018.
- [196] T. El Geitani, S. Golshan, and B. Blais, “Toward high-order CFD-DEM: Development and validation,” *Industrial & Engineering Chemistry Research*, vol. 62, no. 2, pp. 1141–1159, 2023.
- [197] B. Blais, L. Barbeau, V. Bibeau, S. Gauvin, T. El Geitani, S. Golshan, R. Kamble, G. Mirakhori, and J. Chaouki, “Lethe: An open-source parallel high-order adaptive CFD solver for incompressible flows,” *SoftwareX*, vol. 12, pp. 100579:1–9, 2020.
- [198] M. Kronbichler, A. Diagne, and H. Holmgren, “A fast massively parallel two-phase flow solver for microfluidic chip simulation,” *The International Journal of High Performance Computing Applications*, vol. 32, no. 2, pp. 266–287, 2018.
- [199] J. U. Brackbill, D. B. Kothe, and C. Zemach, “A continuum method for modeling surface tension,” *Journal of computational physics*, vol. 100, no. 2, pp. 335–354, 1992.
- [200] E. Olsson and G. Kreiss, “A conservative level set method for two phase flow,” *Journal of computational physics*, vol. 210, no. 1, pp. 225–246, 2005.
- [201] C. S. Peskin, “Numerical analysis of blood flow in the heart,” *Journal of computational physics*, vol. 25, no. 3, pp. 220–252, 1977.

- [202] S. O. Unverdi and G. Tryggvason, “A front-tracking method for viscous, incompressible, multi-fluid flows,” *Journal of computational physics*, vol. 100, no. 1, pp. 25–37, 1992.
- [203] F. Henri, M. Coquerelle, and P. Lubin, “Geometrical level set reinitialization using closest point method and kink detection for thin filaments, topology changes and two-phase flows,” *Journal of Computational Physics*, vol. 448, p. 110704, 2022.
- [204] J. H. Bramble, J. E. Pasciak, and J. Xu, “The analysis of multigrid algorithms with nonnested spaces or noninherited quadratic forms,” *Mathematics of Computation*, vol. 56, no. 193, pp. 1–34, 1991.
- [205] M. L. Bittencourt, C. C. Douglas, and R. A. Feijóo, “Nonnested multigrid methods for linear problems,” *Numerical Methods for Partial Differential Equations: An International Journal*, vol. 17, no. 4, pp. 313–331, 2001.
- [206] T. Hoefer, C. Siebert, and A. Lumsdaine, “Scalable communication protocols for dynamic sparse data exchange,” *ACM Sigplan Notices*, vol. 45, no. 5, pp. 159–168, 2010.
- [207] J. Dürrwächter, M. Kurz, P. Kopper, D. Kempf, C.-D. Munz, and A. Beck, “An efficient sliding mesh interface method for high-order discontinuous Galerkin schemes,” *Computers & Fluids*, vol. 217, p. 104825, 2021.
- [208] S. L. Fuchs, C. Meier, W. A. Wall, and C. J. Cyron, “A novel smoothed particle hydrodynamics and finite element coupling scheme for fluid–structure interaction: The sliding boundary particle approach,” *Computer Methods in Applied Mechanics and Engineering*, vol. 383, p. 113922, 2021.
- [209] M. Mayr and A. Popp, “Scalable computational kernels for mortar finite element methods,” *Engineering with Computers*, 2023.
- [210] C. Burstedde, “Parallel tree algorithms for AMR and non-standard data access,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 46, no. 4, pp. 1–31, 2020.
- [211] M. Mirzadeh, A. Guittet, C. Burstedde, and F. Gibou, “Parallel level-set methods on adaptive tree-based grids,” *Journal of Computational Physics*, vol. 322, pp. 345–364, 2016.
- [212] B. Rüth, B. Uekermann, M. Mehl, P. Birken, A. Monge, and H.-J. Bungartz, “Quasi-Newton waveform iteration for partitioned surface-coupled multiphysics applications,” *International Journal for Numerical Methods in Engineering*, vol. 122, no. 19, pp. 5236–5257, 2021.
- [213] P. Meisrimel and P. Birken, “Waveform relaxation with asynchronous time-integration,” *ACM Transactions on Mathematical Software*, vol. 48, no. 4, pp. 1–22, 2022.
- [214] G. Chourdakis, D. Schneider, and B. Uekermann, “OpenFOAM-preCICE: Coupling openfoam with external solvers for multi-physics simulations,” *OpenFOAM® Journal*, vol. 3, pp. 1–25, 2023.

Paper I ██████████

Enhancing data locality of the conjugate gradient method for high-order matrix-free finite-element implementations

The International Journal of High Performance Computing Applications 2022, Vol. 0(0) 1–21
© The Author(s) 2022
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/10943420221107880
journals.sagepub.com/home/hpc


Martin Kronbichler^{1,2} , Dmytro Sashko³ and Peter Munch^{1,4}

Abstract

This work investigates a variant of the conjugate gradient (CG) method and embeds it into the context of high-order finite-element schemes with fast matrix-free operator evaluation and cheap preconditioners like the matrix diagonal. Relying on a data-dependency analysis and appropriate enumeration of degrees of freedom, we interleave the vector updates and inner products in a CG iteration with the matrix-vector product with only minor organizational overhead. As a result, around 90% of the vector entries of the three active vectors of the CG method are transferred from slow RAM memory exactly once per iteration, with all additional access hitting fast cache memory. Node-level performance analyses and scaling studies on up to 147k cores show that the CG method with the proposed performance optimizations is around two times faster than a standard CG solver as well as optimized pipelined CG and *s*-step CG methods for large sizes that exceed processor caches, and provides similar performance near the strong scaling limit.

Keywords

Conjugate gradient method, data locality, matrix-free implementation, sum factorization, strong scaling

1. Introduction

The conjugate gradient (CG) method is one of the most popular algorithms for the iterative solution of large sparse symmetric positive-definite linear systems arising from discretization of partial differential equations (PDE). While it needs to be combined with strong preconditioners such as multigrid when applied to elliptic equations, the conjugate gradient method with simple preconditioners like the matrix diagonal can be the most efficient choice for parabolic partial differential equations with small to moderate time steps. For example, in computational fluid dynamics, many splitting schemes eventually lead to a positive definite Helmholtz-like equation with a mass matrix and a diffusive operator scaled by the time step and viscosity, see, for example, [Tufo and Fischer \(1999\)](#), [Deville et al. \(2002, Section 6.5\)](#), and [Fehn et al. \(2018\)](#) for application in incompressible flows as well as [Demkowicz et al. \(1990\)](#) and [Guermont et al. \(2021\)](#) for compressible flows. Another important application is the projection with consistent finite-element mass matrices, possibly including some regularization through diffusion ([Kronbichler et al., 2018](#)).

In the conjugate gradient method with simple preconditioners, the matrix-vector product has traditionally been the most expensive operation. With the increase in computing

power through parallelism on the one hand and algorithmic progress on the other hand, the matrix-vector product may in fact be so cheap that attention must be turned to the other operations in the CG method.

On large-scale parallel computers, the global reductions involved in the two inner products in each CG iteration are generally seen as the main threat to strong scaling, addressed by the development of lower-synchronization variants, such as the pipelined conjugate gradient method ([Cornelis et al., 2018](#); [Ghysels and Vanroose, 2014](#)) or *s*-step methods ([Chronopoulos and Gear, 1989](#)). These alternatives rely on mathematical transformations of the basic

¹Institute for Computational Mechanics, Technical University of Munich, Germany

²Department of Mathematics, University of Augsburg, Germany

³School of Mechanical and Mining Engineering, The University of Queensland, Saint Lucia, QLD, Australia

⁴Institute of Material Systems Modeling, Helmholtz-Zentrum Hereon, Germany

Corresponding author:

Martin Kronbichler, Institute for Computational Mechanics, Technical University of Munich, Boltzmannstr 15, Garching b, München 85748, Germany.

Email: martin.kronbichler@uni-a.de

CG algorithm with redundant vector operations that break some dependencies. The s -step method not only allows to combine global communication for several CG iterations into one block, but also to schedule the communication of several matrix-vector products together through matrix-power kernels.

The guiding theme of these recent contributions has been the reduction of the *communication latency*, see also [Eller et al. \(2019\)](#) for a broader overview on large-scale methods. However, less attention has been paid to the *throughput of the memory hierarchy*, that is, bandwidth requirements from and to main memory (RAM). This can be the more severe performance limit in a number of applications, especially for solvers that combine different algorithms in tight sequence. One example is incompressible fluid flow discretized with splitting methods, where the pressure Poisson equation solved with multigrid sets the limit for strong scaling, but the much larger symmetric positive-definite system in the velocity contributes with 50% or more of the runtime ([Fehn et al., 2018](#); [Krank et al., 2017](#)). As an illustration, [Figure 1](#) shows the share of runtime of different operations in a preconditioned conjugate gradient solver as a function of problem size on a single compute node. While the matrix-vector product indeed dominates the runtime for small sizes with less than 3 million degrees of freedom, this is not the case for larger sizes relevant to those fluid dynamics applications where AXPY-style vector updates, dot products and the application of the preconditioner take up two thirds of the total run time.

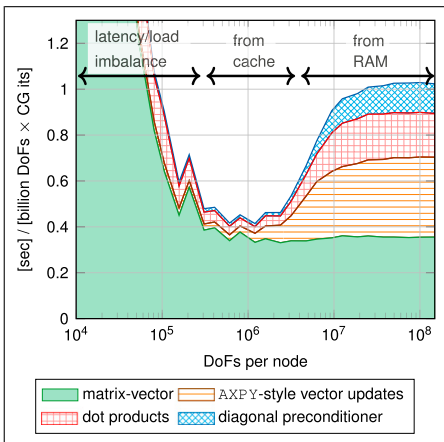


Figure 1. Breakdown of times per CG iteration in the Center of Efficient Exascale Discretizations (CEED) benchmark problem BP4 ([Fischer et al., 2020](#)) with finite elements of degree $p = 5$ on 2×24 cores of Intel Xeon Platinum 8174. See [Section 3.1](#) below for a more detailed explanation of the steps.

The aim of the present work is to design a solver with primary focus on the memory bandwidth behavior of the CG algorithm in the context of high-order finite-element methods implemented with matrix-free sum-factorization algorithms ([Deville et al., 2002](#)). The main novelty is a set of techniques that allow to interleave the vector updates and inner products in a CG iteration with the matrix-vector product for a specific pipelined-like CG formulation originally presented as Algorithm 2.2 in [Chronopoulos and Gear \(1989\)](#). As a result, we are able to perform the access to the three active vectors in inner products and vector updates of a complete CG iteration with a single load from RAM memory for around 90% of the vector entries, serving all other accesses from the fast cache memory on contemporary cache-based CPU architectures. Our experiments show similar performance as for pipelined and s -step methods near the strong scaling limit when all vector entries are hit in the caches, but we reach a significantly higher throughput when the vectors spill out of the caches. While not directly reducing the minimum achievable wall time, our contribution allows to reach a predefined throughput already on a smaller machine.

The proposed techniques rely on introspection of the matrix-vector product and simple preconditioners. The idea of using the structure of the operations in the CG iteration to increase performance is not new and can be traced back to at least [Eisenstat \(1981\)](#). However, the context of minimizing data movement for high-order finite-element solvers within a single iteration appears to be novel. These developments are necessary, because the wide stencils from high-order finite-element methods as well as multi-component systems make traditional optimizations such as matrix-power kernels and temporal wavefront blocking ([Malas et al., 2017](#)) in the context of s -step Krylov methods ineffective.

The implementations used for the present study are available as open-source software on GitHub.¹ They build on the general-purpose finite-element library deal.II ([Arndt et al., 2021](#)) and have been verified on supercomputer scale ([Arndt et al., 2020b](#)). The remainder of this contribution is structured as follows. [Section 2](#) introduces the state of the art of fast matrix-free operator evaluation for higher-order finite-element discretizations. In [Section 3](#), the classical conjugate gradient algorithm as well as pipelined and s -step variants are reviewed in terms of the memory access. [Section 4](#) discusses a variant of CG that avoids the two synchronization points of the conventional CG algorithm when using cheap diagonal preconditioning, whereas [Section 5](#) presents the ingredients necessary to efficiently embed the vector operations into the matrix-vector product. In [Section 6](#), large-scale computations are given to show the effectiveness of the method, before [Section 7](#) summarizes our results.

2. Fast matrix-free operator evaluation

We consider a benchmark problem in the context of high-order finite element methods to investigate the benefits of the proposed techniques, in comparison to well-studied optimized CG alternatives from the literature. It involves the vector-valued Poisson equation in a $d = 3$ dimensional domain $\Omega \subset \mathbb{R}^3$

$$-\nabla^2 \mathbf{u} = \mathbf{f} \quad (1)$$

with the vector field $\mathbf{u}(\mathbf{x}) = (u_1(\mathbf{x}), u_2(\mathbf{x}), u_3(\mathbf{x})) \in (H^1(\Omega))^3$ and a forcing $\mathbf{f} \in (L^2(\Omega))^3$. On the domain boundary $\partial\Omega$, Dirichlet boundary conditions $\mathbf{u} = \mathbf{g}$ are set.

The finite-element discretization is derived from the weak form of equation (1), restricted to a space of polynomials on a mesh of elements Ω_e of the computational domain, $e = 1, \dots, n_{\text{cells}}$. On a hexahedral element Ω_e , the solution interpolation is given by

$$\mathbf{u}_h(\mathbf{x})_{\mathbf{x} \in \Omega_e} = \sum_{j=1}^{3(p+1)^3} \varphi_j(\widehat{\mathbf{x}}(\mathbf{x})) \mathbf{u}_{e,j} \quad (2)$$

Here, $\mathbf{u}_e = [u_{e,j}]$ denotes the vector of unknown coefficients on Ω_e in an expansion with a polynomial basis $\{\varphi_j, j = 1, \dots, 3(p+1)^3\}$. The basis functions are constructed as the tensor product of one-dimensional polynomials of degree p for each of the vector components. Collecting the functions defined on all the elements and inserting the expansions as tentative solutions and test functions into the weak form, we arrive at a matrix system

$$\mathbf{A} \mathbf{u} = \mathbf{b} \quad (3)$$

with a sparse matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, the right-hand-side vector $\mathbf{b} \in \mathbb{R}^n$ and the discrete solution vector $\mathbf{u} \in \mathbb{R}^n$. The number $n \sim 3n_{\text{cells}}p^3$ denotes the number of degrees of freedom (DoFs), counting the unique free coefficients in the expansion. The solution of this matrix system is the subject of the present study.

The relatively dense coupling of degrees of freedom in the matrix stencil makes sparse matrix-vector products in iterative solvers inefficient for higher-order finite elements with degree $p \geq 2$. Considerable speedups can be obtained by replacing the sparse matrix-vector product by a matrix-free evaluation of the action of the matrix on a vector. Whereas stencil-like approaches are most beneficial for the lowest-order elements on structured meshes (Bauer et al., 2018), the method of choice for hexahedral elements with general deformed shapes and higher degrees is to compute the integrals underlying the finite-element method on the fly (Brown, 2010; Deville et al., 2002; Fischer et al., 2020; Kronbichler and Kormann, 2012). The matrix-vector product is computed as a sum of cell-wise contributions

$$\mathbf{v} = \mathbf{A} \mathbf{u} = \sum_{e=1}^{n_{\text{cells}}} \mathbf{P}_e^T \mathbf{A}_e (\mathbf{P}_e \mathbf{u}) \quad (4)$$

where \mathbf{A}_e is the representation of the operator on element Ω_e and \mathbf{P}_e denotes the local-to-global mapping of unknowns such that $\mathbf{u}_e = \mathbf{P}_e \mathbf{u}$ gives restriction of the global solution vector \mathbf{u} to the element. The local operation $\mathbf{A}_e \mathbf{u}_e$ is again implemented in a matrix-free fashion without building the element stiffness matrix \mathbf{A}_e

$$\begin{aligned} [\mathbf{A}_e \mathbf{u}_e]_i &= \int_{\Omega_e} (\nabla \varphi_i)^T \nabla \mathbf{u}_h \, d\mathbf{x} \\ &= \sum_{q=1}^{n_q} \left(\widehat{\nabla} \varphi_i \right)^T \mathbf{J}_{e,q}^{-1} (w_q \det \mathbf{J}_{e,q}) \mathbf{J}_{e,q}^{-T} \sum_{j=1}^{3(p+1)^3} \widehat{\nabla} \varphi_j \mathbf{u}_{e,j} \end{aligned} \quad (5)$$

The integrals are approximated by numerical quadrature on n_q points. In this work, we consider the BP4 benchmark problem proposed by Fischer et al. (2020), which selects the tensor-product Gaussian quadrature formula with $n_q = (p+2)^3$ points $\widehat{\mathbf{x}}_q$ per cell and the associated quadrature weight w_q . The integrals are transformed to reference coordinates $\widehat{\mathbf{x}}$ via a polynomial mapping $\widehat{\mathbf{x}}(\mathbf{x})$ and the derivatives in real space ∇ are transformed to derivatives in reference coordinates $\widehat{\nabla}$ by multiplication with the inverse and transpose of the Jacobian $[\mathbf{J}_e(\widehat{\mathbf{x}})]_{ij} = \frac{\partial x_i}{\partial \widehat{x}_j}$. The local result $\mathbf{A}_e \mathbf{u}_e$ is obtained by evaluating equation (5) for all test functions φ_i , $i = 1, \dots, 3(p+1)^3$.

The efficiency of the matrix-free algorithm (4)–(5) crucially depends on evaluating $\widehat{\nabla} \mathbf{u}_h$ at the quadrature points and the multiplication by the test function gradient $\widehat{\nabla} \varphi_i$ as well as the summation over quadrature points, respectively. For tensor-product shape functions that are integrated on a tensor-product quadrature formula, sum factorization allows to decompose these two steps into a series of one-dimensional interpolations of total cost $\mathcal{O}(p^{d+1})$ per element in d dimensions (or $\mathcal{O}(p)$ per unknown), compared to the naive evaluation cost of $\mathcal{O}(p^{2d})$. The sum-factorization approach has been developed in the context of the spectral element method by Orszag (1980), Patera (1984), and Tufo and Fischer (1999), see also the book by Deville et al. (2002) as well as recent implementation and vectorization studies by Kronbichler and Kormann (2012, 2019), Świrydowicz et al. (2019), Fischer et al. (2020), Sun et al. (2020), Moxey et al. (2020), and Kempf et al. (2021).

2.1. Experimental setup

Our experiments use the implementation of matrix-free operator evaluation in the deal.II finite-element library (Arndt et al., 2020a, 2021), described in Kronbichler and Kormann (2012, 2019). The main computational kernels are fully vectorized across elements, that is, operation (5) is

evaluated on several cells for the different lanes in the single-instruction/multiple-data (SIMD) paradigm, and use an even-odd decomposition (Solomonoff, 1992) in sum factorization to further reduce the arithmetic cost. The solution vectors store unique unknowns, which necessitates indirect addressing for the access of elemental data, represented as a matrix \mathbf{P}_e in equation (4). Indirect addressing involves additional instructions compared to duplicating unknowns shared by several cells as used, for example, in Nek5000 (Fischer et al., 2021), but avoids redundant storage and speeds up the other parts of the solver. In our implementation, the indices describing \mathbf{P}_e use a compressed format of 3^3 four-byte integers, from which all $3 \times (p+1)^3$ indices are deduced on the fly. The meshes are partitioned by space-filling curves according to Bangerth et al. (2011).

The code has been compiled with the GNU compiler g++, version 9.2, with optimization flags `-O3 -march=native -funroll-loops`, which is the compiler with the best performance among GNU, Intel and clang for our code. The experiments have been conducted within a pure MPI setting. To reduce the overhead due to communication between processes within a single compute node, we perform the exchange of ghost values manually via memcpy and MPI-3.0 shared-memory features (Munch et al., 2021), instead of relying on plain MPI_Isend and MPI_Irecv.

Following the benchmark description by Fischer et al. (2020), the algorithms are mainly compared in terms of the throughput, that is, the number of degrees of freedom processed per second (DoFs/s) for one matrix-vector product in this section or one iteration of the conjugate gradient method in the subsequent sections. The throughput is obtained by the ratio of the number of degrees of freedom in the linear system and the measured runtime. The runtime is taken as the minimum of two separate jobs with four experiments each in order to reduce the noise caused by other concurrent jobs on the supercomputer. Apart from isolated outliers, the arithmetic mean of those eight runs is within 2% of the reported minimum.

Unless noted otherwise, the numerical experiments are run on a dual-socket Intel Xeon Platinum 8174 (Skylake) system of the supercomputer SuperMUC-NG.² The CPU cores run at a fixed frequency of 2.3 GHz, which gives an arithmetic peak of 3.5 TFlop/s. The 96 GB of random-access memory (RAM) are connected through 12 channels of DDR4-2666 with a theoretical bandwidth of 256 GB/s and an achieved STREAM triad memory throughput of 205 GB/s.

2.2. Identification of fast matrix-vector product

Contemporary implementations of matrix-free methods with sum factorization often precompute and store the metric terms in $\mathbf{J}_{e,q}^{-1}(w_q \det \mathbf{J}_{e,q}) \mathbf{J}_{e,q}^{-T}$ at each quadrature point and load them during operator evaluation. The precomputed

setup is applicable to deformed (curvilinear) cells and to variable coefficients. As shown in Kronbichler and Ljungkvist (2019), the evaluation (4)–(5) is then memory-bound on modern hardware. For an implementation that aims to maximize the throughput for cell integrals according to Kronbichler and Kormann (2019), it might be more economic to evaluate the metric terms on the fly as well. To identify a suitable method, we compare the following variants regarding the terms representing the geometric factors:

- tri-quadratic geometry evaluated on the fly from $3^3 = 27$ points (“quadratic geomet. compute”), loading 27×3 doubles per cell, giving a matrix-vector product with 395 Flops/DoF for $p = 5$,
- geometry evaluated on the fly from $(p+2)^3$ points at the position of the quadrature points (“isoparametric compute”), loading 3 doubles per quadrature point, yielding 417 Flops/DoF for $p = 5$,
- precompute and load the inverse Jacobian $\mathbf{J}_{e,q}^{-1}$ and the Jacobian determinant times quadrature weight (“inverse Jacobian load”) at each quadrature point, loading 10 doubles per quadrature point, yielding 316 Flops/DoF for $p = 5$,
- precompute and load the final symmetric coefficient tensor, $\mathbf{J}_{e,q}^{-1}(w_q \det \mathbf{J}_{e,q}) \mathbf{J}_{e,q}^{-T}$ (“final tensor load”) at each quadrature point, loading 6 doubles per quadrature point, yielding a matrix-vector product with 267 Flops/DoF for $p = 5$, as done, for example, in Świrydowicz et al. (2019); Fischer et al. (2020).

Figure 2 compares the computational throughput of these variants on a single compute node. The operator evaluation

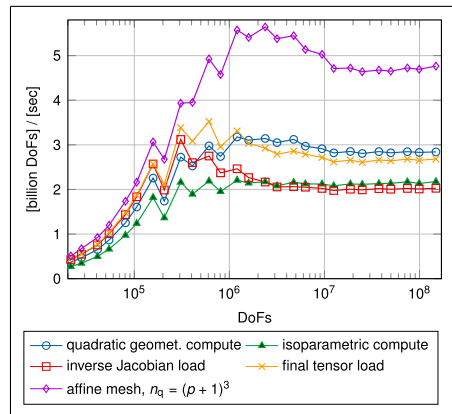


Figure 2. Comparison of different implementations of matrix-free operator evaluation for polynomial degree $p = 5$ on 2×24 cores of Intel Xeon Platinum 8174.

reaches a maximum for intermediate sizes of around 10^6 DoFs when most data fits into caches. As the problem size further increases, data must be fetched from main memory, leading to a slowdown for the cases that are dominated by memory access. We note a slight zig-zag pattern in the reported throughput, which is caused by different costs of ghost exchange, which changes when the number of cells is divisible by 48 leading to cube-like subdomains (higher throughput) or by 64 leading to more irregular MPI subdomains (lower throughput). Figure 2 also presents the throughput of the evaluation on an affine mesh with a constant inverse Jacobian \mathbf{J}^{-1} throughout the whole mesh and using $n_g = (p + 1)^d$ points of Gaussian quadrature, a case studied in detail in Kronbichler and Kormann (2012, 2019). This reduces the arithmetic cost to 206 Flops/DoF and the memory transfer to just the input and output vectors with performance mainly limited by the vector access with indirect addressing.

For large sizes with $n > 10^7$, the “load” variants are memory-limited at slightly more than 200 GB/s, whereas the two “compute” variants involve a memory transfer of 100 GB/s and 140 GB/s for vector sizes of 100 million DoFs, all measured from hardware performance counters with the LIKWID tool (Treibig et al., 2010). Albeit slightly slower than the “load” variants for the in-cache case with $n < 10^6$, this study concentrates on the quadratic geometry representation evaluated on the fly with polynomial degree $p = 5$ for the finite-element expansion (2). The representation of curved geometries differs from the other three options in general, but we argue that a tri-quadratic approximation is nonetheless suitable for many applications. The bulk of a 3D geometry can often be well-represented in such a way, leading to a significant reduction of the memory transfer and cache pressure against the isoparametric high-order case. By contrast, a tri-linear representation (with approximately 10% higher throughput) might be unacceptable in a whole region around strongly curved boundaries. It is conceivable to augment the present strategy with a high-degree (isoparametric) geometry representation of one element layer close to the boundary, without significantly affecting the throughput.

From the throughput values listed in Figure 2 and the operation counts mentioned above, it can be deduced that the matrix-vector product with quadratic geometry runs at 1.1 TFlop/s with 50 million DoFs and at 1.3 TFlop/s with 1.2 million DoFs. While this is clearly below the arithmetic peak of 3.5 TFlop/s, the value is high for this kind of algorithm; the gap to the peak can be explained by the cost of the indirect addressing into the vectors \mathbf{u} , \mathbf{v} , isolated additions and multiplications that cannot be merged into fused multiply-add operations, the throughput of caches, and, for the larger case, insufficient data prefetching from RAM.

The throughput of 2.82 billion DoFs/s with 50 million DoFs for a matrix-free operator evaluation ($p = 5$, quadratic

geometry computation) can be compared to a sparse matrix-vector product: the lowest order $p = 1$ can reach a throughput of between 590 million DoFs/s (separate matrix entries for all three vector components, perfect caching of vector entries) and 1.6 billion DoFs/s (same matrix for all three vector components; only applicable for simple boundary conditions), or between 50 and 147 million DoFs/s for the $p = 5$ case. The effect of high-order matrix-free algorithms being several times faster than low-order matrix-based algorithms on a degree of freedom basis has been

Algorithm 1 Preconditioned conjugate gradient method.

```

1:    $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$ ,  $\mathbf{z}_0 = \mathbf{M}^{-1}\mathbf{r}_0$ ,  $\mathbf{p}_0 = \mathbf{z}_0$ ,  $\mathbf{e}_0 = \mathbf{r}_0^T \mathbf{z}_0$ 
2:    $k = 0$ 
3:   while not converged do
4:      $\mathbf{v}_k = \mathbf{A}\mathbf{p}_k$ 
5:      $\alpha_k = \frac{\mathbf{e}_k}{\mathbf{p}_k^T \mathbf{v}_k}$  | 1st region:r:2
6:      $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$  | 2nd region:r:4/w:2
7:      $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{v}_k$ 
8:     if  $\sqrt{\|\mathbf{r}_{k+1}\|} = \|\mathbf{r}_{k+1}\| < \epsilon$  then
9:       break
10:    end if
11:     $\mathbf{z}_{k+1} = \mathbf{M}^{-1}\mathbf{r}_{k+1}$ 
12:     $\mathbf{e}_{k+1} = \mathbf{r}_{k+1}^T \mathbf{z}_{k+1}$  | 3rd region:r:2
13:     $\beta_k = \frac{\mathbf{e}_{k+1}}{\mathbf{e}_k}$ 
14:     $\mathbf{p}_{k+1} = \mathbf{z}_{k+1} + \beta_k \mathbf{p}_k$  | 4th region: r:2/w:1
15:     $k = k + 1$ 
16:  end while

```

examined in detail, for example, in Kronbichler and Wall (2018).

3. Conjugate gradient algorithm

The high throughput of the matrix-free operator evaluation has important implications for performance tuning of the CG iterative method as the matrix-vector product might no longer be the dominant operation. Despite using an accurate integration with $p + 2$ points per direction, the throughput shown in Figure 2 is around a third of that of simply copying one vector to the other, which achieves a throughput of 8.5 billion DoFs/s at 205 GB/s due to 24 bytes of access per unknown with 8 bytes read, 8 bytes write, 8 bytes of read-for-ownership transfer (Hager and Wellein, 2011) on a dual-socket Intel Xeon 8174 machine.

For preconditioning, this work considers the case of a simple point Jacobi preconditioner, that is, the matrix diagonal. This preconditioner is representative for problems including a strong mass matrix contribution besides the

Laplacian (1), as argued in Fischer et al. (2020). Since the same coefficient is used for all 3 vector components of \mathbf{u} , only the diagonal to a scalar Laplacian (computed with Gauss–Lobatto integration on $p + 1$ points) is stored and applied to all three components.

3.1. Breakdown of runtime

Figure 1 shows a breakdown of the runtime per unknown for one CG iteration, plotted over the number of unknowns for the basic CG variant presented in Algorithm 1. In this study, we consider the termination by the unpreconditioned residual norm $\|\mathbf{r}_k\|$, which involves a third global reduction in each iteration. Other variants exist, and the main performance characteristics carry over similarly. The following kernels are considered:

- The sparse matrix-vector product with matrix-free operator evaluation,
- AXPY-like vector operations ($\mathbf{y} = \mathbf{ax} + \mathbf{y}$),
- Dot product computations (including l_2 norm), and
- The application of the diagonal preconditioner.

The AXPY-like vector operations and preconditioner application do not involve any communication, the matrix-vector product communicates between nearest neighbors in the mesh (e.g., 26 on a cube geometry with perfect split), whereas the dot product involves a reduction among all participating processes. The experiment of Figure 1 has been conducted on a single compute node with 48 cores.

In the left part of the plot in Figure 1 with fewer than 10^5 DoFs, the load imbalance of the partitioning of the mesh elements onto 48 processes as well as the latency of the communication between the different cores on the node lead to an approximately constant runtime of 6×10^{-5} seconds per iteration. This appears as a decrease of time per unknown as the size increases in the figure. The latency limitations disappear for $n \sim 10^6$ DoFs, indicating a throughput limitation instead with a plateau in timings per unknown. For very large sizes $n > 10^7$, the data set of the conjugate gradient exceeds the caches and most data needs to be fetched from main memory (RAM). Then, the vector operations start to contribute significantly to the runtime, causing a severe slowdown compared to intermediate sizes.

In order to understand the performance limitations of the CG algorithm, we take a closer look at Algorithm 1. Treating the matrix-vector product and the preconditioner as black boxes, there are four separate regions of vector access in the form of dot products and AXPY-like vector operations. Within each region, loop fusion leading to a single loop over the entries of all vectors in the region may improve the locality of reference. Loop fusion can for example be used to compute the sum needed for the norm $\|\mathbf{r}_{k+1}\|$

already during the computation of \mathbf{r}_{k+1} , avoiding an extra vector load.

Between the regions, however, synchronization points prevent loop fusion and all vector entries need to be touched before starting the next region. For instance, the computation of \mathbf{x}_{k+1} and \mathbf{r}_{k+1} depends on \mathbf{p}_k , \mathbf{r}_k , \mathbf{v}_k , \mathbf{x}_k , and α_k . The latter itself depends on \mathbf{p}_k and \mathbf{v}_k and requires a full vector sweep through them. If the size of the vectors \mathbf{p}_k and \mathbf{v}_k exceeds the capacity of a particular cache level during the computation of the dot product for α_k and the entries are already evicted from the cache in the form of capacity misses, a second load from the upper levels of the memory hierarchy is inevitable. Similarly, during the computation of \mathbf{p}_{k+1} in the fourth region, the vector entries of \mathbf{p}_k and \mathbf{z}_{k+1} would have to be loaded again, despite being touched in the second region and inside the preconditioner, respectively. Note that even with an ideal cache replacement strategy this problem cannot be resolved for vectors considerably larger than the caches.

Summarizing the number of reads in each region of the conjugate gradient algorithm, the preconditioned conjugate

Algorithm 2 Pipelined conjugate gradient method.

```

1:   while not converged do
2:      $\mathbf{q}_k = \mathbf{Aw}_k$ 
3:      $\beta_k = \gamma_{k-1} / \gamma_{k-2}$ 
4:      $\alpha_k = \gamma_{k-1} / (\mathbf{a}_{k-1} - \beta_k \gamma_{k-1})$ 
5:      $\mathbf{p}_k = \mathbf{r}_k + \beta_k \mathbf{p}_{k-1}$ 
6:      $\mathbf{x}_k = \mathbf{x}_k + \alpha_k \mathbf{p}_k$ 
7:      $\mathbf{s}_k = \mathbf{w}_k + \beta_k \mathbf{s}_{k-1}$ 
8:      $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_k \mathbf{s}_k$ 
9:      $\mathbf{z}_k = \mathbf{q}_k + \beta_k \mathbf{z}_{k-1}$ 
10:     $\mathbf{w}_k = \mathbf{w}_{k-1} - \alpha_k \mathbf{z}_k$ 
11:     $\gamma_k = \mathbf{r}_k^T \mathbf{r}_k$ 
12:     $\mathbf{a}_k = \mathbf{w}_k^T \mathbf{r}_k$ 
13:   end while

```

r:7/w:6

Algorithm 3 s-step conjugate gradient method with the aliases $\mathbf{R}_k = \mathbf{T}_k(:, 1:s-1)$ and $\mathbf{Q}_k = \mathbf{T}_k(:, 2:s)$.

```

1:   while not converged do
2:      $\mathbf{T}_k = [\mathbf{r}_k, \mathbf{Ar}_k, \dots, \mathbf{A}^s \mathbf{r}_k]$ 
3:      $\mathbf{B}_k = -\mathbf{W}_{k-1}^{-1} (\mathbf{Q}_k^T \mathbf{p}_{k-1})$ 
4:      $\mathbf{P}_k = \mathbf{R}_k + \mathbf{P}_{k-1} \mathbf{B}_k$ 
5:      $\mathbf{W}_k = \mathbf{Q}_k^T \mathbf{P}_k$ 
6:      $\mathbf{g}_k = \mathbf{P}_k^T \mathbf{r}_k$ 
7:      $\mathbf{a}_k = \mathbf{W}_k^{-1} \mathbf{g}_k$ 
8:      $\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{P}_k \mathbf{a}_k$ 
9:      $\mathbf{r}_k = \mathbf{b} - \mathbf{A}_k \mathbf{x}_k$ 
10:     $\gamma_k = \mathbf{r}_k^T \mathbf{r}_k$ 
11:   end while

```

r:2s/w:0
r:2s+1/w:1
r:s+1/w:1
r:2/w:1

gradient algorithm requires 10 full vector reads in each iteration besides the access for the matrix-vector product and preconditioner, despite only 4 vectors participating in the algorithm (assuming \mathbf{v}_k and \mathbf{z}_{k+1} use the same memory). This number can be slightly reduced to 9 by moving the computation of \mathbf{x}_{k+1} to the 4th region to reuse reads of \mathbf{p}_k .

3.2. Alternative CG methods

For a simpler comparison, we now consider plain conjugate gradient algorithms without preconditioner. The basic version (Algorithm 1 with $\mathbf{z}_{k+1} = \mathbf{r}_{k+1}$ and $\mathbf{M}^{-1} = \mathbf{I}$) requires 9 full vector reads and 3 vector writes besides the access for the matrix-vector product.

In the literature, a series of alternative flavors of the CG algorithm have been developed with the goal to reduce the number of synchronization points, primarily driven by *latency* considerations. However, they naturally also increase the possibility for loop fusion and might therefore also improve the memory transfer (Rupp et al., 2016). As a point of comparison of the algorithm structure, we present Algorithm 2 for the pipelined conjugate gradient method and Algorithm 3 for the s -step conjugate gradient methods, respectively. To simplify the presentation, hereafter we ignore the algorithms' initialization and focus on the structure of the main iteration.

In the pipelined CG method (Ghysels and Vanroose, 2014), the number of synchronization points is reduced to one by introducing additional global auxiliary vectors. Apart from the intended ability to overlap global communication with the matrix-vector product, this also allows vector operations to be concentrated in one vector access region. A naive implementation using a separate loop for each line of Algorithm 2 would yield a total of 15 vector reads per CG iteration for the 7 participating vectors. Using loop fusion reduces the number of reads to 7, the number of involved vectors. It is possible to slightly reduce the memory transfer further by performing the update of \mathbf{x} every other iteration (before and after the update of \mathbf{p}).

In contrast, s -step CG methods (Chronopoulos and Gear, 1989; Naumov, 2016) perform s CG iterations in a single phase, reducing the number of global reductions to 3 per phase, that is, to $3/s$ per CG iteration. This is especially interesting when the global reductions are the bottleneck of the CG algorithm. The global reductions are aggregated by not working simply on vectors but on blocks of s vectors, for example, \mathbf{P}_k instead of \mathbf{p}_k and \mathbf{R}_k instead of \mathbf{r}_k . Similarly, the scalar factor α_k becomes a vector ($\mathbf{a}_k \in \mathbb{R}^s$), β_k a matrix ($\mathbf{B}_k \in \mathbb{R}^{s \times s}$), and dot products become block dot products. The communication time of a block operation is similar to that of a scalar one, since modern networks are latency-bound for global reductions up to a few dozens of values.

In the literature, the operation $[\mathbf{A}\mathbf{r}_k, \dots, \mathbf{A}^s\mathbf{r}_k]$ is referred to as a "matrix-power kernel." It is typically considered to be uncritical for performance, since it only comprises of s

point-to-point communication steps in the worst case. For low-order methods, increasing the number of ghost layers allows to use a single communication step per matrix-power kernel application (Malas et al., 2017), which might be useful if the latency is the limiting factor. Furthermore, it can also enable a higher throughput of the matrix-vector product, since matrix and vector entries can be held in caches. For the high-order (FEM) methods investigated here, however, it does not pay off according to preliminary investigations: The wide stencils lead to a much larger dependency region and quickly saturate caches. Already in the absence of communication, matrix-power kernel applications consisting of 3 matrix-vector products with the present high-order FEM for $p = 5$ yield a lower throughput than performing three operator evaluations in sequence. Currently, we are not aware of more sophisticated implementations for this class of algorithms that could exploit this temporal locality. Furthermore, communication is negatively affected as additional ghost layers involve all unknowns on cells with a high surface-to-volume ratio (MehriDehnavi et al., 2013). As shown in Kronbichler and Kormann (2019), the cost of communicating all solution coefficients from a single layer of elements is already substantial and leads to pronounced slowdown of the matrix-vector product for $p > 3$ in 3D.

In total, $s + 1$ matrix-vector multiplications are performed per iteration and four update regions can be identified with a total of $5s + 4$ reads and $s + 2$ writes per vector entry. Finally, we would like to point out that the version of the s -step CG method investigated in the following is numerically unstable due to the loss of orthogonality of the monomial Krylov subspace (Naumov, 2016). However, as alternative formulations, which are numerically more stable but involve additional steps, are structured similarly, results obtained for this simple version are generally transferable to other approaches.

Similarly to the s -step CG methods, enlarged CG methods (ECG; Grigori and Tissot (2019); Lockhart et al. (2022)) also work on blocks of vectors to accelerate convergence. The motivation for the construction and the way to construct the blocks are somewhat different, but the resulting high-level algorithms are similar from the performance point of view to those of s -step CG. Due to this similarity, we will not consider ECG in the remainder of this work.

4. Minimize data access in standard CG

Inspired by the increased chances to fuse loops over vectors in the pipelined and s -step conjugate gradient methods, we now study a version of CG that has been introduced by (Chronopoulos and Gear, 1989, Algorithm 2.2) and served as a starting point for the derivation of pipelining methods. However, in the present work, we do not further modify the algorithm by Chronopoulos and Gear (1989) and instead

aim to reduce the main memory transfer without introducing additional auxiliary vectors, that inherently increase the memory access.

We start our derivation by noting that the number of synchronization barriers identified in Algorithm 1 can be reduced by using redundant computations of partial sums, which is possible in the case the preconditioner is cheap to apply.

4.1. No preconditioner

We first consider the case of identity preconditioning ($\mathbf{z}_k = \mathbf{r}_k$ and $\mathbf{M}^{-1} = \mathbf{I}$) and aim to perform the computation of contributions to β_k before finalizing the computation of α_k and \mathbf{r}_{k+1} . We therefore expand $\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}$ into

$$\begin{aligned} \mathbf{r}_{k+1}^T \mathbf{r}_{k+1} &= (\mathbf{r}_k - \alpha_k \mathbf{v}_k)^T (\mathbf{r}_k - \alpha_k \mathbf{v}_k) \\ &= \mathbf{r}_k^T \mathbf{r}_k - 2\alpha_k \mathbf{r}_k^T \mathbf{v}_k + \alpha_k^2 \mathbf{v}_k^T \mathbf{v}_k \end{aligned} \quad (6)$$

By computing the three sums for the inner products $\mathbf{r}_k^T \mathbf{r}_k$, $\mathbf{r}_k^T \mathbf{v}_k$, $\mathbf{v}_k^T \mathbf{v}_k$, the ingredients for β_k can be scheduled in parallel to the inner product $\mathbf{p}_k^T \mathbf{v}_k$ needed by α_k , as shown in Algorithm 4. Note that $\gamma_k = \mathbf{r}_k^T \mathbf{r}_k$ is computed explicitly rather than defined recursively from the previous iteration in order to avoid detrimental influence of roundoff errors (Chronopoulos and Gear, 1989; Saad, 1985). While this scheme adds an additional read to \mathbf{r}_k during the summation compared to the computation of $\mathbf{v}_k^T \mathbf{p}_k$ alone, this is compensated by computing \mathbf{r}_{k+1} at the same time as using the respective entry for \mathbf{p}_{k+1} . In addition, the fused scheduling uses \mathbf{p}_k for both \mathbf{x}_{k+1} and \mathbf{p}_{k+1} . In the end, the number of vector access regions is reduced to 2, one before (“pre”) and one after (“post”) the matrix-vector product.

It is also possible to perform the updates to \mathbf{x}_{k+1} only every other iteration, reusing the content of the vector \mathbf{p}_{k-1} and \mathbf{r}_{k-1} before they get updated. All together, the number of vector reads is reduced from 9 in the basic CG iteration to 6.5 in this improved variant.

Algorithm 4 Conjugate gradient method with merged vector operations.

```

1:    $k = 0, \alpha_0 = \beta_0 = 0, \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0, \mathbf{p}_0 = \mathbf{v}_0 = 0$ 
2:   while not converged do
3:      $k = k + 1$ 
4:     if  $k > 1$  odd then | “pre” region:r:3.5/w:2.5
5:        $\mathbf{x}_k = \mathbf{x}_{k-2} + \alpha_{k-1} \mathbf{p}_{k-1}$ 
6:        $+ \frac{\alpha_{k-1}}{\beta_{k-2}} (\mathbf{p}_{k-1} - \mathbf{r}_{k-1})$ 
7:     end if
8:      $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_{k-1} \mathbf{v}_{k-1}$ 
9:      $\mathbf{p}_k = \mathbf{r}_k + \beta_{k-1} \mathbf{p}_{k-1}$ 
10:     $\mathbf{v}_k = \mathbf{A}\mathbf{p}_k$ 

```

(continued)

(continued)

```

11:    $\alpha_k = \mathbf{p}_k^T \mathbf{v}_k$  | “post” region:r:3/w:0
12:    $\gamma_k = \mathbf{r}_k^T \mathbf{r}_k$ 
13:    $c_k = \mathbf{r}_k^T \mathbf{v}_k$ 
14:    $d_k = \mathbf{v}_k^T \mathbf{v}_k$ 
15:    $\alpha_k = \frac{c_k}{a_k}$ 
16:    $\gamma_{k+1} = \gamma_k - 2\alpha_k c_k + \alpha_k^2 d_k$ 
17:   if  $\sqrt{\gamma_{k+1}} < \varepsilon$  then
18:     if  $k$  odd then
19:        $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$ 
20:     else
21:        $\mathbf{x}_{k+1} = \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_k + \frac{\alpha_{k-1}}{\beta_{k-1}} (\mathbf{p}_k - \mathbf{r}_k)$ 
22:     end if
23:     break
24:   end if
25:    $\beta_k = \frac{\gamma_{k+1}}{\gamma_k}$ 
26:   end while

```

Rupp et al. (2016) identified possibilities for additional performance optimizations by the three phases “pre,” “matrix-vector product,” and “post.” Specifically, that contribution proposed to merge the matrix-vector product with the “post” region on graphics processing units (GPUs) for matrix-vector products through sparse matrix representations in order to reduce the number of kernel calls. Building upon this idea, we aim to merge *both* regions with the matrix-vector product, which on the one hand allows to reduce the memory transfer on the CPU, but is also more involved in the context of matrix-free FEM.

4.2. Diagonal preconditioner

The ideas of the previous subsection can be extended to the case of a preconditioner. Under the assumption that the preconditioner is cheap and that there are no long-range dependencies introduced to the computation of $\mathbf{z}_{k+1} = \mathbf{M}^{-1} \mathbf{r}_{k+1}$, it is more economic to apply the preconditioner several times.

Following equation (6), we decompose the computation of the numerator for β_k into several inner products that do not depend on α_k

$$\begin{aligned} \beta_k &= \frac{\mathbf{z}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{z}_k^T \mathbf{r}_k} = \frac{(\mathbf{M}^{-1} \mathbf{r}_{k+1})^T \mathbf{r}_{k+1}}{\mathbf{z}_k^T \mathbf{r}_k} \\ &= \frac{\mathbf{r}_k^T \mathbf{M}^{-1} \mathbf{r}_k - 2\alpha_k \mathbf{r}_k^T \mathbf{M}^{-1} \mathbf{v}_k + \alpha_k^2 \mathbf{v}_k^T \mathbf{M}^{-1} \mathbf{v}_k}{\mathbf{r}_k^T \mathbf{M}^{-1} \mathbf{r}_k} \end{aligned} \quad (7)$$

Thus, β_k can be obtained only based on the value of \mathbf{r}_k and \mathbf{v}_k from the beginning of the iteration, prior to the update of the vectors \mathbf{x}_{k+1} , \mathbf{r}_{k+1} , \mathbf{z}_{k+1} .

Similarly, the value of $\gamma_{k+1} = \|\mathbf{r}_{k+1}\|^2$ for the convergence criterion can be computed in parallel to the reduction for α_k , using the expansion

$$\gamma_{k+1} = \mathbf{r}_k^T \mathbf{r}_k - 2\alpha_k \mathbf{r}_k^T \mathbf{v}_k + \alpha_k^2 \mathbf{v}_k^T \mathbf{v}_k \quad (8)$$

Therefore, given the residual \mathbf{r}_k and the result of the matrix-vector product \mathbf{v}_k , all scalars of the current conjugate gradient iteration can be computed using one reduction region. The vector \mathbf{z}_{k+1} is no longer stored explicitly, since we assume that the application of the preconditioner is cheaper than the read and write of \mathbf{z}_{k+1} . As a result of this restructuring, all vector updates can be clustered in a single region. Simplifying the notation and combining the expressions above, we obtain Algorithm 5.

The reformulated algorithm results in two vector access regions, with loop fusion applicable within each region. As in Algorithm 4, the \mathbf{x}_k update can be delayed and performed only every other iteration. The first fused loop region now consists of 3.5 full vector loads per iteration, plus the load of the preconditioner that—under the assumption that a single diagonal is used for each component of the block PDE system (1)—consists of $1/3$ doubles per vector entry. The number of stores is one for \mathbf{r}_k and one for \mathbf{p}_k as well as one for \mathbf{x}_k every second iteration. The number of vector loads in the second

region is equal to 3 plus $1/3$ for the diagonal preconditioner. The present reformulation results in seven global reductions, which can be computed by summations local to each MPI process and a single MPI_Allreduce carrying 7 variables. Once the seven scalars are available, the coefficients α_k and β_k can be computed locally. The presented reformulation also enables a fusion into the matrix-vector product, as discussed in the next section.

The proposed algorithm relies on three properties of the preconditioner \mathbf{M}^{-1} :

- The preconditioner, which is applied twice in the first vector access region and twice in the second vector access region, is assumed to be cheap to apply, with arithmetic costs hidden behind the memory transfer of the involved vectors.
- We assume that there are no long-range dependencies in the preconditioner, allowing to reuse the respective entries of \mathbf{r}_k and \mathbf{v}_k from caches or registers when $\mathbf{M}^{-1}\mathbf{r}_k$ and $\mathbf{M}^{-1}\mathbf{v}_k$ are computed.
- The memory access induced by the preconditioner is assumed to be less expensive than the aggregated store and load of \mathbf{z}_{k+1} in Algorithm 1.

Algorithm 5 Preconditioned conjugate gradient method with merged vector operations.

```

1:    $k = 0, \alpha_0 = \beta_0 = 0, \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0, \mathbf{p}_0 = \mathbf{v}_0 = 0$ 
2:   while not converged do
3:      $k = k + 1$ 
4:     if  $k > 1$  odd then
5:        $\mathbf{x}_k = \mathbf{x}_{k-2} + \alpha_{k-1}\mathbf{p}_{k-1}$ 
6:        $+ \frac{\alpha_{k-2}}{\beta_{k-2}}(\mathbf{p}_{k-1} - \mathbf{M}^{-1}\mathbf{r}_{k-1})$ 
7:     end if
8:      $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_{k-1}\mathbf{v}_{k-1}$ 
9:      $\mathbf{p}_k = \mathbf{M}^{-1}\mathbf{r}_k + \beta_{k-1}\mathbf{p}_{k-1}$ 
10:     $\mathbf{v}_k = \mathbf{A}\mathbf{p}_k$ 
11:     $\gamma_k = \mathbf{r}_k^T \mathbf{r}_k$ 
12:     $a_k = \mathbf{p}_k^T \mathbf{v}_k$ 
13:     $b_k = \mathbf{r}_k^T \mathbf{v}_k$ 
14:     $c_k = \mathbf{v}_k^T \mathbf{v}_k$ 
15:     $d_k = \mathbf{r}_k^T \mathbf{M}^{-1}\mathbf{r}_k$ 
16:     $e_k = \mathbf{r}_k^T \mathbf{M}^{-1}\mathbf{v}_k$ 
17:     $f_k = \mathbf{v}_k^T \mathbf{M}^{-1}\mathbf{v}_k$ 
18:     $\alpha_k = \frac{d_k}{a_k}$ 
19:    if  $\sqrt{\gamma_k - 2\alpha_k b_k + \alpha_k^2 c_k} < \varepsilon$  then
20:      if  $k$  odd then
21:         $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$ 
22:      else
23:         $\mathbf{x}_{k+1} = \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_k + \frac{\alpha_{k-1}}{\beta_{k-1}}(\mathbf{p}_k - \mathbf{M}^{-1}\mathbf{r}_k)$ 
24:      end if
25:      break
26:    end if
27:     $\beta_k = \frac{d_k - 2\alpha_k e_k + \alpha_k^2 f_k}{d_k}$ 
28:  end while

```

A diagonal preconditioner obviously fulfills these properties, whereas, on the other extreme, a multigrid V-cycle would violate all three requirements. Clearly, it needs to be examined for each preconditioner whether it fits into this scheme on a case-by-case basis, with preconditioners with more global action requiring a separate storage step to get $\mathbf{M}^{-1}\mathbf{r}_{k+1}$ before the reductions for β_k .

5. Combining vector updates with matrix-vector product

In the previous section, the matrix-vector product has been treated as a black box. In order to further improve the data reuse between the vector access regions of Algorithms 4 and 5, we propose to embed the vector updates and dot products into the matrix-free operator evaluation, which allows to re-use hits of the entries of \mathbf{p} , \mathbf{r} , \mathbf{v} in caches during the “post” stages, leading to a single memory read of the vectors \mathbf{p} , \mathbf{r} , \mathbf{v} , and \mathbf{x} in the ideal case (3.83 doubles per unknown).

This is realized by performing the operations identified in the previous section on subranges of the vectors while looping over cells according to equation (4) to exploit temporal locality. In order to produce a valid algorithm, the data dependencies during the matrix-vector product need to be identified and translated into subranges, as detailed in the following three subsections. Note that this approach is more involved than previously proposed algorithms that fuse vector operations following the matrix-vector product into

the loop over unknowns in sparse matrix-vector products (Rupp et al., 2016) or over cells for discontinuous Galerkin schemes, for example, in Kronbichler and Allalen (2018), Charrier et al. (2019) and Munch et al. (2021).

5.1. Data dependencies in matrix-free loops

On a high level, the matrix-vector multiplication depends on the source vector \mathbf{u} and produces the destination vector \mathbf{v} . The cell-wise nature of our matrix-free algorithm, involves a loop through each cell of the mesh that

- reads all unknowns $\mathbf{u}_e = \mathbf{P}_e \mathbf{u}$ attached to a cell, which can be shared with other cells for continuous finite elements, and
- accumulates integral contributions in the same way ($\mathbf{P}_e^T \mathbf{v}_e$).

We can hence refine the dependency statement for each entry of the source and the destination vector: the entry $u_i, i \in \{1, \dots, n\}$, is only needed once the first cell reads its value. Conversely, entry v_i is available as soon as the last cell has added its contribution. From an implementation point of

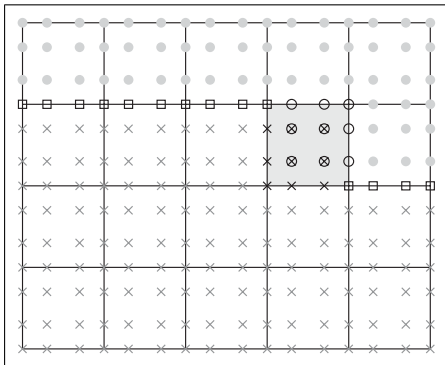


Figure 3. Illustration of data dependencies in matrix-free operator evaluation with degree $p = 3$ for a lexicographic loop through the cells starting from the bottom left. Each symbol represents an unknown. Gray crosses denote unknowns where the result of the operator evaluation is complete before the highlighted cell. The 3×3 unknowns marked with black crosses get the final contribution from the highlighted cell and can schedule the “post” operation afterward together with the unknowns marked with gray crosses. The 3×3 unknowns indicated by black circles indicate unknowns that have their first access on the highlighted cell, thus necessitating to be preceded by the “pre” operation. Black squares denote unknowns with pending integrals, that is, the “pre” operation has already been done, but the “post” operation is not yet possible. Gray disks illustrate unknowns not yet processed.

view, this means that we can postpone the update of u_i until its first usage and use the value of v_i for the dot product as soon as its value has been finalized. In the following, we are going to refer to operations happening before the first read access to \mathbf{u} but still within the matrix-free loop—in line with the region names in Algorithms 4 and 5—as a “pre” operation and to operations happening after the last write access to \mathbf{v} as a “post” operation.

Figure 3 visualizes the data dependencies in a matrix-free operator evaluation as well as its interplay with “pre” and “post” operations. In an MPI-parallel context, the ghost exchange adds additional constraints (Kronbichler and Kormann, 2012, Algorithm 2.1). More precisely, all unknowns owned by a process in the vector \mathbf{u} that need to be sent to remote processes have to perform the “pre” operation before the ghost exchange is initiated. Furthermore, the part of integrals accumulated on remote processes needs to be first sent to owner of the respective entry in the vector \mathbf{v} before the “post” operation can be scheduled on those unknowns. It should, however, be noted that both communication steps can be overlapped with computations on inner cells.

We conclude this subsection by discussing the major differences to matrix-based implementations. The popular compressed row storage and, similarly, other sparse-matrix formats update an entry in the destination vector only once by applying the whole row of the matrix. In such a context, it is obvious when values in the destination vector are available and it is straightforward to determine when to schedule the “post” operation during the matrix-vector product in a merged way, as was exploited by Rupp et al. (2016). However, this relation is not given for the dependency region of the source vector, allowing to embed the “pre” operation closer to the user of vector entries only based on a dependency analysis similar to the one proposed here.

5.2. Batching work from several cells

Tracking the state of each individual vector entry u_i, v_i for scheduling the “pre” and “post” operations would lead to excessive overhead and inhibit loop optimizations, such as vectorization and unrolling of the vector operations in CG. Therefore, the “pre” and “post” operations are tracked on ranges of vector entries. The length of the range is given in multiples of 64, a heuristic value that permits full vectorization with typical SIMD lengths today, except for a single spot at the end of the vectors.

The length of the ranges is crucially influenced by the number of vector entries processed by the matrix-free integrals in between. The intent is to reach a significant share of overlap between the “pre” and “post” ranges, enabling to reuse data read during the “pre” operations even during the “post” operations from the fast cache memory. The range lookup and the callback into matrix-free integration functions come with some overhead in our implementation, which is

especially noticeable for low and intermediate polynomial degrees with small work per cell. We therefore schedule the “pre” and “post” operations not around every individual cell, but around *batches of cells*. The size of the batches is selected as

$$n_{\text{batch}} = \max\left(\left\lfloor \frac{1024}{3(p+1)^3} \right\rfloor, 2\right) n_{\text{simd lanes}} \quad (9)$$

The first expression inside the maximum operation ensures that more cells are grouped together for lower polynomial degrees, by dividing by the number of unknowns on each cell. The resulting number of cells is multiplied by the number of SIMD lanes in the instruction set in order to employ vectorization across elements (Kronbichler and Kormann, 2012). For higher degrees ($p \geq 4$), at least two SIMD groups of cells are used.

Depending on the number of SIMD lanes, the number of vectors accessed in the “pre” and “post” stages, as well as the additional data access for the matrix-free integrals, the criterion given by equation (9) leads to a few thousands to tens of thousands of double-precision values corresponding to up to few hundreds of kB of data. This fits well within modern level-2 or level-3 caches, which is why no additional tuning has been performed.

We have integrated the proposed algorithm into deal.II (Arndt et al., 2020a). It allows users to perform a “pre” and “post” operation during any matrix-free loop by providing—additionally to the cell operation—appropriate anonymous functions in the style of

```
vmult(dst, src) := loop(dst, src, op cell, op pre, op post)
```

Since the operation *op_cell*, which contains the specifics of the considered PDE/physics, is interchangeable, our approach is modular and the proposed algorithms are easily applicable to other PDEs—not just those considered in this publication. For CG, we provide off-the-shelf implementations of *op_pre* and *op_post*.

5.3. Numbering of unknowns

The second ingredient is to minimize the number of ranges with a high number of cell batches between the first and last access in the matrix-free loop. As seen from Figure 3, unknowns located on shared vertices, edges, and faces all have the potential to reach over long distances. This effect is exacerbated when working on blocks of 64 unknowns, because a single entry out of 64 can lead to a delay of the “post” operation. It is therefore crucial to develop a suitable cell traversal and numbering of unknowns. The cell traversal should aim for a high volume-to-surface ratio of the cell batches, because all unknowns located inside the cell batch have an optimal pre-post distance. In this work, a Morton space-filling curve is used for the partitioning of

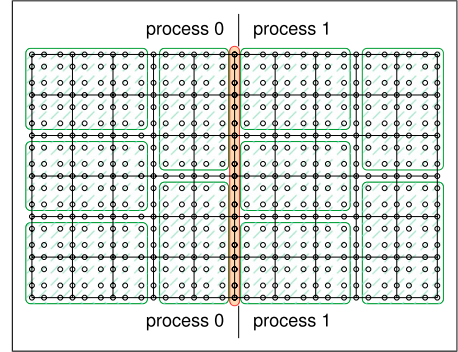


Figure 4. Illustration of the numbering of degrees of freedom for a 2D setup with polynomial degree $p = 3$. Cells are grouped together into batches of 6 cells and the interior unknowns are numbered first (highlighted by green-shaded boxes). The second set of numbers are unknowns located on more than one cell batch (not marked). The third set consists of unknowns that need to be exchanged with remote MPI processes (orange shades).

elements among the processes (Bangerth et al., 2011; Burstedde et al., 2011) and for the process-local mesh traversal.

Given the mesh traversal, unknowns are enumerated in the sequence of the following four steps, see also the illustration in Figure 4:

- In the first step, all unknowns touched only by a single cell batch are enumerated following the ordering of the cells. Except for reaching the next multiple of 64, this group will have a minimal distance of one between the “pre” and “post” phase.
- Next, the enumeration is continued on the unknowns touched by several batches, but not in contact with remote MPI processes. Here, some ranges will have a high distance, whereas others can still be completed reasonably close after the start.
- In the third step, the unknowns owned locally, but requested by remote MPI ranks, are assigned. These unknowns will not profit from overlap between the “pre” and “post” steps, because the “pre” step needs to be done before the initial `MPI_Send` command, whereas the “post” step comes after the final `MPI_Recv` command. A contiguous numbering reduces the ranges of this unfavorable part of the vector to a minimum, besides also facilitating the pack/unpack operations.
- Unknowns that are subject to constraints (not shown in Figure 4), such as Dirichlet boundary conditions, will not receive contributions from the matrix-free integrals with matrix A representing a homogeneous operator. If they are kept in the linear system, like in the implementation of deal.II, they are appended at

the end of the locally owned unknowns and updated during the last cell batch.

Furthermore, the numbering is set up to ensure contiguous numbers of multiple unknowns associated with each vertex, edge, face, and volume, in order to reduce the memory for index storage from $3(p+1)^3$ numbers per cell to 3^3 numbers (for consistently oriented meshes). This reduces the memory requirements of metadata, increases data locality and effectiveness of prefetching as well as allows for packed load operations.

Figure 5 visualizes the benefits of the proposed enumeration algorithm by plotting the cumulative distribution function of the liveliness of each subrange. We define “liveliness” as the number of cell batches processed between the first and the last access, respectively. As a reference, we also show the liveliness of the standard enumeration of degrees of freedom in deal.II (enumeration in cell order). The reduction of the liveliness is clearly visible. For the vector Laplacian, around 76%—in contrast to 54%—of the subranges is processed even in the same batch of cells. While the possibility to process subranges within the same batch of cells is not necessary, we can see similar trends for subranges living less than 10 batches of cells. This is an important threshold: Each cell batch of 16 cells touches 12 kB of unique data (geometry, indices) for the matrix-vector product and

$$16[\text{cells}] \times 3 \cdot 5^3 \left[\frac{\text{unique DoFs}}{\text{cell}} \right] \times 8 \left[\frac{\text{byte}}{\text{double}} \right] = 48[\text{kB}] \quad (10)$$

of unique data per vector or 208 kB for four vectors and the preconditioner. For around 10 cell batches, the data thus reaches the combined size of the L2 and L3 caches per core

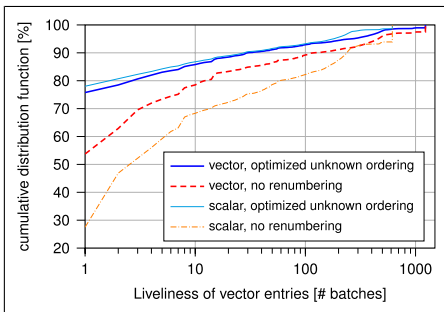


Figure 5. Liveliness of data in vector ranges for the 3D vector and scalar Laplacians with polynomial degree $p = 5$ on 40 MPI processes. The vector Laplacian involves 297 million DoFs subdivided into 1229 cell batches on each MPI process, the scalar Laplacian 99 million DoFs with 615 cell batches.

on the Intel Skylake architecture. For the scalar Laplacian, our heuristics use batches of 32 cells instead due to a lower number of DoFs/cell. This gives slightly better liveliness for our proposed numbering scheme, whereas the case with deal.II’s default numbering scheme has even higher liveliness than in the vector case, as the DoFs with long liveliness are spread to many blocks of 64 DoFs when the number of unknowns per cell is lower.

While the consideration of liveliness is a rather theoretical approach of quantifying the benefits for combining the “pre” operation, the matrix-vector product, and the “post” operation, a clear reduction in the data volume accessed from RAM can be observed. A cache analysis (see Figure 6) conducted on the basis of hardware performance counters using the LIKWID tool (Treibig et al., 2010) reveals that the combined version of the PCG algorithm, as proposed here, only reads 5.7 doubles per degree of freedom once the capacity of the caches is exceeded. This value is lower by 4.6 doubles than the value of 10.3 reads for the naive execution of Algorithm 5. Note that the renumbering proposed above has further benefits beyond the liveliness shown in Figure 5, as the proposed scheme leads to a more linear data access pattern and fewer active streams, improving the effectiveness of hardware prefetching and reducing stress on the translation-lookaside buffers.

5.4. Comparison of CG variants

Since the reduction of the data volume to be transferred from/to main memory is the key strength of the CG algorithm proposed in this work, we conclude this section by

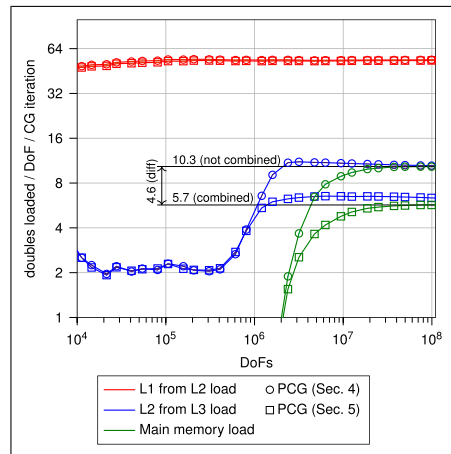


Figure 6. Comparison of measured memory transfer for 2×20 cores of Intel Xeon Gold 6230 for standard and combined versions of Algorithm 5.

comparing the measured read and write data volumes of the basic CG, pipelined CG and s -step CG algorithms (Algorithms 1–3, with loop fusion applied where possible) with the results of the proposed combined Algorithms 4 and 5.

Table 1 shows the predicted memory read and write transfer volumes in different regions of the CG versions. In the proposed combined CG variants, we assume that the reuse of memory reads allows vectors to be read only once across the iteration of the algorithm and, as a consequence, we do not separate estimates of the vector access regions and of the matrix-vector product.

Figure 7 presents both the estimated and measured averaged values of a complete iteration, derived from experiments with 100 iterations. The cost of a single matrix-vector product (“mat-vec”) is 3.0 double data reads and 1.3 double data writes, which is slightly higher than the theoretical expectation (2.0/1.0) due to non-perfect caching and loading of the geometry data. Since the optimization of the memory transfer of this portion of the algorithm is not the focus of the current work, we use the measured values of the matrix-vector multiplication as a baseline transfer also for the CG algorithms.

In Figure 7, one can see that the measured values match well with the predicted ones. Furthermore, it is clear that while the amount of data to be written by the combined versions is comparable with the s -step version, they read up to 5 doubles less data from RAM compared both to the pipelined and the s -step CG schemes. Given the considerations in the previous subsection, this improvement is expected, since a large fraction of the vector entries accessed during the “pre” operation remains in caches until they are read again during the “post” operation.

Compared to the theoretical transfer of 4.8 doubles per degree of freedom, the excess transfer in the combined preconditioned method can be explained to a good extent by the liveliness in Figure 5 and the data-in-flight suggested by equation (10): 13% of vector entries have a liveliness of 10 or more cell batches, which can be expected to give 2 additional reads between the “pre” operation and the matrix-vector product as well as a transfer of 3.3 doubles to the “post” operation for the respective part of the vector. This explains 0.7 out of the 0.9 excess reads of doubles per DoF.

Furthermore, it is worth noting that the cost of the non-preconditioned and of the Jacobi-preconditioned variant of the proposed CG algorithm is very close, underlining that the benefit is even clearer in the preconditioned case.

In the next section, we evaluate the influence of the reduced access to RAM and the reduced number of global reductions on the throughput of CG algorithms for different scenarios of high-order matrix-free finite-element methods.

Table 1. Summary of the modeled ideal memory transfer of vector access regions (see also the annotations in Algorithms 1–5) and matrix-vector multiplication for different CG variants.

| | vector access | | mat-vec | |
|--------------|---------------|-----------|---------------|-------|
| | Read | Write | Read | Write |
| CG | 9 | 3 | 2 | 1 |
| Pipelined CG | 7 | 6 | 2 | 1 |
| s -step CG | $5 + 4/s$ | $1 + 2/s$ | 2 | 1 |
| Combined CG | – | – | 3.5 | 3.5 |
| PCG | 13 | 4 | 2 | 1 |
| Combined PCG | – | – | 3.8 $\bar{3}$ | 3.5 |

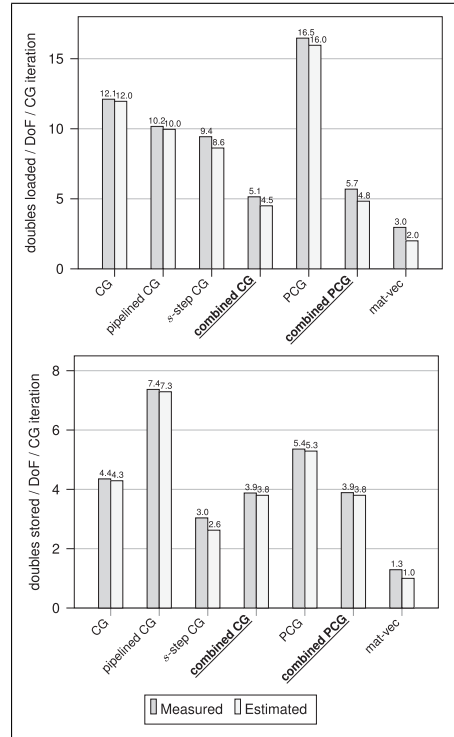


Figure 7. Comparison of measured and estimated memory transfer for various methods on 2×20 cores of Intel Xeon Gold 6230 and 10^8 DoFs, using the measured values of the matrix-vector multiplication as a baseline transfer.

6. Numerical results

In Sections 4–5, we have proposed techniques that reduce the access to main memory during CG iterations. Since reducing the memory access is only a means to the

application goal of increasing the “throughput,” the CG and PCG variants for different numbers of compute nodes are evaluated in the following, including some variations of the benchmark and the behavior for different hardware. Similarly to Section 5.4, basic loop fusion is applied in Algorithms 1–3 during evaluation.

6.1. Node-level performance

Figure 8 shows the throughput on a single compute node for the different CG variants (no preconditioner) as well as the preconditioned (PCG) case with a diagonal preconditioner. For small sizes, the throughput is largely similar between the methods, given that the matrix-vector product is the dominating cost and fast caches can absorb the various vector access patterns. As anticipated by the memory transfer analysis from Section 5.4, the picture changes when going to larger sizes, where the memory-transfer-efficient combined variants are significantly faster. The advantage is particularly impressive considering that the proposed CG and PCG variants, running at 2.36 and 2.13 billion DoFs/s for the largest sizes, are separated from *any* other method by a larger gap than what is observed between the best and worst of the remaining schemes, the s -step CG method ($s = 6$) with 1.46 billion DoFs/s and the preconditioned CG scheme with 0.98 billion DoFs/s.

The mix of memory-intensive operations on vectors and the arithmetically heavy matrix-vector product makes the throughput slightly deviate from the memory-transfer predictions of Figure 7. For example, the throughput of the combined CG scheme of 2.36 billion DoFs/s corresponds to an average memory transfer of around 170 GB/s aggregated over the whole CG solver, whereas the s -step method with 1.46 billion DoFs/s involves an average transfer of 145 GB/s. While neither of the two variants saturates the memory

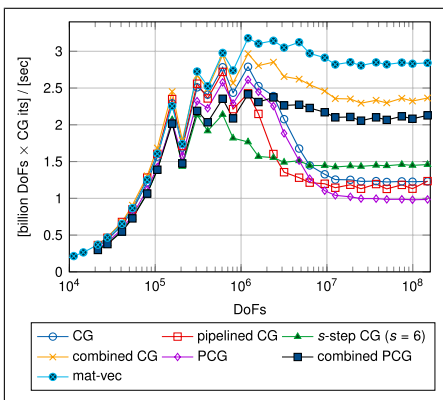


Figure 8. Throughput over the problem size on a single Intel Xeon Platinum 8174 node for the different CG variants.

bandwidth on the present architecture, the achieved bandwidth demonstrates an additional benefit of our CG implementation besides the lower memory transfer: Fusing vector operations into an arithmetic-heavy matrix-vector product allows to use spare memory bandwidth, leading to a better distribution of the memory transfer.

Figure 8 also shows the throughput of the matrix-vector product alone as a point of reference. As its throughput is 20–30% higher than that of the proposed merged variants, without the latter fully saturating the available memory bandwidth, we suppose that further performance improvements could be gained in the proposed algorithms by suitable data prefetching. Note that the slight oscillations in throughput are caused by differences in the amount of data exchange when cells are divisible by 48 or 64 as discussed before.

6.2. Scalability on up to 3072 nodes

Figure 9 shows the throughput for different CG and PCG variants on 512 compute nodes. The plot scales the achieved

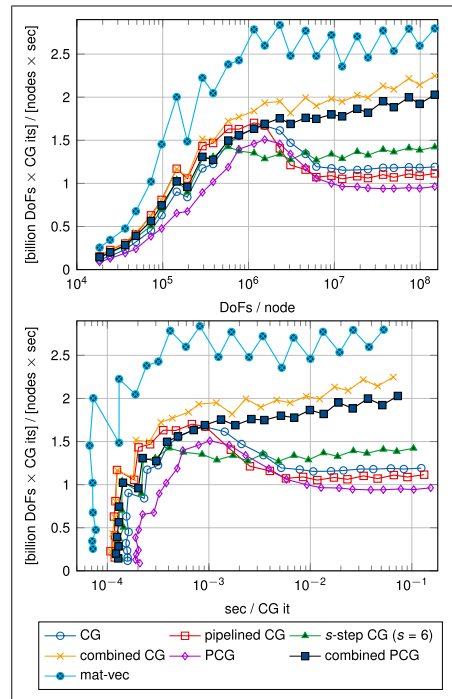


Figure 9. Throughput over the problem size per node (top) and throughput over latency (bottom) on 512 nodes of Intel Xeon Platinum 8174 (24,576 MPI ranks) for various formulations.

throughput by the number of nodes, which allows a direct comparison with Figure 8. It can be seen that the behavior for large sizes is similar to the single-node case, with a slight loss of around 5–10% in the parallel efficiency due to inter-node communication. For intermediate sizes $n \sim 10^6$ per node, however, there is a pronounced difference. In this regime, the timings of a single iteration are in the range of the communication cost in terms of a global reduction, canceling parts of the cache effect.

The experiments show that the proposed combined CG variants achieve a similar performance for small sizes as the pipelined and s -step methods, despite the latency optimization of the latter methods. This can be explained by the number of MPI_Allreduce calls per iteration, which are one for both the combined CG method and the pipelined CG method (albeit overlapped with the matrix-vector product for the latter), whereas the s -step method with $s = 6$ results in an average of 0.5 global reductions per iteration. The scaling limit becomes even clearer when plotting the measured throughput over the time consumed by a single CG iteration in the lower panel of Figure 9, directly showing the lowest possible iteration time. While the CG and PCG methods are slower due to two and three global reductions per iteration, respectively, all other methods take around 1.3×10^{-4} seconds as a minimum time, which is caused by the global reduction combined with a scaling limit of around 8×10^{-5} seconds for the matrix-vector product.

In Figure 10, the throughput on 3072 nodes against the time of a single CG iteration is shown. By comparing with the result on 512 nodes (dotted lines), a slight loss in throughput for larger sizes can be seen, corresponding to a small reduction in parallel efficiency for weak scaling. Near the strong scaling limit in the left part of the figure, an

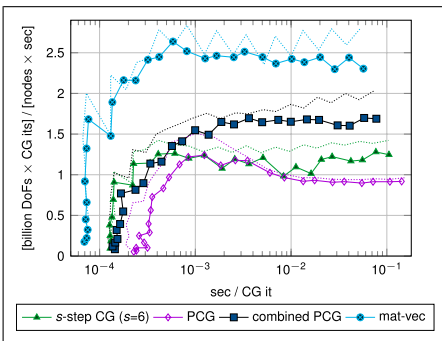


Figure 10. Throughput over the problem size per node (top) and throughput over latency (bottom) on 3072 nodes of Intel Xeon Platinum 8174 (147,456 MPI ranks) for various formulations. The dotted lines show the scaled throughput on 512 nodes (see also Figure 9).

increase in the minimal time can be observed, which is due to the higher cost of the global reductions on a larger scale. However, the increase is similar between the proposed combined PCG algorithm and the baseline methods. More importantly, the combined CG algorithm with preconditioner achieves a throughput that is 35–40% higher than the unpreconditioned s -step method for large sizes, confirming the beneficial behavior of the proposed variant.

6.3. Benchmark variations

In the following, we will consider variants of the benchmark introduced in Section 2. For the sake of simplicity and given the results from the previous subsection, we concentrate on the basic PCG algorithm and the proposed combined PCG algorithm.

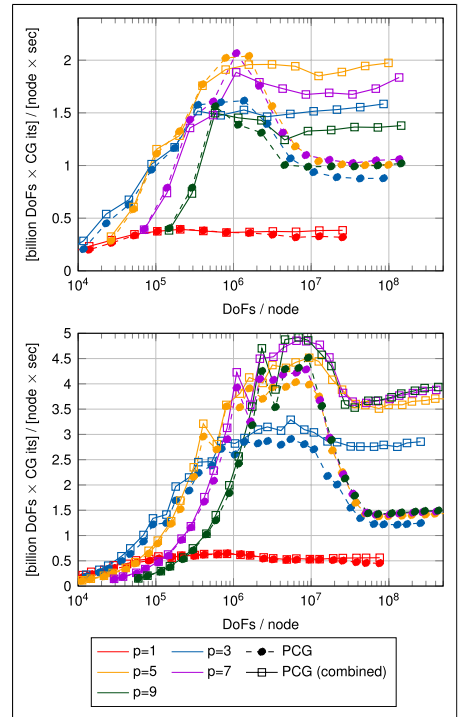


Figure 11. Throughput over the problem size on four nodes of Intel Xeon Platinum 8174 (top) and on one node of 2×64 core AMD Epyc 7742 (bottom) for the BP4 benchmark for different polynomial degrees p , all with quadratic geometry representation and $n_q = (p + 2)^3$ quadrature points.

6.3.1. Variation of the polynomial degree

The results obtained for the polynomial degree $p = 5$ above are transferable also to other polynomial degrees, as shown in Figure 11 (top panel). As examples, $p = 1, 3, 5, 7, 9$ are considered. For $p = 1$, the cost of computations compared to the number of unknowns is overwhelming, as $(p + 2)^3 = 27$ integration points are used per cell compared to one unique unknown per cell, leading to a low throughput of 0.4 GDoFs/sec. This behavior specific to the present matrix-free operator evaluation behaves as $(p + 2)^3/p^3$ and thus gives less work per unknown for higher degrees, as opposed to sparse matrix-vector products (Kolev et al., 2021; Kronbichler and Kormann, 2012). For higher degrees, we can observe significant speedups of the proposed PCG variants compared to the basic PCG, with the highest throughput observed for $p = 5$. Note that the maximal achievable throughput decreases for the combined PCG algorithm as the polynomial degree increases beyond $p \geq 7$, as opposed to constant throughput for the basic PCG scheme. This suggests that the fusion of vector updates within matrix-vector product as proposed in Section 5 loses its benefits due to a limited cache size. Caches need not only hold vector data of increasing size but also larger temporary arrays for sum factorization (Kronbichler and Kormann, 2019), with data of 8 elements in flight on the given AVX-512 hardware. Note that no tuning of the parameters that have been identified in Section 5 has been performed, relying on simple heuristics.

6.3.2. Variation of the geometric description

According to the discussion in Section 2.2, we have concentrated on a tri-quadratic geometry description as a compromise between higher-order geometry representation and high throughput up to now. However, the beneficial behavior observed is transferable to other geometric descriptions as well. Figure 12 compares the proposed algorithm with a basic PCG scheme on the two extrema of matrix-vector products from Figure 2, one loading the inverse Jacobian at each quadrature point and the other using an affine mesh with $n_q = p + 1$ and constant Jacobians. While we observe a speedup of 2.17 in our base case of a bilinear geometry description, the solver is 1.57 \times faster when loading the inverse Jacobians and even 2.27 \times faster in the case of an affine mesh. The relatively low improvement when loading the inverse Jacobians can be understood by recalling the node-level performance analysis of Section 6.1 as the matrix-vector product is itself limited by the memory bandwidth. Therefore, no additional memory transfer can be hidden behind computations, reducing the advantage to the reduction in memory transfer only.

6.3.3. Variation of the partial differential equation

As a next set of tests, we consider variants of the benchmark from Fischer et al. (2020), namely BP1 (scalar mass matrix,

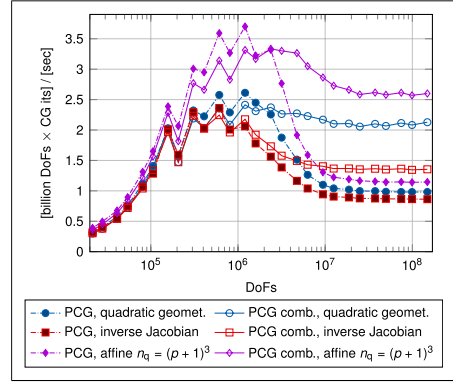


Figure 12. Comparison of throughput of BP4 benchmark with basic preconditioned CG algorithm and the proposed combined variant with different implementations of matrix-free operator evaluation for polynomial degree $p = 5$ on 2×24 cores of Intel Xeon Platinum 8174.

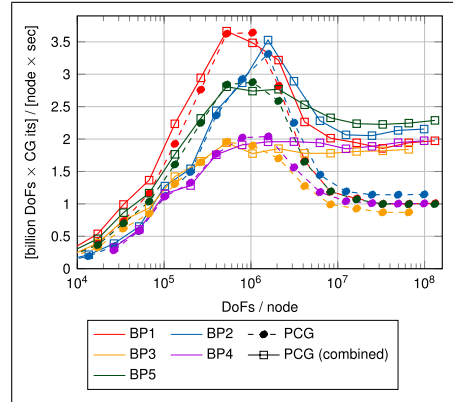


Figure 13. Throughput over the problem size for the standard preconditioned CG scheme and the proposed improved version on #3 nodes of Intel Xeon Platinum 8174 for the CEED benchmark problems BP1 (scalar mass matrix), BP2 (vector-valued mass matrix), BP3 (scalar Laplace matrix), BP4 (vector-valued Laplace matrix), and BP5 (scalar Laplace matrix, collocation setting with Gauss-Lobatto quadrature on $n_q = (p + 1)^2$ points) according to Fischer et al. (2020).

$n_q = p + 2$), BP2 (vectorial mass matrix, $n_q = p + 2$), BP3 (scalar Laplace operator, $n_q = p + 2$), and BP5 (scalar Laplace operator, $n_q = p + 1$, Gauss-Lobatto quadrature). Figure 13 compares the throughput of the basic CG algorithm and of the combined version for BP1–BP5. For large

problem sizes ($\geq 5 \times 10^6$ DoFs/node), a clear trend is visible. While the throughput is limited to 0.8–1.2 GDoFs/sec for the basic CG scheme, the value is around two times higher for the proposed algorithms with 1.8–2.3 GDoFs/sec for all cases. Also note that the BP1, BP2, and BP5 cases with an arithmetically lighter matrix-vector product saturate the RAM bandwidth with around 200 GB/s for the combined CG iteration, whereas BP3 and BP4 reach around 170 GB/s bandwidth, as seen above.

6.4. Comparison of different hardware

In the following, we present results obtained on a dual-socket AMD Epyc 7742 CPU and a Nvidia Tesla V100 GPU. The AMD CPU consists of 2×64 cores running at 2.25 GHz and uses code compiled for the AVX2 instruction set extension (4-wide SIMD). This gives an arithmetic peak performance of 4.61 TFlop/s. The memory configuration uses 2×8 channels of DDR4-3200, resulting in a peak bandwidth of 410 GB/s and a measured STREAM triad bandwidth of 290 GB/s. The size of the last-level cache is 4 MB per core or 512 MB in total. The Nvidia V100 provides an arithmetic peak performance of 7.8 TFlop/s, a peak memory bandwidth of 900 GB/s, and a measured bandwidth of 720 GB/s. The performance specifications of the V100 GPU are considerably higher on the GPU compared to the two CPU systems, but with a less sophisticated cache infrastructure.

6.4.1. Variation of the polynomial degree on Intel and AMD CPUs

The lower panel of Figure 11 shows the experiment from Subsection 6.3.1, varying the polynomial degree on an AMD Epyc 7742 node. Here, we observe a maximal throughput of 4 GDoFs/sec and maximal speedups of $3\times$ compared to the baseline CG solver (compared to 2 GDoFs/sec and $2\times$ speedup in the case of Intel). This difference can be explained by the higher arithmetic performance of the AMD system, shifting the performance limit with an achieved bandwidth of around 270 GB/s closer to the memory throughput limit of 290 GB/s. An interesting observation is the fact that the performance does not drop for the high polynomial degrees $p > 5$. This can be contributed to larger caches as well as to the AVX2 instruction-set extension with vectorization aggregating work from only 4 cells together, which increases the benefit of the combination of “pre,” “mat-vec,” and “post” regions.

6.4.2. BP5 on CPU and GPU

As a last experiment, we run Algorithm 5 on a GPU architecture. Given the much smaller available cache size

compared to compute units, we have not been able to embed the vector access regions into the cell-based evaluation of the matrix-vector product. As a result, we propose to run the three regions “pre,” “mat-vec,” and “post” each as a separate kernel with its own kernel call. Furthermore, the matrix-vector product uses a precomputed final coefficient on the GPU, due to a different balance between arithmetic performance, available registers, and memory bandwidth compared to CPUs, see also the analysis in Świrydowicz et al. (2019). Details on the GPU infrastructure of deal.II can be found in Ljungkvist (2017) and Kronbichler and Ljungkvist (2019).

Figure 14 shows the throughput of the regular and the combined CG method run on a single GPU device on Summit³ (Nvidia V100). For small problem sizes, a clear benefit can be observed due to the reduced number of kernel calls (3). For large problem sizes, a speedup of about 18% with 2.8 GDoF/s is reached. Note that this represents a considerably lower improvement, which is due to the missing overlap between the “pre” and “post” operations. Nonetheless, Algorithm 5 also improves the throughput for lower sizes because of fewer kernel launches. Reducing the number of kernel calls in CG on GPUs has been also the motivation in Aliaga et al. (2013), Dehnavi et al. (2011), Rupp et al. (2016), and Chalmers and Warburton (2020). The contribution by Rupp et al. (2016) was even able to obtain two kernel calls for vector-matrix-multiplication implementations based on sparse matrices. However, the latter concept is not straightforwardly extensible to matrix-free finite-element computations with contributions to the result vector being accumulated from computations on several cells, as discussed in Section 5.1. Since the GPU’s

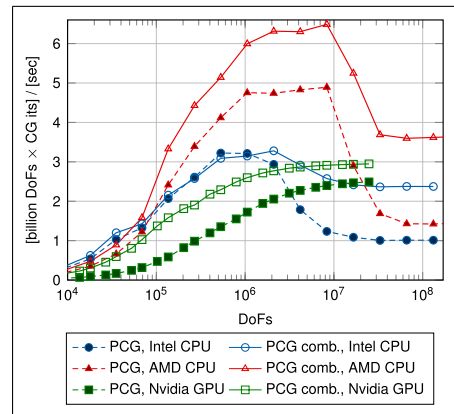


Figure 14. BP5: Throughput over the problem size on a single node for the basic preconditioned CG method and the proposed combined variant.

high-bandwidth memory is limited to 16 GB, the maximum size of the problem that can be run is considerably smaller on the GPU.

With the proposed combined CG method, the CPU results appear more beneficial than the GPU results: Given that both the memory bandwidth and arithmetic performance is considerably higher on a single V100 device than on the dual-socket Intel and AMD systems, one would expect best performance on the GPU. However, the Intel result is only 20% lower than the GPU, and the AMD result from RAM is 20% better than on the GPU, because of the reduction of memory transfer between the “pre” and “post” regions. Furthermore, the CPU reach a higher throughput for moderate sizes when the data fits into caches.

7. Conclusions

We have presented an implementation of the conjugate gradient method that aims to minimize the access to auxiliary vectors for the case of high-order matrix-free finite-element implementations with a diagonal preconditioner. The development was motivated by the observation that matrix-free operator evaluation has become so fast that AXPY-style vector updates, dot products and the application of the preconditioner can consume around two thirds of the total runtime for large problem sizes on modern hardware, relevant for example for fluid dynamics applications. The proposed solver relies on interleaving the vector updates and dot products of the conjugate gradient iteration with the loop through the mesh elements of the matrix-vector product, combined with redundant applications of the preconditioner and summation of auxiliary quantities to break the dependencies. We have shown that around 90% of the vector entries in the three active vectors of a CG iteration can be re-used from fast cache memory, resulting in a single load and store operation for each vector.

Both node-level performance analyses and strong/weak-scaling studies on up to 147,456 CPU cores confirm the suitability of the proposed algorithm for modern hardware. Experiments have been conducted on CPU-based (Intel Xeon Platinum 8174, AMD Epyc 7742) and GPU-based (Nvidia Tesla V100 GPU) compute nodes for a large variety of polynomial degrees, geometric descriptions, and PDEs (scalar/vector-valued mass/Laplace matrix). Compared to a baseline CG solver as well as optimized pipelined CG and s -step CG implementations, speedups of 2–3× have been reported. Besides reducing the memory transfer, the proposed method allows to run memory-heavy vector operations near the arithmetic-heavy matrix-free operator evaluation. As a result, new tuning opportunities for implementing matrix-free methods appear, allowing to gain performance from computing, for example, redundant

geometry information on the fly with reduced memory transfer, an operation that might not be beneficial for the matrix-vector product alone.

Future work aims to extend the algorithm towards the data dependencies imposed by discontinuous Galerkin discretizations as well as more sophisticated preconditioners with longer-range data dependencies. Furthermore, it would be useful to apply analysis and transformation tools from compiler constructions to replace the current manual dependency management for interleaving the matrix-vector product with vector updates and inner products by a more automatic approach based on hardware characteristics, which would make the application to other algorithms, like BiCGStab or GMRES, simpler.

Acknowledgments

The authors acknowledge collaboration with Momme Allalen, Daniel Arndt, Paddy Ó Conbhuí, Prashanth Kanduri, Karl Ljungkvist, Alexander Roschlaub, Bruno Turcksin, as well as the deal. II community.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the Bayerisches Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen (KONWIHR) through the projects “Performance tuning of high-order discontinuous Galerkin solvers for SuperMUC-NG” and “High-order matrix-free finite element implementations with hybrid parallelization and improved data locality.” The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (LRZ, www.lrz.de) through project id pr83te.

ORCID iD

Martin Kronbichler  <https://orcid.org/0000-0001-8406-835X>

Notes

1. https://github.com/kronbichler/mf_data_locality, retrieved on 12 May 2022.
2. <https://top500.org/system/179,566/>, retrieved on 4 January 2021.
3. <https://www.top500.org/system/179,397/>, retrieved on 11 February 2021.

References

- Aliaga JJ, Pérez J, Quintana-Ortí ES, et al. (2013) Reformulated conjugate gradient for the energy-aware solution of linear systems on GPUs. In: 2013 42nd International Conference on Parallel Processing. Lyon, France, 01–04 October 2013, IEEE, pp. 320–329.
- Arndt D, Bangerth W, Blais B, et al. (2020a) The deal.II library, version 9.2. *Journal of Numerical Mathematics* 28(3): 131–146. DOI:10.1515/jnma-2020-0043. <https://dealii.org>
- Arndt D, Bangerth W, Davydov D, et al. (2021) The deal.II finite element library: design, features, and insights. *Computers & Mathematics with Applications* 81: 407–422. DOI:10.1016/j.camwa.2020.02.022
- Arndt D, Fehn N, Kanschat G, et al. (2020b) ExaDG – high-order discontinuous Galerkin for the exa-scale. In: Bungartz HJ, Reiz S, Uekermann B, et al. (eds) *Software for Exascale Computing – SPPEXA 2016–2019, Lecture Notes in Computational Science and Engineering* 136. Cham: Springer International Publishing, 189–224. DOI:10.1007/978-3-030-47956-5_8
- Bangerth W, Burstedde C, Heister T, et al. (2011) Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Transactions on Mathematical Software* 38: 1–28. DOI:10.1145/2049673.2049678
- Bauer S, Drzisga D, Mohr M, et al. (2018) A stencil scaling approach for accelerating matrix-free finite element implementations. *SIAM Journal on Scientific Computing* 40(6): C748–C778. DOI:10.1137/17m1148384
- Brown J (2010) Efficient nonlinear solvers for nodal high-order finite elements in 3D. *Journal of Scientific Computing* 45(1–3): 48–63. DOI:10.1007/s10915-010-9396-8
- Burstedde C, Wilcox LC and Ghattas O (2011) p4est: scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM J. Sci. Comput* 33(3): 1103–1133. DOI:10.1137/10079163. <http://p4est.org>
- Chalmers N and Warburton T (2020) Portable high-order finite element Kernels I: streaming operations. preprint ArXiv: 2009.10917.
- Charrier DE, Hazelwood B, Tutlyayeva E, et al. (2019) Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver. *The International Journal of High Performance Computing Applications* 33(5): 973–986. DOI:10.1177/1094342019842645
- Chronopoulos AT and Gear CW (1989) S-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics* 25(2): 153–168. DOI:10.1016/0377-0427(89)90045-9
- Cornelis J, Cools S and Vanroose W (2018) The communication-hiding conjugate gradient method with deep pipelines. ArXiv e-prints 1801.4728v3.
- Dehnavi MM, Fernández DM and Giannacopoulos D (2011) Enhancing the performance of conjugate gradient solvers on graphic processing units. *IEEE Transactions on Magnetics* 47(5): 1162–1165.
- Demkowicz L, Oden J and Rachowicz W (1990) A new finite element method for solving compressible Navier–Stokes equations based on an operator splitting method and h-p adaptivity. *Computer Methods in Applied Mechanics and Engineering* 84(3): 275–326. DOI:10.1016/0045-7825(90)90081-v
- Deville MO, Fischer PF and Mund EH (2002) *High-Order Methods For Incompressible Fluid Flow*, Vol. 9. Cambridge: Cambridge University Press.
- Eisenstat SC (1981) Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM Journal on Scientific and Statistical Computing* 2(1): 1–4. DOI:10.1137/0902001
- Eller PR, Hoeffler T and Gropp W (2019) Using performance models to understand scalable Krylov solver performance at scale for structured grid problems. In: Proceedings of the ACM International Conference on Supercomputing. Phoenix, AZ, June 26–28, 2019. ACM. DOI:10.1145/3330345.3330358. URL DOI: 10.1145/3330345.3330358
- Fehn N, Wall WA and Kronbichler M (2018) Efficiency of high-performance discontinuous Galerkin spectral element methods for under-resolved turbulent incompressible flows. *International Journal for Numerical Methods in Fluids* 88(1): 32–54. DOI:10.1002/fld.4511
- Fischer P, Kerkemeier S, Peplinski A, et al. (2021) Nek5000 Web page. <https://nek5000.mcs.anl.gov>
- Fischer P, Min M, Rathnayake T, et al (2020) Scalability of high-performance PDE solvers. *The International Journal of High Performance Computing Applications* 34(5): 562–586. DOI: 10.1177/1094342020915762
- Ghysels P and Vanroose W (2014) Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing* 40(7): 224–238. DOI:10.1016/j.parco.2013.06.001 7th Workshop on Parallel Matrix Algorithms and Applications.
- Grigori L and Tisot O (2019) Scalable linear solvers based on enlarged krylov subspaces with dynamic reduction of search directions. *SIAM Journal on Scientific Computing* 41(5): C522–C547.
- Guermont JL, Maier M, Popov B, et al. (2021) Second-order invariant domain preserving approximation of the compressible Navier–Stokes equations. *Computer Methods in Applied Mechanics and Engineering* 375: 113608. DOI:10.1016/j.cma.2020.113608
- Hager G and Wellein G (2011) *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton: CRC Press.
- Kempf D, Heß R, Müthing S, et al. (2021) Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures. *ACM Transactions on Mathematical Software* 47(1): 1–31. DOI:10.1145/3424144
- Kolev T, Fischer P, Min M, et al. (2021) Efficient exascale discretizations: High-order finite element methods. *The International Journal of High Performance Computing*

- Applications* 35(6): 527–552. DOI:10.1177/10943420211020803
- Krank B, Fehn N, Wall WA, et al. (2017) A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. *Journal of Computational Physics* 348: 634–659. DOI:10.1016/j.jcp.2017.07.039
- Kronbichler M and Allalen M (2018) Efficient high-order discontinuous Galerkin finite elements with matrix-free implementations. In: Bungartz HJ, Kranzlmüller D, Weinberg V, et al. (eds) *Advances and New Trends in Environmental Informatics*. Cham: Springer, pp. 89–110. DOI:10.1007/978-3-319-99654-7_7
- Kronbichler M, Diagne A and Holmgren H (2018) A fast massively parallel two-phase flow solver for microfluidic chip simulation. *The International Journal of High Performance Computing Applications* 32(2): 266–287. DOI:10.1177/1094342016671790
- Kronbichler M and Kormann K (2012) A generic interface for parallel cell-based finite element operator application. *Computers and Fluids* 63: 135–147. DOI:10.1016/j.compfluid.2012.04.012
- Kronbichler M and Kormann K (2019) Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM Transactions on Mathematical Software* 45(3): 1–40. DOI:10.1145/3325864
- Kronbichler M and Ljungkvist K (2019) Multigrid for matrix-free high-order finite element computations on graphics processors. *ACM Transactions on Parallel Computing* 6(1): 1–32. DOI:10.1145/3322813
- Kronbichler M and Wall WA (2018) A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers. *SIAM Journal on Scientific Computing* 40(5): A3423–A3448. DOI:10.1137/16M110455X
- Ljungkvist K (2017) Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes. In: HPC '17: Proceedings of the 25th High Performance Computing Symposium. San Diego, CA, USA, June 26–28, 2019: Society for Computer Simulation International, pp. 1–12.
- Lockhart S, Bienz A, Gropp W, et al. (2022) Performance analysis and optimal node-aware communication for enlarged conjugate gradient methods. arXiv preprint arXiv:2203.06144.
- Malas TM, Hager G, Ltaief H, et al. (2017) Multidimensional intratitle parallelization for memory-starved stencil computations. *ACM Transactions on Parallel Computing* 4(3): 1–32. DOI:10.1145/3155290
- MehriDehnavi M, El-Kurdi Y, Demmel J, et al. (2013) Communication-avoiding Krylov techniques on graphic processing units. *IEEE Transactions on Magnetics* 49(5): 1749–1752. DOI:10.1109/TMAG.2013.2244861
- Moxey D, Amici R and Kirby M (2020) Efficient matrix-free high-order finite element evaluation for simplicial elements. *SIAM Journal on Scientific Computing* 42(3): C97–C123. DOI:10.1137/19m1246523
- Munch P, Kormann K and Kronbichler M (2021) hyper.deal: an efficient, matrix-free finite-element library for high-dimensional partial differential equations. *ACM Transactions on Mathematical Software* 47(4): 1–34. DOI:10.1145/3469720
- Naumov M (2016) *S-step and communication-avoiding iterative methods*. Technical Report NVR-2016-003. Santa Clara, CA: NVIDIA.
- Orszag SA (1980) Spectral methods for problems in complex geometries. *Journal of Computational Physics* 37(1): 70–92. DOI:10.1016/0021-9991(80)90005-4
- Patara AT (1984) A spectral element method for fluid dynamics: laminar flow in a channel expansion. *Journal of Computational Physics* 54(3): 468–488. DOI:10.1016/0021-9991(84)90128-1
- Rupp K, Weinbub J, Jünger A, et al. (2016) Pipelined iterative solvers with kernel fusion for graphics processing units. *ACM Transactions on Mathematical Software* 43(2): 11:1–11:27. DOI:10.1145/2907944
- Saad Y (1985) Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM Journal on Scientific and Statistical Computing* 6(4): 865–881. DOI:10.1137/0906059.
- Solomonoff A (1992) A fast algorithm for spectral differentiation. *Journal of Computational Physics* 98(1): 174–177. DOI:10.1016/0021-9991(92)90182-X
- Sun T, Mitchell L, Kulkarni K, et al. (2020) A study of vectorization for matrix-free finite element methods. *The International Journal of High Performance Computing Applications* 34(6): 629–644. DOI:10.1177/1094342020945005
- Świrydowicz K, Chalmers N, Karakus A, et al. (2019) Acceleration of tensor-product operations for high-order finite element methods. *The International Journal of High Performance Computing Applications* 33(4): 735–757. DOI:10.1177/1094342018816368
- Treibig J, Hager G and Wellein G (2010) LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: 2010 39th International Conference on Parallel Processing Workshops. San Diego, CA, USA, 13–16 September 2010, pp. 207–216. DOI:10.1109/ICPPW.2010.38
- Tufo HM and Fischer PF (1999) Terascale spectral element algorithms and implementations. In: Proceedings of the 1999 ACM/IEEE conference on Supercomputing. Portland, OR, USA, 13–19 November 1999: ACM, p. 68. DOI:10.1109/SC.1999.10035

Author biographies

Martin Kronbichler is a Professor at University of Augsburg, Germany. He holds a PhD degree in scientific computing with specialization in numerical analysis from Uppsala University, Sweden (2012). His research interests include high-order finite element methods for flow problems with matrix-free implementations, efficient numerical linear

algebra, and their parallel and high-performance implementation on emerging exascale hardware using generic numerical software.

Dmytro Sashko is a PhD student at The University of Queensland. His research interests lie in the fields of computational fluid dynamics, high-performance computing and software development.

Peter Munch is a research associate and PhD student at Helmholtz-Zentrum Hereon and University of Augsburg. He is one of the principal developers of the open-source finite-element library deal.II. His research interests lie in the fields of high-performance computing, scientific software development, computational fluid mechanics, and computational plasma physics—with a focus on matrix-free methods, software design, and iterative solvers.

Paper II



Efficient Application of Hanging-Node Constraints for Matrix-Free High-Order FEM Computations on CPU and GPU

Peter Munch^{1,2} (), Karl Ljungkvist³, and Martin Kronbichler³

¹ Helmholtz-Zentrum Hereon, Geesthacht, Germany
peterrmuench@gmail.com

² Technical University of Munich, Munich, Germany
³ Uppsala University, Uppsala, Sweden

Abstract. This contribution presents an efficient algorithm for resolving hanging-node constraints on the fly for high-order finite-element computations on adaptively refined meshes, using matrix-free implementations. We concentrate on unstructured hex-dominated meshes and on multi-component elements with nodal Lagrange shape functions in at least one of their components. The application of general constraints is split up into two distinct operators, one specialized in the hanging-node part and a generic one for the remaining constraints, such as Dirichlet boundary conditions. The former implements in-face interpolations efficiently by a sequence of 1D interpolations with sum factorization according to the refinement configuration of the cell. We discuss ways to efficiently encode and decode such refinement configurations. Furthermore, we present distinct differences in the interpolation step on GPU and CPU, as well as compare different vectorization strategies for the latter. Experimental comparisons with a state-of-the-art algorithm that does not exploit the tensor-product structure show that, on CPUs, the additional costs of cells with hanging-node constraints can be reduced by a factor of 5–10 for a Laplace operator evaluation with high-order elements ($k \geq 3$) and affine meshes. For non-affine meshes, the costs for the application of hanging-node constraints can be completely hidden behind the memory transfer. The algorithm has been integrated into the open-source finite-element library `deal.II`.

Keywords: Adaptively refined meshes · Finite element methods · High order · Hanging-node constraints · Matrix-free operator evaluation · Node-level optimization · SIMD vectorization · Manycore optimizations

This work was supported by the Bayerisches Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen (KONWIHR) through the projects “Performance tuning of high-order discontinuous Galerkin solvers for SuperMUC-NG” and “High-order matrix-free finite element implementations with hybrid parallelization and improved data locality”. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (LRZ, www.lrz.de) through project id pr83te.

© Springer Nature Switzerland AG 2022

A.-L. Varbanescu et al. (Eds.): ISC High Performance 2022, LNCS 13289, pp. 133–152, 2022.
https://doi.org/10.1007/978-3-031-07312-0_7

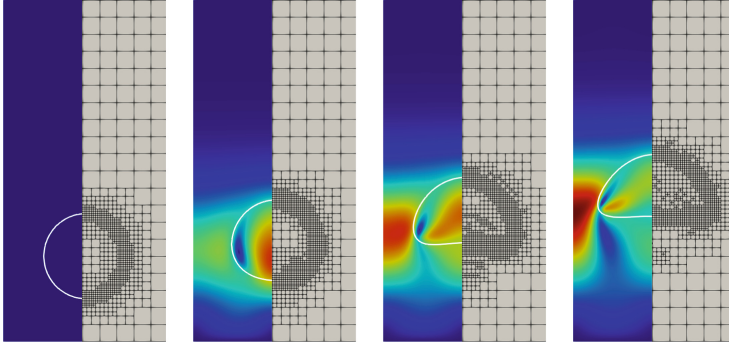


Fig. 1. The “rising-bubble” benchmark in 2D as an example of a simulation using AMR: left) velocity contour and zero level-set isoline and right) mesh resolving the resulting surface-tension forces at the interface accurately. The results have been obtained with the open-source two-phase solver `adaflo` [12].

1 Introduction

Matrix-free high-order finite element methods (FEM) are used to efficiently solve different types of partial differential equations (PDE) with applications in fluid mechanics [8, 11], solid mechanics [7], mesh smoothing [1], or computational plasma physics [21]. The applicability of matrix-free methods to massively parallel computers has been demonstrated multiple times in the past [9].

In order to reduce the computational costs, one can adaptively refine meshes (AMR) to concentrate the work on the most relevant areas of the computational domain, where, e.g., the solution has high gradients or discontinuities (see Fig. 1). One of the ways to refine meshes, the non-conforming refinement strategy, refines cells independently by replacing parent cells by children cells (octants for hexahedral cells) and results in the occurrence of hanging nodes (see Fig. 2a). In order to guarantee the continuity, i.e., H^1 conformity, of the tentative solution at these places, hanging-node constraints have to be applied [25]. Although not strictly needed, many codes limit the difference in refinement of neighboring cells to one (1-irregular mesh), since a more abrupt transition in most cases does not lead to a significant reduction of time to solution to justify the implementation complexity.

Simulations with hanging nodes need iterative solvers that can cope with these nodes in a robust manner (e.g., geometric multigrid methods [13, 20]) and algorithms to apply hanging-node constraints efficiently. Since matrix-free methods need to interpolate the constraints in each operator evaluation of each iteration, the efficient application of hanging-node constraints is a crucial HPC ingredient for the fast solution of PDEs with FEM on adaptively refined meshes and the core of the present publication.

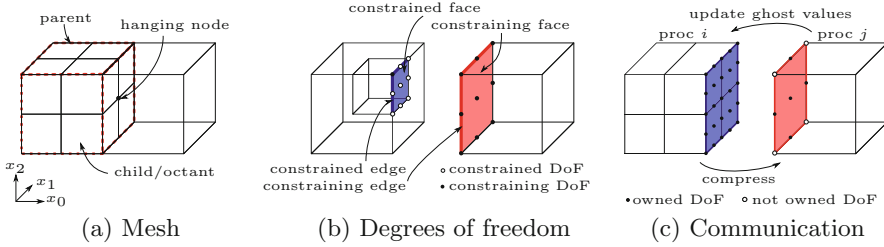


Fig. 2. Definition of the most important terms of hanging-node constraints on a hexahedral mesh, incl. the communication pattern between two processes $i < j$, with one possessing only one unrefined cell and the other all children of a cell.

1.1 Matrix-Free Operator Evaluation

In this work, we consider matrix-free implementations for general meshes, which compute the integrals underlying a finite-element discretization on the fly. Here, the operator evaluation performs a loop over all cells and applies the effect of element stiffness matrices on a vector with the following basic steps [14]:

$$v = \mathcal{A}(u) = \sum_e \mathcal{G}_e^T \circ \mathcal{C}_e^T \circ \tilde{\mathcal{S}}_e^T \circ \mathcal{D}_e \circ \mathcal{S}_e \circ \mathcal{C}_e \circ \mathcal{G}_e \circ u. \quad (1)$$

In the first step, the degrees of freedom (DoFs) relevant for each cell e are gathered by \mathcal{G}_e from the global source vector u . In the remainder of this study, these unknowns are called cell-relevant DoFs. The application of \mathcal{C}_e interpolates from these cell-relevant DoFs to the cell-local values $u_{e,j}$ in the polynomial expansion of the finite-element solution $u_h|_e = \sum_j \varphi_j^{(e)} u_{e,j}$, consistent with all constraints due to hanging nodes and boundary conditions. Subsequently, values and/or gradients of u_h are evaluated at the quadrature points via \mathcal{S}_e and the computed quantities are processed on each quadrature point by \mathcal{D}_e . The application of $\tilde{\mathcal{S}}_e^T$ represents the multiplication by the finite-element test functions and the summation over quadrature points. For simplicity of notation, we assume a symmetric (self-adjoint) PDE operator with $\tilde{\mathcal{S}}_e = \mathcal{S}_e$ in this work. Finally, \mathcal{C}_e and \mathcal{G}_e are applied in reverse order during multiplication by the finite-element test functions and the results are added into the global destination vector v .

The operator \mathcal{G}_e is a Boolean matrix (DoF map) representing indirect addressing into the vectors u and v . In the past decades, significant efforts went into optimizing the evaluation operator \mathcal{S}_e . In particular, the exploitation of the structure of the shape functions and quadrature points allows replacing a general dense interpolation matrix by more efficient procedures. For example, sum factorization [19,22] (see Algorithm 1) performs a sequence of 1D interpolation steps to evaluate vectors at the quadrature points for tensor-product polynomials, reducing the computational complexity from $\mathcal{O}(k^{2d})$ to $\mathcal{O}(dk^{d+1})$ for scalar Lagrange elements of degree k . On a per-unknown metric, operation (1) with sum factorization implies an arithmetic complexity $\mathcal{O}(k)$ and a memory access complexity $\mathcal{O}(1)$. This makes the approach the most efficient way to compute the

Algorithm 1: Function that performs an inplace interpolation from the expansion coefficients $u_{e,i}$ to the quadrature points by a sequence of 1D interpolations. On the GPU, the (thread) indices $[i_0, i_1, i_2]$ are given by the runtime environment and the synchronization between threads in different parallel foreach regions can be accomplished by an explicit function call.

```

1 for direction ← 0 to dim do
2   parolleforeach index ∈ {[i0, i1, i2] | 0 ≤ i0, i1, i2 ≤ k} do
3     /* interpolate along line (def. by index & direction)          */
4     value ← interp_matrix[index[direction]]. * data(index, direction)
5     parolleforeach index ∈ {[i0, i1, i2] | 0 ≤ i0, i1, i2 ≤ k} do
6       data(index) = value

```

action of a discretized differential operator on vectors for higher-order finite elements with degree $k \geq 3$ on general (deformed) meshes [9, 14]. On today’s hardware, the boost in efficiency is primarily due to the reduction in memory access by skipping a memory-intensive assembled matrix in favor of on-the-fly computations on cached data. Implementations specialized for CPUs [1, 14, 15, 21] and GPUs [1, 16, 18, 26] are available in the literature.

Given the nested loop structures with different strides and data dependencies when interpolating in different directions with sum factorization, automatic vectorization leads to poor performance on modern CPUs and explicit outer-loop vectorization based on intrinsics either within a cell, i.e., across DoFs/quadrature points, or across cells, with each vector lane processing another cell, is necessary [15]. The latter is assumed for the CPU implementation, whereas in a GPU implementation, which runs parallel “threads” in a team, a thread works on an individual DoF within cells. Note that the algorithms and performance translate similarly to other alternatives [26], making the conclusions of this publication generic.

1.2 Application of Constraints

The constraint operator \mathcal{C}_e relates cell-local DoFs to cell-relevant DoFs. Cell-local DoFs can be either constrained or not (○ vs. ● in Fig. 2b). Constrained DoFs depend on constraining DoFs in the form of affine combinations $x_i = C_{ij}x_j + b_i$ with possible inhomogeneity \mathbf{b} . Examples of constraints are Dirichlet boundary conditions, periodic boundary conditions, and hanging-node constraints. Although \mathbf{C} is generally sparse and can be efficiently stored in compressed row storage (CRS) format, it becomes locally dense for certain constraint types. For example, hanging-node constraints on faces relate all $n_{\text{dofs_per_face}}$ constrained DoFs of the subsurface on the fine side to the same number of constraining DoFs on the coarse side, see Fig. 2b. For the sake of brevity, we will call faces with DoFs with hanging-node constraints “constrained faces”; in a similar way, we will use the terms “constrained edges” and “constraining faces/edges”. For tensor-

product elements and scalar polynomial elements of degree k , the naive evaluation by a dense matrix of size $\mathcal{O}(n_{\text{dofs_per_face}}^2)$ implies memory and arithmetic costs of $\mathcal{O}(k^{2(d-1)})$. These costs become the bottleneck of matrix-free algorithms of complexity $\mathcal{O}(dk^{d+1})$ for higher k .

1.3 Related Work

The interpolation of data to subcells or subfaces is a common operation in the context of FEM. For example, geometric multigrid methods [20] need to prolongate and restrict between cells and their children. In the case of discontinuous Galerkin methods and meshes with hanging nodes, data of neighboring cells have to be interpolated to subfaces for integration of fluxes on faces [15, 17].

The development of adaptive solvers is a highly active field, as evidenced by the recent publication [23], necessitating advances in fast hanging-node algorithms. Recently, Cervený et al. [6] presented a global operator \mathcal{C} that can handle arbitrary irregular meshes obtained by anisotropic refinement. Kronbichler and Kormann [14] proposed a general way to process constraints during matrix-free loops in the context of FEM by a combined operator $(\mathcal{C} \circ \mathcal{G})_e$. Even though this approach identifies similar rows in the constraint matrix \mathcal{C} to reduce memory access, it suffers from an exceeding complexity of the naive evaluation at higher orders ($\mathcal{O}(k^{2(d-1)})$). In the context of spectral element methods [8, 10] and FEM [16, 18], a special-purpose hanging-node algorithm for 1-irregular meshes with only hypercube-shaped cells has been used that relies on the update of the DoF map \mathcal{G}_e and efficient inplace interpolations. While the previous publications provide a clear understanding of the 2D case, this is not yet the case in 3D, in particular regarding recent advances in modern hardware, such as SIMD vectorization. The main difficulty are the 137 refinement configurations, as opposed to only 13 cases in 2D, and the appearance of constraints along edges.

1.4 Our Contributions

We present an algorithm to efficiently resolve constraints in the form of \mathcal{C}_e and \mathcal{C}_e^T of Eq. (1) with hanging-node contributions in the context of matrix-free FEM on CPUs and GPUs. The algorithm is built on the observation that, for Lagrange elements, the constraint matrix can be factored into a general-purpose operator and a special-purpose operator that can exploit the most efficient interpolation routines, e.g., sum factorization for tensor-product elements, and thus reduce the computational complexity to $\mathcal{O}((d-1) \cdot k^d)$, similarly to the operators developed in [8, 10, 16, 18]. We give a detailed description of the special-purpose operator in 3D, which is crucial for the efficient implementation of our proposed algorithm. We assume that 1) the mesh has hypercube-shaped cells, 2) these cells have at most two children in each direction, and 3) the mesh is 1-irregular.

The algorithm presented in this publication has been integrated into the open-source C++-based FEM library `deal.II` [2, 3]. The implementation is used by default in its matrix-free infrastructure; the correctness is checked by several hundreds of tests.

The remainder of this publication is structured as follows. In Sects. 2 and 3, we introduce the algorithm and discuss data structures and implementation details. In Sect. 4, we present performance results for serial runs and discuss the benefits of the given algorithm for parallel simulations. The results are obtained on Intel and AMD CPUs as well as on NVIDIA GPUs. Conclusions are given in Sect. 5.

2 Algorithm

We split $(\mathcal{C} \circ \mathcal{G})_e$ into three contributions $(\mathcal{C}^{\text{HN}} \circ \mathcal{C}^{\text{GP}} \circ \mathcal{G})_e$ with \mathcal{C}^{HN} dedicated to the hanging-node constraints and \mathcal{C}^{GP} to the remaining (general-purpose) constraints. The sequence of these contributions can be chosen arbitrarily. Two sequences have properties suitable for an HPC implementation and will be used in the following: (i) $\mathcal{C}_e^{\text{HN}} \circ \mathcal{G}_e^a \circ \mathcal{C}^{\text{GP}}$ and (ii) $\mathcal{C}_e^{\text{HN}} \circ \mathcal{C}_e^{\text{GP}} \circ \mathcal{G}_e^b$. Approach (i) applies the general-purpose constraints on the global vectors and then proceeds with operations on the element level, gathering the cell-relevant DoFs and applying the hanging-node constraints on the current cell. In practice, this approach involves a global pre- and postprocessing step $(\mathcal{C}^{\text{GP}}, (\mathcal{C}^{\text{GP}})^T)$ before and after the matrix-free loop (1). In the literature, it is common to use \mathcal{C}^{GP} for the application of homogeneous Dirichlet boundary conditions, for which it simplifies to zeroing out the entries constrained by Dirichlet boundary conditions. We use it also for more complex types of constraints, such as those constraining the normal or tangential components of a vector-valued solution. On a GPU, one would perform the preprocessing step, the matrix-free operator application, and the postprocessing step sequentially by three kernel calls. Approach (ii) applies the general constraints after gathering the cell-relevant DoFs within the loop (1). This approach accesses global vectors only once and operates exclusively on the fixed-size working set of a cell, assuming that caches are large enough to hold all cell-local values, which is the case on modern CPUs.

The operator $\mathcal{C}_e^{\text{HN}}$ independently applies the hanging-node constraints on the element level. The number of constrained and constraining DoFs is the same, i.e., $size(u_e) = size(\mathcal{C}_e^{\text{HN}} \circ u_e)$. By presorting the indices, this operation becomes a simple in-place line or face interpolation, which can be handled efficiently, e.g., by sum factorization. In approach 1), the presort can be accomplished by replacing the global indices of DoFs on constrained edges/faces by the constraining counterparts in the DoF map \mathcal{G}_e^a . For this operation, one needs to consider the orientation of the edges/faces within unstructured meshes, for which an extended version of the algorithm proposed in [24] can be used. In the case of approach (ii), \mathcal{G}_e^b can be constructed by replacing the constrained indices in the DoF map \mathcal{G}_e^a by their constraining counterparts in $\mathcal{C}_e^{\text{GP}}$.

Our algorithm can treat components of vectorial elements individually. However, some components might not be able or might not need to be treated by the proposed algorithm (e.g., non-nodal elements vs. mixed elements with discontinuous Galerkin components). The refinement configuration of a cell and the information on whether our fast hanging-node-constraint algorithm needs to

be applied for the given component can be efficiently combined on the fly by a simple Boolean operation.

3 Implementation Details

In the following, we discuss how to encode the refinement configuration of a hypercube-shaped cell so that the information can be efficiently decoded during the interpolation phase. Furthermore, different vectorization strategies for the interpolation step are presented. Note that, at the time of writing, `deal.II` falls back to the general-purpose algorithm for non-hypercube cells on mixed meshes, which also contain cell shapes like simplices.

3.1 Data Structures

In this subsection, we discuss the data structures of \mathcal{C}^{GP} , of $(\mathcal{C}^{\text{GP}} \circ \mathcal{G})_e$, and of $\mathcal{C}_e^{\text{HN}}$. By moving the hanging-node constraints to a special-purpose data structure, the global operator \mathcal{C}^{GP} is generally sparse so that a matrix-vector multiplication with a sparse matrix (as provided by `cuSPARSE`) is applicable. Data structures of the merged operator $\mathcal{C}_e^{\text{GP}} \circ \mathcal{G}_e$ for CPUs have been proposed in [14]. They consist of an extended DoF map, indicators of constrained DoFs, and pointers to the rows of the constraint matrix. In order to minimize memory consumption, the value array of the sparse matrix is only stored for unique rows.

The hanging-node-constraint operator $\mathcal{C}_e^{\text{HN}}$ requires information regarding the refinement configuration of each cell and appropriate face-subface interpolation matrices. For tensor-product elements, one only needs to store 1D interpolation matrices to the two 1D subfaces. As a result, we were able to derive—by exploiting the structure of $(\mathcal{C}^{\text{HN}} \circ \mathcal{C}^{\text{GP}} \circ \mathcal{G})_e$ —an efficient and flexible algorithm whose memory consumption is $\mathcal{O}(N_{\text{cells}})$ and is independent of the degree k .

3.2 Refinement Configuration

The refinement of a cell relative to the neighboring cells can be described as a pair $(\textit{subcell}, \textit{face})$ in 2D (not considered in the following) and as a triple $(\textit{subcell}, \textit{face}, \textit{edge})$ in 3D. The first entry “subcell” indicates the octant within the parent cell, the second entry “face” the direction along which constrained faces (i.e., coarser neighboring cells) appear, and the third entry “edge” the direction of constrained edges. Note that, if a face is constrained, all its bounding edges are also constrained. Furthermore, we utilize the fact that only one of the faces/edges along a direction can be constrained as the other side necessarily belongs to the same leaf in the octree. Figure 3 visualizes all possible values of the entries of the triple. All 137 resulting valid refinement configurations are:

- the unconstrained case ($\{(0, 0, 0)\}$),
- 56 cases with at least one constrained face

$$\{(\textit{subcell}, \textit{face}, 0) \mid 0 \leq \textit{subcell} < 8 \wedge 1 \leq \textit{face} < 8\},$$

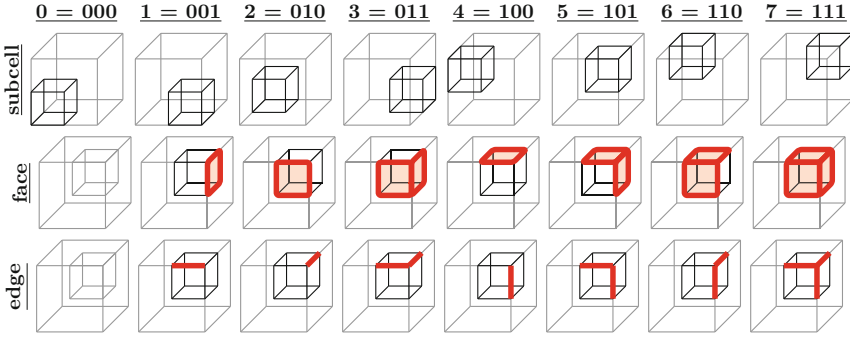


Fig. 3. Depiction of $0 \leq \text{subcell}, \text{face}, \text{edge} < 8$. The latter two refinement-configuration entries are plotted for $\text{subcell} = 5$. The resulting 137 configurations can be described by a triple $(\text{subcell}, \text{face}, \text{edge})$ or 8 bits.

- 56 cases with at least one constrained edge

$$\{(\text{subcell}, 0, \text{edge}) \mid 0 \leq \text{subcell} < 8 \wedge 1 \leq \text{edge} < 8\},$$

- and 24 cases in which a face and the edge orthogonal to it are constrained

$$\{(\text{subcell}, i, i) \mid 0 \leq \text{subcell} < 8 \wedge i \in \{1, 2, 4\}\}.$$

The triple could be encoded by a 9-bit integer. Alternatively, one could exploit the observation that the *face* and *edge* entries are either identical or one of them has the value zero to save a bit and to encode the information as a quadruple (containing: *subcell*, whether at least one face is constrained, whether at least one edge is constrained, non-zero entries of *face/edge*). The corresponding encoding/decoding routines between the triple (a, b, c) and the quadruple $(\alpha, \beta, \gamma, \delta)$ are:

$$\begin{aligned} (\alpha, \beta, \gamma, \delta) &\leftarrow \text{encode}(a, b, c) = (a, b > 0, c > 0, \max(b, c)) \\ (a, b, c) &\leftarrow \text{decode}(\alpha, \beta, \gamma, \delta) = (a, \beta ? \delta : 1, \gamma ? \delta : 1) \end{aligned}$$

3.3 GPU Interpolation

On the GPU, we use an extended version of Algorithm 1 for the application of hanging-node constraints, as presented in Algorithm 3. During the in-place interpolation (see Fig. 4a), threads need to determine whether 1) a DoF is constrained and if so 2) which 1D interpolation matrix should be used. The information can be extracted simply with bitwise operations from the refinement configuration, as shown in Algorithms 2 and 3. Our approach results in threads being idle during the interpolation if the DoFs processed by those threads are not constrained. Nevertheless, it turns out that this approach is very competitive, since it is approximately as expensive as the interpolation step from the nodal coefficients $u_{e,j}$ to the quadrature points as they have the same sequence of operations plus an additional instruction to compute the corresponding masks.

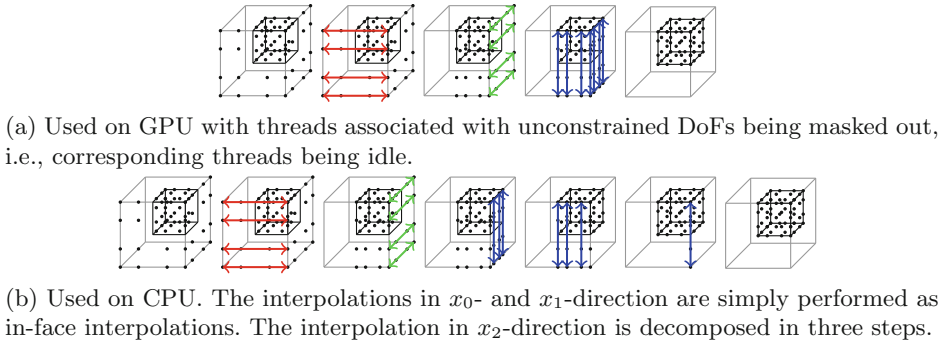


Fig. 4. Hanging-node-constraint application via sum factorization for 3D and $k = 3$ on GPU and CPU for a configuration with coarser neighbors at the right and in front of the highlighted cell.

3.4 CPU Interpolation

On the CPU, we use a different approach to perform the interpolations without checks on the DoF level. Motivated by the fact that only the pair (*face, edge*) determines the interpolation steps and these steps are additive, we can construct an algorithm that has a minimal number of if-statements (one switch-statement for face and one for edge, each with 8 specialized cases) and a limited number of starting points of faces/edges of the subcell, which can be precomputed at compile time. Figure 4b shows, as an example, the interpolation steps for refinement configuration (5,3,0). The interpolations in x_0 - and x_1 -direction are similar to the GPU version. In contrast, we decompose the interpolation in x_2 -direction in three steps in order to prevent interpolating the values along the shared edges twice.

For the purpose of vectorization, `deal.II` provides a wrapper class with a `std::simd`-like interface, which is built around a fixed-size array and translates instructions into the right instruction-set extension [14]. In the vectorization strategy of `deal.II`, each lane of the array is associated with a distinct cell so that each operation on the wrapper class is performed with a single instruction for all cells in parallel. We will call the collection of cells that are processed at the same time a “cell batch”. Implementations of operations \mathcal{S}_e and \mathcal{D}_e only operate on such vectorized data types; in `deal.II`, the merged operator $\mathcal{C}_e^{\text{GP}} \circ \mathcal{G}_e$ performs the laying out of the data in the right (vectorized, struct-of-arrays) format so that the input to $\mathcal{C}_e^{\text{HN}}$ already has this format.

Algorithm 2: Function `is_dof_constrained(direction, conf, index)` \rightarrow *bool* that returns whether a DoF with index $[i_0, i_1, i_2]$ is constrained in the given direction for a specified refinement configuration.

```

1 rotate data structures conf and index to the right by  $(dim - direction - 1)$ 
2 cell_has_edge_constraint  $\leftarrow$  conf.edge[2]
3  $\forall i \in \{0, 1\}.(\text{cell\_has\_face\_constraint\_i} \leftarrow \text{conf.face}[i])$ 
4  $\forall i \in \{0, 1\}.(\text{dof\_is\_on\_face\_i} \leftarrow (\text{conf.subcell}[i] ? k : 0) = \text{index}[i])$ 
5 if  $\exists i \in \{0, 1\}.(\text{dof\_is\_on\_face\_i} \wedge \text{cell\_has\_face\_constraint\_i})$  then
6 |   return True ;                               /* DoF is constrained on face */
7 else if  $(\forall i \in \{0, 1\}.(\text{dof\_is\_on\_face\_i}))$  and cell_has_edge_constraint then
8 |   return True ;                               /* DoF is constrained on edge */
9 else
10 |  return False ;                               /* DoF is not constrained */
```

Algorithm 3: Function that performs an inplace interpolation by a sequence of 1D interpolations for constrained DoFs, used on GPU. See also comments in Algorithm 1.

```

1 for direction  $\leftarrow$  0 to dim do
2 |   paralleforeach index  $\in \{[i_0, i_1, i_2] \mid 0 \leq i_0, i_1, i_2 \leq k\}$  do
3 |   |   interp_matrix  $\leftarrow$  interpolation_matrices[conf.subcell[direction]]
4 |   |   if is_dof_constrained(direction, conf, index) then
5 |   |   |   value  $\leftarrow$  interp_matrix[index[direction]]. * data(index, direction)
6 |   paralleforeach index  $\in \{[i_0, i_1, i_2] \mid 0 \leq i_0, i_1, i_2 \leq k\}$  do
7 |   |   if is_dof_constrained(direction, conf, index) then
8 |   |   |   data(index) = value
```

However, the cells of a cell batch typically have different refinement configurations if no extra measures are taken, making vectorization of the considered algorithms more complicated. We will consider the following vectorization strategies in Subsect. 4.1: 1) **auto**: Cells with hanging-node constraints are processed individually. In this way, we completely rely on optimizing compilers, which is possible, since all if-statements and loop bounds are constant expressions. Data accesses from individual lanes of the struct-of-arrays storage of DoF values, while reading and writing, are necessary. 2) **grouping**: Cells with the refinement configuration are globally grouped together in a preprocessing step. As a result, all cells of a cell batch have the same refinement configuration. 3) **masking**: Here, we keep the sequence of the cells unmodified as in the case of **auto**, however, we process all geometric entities (6 faces and 12 edges) sequentially entity by entity and check whether they are constrained in any of the lanes of the cell batch. We apply a mask, e.g., using the instruction `vblendvpd` with x86/AVX or `vblendmpd` with x86/AVX-512 instruction-set extension, in order to only alter the relevant lanes.

3.5 Costs of Interpolation

We conclude this section by summarizing the number of floating-point operations that are needed to perform the interpolation for an arbitrary refinement configuration ($subcell, face, edge$) in 3D:

$$\mathcal{K}((\bullet, face, edge)) = \mathcal{K}(face) + \mathcal{K}(edge) = \mathcal{O}(k^3).$$

This value is bounded by the costs of the interpolation from the support points to the quadrature points ($\mathcal{K}(cell) = 3(k+1)^3(1+2k)$). The terms are defined (with $|\bullet|$ counting bits) as:

$$\mathcal{K}(face) = \begin{cases} 0 & \text{for } |face| = 0 \\ \mathcal{K}(single\ face) & \text{for } |face| = 1 \\ 2\mathcal{K}(single\ face) - \mathcal{K}(single\ edge) & \text{for } |face| = 2 \\ 3(\mathcal{K}(single\ face) - \mathcal{K}(single\ edge)) & \text{for } |face| = 3 \end{cases}$$

and $\mathcal{K}(edge) = |edge|\mathcal{K}(single\ edge)$ with $\mathcal{K}(single\ edge) = (k+1)(1+2k)$ and $\mathcal{K}(single\ face) = 2(k+1)^2(1+2k)$, being the costs of the in-place interpolation of a single edge/face. The formulas evaluated for $k = 1, 4$ are shown in Table 1.

On the contrary, the arithmetic cost of the general-purpose algorithm for applying the hanging-node constraints on a single face is $\mathcal{O}(k^4)$. Both the hanging-node algorithm and the general-purpose algorithm have—under the assumption that the 1D interpolation matrices and the compressed constraint matrix are in cache for moderate k —a memory cost of $\mathcal{O}(1)$. As a consequence, the difference in performance is due to a different number of floating-point operations and differences in code generation. For the latter, the proposed specialized algorithm can use the polynomial degree and hence loop lengths as a compile-time constant, whereas the generic implementation can not, which on its own causes a 2x-3x difference in performance [15].

4 Experiments and Results

In this section, we investigate the suitability of the proposed algorithm on modern hardware. As a metric, we use the cost η of a cell that is either edge- or face-constrained. We define the cost as $\eta = (T_{\text{HN}} - T_{\text{NO}})/T_{\text{NO}}$, i.e., the ratio of the additional time to process a cell with hanging-node constraints and the time to process a cell without hanging-node constraints. For this purpose, we use two approaches to determine the value of η independently in Subsects. 4.1 and 4.2. The code of our experiments can be found online.¹

Our experiments perform operator evaluations (also referred to as “matrix-vector product” or “vmult”) of a scalar Laplace operator with homogeneous Dirichlet boundaries on two classes of locally refined 3D meshes. The meshes

¹ <https://github.com/peterrum/dealii-matrixfree-hanging-nodes> with the `deal.II` master branch retrieved on March 26 2022, with small adjustments to disable the automatic choice of the vectorization type by the library.

Table 1. Number of FLOPs for edge (e) and face (f) constraints as well as for interpolation from solution coefficients $u_{e,i}$ to values at quadrature points (cell). The numbers in the header indicate the count of constrained faces or edges. The numbers of FLOPs have been verified with hardware counters.

| k | 1e+0f | 2e+0f | 3e+0f | 0e+1f | 1e+1f | 0e+2f | 0e+3f | cell |
|---|-------|-------|-------|-------|-------|-------|-------|------|
| 1 | 6 | 12 | 18 | 24 | 30 | 42 | 54 | 72 |
| 4 | 45 | 90 | 135 | 450 | 495 | 855 | 1215 | 3375 |

Table 2. Runtime analysis in terms of memory transfer and GFLOP/s, as measured with the Likwid tool for $k = 4$ and affine/non-affine `shell` mesh ($L = 7/6$). Run on Intel Cascade Lake Xeon Gold 6230 (2560 GFLOP/s, 202 GB/s).

| | No constraints | | | General-purpose algo | | | Hanging-node algo | | |
|------------|----------------|------|------|----------------------|------|------|-------------------|------|------|
| | s | GB/s | GF/s | s | GB/s | GF/s | s | GB/s | GF/s |
| Affine | 0.028 | 122 | 437 | 0.057 | 86 | 252 | 0.034 | 107 | 386 |
| Non-affine | 0.016 | 194 | 174 | 0.021 | 162 | 149 | 0.016 | 191 | 183 |

are constructed by refining a coarse mesh consisting of a single cell defined by $[-1, +1]^3$ according to one of the following two solution criteria: 1) `shell`: after $L - 3$ uniform refinements, perform three local refinement steps with all cells whose center \mathbf{c} is $|\mathbf{c}| \leq 0.55$, $0.3 \leq |\mathbf{c}| \leq 0.43$, and $0.335 \leq |\mathbf{c}| \leq 0.39$ or 2) `octant`: refine all mesh cells in the first octant L times. Figure 5 shows, as an example, the resulting meshes. Simulations are run with polynomial degrees $1 \leq k \leq 6$ to cover all cases from low- to high-order FEM.

Unless noted otherwise, the numerical experiments are run on a dual-socket Intel Xeon Platinum 8174 (Skylake) system of the supercomputer SuperMUC-NG.² It supports AVX-512 (8-wide SIMD). The 48 CPU cores run at a fixed frequency of 2.3 GHz, which gives an arithmetic peak of 3.5 TFLOP/s. The 96 GB of random-access memory (RAM) are connected through 12 channels of DDR4-2666 with a theoretical bandwidth of 256 GB/s and an achieved STREAM triad memory throughput of 205 GB/s. We use GCC 9.3.0 as compiler with the flags `-march=skylake-avx512 -std=c++17 -O3`. Furthermore, the CPU code uses the vectorization strategy `auto` by default, computations are run with double-precision floating-point numbers and results are reported in 64-bit FLOPs.

4.1 Experiment 1: Serial Simulation

In the first experiment, we execute the program serially to rule out influences of MPI communication and potential load imbalances. In order to nevertheless obtain a realistic per-core memory bandwidth, we execute an instance of the program on all cores of a compute node simultaneously.

² <https://top500.org/system/179566/>, received on November 15, 2021.

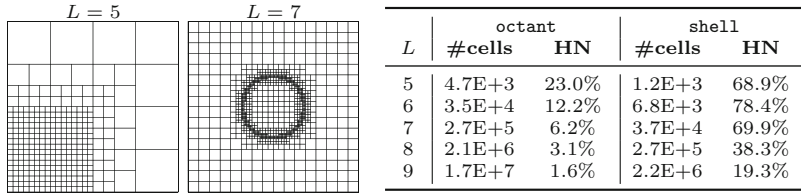


Fig. 5. Cross section of the **octant** geometry (left) and of the **shell** geometry (right) simulation for specified number of refinements. In addition, the number of cells ($\#cells := N_{NO} + N_{HN}$) and the share of cells with hanging-node constraints ($HN := N_{HN} / (N_{NO} + N_{HN})$) are given for the considered refinement numbers.

In the context of such serial experiments, the total simulation time is the sum of the time spent on cells with hanging-node constraints and on cells without hanging-node constraints: $T = N_{HN}T_{HN} + N_{NO}T_{NO} = (N_{NO} + (1 + \eta_1)N_{HN})T_{NO}$. From this formula, we derive an experimental definition of the cost:

$$\eta_1 = (T/T_{NO} - N_{NO})/N_{HN} - 1. \quad (2)$$

The cell counts N_{NO} and N_{HN} are given by the geometry (see Fig. 5), the total simulation time T can be measured, and the time to process a cell without hanging-node constraints T_{NO} can be approximated by running the simulations artificially without hanging-node constraints with runtime \hat{T} , i.e., $T_{NO} \approx \hat{T} / (N_{NO} + N_{HN})$.

Figures 6a and 6b show the throughput of a single operator application (processed number of DoFs per time unit) and the cost η_1 for different degrees k in the **shell** case for the general-purpose algorithm (all constraints, incl. hanging-node constraints, are processed by $\mathcal{C}_e^{\text{GP}}$) and for the specialized hanging-node algorithm. As a reference, the throughput of the simulation without application of any hanging-node constraints is presented. One can observe an overall drop in throughput by 32–63% in the case of the general-purpose algorithm and by 11–34% in the case of the specialized algorithm. This translates into an increase in runtime for evaluating the discrete PDE operator on a cell with hanging nodes by 125–215% and 20–136%, respectively. While the costs are increasing in the case of the general-purpose algorithm, the costs in the specialized case are decreasing to a value of approx. 20%. This difference in behaviors is related to the difference in complexities and to the overhead in the low-degree case $k = 1$.

Furthermore, Figs. 6a and 6b show the results for (high-order) non-affine meshes (dashed lines). In order to deform the analyzed geometries, the transformation function $x_i \leftarrow x_i + \Delta \sin(\pi \cdot x_i)$ with $\Delta = 10^{-8}$ is applied. This transformation makes the matrix-free cell loops memory-bound in the current implementation, since the code loads a 3×3 Jacobian matrix for each quadrature point. In such cases, additional work on cached data can be hidden behind the memory transfer [15], as is verified to be the case for the additional hanging-node interpolations in our simulations: for linear elements, the costs of the proposed implementation are reduced from 136% to 58% and, for all higher degrees, no

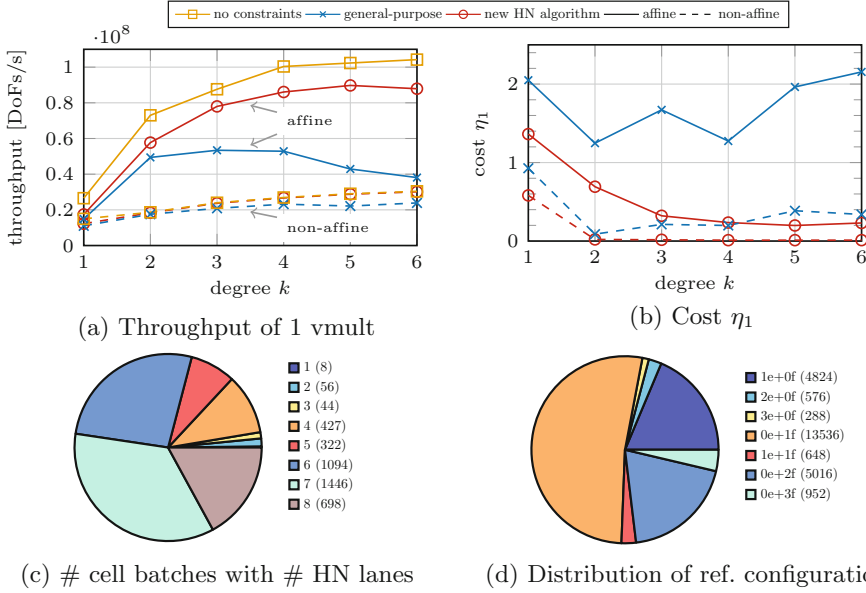


Fig. 6. a–b) Experimental results of the `shell` simulation on the CPU for degrees $1 \leq k \leq 6$ (for $L = 8/8/8/7/7/6$ in the affine case and one less for the non-affine case). c–d) Hanging-node statistics: count of cell batches with the given number of lanes with hanging-node constraints ($L = 7$); distribution of refinement-configuration types (grouped together by the number of edge and face constraints).

overhead can be observed with $\eta_1 \approx 1\%$. In contrast, one can still observe costs of up to 34% in the general-purpose case.

Figure 7a presents the results for the `octant` case. Since the number of hanging nodes is significantly less in this case, the penalty of applying the general-purpose algorithm leads to a throughput reduction of a mere 7–18%, and, in the case of the specialized algorithm, the throughput is comparable to the case without hanging nodes. The fact that the cost η_1 for processing cells with hanging-node constraints is similar to the one in the `shell` case makes us confident that the definition (2) to measure overhead and the experimental results presented here are transferable to other meshes and other refinement configurations.

We also analyzed the algorithm with hardware counters (for a broad overview see Table 2). We could observe an increase in scalar operations both in the case of the special-purpose and the general-purpose algorithm. While, however, the additional scalar operations per DoF decrease with increasing k in the case of the special-purpose algorithm, this is not the case for the general-purpose algorithm. Furthermore, the special-purpose algorithm can be fed with the required data from the fast L1 cache, while the general-purpose algorithm needs to access higher memory levels (incl. main memory) for the entries of the matrix C . In the case of the special-purpose algorithm, the number of branch mispredictions per operation decreases for increasing k .

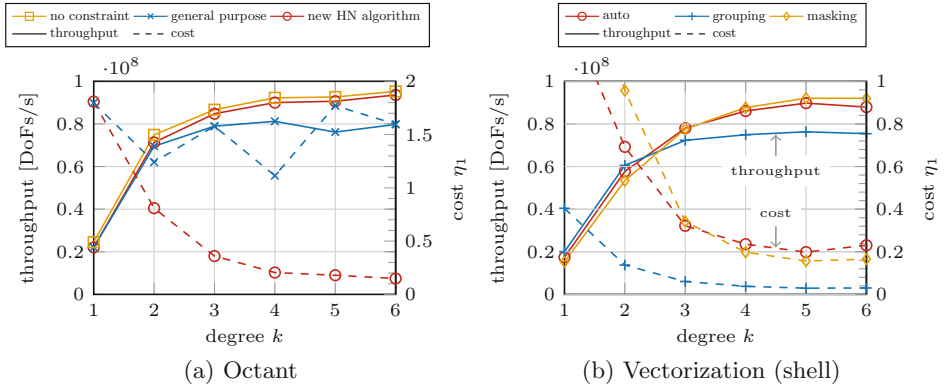


Fig. 7. a) Experimental results of the `octant` simulation for degrees $1 \leq k \leq 6$ (for $L = 7/7/7/6/6/6$). See also the comments in Fig. 6. b) Comparison of experimental results with different vectorization strategies for an affine `shell` mesh. Setup as in Fig. 6.

Figure 7b shows throughput and cost η_1 for the three vectorization strategies presented in Subsect. 3.4 (`auto`, `grouping`, and `masking`) for the `shell` case. One can observe that the strategy `grouping` can significantly reduce the value of the cost η_1 : for degrees $k \geq 2$, it appears as if cells with hanging-node constraints would be similar in cost to regular cells. This is not surprising, since performing one packed operation in contrast to 5–8 scalar operations is significantly cheaper (see Fig. 6c regarding the number of lanes with hanging-node constraints). For low degrees, the reduced costs indeed lead to a (slight) increase in throughput, compared to the (default) `auto` strategy. For higher degrees, the strategy `grouping` reaches a throughput that is 15% lower than the one of the strategy `auto` in the `shell` case. This is related to the fact that the grouping results in discontinuous partitions, which lead—in combination with $\mathcal{O}(k^d)$ working sets of the cells during cell integrals—to a worse cache-locality behavior of the whole matrix-free loop (1) and an increase in cell batches, since a growing number of lanes might not be filled. The vectorization strategy `masking` adds costs by setting up the masks for 18 geometric entities and by increasing the number of conditional branches in contrast to the two switches in `auto`. While, for linear and quadratic elements, these additional costs are dominating, leading to $96\% < \eta_1 < 192\%$, they amortize and higher throughputs than in the case of `auto` are reached for higher degrees ($k \geq 4$). Our experiments (not presented here) have shown that, by switching to single-precision computations and hereby doubling the number of lanes processed by a cell batch, the turning point towards the vectorization strategy `masking` is shifted to lower degrees.

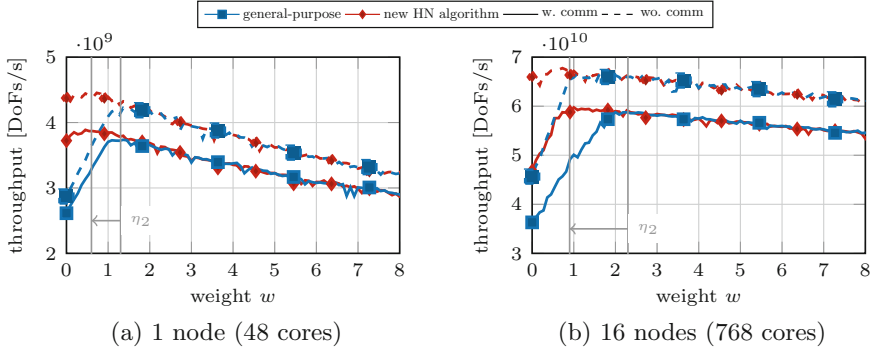


Fig. 8. Time of an operator application with either the general-purpose or the hanging-node algorithm and with communication either enabled or disabled. We used `octant` with $L = 7$ and $L = 9$ number of refinements as geometry and $k = 4$.

4.2 Experiment 2: Parallel Simulation

In this experiment, we distribute the mesh and the work among all processes of a compute node. The time spent on the operator evaluation by process i is proportional to $N_{\text{NO},i} + (1 + \eta)N_{\text{HN},i}$ with $N_{\text{NO},i}$ and $N_{\text{HN},i}$, being the number of cells without and with hanging nodes possessed by that process. The overall time spent by the whole application is $\sim \max_i (N_{\text{NO},i} + (1 + \eta)N_{\text{HN},i})$. If the same number of cells $N_{\text{NO},i} + N_{\text{HN},i}$ is assigned to each process i and additional costs $\eta \gg 1$ are ignored, this leads to load imbalances and to a decreased total throughput. In such situations, one would not distribute the number of cells but the work by assigning each cell a user-specified weight, e.g., the weight of 1.0 to cells without hanging-node constraints and the weight of $w + 1 \geq 1$ to cells with hanging-node constraints. Such weights need to be determined by a tuning process. Small costs are desirable, on the one hand, as the throughput is acceptable without tuning of additional parameters and, on the other hand, because other code regions in the same application might have different—contradicting—requirements for the weights.

The goal of this experiment is to determine a factor η_2 that is given as the weight w for which the execution time is minimal in the `octant` case for $k = 4$. In the optimal case, the value η_2 should be comparable to η_1 from the first experiment.

Figure 8 shows the time of an operator application for different weights w on 1 and 16 compute nodes. As expected, the specialized hanging-node treatment is able to shift η_2 from 130% to 60% and from 230% to 90%. At the same time, it also reaches overall higher throughputs.

Comparing the values η_1 and η_2 , one can observe that $\eta_1 < \eta_2$. Our investigations have revealed that this is related to the communication. The matrix-free operator evaluation (1) updates the ghost values during \mathcal{G} and collects partial results from neighboring processes during a “compress” step in \mathcal{G}^T . The library `deal.II` assigns DoFs to cells/processes in the order of a space-filling curve [4, 5];

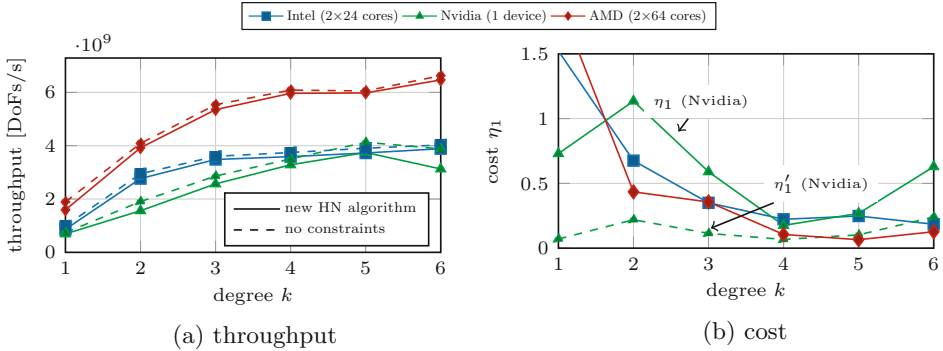


Fig. 9. Experimental results of the `shell` simulation for degrees $1 \leq k \leq 6$.

in particular, all the leaf children of a cell are assigned to the same process. This leads to communication patterns, as indicated in Fig. 2c, in which only the constraining cells need to send data during the update of the ghost values, while during the compression step only constrained cells are sending data. In many cases, constraining and constrained cells are well distributed, but not in the `octant` case. This can be verified by turning off the communication (dashed lines in Fig. 8), for which indeed $\eta_1 \approx \eta_2$. We defer the development of an algorithm for smarter assignment of DoFs to future work.

4.3 Experiment 3: Cross-Platform Validation

In the following, we present the results of parallel experiments for affine `shell` meshes for $1 \leq k \leq 6$ additionally on a dual-socket AMD Epyc 7742 CPU compute node and a single GPU device on Summit³ (Nvidia Tesla V100). The AMD CPU consists of 2×64 cores running at 2.25 GHz and uses codes compiled for the AVX2 instruction-set extension (4-wide SIMD). This gives an arithmetic peak performance of 4.61 TFlop/s. The memory configuration uses 2×8 channels of DDR4-3200, resulting in a peak bandwidth of 410 GB/s and a measured STREAM triad bandwidth of 290 GB/s. The performance specifications of the V100 GPU in terms of GB/s and GFLOP/s are more than twice as high as the ones of the two CPU systems (arithmetic peak performance of 7.8 TFlop/s, peak memory bandwidth of 900 GB/s, and measured bandwidth of 720 GB/s), but with a less sophisticated cache infrastructure. On the AMD CPU, we use `gcc-7.5.0` as compiler with the flags `-O3 -march=znver2 -funroll-loops`, and, on the Nvidia GPU, we use `nvcc 11.0.3/gcc 9.1.0` as compiler with the flags `-O2`. We have chosen the number of refinements to the maximal memory capacity of the given hardware. We did not perform any tuning of the weight parameter w (see Subsect. 4.2) and set its value to zero.

Figure 9a presents the obtained throughput. The AMD system reaches the highest throughput of around 6 GDoFs/s. Nvidia and Intel show similar maximal

³ <https://www.top500.org/system/179397/>, retrieved on November 15, 2021.

throughputs of around 4 GDoFs/s, with Intel having slight advantages at lower polynomial degrees. The lower performance of the Intel hardware setup compared to the AMD setup is mainly related to the different memory bandwidths (205 vs. 290 GB/s).

Figure 9b presents the cost η_1 for the three processor types. All of them start with a high value at low degrees, but reach lower costs (6–25%) for higher degrees ($k \geq 4$). For the Nvidia GPU, we present a second set of results (dashed line) in Fig. 9b. The reason for this is that the hanging-node algorithm is executed—in contrast to our expectations—on every cell, even if its refinement configuration (value “0”) indicates that nothing has to be done (by the warp/block). In such a case, the definition of the cost (2) does not hold and we, therefore, define $\eta'_1 = (T - \hat{T})/\hat{T}$, i.e., as the ratio of the additional time to run a simulation with hanging-node constraints and the time to run the same simulation artificially without hanging-node constraints. The values $6\% \leq \eta'_1 \leq 24\%$ are reasonable, but implicate that simulations with any number of cells with hanging-node constraints have to pay this overall penalty, even if they only have $\approx 1\%$ such cells, as is the situation in the `octant` case.

5 Conclusions and Outlook

We have presented an algorithm for the efficient evaluation of the continuity constraints at hanging nodes for matrix-free high-order FEM computations on unstructured, hex-dominated, mixed meshes and for multi-component elements that contain a Lagrange element in one of their components. The algorithm splits up the application of constraints into a hanging-node part and a general part, using efficient inplace interpolations for the former. For this purpose, the DoF map of the cells has to be updated and the configurations of cell refinements have to be determined as well as efficiently encoded and decoded. In 3D, we require 8 bits to encode all 137 possible configurations. The algorithm is applicable for both CPUs and GPUs with two distinct differences: 1) for the GPU, the application of non-hanging-node constraints, like Dirichlet boundary conditions, can not be merged into the cell loop, but needs to be applied separately and 2) specialized interpolation routines have to be used due to different vectorization strategies.

Experiments have shown that, for high-order finite elements, the costs of cells with hanging-node constraints can be reduced significantly for affine meshes. For low-order elements, we also obtain improvements, but the costs remain noticeable due to conditional branches required for checking the refinement configurations. For high-order non-affine meshes, the application of hanging-node constraints can be completely hidden behind memory access. For the CPU, we have discussed different vectorization strategies and identified that processing cell by cell is the most efficient approach in the context of matrix-free algorithms that are based on vectorization across cells for lower degrees $k \leq 3$, whereas masking is superior for $k > 3$. The benefits of our node-level optimization on parallel applications are significantly reduced load imbalances and higher throughput with a more moderate cell-weighting function.

Future work will extend the algorithm towards the support of more cell shapes (e.g., simplex, wedge, pyramid) in the context of mixed meshes and *hp*-adaptive FEM so that one does not need to fall back to a slower general-purpose algorithm in these cases. Moreover, we intend to perform further performance optimizations, which will target the reduction of overhead in the case of low-order elements, alternative vectorization strategies, and improved parallel distribution of degrees of freedom in order to minimize the communication overhead in the context of hanging-node constraints.

Acknowledgment. The authors acknowledge collaboration with Momme Allalen, Daniel Arndt, Magdalena Schreter, Bruno Turcksin as well as the `deal.II` community.

References

1. Anderson, R., et al.: MFEM: a modular finite element methods library. *Comp. Math. Appl.* **81**, 42–74 (2021)
2. Arndt, D., et al.: The `deal.II` library, version 9.3. *J. Numer. Math.* **29**(3) (2021)
3. Arndt, D., et al.: The `deal.II` finite element library: design, features, and insights. *Comp. Math. Appl.* **81**, 407–422 (2021)
4. Bangerth, W., Burstedde, C., Heister, T., Kronbichler, M.: Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.* **38**(2), 14/1–28 (2011)
5. Burstedde, C., Wilcox, L.C., Ghattas, O.: p4est: scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM J. Sci. Comput.* **33**(3), 1103–1133 (2011)
6. Cervený, J., Dobrev, V., Kolev, T.: Nonconforming mesh refinement for high-order finite elements. *SIAM J. Sci. Comput.* **41**(4), C367–C392 (2019)
7. 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. Num. Meth. Eng.* **121**(13), 2874–2895 (2020)
8. Deville, M.O., Fischer, P.F., Mund, E.H.: High-order methods for incompressible fluid flow. Cambridge University Press (2002)
9. Fischer, P., et al.: Scalability of high-performance PDE solvers. *Int. J. High Perf. Comp. App.* **34**(5), 562–586 (2020)
10. Fischer, P.F., Kruse, G.W., Loth, F.: Spectral element methods for transitional flows in complex geometries. *J. Sci. Comput.* **17**(1), 81–98 (2002)
11. 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. Comp. Phy.* **348**, 634–659 (2017)
12. Kronbichler, M., Diagne, A., Holmgren, H.: A fast massively parallel two-phase flow solver for microfluidic chip simulation. *Int. J. High Perform. Comput. Appl.* **32**(2), 266–287 (2018)
13. Kronbichler, M., et al.: A next-generation discontinuous Galerkin fluid dynamics solver with application to high-resolution lung airflow simulations. In: *SC 2021* (2021)
14. Kronbichler, M., Kormann, K.: A generic interface for parallel cell-based finite element operator application. *Comput. Fluids* **63**, 135–147 (2012)

15. Kronbichler, M., Kormann, K.: Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM Trans. Math. Softw.* **45**(3), 29/1-40 (2019)
16. Kronbichler, M., Ljungkvist, K.: Multigrid for matrix-free high-order finite element computations on graphics processors. *ACM Trans. Parallel Comput.* **6**(1), 2/1-32 (2019)
17. Laughton, E., Tabor, G., Moxey, D.: A comparison of interpolation techniques for non-conformal high-order discontinuous Galerkin methods. *Comput. Methods Appl. Mech. Eng.* **381**, 113820 (2021)
18. Ljungkvist, K.: Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes. In: *SpringSim (HPC)*, pp. 1–1 (2017)
19. Melenk, J.M., Gerdes, K., Schwab, C.: Fully discrete hp-finite elements: fast quadrature. *Comput. Methods Appl. Mech. Eng.* **190**(32), 4339–4364 (2001)
20. Munch, P., Heister, T., Prieto Saavedra, L., Kronbichler, M.: Efficient distributed matrix-free multigrid methods on locally refined meshes for FEM computations. *arXiv preprint [arXiv:2203.12292](https://arxiv.org/abs/2203.12292)* (2022)
21. Munch, P., Kormann, K., Kronbichler, M.: hyper.deal: an efficient, matrix-free finite-element library for high-dimensional partial differential equations. *ACM Trans. Math. Softw.* **47**(4), 33/1–34 (2021)
22. Orszag, S.A.: Spectral methods for problems in complex geometries. *Journal of Computational Physics* **37**(1), 70–92 (1980)
23. Saurabh, K., et al.: Scalable adaptive PDE solvers in arbitrary domains. In: *SC 2021* (2021)
24. Scroggs, M.W., Dokken, J.S., Richardson, C.N., Wells, G.N.: Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes. *ACM Trans. Math. Softw.* (2022). <https://doi.org/10.1145/3524456>
25. Shephard, M.S.: Linear multipoint constraints applied via transformation as part of a direct stiffness assembly process. *Int. J. Num. Meth. Eng.* **20**(11), 2107–2112 (1984)
26. 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)

Paper III

Efficient distributed matrix-free multigrid methods on locally refined meshes for FEM computations

PETER MUNCH, Institute of Mathematics, University of Augsburg, Germany, Institute of Material Systems Modeling, Helmholtz-Zentrum Hereon, Germany, and Institute for Computational Mechanics, Technical University of Munich, Germany

TIMO HEISTER, Mathematical and Statistical Sciences, Clemson University, USA

LAURA PRIETO SAAVEDRA, Department of Chemical Engineering, Polytechnique Montréal, Canada

MARTIN KRONBICHLER, Institute of Mathematics, University of Augsburg and Department of Information Technology, Uppsala University, Sweden

This work studies three multigrid variants for matrix-free finite-element computations on locally refined meshes: geometric local smoothing, geometric global coarsening (both h -multigrid), and polynomial global coarsening (a variant of p -multigrid). We have integrated the algorithms into the same framework—the open-source finite-element library `deal.II`—, which allows us to make fair comparisons regarding their implementation complexity, computational efficiency, and parallel scalability as well as to compare the measurements with theoretically derived performance metrics. Serial simulations and parallel weak and strong scaling on up to 147,456 CPU cores on 3,072 compute nodes are presented. The results obtained indicate that global-coarsening algorithms show a better parallel behavior for comparable smoothers due to the better load balance, particularly on the expensive fine levels. In the serial case, the costs of applying hanging-node constraints might be significant, leading to advantages of local smoothing, even though the number of solver iterations needed is slightly higher. When using p - and h -multigrid in sequence (hp -multigrid), the results indicate that it makes sense to decrease the degree of the elements first from a performance point of view due to the cheaper transfer.

CCS Concepts: • **Mathematics of computing** → **Solvers; Mathematical software performance;** • **Computer systems organization** → **Multicore architectures.**

Additional Key Words and Phrases: multigrid, finite-element computations, linear solvers, matrix-free methods

ACM Reference Format:

Peter Munch, Timo Heister, Laura Prieto Saavedra, and Martin Kronbichler. 2022. Efficient distributed matrix-free multigrid methods on locally refined meshes for FEM computations. *ACM Trans. Parallel Comput.* 1, 1, Article 111 (April 2022), 39 pages. <https://doi.org/10.1145/1122445.1122456>

Authors' addresses: Peter Munch, peter.muench@uni-a.de, Institute of Mathematics, University of Augsburg, Universitätsstraße 12a, 86159 Augsburg, Germany, Institute of Material Systems Modeling, Helmholtz-Zentrum Hereon, Max-Planck-Str. 1, 21502 Geesthacht, Germany, Institute for Computational Mechanics, Technical University of Munich, Boltzmannstr. 15, 85748 Garching b. München, Germany; Timo Heister, heister@clemson.edu, Mathematical and Statistical Sciences, Clemson University, O-110 Martin Hall, Clemson, SC, USA; Laura Prieto Saavedra, laura.prieto-saavedra@polymtl.ca, Department of Chemical Engineering, Polytechnique Montréal, PO Box 6079, Stn Centre-Ville, H3C 3A7, Montreal, QC, Canada; Martin Kronbichler, martin.kronbichler@uni-a.de, Institute of Mathematics, University of Augsburg, Department of Information Technology, Uppsala University, Box 337, 75105 Uppsala, Sweden.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.

1539-9087/2022/4-ART111 \$15.00

<https://doi.org/10.1145/1122445.1122456>

1 INTRODUCTION

Many solvers for finite element methods (FEM) rely on efficient solution methods for second-order partial differential equations (PDEs), e.g., for the Poisson equation:

$$-\Delta u = f, \quad (1)$$

where u is the solution variable and f is the source term. Poisson-like problems also frequently occur as subproblems, e.g., in computational fluid dynamics [10, 30, 59] or in computational plasma physics [80]. Efficient realizations often rely on adaptively refined meshes to resolve geometries or features in the solution.

Multigrid methods are among the most competitive solvers for such problems [36]. The three basic steps of a two-level algorithm are 1) presmoothing, in which the high-frequency error components in the initial guess are removed with a *smoother*, 2) coarse-grid correction, in which a related problem on a coarser grid is solved, requiring *intergrid transfer operators* and a *coarse-grid solver*, and 3) postsmoothing, in which the high-frequency error components introduced during interpolation are removed. Nesting two-level algorithms recursively gives a multigrid algorithm. In library implementations, these steps are generally expressed as operators. The operations can be generally chosen and/or configured by the user and strongly depend on the multigrid variant selected. This publication discusses massively parallel multigrid variants for locally refined meshes, including the efficient implementation of their operators, compares them with each other, and gives recommendations regarding optimal configurations.

1.1 Multigrid variants

Which multigrid approach to choose depends on the way the mesh is generated and on the underlying finite-element space. If the mesh is generated by globally refining each cell of a coarse grid recursively, it is a natural choice to apply geometric multigrid (abbreviated here as h -multigrid), which uses the levels of the resulting mesh hierarchy as multigrid levels. Alternatively, in the context of high-order finite elements, it is possible to create levels by reducing the polynomial order of the shape functions p of the elements, while keeping the mesh the same, as done by polynomial multigrid (abbreviated as p -multigrid). For a very fine, unstructured mesh with low-order elements, it is not as trivial to explicitly construct enough multigrid levels and one might need to fall back to non-nested multilevel algorithms [1, 22, 23] or algebraic multigrid (AMG; see the review by Stüben [97]). These basic multigrid strategies can be nested in hybrid multigrid solvers [32, 71, 84, 85, 89, 93, 98] and, in doing so, one can exploit the advantages of all of them regarding robustness. Most common are hp -multigrid, which combines h - and p -multigrid, and AMG as black-box coarse-grid solver of geometric or polynomial multigrid solvers.

All the above-mentioned multigrid variants are applicable to locally refined meshes. However, local refinement comes with additional options (local vs. global definition of the multigrid levels, resulting in local(-smoothing) and global(-coarsening) versions of geometric/polynomial multigrid) and with additional challenges, which are particularly connected with the presence of hanging-node constraints, as indicated in Figure 1.

1.2 Related work

Some authors of this study have been involved in previous contributions to the research field of multigrid methods. Their implementations are used and extended in this work.

In [27, 28, 64, 66], matrix-free and parallel implementations of geometric local-smoothing algorithms from the deal . II finite-element library were investigated and compared with AMG, demonstrating the benefits of using matrix-free implementations for both CPU and GPU, but also non-optimal scalability due to load imbalance for simple partitioning strategies. Local-smoothing

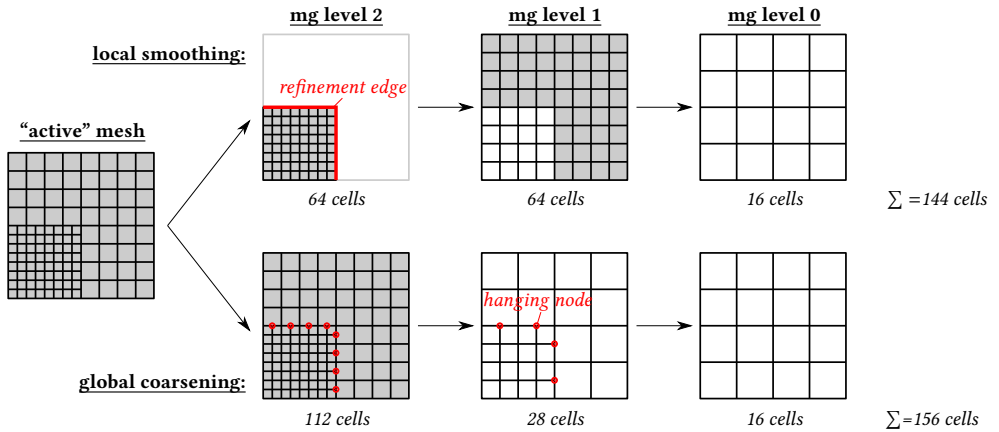


Fig. 1. Visual comparison of geometric multigrid methods for locally refined meshes. Top: (geometric) local smoothing; bottom: (geometric) global coarsening. Local smoothing only considers cells strictly on the same refinement level. This typically introduces an internal boundary (at the refinement edge) when the cells do not cover the whole computational domain. Only if they do (here, for level 1 and 0, not for level 2), one can switch to a coarse-grid solver. Instead, global coarsening considers the whole domain and typically introduces hanging nodes on the multigrid levels. Global coarsening tends to involve work on more cells in total compared to local-smoothing algorithms, but often reduces the number of cells per multigrid level quicker on the finer levels. The gray shading indicates active cells.

algorithms have been a core module of deal . II for many years, however, they gained much maturity, flexibility, and performance due to the above-mentioned thorough studies. In [66], their performance was compared to a state-of-the-art implementation of a Poisson solver from the literature [36]: a speedup of 30% for quadratic elements and a speedup of a factor of 8 for fourth-degree elements on comparable hardware underline the high node-level performance of the code.

Independently of local smoothing, an efficient hybrid multigrid solver for discontinuous Galerkin methods (DG) for globally refined meshes was developed and presented in [32]. It is part of the deal . II-based incompressible Navier–Stokes solver ExaDG [10] and relies on auxiliary-space approximation [5], i.e., on the transfer into a continuous space, as well as on subsequent execution of p -multigrid, h -multigrid, and AMG. That solver was extended to leverage locally refined meshes, using a geometric/polynomial global-coarsening algorithm after the transfer into the continuous space, in order to simulate the flow through a lung geometry [60]. Its functionalities have been generalized and are the foundations of the new global-coarsening implementation in deal . II [7, 9], targeting both h - and p -multigrid, in addition to the established geometric local-smoothing implementation.

1.3 Our contribution

In this publication, we consider three well-known multigrid algorithms for locally refined meshes for continuous higher-order matrix-free FEM: geometric local smoothing, geometric global coarsening (both h -multigrid), and polynomial global coarsening (a variant of p -multigrid). We have implemented them into the same framework, which allows us to compare their implementation complexity and performance for a large variety of problem sizes. This has not been done in an extensive way in the literature, often using only one of them [28, 99]. Furthermore, we rely on matrix-free

Algorithm 1: Multigrid V-cycle called recursively on each level l to solve $\mathbf{Ax} = \mathbf{b}$. It operates on vectors $[\mathbf{b}^{(0)}, \dots, \mathbf{b}^{(L)}]$ and $[\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(L)}]$, which are filled/read on the finest level L , and uses the level operators $\mathbf{A}^{(l)}$, smoothers, intergrid operators, and coarse-grid solvers. In the case of local smoothing, we distinguish between interior DoFs (not labeled specially) and DoFs on the internal boundaries ($\mathbf{x}_E^{(l)}$) as well as decompose the level operator $\mathbf{A}^{(l)}$ into $\mathbf{A}_{SS}^{(l)}$, $\mathbf{A}_{SE}^{(l)}$, $\mathbf{A}_{ES}^{(l)}$, and $\mathbf{A}_{EE}^{(l)}$ (see the explanation in Subsection 2.1). For global coarsening, $\mathbf{A}^{(l)} = \mathbf{A}_{SS}^{(l)}$. The arrows in braces on the right indicate the type of communication at each step: (\Downarrow) involves vertical communication between levels; (\Leftrightarrow) involves horizontal communication on the same level. Additional steps needed by local smoothing are highlighted in gray.

```

1 if  $l = L$  then
2   |  $[b^{(0)}, \dots, b^{(L)}] \leftarrow \mathbf{b};$                                /* copy to multigrid level(s) */
3 if  $l = 0$  then
4   |  $\mathbf{x}^{(0)} \leftarrow \text{CoarseGridSolver}(\mathbf{A}^{(0)}, b^{(0)});$          /* coarse-grid solver:  $\mathbf{A}^{(0)} \stackrel{!}{=} \mathbf{A}_{SS}^{(0)}$  ( $\Leftrightarrow/\Downarrow$ ) */
5 else
6   |  $\mathbf{x}^{(l)} \leftarrow \text{Smoother}(\mathbf{A}_{SS}^{(l)}, \mathbf{0}, 0, \mathbf{b}^{(l)});$            /* presmoothing with  $\mathbf{x}_E^{(l)} = \mathbf{0}$  ( $\Leftrightarrow$ ) */
7   |  $(\mathbf{r}^{(l)}, \mathbf{r}_E^{(l)}) \leftarrow (\mathbf{b}^{(l)} - \mathbf{A}_{SS}^{(l)}\mathbf{x}^{(l)}, -\mathbf{A}_{ES}^{(l)}\mathbf{x}^{(l)});$  /* compute residual ( $\Leftrightarrow$ ) */
8   |  $\mathbf{b}^{(l-1)} \leftarrow b^{(l-1)} + \text{Restrictor}(\mathbf{r}^{(l)}, \mathbf{r}_E^{(l)});$  /* restrict residual ( $\Downarrow$ ) */
9   |  $\text{VCycleLevel}(l-1);$                                            /* recursion */
10  |  $(\mathbf{x}^{(l)}, \mathbf{x}_E^{(l)}) \leftarrow \mathbf{x}^{(l)} + \text{Prolongator}(\mathbf{x}^{(l-1)});$  /* prolongation ( $\Downarrow$ ) */
11  |  $\mathbf{x}^{(l)} \leftarrow \text{Smoother}(\mathbf{A}_{SS}^{(l)}, \mathbf{x}^{(l)}, \mathbf{x}_E^{(l)}, \mathbf{b}^{(l)});$  /* postsmoothing with  $\mathbf{x}_E^{(l)} \neq \mathbf{0}$  ( $\Leftrightarrow$ ) */
12 if  $l = L$  then
13  |  $\mathbf{x} \leftarrow [\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(L)}];$                          /* copy from multigrid level(s) */

```

operator evaluation, which provides optimal, state-of-the-art implementations in terms of node-level performance on modern hardware [66], and hence exercise the methods in a challenging context in terms of communication costs and workload differences.

The algorithms presented in this publication have been integrated into the open-source finite-element library deal . II [8] and are part of its 9.4 release [9]. All experimental results have been obtained with small benchmark programs leveraging on the infrastructure of deal . II. The programs are available on GitHub under <https://github.com/peterrum/dealii-multigrid>.

The results obtained in this publication for continuous FEM are transferable to the DG case, where one does not have to consider hanging-node constraints but fluxes between differently refined cells. In the case of auxiliary-space approximation [60], this difference only involves the finest level and the rest of the multigrid algorithm could be as described in this publication.

The remainder of this work is organized as follows. In Section 2, we give a short overview of multigrid variants applicable to locally refined meshes. Section 3 presents implementation details of our solver, and Section 4 discusses relevant performance metrics. Sections 5 and 6 demonstrate performance results for geometric multigrid and polynomial multigrid, and, in Section 7, the solver is applied to a challenging Stokes problem. Finally, Section 8 summarizes our conclusions and points to further research directions.

2 MULTIGRID METHODS FOR LOCALLY REFINED MESHES

Algorithm 1 presents the basic multigrid algorithm to solve an equation system of the form $\mathbf{Ax} = \mathbf{b}$ arising from the FEM discretization of (1) (\mathbf{A} is the system matrix, \mathbf{b} is the right-hand-side vector containing the source term f and the boundary conditions, and \mathbf{x} is the solution vector). In the first step, data is transferred to the multigrid levels, after which a multigrid cycle (in this study,

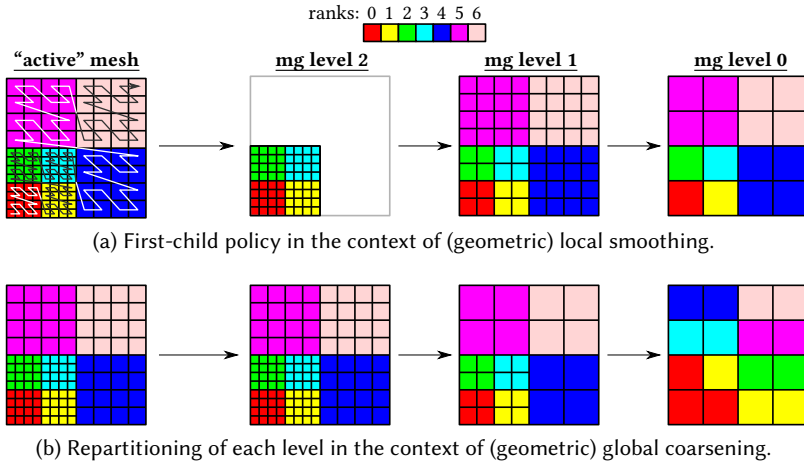


Fig. 2. Visual comparison of two possible strategies for partitioning of the multigrid levels of the mesh shown in Fig. 1 for 7 processes. Colors indicate the ranks of the subdomains. The active level is partitioned uniformly along a space-filling curve, according to a sum of weights for each cell. If a color does not appear (e.g., ranks 4-6 on level 2 in (a)), it means that a process does not have any work on that level, leading to load imbalance during smoothing. If subdomains with the same color do not overlap (e.g., ranks 1-5 on level 0 and 1 in (b)), this leads to more expensive intergrid transfer operators.

a V-cycle with the steps: presmoothing, computation of the residual, restriction, solution on the coarser grid, prolongation, and postsmoothing) is performed. Then, the result is copied back from the multigrid levels. The steps to copy data from and to the multigrid levels are not strictly needed in all cases; however, these steps are required by local smoothing and can be used to switch from double to single precision in order to reduce the costs of multigrid if it is used as a mixed-precision preconditioner [64]. The multigrid algorithm is complemented with the algorithms of a smoother (e.g., Chebyshev smoother [2]) and of a coarse-grid solver once the recursion is terminated. Instead of using multigrid as a solver, we choose to precondition a conjugate-gradient solver [45] with one multigrid cycle per iteration as this is often more robust. These algorithms are not presented here.

The parallelization of multigrid, e.g., based on domain decomposition (see Fig. 2), involves the parallelization of each step of Algorithm 1. Here, two types of communication patterns arise: 1) *horizontal communication* on the same multigrid level during smoothing and residual evaluation, which involves point-to-point ghost-value data exchange between neighboring subdomains and 2) *vertical communication* between multigrid levels during intergrid transfer. In the best case, vertical communication is also a point-to-point communication of some “lower-dimensional” (surface-induced) share of a vector, but it can imply the transfer of essentially the full vector if the levels are partitioned independently and/or the numbers of processes on the levels are reduced extremely (e.g., to one). The communication pattern within the coarse-grid solver depends on the solver chosen and oftentimes also involves vertical and horizontal communication. Global reductions are also common on the coarse grid. We would like to point out that the outer (conjugate-gradient) solver also involves global reductions to compute, e.g., the residual norm once per iteration, needed to determine termination. However, these global reductions are generally negligible due to the dominating costs of multigrid.

The various multigrid algorithms for locally refined meshes differ in the construction of the levels and the concrete details in the implementation of the multigrid steps. We consider two types of geometric multigrid methods: geometric local smoothing in Subsection 2.1 and geometric global coarsening in Subsection 2.2. Figure 1 gives a visual comparison of them and points out the issues resulting from the local or global definition of the levels, which will be discussed extensively in the following. Furthermore, we will detail polynomial global coarsening in Subsection 2.3. In the case of these multigrid variants, the level operator $\mathbf{A}^{(c)}$ can be obtained either recursively via the Galerkin operator $\mathbf{A}^{(c)} := \mathbf{R}^{(c,f)} \mathbf{A}^{(f)} \mathbf{P}^{(f,c)}$ with restriction matrix $\mathbf{R}^{(c,f)}$ and prolongation matrix $\mathbf{P}^{(f,c)}$ constructed geometrically or by rediscrretization. While rediscrretization is not optimal in all cases, it is sufficient for the present experiments, allowing us to benefit from the fact that level operators are independent of each other as, for matrix-free computations, one wants to construct neither $\mathbf{A}^{(c)}/\mathbf{A}^{(f)}$ nor $\mathbf{R}^{(c,f)}/\mathbf{P}^{(f,c)}$ explicitly.

AMG can be used for the solution on locally refined meshes as well. Since the levels are constructed recursively via the Galerkin operator with the restriction and prolongation matrices constructed algebraically, no distinction regarding the local or global definition of the levels is possible. Since we use AMG in the following only as a coarse-grid solver, we refer to the literature for more details: Clevenger et al. [28] present a scaling comparison between AMG and a matrix-free version of local smoothing for a Laplace problem with \mathbf{Q}_2 basis functions, showing a clear advantage of matrix-free multigrid methods on modern computing systems.

2.1 Geometric local smoothing

Geometric local-smoothing algorithms [19, 24, 28, 51, 53, 54, 70, 77, 90, 96] use the refinement hierarchy also for multigrid levels and perform smoothing refinement level by refinement level: cells of less refined parts of the mesh are skipped (see the top part of Figure 1) so that hanging-node constraints do not need to be considered during smoothing. Authors in [3, 20, 50, 77, 100, 102] investigate a version of local smoothing in which smoothing is also performed on a halo of a single coarse cell, which includes hanging-node constraints. We will not consider this form of local smoothing in the following.

The fact that domains on each level might not cover the whole computational domain results in multiple issues. Data needs to be transferred in lines 2 and 13 in Algorithm 1 to and from *all* multigrid levels that are active, i.e., have cells that are not refined further. Moreover, internal interfaces (also known as “refinement edges”), can appear with the need for special treatment. For details, interested readers are referred to [51, 64]. In the following, we summarize key aspects relevant for our investigations.

For the purpose of explanation, let us split the degrees of freedom (DoFs) associated with the cells on an arbitrary level l into the interior ones $\mathbf{x}_S^{(l)}$ and the ones at the refinement edges $\mathbf{x}_E^{(l)}$, leading to the following block structure in the associated matrix system $\mathbf{A}^{(l)} \mathbf{x}^{(l)} = \mathbf{b}^{(l)}$:

$$\begin{pmatrix} \mathbf{A}_{SS}^{(l)} & \mathbf{A}_{SE}^{(l)} \\ \mathbf{A}_{ES}^{(l)} & \mathbf{A}_{EE}^{(l)} \end{pmatrix} \begin{pmatrix} \mathbf{x}_S^{(l)} \\ \mathbf{x}_E^{(l)} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_S^{(l)} \\ \mathbf{b}_E^{(l)} \end{pmatrix}.$$

For presmoothing on level l , homogeneous Dirichlet boundary conditions are applied at the refinement edges ($\mathbf{x}_E^{(l)} = \mathbf{0}$), allowing to skip the coupling matrices. However, when switching to a coarser or finer level, the coupling matrices need to be considered. The residual to be restricted

becomes:

$$\begin{pmatrix} \mathbf{r}_S^{(l)} \\ \mathbf{r}_E^{(l)} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_S^{(l)} \\ \mathbf{b}_E^{(l)} \end{pmatrix} - \begin{pmatrix} \mathbf{A}_{SS}^{(l)} & \mathbf{A}_{SE}^{(l)} \\ \mathbf{A}_{ES} & \mathbf{A}_{EE}^{(l)} \end{pmatrix} \begin{pmatrix} \mathbf{x}_S^{(l)} \\ \mathbf{x}_E^{(l)} \end{pmatrix} \stackrel{\mathbf{x}_E^{(l)}=0}{=} \begin{pmatrix} \mathbf{0} \\ \mathbf{b}_E^{(l)} \end{pmatrix} + \underbrace{\begin{pmatrix} \mathbf{b}_S^{(l)} \\ \mathbf{0} \end{pmatrix} - \begin{pmatrix} \mathbf{A}_{SS}^{(l)} \\ \mathbf{A}_{ES}^{(l)} \end{pmatrix} \mathbf{x}_S^{(l)}}_*. \quad (2)$$

Since $\mathbf{b}_E^{(l)}$ has already been transferred to the coarser level by line 2 in Algorithm 1, one only has to restrict and add the result of term $*$ to the coarser level. During postsmoothing, an inhomogeneous Dirichlet boundary condition is applied with boundary values prescribed by the coarser level. This can be achieved, e.g., by modifying the right-hand side ($\mathbf{b}_S^{(l)} \leftarrow \mathbf{b}_S^{(l)} - \mathbf{A}_{SE}^{(l)} \mathbf{x}_E^{(l)}$).

A natural choice to partition the multigrid levels for local-smoothing algorithms is to partition the active level and let cells on lower refinement levels inherit the rank of their children. A simple variant of it is the “first-child policy” (see Fig. 2a or [28]): it recursively assigns the parent cell the rank of its first child cell. Since in adaptive FEM codes the parents of locally owned and ghost cells are generally already available on processes due to tree-like data-structure storage of adaptively refined meshes [15, 26], no additional data structures need to be constructed and saved, but the storage of an additional flag (multigrid rank of the cell) is enough, leading to low memory consumption. Furthermore, intergrid transfer operations are potentially cheap as data is mainly transferred locally. A disadvantage—besides of having to consider the edge constraints—is the potential load imbalance on the levels, as discussed in [28]. This load imbalance could be alleviated by partitioning each level for itself. Such alternative partitioning algorithm would lead to similar problems as in the case of global-coarsening algorithms (discussed next) and, as a result, the needed data structures would become more complex. This would counter the claimed simplicity of the data structures of this method; hence, we will consider geometric local smoothing only with “first-child policy” in this publication.

Furthermore, the fact that the transfer to and from the multigrid levels involves all active levels prevents an early switch to a coarse-grid solver on levels finer than those that are indeed not locally refined anymore, i.e., $\mathbf{A}_{SS}^{(l)} = \mathbf{A}^{(l)}$. On the other hand, the absence of hanging nodes allows the usage of smoothers that have been developed for uniformly refined meshes, e.g., patch smoothers [11, 55], which are superior for anisotropic meshes.

Since geometric local smoothing is the only local-smoothing approach we will consider here, we will call it simply—as common in the literature—*local smoothing* in the following.

2.2 Geometric global coarsening

Geometric global-coarsening algorithms [20, 21] coarsen all cells simultaneously, translating to meshes with hanging nodes also on coarser levels of the multigrid hierarchy (see the bottom part of Figure 1). The computational complexity—i.e., the total number of cells to be processed—is slightly higher than in the case of local smoothing and might be non-optimal for some extreme examples of meshes [19, 54].

The fact that all levels cover the whole computational domain has the advantage that no internal interfaces need to be considered and the transfer to/from the multigrid levels becomes a simple copy operation to/from the finest level. However, hanging nodes have to be considered during the application of the smoothers on the levels. While this typically does not require new algorithms for basic smoothers as the infrastructure to support adaptive meshes can be simply applied to coarser representations, the operator evaluation and the applicable smoothers might become more expensive per cell by the need to resolve hanging-node constraints [81]. On the other hand, global-coarsening approaches show—for comparable smoothers—a better convergence behavior, which improves with the number of smoothing iterations [13, 14].

As the work on the levels generally increases compared to local smoothing, it is a valid option to repartition each level separately (see Fig. 2b). On the one hand, this implies a higher pressure on the transfer operators, since they need to transfer data between independent meshes,¹ requiring potentially complex internal data structures, which describe the connectivities, and involved setup routines.² On the other hand, it opens the possibility to control the load balance between processes and the minimal granularity of work per process (by removing processes on the coarse level in a controlled way, allowing to switch to subcommunicators [99]) as well as to apply a coarse-grid solver on any level. For the same reason, the construction of full multigrid solvers (FMG), which visit the finest level only a few times, is easier, since any level can be used as a starting point for the recursion. Not selecting the actual coarsest grid but some more refined level allows to relax the unfavorable complexity of FMG in terms of $O(L^2)$ by reducing the time spent on the coarse levels.

2.3 Polynomial global coarsening

Polynomial global-coarsening algorithms [12, 17, 18, 25, 29, 33, 35, 37, 38, 40, 41, 43, 44, 46, 48, 49, 52, 67, 72–76, 83, 85, 87, 88, 91, 94, 95, 99] are based on keeping the mesh size h constant on all levels, but reducing the polynomial degree p of shape functions, e.g., to $p = 1$. Hence, the multigrid levels in this case have the same mesh but different polynomial orders. There are various strategies to reduce the order of the polynomial degree [32, 42]: the most common is the bisection strategy, which repeatedly halves the degree $p^{(c)} = \lfloor p^{(f)}/2 \rfloor$. This strategy reduces the number of DoFs in the case of a globally refined mesh similarly to the geometric multigrid strategies and provides a compromise between the number of V-cycles necessary to reduce the residual norm below a desired threshold and the cost of a single V-cycle [32].

The statements made in Section 2.2 about geometric global coarsening are also valid for polynomial global coarsening. However, the number of unknowns is reduced uniformly on each subdomain in contrast to geometric global coarsening, which obviates repartitioning of the level grids. This leads to a transfer operation that mainly works on locally owned DoFs.

In the following, we call geometric global coarsening simply *global coarsening* and polynomial global coarsening *polynomial coarsening*. As a reference, polynomial local smoothing arises, e.g., in the context of p - or hp -refinement, where only cells with the finest degree are smoothed [79].

3 IMPLEMENTATION DETAILS

In this section, we detail efficient implementations of the multigrid ingredients for locally refined meshes used by Algorithm 1. We start with the handling of constraints. Then, we proceed with the matrix-free evaluation of operator A , which is needed on the active and the multigrid levels, as well as with smoothers and coarse-grid solvers. This section is concluded with an algorithmic framework to efficiently realize all considered kinds of matrix-free transfer operators.

3.1 Handling constraints

Constraints need to be considered—with slight differences—in the case of both local-smoothing and global-coarsening algorithms. First, we impose Dirichlet boundary conditions in a strong form and express them as constraints. Secondly, hanging-node constraints, which force the solution of the refined side to match the polynomial expansion on the coarse side, need to be considered to maintain H^1 regularity of the tentative solution [92]. In a general way, these constraints can

¹In deal. II, one needs to create a sequence of grids for global coarsening (each with its own hierarchical description). This is generally acceptable, since repartitioning of each level often leads to non-overlapping trees so that a single data structure containing all geometric multigrid levels would have little benefit for reducing memory consumption.

²Sundar et al. [99] present a two-step setup routine: the original fine mesh is coarsened and the resulting “surrogate mesh” is repartitioned. For space-filling-curve-based partitioning, this approach turns out to be highly efficient.

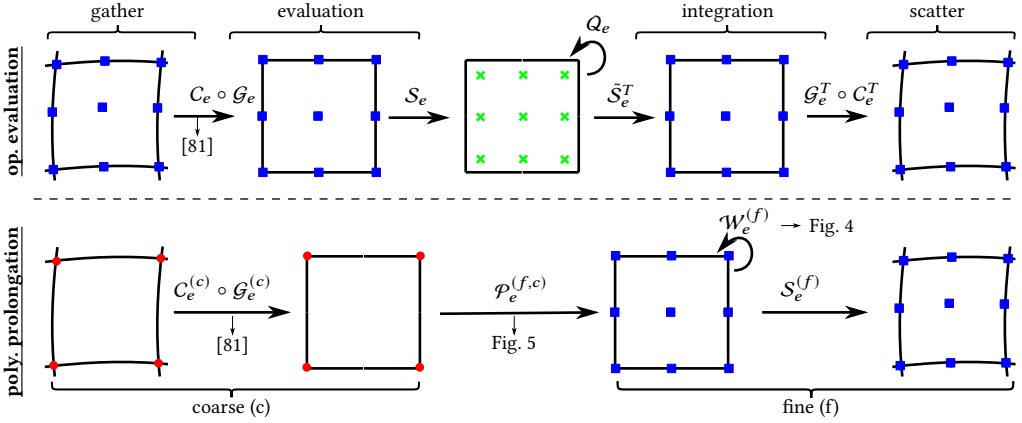


Fig. 3. Basic steps of a matrix-free operator evaluation according to (3) and of a matrix-free polynomial prolongation according to (8) for a single cell e .

be expressed as $x_i = \sum_j c_{ij} x_j + b_i$, where x_i is a constrained DoF, x_j a constraining DoF, c_{ij} the coefficient relating the DoFs, and b_i a real value, which can be used to consider inhomogeneities. We do not eliminate constraints, but use a condensation approach [16, 101].

3.2 Matrix-free operator evaluation

Instead of assembling the system matrix A and performing matrix-vector multiplications of the form Ax , the matrix-free operator evaluation computes the underlying finite-element integrals to represent $\mathcal{A}(x)$. This is motivated by the fact that many iterative solvers and smoothers do not need the matrix A explicitly, but only its action on a vector. On a high level, matrix-free operator evaluation can be derived in two steps: 1) loop switching $v = Au = (\sum_e R_e^T A_e R_e) u = \sum_e R_e^T (A_e (R_e u)) = \sum_e R_e^T (A_e u_e)$, i.e., replacing the sequence “assembly–mat-vec” by a loop over cells with the three steps “gathering–application of the element stiffness matrix–scattering”, and 2) exploitation of the structure of the element stiffness matrix. The element stiffness matrix, e.g., of a Laplace operator, is given by $A_{ij}^{(e)} = \sum_q (\nabla_\xi N_{iq}, (J_q^{-1} |J_q| J_q^{-T} w_q) \nabla_\xi N_{jq})_{\Omega_e}$. Exploiting again the fact that the action of this local matrix on a vector is needed enables to interpret Nu_e as the interpolation of values associated with polynomial basis functions u_e to quantities at the quadrature points, which implies that $\nabla_\xi Nu_e$ computes the gradients at these points. The expression $(J_q^{-1} |J_q| J_q^{-T} w_q)$ is then simply a factor applied to the derived quantities at the quadrature points. The final structure of a matrix-free operator evaluation in the context of continuous finite elements in operator form is:

$$v = \mathcal{A}(x) = \sum_e \mathcal{G}_e^T \circ C_e^T \circ \tilde{S}_e^T \circ Q_e \circ S_e \circ C_e \circ \mathcal{G}_e x. \quad (3)$$

This structure is depicted in Figure 3. For each cell e , cell-relevant values are gathered with operator \mathcal{G}_e , constraints are—as discussed in Subsection 3.1—applied with C_e , and values, gradients, or Hessians are computed at the quadrature points with S_e . These quantities are processed by a quadrature-point operation Q_e ; the result is integrated and summed into the result vector v by applying \tilde{S}_e^T , C_e^T , and \mathcal{G}_e^T . In this publication, we consider symmetric (self-adjoint) PDE operators with $\tilde{S}_e = S_e$.

In the literature, specialized implementations for GPUs [4, 57, 64, 68, 69] and CPUs [4, 62, 63, 80, 82] for operations as expressed in (3) have been presented. For tensor-product (quadrilateral and hexahedral) elements, a technique known as sum factorization [78, 86] is often applied, which allows to replace full interpolations from the local solution values to the quadrature points by a sequence of 1D steps. In the context of CPUs, it is an option to vectorize across elements [62, 63], i.e., perform $(S^T \circ Q \circ S)_e$ for multiple cells in different lanes of the same instructions. This necessitates the data to be laid out in a struct-of-arrays fashion. The reshuffling of the data from array-of-structs format to struct-of-arrays format and back can be done, e.g., by \mathcal{G}_e , while looping through all elements [62]. Note that our implementation performs $C_e \circ \mathcal{G}_e x$ for a single SIMD batch of cells at a time in order to keep intermediate results in caches and reduce global memory traffic. For the application of hanging-node constraints, we use the special-purpose algorithm presented in [69, 81], which is based on updating the DoF map \mathcal{G}_e and applying in-place sum factorization for the interpolation during the application of edge and face constraints. Even though this algorithm is highly optimized and comes with only a small overhead for high-order finite elements ($< 20\%$ per cell with hanging nodes), the additional steps are not for free particularly for linear elements. This might lead to load imbalances in a parallel setting when some processes have a disproportionately high number of cells with hanging nodes, see the quantitative analysis in [81].

Despite relying on matrix-free algorithms overall, some of the multigrid ingredients (e.g., smoother and coarse-grid solver) need an explicit representation of the linear operator in the form of a matrix or a part of it (e.g., its diagonal). Based on the matrix-free algorithm (3) applied to the local unit vector e_i , the local matrix can be computed column by column:

$$A_e(:, i) = S_e^T \circ Q_e \circ S_e e_i. \quad (4)$$

The resulting element matrix can be assembled as usual, including the resolution of the constraints. Computing the diagonal is slightly more complex when attempting to immediately write into a vector for the diagonal without complete matrix assembly. In our code, we choose to compute the j -th entry of the locally relevant diagonal contribution via

$$d_e(j) = \sum_i [C_e^T A_e(:, i)] C_e(i, j) \quad \forall j \in \{j \mid C_{e,ji} \neq 0\}, \quad (5)$$

i.e., we apply the local constraint matrix C_e from the left to the i -th column of the element matrix (computed via (4)) and apply C_e again from the right. This approach needs as many basis vector applications as there are shape functions per cell. The local result can be simply added to the global diagonal via $d = \sum_e \mathcal{G}_e^T d_e$. For cells without constrained DoFs, (5) simplifies to $d_e(i) = A_e(i, i)$.

3.3 Smoother

As the derivation of good matrix-free smoothers is an active research topic, we use Chebyshev iterations around a point-Jacobi method for smoothing [2]. This setup needs an operator evaluation and its diagonal representation according to Subsection 3.2. It is run on the levels defined by either the global-coarsening or the local-smoothing multigrid algorithm. In the case of local smoothing, refinement edges can be treated as homogeneous Dirichlet boundaries under the assumption that the right-hand-side vector is suitably modified.

3.4 Coarse-grid solver

The algorithms described in Subsection 3.2 allow to set up traditional coarse-grid solvers, e.g., a Jacobi solver, a Chebyshev solver, direct solvers, but also AMG. In this publication, we mostly apply AMG as a coarse-grid solver, since we use it either on very coarse meshes (in this case, it falls back

to a direct solver) or for problems discretized with linear elements, for which AMG solvers are competitive [66].

3.5 Transfer operator

The prolongation operator $\mathcal{P}^{(f,c)}$ prolongates the result \mathbf{x} from a coarse space c to a fine space f (between either different geometric grids or different polynomial degrees):

$$\mathbf{x}^{(f)} = \mathcal{P}^{(f,c)} \mathbf{x}^{(c)}$$

According to the literature [14, 99], this can be done in three steps:

$$\mathbf{x}^{(f)} = \mathcal{W}^{(f)} \circ \tilde{\mathcal{P}}^{(f,c)} \circ C^{(c)} \mathbf{x}^{(c)} \quad (6)$$

with $C^{(c)}$ setting the values of constrained DoFs on the coarse mesh, particularly resolving the hanging-node constraints, $\tilde{\mathcal{P}}^{(f,c)}$ performing the prolongation on the discontinuous space as if no hanging nodes were present, and the weighting operator $\mathcal{W}^{(f)}$ zeroing out the DoFs constrained on the fine mesh.

In order to derive a matrix-free implementation, one can express (6) for nested meshes as loops over all (coarse) cells (see also Figure 3):

$$\mathbf{x}^{(f)} = \sum_{e \in \{\text{cells}\}} \mathcal{S}_e^{(f)} \circ \mathcal{W}_e^{(f)} \circ \mathcal{P}_e^{(f,c)} \circ C_e^{(c)} \circ \mathcal{G}_e^{(c)} \mathbf{x}^{(c)} \quad (7)$$

Here, $C_e^{(c)} \circ \mathcal{G}_e^{(c)}$ gathers the cell-relevant coarse DoFs and applies the constraints just as in the case of the matrix-free operator evaluation (3). The operator $\mathcal{P}_e^{(f,c)}$ embeds the coarse-space solution into the fine space, whereas $\mathcal{S}_e^{(f)}$ sums the result back to a global vector. Since multiple cells could add to the same global entry of the vector $\mathbf{x}^{(f)}$ during the cell loop, the values to be added have to be weighted with the inverse of the valence of the corresponding DoF. This is done by $\mathcal{W}_e^{(f)}$, which also ignores constrained DoFs (zero valence) in order to be consistent with (6). Figure 4 shows, as an example, the values of $\mathcal{W}^{(f)}$ for a simple mesh for a scalar Lagrange element of degree $1 \leq p \leq 3$.

We construct the element prolongation matrices as

$$\left(\mathbf{P}_e^{(f,c)} \right)_{ij} = \left(\mathbf{M}_e^{(f)} \right)^{-1} \left(\phi_i^{(f)}, \phi_j^{(c)} \right)_{\Omega_e} \quad \text{with} \quad \left(\mathbf{M}_e^{(f)} \right)_{ij} = \left(\phi_i^{(f)}, \phi_j^{(f)} \right)_{\Omega_e},$$

where $\phi^{(c)}$ are the shape functions on the coarse cell and $\phi^{(f)}$ on the fine cell. Instead of treating each “fine cell” on its own, we group direct children of the coarse cells together and define the fine shape functions on the appropriate subregions. As a consequence, the “finite element” on the fine mesh depends both on the actual finite-element type (like on the polynomial degree and continuity) and on the refinement type, as indicated in Figure 5. For local smoothing, there is only one refinement configuration, since all cells on a finer level are children of cells on the coarser level. In the case of polynomial global coarsening, the mesh stays the same and only the polynomial degree is uniformly changed for all cells. In the case of geometric global coarsening, cells on a finer level are either identical to the cells or direct children of cells on the coarser level, but the element and its polynomial degree stay the same. This necessitates two prolongation cases, an identity matrix for non-refined cells and the coarse-to-fine embedding. We define the set of all coarse-fine cell pairs connected via the same element prolongation matrix as category C .

Since $\mathcal{P}_e^{(f,c)} = \mathcal{P}_{C(e)}^{(f,c)}$, i.e., all cells of the same category $C(e)$ have the same element prolongation matrix, and in order to be able to apply them for multiple elements in one go in a vectorization-over-elements fashion [62] as in the case of matrix-free loops (3), we loop over the cells type by

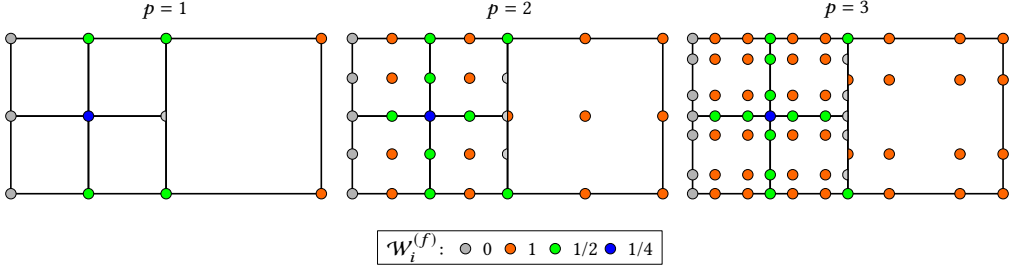


Fig. 4. Example for entries i of $\mathcal{W}^{(f)}$ for a simple mesh configuration with Dirichlet boundary at the left face for a scalar continuous Lagrange element of degree $1 \leq p \leq 3$. In our implementation, constrained DoFs do not contribute to the valence of constraining DoFs, which results in valences of one for DoFs inside constraining (coarse) edges/faces. To reduce overhead by $\mathcal{W}^{(f)}$, our implementation utilizes that, for $p > 2$, all DoFs of a geometric entity (vertex, edge, ...) have the same valence (stored per cell as 9 integers in 2D and 27 in 3D).

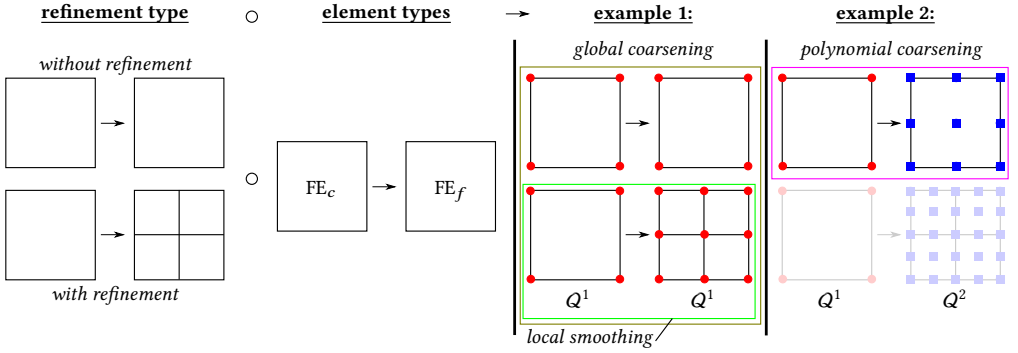


Fig. 5. Left: Construction of the element prolongation $\mathcal{P}_e^{(f,c)}$, based on the **refinement-type** (with/without refinement) and **element-types pair** (coarse and fine FE). Right: Examples for prolongation with and without refinement for equal-degree and different-degree finite elements ($p^{(c)} = p^{(f)} = 1$ vs. $p^{(c)} = 1, p^{(f)} = 2$). Relevant prolongation types for local smoothing, global coarsening, and polynomial coarsening are highlighted. Note that, in the case of global coarsening, two types of prolongation (categories) are needed.

type so that (7) becomes:

$$\mathbf{x}^{(f)} = \sum_c \sum_{e \in \{e | C(e)=c\}} \mathcal{S}_e^{(f)} \circ \mathcal{W}_e^{(f)} \circ \mathcal{P}_e^{(f,c)} \circ \mathcal{C}_e^{(c)} \circ \mathcal{G}_e^{(c)} \mathbf{x}^{(c)}. \quad (8)$$

We choose the restriction operator as the transpose of the prolongation operator

$$\mathcal{R}^{(c,f)} = \left(\mathcal{P}^{(f,c)} \right)^T \quad \text{using locally} \quad \mathcal{R}_e^{(c,f)} = \left(\mathcal{P}_e^{(f,c)} \right)^T.$$

We conclude this subsection with discussing the appropriate data structures for transfer operators, beginning with the ones needed for both global geometric and polynomial coarsening. Since global-coarsening algorithms smoothen on the complete computational domain, data structures only need to be able to perform two-level transfers (8) independently between arbitrary fine (f) and coarse (c) grids. $\mathcal{C}_e^{(c)} \circ \mathcal{G}_e^{(c)}$ is identical to $\mathcal{C}_e \circ \mathcal{G}_e$ in the matrix-free loop (3) so that specialized algorithms and

data structures [81] can be applied and reused. $\mathcal{S}_e^{(f)}$ needs the indices of DoFs for the given element e in order to be able to scatter the values, and $\mathcal{W}_e^{(f)}$ stores the weights of DoFs (or of geometric entities - see also Figure 4) for the given element e . $\mathcal{P}_{C(e)}^{(f,c)}$ (and $\mathcal{R}_{C(e)}^{(c,f)}$) need to be available for each category. We regard them as general operators and choose the most efficient way of evaluation based on the element types: simple dense-matrix representation vs. some substructure with sum factorization based on smaller 1D prolongation matrices. The category of each cell has to be known for each cell.

Alongside these process-local data structures, one needs access to all constraining DoFs on the coarse level required during $C_e^{(c)} \circ \mathcal{G}_e^{(c)}$ and to the DoFs of all child cells on the fine level during the process-local part of $\mathcal{S}_e^{(f)}$, which is concluded by a data exchange. If external vectors do not allow access to the required DoFs, we copy data to/from internal temporal global vectors with appropriate ghosting [62]. We choose the coarse-side identification of cells due to its implementation simplicity and structured data access at the price of more ghost transfer.³ For setting up the communication pattern, we use consensus-based sparse dynamic algorithms [6, 47].

For the sake of separation of concerns, one might create three classes to implement a global-coarsening transfer operator as we have done in deal . II. The relation of these classes is shown in Figure 6: the multigrid transfer class (MGTransferGlobalCoarsening) delegates the actual transfer tasks to the right two-level implementation (MGTwoLevelTransfer), which performs communications needed as well as evaluates (8) for each category and cell by using category-specific information from the third class (MGTransferSchemes).

The discussed data structures are applicable also to local smoothing. The fact that only a single category is available in this case allows to simplify many code paths. However, one needs data structures that match DoF indices on the active level and on *all* multigrid levels that share a part of the mesh with the active level in addition to the two-level data structures. For further details on implementation aspects of transfer operators for local smoothing, see [28, 64] and the documentation of MGTransferMatrixFree class in deal . II.

4 PERFORMANCE METRICS

In Section 3, we have presented efficient implementations of the operators in Algorithm 1 for local smoothing, global coarsening, and polynomial coarsening. Most of the discussion was independent of the multigrid variant chosen, highlighting the similarities from the implementation point of view. The main differences arise naturally from the local or global definition of levels: e.g., one might need to consider hanging-node constraints during matrix-free loops when doing global coarsening or polynomial coarsening. Local smoothing, on the other hand, has the disadvantage of performing additional steps: 1) global transfer to/from multigrid levels and 2) special treatment of refinement edges. In the following sections, we quantify the influence of the costs of the potentially more expensive operator evaluations and of the additional operator evaluations, related to the choice of the multigrid level definition.

Our primary goal is to minimize the **time to solution**. It consists of setup costs and the actual solve time, which is the product of the **number of iterations** and the **time per iteration**. We will disregard the setup costs, since they normally amortize in time-dependent simulations, where one

³Sundar et al. [99] showed that, by assigning all children of a cell to the same process, one can easily derive an algorithm that allows to perform the cell-local prolongation/restriction on the fine side, potentially reducing the amount of data to be communicated during the transfer. Since we allow levels to be partitioned arbitrarily in our implementation, we do not use this approach. Furthermore, one should note that the algorithm proposed there does not allow to apply the constraints $C_e^{(c)}$ during a single cell loop as in (8), but needs a global preprocessing step as in (6), potentially requiring additional sweeps through the whole data with access to the slow main memory.

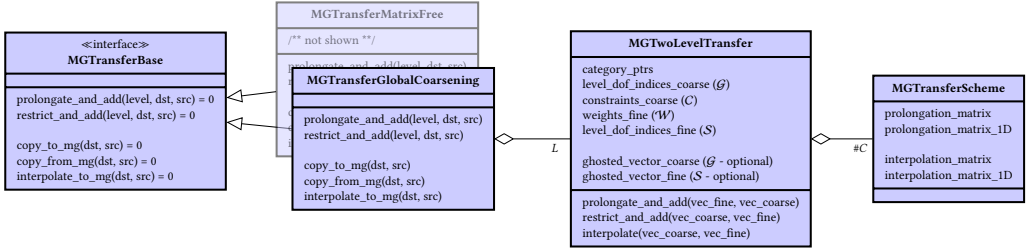


Fig. 6. UML diagram of the global-coarsening transfer operator `MGTransferGlobalCoarsening` in deal.II. It implements the base class `MGTransferBase`, which is also the base class of `MGTransferMatrixFree` (local smoothing), and delegates its prolongation/restriction/interpolation tasks to the right `MGTwoLevelTransfer` instance. Each of these instances is defined between two levels and loops over categories/cells in order to evaluate (8) by using the prolongation matrices from the correct `MGTransferScheme` object. Furthermore, it is responsible for the communication, for which it has two optional internal vectors with appropriate ghosting. `MGTwoLevelTransfer` objects can be initialized for geometric or polynomial global coarsening, enabling support for (global) h -, p -, and hp -multigrid in `MGTransferGlobalCoarsening`.

does not remesh every time step, and ways to optimize the setup of global-coarsening algorithms are known in the literature [99]. The time of the solution process strongly depends on the choice of the smoother, which influences the number of iterations and, as a result, also the time to solution. Since different iteration numbers might distort the view on the performance of the actual multigrid algorithm, we will also consider the value of the time per iteration as an important indicator of the computational performance of multigrid algorithms particularly to quantify the additional costs in an iteration.

In order to get a first estimate of the benefits of an algorithm compared to another, one can derive following metrics purely from geometrical information:

- The **serial workload** can be estimated as the sum of the number of cells on all levels $W_s = \sum_l C_l$, with C_l being the number of cells on level l . This metric is based on the assumption that all cells have the same costs, which is not necessarily true in the context of hanging nodes [81].
- The **parallel workload** can be estimated as the sum of the maximum number of cells owned by any process p on each level: $W_p = \sum_l \max_p C_l^p$, i.e., the critical path of the cells. In the ideal case, one would expect that $C_l^p = C_l/P$ with P being the number of processes and therefore $W_p = W_s/P$. However, due to potential load imbalances, the work might not be well distributed on the levels, i.e., $\max_p (C_l^p) > C_s/P$. Since one can theoretically only proceed to the next level once all processes have finished processing a level, load imbalances will result in some processes waiting at imaginary barriers. We say “imaginary barriers” as level operators only involve point-to-point communication between neighboring processes in the grid for our implementation. Nevertheless, this simplified point of view is acceptable, since multigrid algorithms are multiplicative Schwarz methods between levels, inherently leading to a sequential execution of the levels. We define **parallel workload efficiency** as $W_s/(W_p \cdot P)$, as has been also done in [28].
- We define **horizontal communication efficiency** as 50% of the number of ghost cells accumulated over all ranks and divided by the total number of cells. The division by two is necessary to take into account that only one neighbor is updating the ghost values. As such, this ratio can be seen as a proxy of how much information needs to be communicated when

computing residuals and updating ghost values. As this number counts cells, it is independent of the polynomial degree of the element chosen. The element degree used determines the absolute amount of communication necessary. Note that, in reality, only degrees of freedom located at the interface have to be exchanged such that the fraction of the solution that needs to be communicated is less than the fraction of those cells.

- **Vertical communication efficiency** is the share of fine cells that have the same owning process as their corresponding coarse cell (parent). This quantity gives an indication on the efficiency of the transfer operator and on how much data has to be exchanged. A small number indicates that most of the data has to be completely permuted, involving a large volume of communication. This metric has been considered in [28] as well.
- Increasing the **number of (multigrid) levels** leads to additional synchronization points and communication steps and, as a result, might lead to increased latency. The absolute value of the minimum time per V-cycle can be roughly approximated by

$$\Delta t_{\text{Latency}}^{\text{V-Cycle}} = L \left(\underbrace{(k-1)}_{S_{\text{pre}}} + \underbrace{1}_{\text{residual}} + \underbrace{1}_{\mathcal{R}} + \underbrace{1}_{\mathcal{P}} + \underbrace{k}_{S_{\text{post}}} \right) \Delta t_{\text{latency}}^{\text{vmult}} = 2L(k+1)\Delta t_{\text{latency}}^{\text{vmult}} \quad (9)$$

as the product of $t_{\text{latency}}^{\text{vmult}}$ (the latency of an operator evaluation consisting of one update to the ghost values in the source vector and one step to send integral contributions in the destination vector back to the owner) and the sum of all communication steps accumulated over presmoothing, residuum evaluation, restriction, prolongation, and postsmoothing on all levels. In this approximation, we assume that the coarse-grid solver does not perform any communication and has negligible latency. Furthermore, the smoother is assumed to have the same communication cost as one matrix-vector product in each of the k iterations, which is the case for simple Chebyshev iterations around point-Jacobi preconditioners [58, 65].

The metrics “parallel workload efficiency”, “horizontal communication efficiency”, and “vertical communication efficiency” as well as “latency” (at the scaling limit) all influence the experimentally measurable “parallel (strong-scaling) efficiency” of a multigrid solver.

Furthermore, **memory consumption** of the grid class is a secondary metric.⁴ A common argument supporting the usage of local-smoothing algorithms is that no space is needed for potentially differently partitioned meshes and complex data structures providing the connectivity between them, since the multigrid algorithm can simply reuse the already existing mesh hierarchy also for the multigrid levels [28].

Examples

In the experimental sections 5 and 6, we consider two types of static 3D meshes, as has been also done in [28]. They are obtained by refining a coarse mesh consisting of a single cell defined by $[-1, 1]^3$ according to one of the following two solution criteria:

- **octant**: refine all mesh cells in the first octant $[-1, 0]^3$ L times and
- **shell**: after $L - 3$ uniform refinements, perform three local refinement steps with all cells whose center \mathbf{c} is $|\mathbf{c}| \leq 0.55$, $0.3 \leq |\mathbf{c}| \leq 0.43$, and $0.335 \leq |\mathbf{c}| \leq 0.39$.

These two meshes are relevant in practice, since similar meshes occur in simulations of flows with far fields and of multi-phase flows with bubbles [59] or any kind of interfaces. All the refinement procedures are completed by a closure after each step, ensuring one-irregularity in the sense that two leaf cells may only differ by one level if they share a vertex. Figure 7 shows the considered

⁴We use the memory-consumption output provided by deal.II. No particular efforts have been put in reducing the memory consumption of the triangulations in the case of global coarsening.

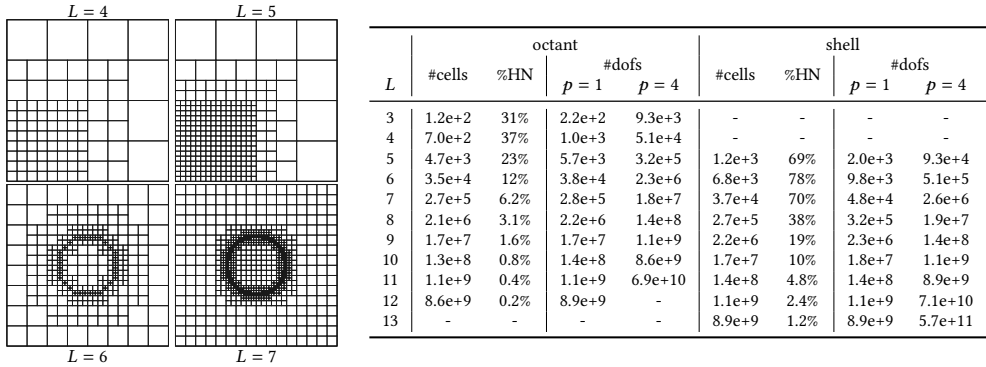


Fig. 7. Cross section at the center of the geometries of the octant (top) and the shell (bottom) simulation. Additionally, the number of cells, the share of cells with hanging-node constraints, and the number of DoFs (for a scalar Lagrange element with $p = 1$ and $p = 4$) are given for each refinement case.

meshes and provides numbers regarding the cell count for $3 \leq L \leq 13$. All meshes are partitioned along space-filling curves [15, 26] with the option to assign cells weights.

Tables 1 and 2 give—as examples—evaluated numbers for geometrical metrics of the two considered meshes for a single process and for 192 processes with cells constrained by hanging nodes weighted with the factor of 2 for partitioning, compared to the rest of the cells. For a single process, only workload and memory consumption are shown. In the tables, a bold face indicates the most beneficial behavior among the listed variants.

Starting with the octant case, one can see that the serial workload in the case of local smoothing and global coarsening is similar, with local smoothing having consistently less work. The workload of each level is depicted in Figure 8. The behavior of the memory consumption is similar to the one of the workload: global coarsening has a slightly higher memory consumption, since it explicitly needs to store the coarser meshes as well; however, the second finest mesh has already approximately one eighth of the size of the coarsest triangulation so that the overhead is small. In the parallel case, the memory consumption differences are higher: this is related to the fact that—in the case of global coarsening—overlapping ghosted forests of trees need to be saved. In contrast, the workload in the parallel case is much lower in the case of global coarsening: while global coarsening is able to reach efficiencies higher than 90%, the efficiency is only approximately 50–60% in the case of local smoothing. The high value in the global-coarsening case was to be expected, since we repartition each level during construction. The low value in the case of local smoothing can be seen in Figure 9, where the minimum, maximum, and average workload are shown for each level. The workload on the finest level is optimally distributed between processes with cells, however, there are some processes without any cells, i.e., any work, on that level. On the second level, most processes can reduce the number of cells nearly optimally by a factor of 8, but processes idle on the finest level start to participate in the smoothing process with a much higher number of cells, similarly to what other processes process on the finest level. This pattern of discrepancy between the minimum and the maximum number of cells on the lower levels continues and, as a consequence, the maximum workload per level is higher, increasing the overall parallel workload. Figure 9 might also give the impression of a load imbalance in the case of global coarsening, since the minimum workload on the finest level is the half of the maximum value. This is related to the way we partition—penalizing cells that have hanging nodes with a weight of 2—and to the fact that a lot of cells with hanging nodes are clustered locally. The actual resulting load imbalance is small, as shown in Figure 12.

Table 1. Geometrical multigrid statistics for the octant test case for different numbers of refinements (wl: serial/parallel workload, wl-eff: parallel workload efficiency, v-eff: vertical communication efficiency, h-eff: horizontal communication efficiency; mem: memory consumption in bytes). Figures 8 (serial) and 9 (parallel) give a detailed breakdown of the workload of each multigrid level for $L = 8$.

| L | 1 process | | | | 192 processes | | | | | | | | | |
|-----|-----------------|--------|-------------------|--------|-----------------|--------|-------|-------|---------|-------------------|--------|-------|--------|---------|
| | local smoothing | | global coarsening | | local smoothing | | | | | global coarsening | | | | |
| | wl | mem | wl | mem | wl | wl-eff | v-eff | h-eff | mem | wl | wl-eff | v-eff | h-eff | mem |
| 3 | 1.4e+2 | 6.6e+4 | 1.4e+2 | 8.7e+4 | 1.8e+1 | 3% | 89% | 60% | 1.3e+6 | 2.5e+1 | 3% | 17% | 1.0e+5 | 2.7e+6 |
| 4 | 8.0e+2 | 3.3e+5 | 8.4e+2 | 4.2e+5 | 2.2e+1 | 18% | 85% | 56% | 1.2e+7 | 3.3e+1 | 13% | 4% | 1.0e+5 | 1.5e+7 |
| 5 | 5.4e+3 | 2.0e+6 | 5.6e+3 | 2.5e+6 | 7.2e+1 | 38% | 88% | 58% | 5.3e+7 | 6.7e+1 | 43% | 1% | 59% | 6.8e+7 |
| 6 | 4.0e+4 | 1.4e+7 | 4.0e+4 | 1.7e+7 | 4.0e+2 | 51% | 96% | 65% | 1.3e+8 | 2.8e+2 | 76% | 6% | 66% | 2.0e+8 |
| 7 | 3.1e+5 | 1.1e+8 | 3.1e+5 | 1.3e+8 | 2.7e+3 | 58% | 99% | 75% | 4.2e+8 | 1.7e+3 | 92% | 13% | 75% | 6.2e+8 |
| 8 | 2.4e+6 | 8.6e+8 | 2.4e+6 | 9.9e+8 | 2.0e+4 | 62% | 99% | 84% | 1.8e+9 | 1.3e+4 | 96% | 23% | 84% | 2.4e+9 |
| 9 | 1.9e+7 | 6.8e+9 | 1.9e+7 | 7.8e+9 | 1.6e+5 | 64% | 99% | 91% | 1.0e+10 | 1.0e+5 | 98% | 38% | 91% | 1.3e+10 |

Table 2. Geometrical multigrid statistics for the shell test case for different numbers of refinements. The label “-” indicates that the simulation ran out of memory.

| L | 1 process | | | | 192 processes | | | | | | | | | |
|-----|-----------------|--------|-------------------|--------|-----------------|--------|-------|-------|---------|-------------------|--------|-------|-------|---------|
| | local smoothing | | global coarsening | | local smoothing | | | | | global coarsening | | | | |
| | wl | mem | wl | mem | wl | wl-eff | v-eff | h-eff | mem | wl | wl-eff | v-eff | h-eff | mem |
| 5 | 1.4e+3 | 6.0e+5 | 1.8e+3 | 9.5e+5 | 3.1e+1 | 22% | 80% | 55% | 3.2e+7 | 4.5e+1 | 21% | 2% | 56% | 4.7e+7 |
| 6 | 7.8e+3 | 3.2e+6 | 9.2e+3 | 4.4e+6 | 1.6e+2 | 26% | 89% | 59% | 7.7e+7 | 1.0e+2 | 47% | 12% | 59% | 1.3e+8 |
| 7 | 4.2e+4 | 1.7e+7 | 4.9e+4 | 2.2e+7 | 7.6e+2 | 28% | 96% | 66% | 1.5e+8 | 4.3e+2 | 58% | 36% | 65% | 3.0e+8 |
| 8 | 3.1e+5 | 1.2e+8 | 3.4e+5 | 1.4e+8 | 4.7e+3 | 34% | 99% | 76% | 4.4e+8 | 2.3e+3 | 75% | 78% | 75% | 7.7e+8 |
| 9 | 2.5e+6 | 8.9e+8 | 2.5e+6 | 1.1e+9 | 3.5e+4 | 36% | 99% | 85% | 1.8e+9 | 1.5e+4 | 86% | 93% | 84% | 2.7e+9 |
| 10 | - | - | - | - | 2.7e+5 | 38% | 99% | 91% | 1.0e+10 | 1.1e+5 | 92% | 97% | 91% | 1.3e+10 |

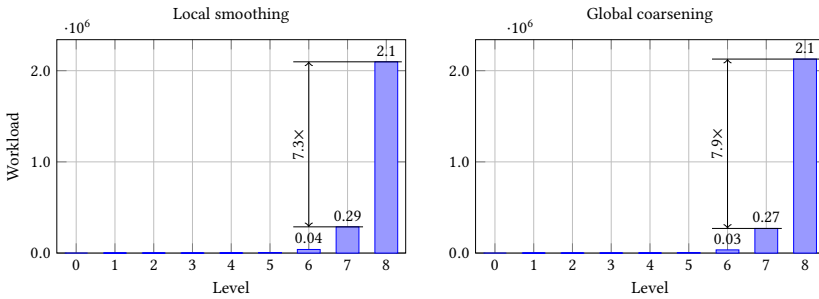


Fig. 8. Workload of each multigrid level of an octant simulation with a single process for $L = 8$.

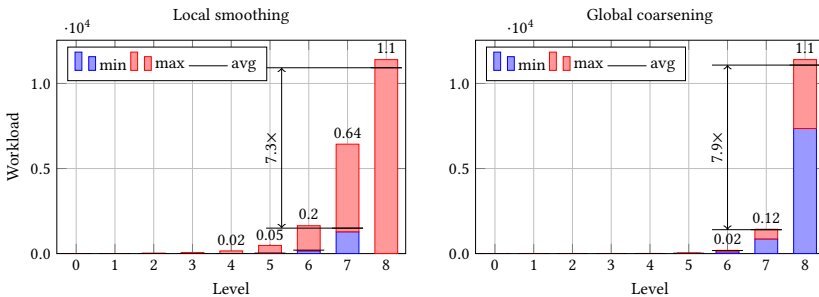


Fig. 9. Workload of each multigrid level of an octant simulation with 192 processes for $L = 8$.

The difference between local smoothing and global coarsening in the parallel workload comes at a price. While the vertical communication efficiency is—by construction—high in the local-smoothing case, this is not true in the case of global coarsening: 20% or less is not uncommon, requiring the permutation of data during transfer. The horizontal communication costs are similar in the case of local smoothing and global coarsening. Assuming that 1) pre- and postsmoothing are the most time-consuming steps, 2) the parallel workload is the relevant metric, and 3) the transfer between levels is not dominant in this case, the values indicate that global coarsening might be twice as fast as local smoothing. However, if the transfer is the bottleneck, the picture might look differently so that a conclusive statement only based on geometrical metrics is not possible in this case.

The observations for the octant case are also made, in a more pronounced form, in the shell case. Here, the vertical communication is more favorable in the case of global coarsening and the workload efficiency is significantly worse in the local-smoothing case so that one can expect a noticeable speedup when using global coarsening.

In summary, one can state the following: in the serial case, global coarsening has to process at least as many cells as local smoothing so that one can expect that the latter has—with the assumption that the number of iterations is the same—an advantage regarding throughput, particularly since no hanging-node constraints have to be applied. With an increasing number of processes, the workload might not be well distributed in the case of local smoothing, leading to a drop in parallel efficiency. In contrast, this is—by construction—not an issue in the global-coarsening case, since the work is explicitly redistributed between the levels. The price is that one might need to send around a lot of data during restriction and prolongation. The number of levels in the case of local smoothing and global coarsening is the same, leading to potentially same scaling limits $\mathcal{O}(L)$. However, with appropriate partitioning of the levels, one could decrease the number of participating processes on the coarser levels and switch to a coarse-grid solver at an earlier stage in the global-coarsening case, leading to a better scaling limit due to lower latency. In the following, we show experimentally that the above statements can be verified and take a more detailed look at the influence of the mutually contradicting requirements “good workload balance” and “cheap transfer” on the parallel efficiency.

Making definite general conclusions is difficult as they depend on the number of processes as well as on the type of the coarse mesh and of the refinement. While the two meshes considered here are prototypical for many problems we have encountered in practice, it is clear that the statements made in this publication can not hold in all cases, since it is easy to construct examples that favor one multigrid variant over the other. However, the two meshes clearly demonstrate the dominating aspects at various problem sizes; the actual crossover point, however, might be problem- and hardware-specific.

5 PERFORMANCE ANALYSIS: H-MULTIGRID

In the following, we solve a 3D Poisson problem with homogeneous Dirichlet boundary conditions and a constant right-hand side in order to compare the performance of local-smoothing and global-coarsening algorithms for the octant and shell test cases, as introduced in Section 4. The results for a more complex setup with non-homogeneous Dirichlet boundary conditions are shown in Table 6 in Appendix A.

We will use a continuous Lagrange finite element, whose shape functions are defined as the tensor products of 1D shape functions with degree p . For quadrature, we consider the consistent Gauss–Legendre quadrature rule with $(p + 1)^3$ points.

We start with investigating the serial performance, proceed with parallel execution with moderate numbers of processes, and finally analyze the parallel behavior on the large scale (150k processes). We conclude this section with investigation of an alternative partitioning scheme for global coarsening.

In order to obtain the best performance, the experiments are configured in the following way:

- Cells with hanging-node constraints are weighted by the factor of 2.
- The conjugate-gradient solver is run until a reduction of the l_2 -norm of the unpreconditioned residual by 10^4 is obtained. We choose a rather coarse tolerance, since this is a common value for the solution of time-dependent problems, such as the Navier–Stokes equations, where good initial guesses can be obtained by projection and extrapolation without the need to converge multigrid to many digits. Similarly, coarse tolerances also indicate the costs of solving in a full-multigrid scenario with the finest level only correcting against the next coarser one.
- The conjugate-gradient solver is preconditioned by a single V-cycle of either a local-smoothing or a global-coarsening multigrid algorithm.
- To increase the throughput, all operations in the multigrid V-cycle are run with single-precision floating-point numbers, while the conjugate-gradient solver is run in double precision [64].
- We use a Chebyshev/point-Jacobi smoother of degree 3 on all levels.
- As coarse-grid solver, we use two V-cycles of AMG (double-precision, ML [34] with parameters shown in Appendix C).

The results of performance studies leading to the decision on the configuration described above are presented in Tables 7–11 in Appendix A. All experiments have been conducted on the SuperMUC-NG supercomputer. Its compute nodes have 2 sockets (each with 24 cores of Intel Xeon Skylake) and the AVX-512 ISA extension so that 8 doubles or 16 floats can be processed per instruction. A detailed specification of the hardware is given in Table 3. The parallel network is organized into islands of 792 compute nodes each. The maximum network bandwidth per node within an island is $100\text{Gbit/s}=12.5\text{GB/s}^5$ via a fat-tree network topology. Islands are connected via a pruned-tree network architecture (pruning factor 1:4). We have run all experiments six times and report the best timings of the last five runs. We would like to note that executions on SuperMUC-NG are relatively noisy even for small-scale simulations with hundreds of processes, which results in performance degeneration by up to 50%. Our investigations revealed that this has only a minor effect on the execution times of the multigrid variants in comparison. Nevertheless, to rule out any source of inconsistency in the results, we report timings for all multigrid variants from the same job execution in each table and figure. Apart from these machine-caused outliers, the statistical distribution is within a few percent of the minimum and not explicitly reported in the following figures.

For local smoothing and global coarsening, we use different implementations of the transfer operator from deal.II (see Subsection 3.5). In order to demonstrate that they are equivalent and results shown in the following are indeed related to the definition of multigrid levels and the resulting different algorithms, Table 12 in Appendix A presents a performance comparison of these implementations for uniformly refined meshes of a cube, for which both algorithms are equivalent.

For global coarsening, we have also investigated the possibility to decrease the number of participating processes and to switch to the coarse-grid solver earlier. For our test problems, we could not see any obvious benefits for the time to solution so that we do not use these features of global coarsening in this publication, but defer their investigation to future work.

5.1 Serial runs: overview

Figure 10 gives an overview of the time shares during the solution process in a serial shell simulation for local smoothing and global coarsening. Without going into details of the actual

⁵<https://doku.lrz.de/display/PUBLIC/SuperMUC-NG>, retrieved on February 26, 2022.

Table 3. Specification of the hardware system used for evaluation. Memory bandwidth is according to the STREAM triad benchmark (optimized variant without read for ownership transfer involving two reads and one write), and GFLOP/s are based on the theoretical maximum at the AVX-512 frequency. The dgemm performance is measured for $m = n = k = 12,000$ with Intel MKL 18.0.2. We measured a frequency of 2.5 GHz with AVX-512 dense code for the current experiments. The empirical machine balance is computed as the ratio of measured dgemm performance and STREAM bandwidth from RAM memory.

| Intel Skylake Xeon Platinum 8174 | |
|--|--|
| cores | 2×24 |
| frequency base (max AVX-512 frequency) | 2.7 GHz |
| SIMD width | 512 bit |
| arithmetic peak (dgemm performance) | 4147 GFLOP/s (3318 GFLOP/s) |
| memory interface | DDR4-2666, 12 channels |
| STREAM memory bandwidth | 205 GB/s |
| empirical machine balance | 14.3 FLOP/Byte |
| L1-/L2-/L3-/MEM size | 32kB (per core)/1MB (per core)/66MB (shared)/96GB(shared) |
| compiler + compiler flags | g++, version 9.1.0, -O3 -funroll-loops -march=skylake-avx512 |

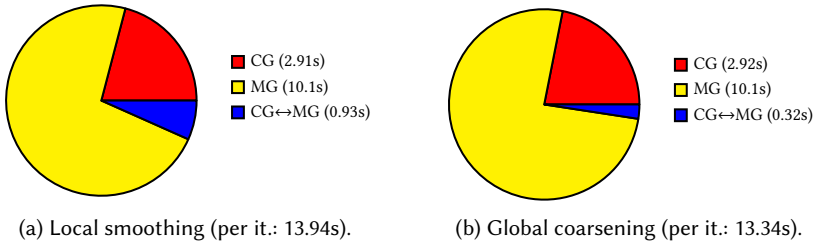


Fig. 10. Time per iteration spent for conjugate-gradient solver (CG), multigrid preconditioner (MG), as well as transfer between solver and preconditioner (CG↔MG) in a serial shell simulation with $k = 4$ and $L = 9$.

numbers, one can see that most of the time is spent in the multigrid preconditioner in the case of both local smoothing and global coarsening (72%/76%). It is followed by the other operations in the outer conjugate-gradient solver (21%/22%). The least time is spent for transferring data between preconditioner and solver (7%/2%). The higher time of the transfer in the local-smoothing case is not surprising, since the transfer involves all multigrid levels sharing cells with the active level. Since the overwhelming share of solution time is taken by the multigrid preconditioner, all detailed analysis in the remainder of this work concentrates on the multigrid V-cycle.

One should note that spending 72%/76% of the solution time within the multigrid preconditioner is already low, particularly taking into account that the outer conjugate-gradient solver also performs its operator evaluations (1 matrix-vector multiplication per iteration) in an efficient matrix-free fashion. The low costs are related to the usage of single-precision floating-point numbers and to the low number of pre-/postsMOOTHING steps, which results in a total of 6–7 matrix-vector multiplications per level and iteration.

5.2 Serial run

Tables 4 and 5 show the number of iterations and the time to solution for the octant and the shell test cases run serially with local smoothing and global coarsening as well as with the polynomial degrees $p = 1$ and $p = 4$.

It is well visible that local smoothing has to perform at least as many iterations as global coarsening, with the difference in iterations limited to 1 in the examples considered. This difference

Table 4. Number of iterations and time to solution for local smoothing (LS) and global coarsening (GC) with 1 process and 192 processes for the octant simulation case. Figures 11 (serial) and 12 (parallel) give a detailed breakdown of the times of one V-cycle for the case shaded in gray, $L = 8$ and $p = 4$.

| L | 1 process | | | | | | | | 192 processes (4 nodes) | | | | | | | |
|----|-----------|---------------|------|---------------|---------|---------------|------|---------------|-------------------------|--------|------|---------------|---------|--------|------|---------------|
| | $p = 1$ | | | | $p = 4$ | | | | $p = 1$ | | | | $p = 4$ | | | |
| | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | | |
| #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | |
| 3 | 4 | 5.0e-4 | 4 | 6.0e-4 | 4 | 3.8e-3 | 4 | 4.3e-3 | 4 | 1.2e-3 | 4 | 1.0e-3 | 4 | 2.5e-3 | 4 | 2.5e-3 |
| 4 | 4 | 1.8e-3 | 4 | 2.5e-3 | 4 | 1.8e-2 | 4 | 2.0e-2 | 4 | 2.6e-3 | 4 | 1.7e-3 | 4 | 4.3e-3 | 4 | 4.1e-3 |
| 5 | 4 | 8.6e-3 | 4 | 1.2e-2 | 4 | 1.1e-1 | 3 | 8.7e-2 | 4 | 5.3e-3 | 4 | 2.6e-3 | 4 | 6.9e-3 | 3 | 5.0e-3 |
| 6 | 4 | 5.1e-2 | 4 | 6.5e-2 | 4 | 8.8e-1 | 3 | 6.5e-1 | 4 | 5.6e-3 | 4 | 4.1e-3 | 4 | 1.7e-2 | 3 | 1.0e-2 |
| 7 | 4 | 3.6e-1 | 3 | 3.2e-1 | 4 | 6.9e+0 | 3 | 5.0e+0 | 4 | 1.3e-2 | 3 | 5.7e-3 | 4 | 8.4e-2 | 3 | 5.0e-2 |
| 8 | 4 | 2.8e+0 | 3 | 2.3e+0 | 4 | 5.3e+1 | 3 | 3.8e+1 | 4 | 3.9e-2 | 3 | 2.1e-2 | 4 | 7.2e-1 | 3 | 4.3e-1 |
| 9 | 4 | 2.2e+1 | 3 | 1.8e+1 | - | - | - | - | 4 | 2.3e-1 | 3 | 1.3e-1 | - | - | - | - |

Table 5. Number of iterations and time to solution for local smoothing (LS) and global coarsening (GC) with 1 process and 192 processes for the shell simulation case.

| L | 1 process | | | | | | | | 192 processes (4 nodes) | | | | | | | |
|----|-----------|---------------|------|--------|---------|---------------|------|---------------|-------------------------|--------|------|---------------|---------|--------|------|---------------|
| | $p = 1$ | | | | $p = 4$ | | | | $p = 1$ | | | | $p = 4$ | | | |
| | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | | |
| #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | |
| 5 | 5 | 3.6e-3 | 4 | 7.1e-3 | 4 | 3.1e-2 | 4 | 4.5e-2 | 5 | 6.6e-3 | 4 | 3.1e-3 | 4 | 6.4e-3 | 4 | 6.3e-3 |
| 6 | 5 | 1.5e-2 | 4 | 2.8e-2 | 4 | 1.9e-1 | 4 | 2.2e-1 | 5 | 6.8e-3 | 4 | 4.1e-3 | 4 | 1.0e-2 | 4 | 9.2e-3 |
| 7 | 5 | 7.3e-2 | 4 | 1.3e-1 | 4 | 1.0e+0 | 4 | 1.2e+0 | 5 | 9.2e-3 | 4 | 5.6e-3 | 4 | 2.3e-2 | 4 | 1.8e-2 |
| 8 | 5 | 5.1e-1 | 4 | 7.3e-1 | 4 | 7.8e+0 | 4 | 7.9e+0 | 5 | 1.7e-2 | 4 | 1.2e-2 | 4 | 1.1e-1 | 4 | 6.9e-2 |
| 9 | 5 | 3.7e+0 | 4 | 4.3e+0 | 4 | 5.8e+1 | 4 | 5.6e+1 | 5 | 6.3e-2 | 4 | 3.5e-2 | 4 | 8.5e-1 | 4 | 5.2e-1 |
| 10 | - | - | - | - | - | - | - | - | 5 | 3.9e-1 | 4 | 1.9e-1 | - | - | - | - |

is due to the additional smoothening applied to certain cells in the course of global coarsening. Note that global coarsening also benefits from the simple setup of a smooth solution with artificial refinement, see Table 6 in the appendix for a somewhat more realistic test case. This comes with a higher serial workload for global coarsening, which is also visible in the times of a single V-cycle (not shown). Nevertheless, the lower number of iterations leads to a smaller time to solution in the case of global coarsening in some instances. Generally, the global-coarsening V-cycle is relatively more expensive in the case of the shell simulation with linear elements: This is not surprising due to the higher number of cells with hanging-node constraints (see also Figure 7) in the shell case and the higher overhead of linear elements for application of hanging-node constraints, as analyzed by [81].

Figure 11 shows the distribution of times spent on each multigrid level and in each multigrid stage for the octant case with $L = 8$ and $p = 4$. While the overall runtimes are similar for local smoothing and global coarsening, distinct (and expected) differences are visible for the individual ingredients: The restriction and prolongation steps take about the same time in both cases. The presmoothing, residual, and postsmoothing steps are slightly more expensive in the global-coarsening case, which is related to the observation that the evaluation of the level operator $\mathbf{A}^{(l)}$ ($2/1/3$ -times) is the dominating factor. For local smoothing, the computation of $\mathbf{A}_{ES}^{(l)} \mathbf{x}_S^{(l)}$ is realized as a side product of the application of $\mathbf{A}^{(l)}$, hence limiting its impact on the residual step. The only visible additional cost of local smoothing is $\mathbf{A}_{SE}^{(l)} \mathbf{x}_E^{(l)}$ for the modification of the right-hand-side vector for postsmoothing to incorporate inhomogeneous Dirichlet boundary conditions (edge step).

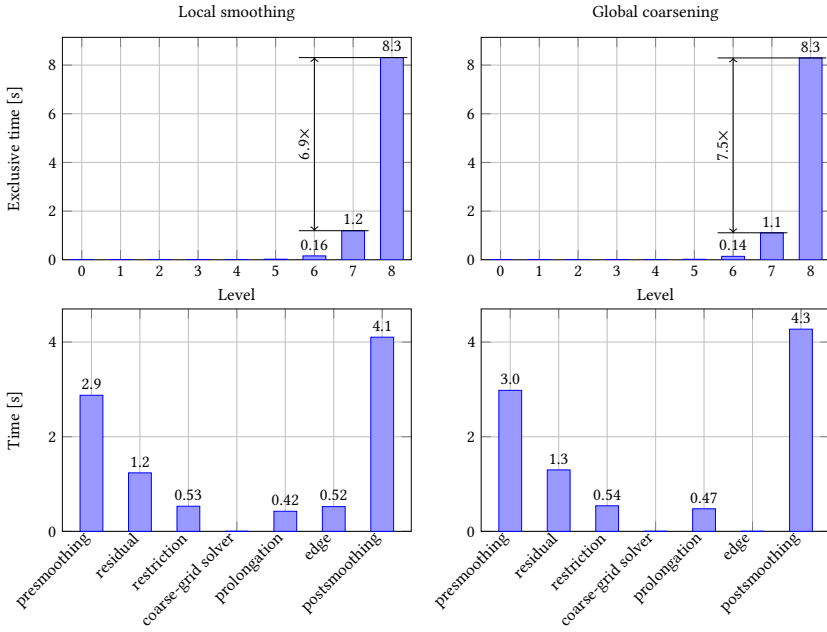


Fig. 11. Profile of one V-cycle of an octant simulation with a single process for $L = 8$ and $p = 4$

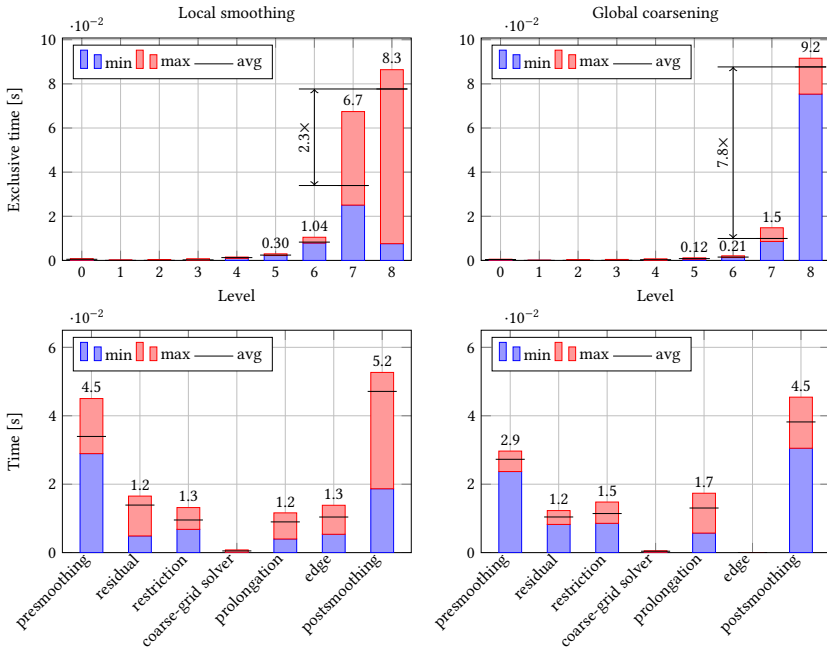


Fig. 12. Profile of one V-cycle of an octant simulation with 192 processes for $L = 8$ and $p = 4$

However, its evaluation is less expensive than the application of $\mathbf{A}^{(l)}$, since only DoFs in proximity to the interface are updated.

5.3 Moderately parallel runs

We report our findings of simulations with 192 processes on 4 compute nodes. Tables 4 and 5 confirm the significance of a good workload balance for the time to solution, as indicated by Tables 1 and 2. The number of processes does not influence the number of iterations due to the chosen smoother. Speedups—compared to local smoothing—are reached: up to 2.3/1.7 for the octant ($p = 1/p = 4$) and up to 2.1/1.6 for the shell case if the different iteration numbers are considered. Normalized per solver iteration, the advantage of global coarsening in these four cases is 2.0, 1.3, 1.7 and 1.6, respectively. Figure 12 illustrates this behavior by showing the distribution of the minimum/maximum/average times spent on each multigrid level and each multigrid stage for the octant case with $L = 8$. In the case of global coarsening, it is well visible that the load is equally distributed and the time spent on the levels is significantly reduced level by level for the higher levels. For the finest ones, local smoothing shows a completely different picture. On the finest level, there are processes with hardly any work, but nevertheless the average work is close to the maximum value, indicating that the load is well-balanced among the processes with work. However, on the second finest level, a significant workload imbalance—by a factor of 2.8—is visible also among the processes with work. This leads to the situation that the maximum time spent on the second finest level is just slightly less than the one on the finest level, contradicting our expectation of a geometric series and leading to the observed increase in the total runtime.

5.4 Large-scale parallel run

Figures 13 and 14 show results of scaling experiments starting with 1 compute node (48 processes) up to 3,072 nodes (147,456 processes). Besides the times of a single V-cycle, we plot the *normalized throughput* (DoFs per process and time per iteration) against the *time per iteration*. The plot can be read in the following way: far from the scaling limit (right top corner of the plot), simulations take longer, but are more efficient (parallel efficiency of one); at the scaling limit (left bottom corner of the plots), simulations reach shorter times at the cost of lower efficiencies. Here, a horizontal behavior from the right to the left corresponds to ideal strong scaling. If different lines coincide, we observe ideal weak scaling. Based on this plot, one can judge the resource utilization and parallel efficiency of a simulation if a certain time to solution should be obtained.

The throughput for the polynomial degree $p = 4$ is higher by a factor of 3 than for $p = 1$. This is expected and related to the used matrix-free algorithms and their node-level performance, which improves with the polynomial order [66]. Just as in the moderately parallel case (see Subsection 5.3), we can observe better timings in the case of global coarsening for a large range of configurations (max. speedup: octant 1.9/1.4 for $p = 1/p = 4$, shell: 2.4/2.4). The number of iterations of local smoothing is 4 for both cases and all refinement numbers. For global coarsening, it is 4 for the sphere case and decreases from 4 to 2 with increasing number of refinements in the case of quadrant so that the actual speedups reported above are even higher. The high speedup numbers of global coarsening in the shell simulation case are particularly related to its high workload and vertical efficiency, as shown in Table 5.

The normalized plots give additional insights. Apart from the obvious observation that the minimal time to solution increases with increasing number of refinements (left bottom corner of the plots in Figures 13 and 14) and global coarsening starts with higher throughputs, one can also see that the decrease in parallel efficiency is more moderate in the case of global coarsening. For the example of the octant case with $p = 1/L = 10$, one can increase the number of processes by a factor of 16 and still obtain a throughput per process that is higher than the one in the case

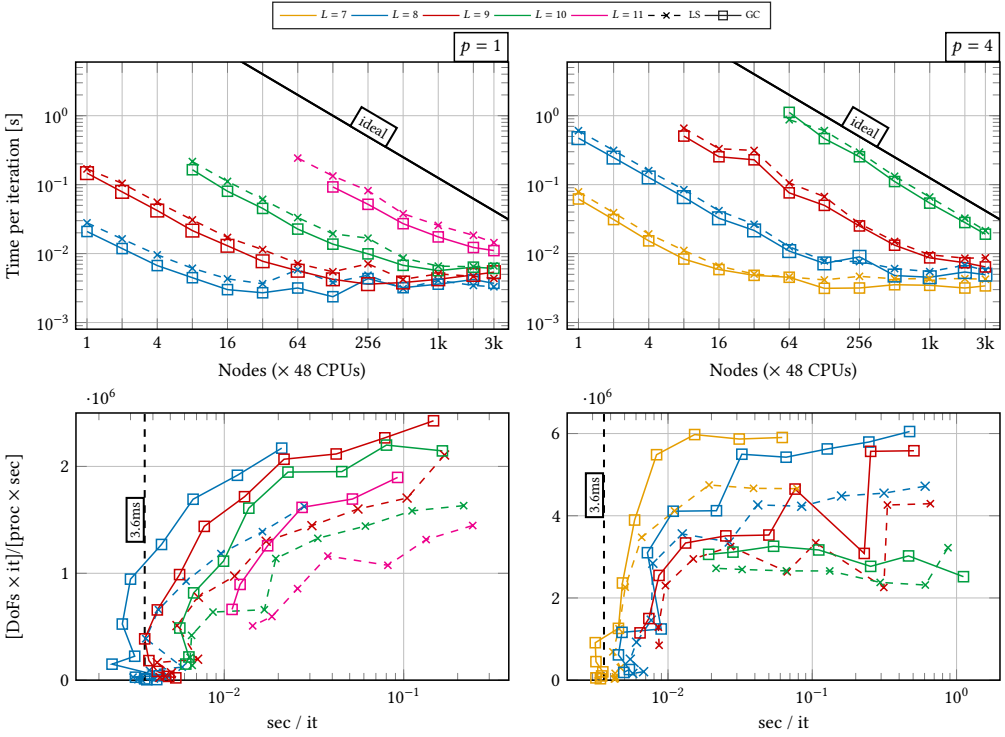


Fig. 13. Strong-scaling comparison of local smoothing (LS) and global coarsening (GC) for octant for $p = 1$ and $p = 4$. The label “3.6ms” indicates the theoretical scaling limit for $L = 9$.

of single-node computations of local smoothing, which normally shows a kink in efficiency at early stages (particularly visible in the shell case, which matches the findings made in [28]). A further observation is that the lines for global coarsening overlap far from the scaling limit, i.e., the throughput is independent of the number of processes and the number of refinements. This is not the case for local smoothing, where the throughput deteriorates with the number of levels, indicating load-balance problems. The simulations with $p = 4$ show similar trends, but the lines are not as smooth, possibly due to the decreased granularity for higher orders.

With (9), one can approximate a latency $\Delta t_{\text{Latency}}^{\text{V-cycle}} = 3.6\text{ms}$ for $L = 9$ and $\Delta t_{\text{Latency}}^{\text{vmult}} = 50\ \mu\text{s}$. This value is visualized with vertical lines in the diagrams. One can see that the approximation indeed matches well for $p = 1$, indicating that the solvers are limited by the latency in these ranges. Due to the similarity between the numbers of communication steps in the case of local smoothing and global coarsening, the minimal times reached are similar. For $p = 4$, the discrepancy between the approximated limit and the measurements is higher; the minimal times reached by global coarsening are smaller, indicating that the latency limit has not been reached yet and load imbalances still have a noticeable effect. For less refinements, e.g., $L = 7$, better correspondence can be observed. Apart from these observations, we would like to emphasize that, in practice, one would run at higher efficiencies (left-top corner in the normalized plot), for which both $p = 1$ and $p = 4$ are dominated by load-imbalance effects.

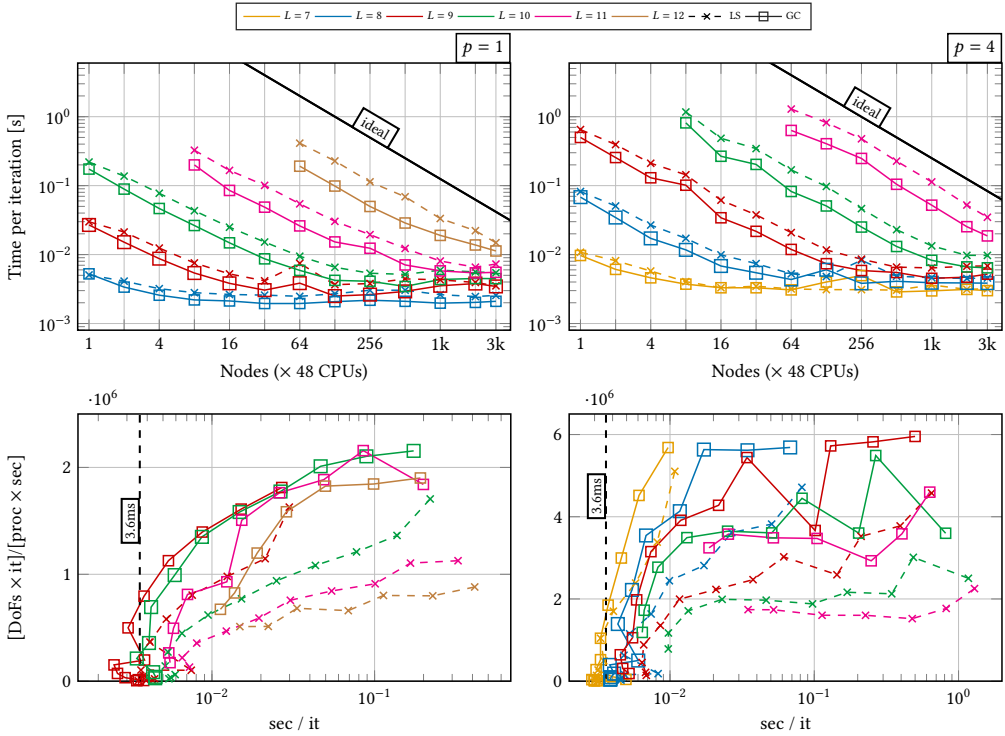


Fig. 14. Strong-scaling comparison of local smoothing (LS) and global coarsening (GC) for shell for $p = 1$ and $p = 4$. The label “3.6ms” indicates the theoretical scaling limit for $L = 9$.

5.5 First-child policy as alternative partitioning strategy for global coarsening

Subsection 5.3 indicates that local smoothing with first-child policy might suffer from deteriorated reduction rates of the maximum number of cells on each level; in particular, there might be processes without any cells, i.e., any work, increasing the critical path, although the vertical efficiency is optimal. In this section, we consider the first-child policy as an alternative for partitioning of the levels for global coarsening.

Figure 15 presents the timings of large-scale octant simulations for 1) local smoothing, 2) global coarsening with default partitioning, and 3) global coarsening with first-child policy for $p = 1/p = 4$. The timings of the latter show similar trends as global coarsening with default partitioning and are lower than the ones of local smoothing. In order to explain this counterintuitive observation, Figure 15 provides the maximum number of cells on each multigrid level of the three approaches for $L = 11$ on 256 nodes. Since global coarsening with first-child policy does not perform any repartitioning, better reduction trends as in the local-smoothing case on all levels can not be expected, however, one can observe that, for the local section of the refinement tree, the number of cells on the finest levels is reduced nearly as well as in the case of the default partitioner, i.e., the parallel workload and the parallel efficiency are not much worse, nonetheless, with higher vertical efficiency. This is not possible for the lower levels, but the behavior of the finest levels dominates the overall trends, since they take the largest time of the computation. One should note that local smoothing has access to the same cells, but simply skips them during smoothing of a

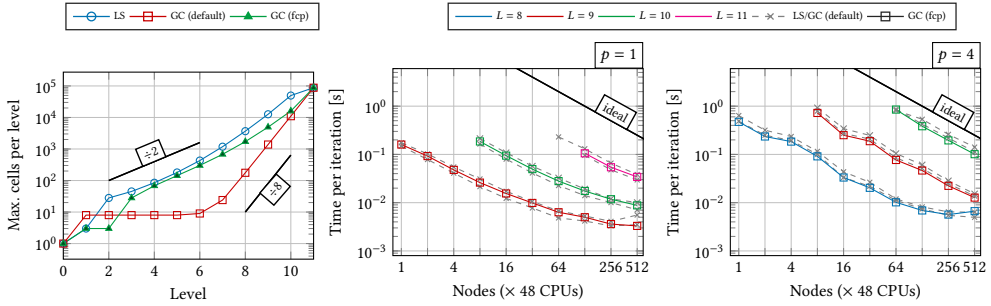


Fig. 15. Left: Maximum number of cells per level ($L = 11$, 256 nodes) and right: strong scaling of global coarsening with first-child policy (fcp) in comparison to local smoothing (LS) and geometric global coarsening (GC) with default policy for octant.

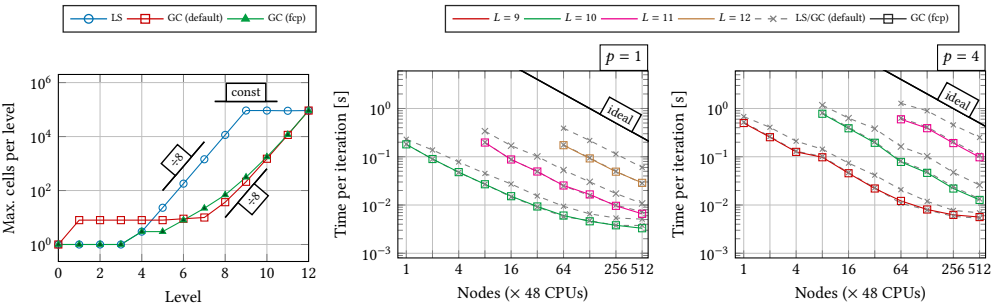


Fig. 16. Left: Maximum number of cells per level ($L = 12$, 256 nodes) and right: strong scaling of global coarsening with first-child policy (fcp) in comparison to local smoothing (LS) and geometric global coarsening (GC) with default policy for shell.

given level, missing the opportunity to reduce the problem size locally. In our experiments, it is not clear whether an optimal reduction of cells is crucial for all configurations: for $p = 1$, global coarsening with default partitioning is faster for most configurations (up to 20%) and, for $p = 4$, global coarsening with first-child policy is faster (up to 10%). We could trace this difference back to the different costs of the transfer, where, for $p = 4$, a reduced data transfer (both within the compute node and across the network) and, for $p = 1$, a better load balance is beneficial.

In the shell case (Figure 16), we observe that both global-coarsening partitioning strategies result in very similar reduction rates, which can be traced back to the fact that both strategies lead to comparable partitionings of the levels (see also Table 5, which shows a very high vertical efficiency of the default partitioning) and—as a consequence—to very similar solution times ($\pm 10\%$). Local smoothing only reduces the maximum number of cells per level optimally once all locally refined cells have been processed and the levels that had been constructed via global refinement have been reached. This case stresses the issue of load imbalances related to reduction rates significantly differing between processes.

6 PERFORMANCE ANALYSIS: P-MULTIGRID

In this section, we consider p -multigrid. The settings are as described in Section 5. The degrees on the levels are obtained by a bisection strategy. On the coarsest level ($p = 1$, fine mesh with hanging nodes), we run a single V-cycle of either a local-smoothing or a global-coarsening geometric

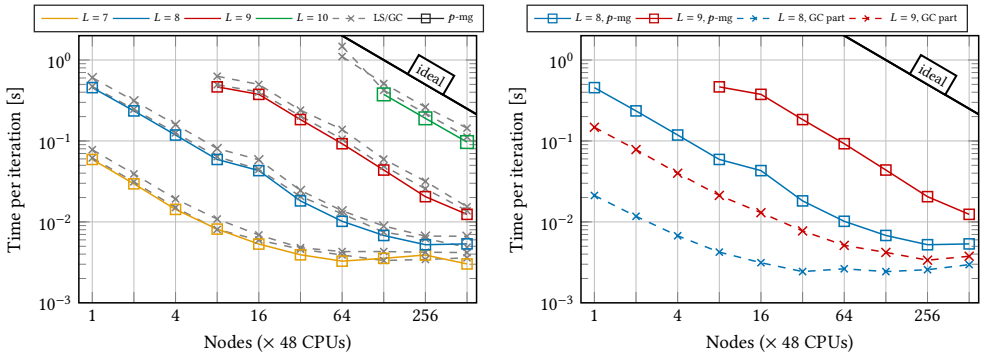


Fig. 17. Strong scaling of p -multigrid for octant for $p = 4$. p -multigrid switches to a global-coarsening coarse-grid solver immediately once linear elements have been reached. Left: Comparison with local smoothing (LS) and geometric global coarsening (GC) for $p = 4$. Right: Comparison with geometric global coarsening (GC) for $p = 1$ (coarse-grid problem of p -multigrid).

multigrid solver in an hp context. In Appendix B, we present a comparison with state-of-the-art AMG solvers [31, 34] as coarse-grid solvers of p -multigrid on 16 nodes. The results show better timings in favor of geometric multigrid, which also turned out to be more robust with a single V-cycle.

Figure 17 presents a strong-scaling comparison of the p -multigrid version of the global-coarsening algorithm with local-smoothing and global-coarsening variants of h -multigrid for the octant case for $p = 4$.⁶ As we use a bisection strategy in the context of p -multigrid, the overall (hp -)multigrid algorithm has two additional levels compared to pure h -multigrid, given the same fine mesh, but with additional levels for $p = 1$ and $p = 2$. In our experiments, we observe that p -multigrid needs at least as many iterations as the pure (global-coarsening) h -multigrid algorithm (with small differences of at most 1). Since this might be related to the chosen problem and might be different on unstructured meshes as well as for problems with high-contrast coefficients and boundary layers, we consider “time per iteration” in the following.

One can see that one cycle of p -multigrid is 10–15% faster than h -multigrid for moderate numbers of processes. The reason for this is that the smoother application on the finest level is equally expensive, however, the transfer between the two finest levels is cheaper: due to the same partitioning of these levels, the data is mainly transferred between cells that are locally owned on both the coarse and the fine level. At the scaling limit (not shown), p -multigrid falls behind h -multigrid regarding performance. This is related to the increased latency due to a higher number of levels.

For the sake of completeness, Figure 17 includes the times spent in the geometric global-coarsening part, i.e., the timings for the V-cycle for $p = 1$ for $L = 8/L = 9$ as the coarse-grid problem of the p -multigrid solver. Due to the $\sim 64\times$ smaller size, its share is negligible for a wide range of nodes, but it becomes noticeable at the scaling limit.

7 APPLICATION: VARIABLE-VISCOSITY STOKES FLOW

We conclude this publication by presenting preliminary results of a practical application from geosciences by integrating the global-coarsening framework into the mantle convection code ASPECT [39, 61] and comparing it against the existing local-smoothing implementation [27].

⁶The raw data for $L = 9$ is provided in Appendix B.

We consider the variable-viscosity Stokes problem

$$\begin{aligned} -\nabla \cdot (2\eta\varepsilon(\mathbf{u})) + \nabla p &= f \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned}$$

with a Q2-Q1 Taylor-Hood discretization of velocity \vec{u} , pressure p , and a given viscosity $\eta(x) > 0$, the symmetric gradient $\varepsilon(\mathbf{u}) = 1/2\nabla\mathbf{u} + 1/2(\nabla\mathbf{u})^T$, and forcing f . The resulting linear system

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix}$$

is solved with a Krylov method (in our tests using IDR(2), see [27]) preconditioned using a block preconditioner

$$P^{-1} = \begin{pmatrix} A & B^T \\ 0 & -S \end{pmatrix}^{-1}$$

where the Schur complement $S = BA^{-1}B^T$ is approximated using a mass matrix weighted by the viscosity. The inverses of the diagonal blocks of A and the Schur complement approximation \hat{S} are each approximated by applying a single V-cycle of geometric multigrid using a Chebyshev smoother of degree 4 and implemented in a matrix-free fashion. Each IDR(2) iteration consists of 3 matrix-vector products and 3 preconditioner applications. Operator and preconditioner are set up to act on the complement of the nullspace containing the constant pressures. For a detailed description of the preconditioner see [27].

We consider a 3D spherical shell benchmark problem called “nsinker_spherical_shell” that is part of ASPECT. A set of 7 heavy sinkers is placed in a spherical shell with inner radius 0.54, outer radius 1.0, and no-flow boundary conditions. The flow is driven by the density difference of the sinkers ($\beta = 10$) and the gravity of magnitude 1 as follows, where c_i are the centers of the n sinkers⁷:

$$\begin{aligned} \Phi(x) &= \prod_{i=1}^n \left[1 - \exp \left(-\delta \max \left[0, |c_i - x| - \frac{\omega}{2} \right]^2 \right) \right], \\ \eta(x) &= \Phi(x)\eta_{\min} + (1 - \Phi(x))\eta_{\max}, \\ f(x) &= (0, 0, \beta(\Phi(x) - 1)), \end{aligned}$$

with $\delta = 200$, $\omega = 0.1$, $\eta_{\min} = 0.1$, and $\eta_{\max} = 10$. The viscosity is evaluated at the quadrature points of each cell on the finest level, averaged using harmonic averaging on each cell, and then interpolated to the coarser multigrid levels using the multigrid transfer operators (see also the function `interpolate_to_mg()` in Figure 6) for use in the multigrid preconditioner.

The initial mesh (consisting of 96 coarse cells, 4 initial refinement steps and a high-order manifold description) is refined adaptively using a gradient jump error estimator (see [56]) of the velocity field roughly doubling the number of unknowns in each step, see Figure 18.

In the following, we compare local smoothing and global coarsening regarding number of iterations, time to solution, and time for a single V-cycle of the A block in Figure 19. Computations are done on TACC Frontera⁸ on 7168 processes (128 nodes with 56 cores each). While iteration numbers are very similar, the simulation with global coarsening is about twice as fast in total. Each V-cycle is up to four times faster, which is not surprising, since the mesh has—for a high number of refinements—a workload efficiency of approximately 20%. The results suggest that the findings regarding iteration numbers and performance obtained in Section 5 for a simple Poisson problem are also applicable for this non-trivial vector-valued problem.

⁷Their locations are given in <https://github.com/peterrum/dealii-multigrid/mantle-convection>.

⁸Using deal.II v9.4.0 with 64bit indices, Intel 19.1.0.20200306 with -O3 and -march=native

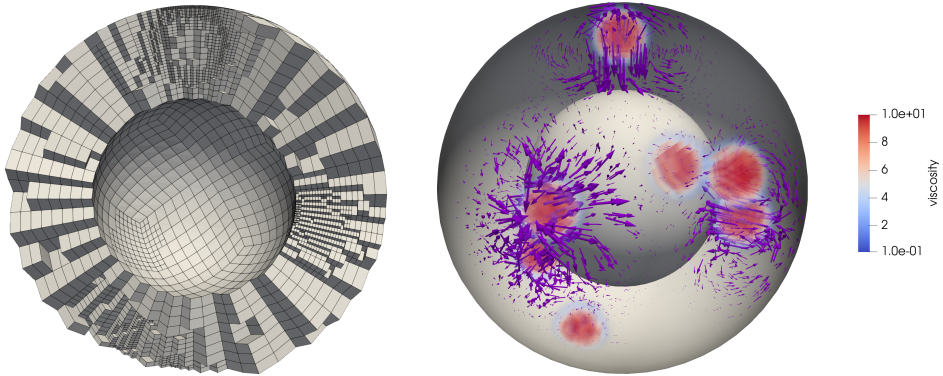


Fig. 18. Visualization of Stokes flow in a spherical shell: Adaptively refined mesh with two global and four additional local refinements (left); Solution shown together with the high-viscosity sinks (red) and velocity vector field in purple (right).

| L | #DoFs [1e6] | local smoothing | | | | | | global coarsening | | | | | |
|-----|-------------|-----------------|-------|-------|-----|-----------|-------------|-------------------|-------|-------|-----|-----------|-------------|
| | | wl-eff | v-eff | h-eff | #it | solve [s] | V-cycle [s] | wl-eff | v-eff | h-eff | #it | solve [s] | V-cycle [s] |
| 5 | 10.0 | 95% | 93% | 60% | 24 | 1.11 | 0.011 | 77% | 67% | 61% | 25 | 0.50 | 0.002 |
| 6 | 20.7 | 52% | 96% | 62% | 25 | 1.54 | 0.006 | 77% | 4% | 63% | 25 | 1.47 | 0.004 |
| 7 | 43.2 | 35% | 97% | 66% | 27 | 3.20 | 0.010 | 86% | 1% | 66% | 25 | 1.55 | 0.006 |
| 8 | 88.0 | 27% | 98% | 69% | 27 | 4.94 | 0.021 | 91% | 1% | 69% | 27 | 3.22 | 0.015 |
| 9 | 178.0 | 22% | 99% | 73% | 28 | 9.81 | 0.046 | 95% | 3% | 73% | 24 | 3.60 | 0.019 |
| 10 | 355.0 | 20% | 100% | 77% | 29 | 15.36 | 0.104 | 97% | 1% | 76% | 26 | 7.84 | 0.035 |
| 11 | 715.7 | 21% | 100% | 80% | 27 | 26.38 | 0.217 | 99% | 2% | 80% | 25 | 11.97 | 0.068 |
| 12 | 1441.4 | 20% | 100% | 83% | 27 | 49.37 | 0.447 | 99% | 2% | 83% | 26 | 21.77 | 0.140 |
| 13 | 2896.7 | 19% | 100% | 86% | 27 | 107.24 | 0.980 | 100% | 4% | 86% | 25 | 35.49 | 0.262 |
| 14 | 5861.9 | 21% | 100% | 89% | 25 | 181.42 | 1.821 | 100% | 5% | 88% | 30 | 78.46 | 0.518 |

Fig. 19. Performance for the Stokes-flow problem with 7168 processes with number of iterations (#it), time of a single multigrid V-cycle of the velocity block, total solution time, and efficiency metrics.

8 SUMMARY AND OUTLOOK

We have compared geometric local-smoothing and geometric global-coarsening multigrid implementations for locally refined meshes. They are based on optimal node-level performance through matrix-free operator evaluation and have been integrated into the same finite-element library (deal.II) in order to enable a fair comparison regarding implementation complexity and performance.

From the implementation point of view, the two multigrid versions are—except for some subtle differences—similar, requiring special treatment of either refinement edges or hanging-node constraints during the application of the smoother and the transfer operator. For the latter, one can usually rely on existing infrastructure for the application of constraints [62, 81]. During transfer, global coarsening needs to transfer between cells that might be refined or not. In order to be able to vectorize the cell-local transfer, we categorize cells within the transfer operator and process categories in one go. In the case of local smoothing, it is possible but not common to repartition the multigrid levels; instead, one uses local strategies for partitioning. For global coarsening, we investigated two partitioning strategies: one that optimally balances workload during smoothing via repartitioning each level and one that minimizes the data to be communicated during the transfer phase.

In a large number of experiments, we have made the observation that, for serial simulations, geometric local smoothing is faster than geometric global coarsening (if the number of iterations is the same), since the total number of cells to perform smoothing on is less and the need for evaluating hanging-node constraints on each level might be noticeable. For parallel simulations, an equally distributed reduction of cells is beneficial. If this is not given, load imbalance is introduced and the critical path of cells, i.e., the time to solution, is increased. In the case of local smoothing, there might be a non-negligible number of processes without cells, naturally introducing load imbalance already on the finest—computationally most expensive—levels. Global coarsening with repartitioning alleviates this problem and reaches optimal parallel workload. It comes, however, with the disadvantage of expensive transfer steps due to permutation of the data. We made the observation that global coarsening with non-optimal partitions for smoothing but with local transfer allows—for the examples considered—to reduce the number of cells on the finest levels surprisingly well, introducing only a small load imbalance. We could not make a definite statement on the choice of partitioning strategies for global coarsening, since it very much depends on whether the transfer or the load imbalance is the bottleneck for the given problem. Regardless of the type of partitioning of the levels, we have shown that global coarsening outperforms local smoothing by up a factor of 2.4 in terms of time to solution for both moderately large and large-scale simulations.

We have also considered polynomial global coarsening (p -multigrid). Its implementation is conceptually analogous to the one of geometric global coarsening with similar involved algorithms. Far from the scaling limit, p -multigrid shows better timings per iteration, which can be explained by the cheaper intergrid transfer. At the scaling limit, the introduction of additional multigrid levels (when combining h and p multigrid) is noticeable and leads to slower times due to latency. Generally, when p - and h -multigrid are used in sequence (hp -multigrid), reducing the degree p first makes sense from a performance point of view due to cheaper transfer as long as the number of iterations does not increase.

Global-coarsening algorithms also give the possibility to easily remove ranks from MPI communicators, allowing to increase the granularity of the problem on each level. Once the problem size can not be reduced by a sufficient factor anymore, one can switch to the coarse-grid solver even on a level with hanging nodes. Furthermore, h and p global coarsening could be done in one go: while this might lead to an overly aggressive coarsening with a consequential deterioration of the convergence rate, the reduced number of levels could result in an improved strong-scaling behavior due to a reduced latency in some cases. The investigation of these topics is deferred to future work.

In this publication, we focused on geometrically refined meshes consisting only of hexahedral shaped cells, where all cells have the same polynomial degree p . However, the presented algorithms work for p - and hp -adaptive problems, where the polynomial degree varies for each cell, as well as for simplex and mixed meshes as the implementation in deal . II [7] shows. Moreover, they are—as demonstrated in [60]—applicable also for auxiliary-space approximation for DG.

ACKNOWLEDGMENTS

The authors acknowledge collaboration with Maximilian Bergbauer, Thomas C. Clevenger, Ivo Dravins, Niklas Fehn, Marc Fehling, and Magdalena Schreter as well as the deal . II community.

This work was supported by the Bayerisches Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen (KONWIHR) through the projects “Performance tuning of high-order discontinuous Galerkin solvers for SuperMUC-NG” and “High-order matrix-free finite element implementations with hybrid parallelization and improved data locality”. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (LRZ, www.lrz.de) through project id pr83te.

Timo Heister was partially supported by the National Science Foundation (NSF) Award DMS-2028346, OAC-2015848, EAR-1925575, by the Computational Infrastructure in Geodynamics initiative (CIG), through the NSF under Award EAR-0949446 and EAR-1550901 and The University of California – Davis, and by Technical Data Analysis, Inc. through US Navy STTR Contract N68335-18-C-0011. Clemson University is acknowledged for generous allotment of compute time on Palmetto cluster.

This research is part of the Frontera computing project at the Texas Advanced Computing Center. Frontera is made possible by National Science Foundation award OAC-1818253.

A APPENDIX TO SECTION 5

Table 6. Number of iterations and time to solution for local smoothing (LS) and global coarsening (GC) for the octant simulation case with 768 processes (16 nodes) with given analytical solution: $u(\vec{x}) = \left(\frac{1}{\alpha\sqrt{2\pi}}\right)^3 \exp(-\|\vec{x} - \vec{x}_0\|/\alpha^2)$ with $\vec{x}_0 = (-0.5, -0.5, -0.5)^T$ and $\alpha = 0.1$. Right-hand-side function $f(\vec{x})$ and inhomogeneous Dirichlet boundary conditions have been selected appropriately.

| L | $p = 1$ | | | | $p = 4$ | | | |
|-----|---------|--------|----|--------|---------|--------|----|--------|
| | LS | | GC | | LS | | GC | |
| | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] |
| 3 | 4 | 1.4e-3 | 4 | 1.0e-3 | 4 | 2.9e-3 | 4 | 2.9e-3 |
| 4 | 4 | 2.8e-3 | 4 | 1.7e-3 | 4 | 4.7e-3 | 4 | 4.2e-3 |
| 5 | 4 | 4.0e-3 | 4 | 2.7e-3 | 4 | 6.9e-3 | 4 | 6.5e-3 |
| 6 | 4 | 5.7e-3 | 4 | 4.2e-3 | 4 | 1.2e-2 | 4 | 1.0e-2 |
| 7 | 4 | 8.2e-3 | 4 | 6.1e-3 | 4 | 2.8e-2 | 4 | 2.4e-2 |
| 8 | 5 | 2.2e-2 | 4 | 1.2e-2 | 5 | 2.8e-1 | 4 | 1.7e-1 |
| 9 | 5 | 8.8e-2 | 4 | 5.2e-2 | 5 | 2.5e+0 | 4 | 1.6e+0 |
| 10 | 5 | 6.1e-1 | 4 | 3.8e-1 | - | - | - | - |

Table 7. Number of iterations and time to solution for local smoothing, global coarsening, and AMG (ML [34]) for the octant simulation case with 768 processes (16 nodes). For local smoothing and global coarsening, results are shown for **Chebyshev smoothing degrees** of $k = 3$ and $k = 6$.

| L | local smoothing | | | | $p = 1$ global coarsening | | | | AMG (ML) | $p = 4$ | | | | | | | | |
|-----|-----------------|--------|---------|--------|---------------------------|--------|---------|--------|----------|---------|----|---------|----|---------|----|---------|----|--------|
| | $k = 3$ | | $k = 6$ | | $k = 3$ | | $k = 6$ | | | $k = 3$ | | $k = 6$ | | $k = 3$ | | $k = 6$ | | |
| | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] |
| 3 | 4 | 1.3e-3 | 3 | 1.4e-3 | 4 | 1.0e-3 | 2 | 7.0e-4 | 1 | 6.0e-3 | 4 | 2.6e-3 | 3 | 2.5e-3 | 4 | 2.5e-3 | 2 | 1.9e-3 |
| 4 | 4 | 2.8e-3 | 3 | 2.9e-3 | 4 | 1.7e-3 | 2 | 1.3e-3 | 1 | 2.5e-2 | 4 | 4.3e-3 | 3 | 4.6e-3 | 4 | 4.0e-3 | 2 | 3.0e-3 |
| 5 | 4 | 4.3e-3 | 3 | 4.6e-3 | 4 | 2.6e-3 | 2 | 2.3e-3 | 5 | 4.2e-3 | 4 | 6.1e-3 | 3 | 6.6e-3 | 3 | 4.9e-3 | 2 | 5.1e-3 |
| 6 | 4 | 5.8e-3 | 3 | 6.4e-3 | 4 | 4.3e-3 | 2 | 3.4e-3 | 8 | 1.5e-2 | 4 | 9.9e-3 | 3 | 1.1e-2 | 3 | 7.2e-3 | 2 | 7.1e-3 |
| 7 | 4 | 8.2e-3 | 3 | 9.3e-3 | 3 | 4.5e-3 | 2 | 4.9e-3 | 6 | 1.2e-2 | 4 | 2.7e-2 | 3 | 3.0e-2 | 3 | 1.8e-2 | 2 | 1.7e-2 |
| 8 | 4 | 1.7e-2 | 3 | 2.1e-2 | 3 | 9.1e-3 | 2 | 1.0e-2 | 6 | 3.0e-2 | 4 | 2.3e-1 | 3 | 2.6e-1 | 3 | 1.4e-1 | 2 | 1.4e-1 |
| 9 | 4 | 7.1e-2 | 3 | 8.6e-2 | 3 | 3.8e-2 | 2 | 4.3e-2 | 7 | 2.1e-1 | 4 | 2.0e+0 | 3 | 2.3e+0 | 3 | 1.2e+0 | 2 | 1.3e+0 |
| 10 | 4 | 5.0e-1 | 3 | 6.1e-1 | 2 | 1.9e-1 | 2 | 3.0e-1 | 7 | 1.6e+0 | - | - | - | - | - | - | - | - |

Table 8. Number of iterations and time to solution for local smoothing (LS) and global coarsening (GC) for the octant simulation case with 768 processes (16 nodes). All operations in the outer CG solver are run with double-precision floating-point numbers and in the multigrid V-cycle with the following **multigrid number types**: single- or double-precision floating-point numbers.

| L | double | | | | | | | | | | | | float | | | | | |
|----|---------|--------|----|--------|----|--------|---------|--------|----|--------|----|--------|---------|--------|----|---------|--|--|
| | $p = 1$ | | | | | | $p = 4$ | | | | | | $p = 1$ | | | $p = 4$ | | |
| | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | | |
| 3 | 4 | 1.3e-3 | 4 | 9.0e-4 | 4 | 2.6e-3 | 4 | 2.7e-3 | 4 | 1.3e-3 | 4 | 1.0e-3 | 4 | 2.4e-3 | 4 | 2.4e-3 | | |
| 4 | 4 | 2.8e-3 | 4 | 1.6e-3 | 4 | 4.5e-3 | 4 | 4.1e-3 | 4 | 2.7e-3 | 4 | 1.7e-3 | 4 | 4.1e-3 | 4 | 4.1e-3 | | |
| 5 | 4 | 4.0e-3 | 4 | 2.6e-3 | 4 | 6.6e-3 | 3 | 5.1e-3 | 4 | 4.0e-3 | 4 | 2.6e-3 | 4 | 6.1e-3 | 3 | 4.8e-3 | | |
| 6 | 4 | 7.8e-3 | 4 | 4.0e-3 | 4 | 1.1e-2 | 3 | 9.6e-3 | 4 | 6.0e-3 | 4 | 4.1e-3 | 4 | 1.1e-2 | 3 | 7.3e-3 | | |
| 7 | 4 | 8.9e-3 | 3 | 4.9e-3 | 4 | 3.8e-2 | 3 | 2.7e-2 | 4 | 8.3e-3 | 3 | 4.8e-3 | 4 | 2.7e-2 | 3 | 1.8e-2 | | |
| 8 | 4 | 2.1e-2 | 3 | 1.0e-2 | 4 | 4.0e-1 | 3 | 2.4e-1 | 4 | 1.7e-2 | 3 | 9.2e-3 | 4 | 2.3e-1 | 3 | 1.4e-1 | | |
| 9 | 4 | 9.9e-2 | 3 | 5.1e-2 | 4 | 3.2e+0 | 3 | 1.9e+0 | 4 | 7.1e-2 | 3 | 3.9e-2 | 4 | 2.0e+0 | 3 | 1.2e+0 | | |
| 10 | 4 | 7.7e-1 | 2 | 2.8e-1 | - | - | - | - | 4 | 5.0e-1 | 2 | 1.9e-1 | - | - | - | - | | |

Table 9. Number of iterations for local smoothing (LS) and global coarsening (GC) for the octant simulation with 768 processes (16 nodes) and for different **global relative solver tolerances**.

| L | 10^{-4} | | | | | | | | | | | | | | | | 10^{-6} | | | | 10^{-8} | | | | 10^{-10} | | | |
|----|-----------|----|---------|----|---------|----|---------|----|---------|----|---------|----|---------|----|---------|----|-----------|----|---------|----|-----------|--|--|--|------------|--|--|--|
| | $p = 1$ | | $p = 4$ | | $p = 1$ | | $p = 4$ | | $p = 1$ | | $p = 4$ | | $p = 1$ | | $p = 4$ | | $p = 1$ | | $p = 4$ | | | | | | | | | |
| | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | LS | GC | | | | | | | | |
| 3 | 4 | 4 | 4 | 4 | 4 | 6 | 5 | 6 | 6 | 7 | 6 | 8 | 7 | 9 | 8 | 10 | 9 | 10 | 9 | | | | | | | | | |
| 4 | 4 | 4 | 4 | 4 | 4 | 6 | 6 | 6 | 5 | 8 | 7 | 8 | 7 | 10 | 9 | 10 | 9 | 10 | 9 | | | | | | | | | |
| 5 | 4 | 4 | 4 | 4 | 3 | 6 | 5 | 6 | 5 | 8 | 7 | 8 | 7 | 10 | 8 | 10 | 8 | 10 | 8 | | | | | | | | | |
| 6 | 4 | 4 | 4 | 3 | 6 | 5 | 6 | 5 | 8 | 7 | 8 | 6 | 10 | 8 | 10 | 8 | 10 | 8 | 8 | | | | | | | | | |
| 7 | 4 | 3 | 4 | 3 | 6 | 5 | 6 | 4 | 8 | 6 | 8 | 6 | 10 | 8 | 10 | 8 | 10 | 8 | 8 | | | | | | | | | |
| 8 | 4 | 3 | 4 | 3 | 6 | 4 | 6 | 4 | 8 | 6 | 8 | 6 | 9 | 8 | 10 | 7 | 10 | 7 | 7 | | | | | | | | | |
| 9 | 4 | 3 | 4 | 3 | 6 | 4 | 6 | 4 | 8 | 6 | 8 | 6 | 9 | 7 | 10 | 7 | 10 | 7 | 7 | | | | | | | | | |
| 10 | 4 | 2 | - | - | 6 | 4 | - | - | 8 | 6 | - | - | 9 | 7 | - | - | - | - | - | | | | | | | | | |

Table 10. Time to solution for global coarsening for the octant simulation with 768 processes (16 nodes) and different cell weights for cells near hanging nodes compared to regular cells.

| L/w | $p = 1$ | | | | | $p = 4$ | | | | |
|-----|---------|--------|--------|--------|--------|---------|--------|--------|--------|--------|
| | 1.0 | 1.5 | 2.0 | 2.5 | 3.0 | 1.0 | 1.5 | 2.0 | 2.5 | 3.0 |
| 3 | 1.1e-3 | 1.0e-3 | 1.0e-3 | 1.0e-3 | 1.0e-3 | 2.7e-3 | 2.6e-3 | 2.6e-3 | 2.5e-3 | 2.5e-3 |
| 4 | 1.8e-3 | 1.8e-3 | 1.8e-3 | 1.7e-3 | 1.7e-3 | 3.9e-3 | 3.9e-3 | 3.9e-3 | 3.9e-3 | 3.9e-3 |
| 5 | 2.5e-3 | 2.7e-3 | 2.7e-3 | 2.6e-3 | 2.8e-3 | 4.4e-3 | 4.7e-3 | 4.6e-3 | 4.5e-3 | 4.6e-3 |
| 6 | 4.0e-3 | 4.1e-3 | 4.1e-3 | 4.2e-3 | 4.1e-3 | 7.3e-3 | 7.1e-3 | 7.0e-3 | 7.1e-3 | 7.0e-3 |
| 7 | 4.8e-3 | 4.8e-3 | 4.7e-3 | 4.4e-3 | 5.4e-3 | 1.8e-2 | 1.7e-2 | 1.6e-2 | 1.7e-2 | 1.8e-2 |
| 8 | 1.1e-2 | 1.0e-2 | 9.1e-3 | 8.5e-3 | 8.2e-3 | 1.1e-1 | 1.0e-1 | 1.0e-1 | 1.0e-1 | 1.0e-1 |
| 9 | 5.3e-2 | 4.4e-2 | 3.8e-2 | 3.5e-2 | 3.5e-2 | 9.0e-1 | 8.2e-1 | 7.9e-1 | 7.8e-1 | 7.9e-1 |
| 10 | 2.3e-1 | 1.9e-1 | 1.7e-1 | 1.5e-1 | 1.5e-1 | - | - | - | - | - |

Table 11. Time to solution for global coarsening for the octant simulation with **24,576 processes** (512 nodes) and **different cell weights** for cells near hanging nodes compared to regular cells.

| L/w | $p = 1$ | | | | | $p = 4$ | | | | |
|-------|---------|--------|--------|--------|--------|---------|--------|--------|--------|--------|
| | 1.0 | 1.5 | 2.0 | 2.5 | 3.0 | 1.0 | 1.5 | 2.0 | 2.5 | 3.0 |
| 3 | 1.4e-3 | 1.3e-3 | 1.4e-3 | 1.6e-3 | 1.4e-3 | 3.1e-3 | 3.0e-3 | 3.0e-3 | 2.9e-3 | 2.9e-3 |
| 4 | 2.1e-3 | 2.2e-3 | 2.1e-3 | 2.0e-3 | 2.1e-3 | 4.3e-3 | 4.4e-3 | 4.3e-3 | 4.3e-3 | 4.3e-3 |
| 5 | 2.9e-3 | 2.9e-3 | 2.9e-3 | 2.9e-3 | 2.9e-3 | 4.6e-3 | 5.0e-3 | 4.6e-3 | 5.1e-3 | 4.7e-3 |
| 6 | 4.5e-3 | 4.1e-3 | 4.1e-3 | 4.2e-3 | 4.1e-3 | 7.0e-3 | 7.0e-3 | 7.1e-3 | 6.9e-3 | 6.6e-3 |
| 7 | 5.9e-3 | 4.8e-3 | 5.7e-3 | 5.1e-3 | 5.6e-3 | 1.1e-2 | 1.1e-2 | 1.1e-2 | 1.1e-2 | 1.0e-2 |
| 8 | 7.6e-3 | 9.0e-3 | 7.9e-3 | 9.2e-3 | 8.9e-3 | 1.5e-2 | 1.5e-2 | 1.6e-2 | 1.5e-2 | 1.4e-2 |
| 9 | 1.2e-2 | 1.1e-2 | 1.1e-2 | 1.1e-2 | 1.0e-2 | 4.9e-2 | 4.2e-2 | 4.3e-2 | 4.2e-2 | 4.2e-2 |
| 10 | 1.8e-2 | 1.5e-2 | 1.4e-2 | 1.3e-2 | 1.3e-2 | 3.7e-1 | 3.4e-1 | 3.4e-1 | 3.4e-1 | 3.4e-1 |
| 11 | 7.6e-2 | 6.8e-2 | 5.7e-2 | 5.4e-2 | 5.3e-2 | - | - | - | - | - |

Table 12. Number of iterations and time to solution for local smoothing (LS) and global coarsening (GC) for a uniformly refined mesh of a cube (**without hanging nodes**) with 768 processes (16 nodes).

| L | $p = 1$ | | | | $p = 4$ | | | |
|-----|---------|---------|-------|---------|---------|---------|-------|---------|
| | LS | | GC | | LS | | GC | |
| | $\#i$ | $t[s]$ | $\#i$ | $t[s]$ | $\#i$ | $t[s]$ | $\#i$ | $t[s]$ |
| 3 | 4 | 1.80e-3 | 4 | 1.80e-3 | 4 | 2.90e-3 | 4 | 3.20e-3 |
| 4 | 4 | 2.50e-3 | 4 | 2.40e-3 | 4 | 3.90e-3 | 4 | 4.00e-3 |
| 5 | 4 | 3.30e-3 | 4 | 3.20e-3 | 4 | 5.90e-3 | 4 | 6.40e-3 |
| 6 | 4 | 4.60e-3 | 4 | 5.40e-3 | 4 | 1.61e-2 | 4 | 1.69e-2 |
| 7 | 4 | 9.00e-3 | 4 | 9.00e-3 | 4 | 1.29e-1 | 4 | 1.29e-1 |
| 8 | 4 | 4.18e-2 | 4 | 4.21e-2 | 4 | 1.11e+0 | 4 | 1.10e+0 |
| 9 | 4 | 3.00e-1 | 4 | 2.98e-1 | - | - | - | - |

B APPENDIX TO SECTION 6

Table 13. Number of iterations and time to solution for local smoothing (LS), global coarsening (GC), and AMG (ML [34] and BoomerAMG [31]) as **coarse-grid solver** of p -multigrid for the octant simulation with 768 processes (16 nodes) for $p = 4$. For AMG, different numbers of V-cycles $\#v$ are investigated. AMG parameters used are shown in Appendix C.

| L | LS | | GC | | AMG (ML) | | | | | | | | AMG (BoomerAMG) | |
|-----|-------|---------|-------|---------|----------|---------|---------|---------|---------|---------|---------|---------|-----------------|---------|
| | $\#i$ | $t[s]$ | $\#i$ | $t[s]$ | $\#v=1$ | | $\#v=2$ | | $\#v=3$ | | $\#v=4$ | | $\#v=1$ | |
| | | | | | $\#i$ | $t[s]$ | $\#i$ | $t[s]$ | $\#i$ | $t[s]$ | $\#i$ | $t[s]$ | $\#i$ | $t[s]$ |
| 3 | 4 | 2.60e-3 | 4 | 2.20e-3 | 4 | 1.70e-3 | 4 | 2.00e-3 | 4 | 2.20e-3 | 4 | 2.50e-3 | 4 | 1.70e-3 |
| 4 | 4 | 5.10e-3 | 4 | 3.90e-3 | 4 | 5.40e-3 | 4 | 8.30e-3 | 4 | 1.11e-2 | 4 | 1.39e-2 | 4 | 4.90e-3 |
| 5 | 4 | 7.60e-3 | 4 | 6.10e-3 | 4 | 1.17e-2 | 4 | 1.95e-2 | 4 | 2.74e-2 | 4 | 3.52e-2 | 4 | 1.17e-2 |
| 6 | 4 | 1.11e-2 | 4 | 9.20e-3 | 5 | 2.37e-2 | 4 | 3.24e-2 | 4 | 4.45e-2 | 4 | 5.96e-2 | 5 | 3.60e-2 |
| 7 | 4 | 2.35e-2 | 4 | 2.11e-2 | 6 | 4.70e-2 | 5 | 5.47e-2 | 4 | 5.59e-2 | 4 | 6.72e-2 | 6 | 1.04e-1 |
| 8 | 4 | 1.81e-1 | 4 | 1.72e-1 | 7 | 3.26e-1 | 5 | 2.65e-1 | 4 | 2.37e-1 | 4 | 2.63e-1 | 7 | 4.76e-1 |
| 9 | 4 | 1.53e+0 | 4 | 1.51e+0 | 9 | 3.54e+0 | 7 | 2.97e+0 | 6 | 2.75e+0 | 5 | 2.46e+0 | 8 | 3.83e+0 |

Table 14. Number of iterations and time of a single iteration for h -multigrid (local smoothing (LS), global coarsening (GC)) and p -multigrid (local smoothing or global coarsening as coarse-grid solver) for $L = 9$ and $p = 4$.

| #nodes | h -mg | | | | p -mg | | | |
|--------|---------|---------|----|---------|---------|---------|----|---------|
| | LS | | GC | | LS | | GC | |
| | #i | t[s] | #i | t[s] | #i | t[s] | #i | t[s] |
| 8 | 4 | 6.30e-1 | 3 | 4.95e-1 | 4 | 4.75e-1 | 4 | 4.66e-1 |
| 16 | 4 | 4.99e-1 | 3 | 4.03e-1 | 4 | 3.83e-1 | 4 | 3.77e-1 |
| 32 | 4 | 2.40e-1 | 3 | 2.03e-1 | 4 | 1.87e-1 | 4 | 1.84e-1 |
| 64 | 4 | 1.39e-1 | 3 | 1.05e-1 | 4 | 9.57e-2 | 4 | 9.25e-2 |
| 128 | 4 | 5.97e-2 | 3 | 4.90e-2 | 4 | 4.49e-2 | 4 | 4.37e-2 |
| 256 | 4 | 3.17e-2 | 3 | 2.57e-2 | 4 | 2.35e-2 | 4 | 2.05e-2 |
| 512 | 4 | 1.54e-2 | 3 | 1.37e-2 | 4 | 1.38e-2 | 4 | 1.24e-2 |

C AMG PARAMETERS

Listing 1. ML [34] (Trilinos 12.12.1)

```
Teuchos::ParameterList parameter_list;
ML_Epetra::SetDefaults("SA", parameter_list);

parameter_list.set("smoother: type", "ILU");
parameter_list.set("coarse: type", coarse_type);
parameter_list.set("initialize random seed", true);
parameter_list.set("smoother: sweeps", 1);
parameter_list.set("cycle applications", 2);
parameter_list.set("prec type", "MGV");
parameter_list.set("smoother: Chebyshev alpha", 10.);
parameter_list.set("smoother: ifpack overlap", 0);
parameter_list.set("aggregation: threshold", 1e-4);
parameter_list.set("coarse: max size", 2000);
```

Listing 2. BoomerAMG [31] (PETSc 3.14.5)

```
PCHYPRESetType(pc, "boomeramg");

set_option_value("-pc_hypre_boomeramg_agg_nl", "2");
set_option_value("-pc_hypre_boomeramg_max_row_sum", "0.9");
set_option_value("-pc_hypre_boomeramg_strong_threshold", "0.5");
set_option_value("-pc_hypre_boomeramg_relax_type_up", "SOR/Jacobi");
set_option_value("-pc_hypre_boomeramg_relax_type_down", "SOR/Jacobi");
set_option_value("-pc_hypre_boomeramg_relax_type_coarse", "Gaussian-elimination");
set_option_value("-pc_hypre_boomeramg_grid_sweeps_coarse", "1");
set_option_value("-pc_hypre_boomeramg_tol", "0.0");
set_option_value("-pc_hypre_boomeramg_max_iter", "2");
```

REFERENCES

- [1] Mark Adams. 2002. Evaluation of three unstructured multigrid methods on 3D finite element problems in solid mechanics. *Internat. J. Numer. Methods Engrg.* 55, 5 (2002), 519–534.
- [2] Mark Adams, Marian Brezina, Jonathan Hu, and Ray Tuminaro. 2003. Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *J. Comput. Phys.* 188, 2 (2003), 593–610.
- [3] Mark Adams, Phillip Colella, Daniel T. Graves, Jeff N. Johnson, Noel D. Keen, Terry J. Ligocki, Daniel F. Martin, Peter W. McCorquodale, David Modiano, Peter O. Schwartz, T.D. Sternberg, and Brian Van Straalen. 2014. Chombo software package for AMR applications design document. *Lawrence Berkeley National Laboratory Technical Report LBNL-6616E* (2014).
- [4] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Veselin Dobrev, Johann Dudouit, Aaron Fisher, Tzanio Kolev, Will Pazner, Mark Stowell, Vladimir Tomov, Ido Akkerman, Johann Dahm, David Medina, and Stefano Zampini. 2021. MFEM: A modular finite element methods library. *Computers & Mathematics with Applications* 81 (2021), 42–74. DOI: <http://dx.doi.org/10.1016/j.camwa.2020.06.009>

- [5] Paola F. Antonietti, Marco Sarti, Marco Verani, and Ludmil T. Zikatanov. 2017. A uniform additive Schwarz preconditioner for high-order discontinuous Galerkin approximations of elliptic problems. *Journal of Scientific Computing* 70, 2 (2017), 608–630.
- [6] Daniel Arndt, Wolfgang Bangerth, Bruno Blais, Thomas C. Clevenger, Marc Fehling, Alexander V. Grayver, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Pelteret, Reza Rastak, Ignacio Tomas, Bruno Turcksin, Zhuoran Wang, and David Wells. 2020. The deal.II library, Version 9.2. *Journal of Numerical Mathematics* 28, 3 (2020), 131–146. DOI : <http://dx.doi.org/10.1515/jnma-2020-0043>
- [7] Daniel Arndt, Wolfgang Bangerth, Bruno Blais, Marc Fehling, Rene Gassmüller, Timo Heister, Luca Heltai, Uwe Köcher, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Pelteret, Sebastian Proell, Konrad Simon, Bruno Turcksin, David Wells, and Jiaqi Zhang. 2021a. The deal.II library, Version 9.3. *Journal of Numerical Mathematics* 29, 3 (2021), 171–186. DOI : <http://dx.doi.org/10.1515/jnma-2021-0081>
- [8] Daniel Arndt, Wolfgang Bangerth, Denis Davydov, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Jean-Paul Pelteret, Bruno Turcksin, and David Wells. 2021b. The deal.II finite element library: Design, features, and insights. *Computers & Mathematics with Applications* 81 (2021), 407–422. DOI : <http://dx.doi.org/10.1016/j.camwa.2020.02.022>
- [9] Daniel Arndt, Wolfgang Bangerth, Marco Feder, Marc Fehling, Rene Gassmüller, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Pelteret, Simon Sticko, Bruno Turcksin, and David Wells. 2022. The deal. II Library, Version 9.4. *Journal of Numerical Mathematics* in press (2022). DOI : <http://dx.doi.org/10.1515/jnma-2022-0054>
- [10] Daniel Arndt, Niklas Fehn, Guido Kanschat, Katharina Kormann, Martin Kronbichler, Peter Munch, Wolfgang A. Wall, and Julius Witte. 2020. ExaDG: High-Order Discontinuous Galerkin for the Exa-Scale. In *Software for Exascale Computing - SPPEXA 2016-2019*, Hans-Joachim Bungartz, Severin Reiz, Benjamin Uekermann, Philipp Neumann, and Wolfgang E. Nagel (Eds.). Springer International Publishing, Cham, 189–224.
- [11] Douglas Arnold, Richard Falk, and Ragnar Winther. 1997. Preconditioning in H(div) and applications. *Mathematics of computation* 66, 219 (1997), 957–984.
- [12] Harold L. Atkins and Brian T. Helenbrook. 2005. Numerical Evaluation of P-Multigrid Method for the Solution of Discontinuous Galerkin Discretizations of Diffusive Equations. In *17th AIAA Computational Fluid Dynamics Conference. 17th AIAA Computational Fluid Dynamics Conference*; 6–9 Jun. 2005, American Institute of Aeronautics and Astronautics, Toronto, Ontario Canada, 1–11. DOI : <http://dx.doi.org/10.2514/6.2005-5110>
- [13] Eugenio Aulisa, Sara Calandrini, and Giacomo Capodaglio. 2018. An improved multigrid algorithm for n-irregular meshes with subspace correction smoother. *Computers & Mathematics with Applications* 76, 3 (2018), 620–632.
- [14] Eugenio Aulisa, Giacomo Capodaglio, and Guoyi Ke. 2019. Construction of h-refined continuous finite element spaces with arbitrary hanging node configurations and applications to multigrid algorithms. *SIAM Journal on Scientific Computing* 41, 1 (2019), A480–A507.
- [15] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. 2011. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Software* 38, 2 (2011), 14:1–28. DOI : <http://dx.doi.org/10.1145/2049673.2049678>
- [16] Wolfgang Bangerth and Oliver Kayser-Herold. 2009. Data structures and requirements for hp finite element software. *ACM Trans. Math. Software* 36, 1 (2009), 4:1–31.
- [17] Francesco Bassi, Antonio Ghidoni, Stefano Rebay, and P Tesini. 2009. High-order accurate p-multigrid discontinuous Galerkin solution of the Euler equations. *International journal for numerical methods in fluids* 60, 8 (2009), 847–865.
- [18] Francesco Bassi and Stefano Rebay. 2003. Numerical Solution of the Euler Equations with a Multiorder Discontinuous Finite Element Method. In *Computational Fluid Dynamics 2002*, S.W. Armfield (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 199–204. DOI : http://dx.doi.org/10.1007/978-3-642-59334-5_27
- [19] Peter Bastian and Christian Wieners. 2006. Multigrid methods on adaptively refined grids. *Computing in Science & Engineering* 8, 6 (2006), 44–54.
- [20] Roland Becker and Malte Braack. 2000. Multigrid techniques for finite elements on locally refined meshes. *Numerical linear algebra with applications* 7, 6 (2000), 363–379.
- [21] Roland Becker, Malte Braack, and Thomas Richter. 2007. Parallel multigrid on locally refined meshes. In *Reactive Flows, Diffusion and Transport*. Springer, 77–92.
- [22] Marco L. Bittencourt, Craig C. Douglas, and Raúl A. Feijóo. 2001. Nonnested multigrid methods for linear problems. *Numerical Methods for Partial Differential Equations: An International Journal* 17, 4 (2001), 313–331.
- [23] James H. Bramble, Joseph E. Pasciak, and Jinchao Xu. 1991. The analysis of multigrid algorithms with nonnested spaces or noninherited quadratic forms. *Math. Comp.* 56, 193 (1991), 1–34.
- [24] Achi Brandt. 1977. Multi-level adaptive solutions to boundary-value problems. *Mathematics of computation* 31, 138 (1977), 333–390.
- [25] Jed Brown. 2010. Efficient nonlinear solvers for nodal high-order finite elements in 3D. *Journal of Scientific Computing* 45, 1-3 (2010), 48–63. DOI : <http://dx.doi.org/10.1007/s10915-010-9396-8>

- [26] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. 2011. p4est : Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing* 33, 3 (2011), 1103–1133. DOI : <http://dx.doi.org/10.1137/100791634>
- [27] Thomas C. Clevenger and Timo Heister. 2021. Comparison between algebraic and matrix-free geometric multigrid for a Stokes problem on adaptive meshes with variable viscosity. *Numerical Linear Algebra with Applications* 28, 5 (2021), e2375. DOI : <http://dx.doi.org/10.1002/nla.2375>
- [28] Thomas C. Clevenger, Timo Heister, Guido Kanschat, and Martin Kronbichler. 2020. A flexible, parallel, adaptive geometric multigrid method for FEM. *ACM Trans. Math. Software* 47, 1 (2020), 7:1–27.
- [29] David Darmofal and Krzysztof Fidkowski. 2004. Development of a Higher-Order Solver for Aerodynamic Applications. In *42nd AIAA Aerospace Sciences Meeting and Exhibit*. American Institute of Aeronautics and Astronautics, Reston, Virginia, 1–12. DOI : <http://dx.doi.org/10.2514/6.2004-436>
- [30] Michel O. Deville, Paul F. Fischer, and Ernest H. Mund. 2002. *High-order methods for incompressible fluid flow*. Vol. 9. Cambridge University Press.
- [31] Robert D Falgout, Jim E Jones, and Ulrike Meier Yang. 2006. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical solution of partial differential equations on parallel computers*. Springer, 267–294.
- [32] Niklas Fehn, Peter Munch, Wolfgang A. Wall, and Martin Kronbichler. 2020. Hybrid multigrid methods for high-order discontinuous Galerkin discretizations. *J. Comput. Phys.* 415 (2020), 109538. DOI : <http://dx.doi.org/10.1016/j.jcp.2020.109538>
- [33] Krzysztof J. Fidkowski, Todd A. Oliver, James Lu, and David L. Darmofal. 2005. p-Multigrid solution of high-order discontinuous Galerkin discretizations of the compressible Navier-Stokes equations. *J. Comput. Phys.* 207, 1 (2005), 92–113. DOI : <http://dx.doi.org/10.1016/j.jcp.2005.01.005>
- [34] Michael W Gee, Christopher M Siefert, Jonathan J Hu, Ray S Tuminaro, and Marzio G Sala. 2006. *ML 5.0 smoothed aggregation user's guide*. Technical Report. Technical Report SAND2006-2649, Sandia National Laboratories.
- [35] A. Ghidoni, A. Colombo, F. Bassi, and S. Rebay. 2014. Efficient p -multigrid discontinuous Galerkin solver for complex viscous flows on stretched grids. *International Journal for Numerical Methods in Fluids* 75, 2 (2014), 134–154. DOI : <http://dx.doi.org/10.1002/flid.3888>
- [36] Amir Gholami, Dhairya Malhotra, Hari Sundar, and George Biros. 2016. FFT, FMM, or Multigrid? A comparative Study of State-Of-the-Art Poisson Solvers for Uniform and Nonuniform Grids in the Unit Cube. *SIAM Journal on Scientific Computing* 38, 3 (2016), C280–C306. DOI : <http://dx.doi.org/10.1137/15M1010798>
- [37] Xian-Zhong Guo and I. Norman Katz. 1998. Performance enhancement of the multi-p preconditioner. *Computers & Mathematics with Applications* 36, 4 (1998), 1–8.
- [38] Xian-Zhong Guo and I. Norman Katz. 2000. A Parallel Multi-p Method. *Computers & Mathematics with Applications* 39, 9-10 (2000), 115–123.
- [39] Timo Heister, Juliane Dannberg, Rene Gassmüller, and Wolfgang Bangerth. 2017. High Accuracy Mantle Convection Simulation through Modern Numerical Methods. II: Realistic Models and Problems. *Geophysical Journal International* 210, 2 (2017), 833–851. DOI : <http://dx.doi.org/10.1093/gji/ggx195>
- [40] Brian T. Helenbrook and Harold L. Atkins. 2006. Application of p-Multigrid to Discontinuous Galerkin Formulations of the Poisson Equation. *AIAA Journal* 44, 3 (2006), 566–575. DOI : <http://dx.doi.org/10.2514/1.15497>
- [41] Brian T. Helenbrook and Harold L. Atkins. 2008a. Solving Discontinuous Galerkin Formulations of Poisson's Equation using Geometric and p Multigrid. *AIAA Journal* 46, 4 (2008), 894–902. DOI : <http://dx.doi.org/10.2514/1.31163>
- [42] Brian T Helenbrook and Harold L. Atkins. 2008b. Solving discontinuous Galerkin formulations of Poisson's equation using geometric and p multigrid. *AIAA journal* 46, 4 (2008), 894–902.
- [43] Brian T. Helenbrook and Brendan S. Mascarenhas. 2016. Analysis of Implicit Time-Advancing p-Multigrid Schemes for Discontinuous Galerkin Discretizations of the Euler Equations. In *46th AIAA Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics, Washington, D.C. DOI : <http://dx.doi.org/10.2514/6.2016-3494>
- [44] Brian T. Helenbrook, Dimitri Mavriplis, and Harold L. Atkins. 2003. Analysis of “p”-Multigrid for Continuous and Discontinuous Finite Element Discretizations. In *16th AIAA Computational Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics, Orlando, Florida. DOI : <http://dx.doi.org/10.2514/6.2003-3989>
- [45] Magnus Rudolph Hestenes and Eduard Stiefel. 1952. *Methods of conjugate gradients for solving linear systems*. Vol. 49. NBS Washington, DC.
- [46] Koen Hillewaert, P Wesseling, E Oñate, J Périaux, Jean-François Remacle, Nicolas Cheveaugeon, Paul-Emile Bernard, and Philippe Geuzaine. 2006. Analysis of a hybrid p-multigrid method for the discontinuous Galerkin discretisation of the Euler equations. In *Proceedings of the European Conference on Computational Fluid Dynamics*, Pieter Wesseling (Ed.). ECCOMAS CFD 2006, Egmond aan Zee, Netherlands. DOI : <http://dx.doi.org/90-9020970-0>
- [47] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. 2010. Scalable communication protocols for dynamic sparse data exchange. *ACM Sigplan Notices* 45, 5 (2010), 159–168.


- [48] Ning Hu, Xian-Zhong Guo, and I. Norman Katz. 1997. Multi-p Preconditioners. *SIAM Journal on Scientific Computing* 18, 6 (1997), 1676–1697.
- [49] Ning Hu and I. Norman Katz. 1995. Multi-P Methods: Iterative Algorithms for the P-Version of the Finite Element Analysis. *SIAM Journal on Scientific Computing* 16, 6 (1995), 1308–1332.
- [50] Oleg Iliev and Dimitar Stoyanov. 2001. Multigrid-adaptive local refinement solver for incompressible flows. In *International Conference on Large-Scale Scientific Computing*. Springer, 361–368.
- [51] Bärbel Janssen and Guido Kanschat. 2011. Adaptive multilevel methods with local smoothing for H^1 - and H^1 -curl-conforming high order finite element methods. *SIAM Journal on Scientific Computing* 33, 4 (2011), 2095–2114.
- [52] Zhenhua Jiang, Chao Yan, Jian Yu, and Wu Yuan. 2015. Practical aspects of p-multigrid discontinuous Galerkin solver for steady and unsteady RANS simulations. *International Journal for Numerical Methods in Fluids* 78, 11 (2015), 670–690. DOI : <http://dx.doi.org/10.1002/flid.4035>
- [53] Jean-Christophe Jouhaud, Marc Montagnac, and Loïc P. Tourrette. 2005. A multigrid adaptive mesh refinement strategy for 3D aerodynamic design. *International journal for numerical methods in fluids* 47, 5 (2005), 367–385.
- [54] Guido Kanschat. 2004. Multilevel methods for discontinuous Galerkin FEM on locally refined meshes. *Computers & structures* 82, 28 (2004), 2437–2445.
- [55] Guido Kanschat and Youli Mao. 2015. Multigrid methods for Hdiv-conforming discontinuous Galerkin methods for the Stokes equations. *Journal of Numerical Mathematics* 23, 1 (2015), 51–66.
- [56] D. W. Kelly, J. P. De S. R. Gago, O. C. Zienkiewicz, and I. Babuska. 1983. A posteriori error analysis and adaptive processes in the finite element method: Part I—error analysis. *Internat. J. Numer. Methods Engrg.* 19, 11 (1983), 1593–1619. DOI : <http://dx.doi.org/https://doi.org/10.1002/nme.1620191103>
- [57] Tzanio Kolev, Paul Fischer, Misun Min, Jack Dongarra, Jed Brown, Veselin Dobrev, Tim Warburton, Stanimire Tomov, Mark S Shephard, Valeria Abdelfattah, Ahmad Barra, Natalie Beams, Jean-Sylvain Camier, Noel Chalmers, Yohann Dudouit, Ali Karakas, Ian Karlin, Stefan Kerkemeier, Yu-Hsiang Lan, David Medina, Elia Merzari, Aleksandr Obabko, Will Pazner, Thilina Rathnayake, Cameron W Smith, Lukas Spies, Kasia Swirydowicz, Jeremy Thompson, Ananias Tomboulides, and Vladimir Tomov. 2021. Efficient exascale discretizations: High-order finite element methods. *The International Journal of High Performance Computing Applications* 35, 6 (2021), 527–552.
- [58] Martin Kronbichler and Momme Allalen. 2018. Efficient high-order discontinuous Galerkin finite elements with matrix-free implementations. In *Advances and new trends in environmental informatics*. Springer, 89–110.
- [59] Martin Kronbichler, Ababacar Diagne, and Hanna Holmgren. 2018. A fast massively parallel two-phase flow solver for microfluidic chip simulation. *International Journal of High Performance Computing Applications* 32, 2 (2018), 266–287. DOI : <http://dx.doi.org/10.1177/1094342016671790>
- [60] Martin Kronbichler, Niklas Fehn, Peter Munch, Maximilian Bergbauer, Karl-Robert Wichmann, Carolin Geitner, Momme Allalen, Martin Schulz, and Wolfgang A Wall. 2021. A next-generation discontinuous Galerkin fluid dynamics solver with application to high-resolution lung airflow simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'21*. Association for Computing Machinery (ACM), St. Louis, MO, USA, 1–15.
- [61] M. Kronbichler, T. Heister, and W. Bangerth. 2012. High Accuracy Mantle Convection Simulation through Modern Numerical Methods. *Geophysical Journal International* 191 (2012), 12–29. DOI : <http://dx.doi.org/10.1111/j.1365-246X.2012.05609.x>
- [62] Martin Kronbichler and Katharina Kormann. 2012. A generic interface for parallel cell-based finite element operator application. *Computers & Fluids* 63 (2012), 135–147. DOI : <http://dx.doi.org/10.1016/j.compfluid.2012.04.012>
- [63] Martin Kronbichler and Katharina Kormann. 2019. Fast Matrix-Free Evaluation of Discontinuous Galerkin Finite Element Operators. *ACM Trans. Math. Software* 45, 3 (2019), 28:1–40. DOI : <http://dx.doi.org/10.1145/3325864>
- [64] Martin Kronbichler and Karl Ljungkvist. 2019. Multigrid for matrix-free high-order finite element computations on graphics processors. *ACM Transactions on Parallel Computing* 6, 1 (2019), 2:1–32.
- [65] Martin Kronbichler, Dmytro Sashko, and Peter Munch. 2022. Enhancing data locality of the conjugate gradient method for high-order matrix-free finite-element implementations. *International Journal of High Performance Computing Applications* in press (2022). DOI : <http://dx.doi.org/10.1177/10943420221107880>
- [66] Martin Kronbichler and Wolfgang A. Wall. 2018. A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers. *SIAM Journal on Scientific Computing* 40, 5 (2018), A3423–A3448.
- [67] Chunlei C. Liang, Ravishekar Kannan, and Z.J. Wang. 2009. A p-multigrid spectral difference method with explicit and implicit smoothers on unstructured triangular grids. *Computers & Fluids* 38, 2 (2009), 254–265. DOI : <http://dx.doi.org/10.1016/j.compfluid.2008.02.004>
- [68] Karl Ljungkvist. 2014. Matrix-free finite-element operator application on graphics processing units. In *European Conference on Parallel Processing*. Springer, 450–461.
- [69] Karl Ljungkvist. 2017. Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes.. In *SpringSim (HPC)*.

- [70] S Lopez and Raffaele Casciaro. 1997. Algorithmic aspects of adaptive multigrid finite element analysis. *International journal for numerical methods in engineering* 40, 5 (1997), 919–936.
- [71] Cao Lu, Xiangmin Jiao, and Nikolaos Missirlis. 2014. A Hybrid Geometric+Algebraic Multigrid Method with Semi-Iterative Smoothers. *Numerical Linear Algebra with Applications* 21, 2 (2014), 221–238. DOI : <http://dx.doi.org/10.1002/nla.1925>
- [72] Hong Luo, Joseph D. Baum, and Rainald Löhner. 2006. A p-multigrid discontinuous Galerkin method for the Euler equations on unstructured grids. *J. Comput. Phys.* 211, 2 (2006), 767–783. DOI : <http://dx.doi.org/10.1016/j.jcp.2005.06.019>
- [73] Hong Luo, Joseph D. Baum, and Rainald Löhner. 2008. Fast p-Multigrid Discontinuous Galerkin Method for Compressible Flows at All Speeds. *AIAA Journal* 46, 3 (2008), 635–652. DOI : <http://dx.doi.org/10.2514/1.28314>
- [74] Yvon Maday and Rafael Munoz. 1988. Spectral element multigrid. II. Theoretical justification. *Journal of Scientific Computing* 3, 4 (1988), 323–353. DOI : <http://dx.doi.org/10.1007/BF01065177>
- [75] Brendan S. Mascarenhas, Brian T. Helenbrook, and Harold L. Atkins. 2009. Application of p-Multigrid to Discontinuous Galerkin Formulations of the Euler Equations. *AIAA Journal* 47, 5 (2009), 1200–1208. DOI : <http://dx.doi.org/10.2514/1.39765>
- [76] Brendan S. Mascarenhas, Brian T. Helenbrook, and Harold L. Atkins. 2010. Coupling p-multigrid to geometric multigrid for discontinuous Galerkin formulations of the convection-diffusion equation. *J. Comput. Phys.* 229, 10 (2010), 3664–3674. DOI : <http://dx.doi.org/10.1016/j.jcp.2010.01.020>
- [77] Stephen F. McCormick. 1989. *Multilevel adaptive methods for partial differential equations*. SIAM.
- [78] J. Markus Melenk, Klaus Gerdes, and Christoph Schwab. 2001. Fully discrete hp-finite elements: fast quadrature. *Computer Methods in Applied Mechanics and Engineering* 190, 32 (2001), 4339–4364. DOI : [http://dx.doi.org/10.1016/S0045-7825\(00\)00322-4](http://dx.doi.org/10.1016/S0045-7825(00)00322-4)
- [79] William F. Mitchell. 2010. The hp-multigrid method applied to hp-adaptive refinement of triangular grids. *Numerical Linear Algebra with Applications* 17, 2-3 (2010), 211–228.
- [80] Peter Munch, Katharina Kormann, and Martin Kronbichler. 2021. hyper.deal: An Efficient, Matrix-free Finite-element Library for High-dimensional Partial Differential Equations. *ACM Trans. Math. Software* 47, 4 (2021), 33:1–34. DOI : <http://dx.doi.org/10.1145/3469720>
- [81] Peter Munch, Karl Ljungkvist, and Martin Kronbichler. 2022. Efficient application of hanging-node constraints for matrix-free high-order FEM computations on CPU and GPU. In *High Performance Computing*. Springer International Publishing, Cham, 133–152.
- [82] Steffen Müthing, Marian Piatkowski, and Peter Bastian. 2017. High-performance implementation of matrix-free high-order discontinuous Galerkin methods. *arXiv preprint arXiv:1711.10885* (2017).
- [83] Cristian R. Nastase and Dimitri J. Mavriplis. 2006. High-order discontinuous Galerkin methods using an hp-multigrid approach. *J. Comput. Phys.* 213, 1 (2006), 330–357. DOI : <http://dx.doi.org/10.1016/j.jcp.2005.08.022>
- [84] Benedict O'Malley, József Kópházi, Richard P. Smedley-Stevenson, and Monroe D. Eaton. 2017a. Hybrid Multi-level solvers for discontinuous Galerkin finite element discrete ordinate diffusion synthetic acceleration of radiation transport algorithms. *Annals of Nuclear Energy* 102, April (2017), 134–147. DOI : <http://dx.doi.org/10.1016/j.anucene.2016.11.048>
- [85] Benedict O'Malley, József Kópházi, Richard P. Smedley-Stevenson, and Monroe D. Eaton. 2017b. P-multigrid expansion of hybrid multilevel solvers for discontinuous Galerkin finite element discrete ordinate (DG-FEM-SN) diffusion synthetic acceleration (DSA) of radiation transport algorithms. *Progress in Nuclear Energy* 98 (2017), 177–186. DOI : <http://dx.doi.org/10.1016/j.pnucene.2017.03.014>
- [86] Steven A. Orszag. 1980. Spectral methods for problems in complex geometries. *J. Comput. Phys.* 37, 1 (1980), 70–92. DOI : [http://dx.doi.org/10.1016/0021-9991\(80\)90005-4](http://dx.doi.org/10.1016/0021-9991(80)90005-4)
- [87] Sachin Premasuthan, Chunlei Liang, Antony Jameson, and Zhi Wang. 2009. A p-Multigrid Spectral Difference Method For Viscous Compressible Flow Using 2D Quadrilateral Meshes. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*. American Institute of Aeronautics and Astronautics, Orlando, Florida. DOI : <http://dx.doi.org/10.2514/6.2009-950>
- [88] Einar M. Rønquist and Anthony T. Patera. 1987. Spectral element multigrid. I. Formulation and numerical results. *Journal of Scientific Computing* 2, 4 (1987), 389–406. DOI : <http://dx.doi.org/10.1007/BF01061297>
- [89] Johann Rudi, Omar Ghattas, A. Cristiano I. Malossi, Tobin Isaac, Georg Stadler, Michael Gurnis, Peter W. J. Staar, Yves Ineichen, Costas Bekas, and Alessandro Curioni. 2015. An extreme-scale implicit solver for complex PDEs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*. ACM Press, New York, USA, 1–12. DOI : <http://dx.doi.org/10.1145/2807591.2807675>
- [90] Anton Schüller. 2013. *Portable Parallelization of Industrial Aerodynamic Applications (POPINDA): Results of a BMBF Project*. Vol. 71. Springer Science & Business Media.
- [91] Khosro Shahbazi, Dimitri J. Mavriplis, and Nicholas K. Burgess. 2009. Multigrid algorithms for high-order discontinuous Galerkin discretizations of the compressible Navier-Stokes equations. *J. Comput. Phys.* 228, 21 (2009), 7917–7940. DOI : <http://dx.doi.org/10.1016/j.jcp.2009.07.013>

- [92] Mark S Shephard. 1984. Linear multipoint constraints applied via transformation as part of a direct stiffness assembly process. *Internat. J. Numer. Methods Engrg.* 20, 11 (1984), 2107–2112.
- [93] Jörg Stiller. 2016a. Nonuniformly weighted Schwarz smoothers for spectral element multigrid. *Journal of Scientific Computing* 72 (2016), 81–96. DOI : <http://dx.doi.org/10.1007/s10915-016-0345-z>
- [94] Jörg Stiller. 2016b. Robust multigrid for high-order discontinuous Galerkin methods: A fast Poisson solver suitable for high-aspect ratio Cartesian grids. *J. Comput. Phys.* 327 (2016), 317–336. DOI : <http://dx.doi.org/10.1016/j.jcp.2016.09.041>
- [95] Jörg Stiller. 2017. Robust Multigrid for Cartesian Interior Penalty DG Formulations of the Poisson Equation in 3D. In *Spectral and High Order Methods for Partial Differential Equations ICOSAHOM 2016. Lecture Notes in Computational Science and Engineering*, M. Bittencourt (Ed.). Vol. 119. Springer, Cham, 189–201. DOI : http://dx.doi.org/10.1007/978-3-319-65870-4_12
- [96] Mario Storti, N. Nigro, and Sergio Idelsohn. 1991. Multigrid methods and adaptive refinement techniques in elliptic problems by finite element methods. *Computer methods in applied mechanics and engineering* 93, 1 (1991), 13–30.
- [97] Klaus Stüben. 2001. A review of algebraic multigrid. *J. Comput. Appl. Math.* 128, 1-2 (2001), 281–309. DOI : [http://dx.doi.org/10.1016/S0377-0427\(00\)00516-1](http://dx.doi.org/10.1016/S0377-0427(00)00516-1)
- [98] Hari Sundar, George Biros, Carsten Burstedde, Johann Rudi, Omar Ghattas, and Georg Stadler. 2012. Parallel geometric-algebraic multigrid on unstructured forests of octrees. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC'12*. IEEE Computer Society Press, Salt Lake City, UT, USA, 1–11.
- [99] Hari Sundar, Georg Stadler, and George Biros. 2015. Comparison of multigrid algorithms for high-order continuous finite element discretizations. *Numerical Linear Algebra with Applications* 22, 4 (2015), 664–680. DOI : <http://dx.doi.org/10.1002/nla.1979>
- [100] Haijun Wu and Zhiming Chen. 2006. Uniform convergence of multigrid V-cycle on adaptively refined finite element meshes for second order elliptic problems. *Science in China Series A: Mathematics* 49, 10 (2006), 1405–1429.
- [101] Pamela Zave and Werner C. Rheinboldt. 1979. Design of an adaptive, parallel finite-element system. *ACM Trans. Math. Software* 5, 1 (1979), 1–17.
- [102] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software* 4, 37 (May 2019), 1370. DOI : <http://dx.doi.org/10.21105/joss.01370>

Paper IV

Cache-optimized and low-overhead implementations of additive Schwarz methods for high-order FEM multigrid computations

The International Journal of High Performance Computing Applications 2023, Vol. 0(0) 1–18
© The Author(s) 2023
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/10943420231217221
journals.sagepub.com/home/hpc


Peter Munch¹  and Martin Kronbichler^{1,2} 

Abstract

This contribution presents data-locality optimizations of the additive Schwarz method (ASM) based on the fast-diagonalization method defined on overlapping cell-centric and vertex-star patches in the context of high-order matrix-free finite-element computations on modern CPU-based hardware. The developments are guided by detailed performance models of the ASM in the context of Chebyshev iterations when used as smoothers for p -multigrid. The proposed efficient implementation of ASM adopts concepts known from cell-loop infrastructures for efficient operator evaluation, in particular, the storage of information per geometric entity and the cache-friendly interleaving of cell loops and vector updates as a means to increase data locality. We use the latter concept for both applying the weights required by ASM and performing the vector updates required by the Chebyshev iteration, which are memory-bound operations with non-negligible costs in comparison to efficient operator evaluation. Furthermore, the solution of a scalar Poisson problem on a highly anisotropic and an unstructured mesh with p -multigrid using the developed smoothers indicates that efficient implementations of the additive Schwarz method can outperform optimized point-Jacobi preconditioners already for simple problems despite being more than twice as expensive per iteration. Even though ASM introduces additional communication steps per smoother application, the reduced number of iterations can lead to improved parallel scalability for intermediate problem sizes. At the scaling limit, the results are inconclusive due to these two opposing effects.

Keywords

High-order finite-element methods, multigrid smoothers, matrix-free operator evaluation, node-level optimization

1. Introduction

Multigrid methods are among the most competitive solvers for the linear systems arising upon discretization of second-order partial differential equations (PDEs) (Gholami et al., 2016), for example, for the Poisson equation $-\Delta u = f$, where u is the solution variable and f is the source term. Poisson-like problems occur as subproblems in many application fields, for example, in computational fluid dynamics (cf. Arndt et al. (2020); Deville et al. (2002); Fischer et al. (2022)) or in computational plasma physics (Munch et al., 2021), making the detailed investigation of the performance of the multigrid methods for solving $Ax = b$ on state-of-the-art hardware highly relevant.

Multigrid algorithms use smoothers to reduce different frequencies in the iteration error on a hierarchy of levels of different numerical resolution. During presmoothing, the high-frequency error components in the initial guess are removed, whereas postsmoothing aims to damp the

high-frequency error components introduced when interpolating corrections from a coarse to a fine level. Smoothers govern the number of multigrid cycles needed to obtain convergence, determine the allowable factor of coarsening from one level to the next, and are, as a consequence, critical for the total multigrid efficiency.

A widely used smoother for high-order finite-element methods (FEM) is the iterated point-Jacobi method applying the inverse diagonal, which has a *low cost* per iteration

¹High-Performance Scientific Computing, University of Augsburg, Augsburg, Germany

²Applied Numerics, Faculty of Mathematics, Ruhr University Bochum, Bochum, Germany

Corresponding author:

Peter Munch, High-Performance Scientific Computing, University of Augsburg, Universitätsstraße 12a, Augsburg 86159, Germany.
Email: peter.muench@uni-a.de

(Kronbichler and Wall, 2018) and necessitates intermediate levels, when used in a p -multigrid setting (Fehn et al., 2020). As an alternative, smoothers built around a more *expensive* additive Schwarz method (ASM; also known as “hybrid Schwarz method”; cf. Lottes and Fischer (2005)) potentially allow to skip intermediate levels and to perform fewer smoother iterations. ASM performs independent approximate solves on sub-blocks of the global operator, whose results are then combined. The fast-diagonalization method allows for very efficient local solution, which, in turn, necessitates to consider the cost of the rest of the solution chain, for example, due to memory access for indices and weights. In this publication, we propose an efficient novel implementation of ASM, which reduces the organizational overhead and interleaves arithmetically intensive steps, like the evaluation of integrals or matrix inversion, with vector updates in a single sweep through data. The proposed algorithms increase the cache hit rates. We investigate the performance in the context of a state-of-the-art matrix-free infrastructure not only with focus on the node-level performance but also for strong scaling on large-scale supercomputers and compare the results to optimized point-Jacobi-based smoothers. Despite a wide range of previous contributions, the topics of data locality and possible performance optimizations of the different algorithms have not been studied before. On a related aspect, recent works in NekRS (Phillips et al., 2022; Phillips and Fischer, 2022) also compare these two smoothers but using a more basic implementation with several sweeps through data.

The algorithms presented in this publication have been integrated into the open-source finite-element library deal.II (Arndt et al., 2021, 2022). All experimental results have been obtained with small benchmark programs, which are available on GitHub.¹

The remainder of this work is organized as follows. First, we provide an overview of the considered smoothers and detail our high-performance implementations. Next, we introduce the other components of a multigrid solver, present a simplified performance model aiming to incorporate the costs of the smoothers, and demonstrate performance results for different meshes. Finally, we summarize our conclusions and point to further research directions.

2. Preconditioners and smoothers

In the following, we introduce the ASM as a preconditioner and then embed it into Chebyshev iterations, which allow to execute the preconditioner multiple times, as common in the context of smoothers for multigrid.

2.1. Additive Schwarz method

The additive Schwarz method has the structure

$$\mathbf{v} = P^{-1}\mathbf{u} = \sum_i R_i^\top \widehat{A}_i^{-1} R_i \mathbf{u} \quad \text{w.} \quad \widehat{A}_i = R_i A R_i^\top. \quad (1)$$

It solves a subproblem on each block \widehat{A}_i of the matrix A and combines the results of the independent solves. The blocks are given by the application of a restriction matrix R_i and might overlap. In the literature, there are different ways to define the blocks. In this publication, we consider variants of cell-centric patches and vertex-star patches. Figure 1 shows—for a structured and an unstructured mesh—cell-centric patches with different overlap n with neighboring cells and vertex-star patches. In the case of a cell-centric patch with $n = 1$ (minimal overlap), all the unknowns associated with a cell constitute a block. Both cell-centric patches with $n > 1$ and vertex-star patches may involve an arbitrary number of cells on unstructured meshes. This makes the development of an efficient implementation of the inverse \widehat{A}_i^{-1} for such meshes challenging, since tensor-product structures are typically used to create high-quality yet cheap-to-apply approximations of the inverse (Brubeck and Farrell, 2022; Lynch et al., 1964). A potential remedy in the case of cell-centric patches is to only consider face neighbors (Fischer et al., 2000), as illustrated in Figure 1(c).

Unknowns located in the overlapping region receive several contributions in the ASM algorithm (1). While the multiplicative Schwarz method makes use of this ambiguity in the algorithm definition by considering an updated residual after each loop iteration i , ASM needs to introduce a weighting in order to obtain a partition of unity or, alternatively, apply a global damping parameter in the multigrid iteration (Loisel et al., 2008). Table 1 shows possible weighting strategies. Weights can be applied *locally* cell by cell (left part of the table) where the most natural strategy is to apply them after the local inverse (*post*). The weighting can also be split into parts before and after the local inverse, preserving the symmetry of P^{-1} (labeled *symm* in the table). In the literature, different approaches to define the weighting (diagonal) matrix W_i have been used. For instance, Stiller (2017) proposed a smooth function defined between 0 and one on a cell and a small overlap. In many instances, the weights are the inverse of the valence of the unknowns. In this case, the local weights are $W_i = R_i W$, which enables to apply W *globally* on the entries in the final vector, as indicated in Table 1 (right). A third option is to define the value of each entry in \mathbf{v} in equation (1) only from a single patch solve (cf. Cai and Sarkis (1999)). This implies that W_i becomes a Boolean matrix, which can be considered by a modified scatter operator ($R_i W_i$), and also allows to skip a communication step, which is the main motivation to use this strategy in the context of high-performance computing.

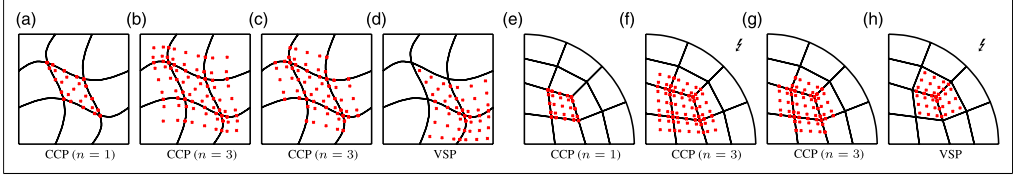


Figure 1. Examples of cell-centric patches (CCP) and vertex-star patches (VSP) on structured and unstructured meshes. Red crosses indicate the location of unknowns. Lightning bolt symbols indicate patch types that lack geometric structure needed for tensor-product evaluation and, therefore, will not be considered. In the case of cell-centric patches with $n > 1$, one can consider all neighbors or just face neighbors (c, g).

Table 1. Four ways to apply weights in the case of ASM: local versus global and postprocessing (post) versus symmetric decomposition (symm) with $\widehat{W}_i := \sqrt{W}_i$ and $\widehat{W} := \sqrt{W}$.

| | Local | Global |
|------|--|--|
| post | $\mathbf{v} = \sum R_i^\top W_i A_i^{-1} R_i \mathbf{u}$ | $\mathbf{v} = W (\sum R_i^\top A_i^{-1} R_i) \mathbf{u}$ |
| symm | $\mathbf{v} = \sum R_i^\top \widehat{W}_i A_i^{-1} \widehat{W}_i R_i \mathbf{u}$ | $\mathbf{v} = \widehat{W} (\sum R_i^\top A_i^{-1} R_i) (\widehat{W} \mathbf{u})$ |

We will call this approach restricted additive Schwarz method (RAS) as some authors do. In the literature, the term ‘‘RAS’’ is used inconsistently and often refers generally to ASM with weighting.

One issue with the definition (1) is that it relies on the assembly of the matrix A , which leads to prohibitive costs in the context of higher-order methods. As indicated above, low-rank approximations to A_i are common, typically derived from an auxiliary operator defined on a nearby Cartesian mesh (Couzy, 1995; Phillips et al., 2022; Witte et al., 2021). This choice relies on the representation of the Laplacian A_i^{cart} on a Cartesian mesh as the tensor product of 1D stiffness and mass matrices. For example, the fast-diagonalization method (FDM; cf. Lynch et al. (1964)) gives an explicit formula for the inverse in 2D as

$$(A_i^{\text{cart}})^{-1} = T_i \otimes T_0 (\Lambda_1 \otimes I + I \otimes \Lambda_0)^{-1} T_1^\top \otimes T_0^\top,$$

with $T_i \in \mathbb{R}^{n \times n}$ and $\Lambda_i \in \mathbb{R}^n$ being the (orthonormal) eigenvectors and the diagonal matrix of eigenvalues, obtained on a patch level from a generalized eigendecomposition $K_i T_i = M_i T_i \Lambda_i$.

2.2. Chebyshev iteration

The Chebyshev iteration (Adams et al., 2003) of degree k is given by a three-term recurrence. For matrix A and a base preconditioner P (e.g., point Jacobi or ASM), it reads

$$x_1 = \alpha_1 P^{-1} b, \quad (2a)$$

$$x_2 = (1 + \alpha_2) x_1 + \beta_2 P^{-1} r_1, \quad (2b)$$

$$x_{n+1} = (1 + \alpha_n) x_n - \alpha_n x_{n-1} + \beta_n P^{-1} r_n, \quad (2c)$$

for $n < k$ with the residual $r_n = b - Ax_n$. We set the coefficients according to the 1st-kind and the recently proposed 4th-kind Chebyshev polynomials (Lottes, 2022; Phillips and Fischer, 2022) and estimate the maximal eigenvalue by a power iteration. The 1st-kind Chebyshev polynomials set a lower eigenvalue bound as a range for damping the spectrum to $\lambda_{\min} = \lambda_{\max}/20$ (Kronbichler and Wall, 2018). This is in accordance with Adams et al. (2003), which also shared the observation that the convergence is not particularly sensitive to the actual factor.

3. Implementation

In the case that the preconditioner P^{-1} is applied several times within the smoother, the residual ($r = b - Ax$) needs to be computed after each iteration. An efficient smoother therefore aims to quickly evaluate Ax and $P^{-1}r$, including data-locality considerations across these steps and the vector updates. In this publication, we concentrate on the two challenges of ASM, restriction R_i and vector updates corresponding to weighting and Chebyshev iterations, and assume that the block solvers are optimally implemented. Before discussing these challenges, we shortly revisit cell-loop-based matrix-free operator evaluation, from which we can adopt building blocks.

In the following, all experiments are run, unless stated otherwise, on a dual-socket Intel Cascade Lake Xeon Gold 6230 system: 40 cores, 5.1 single-precision TFLOP/s with AVX-512 SIMD instructions, 180 GB/s of memory bandwidth measured by the STREAM triad benchmark, 32 kB L1-cache (private), 1 MB L2-cache (private), and 14 MB L3-cache shared between 10 cores. We run all experiments at least 10 times and report the best timings of these runs. The statistical distribution is close to the minimal time within a few percent. The code is parallelized via MPI, and all cores of the compute node are used to saturate the memory bandwidth. All experiments are run with single-precision floating-point-number arithmetic (FP32) due to our intention to use the implementations within a

mixed-precision multigrid preconditioner. However, the implementations developed and the results obtained are also valid for double-precision numbers (FP64).

3.1. Matrix-free operator evaluation

In this work, we consider a matrix-free evaluation of the operator action $\mathbf{v} = \mathbf{A}\mathbf{u}$ for general meshes, which computes the integrals underlying a finite-element discretization on the fly. Here, a loop over all cells is performed, applying the element stiffness matrix on a vector restricted to the unknowns (degrees of freedom; DoFs) of the cell (Kronbichler and Kormann, 2012)

$$\mathbf{v} = \mathbf{A}(\mathbf{u}) = \sum_i \mathbf{R}_i^\top \mathbf{A}_i \mathbf{R}_i \mathbf{u}. \quad (3)$$

In practice, \mathbf{A}_i needs not be computed explicitly but can be expressed as the action of the constituents from trial solution and test functions in a quadrature approach $\mathbf{A}_i = \mathbf{S}_i^\top \mathbf{D}_i \mathbf{S}_i$. For a Laplace operator, \mathbf{D}_i is represented by applying $w_q |J_q| J_q^{-1} J_q^{-\top}$ to the gradient at each quadrature point and \mathbf{S}_i evaluates the gradients at the quadrature points, for which—in the case of tensor-product elements—a technique known as sum factorization (Orszag, 1980; Melenk et al., 2001) can be applied. There is an extensive literature on optimizing (3) for CPUs (Anderson et al., 2021; Kronbichler and Kormann, 2012, 2019; Munch et al., 2021) and GPUs (Chalmers et al., 2023; Kolev et al., 2021; Ljungkvist, 2017). In the following, we shortly discuss CPU-specific optimizations adopted in the context of ASM.

Vectorization across elements (Kronbichler and Kormann, 2012, 2019) performs multiplication by \mathbf{A}_i for several cells in different lanes of the same SIMD instruction and gives the highest throughput under the assumption that the operations between lanes do not diverge too much. This is an outer-loop vectorization compared to the typical FEM workflow, necessitating a data rearrangement in a struct-of-arrays fashion during gathering/scattering with \mathbf{R}_i .

At each quadrature point, the *mapping metrics* $J_q^{-\top}$ and $|J_q|$ are needed. Storing and loading the information for each quadrature point gives a memory-bound algorithm (Kronbichler and Kormann, 2019). In our experiments, we use, on affine meshes, compression that is applicable as J_q is the same on several quadrature points and compute the metric terms from a triquadratic representation of a deformed cell geometry (Kronbichler et al., 2022).

3.2. Restriction and data locality in cell loops

The restriction matrix \mathbf{R}_i maps local indices to vector indices in both the ASM loop case (1) and the matrix-free loop case (3). Note that \mathbf{R}_i might also resolve constraints of type $x_i = \sum_j c_{ij} x_j + b_i$, for example, hanging-node constraints, implying a non-pure Boolean matrix. Without loss of generality, we

consider here only homogeneous Dirichlet constraints ($x_i = 0$) and refer the reader to Kronbichler and Kormann (2012); Munch et al. (2022). For the sake of simplicity, we first discuss implementation details on \mathbf{R}_i for the cells of a FEM mesh and study the adoption to patches subsequently.

In the worst case, \mathbf{R}_i has to store all indices. This implies an indirect vector access with each entry of the vector read individually. In the context of, for example, discontinuous Galerkin methods, it is possible to store only the index of an unknown of a cell, assuming that all DoFs related to a cell are enumerated contiguously. A similar strategy can be also adopted to standard high-order FEM in the case that DoFs are contiguous on each geometric entity, as proposed by Kronbichler et al. (2022). Here, one needs to store only the first index of each entity of a cell (3^d). Based on this information, we load vector entries directly into the part of a local buffer dedicated to the geometric entity. For unstructured meshes, one needs to reorient entries on lines and faces Scroggs et al. (2022), which correspond to an in-place permutation for Lagrange elements. The information can be encoded with 1 bit per line and 3 bits per face, that is, with 4 bits (1 Byte) in 2D and 30 bits (4 Bytes) in 3D. This *efficient index storage* is visualized in Figure 2(a).

Furthermore, the restriction matrix \mathbf{R}_i defines the read/write-access dependency of a cell. This information can be used to derive algorithms with improved data locality, needed because many implementations of matrix-free high-order finite-element algorithms with sum factorization are limited by the bandwidth from main memory (RAM) on modern hardware. As a result, performance optimizations aim at improving the data locality, especially on CPU architectures with their limited RAM bandwidth. A simple strategy to utilize different levels of the cache within matrix-free operator evaluation is to apply an iteration sequence over cells with a high reuse of data from previous cells. This is the case, for example, for the Morton order chosen here (Bangerth et al., 2011). In addition, in order to increase the throughput of algorithms that consist of compute-intensive *cell loops* and memory-bound *vector updates* operating on the same set of vectors, these two steps can be *interleaved*.

To derive a strategy for interleaving cell loops and vector updates, we exploit the facts (1) that a vector entry only has to be in a valid state once the first cell accesses the corresponding index j , that is, the cell with index $\min(\{ij \in \mathbf{R}_i\})$, and (2) that a vector entry has received all contributions or is not needed anymore, that is, can be overridden, once the last cell with the corresponding dependency region, that is, the cell with index $\max(\{ij \in \mathbf{R}_i\})$, has processed it. For the sake of simplicity and without loss of generality, we assume that cells are processed sequentially according to their index i . Naturally, tracking the state of individual vector entries and performing vector updates before and after each cell operation introduces an excessive overhead.

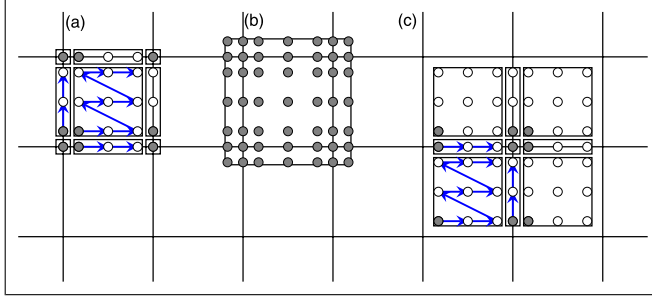


Figure 2. Visualization of index storage in 2D for $p = 4$ with filled circles indicating stored information; the rest is deduced by contiguous storage, indicated by arrows (exemplarily shown for a few geometric entities). (a) cell-centric patch with $n = 1$, (b) cell-centric patch with $n = 2$, (c) vertex-star patch.

In [Kronbichler et al. \(2022\)](#), a suitable generic software infrastructure for arbitrary vector operations on the destination vector and the source vector was proposed. A user-defined pre-operation is run on a range of vector entries just before these are touched by the cell loop the first time. Upon exit of the pre-operation, the entries in the source vector are ready for being read by the cell integrals A_i and the destination vector is in a state to receive contributions of cell integrals at the respective indices. Furthermore, a user-defined post-operation is run on a range of entries just after the last access in the cell loop. Here, both source and destination vector can be modified, since the cell integrals do not access these entries again. As a heuristic when to interrupt the cell loop for working on the vector-update operations, we require that the accumulated data of the cells' work fits into the fast L2 cache to maximize the L2 hit rate during vector updates, which implies a partitioning of the cell index space $[1, N_c]$

$$\mathbf{v} = A(\mathbf{u}) = \sum_c \underbrace{\sum_{r_c \leq i < r_{c+1}} R_i^\top A_i R_i}_{*} \mathbf{u},$$

where the inner sum $*$ is interleaved with vector updates on appropriate index ranges.

In a parallel setting, R_i also specifies the needed ghosting of the vectors. Furthermore, the set of cells with remote dependencies can be treated as an own partition so that communication can be performed asynchronously with the loop over cells inside the domain.

Instead of tracking the state of individual entries for vector operations, we track the state of cache lines. In consequence, cache lines can remain active over long periods if individual entries are accessed at different times. We define the *liveliness* of a cache line as the difference of the indices of the last and the first cell range accessing a cache line ($c_{\max} - c_{\min} + 1$). To minimize the overall liveliness, we enumerate the DoFs with

the strategy proposed in [Kronbichler et al. \(2022\)](#): in a preprocessing step, we determine, for each DoF, a tuple containing an index it would get in touch order of the cell loop, the number of cell ranges modifying the entry, and the information whether the values depend on remote processes; sorting according to the tuple gives the new ordering.

Expressing algorithms in a way that they fit the pre/post-concept needs to respect the dependency region of the matrix-vector product in terms of the overlap of unknowns and might involve algorithmic transformations. In [Kronbichler et al. \(2022\)](#), such a process was shown for preconditioned conjugate-gradient solvers. In the current work, we propose novel operations for ASM and Chebyshev iterations and analyze their advantages against established software concepts in the finite-element community.

In the present publication, we focus on CPU-type hardware, where dependency regions associated with a moderate number of compute units can be explicitly defined. However, [Kronbichler et al. \(2022\)](#) indicate that expressing algorithms in the form of pre/post-operations also benefits GPU implementations, since the associated loop fusion allows to reduce the latency due to the reduced number of kernel launches.

3.3. Low-overhead additive Schwarz method

The structure of the ASM loop (1) is the same as in the case of (3). In the case of minimal overlap ($n = 1$), the restriction operator R_i is even identical to the one of the matrix-free cell loop, allowing to reuse existing implementations of (3). The index sets for the other patch types are distinct, necessitating the setup of an additional set of restriction operators. Note that the overlapping nature of these types of patches leads to challenges, since more indices might need to be stored.

The Cartesian auxiliary mesh of cell-centric patches needed for FDM is generated by the extents of the cell under consideration and of its face neighbors. The extensions are determined based on the harmonic weight of the distances of

opposing faces. The grid sizes are used to scale 1D reference mass and stiffness matrices, which are locally assembled and decomposed for FDM. Boundary conditions are applied according the underlying PDE problem. The algorithm adopted from NekRS (Fischer et al., 2022) is visualized in Figure 3. In the same way as in the operator evaluation (3), we vectorize across elements. However, the transformation matrices (T) might differ for each cell. In order to minimize the memory consumption, we reuse the 1D matrices between cell batches if possible.

3.3.1. Cell-centric patches with $n = 1$. Patches centered around cells with minimal overlap ($n = 1$) consist of all $(p + 1)^d$ unknowns of a cell, including the efficient index storage (see Figure 2(a)).

For the application of the weights, we have multiple options (Table 1). If we apply the weights locally during the cell loop, we can store them cell-wise. Furthermore, the weights allow for a similar compression as the indices of unknowns, leading to 3^d stored weights per cell (see Figure 4(a)). The variants of post/symm only differ in when the weights are applied to the local buffers. If the weights are applied globally, we propose to interleave the operation with the cell loop. An issue of the global application of the weights in the context of symm is that the source vector might be immutable, requiring an additional vector.

A rough estimate indicates that the compressed storage is the most efficient variant ($\sim(3/p)^d$ overhead per unknown), followed by the global application (~ 1). Table 2 confirms these expectations experimentally for $p = 4$. Nonetheless, applying the weights globally in an interleaved way is close to be the best option of cell-wise weights; in both cases, the cost of weighting is nearly negligible. As a reference, the costs of global sequential application of weights are 35%–45% higher.

3.3.2. Cell-centric patches with $n > 1$. Patches centered around cells with $n > 1$ consist of unknowns inside of neighboring cells. For structured meshes, it is straightforward to consider all neighbors with $(p + n - 1)^d$ unknowns. In the case of unstructured meshes, we skip neighbors that are not connected

via a face in order to preserve the tensor-product structure, leading to a slight reduction in the number of unknowns.

The increased overlap results in less data access and smaller transformation matrices in certain directions for cells located at the boundary. For the sake of simplicity, we embed these small matrices in the full ones with a negligible impact on the computational work.

The set of ghost indices for $n > 1$ is a superset of the ones of (3). This allows us to set up ghosted vectors for the preconditioner and use them also in the operator evaluation (3), without copying between vector layouts.

In order to access the vectors, one cannot use the compressed gather approach from R_p , since unknowns on up to 5^d geometric entities are accessed. A 5^d -compression scheme is conceivable, however, would only benefit very high orders.

Similarly, weights cannot be compressed, since they can already differ within the same geometric entity. As a result, one can only store the weights either globally with overhead 1 or locally with overhead $((p + n - 1)/p)^d$ (see Figure 4(b)). Table 2 shows timing comparisons. The increased overlap renders the operation $> 4\times$ more expensive. Not surprisingly, a global interleaved application of the weights is the most efficient approach in this case, benefiting from the fast cache. As symm has to modify the source vector or a temporary copy of it, the costs are higher than those of the post-weight application for most settings. Sequential application of the weights is $\approx 8\%$ slower. This is a smaller value than the one in the case of minimal overlap, which is not surprising due to the higher base costs.

3.3.3. Vertex-star patches. We consider vertex-star patches only in the case of structured meshes consisting of precisely 2^d cells, that is, do not build patches at the boundary. Vertex-star patches consist of $(2p - 1)^d$ unknowns from 3^d geometric entities, namely, one vertex, six lines, 12 quadrilaterals, and eight hexahedra. One can regard this as a shifted view on the entities. Hence, the compression approaches of cell-centric patches with $n = 1$ can be adopted (see Figures 2(c) and 4(c)), reducing the organizational overhead. It is a natural choice to apply them locally on the patch. Indeed, our experiments

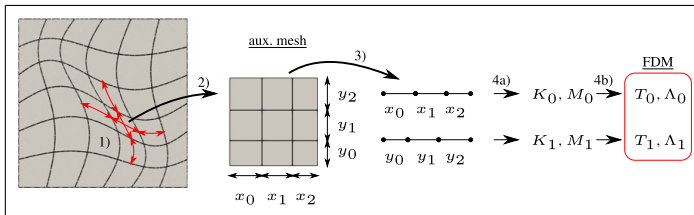


Figure 3. Visualization of the setup process of FDM for a 2D cell: (1) determine extension of a cell and its face neighbors; use them (2) to construct an auxiliary Cartesian mesh and (3) to set up auxiliary 1D meshes; (4) discretize the latter with FEM, which gives K_j/M_j and T_j/Λ_j .

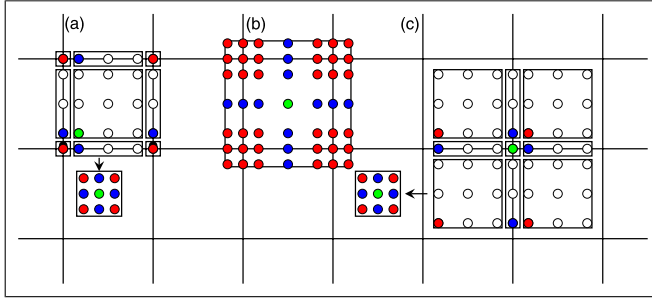


Figure 4. Visualization of cell-local weight storage in 2D for $p = 4$ with filled circles indicating stored information; weights labeled with hollow circles are not needed to be stored. (a) cell-centric patch with $n = 1$, (b) cell-centric patch with $n = 2$, (c) vertex-star patch.

Table 2. Hardware-performance counter measurements of different ways to apply weights in the context of FDM (float, $p = 4$, $d = 3$, 67 MDof, Cartesian mesh), using LKQWID (Treibig et al., 2010). The nomenclature, for example, 1-g-s-i, is as follows: the first identifier indicates the overlap ($n = 1, 2$ or v: vertex); the second specifies how the weights are applied (c: locally compressed, l: locally cell-wise, g: globally); the third identifies the sequence of the global application (s: sequential, p: pre-post interleaved with the loop over cells/vertices); and the last identifier specifies whether the source vector is constant (i) or mutable (m). In the case that no weighting is applied (l), “*-g-s-m” and “*-g-p-m” indicate whether the destination vector is zeroed out before or during the cell loop in form of a pre-operation. Measured quantities are data volume per DoF read/written from/to RAM (r/w) and the time of 10 operator applications. Values highlighted in bold face indicate the fastest setting for a given configuration.

| | No weights (l) | | | post | | | symm | | |
|---------|----------------|------------|-------------|-------------|------------|-------------|-------------|------------|-------------|
| | r | w | $t(s)$ | r | w | $t(s)$ | r | w | $t(s)$ |
| l-c | | | | 3.4 | 1.3 | 0.11 | 3.4 | 1.3 | 0.12 |
| l-l | | | | 6.3 | 1.3 | 0.15 | 6.3 | 1.3 | 0.16 |
| l-g-s-i | | | | | | | 8.8 | 4.2 | 0.25 |
| l-g-s-m | 3.8 | 2.1 | 0.14 | 5.8 | 3.1 | 0.19 | 7.8 | 4.2 | 0.24 |
| l-g-p-i | | | | | | | 5.3 | 2.3 | 0.14 |
| l-g-p-m | 2.9 | 1.3 | 0.11 | 4.1 | 1.3 | 0.12 | 4.2 | 2.3 | 0.13 |
| 2-l | | | | 16.6 | 3.1 | 0.60 | 16.6 | 3.1 | 0.62 |
| 2-g-s-i | | | | | | | 18.5 | 5.5 | 0.57 |
| 2-g-s-m | 13.6 | 3.5 | 0.46 | 15.5 | 4.5 | 0.51 | 17.5 | 5.5 | 0.56 |
| 2-g-p-i | | | | | | | 18.9 | 4.5 | 0.53 |
| 2-g-p-m | 13.3 | 3.0 | 0.45 | 15.4 | 3.4 | 0.47 | 17.3 | 4.5 | 0.51 |
| v-c | | | | 6.0 | 2.5 | 0.35 | 6.0 | 2.5 | 0.35 |
| v-l | | | | 13.5 | 2.6 | 0.48 | 13.5 | 2.7 | 0.48 |
| v-g-s-i | | | | | | | 11.1 | 5.1 | 0.46 |
| v-g-s-m | 6.2 | 3.2 | 0.35 | 8.1 | 4.1 | 0.40 | 10.1 | 5.1 | 0.45 |
| v-g-p-i | | | | | | | 9.8 | 3.8 | 0.39 |
| v-g-p-m | 5.5 | 2.5 | 0.33 | 7.2 | 2.7 | 0.35 | 8.3 | 3.8 | 0.37 |

confirm the efficiency of this choice (see Table 2). Global weighting with interleaving is close to the best option again.

3.3.4. Liveliness analysis. From the previous experiments, we can deduce that the global application of the weights, interleaved with the vector updates, is an attractive alternative, especially for cell-centric patches with $n > 1$. Furthermore, it is observable that the benefit of interleaving cell loops and vector updates decreases slightly for $n > 1$ and vertex-star patches. To support this observation, we compare the

liveliness of cache lines for the different methods in Figure 5(a)). Cache lines live longer for the latter two options, resulting in increased probability of cache spills, particularly due to increased work-set sizes for the cell-local operations. Note that, for enumerating unknowns, we used the strategy that is designed for cell loops and, as a consequence, favors cell-centric patches with $n = 1$.

We also consider other enumeration strategies by using patch ranges instead of cell ranges in the enumeration algorithm presented above. Figure 5(b) illustrates an

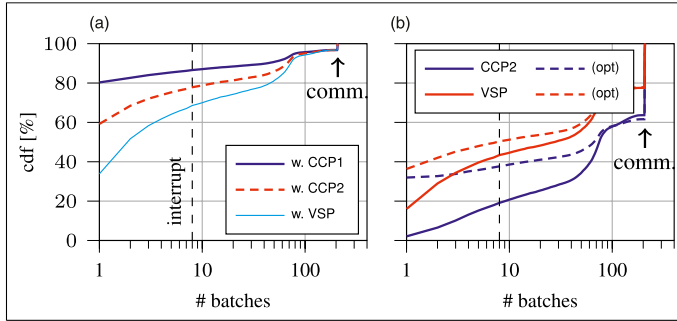


Figure 5. Cumulative distribution function of liveness of cache lines in a vector for a 3D scalar Laplacian with polynomial degree $p = 4$ on 40 MPI processes. The Laplacian involves 67 million DoFs subdivided into 208 batches on each MPI process. The cell loop is interrupted after eight batches. Left: throughput of cell-centric patches with $n = 1$ for different enumeration schemes. Right: throughput of cell-centric patches with $n = 2$ and vertex-star patches with an enumeration enforced by cell-centric patches with $n = 1$ or an optimal enumeration. (a) cell-centric patch $n = 1$, (b) cell-centric patch $n = 2$ and vertex-star patch.

increased number of entities with low liveness. This increases the interleavable data by 14% in the case of vertex-star patches and doubles their amount for cell-centric patches. Tables 3 and 4 show the influence of the optimized enumeration on the run times. As expected, the effect on the times in the vertex-star case is not large despite a clearly reduced memory access, indicating in-core bottlenecks. In the cell-centric case with $n = 2$, the time is reduced by 8%.

Note that A and P^{-1} might favor different enumeration strategies. However, the cost of copies between vectors with additional global vector access makes this inferior to choosing one enumeration scheme, although it might decrease the throughput of other code parts (see Tables 3 and 4). Supported by these observations, we use a cell-centric enumeration throughout the remainder of this work.

3.3.5. Conclusions. In order to conclude the discussion on the different patch types for ASM, we present a comparison of their memory consumption and computational complexity in Table 5. The numbers are shown for both Cartesian and general meshes and have been verified with hardware counters (Treibig et al., 2010). The patch type also determines the requirements regarding communication: while cell-centric patches need n layers of ghost unknowns, vertex-star patches need p layers. In the latter case, however, the extension is needed in a single direction. Patch smoothers generally communicate twice during both gathering and scattering in order to collect partial results. In the case of RAS, the latter communication step can be skipped by definition. Furthermore, Figure 6 gives a throughput and memory-access analysis comparison for the different approaches. One can see that the cost of applying the cell-centric patch smoothers with $n = 1$ is comparable to

Table 3. Measurements of different ways to apply weights in the context of FDM with renumbering favoring cell-centric patches with $n = 2$. For the setup, see Table 2.

| | No weights (I) | | | post | | | symm | | |
|---------|----------------|-----|------|------|-----|------|------|-----|------|
| | r | w | t(s) | r | w | t(s) | r | w | t(s) |
| 1-g-p-m | 4.8 | 1.3 | 0.12 | 6.1 | 1.4 | 0.13 | 6.3 | 2.4 | 0.15 |
| 2-g-p-m | 12.0 | 2.7 | 0.42 | 14.0 | 3.1 | 0.45 | 15.5 | 4.1 | 0.47 |

Table 4. Measurements of different ways to apply weights in the context of FDM with renumbering favoring vertex-star patches. For the setup, see Table 2.

| | No weights (I) | | | post | | | symm | | |
|---------|----------------|-----|------|------|-----|------|------|-----|------|
| | r | w | t(s) | r | w | t(s) | r | w | t(s) |
| 1-g-p-m | 3.4 | 1.3 | 0.12 | 4.9 | 1.5 | 0.14 | 5.1 | 2.5 | 0.16 |
| v-g-p-m | 5.2 | 2.4 | 0.33 | 6.8 | 2.6 | 0.35 | 7.8 | 3.7 | 0.37 |

the one of the operator evaluation (3). Larger overlaps due to $n > 1$ or vertex-star patches lead to a drop in throughput by a factor of 4.2 and 2.9, respectively. Consequently, they are only beneficial if the number of smoothing steps and/or outer iterations can be reduced substantially. ASM with vertex-star patches tends to have a higher throughput than ASM with cell-centric patches ($n \geq 2$) up to $p = 6$. After that point, the throughput decreases due to increased work per unknown. In Figure 7, the performance is summarized by means of a roofline model (Williams et al., 2009) using measurements from performance counters. There are two main reasons why the performance limit (bandwidth) is not reached on the Intel system: on the one hand, indirect access

Table 5. Comparison of memory consumption and number of floating-point operations of cell integral and the considered patch smoothers. Values are given per cell, which correspond to p^d unique DoFs. For the sake of brevity, we introduce q , the size of the patch in each direction ($(p + 1)$ or $(p + 2n - 1)$ or $(2p - 1)$).

| | cell integral | | cell-centric patch ($n = 1$) | | cell-centric patch ($n > 1$) | | vertex-star patch | |
|-----------------------------------|---------------|---|--------------------------------|---------------------|---|-------|-------------------|------------|
| | data | FLOPs | data | FLOPs | data | FLOPs | data | FLOPs |
| gather/restrict cell operation | 3^d | — | 3^d | — | $(p + 2n - 1)^d$ | — | 3^d | — |
| - Cartesian | 0 | $\frac{4(2p+1)}{p+1} \cdot d \cdot (p+1)^{d+1}$ | ← | 0 | $q^d + \frac{2(2q-1)}{q} \cdot d \cdot q^{d+1}$ | → | | |
| - general | $d \cdot 3^d$ | $\mathcal{O}(dp^{d+1})$ | ← | $q^d + d \cdot q^2$ | $q^d + \frac{2(2q-1)}{q} \cdot d \cdot q^{d+1}$ | → | | |
| metrics/weights | 3^d | $(p+1)^d$ | 3^d | $(p+1)^d$ | p^d | p^d | 3^d | $(2p-1)^d$ |

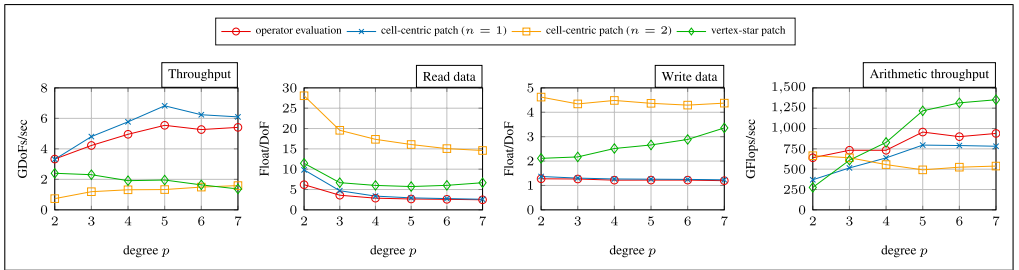


Figure 6. Comparison of throughput, read/write data per DoF, floating-point operations for operator evaluation, cell-centric patches, and vertex-star patches for different degrees p . Single node.

makes the load/store operations from/to main memory more expensive and, on the other hand, memory-intensive and compute-intensive sections are not completely overlapped by the hardware. A remedy to the last point would be to process matrix-free integral computations within a cell layer by layer, as proposed by [Kronbichler and Kormann \(2019\)](#).

3.4. Data locality in Chebyshev iterations

As a next algorithmic development, we explore the efficient evaluation of the Chebyshev iteration (2a)–(2c), which alternately executes A and P^{-1} together with vector updates. We propose different levels of optimizations, depending on the structure of A and P^{-1} . [Figure 9\(a\)](#) shows the read and write operations needed if both A and P^{-1} are considered as a black-box operation in terms of a BLAS level-2 interface, with the source vector ready in its entirety before the product is executed and similarly for the destination vector. The accumulated numbers of ideal read and write operations for sizes larger than the caches are $r \sim 9k$ and $w \sim 4k$. Note that only vector data are counted and the coefficient data of A and P^{-1} as well as non-perfect caching are ignored.

Since A and ASM are local operators on cells or around vertices, their evaluation can be interleaved with vector

updates. Furthermore, a point-Jacobi method applies the inverse diagonal of the system matrix A without any communication, which allows further optimizations. We use the point-Jacobi method as a reference, for which we shortly discuss an efficient implementation in the following. [Figure 9\(b\)–\(c\)](#) present two types of optimizations. We start with discussing optimization 2, which assumes that a diagonal preconditioner is given and A relies on a loop over cells with access to a compact set of unknowns. Here, all vector updates, including scaling by the diagonal matrix, can be performed in a post-operation (with zeroing during pre), which is comparable to the strategy of [Kronbichler and Allalen \(2018\)](#). The theoretical read/write data volume can be significantly reduced: $r \sim 4k$, $w \sim k$.

Completely merging A and P^{-1} , as in the case of a diagonal matrix, is not straightforward in the case of ASM: it requires nesting of two cell loops (also known as “power kernel”, see e.g., [Malas et al. \(2017\)](#) and the related work of [Akkurt et al. \(2022\)](#) for DG-type discretizations). Our preliminary investigations have shown that running the two cell loops in sequence is faster for higher-order elements, because the nested loops increase the active working set, resulting in the deterioration of the data locality of the outer loop particularly due to the MPI communication requirements. As a result, the

preferred algorithm is to let A write into a temporary vector, which is subsequently read by P^{-1} , requiring one write and three read operations in addition. One of the three additional read operations is related to the fact that x has to be read during A and P^{-1} . Nevertheless, since all other vector updates can be interleaved with the cell loops, the overall complexity can be reduced to $r \sim 7k$, $w \sim 2k$.

In addition to the theoretically derived quantities, we show experimentally measured data transfer and the obtained throughputs for $k = 3$ in Figure 8. A clear speedup is visible by using one of the optimizations from Figure 9: in the case of the point-Jacobi preconditioner, one smoothing step has the cost of about one operator evaluation, which corresponds to a speedup of 90%. In the case of ASM with $n = 1$, the speedup is more

moderate (20%–25%): the cost is bounded by the costs of two operator evaluations, since two cell loops are run in sequence. Not surprisingly for $n > 1$ and vertex-star patches, the cost of smoothing is dominated by ASM and the benefit of merging the vector updates decreases to 10%.

Figure 10 shows the benefits of the optimizations by varying k and the mesh type. It is clear that the cost of smoothing increases linearly in all cases. However, in the case of cheap smoothers, more iterations can be executed at the same costs. Figure 7 presents the performance of the optimized code in a roofline model, indicating a slight decrease in arithmetic intensity.

We conclude this section with some remarks on the cache-friendly global application of weights in ASM. As

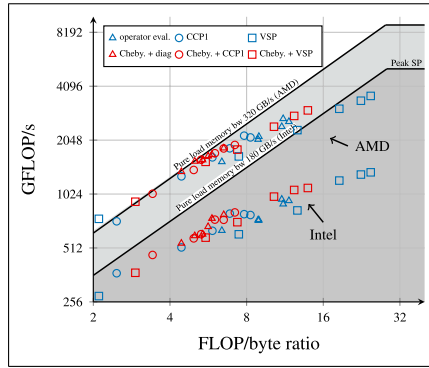


Figure 7. Roofline model of the application of the operator evaluation A , different P^{-1} , and Chebyshev iterations of degree $k = 3$ with different P^{-1} for the measurement data from Figures 6 and 8. Different points of the same category correspond to different polynomial degrees ($2 \leq p \leq 7$). Cell-centric patches with overlap $n = 2$ are not shown due to similar performance characteristics to the application of A . On the Intel system, the bandwidth and the number of floating-point operations are extracted from hardware-performance counters. On the AMD system, where direct measurements have not been possible, the memory access of the Intel system is assumed as a model.

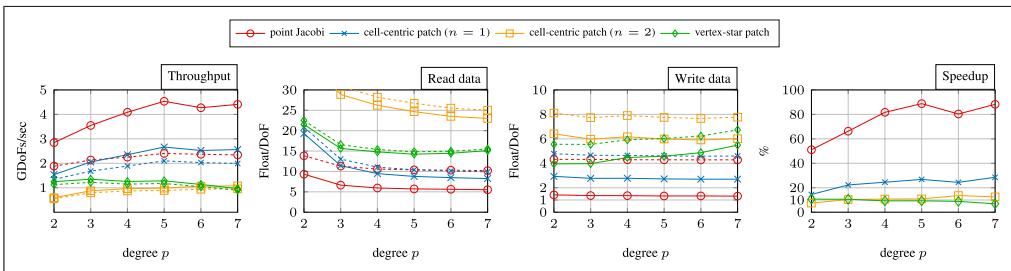


Figure 8. Comparison of throughput and read/write data of Chebyshev iterations of degree $k = 3$ for point Jacobi, cell-centric patches, and vertex-star patches for different degrees p and optimizations (base vs optimization 1 for patch smoothers and optimization 2 for point Jacobi). Dashed lines indicate the naive base algorithm, and solid lines indicate the best possible optimization. Single node. FDM with symm.

| | |
|--|-----------|
| (a) | |
| 1 : $x \leftarrow P^{-1}b$ | $1w + 2r$ |
| $x \leftarrow \beta x$ | $1w + 1r$ |
| <hr/> | |
| 2 : $t \leftarrow Ax$ | $1w + 2r$ |
| $t \leftarrow b - t$ | $1w + 2r$ |
| $y \leftarrow P^{-1}(t)$ | $1w + 2r$ |
| $y \leftarrow (1 + \alpha)x + \beta y$ | $1w + 2r$ |
| $x \leftrightarrow y$ | |
| <hr/> | |
| i : $t \leftarrow Ax$ | $1w + 2r$ |
| $t \leftarrow b - t$ | $1w + 2r$ |
| $u \leftarrow P^{-1}(t)$ | $1w + 2r$ |
| $y \leftarrow (1 + \alpha)x - \alpha y + \beta u$ | $1w + 3r$ |
| <hr/> | |
| (b) | |
| 1 : $x \leftarrow \beta P^{-1}b$ | $1w + 2r$ |
| 2 : $t \leftarrow b - Ax$ | $1w + 3r$ |
| $u \leftarrow (1 + \alpha)x + \beta P^{-1}t$ | $1w + 3r$ |
| $x \leftrightarrow u; y \leftrightarrow u$ | |
| <hr/> | |
| i : $t \leftarrow b - Ax$ | $1w + 3r$ |
| $u \leftarrow (1 + \alpha)x - \alpha y + \beta P^{-1}t$ | $1w + 4r$ |
| $x \leftrightarrow u; y \leftrightarrow u$ | |
| <hr/> | |
| (c) | |
| 1 : $x \leftarrow \beta P^{-1}b$ | $1w + 2r$ |
| 2 : $t \leftarrow (1 + \alpha)x + \beta P^{-1}(b - Ax)$ | $1w + 3r$ |
| $x \leftrightarrow t; y \leftrightarrow t$ | |
| <hr/> | |
| i : $t \leftarrow (1 + \alpha)x - \alpha y + \beta P^{-1}(b - Ax)$ | $1w + 4r$ |
| $x \leftrightarrow t; y \leftrightarrow t$ | |

Figure 9. Pseudo code of different versions of Chebyshev iteration, including vector read/write estimates. The values have been verified with hardware counters. The vector x contains x_i at the beginning and x_{i+1} at the end of a step. In the case of a Chebyshev iteration, the vector y contains x_{i-1} at the beginning of a step and x_i at the end of a step. The vectors t and u are temporary vectors. For the sake of simplicity, we skipped the indices of the Chebyshev coefficients α and β . (a) Base, (b) Optimization 1 (matrix-free P^{-1} , matrix-free A), and (c) Optimization 2 (matrix-free P^{-1} , matrix-free A).

discussed above, we can interleave this application with the matrix-vector multiplication. The resulting pre/post-operations are merged—as indicated in Figure 11—with the ones resulting from optimization 1 from Figure 9.

Note that the optimizations presented are also applicable to relaxation schemes with the update formula $x_i = \omega P^{-1}b$ and $x_{n+1} = x_n + \omega P^{-1}r_n$. The resulting pseudo code is shown in the appendix.

4. Application: Multigrid

In this section, we embed the developed and optimized smoothers into a p -multigrid solver with optimizations for large scales according to Munch et al. (2023).

The three basic steps of a two-level algorithm are (1) presmoothing, (2) coarse-grid correction, in which a related problem on a coarser grid is solved, and (3) postsmoothing. For the coarse-grid problem, the residual $b - Ax$ is computed and restricted. After the coarse solution, which might in turn invoke the two-level algorithm recursively, the resulting solution is prolonged and added to the fine solution. Nesting two-level algorithms recursively gives a multigrid algorithm. We use multigrid as preconditioner for a conjugate-gradient (CG) or GMRES(15) iterative solver, which implies that the initial guess on the levels is zero, allowing to skip the computation of the residual during the first iteration of presmoothing. We use the CG method for point Jacobi as well as for ASM with symmetric application of weights and GMRES with iterated classical Gram-Schmidt algorithm else.

In the setting of p -multigrid, the levels are defined by coarsening the polynomial degrees of the elements. The most common approach is to halve the polynomial degree (*bisect*), which reduces the number of unknowns per level by approximately 2^d . However, one can also decrease the

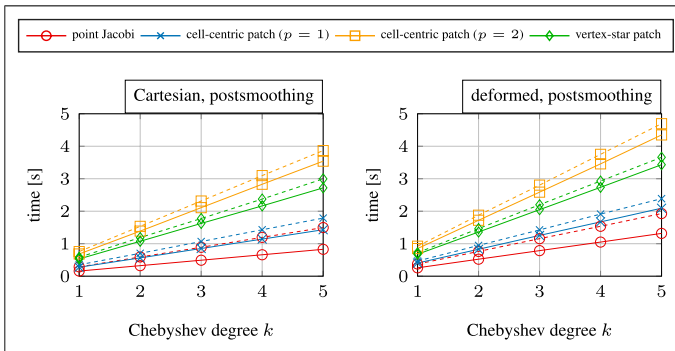


Figure 10. Time of 10 postsmoothing steps for different Chebyshev degrees, $p = 4$ and 68 MDof for Cartesian and deformed structured meshes. Dashed lines indicate the naive base algorithm, whereas solid lines indicate the best possible optimization. Single node. FDM with symm.

polynomial degree one by one (*by-one*) or skip refinement levels and jump to linear elements directly (*to-one*).

The cost of the multigrid solver can be modeled as the sum of the cost of each component

$$\mathcal{T} = N_{it}(\overline{\mathcal{T}}_{CG/GMRES}(N_{it}) + \mathcal{T}_V) \quad (4a)$$

$$\mathcal{T}_V = \mathcal{T}_{pre}(k) + \mathcal{T}_r + \mathcal{T}_{rest} + \mathcal{T}_{coarse} + \mathcal{T}_{pro} + \mathcal{T}_{post}(k) \quad (4b)$$

Here, the cost of pre- and postsmoothing ($\mathcal{T}_{pre} + \mathcal{T}_{post}$) increases—as discussed previously—linearly with the number of the smoothing steps, since $(2k - 1)$ matrix-vector products with A and $2k$ preconditioner applications have to be performed. The cost of the evaluation

of the residual is similar to the one of a matrix-vector product with A . The cost of the intergrid-transfer operators is influenced by the way the levels are constructed, however, again related to the cost of A , since they can be evaluated on the fly during a cell loop (Kronbichler and Wall, 2018; Munch et al., 2023). In the ideal case, the cost of the coarse grid decreases by $(p_f/p_c)^d$.

Phillips and Fischer (2022) recently advocated the usage of one-sided V-cycles that apply all smoothing steps during presmoothing. The cost of the V-cycle of this approach is also described by (4b), since then $\mathcal{T}'_{pre} = \mathcal{T}_{pre} + \mathcal{T}_{post}$ and $\mathcal{T}'_{post} = 0$. As this approach makes the overall multigrid preconditioner non-symmetric, we will only consider it for ASM with the weight application post.

As a conclusion of this introduction into multigrid, Figure 12 presents the profile of a single iteration of the solution process for ASM (symm and post application of the weights, $n = 1, k = 2$) for a regular (two-sided) V-cycle and a one-sided V-cycle. It is well visible that the main additional cost in the case of post is the increased cost of the outer solver (GMRES), which could be compensated by less iterations and coarse-grid solves, as will be investigated next.

In the following, we solve the Poisson equation with constant right-hand side and homogeneous Dirichlet boundary conditions, using three challenging meshes shown in Figure 13. We terminate after a relative residual reduction of 10^{-5} is reached. To increase the throughput, all operations in the multigrid V-cycle are run with FP32, while the outer solver is run with FP64 (Kronbichler and Jungkvist 2019). We have performed extensive experiments varying (1) the Chebyshev degree k , (2) the type of decreasing p , (3) point Jacobi or ASM (cell-centric patches with $n = 1$ and $n = 2$, vertex-star patches) as preconditioner for the Chebyshev iteration, (4) first or fourth kind of Chebyshev polynomials, and (5) one-sided

| (a) | |
|---|-----------|
| 1: $x \leftarrow \beta \tilde{W} P_e^{-1} b$ | $1w + 2r$ |
| 2: $t \leftarrow b - Ax$ | $1w + 3r$ |
| $u \leftarrow (1 + \alpha)x + \beta \tilde{W} P_e^{-1} t$ | $1w + 3r$ |
| $x \leftrightarrow u; y \leftrightarrow u$ | |
| i : $t \leftarrow b - Ax$ | $1w + 3r$ |
| $u \leftarrow (1 + \alpha)x - \alpha y + \beta \tilde{W} P_e^{-1} t$ | $1w + 4r$ |
| $x \leftrightarrow u; y \leftrightarrow u$ | |
| (b) | |
| 1: $(x, t) \leftarrow \beta \tilde{W} P_e^{-1} \tilde{W} b$ | $2w + 3r$ |
| 2: $t \leftarrow b - Ax$ | $1w + 3r$ |
| $(u, t) \leftarrow (1 + \alpha)x + \beta \tilde{W} P_e^{-1} \tilde{W} t$ | $2w + 3r$ |
| $x \leftrightarrow u; y \leftrightarrow u$ | |
| i : $t \leftarrow b - Ax$ | $2w + 3r$ |
| $(u, t) \leftarrow (1 + \alpha)x - \alpha y + \beta \tilde{W} P_e^{-1} \tilde{W} t$ | $2w + 4r$ |
| $x \leftrightarrow u; y \leftrightarrow u$ | |

Figure 11. Optimization I for a Chebyshev iteration with FDM and global weighting. Expressions with a tuple as result need to use a temporary vector. (a) Optimization I (FDM with global post, matrix-free A), (b) Optimization I (FDM with global symm, matrix-free A).

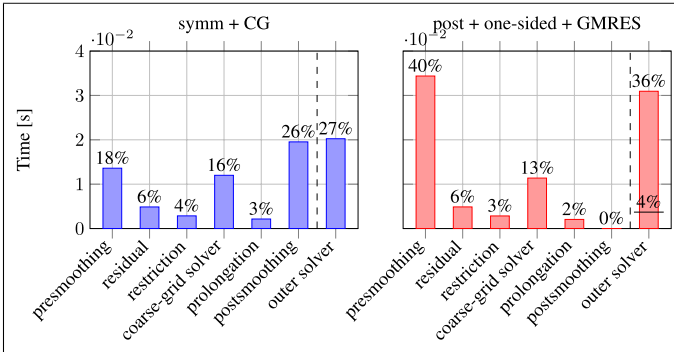
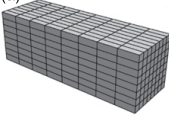
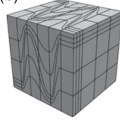


Figure 12. Profile of the finest multigrid level and the outer solver. The timings of GMRES include the orthogonalization step with the classical Gram-Schmidt algorithm and an additional preconditioner application to recover the solution.

| name | $p = 4$ | | | $p = 7$ | | |
|-------------|---------|---------|------------|---------|--------|------------|
| | L | cells | DoFs | L | cells | DoFs |
| anisotropic | 7 | 262,144 | 16,974,593 | 6 | 32,768 | 11,390,625 |
| Kershaw | 5 | 110,592 | 7,189,057 | 4 | 13824 | 4,826,809 |
| 3D ball | 5 | 131,072 | 8,438,273 | 4 | 16384 | 5,657,793 |

(a)


(b)


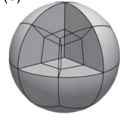
(c)


Figure 13. Considered 3D meshes: coarse version of (a) anisotropic Cartesian mesh ($x_2/x_0 = 10$), (b) Kershaw mesh for $\epsilon = 0.3$, and (c) a ball.

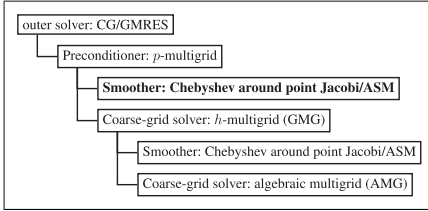


Figure 14. Diagram of the proposed solver.

versus two-sided V-cycles. In the following, we summarize the results and pick out some challenging mesh configurations. Detailed tables with all results can be found in the companion material, which is available in the accompanying GitHub repository.

As coarse-grid solver, we use geometric multigrid with the same smoothers as for p -multigrid. This implies that, in the case of $n = 2$, the smoother is built around FDM with $n = 1$ and, in the case of vertex-star patches, around the inverse diagonal of A . In the latter instance, we additionally increase the number of smoothing steps by 2. As coarse-grid solver of geometric multigrid, we use AMG from the ML (cf. Gee et al. (2006)) package, which, in the presented experiments, falls back to a direct solver due to the small coarse-grid sizes. We note that switching directly to AMG as coarse-grid solver of p -multigrid, instead of GMG, would be possible and would enable the application of the solver to more general unstructured meshes. However, this requires domain-specific parameter tuning, which we defer to future work. Figure 14 summarizes the structure of the solver.

4.1. Anisotropic Cartesian mesh

In the first experiment, we solve the Poisson equation on an anisotropic Cartesian mesh, which is uniform in x_0 and x_1

Table 6. Best configurations for anisotropic Cartesian mesh: $p = 4$. The triple in the parentheses specifies the number of pre- and postsmoothing steps as well as the kind of the Chebyshev polynomial used. Values highlighted in bold face indicate the fastest setting for a given configuration.

| Type | Fastest configuration | #it | t(s) | $x_0 = x_2$ | |
|-------------------|--------------------------------|-----------|-------------|-------------|------|
| | | | | #it | t(s) |
| Diagonal | (5, 5, 4), by-one | 60 | 6.52 | 3 | 0.22 |
| FDM (symm) | (3, 3, 1), by-one, CCPI | 34 | 3.89 | 4 | 0.23 |
| FDM (post) | (8, 0, 4), by-one, CCPI | 23 | 3.68 | 4 | 0.22 |

Table 7. Anisotropic Cartesian mesh: $p = 7$.

| Type | Fastest configuration | #it | t(s) | $x_0 = x_2$ | |
|-------------------|---------------------------------|-----------|-------------|-------------|------|
| | | | | #it | t(s) |
| Diagonal | (4, 4, 1), by-one | 58 | 4.31 | 4 | 0.17 |
| FDM (symm) | (4, 4, 1), by-one, CCPI | 20 | 2.21 | 4 | 0.15 |
| FDM (post) | (10, 0, 4), by-one, CCPI | 13 | 1.88 | 4 | 0.18 |

directions but stretched in x_2 direction by a factor of 50. The companion material also contains results for $x_2/x_0 \in \{1, 2, 5, 10, 20\}$. Tables 6–7 present, for $p = 4$ and $p = 7$, the best configurations we have identified by parameter tuning. Furthermore, run times are shown for a simulation without stretch in the last two table columns. The following observations can be made. Solving the anisotropic problem is at least $10\times$ more expensive. The reason for this is two-fold: on the one hand, the number of iterations needed increases and, on the other hand, the optimal preconditioner is more expensive due to the need to choose more expensive smoothers (higher overlap, more smoothing steps). Overall, overlap $n = 1$ and the coarsening sequence by-one are the most beneficial. In our experiments, the application of weights as postprocessing step with one-sided V-cycles and 4th-kind Chebyshev polynomial turned out to be the fastest. Overall, the sequence to-one requires the highest number of iterations, resulting in the lowest throughput.

Note: In the literature, semi-coarsening or line smoothers have been proposed to ensure robustness for anisotropic problems Trottenberg et al. (2000). These approaches are more robust for anisotropic meshes than the proposed patch smoothers, but their implementation in general finite-element methods and with high-order bases is not trivial, especially regarding the concept of line smoothers.

4.2. Anisotropic structured Kershaw mesh

In this subsection, we consider a Kershaw mesh (cf. Kershaw (1981)), the basis of the BPS benchmarks

Table 8. Kershaw mesh: $p = 4$.

| Type | Fastest configuration | #it | t(s) | $\epsilon = 1$ | |
|-------------------|--------------------------------|-----------|-------------|----------------|------|
| | | | | #it | t(s) |
| Diagonal | (3, 3, 1), bisect | 75 | 3.41 | 3 | 0.10 |
| FDM (symm) | (2, 2, 1), bisect, CCPI | 51 | 2.27 | 4 | 0.10 |
| FDM (post) | (3, 3, 1), bisect, CCPI | 36 | 2.23 | 4 | 0.10 |

Table 9. Kershaw mesh: $p = 7$.

| Type | Fastest configuration | #it | t(s) | $\epsilon = 1$ | |
|-------------------|--------------------------------|-----------|-------------|----------------|------|
| | | | | #it | t(s) |
| Diagonal | (4, 4, 1), bisect | 94 | 2.13 | 4 | 0.07 |
| FDM (symm) | (3, 3, 1), bisect, CCPI | 53 | 1.36 | 4 | 0.07 |
| FDM (post) | (4, 4, 1), bisect, CCPI | 40 | 1.37 | 3 | 0.07 |

proposed by the Center for Efficient Exascale Discretization (CEED) within the DOE Exascale Computing Project (ECP) (cf. Kolev et al., 2021a) in order to fairly compare high-order finite-element implementations and libraries. On the coarsest level, it has six subdivisions in each direction and the parameter $0 < \epsilon \leq 1$ determines the local aspect ratios and makes the solution increasingly challenging as ϵ decreases. We analyze the cases with $L = 5$ for $p = 4$ and $L = 4$ for $p = 7$, for $\epsilon = 0.3$. The companion material also contains results for $\epsilon \in \{0.5, 0.7, 0.9, 0.99, 1.0\}$. The local anisotropy is 146/144. A coarse version of the mesh is shown in Figure 13(b).

Tables 8–9 present the results for $p = 4$ and $p = 7$. As a reference, we again show the timings on a mesh with $\epsilon = 1$ (Cartesian mesh). While the times in the Cartesian case are similar for the optimal point-Jacobi and FDM setups, we see, for deformed meshes, a speedup of 53% and 56% in favor of FDM. In the simulation, the combination of bisect, two-sided V-cycle, and $n = 1$ turned out to be the fastest. For $p = 7$, similar observations can be made.

Table 10 shows, for $p = 4$, the best configurations for cell-centric FDM with $n = 1$ and $n = 2$ as well as for vertex-star FDM. Overall, one can observe that cheaper smoothers (diagonal, cell-centric FDM with $n = 1$) need more iterations, and vertex-star patches are somewhat slower than the cell-centric variants, despite being more efficient per iteration ($n > 1$). The increase in linear iterations might be related to how we construct the Cartesian surrogate meshes, to the skip of the boundary patches and to the coarse-grid solver (h -multigrid with Chebyshev/diagonal); nevertheless, our observations are consistent with those from the literature (cf. Witte et al. (2021)).

4.2.1. Cross-platform validation. As cross-platform validation, we present timings obtained on a dual-socket AMD

Table 10. Kershaw mesh: $p = 4$; alternative categorization.

| Type | Fastest configuration | #it | t(s) | $\epsilon = 1$ | |
|-------------------|--------------------------------|-----------|-------------|----------------|------|
| | | | | #it | t(s) |
| Diagonal | (3, 3, 1), bisect | 75 | 3.41 | 3 | 0.10 |
| FDM (CCP1) | (3, 3, 1), bisect, post | 36 | 2.23 | 4 | 0.10 |
| FDM (CCP2) | (2, 2, 1), bisect, post | 40 | 2.77 | 5 | 0.15 |
| FDM (VSP) | (2, 2, 1), bisect, symm | 49 | 2.82 | 4 | 0.13 |

Table 11. Kershaw mesh: $p = 4$; alternative categorization; AMD.

| Type | configuration | #it | t(s) | $\epsilon = 1$ | |
|-------------------|--------------------------------|-----------|-------------|----------------|-------|
| | | | | #it | t(s) |
| Diagonal | (3, 3, 1), bisect | 75 | 1.13 | 3 | 0.035 |
| FDM (CCP1) | (3, 3, 1), bisect, post | 36 | 0.90 | 4 | 0.038 |
| FDM (CCP2) | (2, 2, 1), bisect, post | 40 | 1.14 | 5 | 0.068 |
| FDM (VSP) | (2, 2, 1), bisect, symm | 49 | 1.18 | 4 | 0.051 |

EPYC 7713 compute node. The AMD CPU consists of 2×64 cores running at 2.2 GHz and uses codes compiled for the AVX2 instruction-set extension (8-wide SIMD for FP32). This gives an arithmetic peak performance of 9.0 TFlop/s. It has a measured STREAM triad bandwidth of 320–340 GB/s. Table 11 shows timings for $p = 4$ for the best configurations identified for the Intel processor in Table 10. Compared to the Intel system, one can observe speedup factors of 2.4–2.5 for patch smoothers and a $3.0\times$ speedup in the case of the diagonal. These values are somewhat better than what one would expect from increased bandwidth and compute performance ($1.8\times$ each) and indicate that non-arithmetic parts of the code are executed more quickly, leading to a better use of the memory bandwidth. The results also show that the AMD system favors the diagonal preconditioner with its lower arithmetic intensity, resulting in times that are slightly lower than the ones for cell-centric patches with $n = 2$ and for vertex-star patches. This observation can be explained by a more balanced cache system on the AMD node, with higher L3 bandwidth (but lower L2 bandwidth). This is also underlined by the roofline model in Figure 7, where the performance limit (bandwidth) is reached for most data points of operator evaluation and cell-centric ASM.

4.2.2. Comparison with NekRS. A similar setup (36 cells in each direction, $p = 7$, linear mapping, RAS, GMRES(30) + p -multigrid + AMG, overlap $n = 2$) was recently also run with NekRS (cf. Phillips and Fischer (2022)) on six GPUs on Summit. The comparison with our implementation run on six compute nodes on SuperMUC NG is as follows:

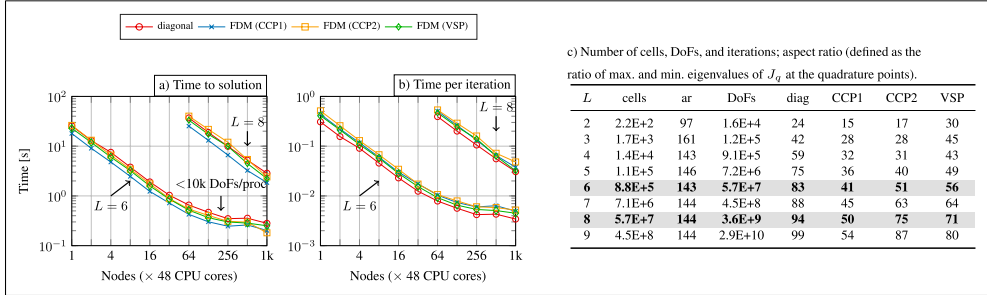


Figure 15. Strong-scaling comparison of different smoothers for the Kershaw geometry with $\epsilon = 0.3$, $L = 6/L = 8$, $p = 4$. (a) Time to solution, (b) Time per iteration. (c) Number of cells, DoFs, and iterations; aspect ratio (defined as the ratio of max. and min. eigenvalues of J_q at the quadrature points).

| ϵ | config. | NekRS | | | Present code | | |
|------------|------------|-------|------|---------|--------------|------|---------|
| | | it | t(s) | t/it(s) | it | t(s) | t/it(s) |
| 1.0 | (2, 2, 1) | 8 | 0.09 | 0.011 | 13 | 0.19 | 0.014 |
| 0.3 | (5, 5, 1) | 28 | 0.67 | 0.024 | 47 | 1.57 | 0.033 |
| 0.05 | (12, 0, 4) | 88 | 2.40 | 0.027 | 67 | 2.61 | 0.037 |

Even though the number of iterations is different, possibly due to different Chebyshev weights and AMG implementations (BoomerAMG vs ML), one can see that the times per iteration are comparable, underlying the high quality of our implementation. The higher throughput on the GPU (30%) is expected, since overlap $n = 2$ benefits from the higher bandwidth of the GPU ($> 3.5 \times$).

4.2.3. Parallel scaling. The focus of this publication is the node-level optimization of smoothers. In this subsection, we shortly investigate their parallel scalability. The numerical experiments are run on a dual-socket 24-core Intel Xeon Platinum 8174 (Skylake) system of the supercomputer SuperMUC-NG (measured STREAM triad memory throughput of 205 GB/s) with up to 1024 compute nodes (49k processes).

For $p = 4$, we compare the best configurations of point Jacobi, cell-centric patches with $n = 1$ and $n = 2$ as well as vertex-star patches, which we have identified on a single node for a given mesh configuration (Table 10).

Figure 15 shows the time to solution, the time of each iteration, and the number of iterations. One can see that cell-centric patch smoothing with $n = 1$ is the most competitive option overall despite the higher cost per iteration, compared to point Jacobi in terms of communication steps. This is due to a lower iteration count. Cell-centric smoothing with $n = 2$ has the lowest throughput both on a single node and in terms of scaling. Overall, we reach excellent strong

Table 12. Unstructured mesh of a 3D ball: $p = 4$.

| Type | Fastest configuration | #it | t(s) |
|-------------------|--------------------------------|----------|-------------|
| Diagonal | (2, 2, 4), bisect | 6 | 0.27 |
| FDM (symm) | (2, 2, 4), bisect, CCP1 | 4 | 0.22 |
| FDM (post) | (1, 1, 1), bisect, CCP1 | 6 | 0.26 |

Table 13. Unstructured mesh of a 3D ball: $p = 7$.

| Type | Fastest configuration | #it | t(s) |
|-------------------|--------------------------------|----------|-------------|
| Diagonal | (4, 4, 4), by-one | 3 | 0.20 |
| FDM (symm) | (2, 2, 4), bisect, CCP1 | 5 | 0.14 |
| FDM (post) | (1, 1, 1), bisect, CCP1 | 7 | 0.16 |

scalability: we get more than 66/83% parallel efficiency upon an increase in the number of nodes by a factor of 64/16.

However, it is visible that the solver is not robust for increasing number of refinements. This is particularly the case for cell-centric ($n > 1$) and vertex-star patches, for which we use weaker smoothers on the coarser levels. A remedy would be to increase the number of smoothing steps on the coarser levels or to apply h -multigrid (before coarsening p). Using p -multigrid and switching to AMG directly is less robust without adjusting the AMG parameters. Detailed investigations on these topics are deferred to future work.

We conclude this subsection by discussing RAS. In our simulations, RAS requires fewer iterations in certain cases, whereas in other cases, normal (postprocessing) averaging needs fewer iterations. This observation is consistent with the literature (Phillips and Fischer, 2022). Nevertheless, RAS allows to skip one communication step. We observed—in comparison to post for $n = 2$ and 64 compute nodes—a speedup of up to 20% at the scaling limit. Further speedup

might be possible, taking into consideration that we do not specialize the weighting infrastructure and only disable communication. However, we would like to note that using RAS on the levels of the coarse (h -multigrid) solver increases the number of iterations so that we apply the post-strategy on these levels.

4.3. Unstructured mesh of a 3D ball

As a concluding example, we show the results of simulations on a mesh of a 3D ball. The setup is less challenging than the first two examples in terms of anisotropy and is intended to demonstrate that the algorithms developed in this work—with the exception of the vertex-star patches—are easily extensible to the unstructured case. Tables 12–13 show the results for $p = 4$ and $p = 7$. Overall, the timings of using diagonal and ASM with cell-centric patches with $n = 1$ are very competitive, with a slight advantage of the ASM case. The fact that ASM can compete with a highly optimized point-Jacobi implementation underlies the high quality of our ASM implementations, which are built around the cache-optimized and low-overhead concepts proposed in this publication.

5. Conclusions and outlook

We have presented optimized implementations of the additive Schwarz method accelerated by Chebyshev sweeps for different types of patches in the context of p -multigrid for high-order FEM. The proposed optimizations contain the compression of infrastructural data structures and the improvement of data locality by working on index ranges during weighting required by ASM and during the Chebyshev vector updates. An extensive parameter study has been conducted, showing that ASM built around cell-centric patches can outperform optimized point-Jacobi-based smoothers even for simple setups, although the latter allow for more node-level optimizations. The results are inconclusive regarding the type of weighting in ASM, since the additional costs (more iterations vs GMRES incl. orthogonalization) seem to balance each other.

As a concluding remark, one can state that finding optimal sets of parameters for multigrid (smoothers) is a non-trivial task due to the large search space, the problem dependency as well as the hardware properties. In large production runs, an auto-tuning step—as proposed by Phillips et al. (2022)—is unavoidable in order to reach the optimal usage of hardware and the minimum time to solution.

A natural future extension of the present work concerns the case of locally refined meshes with hanging nodes, where additional developments to incorporate continuity constraints on a patch level are needed and conclusions regarding performance or load balancing might change.

Acknowledgements

The authors acknowledge collaboration with Maximilian Bergbauer, Ivo Dravins, Nikas Fehn, Guido Kanschat, Magdalena Schreter–Fleischhacker, Michał Wichrowski, and Julius Witte as well as the deal.II community. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (LRZ, www.lrz.de) through project id pr83te.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the Bayerisches Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen (KONWIHR) through the project “High-order matrix-free finite element implementations with hybrid parallelization and improved data locality.”

ORCID iDs

Peter Munch  <https://orcid.org/0000-0003-2368-8533>

Martin Kronbichler  <https://orcid.org/0000-0001-8406-835X>

Supplemental Material

Supplemental material for this article is available online.

Note

1. <https://github.com/peterrum/dealii-asm>

References

- Adams M, Brezina M, Hu J, et al. (2003) Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *Journal of Computational Physics* 188(1): 593–610.
- Akkurt S, Witherden F and Vincent P (2022) Cache blocking strategies applied to flux reconstruction. *Computer Physics Communications* 271: 108193.
- Anderson R, Andrej J, Barker A, et al. (2021) MFEM: a modular finite element methods library. *Computers & Mathematics with Applications* 81: 42–74.
- Arndt D, Fehn N, Guido K, et al. (2020) ExaDG: high-order discontinuous Galerkin for the exa-scale. In: Bungartz HJ, Reiz S, Uekermann B, et al. (eds) *Software for Exascale Computing - SPPEXA 2016–2019*. Cham: Springer International Publishing, pp. 189–224.
- Arndt D, Bangerth W, Davydov D, et al. (2021) The deal. II finite element library: design, features, and insights. *Computers & Mathematics with Applications* 81: 407–422.

- Arndt D, Bangerth W, Feder M, et al. (2022) The deal. II library, version 9.4. *Journal of Numerical Mathematics* 30(3): 231–246.
- Bangerth W, Burstedde C, Heister T, et al. (2011) Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Transactions on Mathematical Software* 38(1): 1–28.
- Brubeck PD and Farrell PE (2022) A scalable and robust vertex-star relaxation for high-order FEM. *SIAM Journal on Scientific Computing* 44: A2991–A3017.
- Cai X-C and Sarkis M (1999) A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing* 21(2): 792–797.
- Chalmers N, Mishra A, McDougall D, et al. (2023) HipBone: a performance-portable graphics processing unit-accelerated C++ version of the NekBone benchmark. *The International Journal of High Performance Computing Applications* 37(5): 10943420231178552.
- Couzy W (1995) *Spectral Element Discretization of the Unsteady Navier-Stokes Equations and its Iterative Solution on Parallel Computers*. Lausanne: EPFL. Technical Report. Available at: https://infoscience.epfl.ch/record/31858/files/EPFL_TH1380.pdf
- Deville MO, Fischer PF and Mund EH (2002) *High-order Methods for Incompressible Fluid Flow*. Cambridge: Cambridge University Press.
- Fehn N, Munch P, Wall WA, et al. (2020) Hybrid multigrid methods for high-order discontinuous Galerkin discretizations. *Journal of Computational Physics* 415: 109538.
- Fischer PF, Tufo HM and Miller NI (2000) An overlapping Schwarz method for spectral element simulation of three-dimensional incompressible flows. *Parallel Solution of Partial Differential Equations*. Berlin: Springer, pp. 159–180.
- Fischer P, Kerkemeier S, Min M, et al. (2022) NekRS, a GPU-accelerated spectral element Navier–Stokes Solver. *Parallel Computing* 114: 102982.
- Gee MW, Siefert CM, Hu JJ, et al. (2006) ML 5.0 smoothed aggregation user’s guide. Technical Report. Available at: https://www.researchgate.net/profile/Michael-Gee-3/publication/51992105_ML_50_Smoothed_Aggregation_User’s_Guide/links/0046353395404c2655000000/ML-50-Smoothed-Aggregation-Users-Guide.pdf
- Gholami A, Malhotra D, Sundar H, et al. (2016) FFT, FMM, or multigrid? A comparative study of state-of-the-art Poisson solvers for uniform and nonuniform grids in the unit cube. *SIAM Journal on Scientific Computing* 38: C288–C306.
- Kershaw DS (1981) Differencing of the diffusion equation in Lagrangian hydrodynamic codes. *Journal of Computational Physics* 39(2): 375–395.
- Kolev T, Fischer P, Austin AP, et al. (2021a) CEED ECP Milestone Report: High-order Algorithmic Developments and Optimizations for Large-scale GPU-accelerated Simulations. Technical Report. Available at: <https://www.osti.gov/servlets/purl/1845639>
- Kolev T, Fischer P, Min M, et al. (2021b) Efficient exascale discretizations: high-order finite element methods. *The International Journal of High Performance Computing Applications* 35(6): 527–552.
- Kronbichler M and Allalen M (2018) Efficient high-order discontinuous Galerkin finite elements with matrix-free implementations. *Advances and New Trends in Environmental Informatics*. Berlin: Springer, pp. 89–110.
- Kronbichler M and Kormann K (2012) A generic interface for parallel cell-based finite element operator application. *Computers & Fluids* 63: 135–147.
- Kronbichler M and Kormann K (2019) Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM Transactions on Mathematical Software* 45(3): 1–40.
- Kronbichler M and Ljungkvist K (2019) Multigrid for matrix-free high-order finite element computations on graphics processors. *ACM Transactions on Parallel Computing* 6(1): 1–32.
- Kronbichler M and Wall WA (2018) A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers. *SIAM Journal on Scientific Computing* 40: A3423–A3448.
- Kronbichler M, Sashko D and Munch P (2022) Enhancing data locality of the conjugate gradient method for high-order matrix-free finite-element implementations. *The International Journal of High Performance Computing Applications* 37: 61–81.
- Ljungkvist K (2017) Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes. In: *HPC ’17: Proceedings of the 25th High Performance Computing Symposium*. San Diego, CA, USA, June 26–28, 2019: Society for Computer Simulation International, pp. 1–12.
- Loisel S, Nabben R and Szyld DB (2008) On hybrid multigrid-Schwarz algorithms. *Journal of Scientific Computing* 36(2): 165–175.
- Lottes J (2022) Optimal polynomial smoothers for multigrid V-cycles. arXiv. Available at: <https://arxiv.org/abs/2202.08830>
- Lottes JW and Fischer PF (2005) Hybrid multigrid/Schwarz algorithms for the spectral element method. *Journal of Scientific Computing* 24: 45–78.
- Lynch RE, Rice JR and Thomas DH (1964) Direct solution of partial difference equations by tensor product methods. *Numerische Mathematik* 6: 185–199.
- Malas TM, Hager G, Ltaief H, et al. (2017) Multidimensional intratile parallelization for memory-starved stencil computations. *ACM Transactions on Parallel Computing* 4(12): 1–32.
- Melenk JM, Gerdes K and Schwab C (2001) Fully discrete hp-finite elements: fast quadrature. *Computer Methods in Applied Mechanics and Engineering* 190: 4339–4364.
- Munch P, Kormann K and Kronbichler M (2021) Hyper.deal: an efficient, matrix-free finite-element library for high-dimensional partial differential equations. *ACM Transactions on Mathematical Software* 47: 1–34.

- Munch P, Ljungkvist K and Kronbichler M (2022) Efficient application of hanging-node constraints for matrix-free high-order fem computations on CPU and GPU. In: Proceedings of the High performance computing: 37th international conference, ISC high performance 2022, Hamburg, Germany, May 29–June 2, 2022, pp. 133–152. Springer.
- Munch P, Heister T, Prieto Saavedra L, et al. (2023) Efficient distributed matrix-free multigrid methods on locally refined meshes for FEM computations. *ACM Transactions on Parallel Computing* 10: 1–38.
- Orszag SA (1980) Spectral methods for problems in complex geometries. *Journal of Computational Physics* 37: 70–92.
- Phillips M and Fischer P (2022) Optimal Chebyshev smoothers and one-sided V-cycles. arXiv. Available at: <https://arxiv.org/abs/2210.03179>
- Phillips M, Kerkemeier S and Fischer P (2022) Tuning spectral element preconditioners for parallel scalability on GPUs. *Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing*. Philadelphia, PA: SIAM, pp. 37–48.
- Scroggs MW, Dokken JS, Richardson CN, et al. (2022) Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes. *ACM Transactions on Mathematical Software* 48(2): 1–23.
- Stiller J (2017) Nonuniformly weighted Schwarz smoothers for spectral element multigrid. *Journal of Scientific Computing* 72: 81–96.
- Treibig J, Hager G and Wellein G (2010) LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of PSTI2010, San Diego, CA, 13–16 September 2010, pp. 207–216.
- Trottenberg U, Oosterlee CW and Schuller A (2000) *Multigrid*. Amsterdam: Elsevier.
- Williams S, Waterman A and Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* 52: 65–76.
- Witte J, Arndt D and Kanschat G (2021) Fast tensor product Schwarz smoothers for high-order discontinuous Galerkin methods. *Computational Methods in Applied Mathematics* 21(3): 709–728.

Author biographies

Peter Munch is a research associate and PhD student at University of Augsburg. He is one of the principal developers of the open-source finite-element library deal.II. His research interests lie in the fields of high-performance computing, scientific software development, computational fluid mechanics, and computational plasma physics—with a focus on matrix-free methods, software design, and iterative solvers.

Martin Kronbichler is a Professor at Ruhr University Bochum, Germany. He holds a PhD degree in scientific computing with specialization in numerical analysis from Uppsala University, Sweden (2012). His research interests include high-order finite element methods for flow problems with matrix-free implementations, efficient numerical linear algebra, and their parallel and high-performance implementation on emerging exascale hardware using generic numerical software.

Appendix

Figure 16 shows pre/post-optimizations for a basic relaxation iteration $x_1 = \omega P^{-1}b$, $x_{n+1} = x_n + \omega P^{-1}r_n$ for $n < k$, with the relaxation parameter ω .

| Base: | | | |
|---|-----|--|-----------|
| 1: | x | $\leftarrow P^{-1}b$ | $1w + 2r$ |
| | x | $\leftarrow \omega x$ | $1w + 1r$ |
| i : | t | $\leftarrow Ax$ | $1w + 2r$ |
| | t | $\leftarrow b - t$ | $1w + 2r$ |
| | u | $\leftarrow P^{-1}(t)$ | $1w + 2r$ |
| | x | $\leftarrow x + \omega u$ | $1w + 2r$ |
| Opt. 1 (matrix-free P^{-1} and A): | | | |
| 1: | x | $\leftarrow \omega P^{-1}b$ | $1w + 2r$ |
| i : | t | $\leftarrow \omega(b - Ax)$ | $1w + 3r$ |
| | x | $\leftarrow x + P^{-1}t$ | $1w + 2r$ |
| Opt. 2 (diagonal P , matrix-free A): | | | |
| 1: | x | $\leftarrow \omega P^{-1}b$ | $1w + 2r$ |
| i : | t | $\leftarrow x + \omega P^{-1}(b - Ax)$ | $1w + 3r$ |
| | x | $\leftrightarrow t$ | |

Figure 16. Pseudo code of different versions of relaxation iteration, including vector read/write estimates. Comments of Figure 11 also hold here.

Paper V 

STAGE-PARALLEL FULLY IMPLICIT RUNGE–KUTTA IMPLEMENTATIONS WITH OPTIMAL MULTILEVEL PRECONDITIONERS AT THE SCALING LIMIT*

PETER MUNCH[†], IVO DRAVINS[‡], MARTIN KRONBICHLER[§], AND MAYA NEYTCHEVA[‡]

Dedicated to the memory of Owe Axelsson

Abstract. We present an implementation of a stage-parallel preconditioner for Radau IIA type fully implicit Runge–Kutta methods, which approximates the inverse of the Runge–Kutta matrix A_Q from the Butcher tableau by the lower triangular matrix resulting from an LU decomposition and diagonalizes the system with as many blocks as stages. For the transformed system, we employ a block preconditioner where each block is distributed and solved by a subgroup of processes in parallel. For combination of partial results, we use either a communication pattern resembling Cannon’s algorithm or shared memory. A performance model and a large set of performance studies (including strong-scaling runs with up to 150k processes on 3k compute nodes) conducted for a time-dependent heat problem, using matrix-free finite element methods, indicate that the stage-parallel implementation can reach higher throughputs near the scaling limit. The achievable speedup increases linearly with the number of stages and is bounded by the number of stages. Furthermore, we show that the presented stage-parallel concepts are also applicable to the case that A_Q is directly diagonalized, which requires either complex arithmetic or solutions of two-by-two blocks, both exposing about half the parallelism. Alternatively to distributing stages and assigning them to distinct processes, we discuss the possibility of batching operations from different stages together.

Key words. implicit Runge–Kutta methods, Radau quadrature, stage-parallel preconditioning, finite element methods, matrix-free methods, geometric multigrid, massively parallel

MSC codes. 65Y05, 65M55, 68W10

DOI. 10.1137/22M1503270

1. Introduction. Runge–Kutta methods are widely used time-integration schemes to solve ordinary differential equations (ODEs) of the form

$$\frac{dy}{dt} = f(t, y).$$

We consider a partial differential equation rewritten as a system of ODEs, after using finite element methods (FEM) to discretize in space. The basic algorithm is to advance

* Received by the editors June 15, 2022; accepted for publication (in revised form) December 20, 2022; published electronically July 18, 2023.

<https://doi.org/10.1137/22M1503270>

Funding: This work was supported by the Bayerisches Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen (KONWIHR) through the project “High-order matrix-free finite element implementations with hybrid parallelization and improved data locality.” The Gauss Centre for Supercomputing e.V. (<https://www.gauss-centre.eu>) funded this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (LRZ, <https://www.lrz.de>) through project id pr83te. The work of the second author (fully) and the fourth author (partly) was supported by research grant VR-2017-03749, financed by the Swedish Research Council.

[†]Corresponding author. Helmholtz-Zentrum Hereon, Geesthacht, 21502, and High-Performance Scientific Computing, University of Augsburg, Augsburg, 86159, Germany (peter.muench@uni-a.de).

[‡]Department of Information Technology, Uppsala University, Uppsala, SE-75105, Sweden (ivo.dravins@it.uu.se, maya.neytcheva@it.uu.se).

[§]High-Performance Scientific Computing, University of Augsburg, Augsburg, 86159, Germany, and Uppsala University, Uppsala, SE-75105, Sweden (martin.kronbichler@uni-a.de).

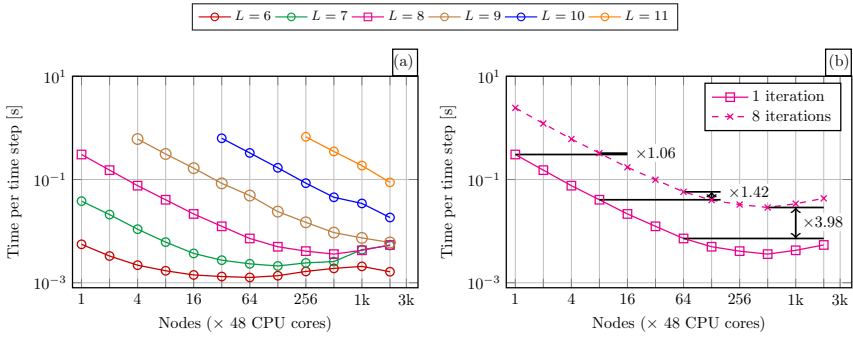


FIG. 1. (a) Time of a single conjugate gradient iteration preconditioned by GMG for different numbers of refinements $6 \leq L \leq 11$ and linear Lagrange elements ($k = 1$). (b) Example visualizing the benefit of stage parallelism for $Q = 8$ by comparing the time of a single iteration with the one of eight iterations and providing idealized speedups in the case that Q iterations are solved in parallel by Q subgroups.

the solution to the next time step by a linear combination of Q intermediate stage function values:

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \tau \sum_{1 \leq q \leq Q} b_q \mathbf{k}_q \quad \text{with} \quad \mathbf{k}_i = f \left(t_n + c_i \tau, \mathbf{y}_n + \tau \sum_{1 \leq j \leq Q} a_{ij} \mathbf{k}_j \right),$$

where t_n is the time at time step n and τ is the current time step size. The Butcher

tableau $\begin{array}{c|c} \mathbf{c}_Q & A_Q \\ \hline & \mathbf{b}_Q^T \end{array}$ is a compact notation for these methods in terms of a matrix A_Q as well as two vectors \mathbf{b}_Q and \mathbf{c}_Q .

There is a vast literature on optimizing Runge–Kutta methods. The investigations include improving the accuracy and the stability region as well as performance optimization. Low-storage Runge–Kutta methods [20], for instance, update the solution step by step so that the intermediate results for all stages need not be stored simultaneously, which might be an advantage for memory-intensive applications, e.g., computational plasma physics [29]. Implicit Runge–Kutta methods (IRK) require the solution of systems of increased size so that the development of efficient solvers is crucial [1, 5, 6, 9, 11, 12, 14, 19, 30, 31, 32].

In this work, we investigate solvers for fully implicit Runge–Kutta methods, focusing on the parallelization of the solution process over all the stages [18]. Stage-parallel approaches and, similarly, parallel-in-time approaches [10, 25] (out of the scope of this work) are motivated by the disappointing scaling of algorithms, e.g., of iterative solvers, on distributed systems when parallelism is only exploited in the spatial domain. These algorithms cannot run faster than a certain threshold even if more hardware resources are added. Figure 1(a) shows, as an example, the times for one conjugate gradient iteration preconditioned by a single V-cycle of geometric multigrid (GMG) we use in the experimental section when run across a large range of number of processes. It is clearly visible that the times flatten out when the work per process becomes too little. This phenomenon is also known as the “scaling limit.” In the context of GMG, the minimum time of one iteration is approximately proportional to the number of levels. Since the total solution time of IRK is approximately the

solution time accumulated over all stages run sequentially, the behavior of the overall solution time is similar under the assumption that each block corresponding to a stage can be solved by GMG.

As an alternative to the sequential solution process of each stage, one could assume that it is possible to solve the stages by Q process groups with GMG independently. In consequence, while it might take longer to solve for one stage, the stages can be solved in parallel. A speedup can be expected if the solution time of one stage does not increase significantly; if the slowdown is $\sim Q$, no speedup can be expected. Figure 1(b) presents, as an example, the solution times for a single GMG iteration and for eight GMG iterations (assuming $Q = 8$). In addition, it shows ideal speedups we can expect by solving one GMG iteration in parallel by each subgroup with one-eighth of the processes. The speedup increases with an increasing number of processes. Away from the scaling limit, the ideal speedup is only minor (a few percent), which will be dominated by organizational overheads in practice. At the scaling limit, the ideal speedup approaches Q . In summary, in the case that the IRK algorithm could be reformulated such that stages can be solved independently, this additional level of parallelism might allow one to increase the granularity of the subproblems to solve and better utilize the capacity of the given hardware resources, such as to reach lower times to solution. In this publication, we show that such a reformulation is possible and, for a simple benchmark, we can achieve significant speedup at the scaling limit in this way.

Stage-parallel approaches for IRK methods have been investigated in the literature but rarely implemented (see section 3). In particular, a critical discussion on the benefits of stage-parallel IRK regarding performance and on the challenges regarding efficient implementation in the context of more sophisticated global preconditioners and optimal preconditioners for the blocks, such as multigrid, is lacking.¹ In order to address this issue, we consider a direct factorization of the linear system arising from the IRK method and extend a novel preconditioner for IRK introduced in [7]. Our results are based on benchmark programs leveraging the infrastructure of the open-source FEM library `deal.II` [3, 4] and are available on GitHub at <https://github.com/peterrum/dealii-spirk>.

As a critical remark, we note that running IRK and, generally, time-stepping schemes at the scaling limit means that the user has enough hardware resources and computational budget. However, most scientists are not in such a “luxurious” position. Hence, using stage-parallel Runge–Kutta methods and potentially parallel-in-time algorithms may not be beneficial or might even be disadvantageous if applied far from the scaling limit due to some—unavoidable—organizational overhead.

The remainder of this contribution is organized as follows. Sections 2 and 3 provide a short description of the linear systems arising from the IRK method and of stage-parallel solution procedures, followed by a discussion of their building blocks and related work. Section 4 presents implementation details of the building blocks, and section 5 discusses relevant performance models. Sections 6 and 7 demonstrate performance results for the solution of the heat equation discretized by low- and high-order FEM with different stage-parallel solution procedures and compare the results to those of non-stage-parallel versions of the solvers. Finally, sections 8 and 9 summarize our findings and point to further research directions.

2. Fully implicit Runge–Kutta methods and stage-parallel solution approaches. In the following, we summarize key aspects of the IRK methods and

¹We define “optimal” as the solver with high node-level performance whose number of iterations is independent of the number of DoFs and of the number of processes.

stage-parallel solution procedures analyzed in this publication. We consider the subclass of IRK methods referred to as Radau IIA methods. For Q stages, the method order is given by $2Q - 1$.

We restrict ourselves to the case of a linear system of equations of the form

$$M \frac{\partial \mathbf{u}(t)}{\partial t} + K \mathbf{u}(t) = \mathbf{g}(t),$$

where M denotes the mass matrix and K the stiffness matrix. A fully implicit Runge–Kutta method computes the solution at the next time level as

$$\mathbf{u}_{m+1} = \mathbf{u}_m + \tau \sum_{q=1}^Q b_q \mathbf{k}_q.$$

The values of \mathbf{k}_q at the stages $q = 1, \dots, Q$ are found by the usual Runge–Kutta defining equations. In the following, we restrict ourselves to the first time step, $m = 0$. Using the linearity of the term $K \mathbf{u}$ and doing basic algebraic manipulations, the stage values are found by solving the linear system

$$\begin{bmatrix} M + \tau a_{11}K & \tau a_{12}K & \dots & \tau a_{1Q}K \\ \tau a_{21}K & M + \tau a_{22}K & \dots & \tau a_{2Q}K \\ \vdots & \vdots & \ddots & \vdots \\ \tau a_{Q1}K & \tau a_{Q2}K & \dots & M + \tau a_{QQ}K \end{bmatrix} \begin{bmatrix} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \vdots \\ \mathbf{k}_Q \end{bmatrix} = \begin{bmatrix} -K \mathbf{u}_0 + \mathbf{g}(t_0 + c_1\tau) \\ -K \mathbf{u}_0 + \mathbf{g}(t_0 + c_2\tau) \\ \vdots \\ -K \mathbf{u}_0 + \mathbf{g}(t_0 + c_Q\tau) \end{bmatrix},$$

which is expressed using Kronecker products as

$$(\mathbb{I}_Q \otimes M + \tau A_Q \otimes K) \mathbf{k} = \bar{\mathbf{g}} - (\mathbb{I}_Q \otimes K)(\mathbf{e}_Q \otimes \mathbf{u}_0).$$

Next, we multiply both sides by $(A_Q^{-1} \otimes \mathbb{I}_n)$ and use the relation $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$ to obtain

$$(2.1) \quad \underbrace{(A_Q^{-1} \otimes M + \tau \mathbb{I}_Q \otimes K)}_{\mathcal{A}} \mathbf{k} = (A_Q^{-1} \otimes \mathbb{I}_n) \bar{\mathbf{g}} - (A_Q^{-1} \otimes K)(\mathbf{e}_Q \otimes \mathbf{u}_0),$$

which is the form we utilize in this study. Following Butcher [12], one can construct the spectral decomposition of $A_Q^{-1} = S \Lambda S^{-1}$ and use this to transform the matrix:

$$(2.2) \quad \mathcal{A} = (A_Q^{-1} \otimes M + \tau \mathbb{I}_Q \otimes K) = (S \otimes \mathbb{I}_n)(\Lambda \otimes M + \tau \mathbb{I}_Q \otimes K)(S^{-1} \otimes \mathbb{I}_n).$$

The inverse of the matrix and the solution of the stages are explicitly given as

$$(2.3) \quad \mathbf{k} = \underbrace{(S \otimes \mathbb{I}_n)(\Lambda \otimes M + \tau \mathbb{I}_Q \otimes K)^{-1}(S^{-1} \otimes \mathbb{I}_n)}_{\mathcal{A}^{-1}} ((A_Q^{-1} \otimes \mathbb{I}_n) \bar{\mathbf{g}} - (A_Q^{-1} \otimes K)(\mathbf{e}_Q \otimes \mathbf{u}_0)).$$

In the context of Radau IIA methods, Λ is diagonal and contains $\lfloor Q/2 \rfloor$ complex-conjugate eigenvalue pairs as well as one real eigenvalue in the case of odd Q , and the matrix S contains the eigenvectors. Hence, $(\Lambda \otimes M + \tau \mathbb{I}_Q \otimes K)$ is block-diagonal and its inverse is given by the inverse of each block $(\lambda_q M + \tau K)$, which can be computed independently. In practice, one solves blocks corresponding to complex-conjugate eigenvalue pairs together, necessitating the solution of $\lfloor Q/2 \rfloor$ complex blocks via complex arithmetic or the transformation into two-by-two real blocks [32] and the solution of one real block in the case of odd Q .

In the literature, there are additional ways to factorize A_Q/A_Q^{-1} and to obtain a real block system more directly: The real Schur complement [31, 35] leads to block triangular matrices \hat{S}, \hat{S}^{-1} and block-diagonal matrix $\hat{\Lambda}$ with two-by-two blocks of the form $\begin{bmatrix} \Re(\lambda_q) & \alpha \\ -\Im(\lambda_q)^2/\alpha & \Re(\lambda_q) \end{bmatrix}$ for an arbitrary constant α .

Alternatively to factorizing \mathcal{A} directly, one can also solve the system (2.1) iteratively with a Krylov solver, such as GMRES, with the help of a preconditioner. We note that, for iterative solvers with suitable preconditioners, only the action of the matrix $(A_Q^{-1} \otimes M + \tau \mathbb{I}_Q \otimes K)$ on a vector needs to be implemented, rather than the matrix itself.

Based on an observation by Axelsson [5], namely, that the matrices A_Q and A_Q^{-1} have a dominating lower triangular part, Axelsson and Neytcheva [7] proposed to decompose $A_Q^{-1} = LU$. Here, matrix U has a unit diagonal, implying that all eigenvalues of $L^{-1}A_Q^{-1}$ are equal to one, which makes L suitable for constructing a preconditioner for (2.1). Independently of [7], Masud Rana et al. [27] used a similar idea to derive a block preconditioner based on A_Q in which each block is solved by AMG and the application is done via forward substitution, leading to a sequential execution of the stages. In contrast, Axelsson and Neytcheva [7] proposed to employ the spectral decomposition $L = \tilde{S}\tilde{\Lambda}\tilde{S}^{-1}$ in order to obtain a preconditioner allowing for stage parallelism and real arithmetic. The inverse of the preconditioner is given as

$$(2.4) \quad P^{-1} = (\tilde{S} \otimes \mathbb{I}_n)(\tilde{\Lambda} \otimes M + \tau \mathbb{I}_Q \otimes K)^{-1}(\tilde{S}^{-1} \otimes \mathbb{I}_n).$$

Just as before, the term $(\tilde{\Lambda} \otimes M + \tau \mathbb{I}_Q \otimes K)$ is block-diagonal. The spectral decomposition of L is always real, which is the main motivation to use L instead of A_Q^{-1} . Hence, Q real blocks can be solved independently, in contrast to the complex case with only $\lceil Q/2 \rceil$ independent blocks. An analysis of the eigenvalues of the preconditioned system is provided in [6]. Below, we drop the tilde in $\tilde{\Lambda}$ and \tilde{S} as the meaning of these symbols is clear in the context in which they are used.

In (2.1), (2.3), and (2.4), one can identify multiple independent operations:

1. the right-hand-side function \mathbf{g} can be evaluated independently for each stage,
2. the matrix-vector multiplication with the mass matrix M and the stiffness matrix K in $(\mathbb{I}_q \otimes M)\mathbf{k}$ and $(\mathbb{I}_Q \otimes K)\mathbf{k}$ can be performed independently for each stage,² and
3. Q or $\lceil Q/2 \rceil$ blocks involving $(\lambda_q M + \tau K)$ can be solved independently.

A parallel execution across the blocks is a natural choice on modern supercomputers.

Obviously, the combination of the partial results from the stage-parallel execution via multiplication by $(A_Q^{-1} \otimes \mathbb{I}_n)$, $(S^{-1} \otimes \mathbb{I}_n)$, or $(S \otimes \mathbb{I}_n)$ is not independent. This step corresponds to a linear combination of the stage vectors, representing a basis change in the latter two cases. Such an operation might be challenging in parallel, especially on distributed memory systems if each stage is assigned to a distinct process, and the parallel communication might counteract the benefits of the parallel execution of other parts of the algorithm.

In the following discussions of parallelization and implementation aspects, we omit the option of directly factorizing \mathcal{A} and consider it only in section 7, pointing out that the proposed concepts are applicable in that context as well. For the realization of a stage-parallel iterative solver including the stage-parallel preconditioner, one needs an efficient parallel implementation of basic tensor operations from (2.1) and (2.4):

²Note that the following decomposition is applicable: $A_Q^{-1} \otimes M = (A_Q^{-1} \otimes \mathbb{I}_n)(\mathbb{I}_Q \otimes M)$.

- “generalized vector scaling” with $C_q \in \mathbb{R}^{n \times n}$,

$$\mathbf{v} = \text{diag}(C_1, \dots, C_Q)\mathbf{u} \leftrightarrow \mathbf{v}_q = C_q\mathbf{u}_q,$$

which simplifies with $C_q = C \in \mathbb{R}^{n \times n}$ to

$$(2.5) \quad \mathbf{v} = (\mathbb{I}_Q \otimes C)\mathbf{u} \leftrightarrow \mathbf{v}_q = C\mathbf{u}_q;$$

- “generalized matrix-vector product” with $D \in \mathbb{R}^{Q \times Q}$,

$$(2.6) \quad \mathbf{v} = (D \otimes \mathbb{I}_n)\mathbf{u} \leftrightarrow \mathbf{v}_i = \sum_{1 \leq j \leq Q} D_{ij}\mathbf{u}_j.$$

Furthermore, we analyze the benefits of the stage-parallel solver in comparison to its sequential counterpart. For the blocks, we use GMG [28] as an efficient solver with state-of-the-art parallel scaling.

We call the algorithm (2.1) with preconditioner (2.4) *stage-parallel IRK* (in the figures abbreviated as *SPIRK*) when the blocks are solved in parallel. When the blocks are solved sequentially, we simply use *IRK*.

3. Related work. One of the few examples for a work considering stage-parallel implementations of IRK in the currently available literature is that of Pazner and Persson [30], who study time-dependent nonlinear problems and develop a stage-parallel IRK preconditioner using a block-Jacobi solver across the processes around a local ILU. They analyze different block sizes: one that takes into account all coupling terms between stages (“stage-coupled”) and one that ignores them (“stage-uncoupled”). In the latter case, the blocks corresponding to the stages are independent of each other and are solved in a stage-parallel way. The stages are solved by subgroups of processes, leading to larger individual spatial blocks and, hence, to better efficiency of ILU in terms of number of iterations. The structure of the stage-parallel solver is very similar to the inner term of (2.4). However, we note that the basis changes \tilde{S} and \tilde{S}^{-1} imply a coupling of the stages. In section 6, we show that the cost of the basis changes is small, enabling a stage-parallel “stage-coupled” preconditioner with a similar cost as for the stage-parallel “stage-uncoupled” preconditioner proposed in [30].

Although not stated explicitly, the block-Jacobi preconditioners from [26, 33] could be trivially parallelized over stages. Despite their simplicity, they are not efficient preconditioners and not robust with an increasing number of stages, as recently shown in [27]. In [33], the authors also study block Gauss–Seidel preconditioners, which make the solution inherently serial due to forward and backward substitutions. As investigated in [27], the approach built around L from the LU decomposition of the Butcher tableau, which can be implemented in a stage-parallel way [7], can outperform the Gauss–Seidel approach in regard to the number of iterations.

We conclude this section with comments on diagonally implicit Runge–Kutta methods (DIRK), which involve a lower-triangular stage matrix A_Q . Similarly to the Gauss–Seidel case, a naive realization leads to a sequential dependence between the solved stages, albeit without the need for an outer solver. DIRK methods generally have worse accuracy and stability properties than fully implicit (Radau IIA) methods for a given number of stages. In order to overcome the sequential solution nature of DIRK, one could consider performing a spectral decomposition of its Runge–Kutta matrix A_Q . For diagonalizable A_Q , this would lead to a solution procedure as in (2.3), which would allow for stage parallelism. In the case that all eigenvalues are real and unique, each stage could be solved independently, just like in (2.4), however,

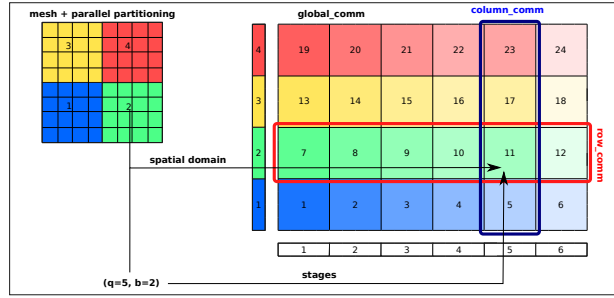


FIG. 2. Three MPI communicators (for a hypothetical setup with 24 processes, $Q = 6$ stages, and $B = 4$ partitions) used to simplify communications in stage-parallel IRK: `global_comm` lists all involved processes, whereas `column_comm` and `row_comm` collect processes owning the same stage and partition of the computational domain, respectively. Furthermore, the mapping between the stage/partition pair and the rank in the global communicator is indicated. Adopted from [29].

without outer iterations. We do not consider DIRK methods in this work, since the implementation of stage-parallel DIRK would be similar, with possible trade-offs in terms of different iteration counts in the iterative solvers or the time-stepping properties, respectively.

4. Implementation details. In the following, we discuss pure MPI implementations of the stage-parallel IRK; in some of our experiments, we use MPI’s shared-memory features. The algorithms can be easily generalized to task-based implementations, as provided by OpenMP, and to hybrid implementations (MPI + X). In subsections 4.1–4.3, we describe an approach where stages are distributed and assigned to distinct processes, and, in subsection 4.4, we describe a way to batch operations from different stages. Both approaches aim to increase the parallelism in the solver and, thereby, to increase the sizes of the subproblems, in order to better utilize the resources of a massively parallel computer.

4.1. Domain decomposition. We decompose the mesh of our spatial computational domain into B partitions. In the case of IRK, we assign each partition to a (MPI) process. In the case of stage-parallel IRK, we assign each process a pair $(q, b) \in [1, Q] \times [1, B]$ consisting of a stage and a spatial partition. For the sake of simplicity, we enumerate the processes lexicographically, as shown in Figure 2. However, a basic preprocessing step based on virtual topologies allows us to use any enumeration of processes, as is needed in later discussions.

In order to perform operations between processes with the same stage (e.g., to solve the inner blocks) or the same partition (e.g., for the basis change), we use additional subcommunicators `column_comm` and `row_comm`. This is a common approach in the context of distributed matrix-matrix-multiplication implementations [34] and finite-difference stencil computations on Cartesian meshes [17]. A similar approach has been used in [29] to solve the six-dimensional (6D) Vlasov–Poisson equation on the tensor product of a 3D geometric domain and a 3D velocity-space domain.

4.2. Parallel data distribution. A natural choice is to let each process (q, b) own the locally relevant part of the vectors associated with the stage q and to let it update the locally owned part of the solution vector during evaluation of the right-hand-side function, of the matrix-vector product, and of the block solvers.

Downloaded 07/19/23 to 129.187.254.46 . Redistribution subject to SIAM license or copyright; see https://pubs.siam.org/terms-privacy

For the operation $\mathbf{v} = (\mathbb{I}_Q \otimes C)\mathbf{u}$, setting $\mathbf{v}_q = C\mathbf{u}_q$ does not require communication between stages. However, each process needs access to the local part of the matrix $C \in \mathbb{R}^{n \times n}$ (in our case M and K). The need for local access to the matrix means that certain data structures have to be duplicated Q times on distributed systems. This could be addressed by using shared memory. Since all our experiments are run without assembling any matrix (“matrix-free approach”), which is memory-efficient [22], we defer the investigation of shared memory to future work.

In contrast, the operation $\mathbf{v} = (D \otimes \mathbb{I}_n)\mathbf{u}$ only needs access to a small matrix $D \in \mathbb{R}^{Q \times Q}$, which can easily be replicated on all processes. The major challenge for this operation is that a process needs to access the local vector entries of all stages, e.g., $\mathbf{v}_2 = D_{11}\mathbf{u}_1 + D_{21}\mathbf{u}_2 + \dots + D_{2Q}\mathbf{u}_Q$. The gathering of the needed vector entries from all stages is not feasible as this would mean that the vectors are duplicated Q times. In the next sections, we discuss an appropriate communication pattern to alleviate this problem and compare its performance to the one of a shared-memory approach, in which the processes have direct read access to the needed entries of all stages.

4.3. Distributed tensor operations. In order to derive a memory-efficient implementation of the operation $\mathbf{v} = (D \otimes \mathbb{I}_n)\mathbf{u}$, which is needed for the linear combinations during the setup of the right-hand-side vector and during the matrix-vector multiplication as well as for the basis changes, one can exploit associativity:

$$(4.1) \quad \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_Q \end{bmatrix} = \underbrace{\begin{bmatrix} D_{11}\mathbf{u}_1 \\ D_{21}\mathbf{u}_1 \\ \vdots \\ D_{Q1}\mathbf{u}_1 \end{bmatrix} + \begin{bmatrix} D_{12}\mathbf{u}_2 \\ D_{22}\mathbf{u}_2 \\ \vdots \\ D_{Q2}\mathbf{u}_2 \end{bmatrix} + \dots + \begin{bmatrix} D_{1Q}\mathbf{u}_Q \\ D_{2Q}\mathbf{u}_Q \\ \vdots \\ D_{QQ}\mathbf{u}_Q \end{bmatrix}}_{\mathbf{v}_i = \sum_j D_{ij}\mathbf{u}_j} = \underbrace{\begin{bmatrix} D_{11}\mathbf{u}_1 \\ D_{22}\mathbf{u}_2 \\ \vdots \\ D_{QQ}\mathbf{u}_Q \end{bmatrix} + \begin{bmatrix} D_{12}\mathbf{u}_2 \\ D_{23}\mathbf{u}_3 \\ \vdots \\ D_{Q1}\mathbf{u}_1 \end{bmatrix} + \dots + \begin{bmatrix} D_{1Q}\mathbf{u}_Q \\ D_{21}\mathbf{u}_1 \\ \vdots \\ D_{Q(Q-1)}\mathbf{u}_{(Q-1)} \end{bmatrix}}_{\mathbf{v}_i = \sum_k D_{ij}\mathbf{u}_j \text{ with } j=(i+k)\%Q}.$$

If we consider each summand as a computation step, we realize that each process needs access to a different part of the source vector \mathbf{u} associated with a stage. The final communication pattern is hence as follows. After each computation step, the matrix is circularly shifted to the left and the source vector upward. The algorithm is similar to Cannon’s algorithm [13], which is designed for distributed matrix-matrix multiplications. The crucial difference is that the present work operates on tensors and with fully replicated D . In the case that \mathbf{u} is distributed, we obtain a circular communication pattern, which can be built around a sequence of calls to `MPI_Sendrecv_replace`, operating on `comm_row`.

4.4. Batching of operations. Subsections 4.1–4.3 consider an approach that also employs parallelism across the stages by assigning each stage to a distinct compute unit. For fixed computational resources and problem sizes, this increases the size of the spatial subproblems and reduces the number of communication steps. Alternatively, one could increase the local work by processing Q stages on the same compute unit in a batched fashion. To be efficient, all operations of the form $\mathbf{v} = (\mathbb{I}_Q \otimes C)\mathbf{u}$ in (2.1) and of the solvers of the steps in (2.4) need to support this matrix-times-multivector processing mode.

For the GMG solver used for the experiments in sections 6 and 7, all ingredients in terms of smoother, prolongator/restrictor, and coarse-grid solver need to support batching. For simple smoothers and coarse-grid solvers, like Chebyshev iterations around a point-Jacobi method, this is the case, whereas it is typically not the case for algebraic multigrid.

Batching is more efficient in terms of memory consumption, since shared data structures, particularly the matrices M and K , only need to be stored once. Fur-

thermore, also data structures pertaining to matrix-free evaluation need to be loaded via a shared resource, the bus from main memory, only once per iteration, having an advantage in terms of memory transfer in the case in which all stages are processed during a single cell loop where data from caches are still hot. For instance, many matrix-free implementations load metric terms, e.g., the Jacobian matrix of size $\mathbb{R}^{3 \times 3}$ in three dimensions, for each quadrature point during cell loops. Even though compression schemes are sometimes used for simple meshes, e.g., with affine shapes [22, 23], or metric terms are computed on the fly to reduce memory transfer [24], matrix-free operator evaluation benefits from batching in the sense of more beneficial data access to compute ratios.

As a final remark, we point out that batching resembles the monolithic solution procedure of [15], in which the coupling terms are not skipped during smoothing and the coarse-grid solver. However, the development of appropriate efficient smoothers is not trivial, possibly involving vertex-star patches [1] with no obvious fast (matrix-free) inversion in the general case. Apart from this sequential challenge, the performance of the monolithic solution procedure can be expected to be comparable.

5. Performance modeling. In this section, we derive performance models for sequential and stage-parallel IRK. Since the application of the preconditioner P^{-1} is the most expensive ingredient in both cases (see the results in section 6), we consider it in detail. The statements made for the preconditioner can be straightforwardly transferred to other parts of the algorithm.

Figure 3(a) shows a possible trace of a serial execution of IRK, and Figure 3(b) presents a parallel execution of IRK (with three processes). In both cases, the basis change S^{-1} , the inner block solvers B_q , and the basis change S are executed in sequence so that one gets for the total runtime of the application of the preconditioner,

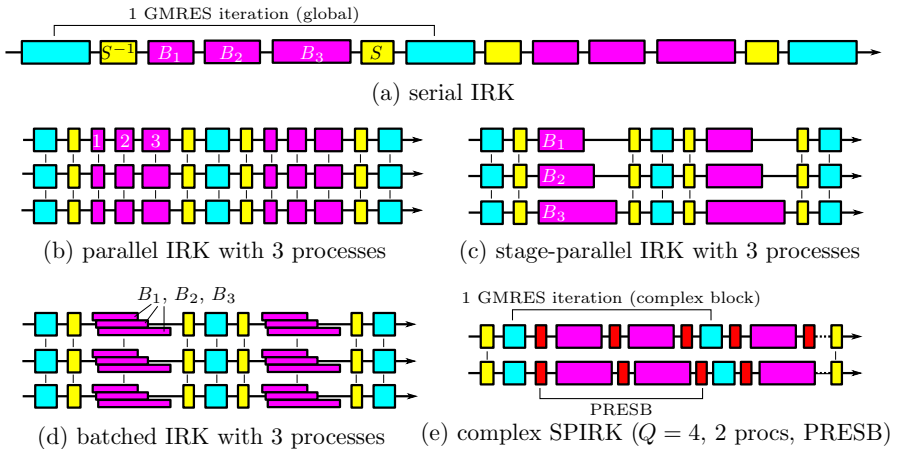


FIG. 3. (a)–(d) Visualization of a serial, a parallel, a stage-parallel, and a batched execution of the application of preconditioner P^{-1} of IRK with $Q = 3$. The main components are the GMRES solver, the basis changes (S, S^{-1}), and the block solvers (B_q). (e) Visualization of a stage-parallel execution of the complex IRK with $Q = 4$ and PRESB [8]. Vertical lines indicate communication between processes.

$$(5.1) \quad T_{\text{IRK}}(N) = 2 \cdot T_S(N) + \sum_{1 \leq q \leq Q} T_{B_q}(N),$$

where $T_{\text{IRK}}(N)$ denotes the total time for one application of the preconditioner. Clearly, it is a function of the number of processes N . In the ideal case, $T_{\square}(N) \approx T_{\square}(1)/N$. However, inherently serial parts in the code prevent perfect scaling due to Amdahl's law.

In the stage-parallel case, the steps of the inner block solver are performed in parallel. Since each block may have different properties, the solution processes might differ in their runtime and require different numbers of inner iterations in the case that iterative solvers are used for the blocks. Recall that the block systems are of the form $(\lambda_q M + \tau K)$. The number of stages affects the range of values of λ_q , the discretization affects the entries of M , and the time step τ scales the stiffness matrix K .

The solution of each block is combined during the application of S , leading to an unavoidable synchronization. This results in a trace, as indicated in Figure 3(c), and in a total runtime that is determined by the maximum runtime of any of the block solvers:

$$(5.2) \quad T_{\text{SPIRK}}(N) = 2 \cdot T'_S(N) + \max_{1 \leq q \leq Q} (T_{B_q}(N/Q)).$$

Equation (5.2) also describes the runtime in the case of the batched approach; see Figure 3(d).

Based on (5.1) and (5.2), one can expect the following parallel performance behavior. The timings are comparable if (1) T_S and T'_S are comparable, (2) the block solvers are scaling nearly ideally, i.e., $T_{B_q}(N) \approx T_{B_q}(1)/N$, and (3) the solution times of the block solvers are comparable, i.e., $T_{B_q} \approx T_{B_1}$ for all $q \in \{1, \dots, Q\}$. The parallel performance of the sequential IRK deteriorates if $T_{\square}(N) \gg T_{\square}(1)/N$, which is the case at the scaling limit. The performance of the stage-parallel IRK method deteriorates if there is a significant difference in the solution times of the block solvers, limiting the maximum speedup to $\sum_q T_{B_q} / \max(T_{B_q}) \leq Q$. One can deduce that stage-parallel IRK has advantages only if IRK is at the scaling limit and the solution times of the blocks are comparable.

The discussion above assumes wall-clock times of the solution of a block. We can refine the expressions for iterative solvers:

$$T_{\text{IRK}}(N) = \sum_{1 \leq q \leq Q} N_q^{\text{IT}} \cdot \hat{T}_{B_q}(N) \quad \text{and} \quad T_{\text{SPIRK}}(N) = \max_{1 \leq q \leq Q} (N_q^{\text{IT}} \cdot \hat{T}_{B_q}(N/Q))$$

with \hat{T}_{B_q} being the time of one iteration and N_q^{IT} the number of iterations. For the sake of simplicity, T_S and T'_S are dropped. If we assume that we are at the scaling limit ($\lim_{N \rightarrow \infty} \hat{T}_q(N) \approx \lim_{N \rightarrow \infty} \hat{T}_q(N/Q)$), we get the expressions

$$\lim_{N \rightarrow \infty} T_{\text{IRK}}(N) \sim \sum_{1 \leq q \leq Q} N_q^{\text{IT}} \quad \text{and} \quad \lim_{N \rightarrow \infty} T_{\text{SPIRK}}(N) \sim \max_{1 \leq q \leq Q} (N_q^{\text{IT}}),$$

indicating that it is possible to estimate bounds of maximum speedups based on the number of inner block iterations that can be run in parallel. This estimate gives a simple means to compare the benefits to alternative (stage-parallel) implementations, like the direct factorization (2.3), where one could consider all block solves accumulated over all GMRES iterations for one time step.

In the numerical experiments in section 6 and 7, we evaluate the statements made above.

6. Numerical experiments. In this section, we present performance results of the stage-parallel implementation of (2.1) with preconditioner (2.4). We start with results obtained on 16 compute nodes. In particular, we discuss the performance of a base configuration and the influence of key parameters. We conclude the section with a strong-scaling analysis.

We consider the 3D heat equation $\partial u/\partial t = \Delta u + f$ with the manufactured solution

$$u(x, y, z, t) = \sin(2\pi x) \sin(2\pi y) \sin(2\pi z)(1 + \sin(\pi t)) \exp(-0.5t)$$

on a cube $\Omega = [0, 1]^3$. The source-term function f and the Dirichlet boundary conditions are selected appropriately. The spatial variables are discretized with the finite element method, for which we use the open-source library `deal.II` [3, 4]. The mesh is obtained by L steps of isotropic refinement of a coarse mesh consisting of a single hexahedral cell, giving 2^L mesh cells per spatial direction or 2^{3L} cells in total. We use continuous Lagrange finite elements, defined as the tensor products of 1D finite elements with degree k . For quadrature, we consider the consistent Gauss–Legendre quadrature rule with $(k + 1)^3$ points. Table 1 shows the number of cells and the number of degrees of freedom (DoFs) for $k = 1$ and $k = 4$ for $4 \leq L \leq 11$. The time step τ is set to 0.1, and we run 10 time steps.

As outer solver for (2.1), we apply GMRES. It is run until the l_2 -norm of the residual has been reduced by 10^{12} . As the approximate inverse of each block of (2.4), we use a single V-cycle of the GMG algorithm from `deal.II` [28]. As a smoother, we apply Chebyshev iterations around a point-Jacobi method [2] with degree 5 and, as a coarse-grid solver, we use the algebraic multigrid solver from ML [16]. The corresponding solver diagram is shown in Figure 4. All operator evaluations are performed using the matrix-free infrastructure described in [22, 23] to ensure a high node-level performance, following the current trends of exascale finite-element algorithms described in [21]. Hence, we embed the IRK methods in a—with regard to communication costs—challenging context where differences are most pronounced.

TABLE 1
Number of cells and of DoFs for different numbers of refinements L .

| L | Cells | Degrees of freedom | | L | Cells | Degrees of freedom | |
|-----|---------|--------------------|---------|-----|---------|--------------------|---------|
| | | $k = 1$ | $k = 4$ | | | $k = 1$ | $k = 4$ |
| 4 | 4.1E+03 | 4.9E+03 | 2.7E+05 | 8 | 1.7E+07 | 1.7E+07 | 1.1E+09 |
| 5 | 3.3E+04 | 3.6E+04 | 2.1E+06 | 9 | 1.3E+08 | 1.4E+08 | 8.6E+09 |
| 6 | 2.6E+05 | 2.7E+05 | 1.7E+07 | 10 | 1.1E+09 | 1.1E+09 | 6.9E+10 |
| 7 | 2.1E+06 | 2.1E+06 | 1.4E+08 | 11 | 8.6E+09 | 8.6E+09 | 5.5E+11 |

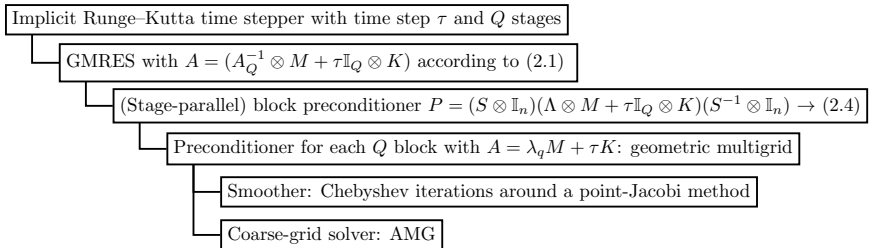


FIG. 4. Diagram of the solver used to solve the heat problem in section 6.

Downloaded 07/19/23 to 129.187.254.46 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

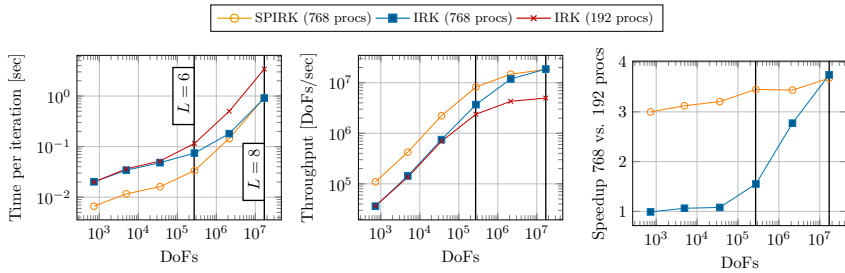


FIG. 5. Comparison of stage-parallel IRK with 768 processes, of IRK with 768 processes, and of IRK with 192 ($=768/4$) processes for $Q=4$ and $k=1$: time and throughput per time step as well as speedup.

All experiments are conducted on the SuperMUC-NG supercomputer. Its compute nodes have 2 sockets (each with 24 cores of Intel Xeon Skylake) and the AVX-512 ISA extension so that 8 doubles can be processed per instruction.³ As compiler, we use g++ (version 9.1.0) with the flags `-O3`, `-funroll-loops`, `-march=skylake-avx512`.

6.1. Moderately parallel runs. We start with the analysis of the performance of the proposed stage-parallel algorithm at small scales with 786 processes (16 compute nodes) as an example. Figure 5 shows the runtime per time step and the throughput for stage-parallel IRK and IRK for $k=1$ and $Q=4$. Furthermore, we present data for IRK with 192 ($=786/4$) processes, to which we compare the obtained speedup. For large problem sizes, IRK with four times the number of processes achieves a speedup of 3.7. However, its performance quickly drops for smaller sizes, and times comparable to the 192-process case are reached. In this range, the number of processes used does not influence the times, and inherently serial parts of the code (like latency of communication and the coarse-grid solver) dominate. In the case of stage-parallel IRK, we see a different behavior: over large refinement ranges, a speedup of > 3 can be reached. The maximum value is lower than in the IRK case ($3.6 < 3.7$).

Figure 6 presents—in accordance with the traces in Figure 3—pie diagrams visualizing the time spent on different parts of the algorithms. In particular, they show the time for basis change S/S^{-1} and for the solution of each block or of the whole preconditioner in the case of IRK or of stage-parallel IRK, correspondingly. The pie diagrams are provided for two refinement configurations: $L=8$ (far away from the scaling limit) and $L=6$ (close to the scaling limit). Starting with $L=8$, one can observe that, in the case of IRK, the block preconditioners are dominating in the total time (75%) and the basis changes are negligible (1%). In the case of stage-parallel IRK, one can see that the time spent on the (single) preconditioner application has decreased, but the times for setting up the right-hand-side vector, the matrix-vector product, and the basis changes during preconditioning have slightly increased. This is not surprising, since these are operations where communication between stages is required and processes are implicitly synchronized by the rotation of the vectors. For $L=6$, the ratio of preconditioning becomes an even more dominating part of the total solution time (90%/77%). However, one can see that the total time spent for preconditioning reduces by a factor of 3.1 in the case of stage-parallel IRK compared to IRK.

³<https://doku.lrz.de/display/PUBLIC/SuperMUC-NG>, retrieved on February 26, 2022.

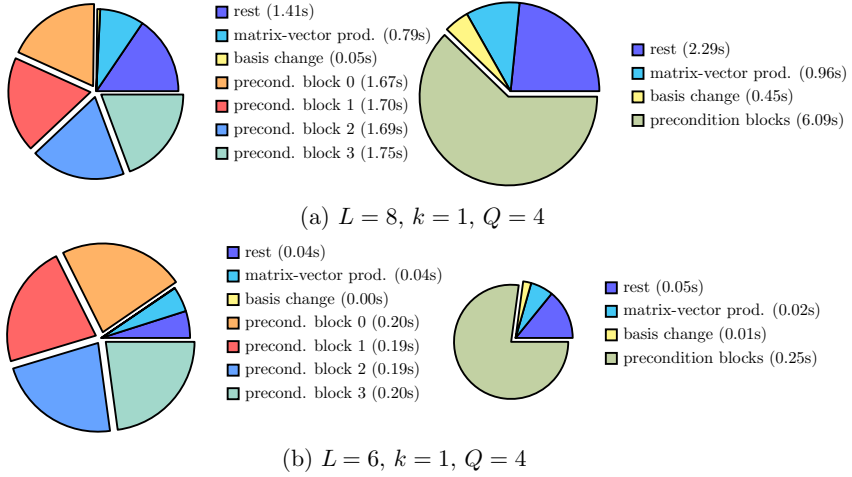


FIG. 6. Time spent for matrix-vector product, basis change, block solvers, and the remaining operations (setup of right-hand-side vector of (2.1), vector updates, etc.) for IRK (left) and stage-parallel IRK (right). The area of the circles indicates the total time compared to the other version.

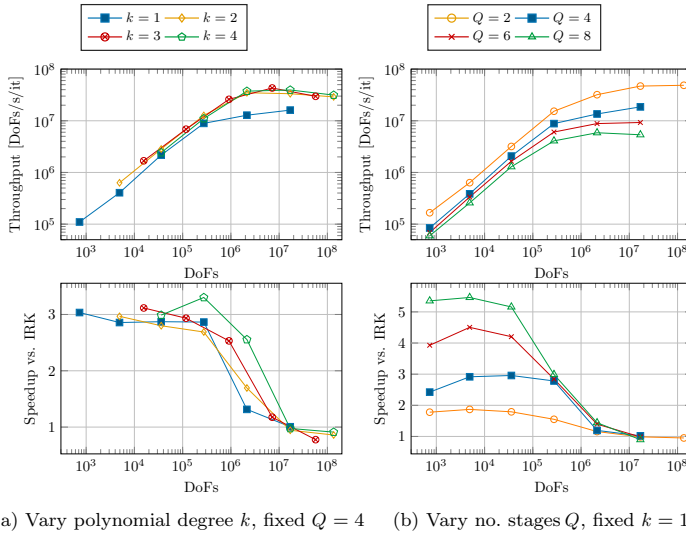


FIG. 7. Influence of parameters on the performance of stage-parallel IRK and the speedup of stage-parallel IRK as compared to IRK on 16 nodes/768 MPI processes.

Influence of key parameters. Figure 7 shows the influence of the parameters “polynomial degree k ” and “number of stages Q ” on the performance and speedup of stage-parallel IRK compared to IRK with the same number of processes. The following observations can be made.

Higher polynomial degrees $k \geq 2$ increase the throughput of matrix-free operator evaluation. As a consequence, the throughput of IRK also rises with increasing

TABLE 2

Comparison of the number of GMRES iterations ($\#G$) and of accumulated V -cycle applications ($\#V$) per time step for noncomplex/complex IRK and stage-parallel IRK. For the stage-parallel IRK, the reported numbers of iterations and cycles are shown for process groups. Since complex stage-parallel IRK runs GMRES independently on the blocks, the number of GMRES iterations and, consequently, the number of V -cycle applications vary between blocks. In this case, we report the minimum and maximum values. Numbers are shown for different Q and $k = 1$, $L = 8$. The additional superscript of $\#V$ specifies the number of blocks a GMG V -cycle considers.

| Noncomplex (sections 2–6) | | | | |
|---------------------------|-------------------|--|-------------------|------------------------------|
| | IRK | | SPIRK | |
| Q | $\#G$ ($\#V^1$) | | $\#G$ ($\#V^1$) | batched $\#G$ ($\#V^Q$) |
| 2 | 5.0 (12.0) | | 5.0 (6.0) | 5.0 (6.0) |
| 4 | 7.0 (32.0) | | 7.0 (8.0) | 7.8 (8.8) |
| 6 | 9.9 (65.3) | | 9.9 (10.9) | 9.9 (10.9) |
| 8 | 11.9 (103.1) | | 11.9 (12.9) | 11.0 (12.0) |

| Complex (section 7) | | | | |
|---------------------|-------------------|-------------------|---------------------|--------------------|
| | IRK-PRESB | IRK-GMG | SPIRK-PRESB | SPIRK-GMG |
| Q | $\#G$ ($\#V^1$) | $\#G$ ($\#V^2$) | $\#G$ ($\#V^1$) | $\#G$ ($\#V^2$) |
| 2 | 6.0 (14.0) | 7.0 (8.0) | 6.0 (14.0) | 7.0 (8.0) |
| 4 | 11.6 (27.1) | 14.0 (16.0) | 5.6–6.0 (13.1–14.0) | 7.0–7.0 (8.0–8.0) |
| 6 | 18.2 (42.4) | 22.0 (25.0) | 6.0–6.2 (14.0–14.4) | 7.0–8.0 (8.0–9.0) |
| 8 | 23.1 (54.2) | 31.3 (35.3) | 5.0–6.1 (12.0–14.2) | 7.0–9.3 (8.0–10.3) |

polynomial degree, since its most time-consuming component, the geometric-multigrid solver relying on a sequence of matrix-vector products, is also implemented in a matrix-free way. Furthermore, increasing k seems to also have a positive effect on the speedup of stage-parallel IRK.

The number of stages Q has the most significant influence on the throughput of IRK and stage-parallel IRK. This is not surprising, since the work per time step increases with the number of stages. However, the observed drop in throughput is more significant than expected from the increase in work alone. This is explained by the number of outer iterations, which increases slightly with the number of stages. The achievable speedup with stage-parallel IRK rises as the number of stages gets higher. In addition, Table 2 shows the average number of GRMES iterations and accumulated numbers of V -cycles that are run in parallel. One can see, on the one hand, that the number of GMRES iterations increases from 5 to 12 when going from $Q = 2$ to $Q = 8$ and, on the other hand, that stage-parallel execution allows one to reduce, e.g., for $Q = 8$, the number of V -cycles run sequentially by a factor of 8 from 103 to 12, matching the measured timings and speedups in Figure 7.

Making definite general conclusions is not straightforward as they depend on the number of processes, the type of the coarse mesh, the refinement, the partial differential equation, and the block solvers. However, we believe that similar trends can be observed in different setups. We would like to note that we use the heat equation as a test problem, for which efficient and well-scaling block preconditioners based on multigrid methods are available. For other classes of problems where optimally scaling block solvers are more challenging to design, we would expect to observe differences between stage-parallel IRK and IRK also for smaller problem sizes. In the worst case, when the blocks are solved by direct methods, stage parallelism might be the only way to parallelize the work. However, there is evidence for advantages of more advanced spatial solvers that do not have optimal parallel scalability, such as the block-Jacobi solver with local ILU as considered by [30].



FIG. 8. Different virtual topologies to increase data locality (a) for the basis change S and (c) for the inner preconditioner. Version (b) introduces padding to guarantee that all stages are on the same shared-memory domain/same compute node.

6.2. Virtual topology and shared memory. The row-major lexicographical enumeration of processes of subsection 4.1 favors the operation $\mathbf{v} = (D \otimes \mathbb{I}_n)\mathbf{u}$, since the data needed are in close proximity and maybe even on the same compute node. However, it is not optimal for the inner (multigrid) solver, for which a column-major enumeration (Figure 8(c)), placing nearby spatial partitions on the same node, would be favorable. In the following, we compare these two virtual topologies.

Furthermore, we investigate the benefits of using shared-memory features of MPI during $\mathbf{v} = (D \otimes \mathbb{I}_n)\mathbf{u}$. For this, a modified row-major enumeration is beneficial. We introduce a padding—as indicated in Figure 8(b)—in such a way that all processes of a stage are assigned to the same compute node similarly as done in [30]. This allows us to skip the communication of complete vectors as their entries can be accessed directly. In order to prevent race conditions, we introduce barriers across the processes in `comm_row`, which are equivalent to the implicit barriers in the case of parallel for loops in `OpenMP`.

Figure 9 shows the experimental results for $k = 1$ and $Q = 4$ on 1, 16, and 64 compute nodes. Generally, the different virtual topologies show similar behaviors. Using shared memory leads to a speedup in a few cases, but, overall, the improvement is not significant in terms of runtimes. This is because the basis change is not the bottleneck, as discussed in the previous sections. Overall, the column-major enumeration seems to give the best results in the intermediate regime. However, it also turns out to be the slowest virtual topology far from the scaling limit, indicating that the increased costs of the basis change cannot be counterbalanced by the faster block solver.

6.3. Large-scale parallel runs. Figure 10 shows the results of scaling experiments starting with 1 compute node (48 processes) up to 3,072 nodes (147,456 processes) for different polynomial degrees ($k = 1/k = 4$) and numbers of stages ($Q = 2/Q = 4/Q = 9$). One can clearly see that stage-parallel IRK reaches lower times to solution at the scaling limit. Figures 11 and 12 give more insights, by providing normalized plots (throughput of one time step per stage) of the same results of all considered values of Q in a single diagram. Far from the scaling limit (right top corner of the plots), IRK tends to be more efficient. Stage-parallel IRK, on the contrary, reaches lower times per time step at the scaling limit (left bottom corner of the plots) at the cost of lower efficiencies. Furthermore, the diagrams allow one to compare the effect of the value of Q , as is similarly done in subsection 6.1 for a moderate number of processes. Here, one can see again that the costs are increasing with the number of stages, particularly also because of the increasing number of outer iterations, which

Downloaded 07/19/23 to 129.187.254.46 . Redistribution subject to SIAM license or copyright; see https://pubs.siam.org/terms-privacy

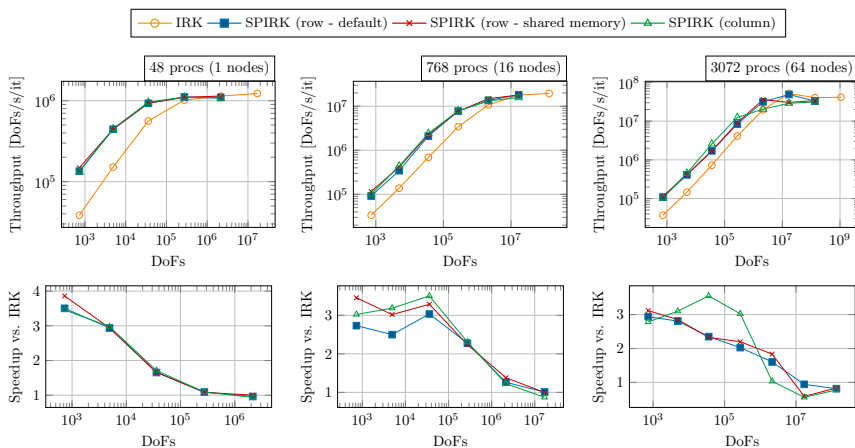


FIG. 9. Comparison of virtual topologies for $k = 1$ and $Q = 4$.

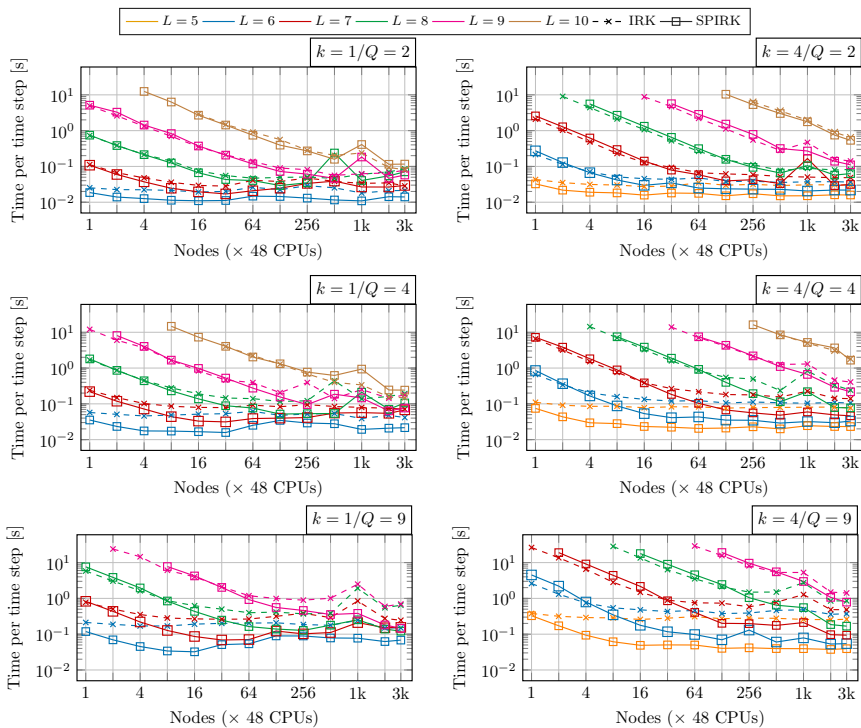


FIG. 10. Strong-scaling comparison (time per time step) of IRK and stage-parallel IRK for $k = 1/k = 4$ and $Q = 2/Q = 4/Q = 9$.

Downloaded 07/19/23 to 129.187.254.46 . Redistribution subject to SIAM license or copyright; see <https://epubs.siam.org/terms-privacy>

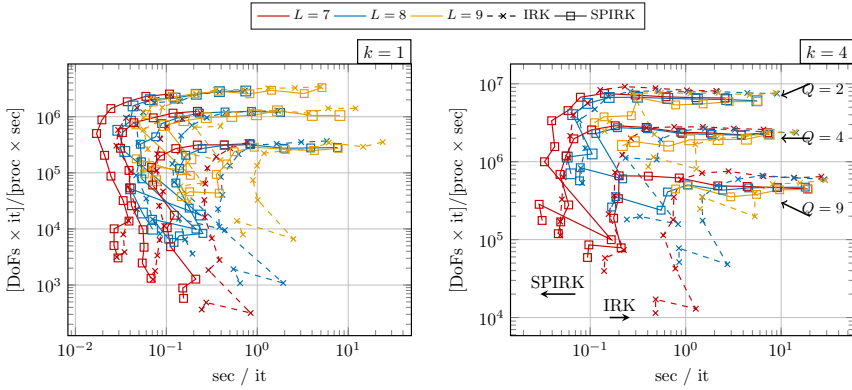


FIG. 11. Strong scaling: throughput of a time step (results from Figure 10).

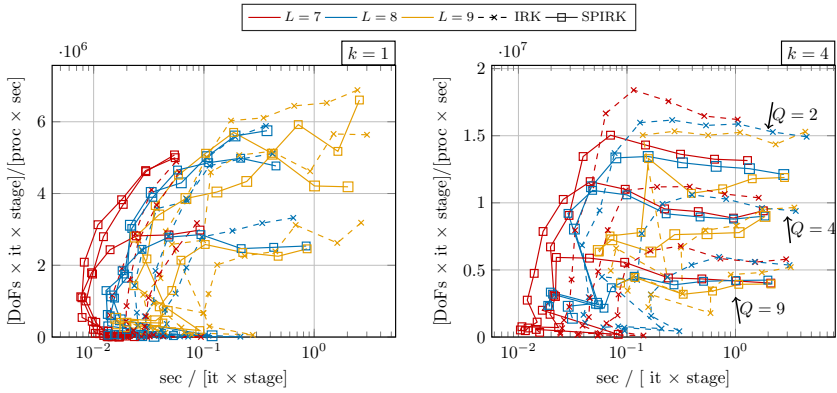


FIG. 12. Strong scaling: throughput per stage (results from Figure 10).

influences the scaling limits as well. However, we recall that a high number of stages allows us to use larger time steps due to an increased accuracy so that the additional costs might amortize.

As a summary, Figure 13 shows the measured speedup of stage-parallel IRK in comparison to IRK, categorized according to the number of DoFs per process. A clear speedup is obtained for less than 10k DoFs per process, i.e., half a million DoFs per node. For larger problem sizes per process, the picture is split. For more than 100k DoFs per process, IRK is consistently faster ($\approx 20\%$).

6.4. Batched execution. For the batched experiments, we replace the coarse-grid solver AMG by Chebyshev iterations around a point-Jacobi method with the same settings as of the smoothers (cf. section 6). Furthermore, we do not set up the coefficients of the Chebyshev polynomials for each block separately, but instead set them up with the approximation of the maximum eigenvalue of all blocks. This choice does not negatively affect the number of GMRES iterations, as indicated by Table 2.

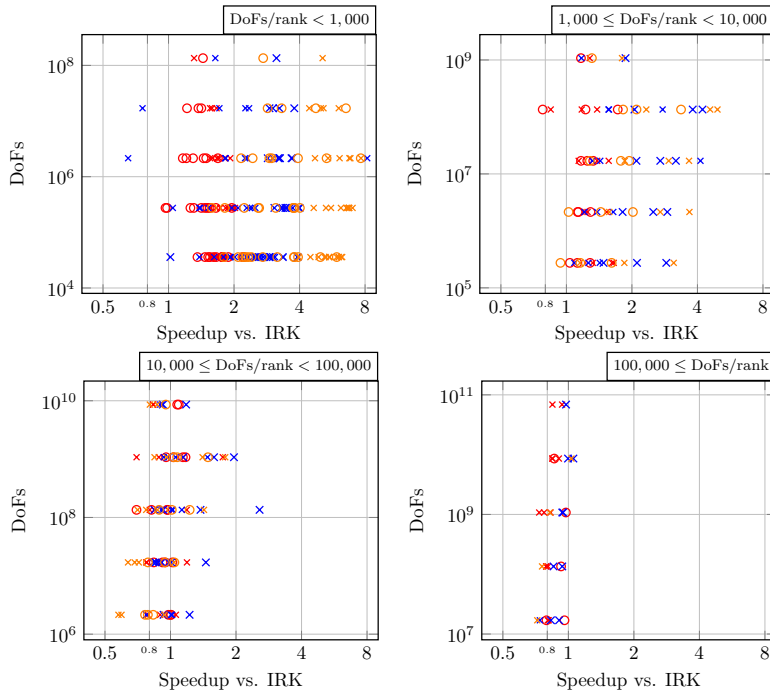


FIG. 13. Strong scaling: speedup categorized according to number of DoFs per process (results from Figure 10). Circles/x indicate $k = 1/k = 4$ and the colors red/blue/orange indicate $Q = 2/Q = 4/Q = 9$.

Figure 14(a) shows the timings of a batched execution of IRK on 16 nodes. The results are somewhat disappointing compared to our performance model and the number of iterations documented in Table 2. The timings are similar to the sequential execution of stages with only a small speedup, e.g., for $Q = 8$, 5%–25%.

The cause for this behavior is the implementation of communication of the intergrid transfer in `deal.II`. Due to a design choice of allowing black-box processing of the stages with a separate vector for each stage, all communication happens sequentially. By collecting all stages in a single vector, this issue could be overcome, as indicated by preliminary results obtained for GMG. Figures 14(b)–(c) show timings and speedup of a single-vector execution, of an execution with Q vectors, and of an execution with Q subgroups, in comparison to a sequential execution.

The results highlight the importance of batching all parts of the code. If this is not possible for all identified code paths, the execution gets serialized and the scalability is limited. The results for GMG indicate that one can expect higher speedups (up to 5.7 for $Q = 8$) for batched execution; however, the speedup will probably be smaller than that of the stage-parallel variants considered before.

We point out that using non-Cartesian meshes might shift the benefit from grouped execution toward a batched one, since loading metric terms only once and not Q times (once per process group) might be beneficial for the matrix-vector products, as discussed in subsection 4.4.

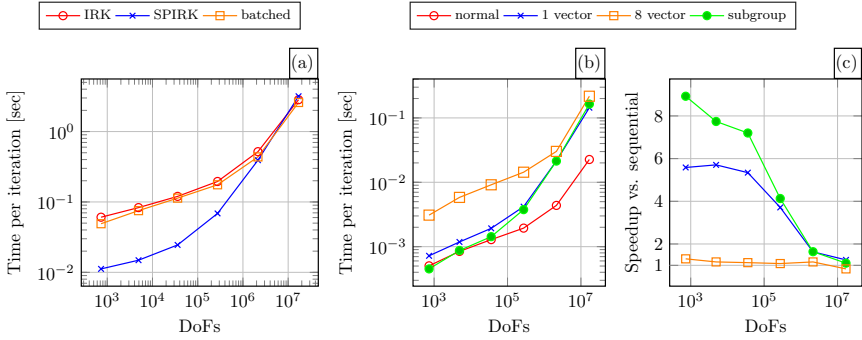


FIG. 14. (a) Comparison of time per iteration of IRK, stage-parallel IRK, and batch IRK for $k = 1$ and $Q = 8$ with 768 processes (16 compute nodes); (b)–(c) time of iteration and speedup of different execution modes to process $Q = 8$ stages in comparison to a sequential execution of the stages.

7. Real versus complex arithmetic—a comparison. For the IRK algorithm described in section 2, we have chosen L as the basis of the preconditioner, since its eigenvalues and eigenvectors are real and the resulting block system to be solved is real as well. However, the proposed concepts regarding stage-parallel implementation—particularly the data distribution and the communication patterns—are also applicable to the complex case, which arises when A_Q^{-1} is diagonalized directly. The advantage of factorizing A_Q^{-1} directly is that no global GMRES iterations—consisting of all Q stages—are needed and one can solve each block individually after the basis change. However, the disadvantage is that each block in (2.3) involves complex numbers:

$$(7.1) \quad \lambda_q M + \tau K = (\Re(\lambda_q) + i \cdot \Im(\lambda_q))M + \tau K = \underbrace{(\Re(\lambda_q)M + \tau K)}_{K'_q} + i \cdot \underbrace{\Im(\lambda_q)M}_{M'_q}$$

with $\lambda_q = \Re(\lambda_q) + i \cdot \Im(\lambda_q)$. Written as a two-by-two block-matrix system, $(K'_q + iM'_q)\mathbf{u}_q = \mathbf{v}_q$ becomes

$$(7.2) \quad \begin{bmatrix} K'_q & -M'_q \\ M'_q & K_q \end{bmatrix} \begin{bmatrix} \Re(\mathbf{u}_q) \\ \Im(\mathbf{u}_q) \end{bmatrix} = \begin{bmatrix} \Re(\mathbf{v}_q) \\ \Im(\mathbf{v}_q) \end{bmatrix}.$$

Since the structure of the resulting system is similar to that of a real Schur complement and, therefore, the algorithms to solve the system are equivalent, we will not detail this approach.

For systems of the type (7.2), PRESB is an efficient preconditioner [8]:

$$(7.3) \quad P_q = \begin{bmatrix} K'_q & -M'_q \\ M'_q & K_q + 2M'_q \end{bmatrix} = \begin{bmatrix} \mathbb{I}_n & -\mathbb{I}_n \\ 0 & \mathbb{I}_n \end{bmatrix} \begin{bmatrix} K'_q + M'_q & 0 \\ M'_q & K_q + M'_q \end{bmatrix} \begin{bmatrix} \mathbb{I}_n & \mathbb{I}_n \\ 0 & \mathbb{I}_n \end{bmatrix}.$$

For approximating the inverse of $(K'_q + M'_q)$, we use a single V-cycle. The corresponding solver diagram is shown in Figure 15(a). Please note that the blocks corresponding to the real and imaginary parts have to be solved in sequence (two V-cycles), possibly limiting the scalability.

Alternatively to PRESB, we also consider GMG applied directly to (7.2), consisting of both real and imaginary blocks. The corresponding solver diagram is shown

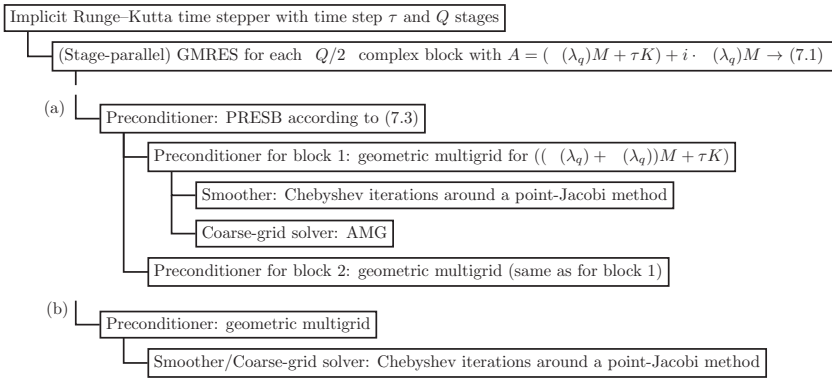


FIG. 15. Diagram of the complex solvers used to solve the heat problem in section 7: (a) PRESB as preconditioner, (b) GMG as preconditioner.

in Figure 15(b). The motivation for this is similar to the one for batched IRK: at the scaling limit, running one V-cycle on a vector with twice as many DoFs might be cheaper than running two V-cycles on smaller vectors in sequence.

The block structure of the transformed system (2.3) has some influence on the algorithms presented in section 4. The fact that only $\lceil Q/2 \rceil$ blocks can be solved independently allows one to parallelize only between $\lceil Q/2 \rceil$ “stages.” Naturally, each process group would be responsible for a stage pair. Furthermore, operations of the form $\mathbf{v} = (D \otimes \mathbb{I}_n)\mathbf{u}$ need minor adjustments, depending on the usage:

- application of A_Q^{-1} ($\mathbf{u}, \mathbf{v} \in \mathbb{R}^{n \times Q}$, $D \in \mathbb{R}^{Q \times Q}$) with stages assigned to $\lceil Q/2 \rceil$ processes and
- application of S^{-1} ($\mathbf{u} \in \mathbb{R}^{n \times Q}$, $\mathbf{v} \in \mathbb{C}^{n \times \lceil Q/2 \rceil}$, $D \in \mathbb{C}^{\lceil Q/2 \rceil \times \lceil Q/2 \rceil}$) and of S ($\mathbf{u} \in \mathbb{C}^{n \times \lceil Q/2 \rceil}$, $\mathbf{v} \in \mathbb{R}^{n \times Q}$, $D \in \mathbb{C}^{Q \times \lceil Q/2 \rceil}$).

In these cases, the distributed tensor algorithms from section 4 can be extended. They work on blocks of two vectors and perform block operations between the rotation steps, e.g.,

$$\begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} D_{11}\mathbf{u}_1 + D_{12}\mathbf{u}_2 \\ D_{21}\mathbf{u}_1 + D_{22}\mathbf{u}_2 \\ \vdots \end{bmatrix} + \begin{bmatrix} D_{13}\mathbf{u}_3 + D_{14}\mathbf{u}_4 \\ D_{23}\mathbf{u}_3 + D_{24}\mathbf{u}_4 \\ \vdots \end{bmatrix} + \dots$$

as block extension of (4.1).

The performance models derived in section 5 are also applicable to the complex case, with more pressure on the block solvers/preconditioners. The basis change—resulting in an implicit synchronization between all stages—has to be performed only once per time step as the $\lceil Q/2 \rceil$ blocks can be solved independently. Each block solve might be more expensive, since they might involve the solution of a two-by-two system. Comparing the complex case with the approximate case in real arithmetic, we give up some possibilities for (stage) parallelization by reducing the number of blocks ($Q \rightarrow \lceil Q/2 \rceil$) and by the sequential execution of two GMG V-cycles in the PRESB case. Not surprisingly, the obtained speedup (Figure 16) is about half of that of the stage-parallel preconditioner, which allows the parallel execution of all stages (see Figure 7). However, only two basis changes need to be performed per time step,

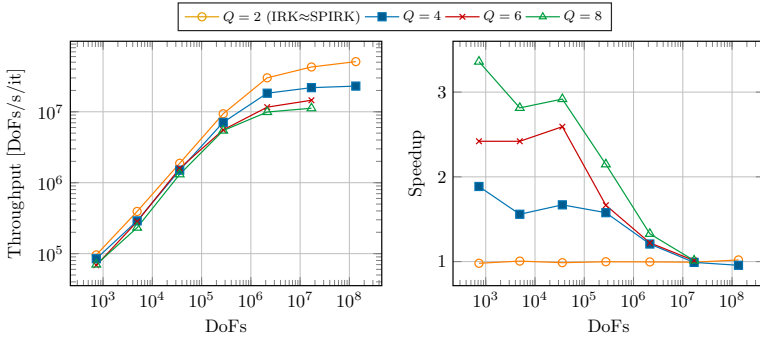


FIG. 16. Throughput and speedup of complex stage-parallel IRK versus complex IRK with PRESB for $k = 1$ with 768 processes (16 compute nodes).

which reduces the number of possibly expensive synchronization points.

According to our performance model, the total execution time at the scaling limit will be dominated by the maximum number of accumulated GMG iterations each process group has to execute. Hence, noncomplex stage-parallel IRK might be advantageous for low values of Q , but complex stage-parallel IRK might be competitive for high Q , since the number of iterations per stage is independent of Q . The results in Figure 17 verify our expectations, by comparing the times of the real-valued IRK studied before against the complex-valued IRK and the stage-parallel IRK solvers. Surprisingly, the two stage-parallel IRK algorithms show similar times for small problem sizes, demonstrating that the growing iteration counts of the outer solver of the noncomplex solver—indicated by the growing gap between the IRK implementations—can be compensated by its better parallel behavior. This can also be seen in Table 2, which shows a similar number of V-cycles to be executed in the context of noncomplex and complex stage-parallel IRK, respectively. However, we would like to note that it is hard to make a fair comparison, since the tolerances have different meanings in the complex and the noncomplex cases. Furthermore, while it is easy to integrate efficient complex arithmetic in our small benchmarks, this might not be the case when the time steppers should be applied in a black-box fashion. Against this background, a more detailed analysis of when to favor one method over the other will be the subject of future work.

Figure 18 compares the solution times of PRESB and GMG as preconditioners. Overall, PRESB seems to be superior compared to GMG due to fewer GMRES iterations (see Table 2). However, the scalability is comparable. In fact, Table 2 shows that GMG needs fewer V-cycles in total, indicating potential for better scalability of GMG once the optimizations discussed in subsection 6.4 have been realized.

Recently, Southworth et al. [32] presented a novel solver for IRK. They also exploit the complex-conjugate property of the eigenvalues and solve the pairs together. For preconditioning the blocks, they apply two V-cycles of (algebraic) multigrid as well. However, it is a bit more involved to solve the blocks, since they have the form $(\Re(\lambda_q)\mathbb{I}_n - \delta t M^{-1}K)^2 + \Im(\lambda_q)\mathbb{I}_n$. The algorithm proposed in that study is, however, inherently stage-serial as it requires the sequential solution of stage (pairs) by construction so that the algorithms presented in our publication are not applicable there.

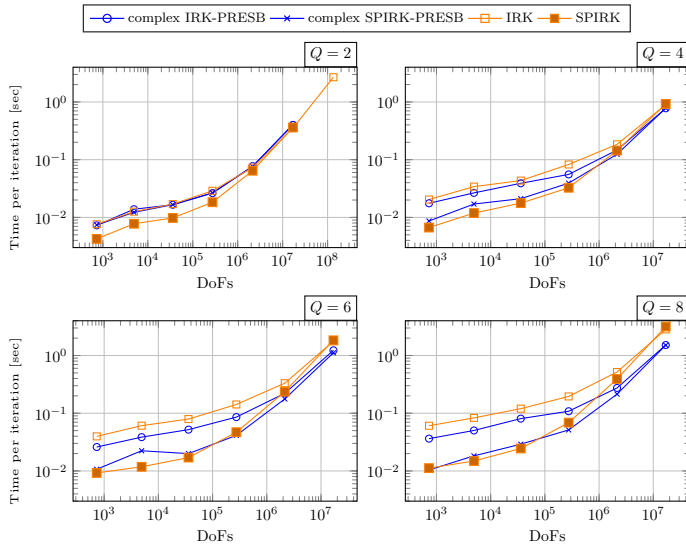


FIG. 17. Comparison of throughputs of noncomplex/complex IRK and stage-parallel IRK for $k = 1$ with 768 processes (16 compute nodes).

8. Possibilities for extensions. The present work has concentrated on building blocks for stage-parallel computations, solving a simple heat equation for demonstration purposes. Note that the iterative solution of (2.1) with a preconditioner built on the idea of using L from the LU decomposition of A_Q has previously been applied to nonsymmetric advection-diffusion equations (with time-dependent coefficients) [6, 27] and pure advection equations [6]. Consequently, the parallelization strategies discussed here are also applicable, with possible deviations on the block level if multi-grid is not an optimal solver and alternatives more suited for convection-dominated problems are employed.

The derivations of efficient stage-parallel algorithms for differential-algebraic systems of equations and for nonlinear partial differential equations, e.g., the Navier–Stokes equations, are still open research. In the following, we propose two simple adaptations of our concepts to nonlinear equations of the form $M\partial\mathbf{u}(t)/\partial t = \mathcal{N}(t, \mathbf{u})$. In this case, the Runge–Kutta scheme reads

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \tau \sum_{1 \leq q \leq Q} b_q \mathbf{k}_q \quad \text{with} \quad M \mathbf{k}_i = \mathcal{N} \left(t_n + c_i \tau, \mathbf{y}_n + \tau \sum_{1 \leq j \leq Q} a_{ij} \mathbf{k}_j \right).$$

The nonlinear system for the values \mathbf{k}_q can be solved, e.g., via a Newton method with the linearized version of the equation,

$$(8.1) \quad \underbrace{\left(A_Q^{-1} \otimes M - \tau \text{diag}(\mathcal{L}_1, \dots, \mathcal{L}_Q) \right)}_{\tilde{\mathcal{J}}} \underbrace{\left(A_Q \otimes \mathbb{I}_n \right)}_{\Delta \mathbf{k}} \Delta \mathbf{k} = \mathbf{f},$$

with the Jacobian \mathcal{L}_q being the linearization of \mathcal{N} regarding the q th stage and \mathbf{f} a suitable residual. Using the approximation of constant blocks $\tilde{\mathcal{L}} \approx \mathcal{L}_q$ for all $q \in$

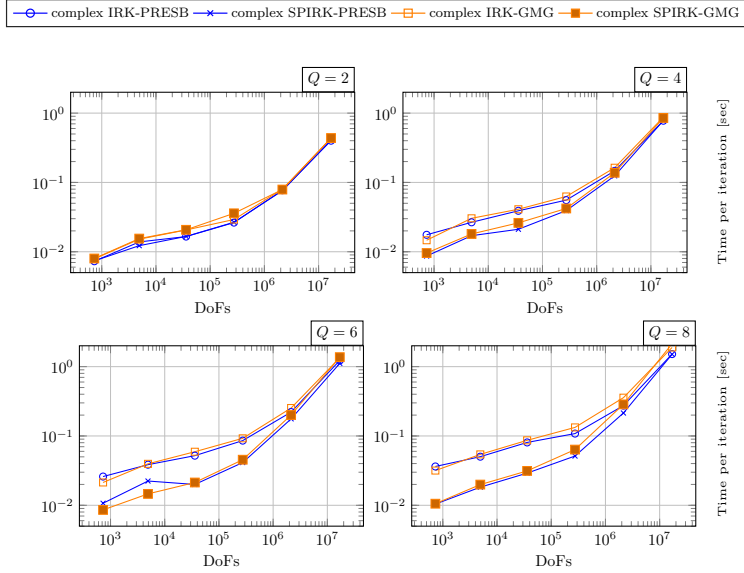


FIG. 18. Comparison of throughputs of complex IRK and stage-parallel IRK with PRESB or GMG as preconditioner for $k = 1$ with 768 processes (16 compute nodes).

$\{1, \dots, Q\}$, we can construct preconditioners for $\hat{\mathcal{J}}$ analogous to those used in the linear case, $P = (S \otimes \mathbb{I}_n)(\Lambda \otimes M - \tau \mathbb{I}_Q \otimes \hat{\mathcal{L}})(S^{-1} \otimes \mathbb{I}_n)$, where Λ and S are the eigenvalues and eigenvectors either of A_Q^{-1} or of its lower triangular factor.

Alternatively, using the real Schur decomposition $A_0^{-1} = Q_0 R_0 Q_0^T$ on (8.1) yields

$$(8.2) \quad \underbrace{(R_Q \otimes M - (Q_Q^T \otimes \mathbb{I}_n) \tau \text{diag}(\mathcal{L}_1, \dots, \mathcal{L}_Q)(Q_Q \otimes \mathbb{I}_n))}_{\hat{\mathcal{J}}} \underbrace{(R_Q^{-1} Q_Q^T \otimes \mathbb{I}_n)}_{\Delta \mathbf{k}} \mathbf{k} = \underbrace{(Q_Q^T \otimes \mathbb{I}_n)}_{\hat{\mathbf{f}}} \mathbf{f}.$$

Instead of directly solving $\Delta \mathbf{k} = (Q_Q R_Q \otimes \mathbb{I}_n) \tilde{\mathcal{J}}^{-1} \hat{\mathbf{f}}$, Southworth et al. [32] proposed to use a Newton-like method with an approximation of $\tilde{\mathcal{J}}$ in terms of a block-diagonal representation. For details, we refer to their publication. Given the algorithms proposed in the present work, it is natural to solve $\tilde{\mathcal{J}}$ iteratively using one of the block-Jacobi preconditioners listed in [32] as the exact matrix-vector product (8.2) can be evaluated with the building blocks of section 2 in a stage-parallel way due to its resemblance with \mathcal{A} in (2.2). It is an open research question to derive an optimal preconditioner in that context, resembling the coupled form (2.4), and evaluate its efficiency against the choice in [32].

9. Conclusions and outlook. For distributed-memory computing platforms, we have presented implementations of IRK algorithms, including the novel preconditioner proposed by Axelsson and Neytcheva [7]. The algorithms allow us to run the matrix-vector multiplication and the preconditioner in parallel by process groups

associated with each stage. Upon a basis change involving all stages, inner block solvers can be applied independently in a black-box fashion. We have identified that the tensor operations $\mathbf{v} = (\mathbb{I}_Q \otimes C)\mathbf{u}$ and $\mathbf{v} = (D \otimes \mathbb{I}_n)\mathbf{u}$ are the main building blocks and have proposed efficient parallel algorithms.

We have presented a detailed performance analysis of the stage-parallel preconditioner implementation for a time-dependent heat problem on up to 150k processes on 3k CPU compute nodes, using state-of-the-art matrix-free GMG solvers. Furthermore, its performance has been compared to that of an implementation not using stage parallelism. We observed that the stage-parallel implementation is able to significantly shift the scaling limit and reaches speedups $\leq Q$. In absolute numbers, the proposed solvers and implementations make it possible to obtain high solver efficiencies down to less than 0.05 seconds per time step for four-stage IRK schemes. However, far from the scaling limit, possible load imbalances between the solvers of the stages and communication overhead lead to a slight drop in performance by 13% lower throughput on average in comparison to the non-stage-parallel IRK implementation.

The algorithms are also applicable to the case when the system matrix arising from the IRK method is directly factorized, requiring complex arithmetic or the solution of two-by-two blocks. This limits the scalability in comparison to the case of the stage-parallel preconditioner. Results, however, show that the lower number of iterations and the scalability balance each other, leading to similar minimum times to solution of stage-parallel direct factorization and preconditioning for a high number of stages ($Q \leq 8$).

We have also discussed batching the operations of all stages instead of assigning each stage to a process group, showing the challenge in terms of black-box interfaces. In future work, we plan to study stage parallelism for nonlinear partial differential equations, e.g., Navier–Stokes equations. Furthermore, we intend to improve the batched implementation as well as make further investigations of its performance and usability within a library context, which might require code generation [15].

Acknowledgments. The authors acknowledge discussions with Ben Southworth regarding extensions of the algorithms toward nonlinear equations and collaboration with the `deal.II` community.

REFERENCES

- [1] R. ABU-LABDEH, S. MACLACHLAN, AND P. E. FARRELL, *Monolithic Multigrid for Implicit Runge-Kutta Discretizations of Incompressible Fluid Flow*, preprint, <https://arxiv.org/abs/2202.07381>, 2022.
- [2] M. ADAMS, M. BREZINA, J. HU, AND R. TUMINARO, *Parallel multigrid smoothing: Polynomial versus Gauss-Seidel*, *J. Comput. Phys.*, 188 (2003), pp. 593–610.
- [3] D. ARNDT, W. BANGERTH, B. BLAIS, M. FEHLING, R. GASSMÖLLER, T. HEISTER, L. HELTAI, U. KÖCHER, M. KRONBICHLER, M. MAIER, P. MUNCH, J.-P. PELTERET, S. PROELL, K. SIMON, B. TURCKINS, D. WELLS, AND J. ZHANG, *The deal.II library, version 9.3*, *J. Numer. Math.*, 29 (2021), pp. 171–186.
- [4] D. ARNDT, W. BANGERTH, D. DAVYDOV, T. HEISTER, L. HELTAI, M. KRONBICHLER, M. MAIER, J.-P. PELTERET, B. TURCKINS, AND D. WELLS, *The deal.II finite element library: Design, features, and insights*, *Comput. Math. Appl.*, 81 (2021), pp. 407–422.
- [5] O. AXELSSON, *Global integration of differential equations through lobatto quadrature*, *BIT*, 4 (1964), pp. 69–86.
- [6] O. AXELSSON, I. DRAVINS, AND M. NEYTCHEVA, *Stage-parallel preconditioners for implicit Runge-Kutta methods of arbitrarily high order, linear problems*, to appear.
- [7] O. AXELSSON AND M. NEYTCHEVA, *Numerical solution methods for implicit Runge-Kutta methods of arbitrarily high order*, in 21st Conference on Scientific Computing, Vysoké Tatry-Podbanské 7, Slovakia, 2020, pp. 11–20.

- [8] O. AXELSSON, M. POURBAGHER, AND D. K. SALKUYEH, *Robust Iteration Methods for Complex Systems with an Indefinite Matrix Term*, preprint, <https://arxiv.org/abs/2110.00537>, 2021.
- [9] T. A. BICKART, *An efficient solution process for implicit runge-Kutta methods*, SIAM J. Numer. Anal., 14 (1977), pp. 1022–1027.
- [10] M. BOLTEN, D. MOSER, AND R. SPECH, *A multigrid perspective on the parallel full approximation scheme in space and time*, Numer. Linear Algebr. Appl., 24 (2017), e2110.
- [11] K. BURRAGE, C. ELDERSHAW, AND R. SIDJE, *A parallel matrix-free implementation of a Runge-Kutta code*, in Proceedings of the Joint Australian-Taiwanese Workshop on Analysis and Applications, Australian National University, Mathematical Sciences Institute, 1999, pp. 83–88.
- [12] J. C. BUTCHER, *On the implementation of implicit Runge-Kutta methods*, BIT, 16 (1976), pp. 237–240.
- [13] L. E. CANNON, *A Cellular Computer to Implement the Kalman Filter Algorithm*, Ph.D. thesis, Montana State University, 1969.
- [14] J. J. B. DE SWART, W. M. LIOEN, AND W. A. VAN DER VEEN, *Specification of PSIDE*, CWI, Amstredam, 1998.
- [15] P. E. FARRELL, R. C. KIRBY, AND J. MARCHENA-MENENDEZ, *Irksome: Automating Runge-Kutta time-stepping for finite element methods*, ACM Trans. Math. Software, 47 (2021), pp. 30/1–26.
- [16] M. W. GEE, C. M. SIEFERT, J. J. HU, R. S. TUMINARO, AND M. G. SALA, *ML 5.0 Smoothed Aggregation User's Guide*, Technical Report SAND2006-2649, Sandia National Laboratories, 2006.
- [17] G. HAGER AND G. WELLEIN, *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press, Boca Raton, FL, 2010.
- [18] K. R. JACKSON AND S. P. NØRSETT, *The potential for parallelism in Runge-Kutta methods. Part I. RK formulas in standard form*, J. Numer. Anal., 32 (1995), pp. 49–82.
- [19] L. O. JAY AND T. BRACONNIER, *A parallelizable preconditioner for the iterative solution of implicit Runge-Kutta-type methods*, J. Comput. Appl. Math., 111 (1999), pp. 63–76.
- [20] C. A. KENNEDY, M. H. CARPENTER, AND R. M. LEWIS, *Low-storage, explicit Runge-Kutta schemes for the compressible Navier-Stokes equations*, Appl. Numer. Math., 35 (2000), pp. 177–219.
- [21] T. KOLEV, P. FISCHER, M. MIN, J. DONGARRA, J. BROWN, V. DOBREV, T. WARBURTON, S. TOMOV, M. S. SHEPHARD, A. ABDELFAHATTAH, V. BARRA, N. BEAMS, J.-S. CAMIER, N. CHALMERS, Y. DUDOUT, A. KARAKUS, I. KARLIN, S. KERKEMEIER, Y.-H. LAN, D. MEDINA, E. MERZARI, A. OBABKO, W. PAZNER, T. RATHNAYAKE, C. W. SMITH, L. SPIES, K. SWIRYDOWICZ, J. THOMPSON, A. TOMBOULIDES, AND V. TOMOV, *Efficient eascale discretizations: High-order finite element methods*, Int. J. High Perform. Comput. Appl., 35 (2021), pp. 527–552.
- [22] M. KRONBICHLER AND K. KORMANN, *A generic interface for parallel cell-based finite element operator application*, Comput. Fluids, 63 (2012), pp. 135–147.
- [23] M. KRONBICHLER AND K. KORMANN, *Fast matrix-free evaluation of discontinuous Galerkin finite element operators*, ACM Trans. Math. Software, 45 (2019), 29.
- [24] M. KRONBICHLER, D. SASHKO, AND P. MUNCH, *Enhancing data locality of the conjugate gradient method for high-order matrix-free finite-element implementations*, Int J. High Perform. Comput. Appl. (2022).
- [25] J. L. LIONS, Y. MADAY, AND G. TURINICI, *A “parareal” in time discretization of PDEs*, C. R. Acad. Sci. Ser. I Math., 332 (2001), pp. 661–668.
- [26] K.-A. MARDAL, T. K. NILSSEN, AND G. A. STAFF, *Order-optimal preconditioners for implicit Runge-Kutta schemes applied to parabolic PDEs*, SIAM J. Sci. Comput., 29 (2007), pp. 361–375.
- [27] M. MASUD RANA, V. E. HOWLE, K. LONG, A. MEEK, AND W. MILESTONE, *A new block preconditioner for implicit Runge-Kutta methods for parabolic PDE problems*, SIAM J. Sci. Comput., 43 (2021), pp. S475–S495.
- [28] P. MUNCH, T. HEISTER, L. PRIETO SAAVEDRA, AND M. KRONBICHLER, *Efficient Distributed Matrix-Free Multigrid Methods on Locally Refined Meshes for FEM Computations*, ACM Trans. Parallel Comput., <https://arxiv.org/abs/2203.12292>, 2023, <https://doi.org/10.1145/3580314>.
- [29] P. MUNCH, K. KORMANN, AND M. KRONBICHLER, *hyper.deal: An efficient, matrix-free finite-element library for high-dimensional partial differential equations*, ACM Trans. Math. Software, 47 (2021), pp. 33/1–34.

- [30] W. PAZNER AND P.-O. PERSSON, *Stage-parallel fully implicit Runge-Kutta solvers for discontinuous Galerkin fluid simulations*, J. Comput. Phys., 335 (2017), pp. 700–717.
- [31] B. S. SOUTHWORTH, O. KRZYSIK, AND W. PAZNER, *Fast solution of fully implicit Runge-Kutta and discontinuous Galerkin in time for numerical PDEs, part II: Nonlinearities and DAEs*, SIAM J. Sci. Comput., 44 (2022), pp. A636–A663.
- [32] B. S. SOUTHWORTH, O. KRZYSIK, W. PAZNER, AND H. D. STERCK, *Fast solution of fully implicit Runge-Kutta and discontinuous Galerkin in time for numerical PDEs, part I: The linear setting*, SIAM J. Sci. Comput., 44 (2022), pp. A416–A443.
- [33] G. A. STAFF, K.-A. MARDAL, AND T. K. NILSEN, *Preconditioning of fully implicit Runge-Kutta schemes for parabolic PDEs*, MIC J., 27 (2006), pp. 109–123.
- [34] R. A. VAN DE GEIJN AND J. WATTS, *Summa: Scalable universal matrix multiplication algorithm*, Concurrency-Pract. Ex., 9 (1997), pp. 255–274.
- [35] E. HAIRER AND G. WANNER, *Solving Ordinary Differential Equations II*, Springer Ser. Comput. Math. 14, Springer, Berlin, 1996.

Paper VI

hyper.deal: An Efficient, Matrix-free Finite-element Library for High-dimensional Partial Differential Equations

PETER MUNCH, Institute for Computational Mechanics, Department of Mechanical Engineering, Technical University of Munich, Germany and Institute of Material Systems Modeling, Helmholtz-Zentrum Hereon, Germany

KATHARINA KORMANN, Max Planck Institute for Plasma Physics, Germany, Department of Mathematics, Technical University of Munich, Germany, and Department of Information Technology, Uppsala University, Sweden

MARTIN KRONBICHLER, Institute for Computational Mechanics, Department of Mechanical Engineering, Technical University of Munich, Germany and Department of Information Technology, Uppsala University, Sweden

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

CCS Concepts: • **Mathematics of computing** → **Mathematical software performance**; *Partial differential equations*; *Solvers*; • **Computer systems organization** → *Multicore architectures*; • **Applied computing** → *Physics*;

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

Authors’ addresses: P. Munch, Institute for Computational Mechanics, Department of Mechanical Engineering, Technical University of Munich, Boltzmannstr. 15, 85748 Garching b. München, Germany, Institute of Material Systems Modeling, Helmholtz-Zentrum Hereon, Max-Planck-Str. 1, 21502 Geesthacht, Germany; email: munch@lnm.mw.tum.de; K. Kormann, Max Planck Institute for Plasma Physics, Boltzmannstr. 2, 85748 Garching b. München, Germany, Department of Mathematics, Technical University of Munich, Boltzmannstr. 3, 85748 Garching b. München, Germany, Department of Information Technology, Uppsala University, Box 337, 75105 Uppsala, Sweden; email: katharina.kormann@ipp.mpg.de; M. Kronbichler, Institute for Computational Mechanics, Department of Mechanical Engineering, Technical University of Munich, Boltzmannstr. 15, 85748 Garching b. München, Germany, Department of Information Technology, Uppsala University, Box 337, 75105 Uppsala, Sweden; email: kronbichler@lnm.mw.tum.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

0098-3500/2021/09-ART33 \$15.00

<https://doi.org/10.1145/3469720>

Additional Key Words and Phrases: Matrix-free operator evaluation, discontinuous Galerkin methods, high-dimensional, high-order, Vlasov–Poisson equation, MPI-3.0 shared memory

ACM Reference format:

Peter Munch, Katharina Kormann, and Martin Kronbichler. 2021. *hyper.deal: An Efficient, Matrix-free Finite-element Library for High-dimensional Partial Differential Equations*. *ACM Trans. Math. Softw.* 47, 4, Article 33 (September 2021), 34 pages.
<https://doi.org/10.1145/3469720>

1 INTRODUCTION

Three-dimensional (3D) problems are today solved in great detail up to supercomputer scale with codes relying on domain decomposition. With the increase in computational power and the advance in algorithms, also the solution of higher-dimensional problems comes within the realm of the possible. With this contribution, we target moderately high-dimensional (\leq **six-dimensional (6D)**) problems with complex geometry requirements in some of the dimensions. Our primary target are kinetic equations that describe the evolution of a distribution function in phase space. Such Boltzmann-type equations are for instance used in the modeling of magnetic confinement fusion, where the evolution of a plasma is described by a distribution function that evolves according to the Vlasov equation coupled to a system of Maxwell’s equations for its self-consistent fields. Other areas of the application of phase space are, e.g., cosmic microwave background radiation or magnetic reconnection in the Earth’s magnetosphere.

In phase space applications, there are two distinct sets of variables, configuration space and velocity space, with the distribution function nonlinearly coupled to additional equations in the configuration space. Especially in configuration space, the domain can be geometrically complex (e.g., torus-like shapes as in the case of tokamak and stellarator fusion reactors), necessitating a flexible description of unstructured meshes as is commonly provided by **finite-element method (FEM)** libraries for grids in up to three dimensions. In the fusion community, the focus has shifted from simulations of the core to simulations of the edge and scrape-off layer, where the geometry of the problem gets more involved and density profiles get less homogeneous.

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

The presented concept is not limited to the description of the phase space. Possible applications of a high-dimensional FEM library, like *hyper.deal*, could for instance include 3D problems that involve a low-dimensional parameter space or low-dimensional Fokker–Planck-type equations, e.g., the Black–Scholes equation for option pricing in mathematical finance.

1.1 Related Work

While the solution of **partial differential equations (PDEs)** on complex domains in up to three dimensions is a well-studied problem, PDEs on complex domains in dimensions higher than three are not tackled by generic FEM libraries to date. Some libraries have started to extend their capabilities to higher dimensions on structured grids. An example is the *YaspGrid* module of the finite-element library *DUNE*, which implements structured grids in arbitrary dimensions [8]. In [37], Helmholtz equations on adaptive structured grids up to $d = 4$ are studied. Higher-dimensional

problems are also solved based on sparse grids [17] or low-rank tensors [6]. However, these techniques require a certain low-rank structure of the solution. In the plasma community, some specialized codes exist that target gyrokinetic or kinetic equations. The Geky11 code [18] is a **discontinuous Galerkin (DG)** solver for plasma physics applications, and a fully kinetic version for Cartesian grids was presented in [22]. The authors use higher-order serendipity elements to reduce the number of **degrees of freedom (DoF)**.

The specifics of domain decomposition in higher dimensions have only been studied recently. In [27], a 6D domain decomposition for a semi-Lagrangian solver on a Cartesian grid of the Vlasov–Poisson system was investigated and the high demands on memory transfer between neighboring processes due to the increased surface-to-volume ratio with increasing dimensionality were highlighted. The parallelization of a similar algorithm has also been addressed in [42]. However, the domain decomposition is limited to configuration space in that work, which poses a strong limit to the scalability of the implementation.

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

1.2 Our Contribution

This work shows a way to extend a low-dimensional general-purpose high-order matrix-free FEM library to high dimensions. We discuss how to cope with possible performance deteriorations due to the “curse of dimensionality” and present special-purpose concepts exploiting the structure of the phase space. In particular, we show how to create a high-dimensional triangulation by taking the tensor product of possibly unstructured partitioned triangulations from a low-dimensional FEM library. This way to construct a triangulation enables us to re-use information related to the low-dimensional finite-element space, quadrature, and mapping and to combine the information on the fly.

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

The concepts described in this work have been implemented and are available under the LGPL 3.0 license as the library `hyper.deal` hosted at <https://github.com/hyperdeal/hyperdeal>. It extends, as an example, the open-source FEM library `deal.II` [4] to high dimensions. Analyses of both the node-level performance and strong/weak scaling conducted for this library confirm the suitability of the proposed concepts for solving partial differential equations in high dimensions.

The remainder of this work is organized as follows. In Section 2, we introduce the model problem equation and discretize it with a skew-symmetric DG approach. Section 3 introduces the concept of a tensor product of partitioned low-dimensional meshes and details its implementation for phase space. In Section 4, we describe a shared-memory vector based on MPI-3.0, which keeps the memory overhead due to ghost regions to a minimum without the requirement to add a second parallelization concept. Section 5 presents performance results for the advection equation, confirming the efficiency of the design decisions made during the implementation. Section 6 explains

how hyper .deal can efficiently be combined with a deal .II-based Poisson solver and shows scaling results for a benchmark problem from plasma physics. Finally, Section 7 summarizes our conclusions and points to further research directions.

2 DISCRETIZATION WITH DG METHODS

2.1 Model Problem

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

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

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

2.2 DG Discretization of the Advection Equation

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

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

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

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

The factor β controls the formulation of the flux: $\beta = \frac{1}{2}$ represents the skew-symmetric version, whereas $\beta = 0$ corresponds to the conservative DG method, see also [26]. Integration $\int_{\Omega} d\Omega$ and derivation ∇ are not performed in the real space but in the reference space $\Omega_0^{(e)}$ and $\Gamma_0^{(e)}$ and require a mapping to the reference coordinates, i.e., $\int_{\Omega} d\Omega = \int_{\Omega^{(e)}} |\mathcal{J}| d\Omega^{(e)}$ and $\nabla = \mathcal{J}^{-T} \nabla_{\vec{\xi}}$, where \mathcal{J} is the Jacobian matrix of the mapping from reference to real space and $|\mathcal{J}|$ its determinant.

To discretize this equation in space, we use a tensor product of **one-dimensional (1D)** nodal polynomials with nodes in the Gauss–Lobatto points. These nodal polynomials are chosen to ensure minimal data access on faces [39]. The integrals are evaluated numerically by weighted sums. We consider both the usual Gauss(–Legendre) quadrature rules and the integration directly in the Gauss–Lobatto points without the need for interpolation (collocation setup [13]). The resulting semi-discrete system has the following form:

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

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

Table 1. Estimated Working Set of Different Stages of a Matrix-free Evaluation of the Advection Operator for ECL with Shape Functions of Polynomial Degree k and n_q Quadrature Points in Each Direction

| | Stage | | Working set |
|-----|--------------------|-------|---|
| (1) | Sum factorization | > | $\max((k+1)^d, n_q^d)$ |
| (2) | Derived quantities | > | $(d+1) \cdot n_q^d$ |
| (3) | Flux computation | \gg | $(k+1)^d + 2 \cdot d \cdot (k+1)^{d-1}$ |

(1) Includes both the source and the destination element vector; (2) includes the buffers needed during testing and face evaluation; and (3) includes the DoFs of neighboring cells, needed during flux computation.

Table 2. Estimated Minimal Memory Consumption for D -dimensional Advection Simulations with N DoFs on p Processes, $n_q = k + 1$, 3 Vectors (one with Ghost Values), and Pre-computed Mapping Data

| | Reason | | Times | | Amount |
|-----|----------------------|---|-------|---|---|
| (1) | Vector | | 3 | . | N |
| (2) | Ghost DoFs (+buffer) | + | 2 | . | $2 \cdot d \cdot N^{\frac{d-1}{d}} \cdot p^{\frac{1}{d}}$ |
| (3) | Mapping | + | | | $d^2 \cdot N$ |

blocks equal to the number of unknowns per element in the case of the consistent Gauss quadrature. For **two-dimensional (2D)** and 3D high-order DG methods, efficient matrix-free operator evaluations for the individual operators \mathcal{A} [29, 30] and \mathcal{M}^{-1} [32] as well as for the merged operator $\mathcal{M}^{-1}\mathcal{A}$ are known in the context of fluid mechanics [28], structural mechanics [11], and acoustic wave propagation [39]. DG methods and matrix-free operator evaluation kernels are part of many low-dimensional general-purpose FEM libraries nowadays [3, 5, 24, 35, 40].

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

2.3 Challenges

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

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

with significantly smaller resolutions could be solved. Even if one would decide not to pre-compute the mapping information, one would need to store the coordinate of each vertex $\mathcal{O}(d)$, what might also already require too much memory, and to solve for the Jacobian $\mathcal{J} \in \mathbb{R}^{d \times d}$ at each quadrature point, which is a $\mathcal{O}(d^3)$ operation.

- (2) **Increased ghost-value exchange due to increased surface-to-volume ratio:** The communication amount scales with $2 \cdot d \cdot N^{(d-1)/d} \cdot p^{1/d}$ for a hypercube-shaped partition (cf. Table 2). According to [30], the MPI ghost-value exchange already leads to a noticeable share of time in purely MPI-parallelized applications (30% for 3D Laplacian) in comparison to the highly efficient matrix-free operator evaluations if computations are performed on a single compute node for 3D problems. For high dimensions, the situation is even worse: an estimation with $d = 6$, $N = 10^{12}$, $p = 1,024 \cdot 48$ (1,024 compute nodes with 48 processes each) gives that the size of the ghost values is at least 72% of the size of the actual solution vector.
- (3) **Decreased efficiency of the operator evaluation due to working sets exceeding the cache capacities:** The working set of a cell with shape functions of polynomial degree k and n_q quadrature points in each direction is at least $\mathcal{O}(\max((k+1)^d, n_q^d))$ (cf. Table 1) so that for high order and/or dimension the data eventually drops out of the cache during each sum-factorization sweep of one cell. This can lead to a significant drop in performance once the data has to be streamed from the slow main memory.

This work shows how these problems can be mitigated by certain design choices: We address problem (1) by restricting ourselves to the tensor product of two grids in 1D–3D. This reduces the size of the mapping data and makes it possible to reuse much of the infrastructure available in a low-dimensional library, such as `deal.II`. We will describe in the next section how such a tensor product can be formed. Problem (2) demonstrates that it is essential to exploit shared-memory parallelism particularly in high dimensions. For the given example, the size of ghost values could be halved to 37% if all 48 processes on a compute node shared their locally owned values. Therefore, we propose a novel shared-memory implementation of finite-element-type vectors, which is based on MPI-3.0. To mitigate problem (3), we try to minimize the number of cache misses due to increased working-set sizes by reorganizing the loops. To reduce the working set with cross-element vectorization, we also allow to use narrower SIMD registers containing data from fewer elements than the given instruction-set extensions allow. For example, we use AVX2 or SSE2 instead of AVX-512, or by working directly with doubles and relying on auto-optimization of the compiler. We defer the investigation of explicit vectorization within elements to future work.

3 HYPER.DEAL: A TENSOR PRODUCT OF TWO MESHES

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

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

ALGORITHM 1: Element-centric Loop

```

/* loop over all cells (cell pairs) */
1 foreach  $(c_x, c_v) \in C_{\vec{x}} \times C_{\vec{v}}$  do
2   process_cell( $c_x, c_v$ )
   /* loop over all  $x$ -faces (face-cell pairs) */
3   for  $0 \leq i < 2 \cdot d_x$  do
4     process_face(face( $c_x, i$ ),  $c_v$ ); /* face( $c, i$ ) returns the  $i$ -th face of the cell  $c$  */
   /* loop over all  $v$ -faces (cell-face pairs) */
5   for  $0 \leq i < 2 \cdot d_v$  do
6     process_face( $c_x$ , face( $c_v, i$ ))

```

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

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

with the blocks being the respective matrices of the \vec{x} -space and the \vec{v} -space. Hence, the inverse \mathcal{J}^{-1} is the inverse of each of its blocks. Furthermore, face integrals over the faces in phase space $\Gamma^{(e)} = (\Gamma_{\vec{x}}^{(e)} \otimes \Omega_{\vec{v}}^{(e)}) \cup (\Omega_{\vec{x}}^{(e)} \otimes \Gamma_{\vec{v}}^{(e)})$ can be split into integration over \vec{x} -space faces $\Gamma_{\vec{x}}^{(e)} \otimes \Omega_{\vec{v}}^{(e)}$ and integration over \vec{v} -space faces $\Omega_{\vec{x}}^{(e)} \otimes \Gamma_{\vec{v}}^{(e)}$. The following relation holds for \vec{x} -space faces: due to $\vec{n}^\top = (\vec{n}_{\vec{x}}^\top, 0)$ for \vec{x} -space faces, $\vec{n} \cdot \vec{a} = \vec{n}_{\vec{x}} \cdot \vec{a}_{\vec{x}}$. Analogously, $\vec{n} \cdot \vec{a} = \vec{n}_{\vec{v}} \cdot \vec{a}_{\vec{v}}$ is true for \vec{v} -space faces. Hence, the specialization of Equation (2) solved by hyper.deal is:

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

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

3.1 Triangulation

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

$$\mathcal{T} := \mathcal{T}_{\vec{x}} \otimes \mathcal{T}_{\vec{v}}. \quad (7)$$

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

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

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

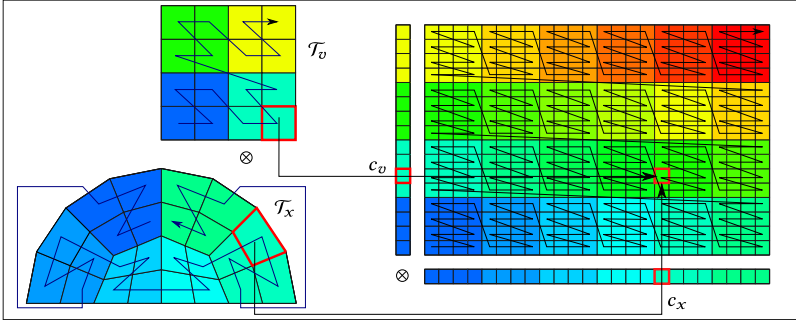


Fig. 1. On-the-fly mesh generation of a high-dimensional distributed triangulation (right) by taking the tensor product of two low-dimensional triangulations (left) from a low-dimensional library (for a hypothetical setup of 24 partitions and of a 6×4 partitioning of the 4D space). Cells are ordered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

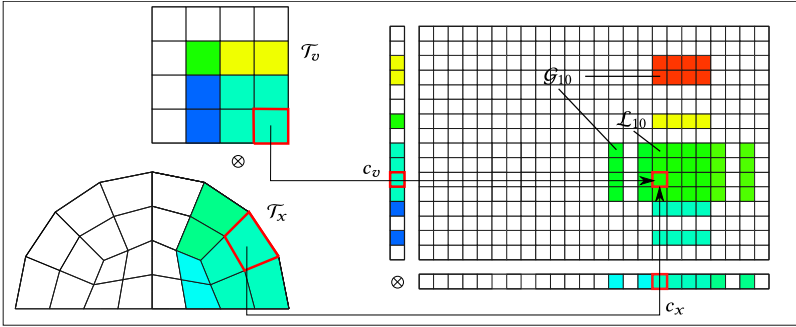


Fig. 2. Local view of an arbitrary process for the hypothetical setup of Figure 1 (here: process 10). Local cells \mathcal{L}_{10} result as tensor product of local cells from each low-dimensional triangulation and ghost cells \mathcal{G}_{10} as tensor product of one local and one ghost cell.

ALGORITHM 2: Face-centric Loop (Boundary Faces not Shown)

```

/* loop over all cells (cell pairs) */
1 foreach  $c \in C_{\vec{x}} \times C_{\vec{v}}$  do
2   | process_cell(c)
   /* loop over all x-faces (face-cell pairs) */
3 foreach  $f \in I_{\vec{x}} \times C_{\vec{v}}$  do
4   | process_face(f)
   /* loop over all v-faces (cell-face pairs) */
5 foreach  $f \in C_{\vec{x}} \times I_{\vec{v}}$  do
6   | process_face(f)

```

(FCL) visit all faces only once in a separate loop (with test functions from both sides of an interior face).

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

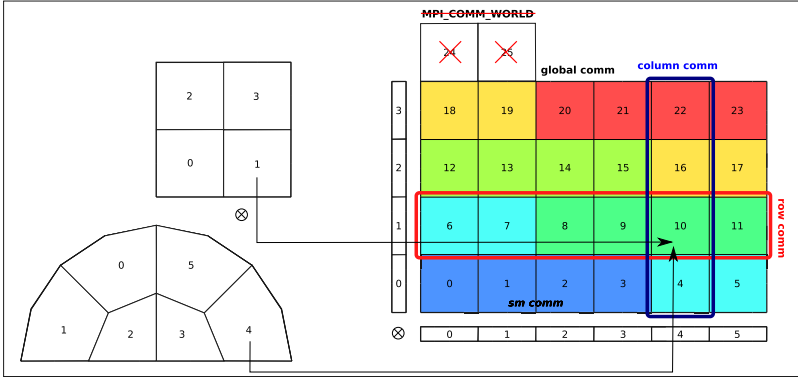


Fig. 3. Four MPI communicators (constructed from MPI_COMM_WORLD with 26 processes and shared-memory domains of size 4) used in hyper.deal for the hypothetical setup of Figure 1: global_comm collects all non-empty partitions; row_/column_comm collects processes owning the same partitions of $\mathcal{T}_{\vec{v}}/\mathcal{T}_{\vec{x}}$;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

3.2 Domain Decomposition

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

$$\mathcal{T}_{\vec{x}} = \bigoplus_{0 \leq i < p_{\vec{x}}} \mathcal{T}_{\vec{x}}^i \quad \text{and} \quad \mathcal{T}_{\vec{v}} = \bigoplus_{0 \leq j < p_{\vec{v}}} \mathcal{T}_{\vec{v}}^j, \quad (9)$$

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

The partition of the phase space belonging to rank $r(i, j)$ is constructed by $\mathcal{T}^{r(i, j)} := \mathcal{T}_{\vec{x}}^i \otimes \mathcal{T}_{\vec{v}}^j$, where ranks are enumerated lexicographically according to $r(i, j) := j \cdot p_{\vec{x}} + i$ with the \vec{x} -rank i being the fastest running index. As a consequence, a quasi-Cartesian partitioning is obtained (see Figure 1). Figure 2 illustrates the neighborhood relations within the tensor product grid

As a consequence, a quasi-Cartesian partitioning is obtained (see Figure 1). Note that it might be advantageous in some cases not to use the full number of processes in order to get a more symmetric decomposition (cf. Figure 3). This is achieved by a subcommunicator of MPI_COMM_WORLD, which we will call global_communicator in the following.

The following relationship holds:

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

so that parallel reduction to distributed $\Omega_{\vec{x}}$ -space and to distributed $\Omega_{\vec{v}}$ -space becomes a collective communication of subsets of processes. This is important in mathematical operations like $\int d\Omega_{\vec{x}}$ and $\int d\Omega_{\vec{v}}$, hence we make the subsets of processes available via the MPI communicators column_comm and row_comm similarly as in distributed matrix-matrix multiplication implementations [43].

We enumerate cells within a subdomain lexicographically to get a global numbering, as depicted in Figure 1. This enables us to determine a globally unique cell ID of locally owned cells and of ghost cells by querying the low-dimensional triangulation cells for their IDs and ranks without the need for communication.

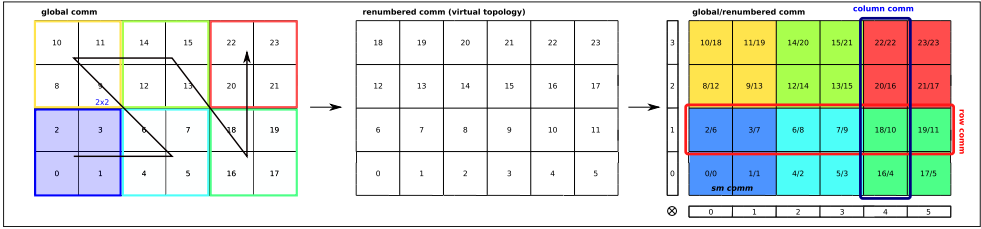


Fig. 4. Renumbering of ranks in the global communicator (via `MPI_Comm_split`) for a hypothetical setup of 26 ranks in `MPI_COMM_WORLD` and a 6×4 partition, processes are grouped in 2×2 blocks, which are ordered along a z-curve. Based on this new global communicator, the partitioning, as described in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

Placing ranks according to $\lfloor r(i, j)/p_{node} \rfloor$ onto the same compute node (with p_{node} being the number of processes per node) leads to striped partitioning (see the colors of the blocks in Figure 3). This results in a bad shape of the union of subpartitions belonging to the same compute node, leading to decreased benefit of shared memory. In order to improve the placing of subpartitions onto the compute nodes without having to change the function r , we block within a Cartesian virtual topology (Figure 4), e.g., by 48 process blocks with 8 processes in \vec{x} -space and with 6 processes in \vec{v} -space. Compute nodes are ordered along a z-curve.

As a final remark, it should be emphasized that the presented partitioning approach delivers good results regarding communication amount and communication pattern if the low-dimensional triangulations $\mathcal{T}_{\vec{x}}$ and $\mathcal{T}_{\vec{v}}$ have been partitioned well by the underlying low-dimensional library. Depending on the given mesh, a space-filling curve approach [7, 10, 44] or a graph-based approach [20, 23] might be beneficial for this. Nevertheless, it should be noted that, generally, PDEs with surface data exchange become increasingly heavier on communication as the dimension increases, including a significant amount of inter-node communication even for an optimal communication layout.

3.3 Elements, DoFs, Quadrature, and Mapping

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

$$\mathcal{P}_k^d = \underbrace{\mathcal{P}_k^1 \otimes \dots \otimes \mathcal{P}_k^1}_{\times d}. \quad (11)$$

The unknowns are discontinuous across cells with $(k+1)^d$ unknowns per cell for a scalar field f . The total number of DoFs for $|C|$ cells is

$$N = |C| \cdot (k+1)^d. \quad (12)$$

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

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

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

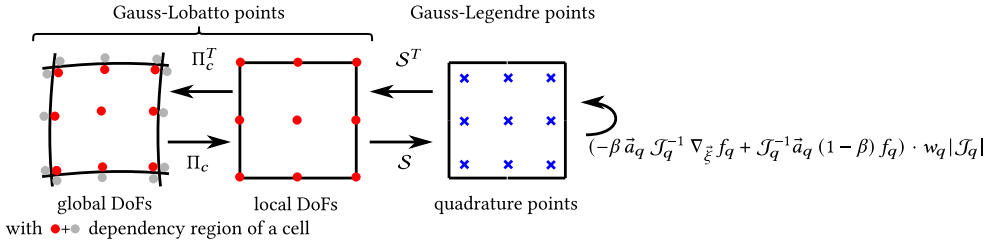


Fig. 5. Visualization of the five steps of a matrix-free cell-integral evaluation for polynomial degree $k = 2$ and number of quadrature points $n_q = 3$.

Table 3. Comparison of the Memory Consumption (in Doubles) of the Mapping Data (Per Quadrature Point) Due to the Jacobian Matrices if the Phase-space Structure is Exploited ($J_{\vec{x}}$ and $J_{\vec{v}}$) and if the Phase-space Structure is not Exploited (J)

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

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

$$Q_{n_q}^d = \underbrace{Q_{n_q}^1 \otimes \dots \otimes Q_{n_q}^1}_{\times d}, \tag{14}$$

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

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

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

3.4 Matrix-free Operator Evaluation

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

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

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

The five steps are visualized in Figure 5.

The most efficient implementations of the basis change (from Gauss–Lobatto to Gauss–Legendre points) perform a sequence of d 1D interpolation sweeps, utilizing the tensor-product form of the shape functions in an even-odd decomposition fashion with $(3 + 2 \cdot \lfloor ((k-1) \cdot (k+1)/2) \rfloor / (k-1))$ FLOPs/DoF (for $k+1 = n_q$) [30]. This operation is known as sum factorization and has its origin in the spectral-element community [13, 34, 36]. Similarly, the testing with the gradient of the shape functions can be performed efficiently with $2d$ sweeps [30].

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

Algorithm 3 shows the pseudocode of a possible matrix-free merged advection and inverse-mass-matrix operator evaluation in the context of ECL. Lines 3, 7, 9, 13, 15, and 19 are evaluated with sum factorization. To reduce the working set, we do not compute all $(d_{\vec{x}} + d_{\vec{v}})$ -derivatives at once, but first we compute the contributions from \vec{x} -space and then the contributions from \vec{v} -space. One could further reduce the working set by loop blocking [30].

Since we loop over cell-cell and cell-face pairs, mapping information of the cells $c_{\vec{x}}$ and $c_{\vec{v}}$ as well as of the faces $f_{\vec{x}}$ and $f_{\vec{v}}$ can be queried from the low-dimensional library independently and combined on the fly. The separate cell IDs $c_{\vec{x}}$ and $c_{\vec{v}}$ only have to be combined when accessing the solution vector, which is the only data structure set up for the whole high-dimensional space.

3.5 Implementation of Operator Evaluations with `hyper.deal`

The library `hyper.deal` provides classes that are built around `deal.II` classes and contain inter alia utility functions needed in Algorithm 3. To enable a smooth start for users already familiar with `deal.II`, we have chosen the same class and function names living in the namespace `hyper.deal`. The relationship between classes in `hyper.deal` and classes in `deal.II` is visualized in the UML diagram in Figure 6. The class `hyperdeal::MatrixFree` is responsible for looping over cells (and faces) as well as for storing precomputed information related to shape functions and precomputed quantities at the quadrature points. The classes `hyperdeal::FEEvaluation` and `hyperdeal::FEFaceEvaluation` (not shown) include functions to read and write cell-/face-local values from a global vector as well as operations at the quadrature points. As an example, Figure 6 shows the implementation of the `hyperdeal::FEEvaluation::submit_gradient()` method, which uses two instances of the `deal.II` class with the same name, one for \vec{x} - and one for \vec{v} -space, for the evaluation of $f(\vec{u}) = \mathcal{J}_{c,q}^{-1} |\mathcal{J}_q| w_q \vec{u}$ for $\vec{u} \in \mathbb{R}^d$.

ALGORITHM 3: Element-centric loop for arbitrary operators and DG integration of a cell batch for advection operator evaluation for vectorization over elements

```

/* loop over all cell pairs */
1 foreach  $c := (c_{\vec{x}}, c_{\vec{v}}) \in C_{\vec{x}} \times C_{\vec{v}}$  do
  /* step 1: gather values (Array of structs (AoS)  $\rightarrow$  struct of arrays (SoA)) */
2   - gather local vector values  $u_i^{(c)}$  on the cell from global input vector  $\vec{u}$ 
  /* step 2: apply advection cell contributions */
3   - interpolate local vector values  $\vec{u}^{(c)}$  onto quadrature points,  $u_h^c(\vec{\xi}_q) = \sum_i \phi_i u_i^{(c)}$  and compute gradients
   $\nabla_{\vec{x}} u_h^c(\vec{\xi}_q)$  in reference coordinate system
4   foreach quadrature index  $q = (q_{\vec{x}}, q_{\vec{v}})$  do
5     - compute convective-term contribution
      
$$\vec{b}_i = -\beta \vec{a}_{\vec{x}}(\hat{x}^{(c_{\vec{x}})}(\vec{\xi}_{q_{\vec{x}}}), \hat{x}^{(c_{\vec{v}})}(\vec{\xi}_{q_{\vec{v}}})) \mathcal{J}_{(c_{\vec{x}})}^{-1} \nabla_{\vec{\xi}} u_h^{(c)}(\vec{\xi}_q) \underbrace{|\mathcal{J}_{q_{\vec{x}}}| |\mathcal{J}_{q_{\vec{v}}}| w_{q_{\vec{x}}} w_{q_{\vec{v}}}}_{"|\mathcal{J}_{(q)}| w_q"}$$

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

7     - evaluate local integrals by quadrature  $b_i = b_i + (\nabla_{\vec{x}} \phi_i^{c_o}, \vec{c}_{\vec{x}} u_h^{(c)})_{\Omega(e)} \approx b_i + \sum_q \nabla_{\vec{x}} \phi_i^{c_o}(\vec{\xi}_q) \cdot \vec{t}_q$ 
8   end
9   repeat lines 3-8 for  $v$  space
  /* step 3: apply advection face contributions (loop over all 2d faces of  $\Omega_c$ ) */
10  foreach  $f \in \mathcal{F}_{(c)}$  do
11    - interpolate values from cell array  $\vec{u}^{(c)}$  to quadrature points of face
12    - if interior face, gather values from neighbor  $\Omega_{e^+}$  of current face
13    - interpolate  $u^+$  onto face quadrature points
14    - compute numerical flux and multiply by quadrature weights; /* not shown here */
15    - evaluate local integrals related to cell  $c$  by quadrature and add into cell contribution  $b_i$ 
16  end
  /* step 4: apply inverse mass matrix */
17  foreach quadrature index  $q = (q_{\vec{x}}, q_{\vec{v}})$  do
18    - prepare integrand at each quadrature point by computing  $t_q = b_i w_{q_{\vec{x}}}^{-1} w_{q_{\vec{v}}}^{-1}$ 
19    - transformation from collocation space to the Gauss-Lobatto space used for vector storage:
      
$$y_i^{(c)} = \sum_q \tilde{\phi}_{iq} \cdot \vec{t}_q \text{ with } \tilde{\phi}_{iq} = \mathcal{V}_{iq}^{-1} \text{ with } \mathcal{V}_{iq} = \phi_i(\vec{\xi}_q)$$

20  end
  /* step 5: scatter values (SoA  $\rightarrow$  AoS) */
21  - set all contributions of cell,  $\vec{y}^{(c)}$ , into global result vector  $\vec{y}$ 
22 end

```

We process a batch of v_{len} cells or faces in a “vectorization over elements” fashion. We do not operate directly on the primitive types double/float but on structs built around intrinsic instructions¹, with each vector lane dedicated to a separate cell of mesh. The maximal number of vector lanes depends on the given hardware; with AVX-512 instruction-set extension, as most modern Intel-based processors have, 8 doubles (i.e., 8 cells in the context of our application) can be processed by a single instruction.

Currently, we vectorize only over elements in \vec{x} -space, whereas the \vec{v} -space is not vectorized. The reason is that the Vlasov–Maxwell and the Vlasov–Poisson models contain heavy operations

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

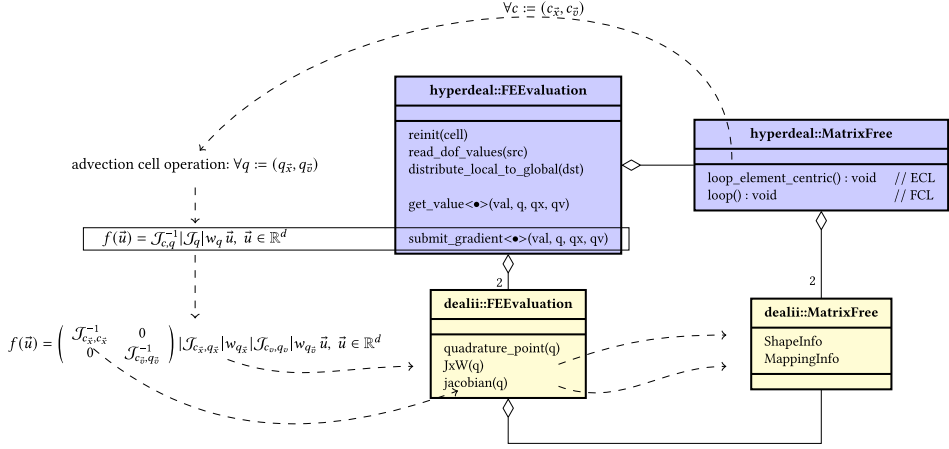


Fig. 6. Class diagram of a part of the matrix-free infrastructure of `hyper.deal`. It presents how classes from `hyper.deal` (namespace `hyperdeal`—highlighted in blue) and from `deal.II` (namespace `dealii`—highlighted in yellow) relate to each other. Only the `hyper.deal` methods are shown that are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

on the full phase space and on the \vec{x} -space, respectively, which benefit from vectorization over \vec{x} . As a consequence, the data structures are already laid out correctly for an efficient matrix-free solution of the lower-dimensional problem.

3.6 Requirements to a Low-dimensional Finite-element Library

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

- (1) The low-dimensional triangulation \mathcal{T} can be partitioned among a user-defined group of processes. Besides locally owned cells \mathcal{L}_i , each process needs a halo of ghost cells.
- (2) 1D scalar (discontinuous) Lagrange shape functions \mathcal{P}_k^1 of polynomial degree k , based on Gauss–Lobatto support points, and 1D quadrature rules with n_q points are accessible. Dimension-independent interpolation kernels based on sum factorization are available.
- (3) The library has access to the mapping data, like \mathcal{J}^{-T} and $|\mathcal{J}| \times w$ both on cells and faces as well as \vec{n} on faces. It is not relevant whether these quantities are pre-computed or recomputed on the fly.
- (4) It is beneficial if the library has the option to work on a batch of cells where the size of the batch can be set arbitrarily. This implies that the needed data is already provided in a vectorized fashion, which allows to skip an additional reshuffling step.

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

4 PARALLELIZATION BY SHARED-MEMORY MPI

The parallelization of the library `hyper.deal` is purely MPI-based. MPI allows to program for distributed systems, which is crucial for solving high-dimensional problems due to their immense memory requirements. A downside of a purely MPI-based code with one rank used per core is that many data structures are created and updated multiple times on the same compute node, although they could be shared. In FEM codes, it is widespread that ghost values filled by standard `MPI_(I)Send/MPI_(I)Recv` reside in an additional section of the solution vector [29]. Depending on the MPI implementation, these operations will be replaced by efficient alternatives, avoiding additional copying and unexpected message buffers if the calling peers are on the same compute node. Nevertheless, the allocation of additional memory, if main memory is scarce, might be unacceptable.

Adding shared-memory libraries, like TBB and OpenMP, to an existing MPI program would allow to use shared memory; however, with the downside of manually annotating and parallelizing all relevant loops. We propose a different approach to exploit shared memory. It is based on the observation that the major time and memory benefit of using shared memory in a purely MPI-parallelized FEM application comes from accessing the part of the solution vector owned by the processes on the same compute node without the need to make explicit copies and buffering them [30, 31]. This is why we propose a new vector class that uses `MPI-3.0` features to allocate shared memory and provides controlled access to it with an otherwise unchanged vector interface.

4.1 Shared-memory Vector

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

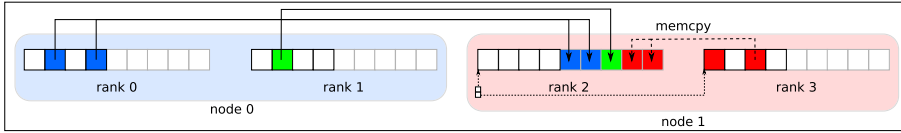
These functions form the basis of the new shared-memory modus of the vector class `dealii::LinearAlgebra::distributed::Vector` and provide memory for the locally-owned unknowns and for unknowns of ghost faces that are not owned by any process on the same compute node. Furthermore, pointers to the arrays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

A natural way to access the solution vector is by specifying vector entry indices and the cell ID for DoFs owned by a cell or by specifying a pair of a cell ID and a face number ($< 2d$) for DoF owned by faces.

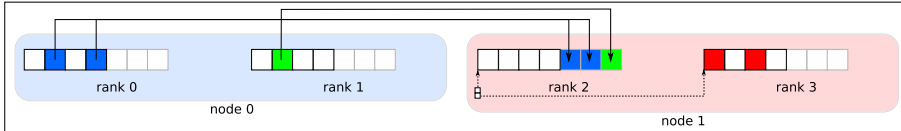
An interpretation layer also provides access to the values of the DoFs of the local and the ghost cells: it returns pointers either to buffers or to the shared memory, depending on the cell type (shared or remote). In this way, the user of the vector class gets the illusion of a pure MPI program, since the new vector has to be added at a single place and only a few functions querying values from the vector (e.g., `FEEvaluation::read_dof_values()` and `::distribute_local_to_global()` in Figure 6) oblivious to the user have to be specialized.

We provide two operation modes:

- In the buffered mode (see Figure 7(a)), memory is allocated also for ghost values owned by the same compute node; these ghost values are updated directly via `memcpy` without an intermediate step via MPI. This mode is necessary if ghost values are modified, as it takes place in FCLs. It promises some performance benefit, since data packing/unpacking can be skipped.



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



(b) A hybrid approach using MPI-3.0 shared-memory features (non-buffered mode) similar to 7a, with the difference that only ghost values that live in different shared-memory domains are updated. Ghost values living in the same shared-memory domain are directly accessed only when needed. The similarity to a thread-based implementation is clear with the differences that vectors are non-contiguous, requiring an indirect access to values owned by other processes, and that each process may manage its own ghost values and send/receive its own medium-sized messages needed to fully utilize the network controller.

Fig. 7. Two hybrid ghost-value-update approaches for a hypothetical setup with two nodes, each with two cores. Only the communication pattern of rank 2 is considered.

- The non-buffered mode (see Figure 7(b)) does not allocate any redundant memory for ghost values owned by the same compute node. This mode works perfectly with ECL, since it is by design free of race conditions.

4.2 Overlapping Communication and Computation

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

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

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

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

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

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

5 PERFORMANCE ANALYSIS FOR HIGH-DIMENSIONAL SCALAR TRANSPORT

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

5.1 Experimental Setup and Performance Metrics

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

The measurement data have been gathered either with user-defined timers or with the help of the script `likwid-mpirun` from the `LIKWID` suite and with suitable in-code `LIKWID` API annotations [38, 41]. The following metrics are used to quantify the quality of the implementations:

- **throughput**: processed DoFs per time unit

$$\text{throughput} = \frac{\text{processed DoFs}}{\text{time}} \stackrel{\text{Equation (12)}}{=} \frac{|C| \cdot (k+1)^d}{\text{time}} \quad (15)$$

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

- **performance**: maximum number of floating-point operations per second;
- **data volume**: the amount of data transferred within the memory hierarchy (we consider the transfer between the L1-, L2-, and L3-caches as well as the main memory); and
- **bandwidth**: data volume transferred between the levels in the memory hierarchy per time.

Our main objective is to decrease the time-to-solution, which, for a fixed discretization, corresponds to increasing the throughput. The measured quantities “performance,” “data volume,” and “bandwidth” are useful, since they show how well the given hardware is utilized and how much additional work or memory transfer is performed compared to the theoretical requirements of the mathematical algorithm.

High-order and high-dimensional problems have a large working set $(k+1)^d$. The evaluation of this expression for $2 \leq k \leq 5$ and $2 \leq d \leq 6$ is presented in Table 4. The k - d configurations with working-set size of $v_{len} \cdot (k+1)^d < L_1$, fitting into the L1 cache, are expected to show good performance; the k - d configurations with working-set size of $L_1 \leq v_{len} \cdot (k+1)^d < L_2$ are

Table 4. DoFs Per Cell: $(k + 1)^d$

| k | 2D | 3D | 4D | 5D | 6D |
|-----|-----------|------------|--------------|--------------|---------------|
| 2 | 9 | 27 | 81 | 243 | 729 |
| 3 | <i>16</i> | <i>64</i> | <i>256</i> | 1,024 | 4,096 |
| 4 | <i>25</i> | <i>125</i> | 625 | 3,125 | 15,625 |
| 5 | <i>36</i> | <i>216</i> | 1,296 | 7,776 | 46,656 |

The k - d configurations with working-set size $v_{len} \cdot (k + 1)^d < L_1$ are highlighted in italics and configurations with working-set size $L_1 \leq v_{len} \cdot (k + 1)^d < L_2$ in bold.

Table 5. Specification of the Hardware System used for Evaluation with Turbo Mode Enabled

| Intel Skylake Xeon Platinum 8174 | |
|--|--|
| Cores | 2 × 24 |
| Frequency base (max AVX-512 frequency) | 2.7 GHz (2.7 GHz) |
| SIMD width | 512 bit |
| Arithmetic peak (dgemm performance) | 4147 GFLOP/s (3318 GFLOP/s) |
| Memory interface | DDR4-2666, 12 channels |
| STREAM memory bandwidth | 205 GB/s |
| Empirical machine balance | 14.3 FLOP/Byte |
| L1-/L2-/L3-/MEM size | 32kB/1MB/66MB (shared)/96GB (shared) |
| compiler + compiler flags | g++, version 9.1.0, -O3 -funroll-loops -march=skylake-avx512 |

Memory bandwidth is according to the STREAM triad benchmark (optimized variant without read for ownership transfer involving two reads and one write), and GFLOP/s are based on the theoretical maximum at the AVX-512 frequency. The dgemm performance is measured for $m = n = k = 12,000$ with Intel MKL 18.0.2. We measured a frequency of 2.5 GHz with AVX-512 dense code for the current experiments. The empirical machine balance is computed as the ratio of measured dgemm performance and STREAM bandwidth from RAM memory.

expected to be performance-critical, since each sum-factorization sweep might drop out of the cache. The latter configurations are, however, the most relevant with regard to high-order and high-dimensional problems.

All performance measurements have been conducted on the SuperMUC-NG supercomputer. Its compute nodes have 2 sockets (each with 24 cores of Intel Xeon Skylake) and the AVX-512 ISA extension so that 8 doubles can be processed per instruction. A detailed specification of the hardware is given in Table 5. The parallel network is organized into islands of 792 compute nodes each. The maximum network bandwidth per node within an island is 100 GBit/s = 12.5 GB/s² due to the fat-tree network topology. Islands are connected via a pruned-tree network architecture (pruning factor 1:4).

The library hyper .deal has been configured in the following way: all processes of a node are grouped into blocks of the size of $48 = 8 \times 6$. All processes in these blocks share their values via the shared-memory vector. The cells in low-dimensional triangulations are enumerated along a Morton curve, which is equally distributed among the processes. We use the highest ISA extension AVX-512 so that eight cells are processed at once. The Jacobian matrices and their determinants are precomputed for \vec{x} - and \vec{v} -space and combined on the fly. The quadrature is based on the Gauss-Legendre formula with $n_q = k + 1$. The skew factor β is set to 0.5 such that the gradients of the solution have to be computed at the cell quadrature points and the values have to be tested by the gradient of the test functions. In the following, we refer to this configuration as “default configuration.”

²<https://doku.lrz.de/display/PUBLIC/SuperMUC-NG>.

5.2 Cell-local Computation

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

5.2.1 Cell Integrals. We first consider all steps in Algorithm 3 related to the cell integrals (lines 2–9 and 17–21), skipping the loops over faces and ignoring the flux computation.

During the cell integrals, values are read from the global vector, a basis change to the Gauss–Legendre quadrature points is performed (with d data sweeps for reading and d for writing), the gradients at the quadrature points are computed ($2d$), the gradients obtained are multiplied with the velocities (d), the values obtained are multiplied with the velocities at the quadrature points ($2d$) and tested by the value and the gradient of the collocation functions ($3d$), the inverse mass matrix is applied ($2d$), and finally the results are written back to the global vector. A total of $12d$ data sweeps are necessary if reading and writing are counted separately. The working set of sum factorization is $v_{len} \cdot (k+1)^{d_1+d_2}$, and the working set of the intermediate values is $v_{len} \cdot \max(d_1, d_2) \cdot (k+1)^{d_1+d_2}$. A comparison with hardware statistics shows that the working set of sum factorization exceeds the size of the L1 cache for configurations $k = 3 / d = 5$ and $k = 5 / d = 4$ so that every data sweep has to fetch the data from the L2 cache.

The theoretical considerations made above are supported by the measurement results in Figure 8, which shows the data traffic between the memory hierarchy levels (data volume per DoF and bandwidth), the floating-point operations per DoF, and the throughput for $k = 3$ and $k = 5$ for $2 \leq d \leq 6$.

In Figure 8(a), one can observe that with increasing dimension the data traffic between the memory hierarchy levels increases as the data volume and the corresponding bandwidth increase. Beginning from the configurations mentioned above ($k = 3 / d = 5$ and $k = 5 / d = 4$), the data have to be fetched from and written back to the L2 cache again during every sweep, resulting in a data volume traffic between the L1 and L2 caches that linearly increases with the number of sweeps. The constant offset of 22 double/DoF is mainly related to the access to the global solution vector and to the mapping data. However, data have to be loaded from the L2 cache also for smaller working-set sizes than the ones of the k - d configurations mentioned above; the main reason for this is that the intermediate values do not fit into the cache any more.

For even higher dimensions, the L3 cache and the main memory have to be accessed during the sweeps. While this operation is negligible for $k = 3$ (see Figure 8(d)), it is performance-limiting in the case of $k = 5$: For $k = 5 / d = 5$, the bandwidth to the L1 cache is limited by the access to the L2 cache (see Figure 8(b)); for $k = 5 / d = 6$, it is even limited by the main memory. In the latter case, the caches are hardly utilized any more and the data have to be fetched from/written back to main memory during every sweep, leading to a bandwidth close to the values measured for the STREAM benchmark. This comes along with a significant performance drop.

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

5.2.2 Local Cell and Face Integrals. In this subsection, we consider all computation steps in Algorithm 3, but ignore the data access to neighboring cells (line 12). This means that face values from neighboring cells are not gathered and face buffers for exterior values are left unchanged. In this way, we are able to demonstrate the effects of increased working sets (of both face buffers) and of the increased number of sweeps. Additional sweeps have to be performed for interpolating values

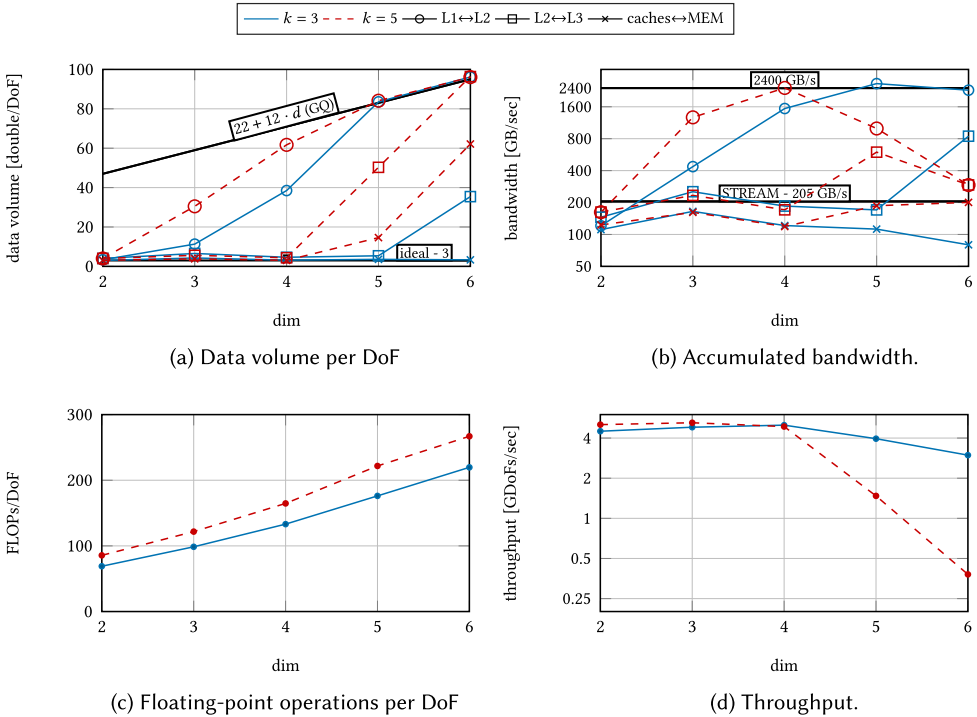


Fig. 8. Node-level analysis of the cell integrals of the advection operator in terms of data transfer, bandwidth, arithmetic operations, and throughput in two to six dimensions.

from the cell quadrature points to the quadrature points of the $2d$ faces as well as for interpolating the values of the neighboring cells onto the quadrature points during the flux computation.

Figure 9 shows the data traffic between the memory hierarchy levels (data volume per DoF and bandwidth), the floating-point operations per DoF, and the throughput for $k = 3$ and $k = 5$ for $2 \leq d \leq 6$. In comparison to the results of the experiments that only consider the cell integrals in Figure 8, the following observations can be made: as expected, the data volume transferred between the cache levels (Figure 8(a) and 9(a)) and the number of floating-point operations approximately double (Figure 8(c) and 9(c)). However, the configurations at which the traffic to the next cache level increases have not changed, indicating that the increase in working set is not limiting the performance here.

The doubling of the data volume to be transferred for $k = 5$ and high dimensions naturally leads to half the throughput (see Figure 9(d)). In the case of $k = 3$, we can also observe a drop of performance in high dimensions. Given that the memory transfer between the L1 and L2 caches reaches 2,400 GB/s or around 22 bytes/cycle, we suspect that the data transfer between the L1 and L2 caches is the main limit in this case. The memory transfer between the L2 and L3 caches is about 550GB/s. This value is significantly less than that is observed for the cell-integral-only run, resulting in the drop of the overall performance by 40% for high dimensions.

5.3 Full Advection Operator

This subsection considers the application of the full advection operator, as shown in Algorithm 3, including the access to neighboring cells during the computation of the numerical flux. Figure 10

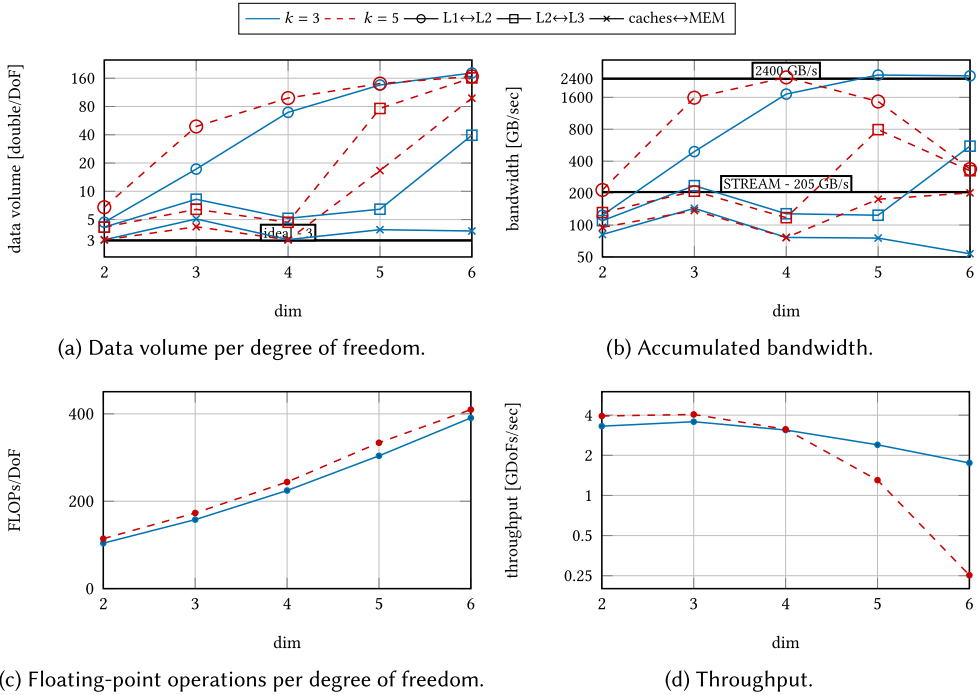


Fig. 9. Node-level analysis of the evaluation of the cell and face intergrals of the advection operator, ignoring loads from neighboring cells.

presents the results of parameter studies of the dimension $2 \leq d \leq 6$ for different polynomial degrees $2 \leq k \leq 5$.

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

The following observations can be made in Figure 10: for working sets that fit into the L1 cache, a higher polynomial degree leads to a higher throughput. For working sets exceeding the size of the L1 cache and only fitting into the L2 cache, the throughput drops. In this region, the curves overlap so that we conclude that the throughput is indeed a function of the working-set size $\approx (k + 1)^d$ and independent of the polynomial degree k and the dimension d individually.

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

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

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

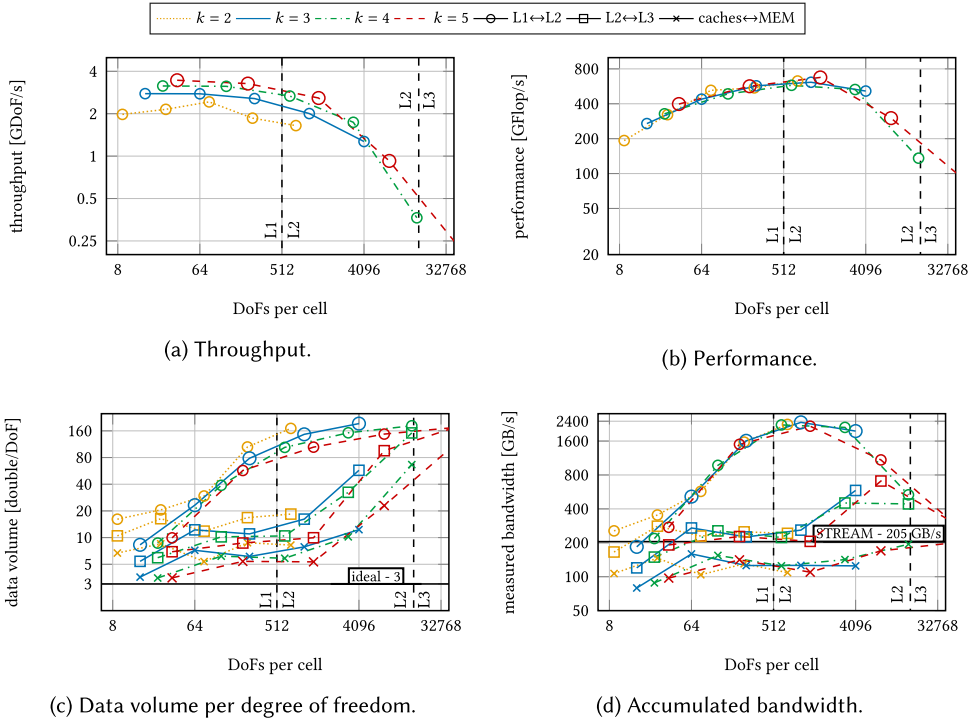


Fig. 10. Node-level analysis of the evaluation of the full advection operator in terms of data transfer, bandwidth, arithmetic operations, and throughput in two to six dimensions. Dashed vertical lines show the limits of the L1 and L2 caches for a hypothetical fully associative cache with optimal replacement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

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

We point out that by selecting the skew-symmetric parameter $\beta = 0.5$, we need to compute both the values and the gradients at the quadrature points as well as to test by the value and the gradient of the test functions. Instead, by using a conservative formulation ($\beta = 0.0$) or a convective formulation ($\beta = 1.0$), it is possible to skip either the computation of the gradients or testing by the gradients, which leads to fewer sum-factorization sweeps and potentially decreases the cache misses. Experiments indeed confirm this and show a speed-up of approximately 14% for both $\beta = 0.0$ and $\beta = 1.0$.

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

Table 6. Relative Performance Due to Flux Computation (Ratio Fig. 9–8)

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

Table 7. Relative Performance Due to Access to the Values of Neighboring Cells (Ratio Fig. 10–9)

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

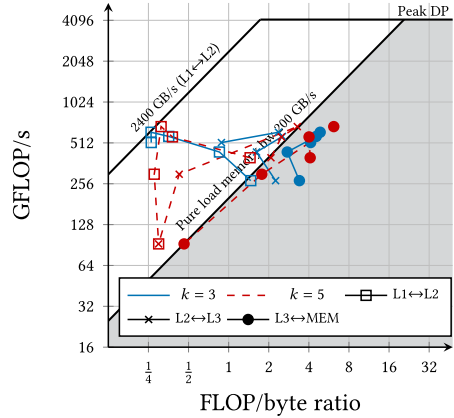


Fig. 11. Roofline model of the application of the full advection operator for $k = 3/5$.

Table 8. Evaluation of the Shared-memory Vector for Different Configurations on a Single Compute Node with 48 Processes

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

In the simulation, there were accumulated 2.1G locally-owned DoFs and 1.6G ghost DoFs for a 6D problem with $k = 3$. Communication and computation has been overlapped in all cases.

5.4 Alternative Implementations

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

The library `hyper.deal` has been developed to be able to compute efficiently on complex geometries both in geometric and velocity space. The upper limit of the performance of the tensor-product approach is given by the consideration of the tensor product of two Cartesian grids, which leads to the same constant diagonal Jacobian matrix at all quadrature points. As a lower limit, one can consider the case that each quadrature point has a unique Jacobian of size $d \times d$. Figure 12(a) shows that the behavior of the default tensor-product setup is similar to that of a pure Cartesian grid simulation with only a small averaged performance penalty of approximately 13%. This observation matches our expectations expressed in Subsection 3.3 and means that in high dimensions, the evaluation of curved meshes in the tensor-product factors is essentially for free, compared to storing the full Jacobian matrices.

Figure 12(b) compares ECL with FCL and shows the clear advantage of the former. We have neither implemented any advanced blocking schemes for ECL or FCL nor are we processing cell and face integrals in alternating order, as it is done in `deal.II` [30], to potentially increase cache efficiency. The fact that ECL still shows a better performance demonstrates the natural cache-friendly property of ECL. The benefit of ECL decreases for high dimensions and high polynomial degrees due to the increased number of sweeps, which is related to the repeated evaluation of the flux terms. Nevertheless, we propose to use ECL for high-dimensional problems because of its suitability for shared-memory computations that reduce the allocated memory.

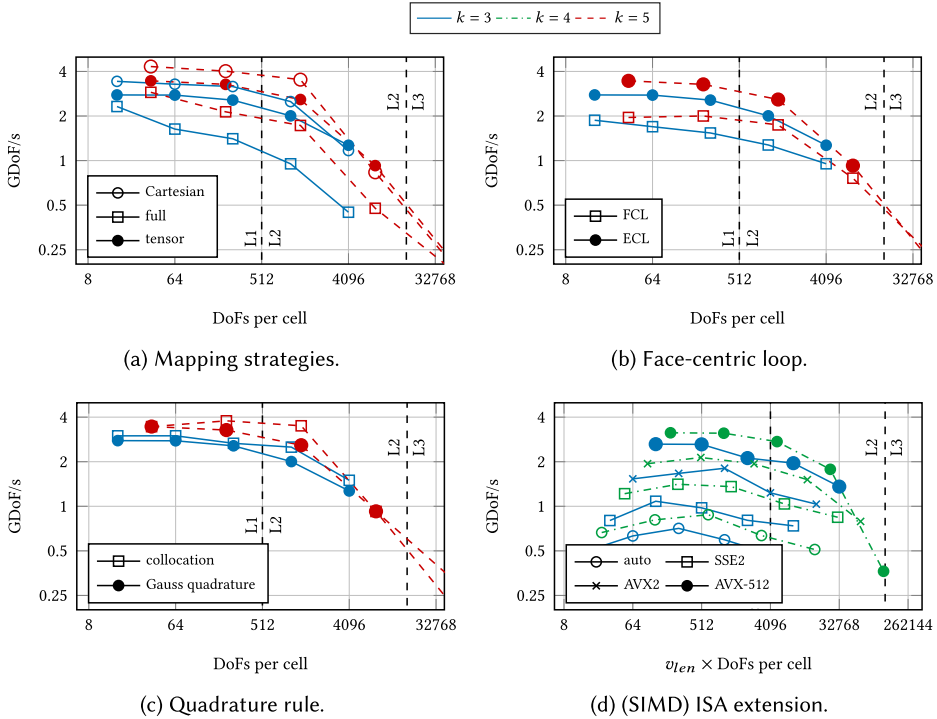


Fig. 12. Node-level analysis of the application of the full advection operator: (a)–(c) comparison of the performance of different algorithms and the default configuration for $k = 3$ and $k = 5$; (d) comparison of the performance of different (SIMD) ISA extensions and the auto-vectorization for $k = 3$ and $k = 4$.

We favor the Gauss–Legendre quadrature method over the collocation methods due to its higher numerical accuracy. This benefit comes at the price of a basis change from the Gauss–Lobatto points to the Gauss quadrature points and vice versa. Figure 12(c) shows a performance drop of 11% on average due to these basis changes as long as the data that should be interpolated remains in the cache.

In the library `hyper.deal`, we currently only support “vectorization over elements.” As a default, the highest instruction-set extension is selected, i.e., the maximum number of cells is processed at once by a core. Since the number of lanes to be used is templated, the user can reduce the number of elements that are processed at once, as it is demonstrated in Figure 12(d). It can be observed that, in general, the usage of higher instruction-set extensions leads to a better throughput. However, once the working set of a cell batch exceeds the size of the cache, a performance drop can be observed. The performance drop leads to the fact that in 6D with cubic elements the throughputs of AVX-512 and of AVX2 are comparable and in 6D with quartic elements SSE2 and AVX2 show the best performance.

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

5.5 Strong and Weak Scaling

In this subsection, we examine the parallel efficiency of the library `hyper.deal`. For this study, we consider the advection operator embedded into a low-storage Runge–Kutta scheme of order 4

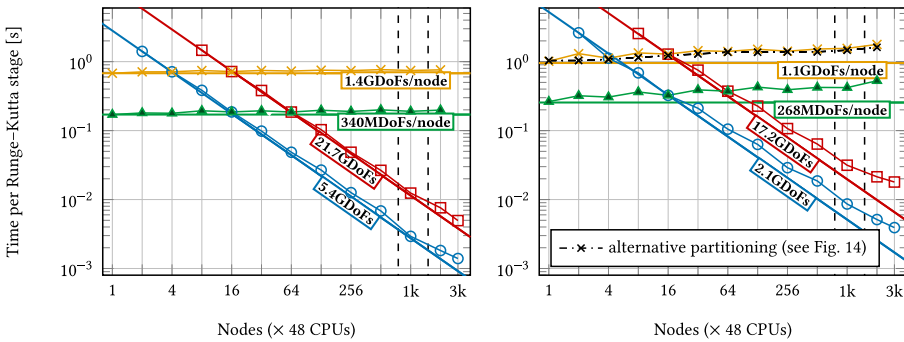
Table 9. Strong and Weak Scaling Configurations

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

left, $d = 4/k = 5$; right, $d = 6/k = 3$.

Table 10. Partitioning the \vec{x} - and \vec{v} -triangulations on up to 3,072 Nodes with 48 Cores

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



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

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

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

with 5 stages, which uses two auxiliary vectors besides the solution vector [25]. From these three vectors, only one (auxiliary) vector has ghost values.

Figure 13 shows strong and weak scaling results of runs on SuperMUC-NG with up to 3,072 nodes with a total of 147,456 cores. We consider two configurations: “ $d = 4 / k = 5$,” an easy configuration, and “ $d = 6 / k = 3$,” a demanding configuration. As examples, we present for each configuration two strong and two weak scaling curves (see Table 9). Table 10 shows the considered process decomposition $p = p_{\vec{x}} \cdot p_{\vec{v}}$.

For the “ $d = 4 / k = 5$ ” configuration, we observe excellent weak-scaling behavior with parallel efficiencies of 89% and 86% for up to 2,048 nodes on the large and the small setup, respectively. We get more than 75/80% efficiency for strong scaling up to the increase in the number of nodes by a factor of 256. For the “ $d = 6 / k = 3$ ” configuration, we see parallel efficiencies of 49/57% for weak scaling. These values are lower than the ones in the 4D case; however, they are still very good in the light of the immense communication amount in the 6D case: As shown in Figure 15, the ghost data to remote nodes amounts to 29% of the solution vector in 6D and only 5% in 4D.

Finally, we analyze the drop in parallel efficiency of the weak-scaling runs of the 6D large-scale simulations. For this, we have slightly modified the setup: we start from a configuration of 8^6 cells with $k = 3$ on one node (32 DoFs in each direction and a total number of 1.1 GDoFs). When

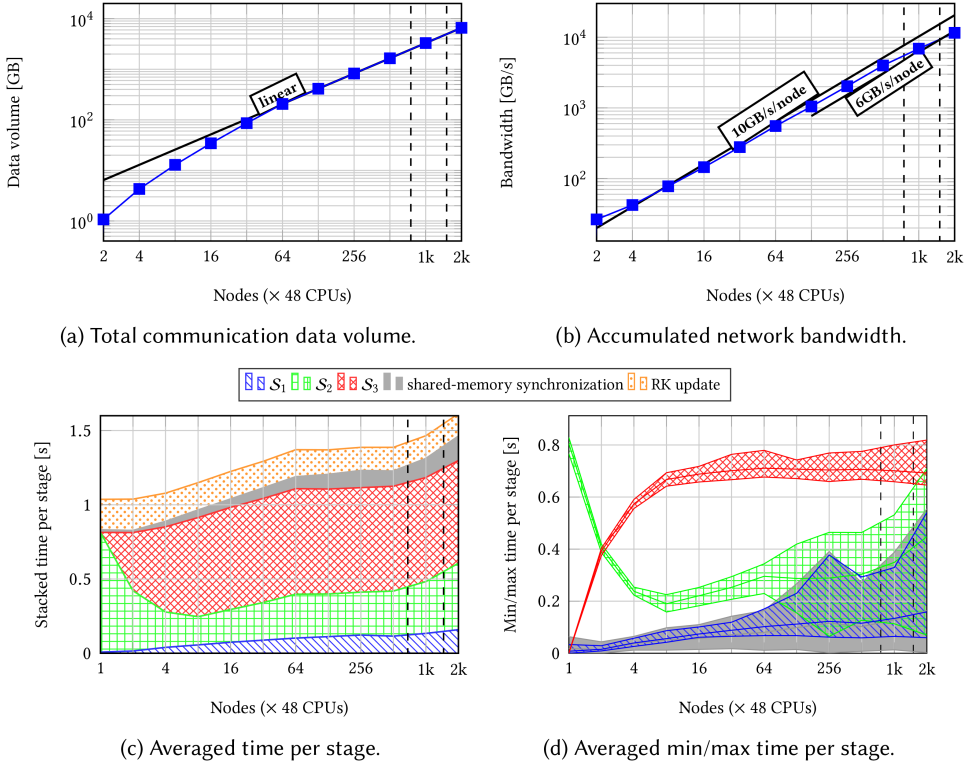


Fig. 14. Details on the weak scaling of a single Runge–Kutta stage of the solution of the advection equation.

doubling the number of processes, we double the number of cells in one direction, starting from direction 1 to direction 6 (and starting over at direction 1). Each time, we double the number of cells along a direction, we also double the number of processes in that direction, keeping the number of processes in the other direction constant. In this way, the number of DoFs per process along \vec{x} and \vec{v} as well as the number of ghost DoFs per process remain constant once all cells at the boundary have periodic neighbors residing on other nodes (number of nodes $\geq 2^{d_x+d_v}$). As a consequence, the computational work load and the communication amount of each node are constant.

The total communication amount of the considered setup increases linearly with the number of processes, as presented in Figure 14(a). Measurements in Figure 14(b) show that the network can handle this increase: the data can be sent with a constant network bandwidth of 10 GB/s per node, which is close to the theoretical 100 Gbit/s as long as the job stays on an island due to the fat-tree network topology. Once the job stretches over multiple islands due to the pruned-tree network architecture, we observe a bandwidth of 6 GB/s, which is related to the fact that only a small ratio of the messages crosses island boundaries.

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

- (1) Start the shared-memory communication and the remote communication by calling `MPI_Irecv` as well as pack and send (via `MPI_Isend`) messages to each neighboring process

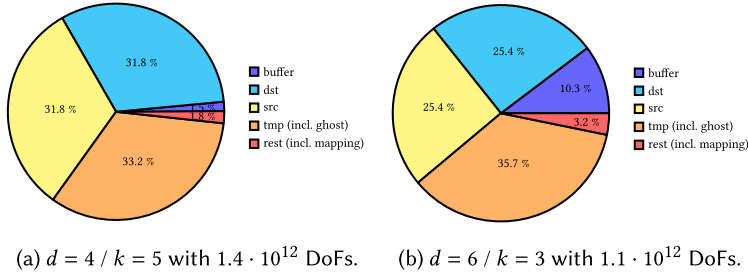


Fig. 15. Approximate memory-consumption distribution of high-dimensional advection application for 48×1024 cores.

residing on remote compute nodes. Furthermore, process S_1 in the overlap of the communication and the computation strategy.

- (2) Finish shared-memory communication and process S_2 .
- (3) Finish the remote ghost-value update by waiting (with `MPI_Waitall`) until all messages have been sent and received as well as process S_3 .
- (4) Synchronize the shared-memory processes, which is needed to prevent race conditions due to the reuse of the source vector during the subsequent Runge–Kutta update steps.
- (5) Perform the remaining Runge–Kutta update steps.

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

On a single node, most of the time is spent for S_2 , since all data are available in the same shared-memory domain. As the number of nodes increases, the time spent for S_3 needing more data from remote processes increases. The time spent for S_1 is comparatively small, which can be attributed to the fact that in 6D nearly all cells have neighbors owned by other processes.

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

The minimum and the maximum time spent for each step differ significantly, which can be attributed to the non-trivial communication pattern and the varying size of S_2 and S_3 . Overall, however, this imbalance caused by the MPI communication is not performance-hindering, as has been demonstrated by the fact that a significant portion of the network bandwidth is used. However, one should keep this imbalance in mind and not attribute it accidentally to other sections of the code.

5.6 Memory Consumption

Figure 15(b) shows the approximated memory consumption for a large-scale simulation from Subsection 5.5 (1,024 nodes, $d = 6 / k = 3$, $1.1 \cdot 10^{12}$ DoFs). A total of 34.6 PB main memory from available 98 PB is used. The largest amount of memory is occupied by the three solution vectors (each 25.4%). The two buffers for MPI communication occupy each 10.3%. One of the buffers is attributed to the ghost-value section of the vector called *tmp*. The remaining data structures, which include inter alia the mapping data, occupy only a small share (3.2%) of the main memory, illustrating the benefit of the tensor-product approach employed by the library `hyper.deal` in constructing a memory-efficient algorithm for arbitrary complex geometries for high dimensions. As reference, the memory consumption for a 4D simulation is shown in Figure 15(a).

5.7 Largest Simulations

In order to demonstrate the large-scale suitability of our code, we have performed simulations on 3,072 compute nodes, the largest possible configuration available on the SuperMUC-NG system. We have conducted a 6D simulation with a curved mesh, which contains $128^6 = 4.4 \cdot 10^{12}$ DoFs (≈ 1.4 GDoFs/node) with a partitioning as specified in Table 10. In this case, we reached a total throughput of $1.1 \cdot 10^{12}$ DoFs/s. This means that each Runge–Kutta stage is processed in 3.7 s and a complete time step consisting of 5 Runge–Kutta stages takes 19.3 s. As a reference, the largest 4D simulation run processed $1,536^4 = 5.56 \cdot 10^{12}$ DoFs with $1.9 \cdot 10^{12}$ DoFs/s.

6 APPLICATION: VLASOV–POISSON

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

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

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

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

The electric field is obtained from the Poisson problem

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

For the time propagation, we use a low-storage Runge–Kutta method of order 4 with 5 stages [25]. Each stage contains the following five steps for evaluating the right-hand side:

- (1) Compute the DoFs of the charge density via integration over the velocity space.
- (2) Compute the right-hand side for the Poisson equation.
- (3) Solve the Poisson equation for ϕ .
- (4) Compute \vec{E} from ϕ .
- (5) Apply the advection operator.

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

6.1 Implementation Details

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

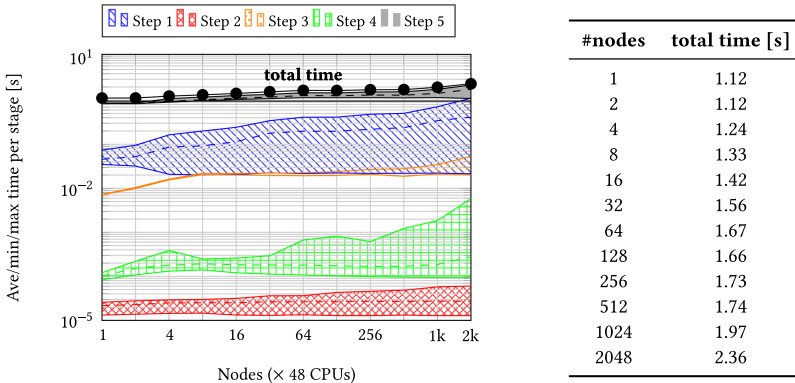


Fig. 16. Weak scaling of a single Runge–Kutta stage of the solution of the Vlasov–Poisson equations on a hypercube. The average (dashed), averaged minimum and averaged maximum runtime of each step are indicated. Additionally, the total runtime is shown with bullets and in the table.

For the solution of the Poisson problem

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

with ψ denoting the test functions, we utilize a matrix-free geometric multigrid solver from deal.II, which uses a Chebyshev smoother with polynomial degree 5 [1] and has settings similar to [14, 33]. The Poisson problem is solved up to a relative tolerance of 10^{-4} by each process group with a constant velocity grid (see `row_comm` in Subsection 3.2). The result of this is that the solution ϕ is available on each process without the need for an additional broadcast step.

The integration of f over the velocity space is implemented via an `MPI_Allreduce` operation over all processes with the same \vec{x} grid partition (i.e., `column_comm`) so that the resulting ρ is available on all processes. We have verified our implementation with a simulation of the Landau damping problem as in [27].

6.2 Weak Scaling

We perform a weak-scaling experiment for the 6D Vlasov–Poisson system, starting from a configuration of 8^6 cells with $k = 3$ on one node. When doubling the number of processes, we double the number of cells in one direction as in Subsection 5.5.

Figure 16 shows the scaling of Steps 1–5 of one Runge–Kutta stage. We can see that the total computing time is dominated by the 6D-advection step, which we have analyzed earlier.

Step 1, which reduces f to ρ , becomes increasingly important as the problem size and the parallelism increase. In this step, an all-reduction is performed over process groups with constant $p_{\vec{x}}$ coordinate in the process grid (called `comm_column` in Subsection 3.2). The amount of data sent by each process corresponds to the number of DoFs in \vec{x} -direction of one process and is thus the same in every experiment. The total amount of data sent/received is therefore proportional to the total number of processes, while the number of reduction steps is only $O(\log(p_{\vec{x}}))$. The scaling experiment shows that the time needed by Step 1 generally increases with the number of nodes and that this step has the worst scaling behavior. We also note that the process grid is designed to optimize the communication of the advection step so that communication patterns of other steps might be suboptimal due to shared-memory blocking, see Figure 4.

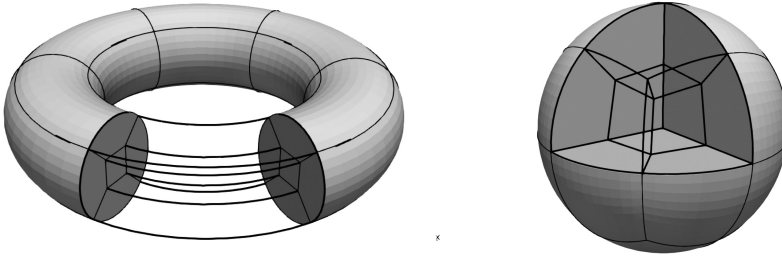


Fig. 17. Coarse grid of the “torus \otimes sphere”-simulation. The torus consists of 30 and the sphere of 32 cells. Curved surface descriptions are used to derive a high-order mapping of the curved surfaces.

Table 11. Weak Scaling of the Problem Size of 1.00e+09 DoFs per Node

| #Nodes | #DoFs | | | Total time per stage [s] |
|--------|--------------|--------------|---------------------------------|--------------------------|
| | \mathbf{x} | \mathbf{v} | $\mathbf{x} \otimes \mathbf{v}$ | |
| 2 | 122.9k | 16.4k | 2.0G | 1.23 |
| 16 | 122.9k | 131.1k | 16.1G | 1.53 |
| 128 | 983.0k | 131.1k | 128.8G | 1.67 |
| 1,024 | 983.0k | 1.0M | 1.0T | 2.03 |

Steps 2–4 are 3D problems, which are mostly negligible. Only the Poisson solver (Step 3) has some impact on the total computing time. Let us note that the 3D parts are solved $p_{\vec{v}}$ times on all subcommunicators (called `comm_row` in Subsection 3.2) with constant $p_{\vec{v}}$ coordinate.

6.3 Tensor Product of a Torus and a Sphere

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

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

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

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

7 SUMMARY AND OUTLOOK

We have presented the finite-element library `hyper.deal`, which efficiently solves high-dimensional partial differential equations on complex geometries with high-order DG methods. It constructs a high-dimensional triangulation via the tensor product of distributed triangulations with dimensions up to three from the low-dimensional FEM library `deal.II` and solves the given

partial differential equation with sum-factorization-based matrix-free operator evaluation. To reduce the memory consumption and the communication overhead, we use a new vector type, which is built around the shared-memory features from MPI-3.0. The proposed algorithms are aligned with the architecture of current pre-exascale machines with high FLOP-per-byte ratios and are expected to also run efficiently on projected exascale machines.

We have compared the node-level performance of the default configuration of `hyper.deal` with alternative algorithms, which are specialized for Cartesian and affine meshes or use collocation integration schemes. Even though the proposed algorithms are not primarily limited by the raw memory bandwidth, our studies reveal that loading less mapping data is most beneficial to improve the performance, and, to a lesser extent, reducing the number of sum-factorization sweeps is beneficial, too. To utilize these advantages on a broader set of configurations, we plan to look into computing the low-dimensional mapping information on the fly and study the benefits of increasing the cache locality during sum-factorization sweeps by a suitable hierarchical cache-oblivious blocking strategy.

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

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

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

APPENDIX

The following code snippets give implementation details on the new shared-memory modus of the vector class `dealii::LinearAlgebra::distributed::Vector`, which is built around MPI-3.0 features (see Section 4).

A new MPI communicator `comm_sm`, which consists of processes from the communicator `comm` that have access to the same shared memory, can be created via:

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

We recommend to create this communicator only once globally during setup and pass it to the vector.

The following code snippet shows the simplified allocation routines of the vector class for the value type `T` and the size `_local_size+_ghost_size`:

```

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

// configure shared memory
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info, "alloc_shared_noncontig", "true");

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

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

Assert(data_this==data_others[rank_sm]);

```

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

```

MPI_Win_free(&win)

```

REFERENCES

- [1] Mark Adams, Marian Brezina, Jonathan Hu, and Ray Tuminaro. 2003. Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *J. Comput. Phys.* 188, 2 (2003), 593–610. DOI: [http://dx.doi.org/10.1016/S0021-9991\(03\)00194-3](http://dx.doi.org/10.1016/S0021-9991(03)00194-3)
- [2] Martin S. Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. 2015. The FEniCS Project Version 1.5. *Arch. Numer. Soft.* 3, 100 (2015), 9–23. DOI: <http://dx.doi.org/10.11588/ans.2015.100.20553>
- [3] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Veselin Dobrev, Yohann Dudouit, Aaron Fisher, Tzanio Kolev, Will Pazner, Mark Stowell, Vladimir Tomov, Ido Akkerman, Johann Dahm, David Medina, and Stefano Zampini. 2021. MFEM: A modular finite element methods library. *Comput. Math. Appl.* 81 (2021), 42–74. DOI: <http://dx.doi.org/10.1016/j.camwa.2020.06.009>
- [4] Daniel Arndt, Wolfgang Bangerth, Bruno Blais, Thomas C. Clevenger, Marc Fehling, Alexander V. Grayver, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Pelteret, Reza Rastak, Ignacio Thomas, Bruno Turcksin, Zhuoran Wang, and David Wells. 2020. The deal.II Library, Version 9.2. *J. Numer. Math.* 28, 3 (2020), 131–146. DOI: <http://dx.doi.org/10.1515/jnma-2020-0043>
- [5] Daniel Arndt, Wolfgang Bangerth, Denis Davydov, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Jean-Paul Pelteret, Bruno Turcksin, and David Wells. 2021. The deal.II finite element library: Design, features, and insights. *Comput. Math. Appl.* 81 (2021), 407–422. DOI: <http://dx.doi.org/10.1016/j.camwa.2020.02.022>
- [6] Markus Bachmayr, Reinhold Schneider, and André Uschmajew. 2016. Tensor networks and hierarchical tensors for the solution of high-dimensional partial differential equations. *Found. Comput. Math.* 16, 6 (2016), 1423–1472. DOI: <http://dx.doi.org/10.1007/s10208-016-9317-9>
- [7] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. 2011. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Software* 38, 2 (2011), 28 pages. DOI: <http://dx.doi.org/10.1145/2049673.2049678>
- [8] Peter Bastian, Christian Engwer, Jorrit Fahlke, Markus Geveler, Dominik Göttsche, Oleg Iliev, Olaf Ippisch, René Milk, Jan Mohring, Steffen Müthing, Mario Ohlberger, Dirk Ribbrock, and Stefan Turek. 2016. Hardware-based efficiency advances in the EXA-DUNE Project. In *Software for Exascale Computing – SPPEXA 2013–2015*, Hans-Joachim Bungartz, Peter Neumann, and Wolfgang E. Nagel (Eds.). Springer International Publishing, Cham, 3–23.
- [9] Gheorghe-Teodor Bercea, Andrew T. T. McRae, David A. Ham, Lawrence Mitchell, Florian Rathgeber, Luigi Nardi, Fabio Luporini, and Paul H. J. Kelly. 2016. A structure-exploiting numbering algorithm for finite elements on extruded

- meshes, and its performance evaluation in Firedrake. *Geosci. Model Dev.* 9, 10 (2016), 3803–3815. DOI: <http://dx.doi.org/10.5194/gmd-9-3803-2016>
- [10] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. 2011. p4est : Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM J. Sci. Comput.* 33, 3 (2011), 1103–1133. DOI: <http://dx.doi.org/10.1137/100791634>
- [11] Denis Davydov, Jean-Paul Pelteret, Daniel Arndt, Martin Kronbichler, and Paul Steinmann. 2020. A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid. *Internat. J. Numer. Methods Engrg.* 121, 13 (2020), 2874–2895. DOI: <http://dx.doi.org/10.1002/nme.6336>
- [12] Andreas Dedner, Robert Klöforn, Martin Nolte, and Mario Ohlberger. 2010. A generic interface for parallel and adaptive scientific computing: Abstraction principles and the DUNE-FEM module. *Computing* 90 (2010), 165–196. DOI: <http://dx.doi.org/10.1007/s00607-010-0110-3>
- [13] Michel O. Deville, Paul F. Fischer, and Ernest H. Mund. 2002. *High-Order Methods for Incompressible Fluid Flow*. Vol. 9. Cambridge University Press, Cambridge.
- [14] Niklas Fehn, Peter Munch, Wolfgang A. Wall, and Martin Kronbichler. 2020. Hybrid multigrid methods for high-order discontinuous Galerkin discretizations. *J. Comput. Phys.* 415 (2020), 109538. DOI: <http://dx.doi.org/10.1016/j.jcp.2020.109538>
- [15] Francis Filbet and Eric Sonnendrücker. 2003. Comparison of Eulerian Vlasov solvers. *Comput. Phys. Communic.* 150, 3 (2003), 247–266. DOI: [http://dx.doi.org/10.1016/S0010-4655\(02\)00694-X](http://dx.doi.org/10.1016/S0010-4655(02)00694-X)
- [16] William J. Gordon and Linda C. Thiel. 1982. Transfinite mappings and their application to grid generation. *Appl. Math. Comput.* 10 (1982), 171–233. DOI: [http://dx.doi.org/10.1016/0096-3003\(82\)90191-6](http://dx.doi.org/10.1016/0096-3003(82)90191-6)
- [17] Wei Guo and Yingda Cheng. 2016. A sparse grid discontinuous Galerkin method for high-dimensional transport equations and its application to kinetic simulations. *SIAM J. Sci. Comput.* 38, 6 (2016), A3381–A3409. DOI: <http://dx.doi.org/10.1137/16M1060017>
- [18] Ammar Hakim, Greg Hammett, Eric L. Shi, and Noah Mandell. 2019. Discontinuous Galerkin schemes for a class of hamiltonian evolution equations with applications to plasma fluid and kinetic problems. *arXiv* 1908.01814 (2019).
- [19] Luca Heltai, Wolfgang Bangerth, Martin Kronbichler, and Andrea Mola. 2021. Propagating geometry information to finite element computations. *ACM Trans. Math. Softw.* 47, 4, Article 32 (2021), 30 pages. <https://doi.org/10.1145/3468428>.
- [20] Michael A. Heroux, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, Kendall S. Stanley, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, and Roger P. Pawlowski. 2005. An overview of the Trilinos project. *ACM Trans. Math. Software* 31, 3 (2005), 397–423. DOI: <http://dx.doi.org/10.1145/1089014.1089021>
- [21] J. S. Hesthaven and T. Warburton. 2008. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer, New York. DOI: <http://dx.doi.org/10.1007/978-0-387-72067-8>
- [22] James Juno, Ammar Hakim, Jason TenBerge, Eric L. Shi, and William Dorland. 2018. Discontinuous Galerkin algorithms for fully kinetic plasmas. *J. Comput. Phys.* 353 (2018), 110–147. DOI: <http://dx.doi.org/10.1016/j.jcp.2017.10.009>
- [23] George Karypis and Vipin Kumar. 1998. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*.
- [24] Dominic Kempf, René Heß, Steffen Müthing, and Peter Bastian. 2021. Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures. *ACM Trans. Math. Software* 47, 1 (2021), 31 pages. DOI: <http://dx.doi.org/10.1145/3424144>
- [25] Christopher A. Kennedy, Mark H. Carpenter, and R. Michael Lewis. 2000. Low-storage, explicit Runge–Kutta schemes for the compressible Navier–Stokes equations. *Appl. Numer. Math.* 35, 3 (2000), 177–219. DOI: [http://dx.doi.org/10.1016/S0168-9274\(99\)00141-5](http://dx.doi.org/10.1016/S0168-9274(99)00141-5)
- [26] David A. Kopriva and Gregor J. Gassner. 2014. An energy stable discontinuous Galerkin spectral element discretization for variable coefficient advection problems. *SIAM J. Sci. Comput.* 36, 4 (2014), A2076–A2099. DOI: <http://dx.doi.org/10.1137/130928650>
- [27] Katharina Kormann, Klaus Reuter, and Markus Rapp. 2019. A massively parallel semi-Lagrangian solver for the six-dimensional Vlasov–Poisson equation. *Int. J. High Perform. Comput. Appl.* 33, 5 (2019), 924–947. DOI: <http://dx.doi.org/10.1177/1094342019834644>
- [28] Benjamin Krank, Niklas Fehn, Wolfgang A. Wall, and Martin Kronbichler. 2017. A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. *J. Comput. Phys.* 348 (2017), 634–659. DOI: <http://dx.doi.org/10.1016/j.jcp.2017.07.039>
- [29] Martin Kronbichler and Katharina Kormann. 2012. A generic interface for parallel cell-based finite element operator application. *Comput. Fluids* 63 (2012), 135–147. DOI: <http://dx.doi.org/10.1016/j.compfluid.2012.04.012>
- [30] Martin Kronbichler and Katharina Kormann. 2019. Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM Trans. Math. Software* 45, 3 (2019), 40 pages. DOI: <http://dx.doi.org/10.1145/3325864>

- [31] Martin Kronbichler, Katharina Kormann, Niklas Fehn, Peter Munch, and Julius Witte. 2019. A Hermite-like basis for faster matrix-free evaluation of interior penalty discontinuous Galerkin operators. *arXiv preprint arXiv:1907.08492* (2019).
- [32] Martin Kronbichler, Svenja Schoeder, Christopher Müller, and Wolfgang A. Wall. 2016. Comparison of implicit and explicit hybridizable discontinuous Galerkin methods for the acoustic wave equation. *Internat. J. Numer. Methods Engrg.* 106, 9 (2016), 712–739. DOI: <http://dx.doi.org/10.1002/nme.5137>
- [33] Martin Kronbichler and Wolfgang A. Wall. 2018. A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers. *SIAM J. Sci. Comput.* 40, 5 (2018), A3423–A3448. DOI: <http://dx.doi.org/10.1137/16M110455X>
- [34] J. Markus Melenk, Klaus Gerdes, and Christoph Schwab. 2001. Fully discrete hp-finite elements: Fast quadrature. *Comput. Methods Appl. Mech. Eng.* 190, 32 (2001), 4339–4364. DOI: [http://dx.doi.org/10.1016/S0045-7825\(00\)00322-4](http://dx.doi.org/10.1016/S0045-7825(00)00322-4)
- [35] Steffen Müthing, Marian Piatkowski, and Peter Bastian. 2017. High-performance implementation of matrix-free high-order discontinuous Galerkin methods. *arXiv preprint arXiv:1711.10885* (2017).
- [36] Steven A. Orszag. 1980. Spectral methods for problems in complex geometries. *J. Comput. Phys.* 37, 1 (1980), 70–92. DOI: [http://dx.doi.org/10.1016/0021-9991\(80\)90005-4](http://dx.doi.org/10.1016/0021-9991(80)90005-4)
- [37] Bram Reys and Tobias Weinzierl. 2017. Complex additive geometric multilevel solvers for Helmholtz equations on spacetrees. *ACM Trans. Math. Software* 44, 1 (2017), 1–36. DOI: <http://dx.doi.org/10.1145/3054946>
- [38] Thomas Roehl, Jan Treibig, Georg Hager, and Gerhard Wellein. 2014. Overhead analysis of performance counter measurements. In *2014 43rd International Conference on Parallel Processing Workshops*, Vol. 2015-May. IEEE, Minneapolis, Minnesota, 176–185. DOI: <http://dx.doi.org/10.1109/ICPPW.2014.34>
- [39] Svenja Schoeder, Katharina Kormann, Wolfgang A. Wall, and Martin Kronbichler. 2018. Efficient explicit time stepping of high order discontinuous Galerkin schemes for waves. *SIAM J. Sci. Comput.* 40, 6 (2018), C803–C826. DOI: <http://dx.doi.org/10.1137/18M1185399>
- [40] Tianjiao Sun, Lawrence Mitchell, Kaushik Kulkarni, Andreas Klöckner, David A. Ham, and Paul H. J. Kelly. 2020. A study of vectorization for matrix-free finite element methods. *Int. J. High Perf. Comput. Appl.* 34, 6 (2020), 629–644. DOI: <http://dx.doi.org/10.1177/1094342020945005>
- [41] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, Wang-Chien Lee (Ed.). IEEE, Piscataway, NJ, 207–216. DOI: <http://dx.doi.org/10.1109/ICPPW.2010.38>
- [42] Takayuki Umeda, Keiichiro Fukazawa, Yasuhiro Nariyuki, and Tatsuki Ogino. 2012. A scalable full-electromagnetic vlasov solver for cross-scale coupling in space plasma. *IEEE T. Plasma Sci.* 40, 5 (2012), 1421–1428. DOI: <http://dx.doi.org/10.1109/TPS.2012.2188141>
- [43] Robert A. Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency-Pract. Ex.* 9, 4 (1997), 255–274.
- [44] Tobias Weinzierl. 2019. The peano software—parallel, automaton-based, dynamically adaptive grid traversals. *ACM Trans. Math. Software* 45, 2 (2019), 41 pages. DOI: <http://dx.doi.org/10.1145/3319797>
- [45] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65. DOI: <http://dx.doi.org/10.1145/1498765.1498785>

Received February 2020; revised May 2021; accepted June 2021

Paper VII



Full length article

On the construction of an efficient finite-element solver for phase-field simulations of many-particle solid-state-sintering processes

Peter Munch^{a,*}, Vladimir Ivannikov^b, Christian Cyron^{b,c}, Martin Kronbichler^{a,d}^a University of Augsburg, Universitätsstraße 12a, Augsburg, 86159, Germany^b Helmholtz-Zentrum Hereon, Max-Planck-Straße 1, Geesthacht, 21502, Germany^c Technical University of Hamburg, Eißendorfer Straße 42, Hamburg, 21073, Germany^d Ruhr University Bochum, Universitätsstraße 150, Bochum, 44801, Germany

ARTICLE INFO

Dataset link: <https://github.com/hpsint/hpsint-data>

Keywords:

Solid-state sintering
 Finite-element computations
 Matrix-free computations
 Jacobian-free Newton–Krylov methods
 Preconditioning
 Grain tracking
 Node-level performance analysis
 Strong-scaling analysis

ABSTRACT

We present an efficient solver for the simulation of many-particle solid-state-sintering processes. The microstructure evolution is described by a system of equations consisting of one Cahn–Hilliard equation and a set of Allen–Cahn equations to distinguish neighboring particles. The particle packing is discretized in space via multicomponent linear adaptive finite elements and implicitly in time with variable time-step sizes, resulting in a large nonlinear system of equations with strong coupling between all components to be solved. Since on average 10k degrees of freedom per particle are necessary to accurately capture the interface dynamics in 3D, we propose strategies to solve the resulting large and challenging systems. This includes the efficient evaluation of the Jacobian matrix as well as the implementation of Jacobian-free methods by applying state-of-the-art matrix-free algorithms for high and dynamic numbers of components, advances regarding preconditioning, and a fully distributed grain-tracking algorithm. We validate the obtained results, examine in detail the node-level performance and demonstrate the scalability up to 10k particles on modern supercomputers. Such numbers of particles are sufficient to simulate the sintering process in (statistically meaningful) representative volume elements. Our framework thus forms a valuable tool for the virtual design of solid-state-sintering processes for pure metals and their alloys.

1. Introduction

Sintering is a physically complex process that includes various mechanisms interacting and competing with each other. The obtained densification and microstructure of the sintered packing are of key interest. The accurate prediction of the powder coalescence for a given material and heating profile is a challenging multiphysics problem, which couples mass transport and mechanics. It is convenient to split the entire sintering process into two stages, as visualized in Fig. 1: the *early stage* and the *later stage*. Initially, the microstructure mainly evolves due to intensive neck-growth and shrinkage, while the particles¹ become strongly non-spherical and grain growth starts to play an important role in the later stage. These rheological differences justify the application of specialized numerical models and methods with different computational costs for each of the stages. For instance, *molecular dynamics* [1,2] provides the most detailed insights into processes taking place during solid-state sintering, but can only be applied

to domains spanning a few particles (or even less). Thus, it is typically not appropriate to predict the densification, which is a hallmark of sintering on the meso- and macroscale. On the contrary, approaches based on *continuum mechanics* [3] can operate on the macroscale but remain phenomenological, since they can predict changes of the geometry of a whole workpiece only with additional assumptions on local material densification and cannot resolve microscopic phenomena, such as grain growth. *Discrete element methods* (DEM) and *phase-field methods* are positioned between the two aforementioned approaches in terms of scale: they can handle packings of hundreds or thousands of particles. While both can capture shrinkage during analysis (provided the corresponding mechanisms are properly included in the model), DEM simulations typically rely on the assumption of nearly spherical particles and remain thus largely limited to early-stage sintering [4]. Capturing both densification and grain growth properly can be crucial for many applications,

* Corresponding author.

E-mail addresses: peter.muench@uni-a.de (P. Munch), vladimir.ivannikov@hereon.de (V. Ivannikov), christian.cyron@hereon.de (C. Cyron), martin.kronbichler@uni-a.de (M. Kronbichler).¹ In practice, a powder particle may contain multiple grains. For reasons of simplicity, we use the terms *particle* and *grain* interchangeably. Such a simplification is admissible for the present work, since, in our studies, each particle always consists of a single grain.<https://doi.org/10.1016/j.commsatsci.2023.112589>

Received 22 June 2023; Received in revised form 15 October 2023; Accepted 16 October 2023

Available online 27 October 2023

0927-0256/© 2023 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (<http://creativecommons.org/licenses/by-nc-nd/4.0/>).

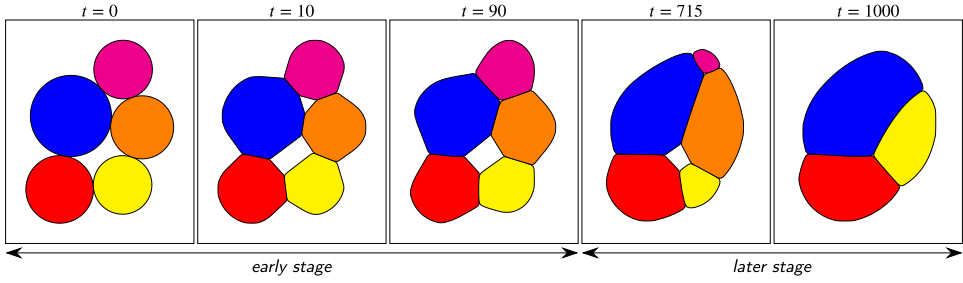


Fig. 1. Visualization of different phases of sintering for a 5-particle packing in 2D. The colors indicate the different grains. Up to $t = 90$ (*early stage*), a clear neck growth is visible and the shape of the particles remains, at least to some extent, spherical; after that (*later stage*), the grain-growth phenomenon plays an important role, with smaller grains disappearing.

for instance, the manufacturing of patient-specific biodegradable magnesium implants [5], where the mechanical properties as well as the biodegradation process may crucially depend on both the geometry and microstructure of an implant [6].

Recently, a number of large-scale simulations of sintering using phase-field methods has been reported. For example, [7] simulated early-stage sintering for packings containing 332, 333, 1172 and 2968 particles. The authors of [8] applied the phase-field framework Pace3D [9] to simulate sintering of 1.2 million particles, using discretization of 2400^3 grid cells. An even larger packing of 3.1 million particles on 2560^3 cells was analyzed in [10], however, for a simpler, ideal grain-growth problem. All these publications have in common that they use *finite difference methods* (FDM) on uniform meshes mostly with explicit time stepping. The *finite element method* (FEM) is successfully used, for instance, in the framework MOOSE [11] to demonstrate the sintering simulation of several hundreds of particles [12] and in the package Tusas [13] to perform various large-scale solidification simulations with up to 270 million unknowns; both using implicit time stepping and Jacobian-free Newton–Krylov (JFNK) methods. The package PRISM–PF [14], in contrast, applies the fast evaluation routines of deal.II to accelerate explicit time stepping in the context of a large set of phase-field simulation cases.

In our previous works, we have tackled modeling of solid-state sintering by multiple numerical methods in close interaction with experimental validation. In [15], we used a FEM-based phase-field approach to simulate shrinkage and neck growth between two particles. To simulate the early-stage sintering process for larger packings of particles, we applied DEM and proposed a novel approach [4] that couples the diffusive mass-transport processes described by an elementary 7-equation model [16] with mechanical interactions of particles arising because of changes of their geometry. Due to the low computational costs of the DEM approach, the developed code is able to simulate the sintering of relatively large packings consisting of 3000–5000 particles on a regular laptop within a few hours. However, this approach is limited to the early stage of sintering and cannot capture non-spherical grains and grain-growth phenomena. In the current work, we extend our phase-field-based code [15] to simulate packing sizes similar to those we considered in the case of DEM, without the limitation to the early sintering stages.

The development of a FEM-based code with implicit time stepping for such scales is a challenging task, since it involves algorithmic developments on many levels. In particular, the number of degrees of freedom (DoFs) N increases linearly with the number of particles (N_p): $N \sim \mathcal{O}(N_c N_p)$, with N_c being the number of order parameters, which are sets of non-neighboring particles. This number is, in practice, rather high (10–20) but independent of the number of particles. Nevertheless, the computational complexity is quadratic $\mathcal{O}(N_c^2 N_p)$, since the surface-coupling terms between particles need to be evaluated.

In this work, we develop efficient evaluation strategies of the Jacobian that fully exploit modern hardware features, reducing the effective complexity to $\mathcal{O}(N_p) - \mathcal{O}(N_c N_p)$. To this end, we adopt Jacobian-free Newton–Krylov approaches and fast matrix-free operator evaluation to solve systems of linear equations arising from implicit time stepping. We, furthermore, discuss efficient preconditioning and propose a fully distributed and improved version of the grain-tracking/remapping algorithm [17]. The latter is crucial to keep the number of order parameters low and, as a result, to make the operator evaluation more efficient. During the whole simulation, we maintain two representations of the particles: a OD representation for postprocessing purposes and a phase field for each order parameter for computation purposes, as shown in Fig. 2. The OD representation is used to detect situations when particles belonging to the same order parameter get too close and to resolve them such that the number of order parameters is minimized. As a consequence, the fast synchronization of both representations is crucial to reduce the computational time and enables then large-scale simulations.

Though the number of order parameters is minimized via grain tracking, it still remains significant. In the context of FEM, each order parameter would correspond, e.g., to a component of a vectorial element. Such a high number of components is not common for FEM applications in other areas, with *density functional theory* as an exception [18,19]. In our case, the number of components may also change between time steps, which poses an additional software challenge.

The remainder of this article is organized as follows. In Section 2, we provide an overview of the governing equations and their FEM discretization. Section 3 outlines the aims of our optimizations, and Section 4 proposes the performance-relevant solver components. Sections 5 and 6 present numerical results and discuss the overall performance of the solver in detail, respectively. Section 7 demonstrates the application of the proposed algorithms in alternative, more advanced, sintering formulations. Finally, Section 8 summarizes our conclusions and points out future research directions. Our novel simulation framework is freely available, as `hpsint`, on GitHub² and uses the open-source library `deal.II` as FEM backend [20,21].

2. Sintering model and its numerical solution

2.1. Governing equations

The classical formulation for modeling solid-state sintering of N_p particles proposed by Wang [22] and adopted in numerous works [23–26] is based on a system of Cahn–Hilliard and Allen–Cahn equations:

$$\frac{\partial c}{\partial t}(\mathbf{x}, t) = \nabla \cdot \left[M \nabla \frac{\delta F}{\delta c} \right], \quad (1a)$$

² <https://github.com/hpsint/hpsint>.

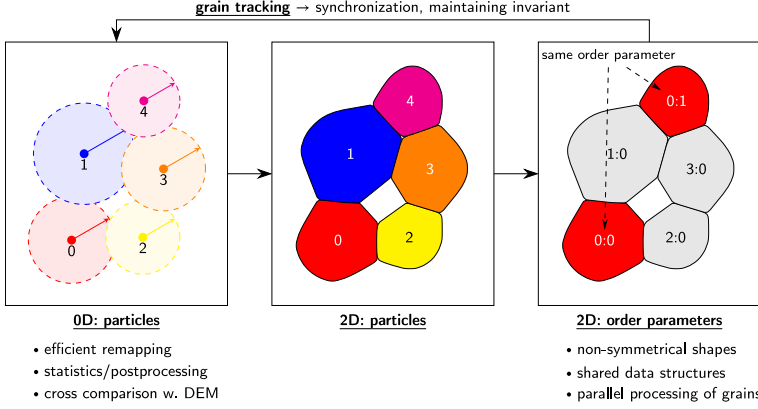


Fig. 2. Overview of types of particle representation considered in this publication and their interaction for a 5-particle packing in 2D. Phase-field simulations are run for order parameters, i.e., sets of particles that do not neighbor. For postprocessing purposes, we maintain a 0D representation with particles described by position, radius, and additional statistical quantities. This reduced model is also used to determine potential contacts of grains and to maintain the invariant that order parameters can only contain non-neighboring particles, which might involve the remapping of solution vectors.

$$\frac{\partial \eta_i}{\partial t}(\mathbf{x}, t) = -L \frac{\delta F}{\delta \eta_i} \quad \text{for } 1 \leq i \leq N_p, \quad (1b)$$

where \mathbf{x} is the position vector in space. The microstructure evolution is described by a conserved variable c and a set of non-conserved unknowns η_i . Variable c can be interpreted as the molar fraction of the overall material and has a magnitude of 1 inside particles and 0 in voids. The unknown η_i describes the position of particle i within the domain such that $\eta_i = 1$ inside the i th particle and $\eta_i = 0$ elsewhere. Due to the local support of η_i , it is common in the literature [17,27,28] to collect non-neighboring particles in groups called *order parameters* and describe all particles in such a group by a single η_i . We have adopted this strategy in this publication. In the following, we implicitly assume that particles are treated in groups unless it is explicitly pointed out that particles are treated individually.

The free energy of system (1) is given by the integral

$$F = \int_{\Omega} \left[f(c, \eta_i) + \frac{1}{2} \kappa_c |\nabla c|^2 + \sum_i \frac{1}{2} \kappa_{\eta_i} |\nabla \eta_i|^2 \right] d\Omega \quad (2)$$

based on the following Landau-type polynomial:

$$f = Ac^2(1-c)^2 + B \left[c^2 + 6(1-c) \sum_i \eta_i^2 - 4(2-c) \sum_i \eta_i^3 + 3 \left(\sum_i \eta_i^2 \right)^2 \right], \quad (3)$$

where A and B are energy coefficients and κ_c and κ_{η_i} are gradient prefactors. These parameters can be extracted from the surface and grain-boundary energy properties of the material by using the relations obtained from the analysis of the behavior of the phase-field variables across the flat surface [24].

Even though it has been recently revealed [29] that the bulk free energy (3) may spontaneously generate the void phase on triple and higher-order junctions, we have still decided to use the original formulation from [22] due to its widespread use and for the sake of simpler validation and comparison with the existing literature. The implementation aspects discussed in the next sections are applicable to similar models based on other free-energy expressions, provided $F(c, \eta_i)$ is a continuous differentiable function.

Parameter L defines the mobility of the grain boundary and is typically set as a constant. The diffusion along different pathways is introduced by the scalar mobility [22]

$$M = M_{vo} \phi + M_{va} (1 - \phi) + \frac{4M_s c^2 (1 - c)^2}{\sum_{i=1}^{N_p} \sum_{j \neq i}^{N_p} \eta_i \eta_j}, \quad (4)$$

where $\phi = c^3(10 - 15c + 6c^2)$. Here, the subscripts vo , va , s and gb denote the mobility coefficients for the volumetric, vapor, surface, and grain-boundary paths, respectively. For the sake of simplicity, the scalar form of the mobility is chosen in the following, whereas a more complex tensorial form [23] is discussed as an extension separately in Section 7. The mobility coefficients can be introduced via the Arrhenius relationship by defining the corresponding prefactors and activation energies [30]. This allows the model to be calibrated with the available experimental data [15].

We note that the surface-mobility term in Eq. (4) (underlined) slightly differs from that used in [7,22,30,31] to enhance the convergence rate of the Newton solver. A more detailed discussion of this alteration can be found in Supplementary Material S1.

2.2. Discretization

We discretize Eq. (1) by means of multicomponent linear Lagrange C^0 finite elements. For this purpose, we reformulate the original system to:

$$\begin{aligned} \frac{\partial c}{\partial t} &= \nabla \cdot [M \nabla \mu], \\ \mu &= \frac{\partial f}{\partial c} - \kappa_c \nabla^2 c, \\ \frac{\partial \eta_i}{\partial t} &= -L \left[\frac{\partial f}{\partial \eta_i} - \kappa_{\eta_i} \nabla^2 \eta_i \right], \end{aligned}$$

by introducing the chemical potential $\mu = \delta F / \delta c$ as auxiliary variable, expanding explicitly the variational derivatives $\delta F / \delta c$ and $\delta F / \delta \eta_i$, and exploiting the definition (2). This leads to the following weak form:

$$\left(v_c, \frac{\partial c}{\partial t} \right) = - \left(\nabla v_c, M \nabla \mu \right), \quad (5a)$$

$$\left(v_{\mu}, \mu \right) = \left(v_{\mu}, \frac{\partial f}{\partial c} \right) + \left(\nabla v_{\mu}, \kappa_c \nabla c \right), \quad (5b)$$

$$\left(v_{\eta_i}, \frac{\partial \eta_i}{\partial t} \right) = - \left(v_{\eta_i}, L \frac{\partial f}{\partial \eta_i} \right) - \left(\nabla v_{\eta_i}, L \kappa_{\eta_i} \nabla \eta_i \right), \quad (5c)$$

where the usual boundary integrals arising after applying integration by parts vanish due to imposition of the no-flux boundary conditions.

We use BDF2 with adaptive time steps for time discretization. The resulting nonlinear system $\mathcal{F}(\mathbf{u}) = 0$, with the vector of unknowns $\mathbf{u} = [c \ \mu \ \eta_i]^T$, is solved by means of a Newton solver. The action of the Jacobian on a factor is either evaluated exactly or approximated

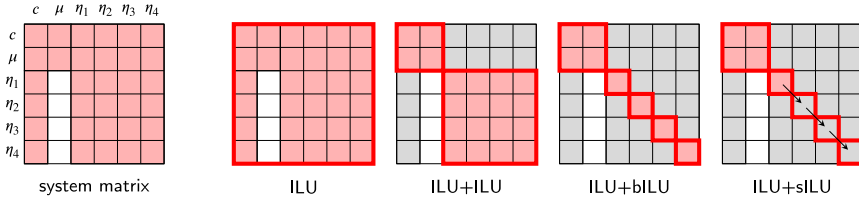


Fig. 3. Visualization of the block sparsity pattern of (1) the system matrix and (2) the considered preconditioners for four order parameters.

by finite differences around the linearization point \mathbf{u}_{lin} (Jacobian-free approach):

$$J(\mathbf{u}_{\text{lin}})\mathbf{p} \approx J'(\mathbf{u}_{\text{lin}})\mathbf{p} = \frac{F(\mathbf{u}_{\text{lin}} + \beta\mathbf{p}) - F(\mathbf{u}_{\text{lin}})}{\beta}, \quad (6)$$

where the parameter β is chosen as described in [32–34]. The linearization of the weak form (5) is derived with respect to variations of the state vector $\delta\mathbf{u}$ as

$$\left(v_c, \frac{\partial \dot{c}}{\partial c} \delta c \right) = - \left(\nabla v_c, \left[\frac{\partial M}{\partial c} \delta c + \frac{\partial M}{\partial \nabla c} \nabla \delta c + \frac{\partial M}{\partial \eta_j} \delta \eta_j + \frac{\partial M}{\partial \nabla \eta_j} \nabla \delta \eta_j \right] \nabla \mu \right), \quad (7a)$$

$$\left(v_\mu, \delta \mu \right) = \left(v_\mu, \frac{\partial^2 f}{\partial c^2} \delta c + \frac{\partial^2 f}{\partial c \partial \eta_j} \delta \eta_j \right) + \left(\nabla v_\mu, \kappa_c \nabla \delta c \right), \quad (7b)$$

$$\left(v_{\eta_i}, \frac{\partial \dot{\eta}_i}{\partial \eta_i} \delta \eta_i \right) = - \left(v_{\eta_i}, L \frac{\partial^2 f}{\partial \eta_i \partial \eta_j} \delta \eta_j \right) - \left(\nabla v_{\eta_i}, L \kappa_p \nabla \delta \eta_i \right). \quad (7c)$$

The notation $\dot{\psi} = \partial \psi / \partial t$ is used here to denote the first time derivative of an arbitrary variable ψ for convenience. Note that (7a) contains derivatives of M with respect to gradients of concentration c and order parameters η_j in order to account for tensorial mobility described in Section 7. The linearized forms of the free-energy function and the mobility are listed in Supplementary Material S2. Fig. 3 shows the sparsity pattern of the resulting Jacobian matrix. The coupling terms introduce a quadratic complexity $\mathcal{O}(N_c^2 N_p)$ in both storage and computational effort, making the assembly of the Jacobian unfeasible.

2.3. Algorithmic overview

Fig. 4 shows an overview of the algorithm used to solve the solid-state-sintering problem. Before solving the nonlinear system with a Newton solver, we optionally run adaptive mesh refinement (AMR) and the grain tracker to detect potential new contacts and to minimize the number of order parameters. After the solution or when the linear/nonlinear solver fails to converge in the prescribed number of iterations, we optionally increase or decrease the time-step size τ . In the following, we investigate this algorithm regarding performance.

3. Performance metric

Our aim is to minimize the computational time for running a solid-state-sintering simulation up to the required physical end time, allowing the simulation of larger problem sizes. For the described solution approach, the runtime can be estimated by the sum of the costs of the nonlinear solution process and other costs, like AMR, grain tracking, and postprocessing:

$$T = T_{\text{sol}} + T_{\text{AMR}} + T_{\text{grain tracking}} + T_{\text{post}} + \dots$$

The cost of the nonlinear solution process [35] is the sum of the costs of each time step:

$$T_{\text{sol}} = \sum T_{\text{sol},i} = N_T \bar{T}_{\text{sol}}.$$

In the following, we consider averaged times, which are indicated by overbars. Under the assumption that we use a Newton solver and solve

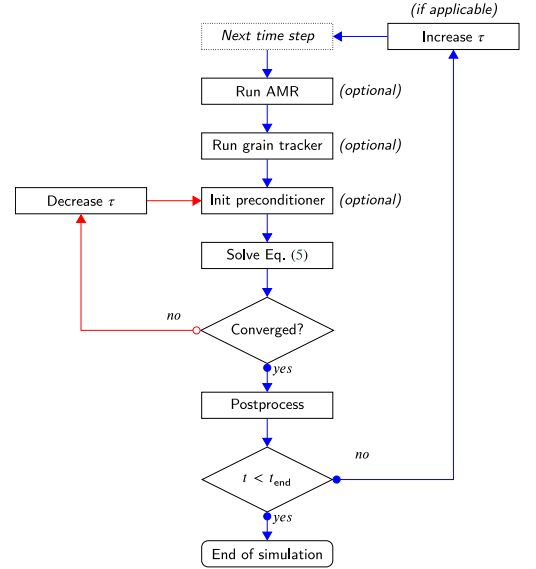


Fig. 4. Simplified flow chart of the solution procedure of the sintering problem with adaptive time stepping and adaptive meshing.

the Jacobian by means of iterations of the preconditioned generalized minimal residual method (GMRES), we can refine the estimates of the costs $\bar{T}_{\text{sol}} = T_{\text{sol}} / N_T$:

$$\begin{aligned} \bar{T}_{\text{sol}} = & \bar{T}_{P,\text{setup}} &> \text{setup preconditioner} \\ & + \bar{N}_N \bar{T}_{J,\text{setup}} &> \text{setup Jacobian} \\ & + \bar{N}_N \bar{N}_R \bar{T}_{\text{residual}} &> \text{residual evaluation} \\ & + \underbrace{\bar{N}_N \bar{N}_L (\bar{T}_{J,\text{apply}} + \bar{T}_{P,\text{apply}} + \bar{T}_{\text{updates}})}_{> \text{single linear iteration}} \end{aligned}$$

with the preconditioner only updated once per nonlinear solve. Here, \bar{N}_N is the number of nonlinear iterations, each of which requires \bar{N}_R residual evaluations and \bar{N}_L iterations of the linear solver. To accelerate the solution process, one needs, on the one hand, to minimize the accumulated number of Jacobian and preconditioner evaluations (each $N_T \bar{N}_N \bar{N}_L$) and of their setup (N_T and $N_T \bar{N}_N$ times, respectively) and, on the other hand, to minimize the cost of their application and construction ($\bar{T}_{J,\text{apply}}$, $\bar{T}_{P,\text{apply}}$, $\bar{T}_{J,\text{setup}}$, $\bar{T}_{P,\text{setup}}$). The costs \bar{T}_{updates} related to the updates of the solution vector are fixed and can hardly be optimized.

We note that these different costs need to be balanced against each other. For example, minimizing only N_T by increasing the time-step sizes can lead to increased \bar{N}_N and \bar{N}_L due to increased nonlinearity

and nonsymmetry of the system of linear equations to be solved. In the present work, we increase the size of the time steps as long as a user-provided threshold regarding the numbers of linear and nonlinear iterations is not violated (see Section 4.5.2). This implies that the main factors we can tune are the costs of the evaluation of the Jacobian ($\overline{T}_{J, \text{apply}}$) and the setup and application costs of the preconditioner ($\overline{T}_{P, \text{setup}}, \overline{T}_{P, \text{apply}}$). The choice of the preconditioner has an effect not only on $\overline{T}_{P, \text{setup}}, \overline{T}_{P, \text{apply}}$ but also on $\overline{N}_P, \overline{N}_N,$ and \overline{N}_L . Hence, an effective preconditioner is characterized by a tradeoff between the setup and application times as well as the resulting iteration counts. Generally $\overline{N}_J \gg \overline{N}_R$, which implies that the costs of the residual evaluation are not crucial. At the same time, this function constitutes the core algorithm of the Jacobian-free implementation. For computational efficiency, this work uses a matrix-free evaluation of the Jacobian, which is conceptually similar to the residual evaluation. Here, the cost of setting up the factors of the Jacobian matrix is small and only involves storing the linearization point and, potentially, precomputing its values at the quadrature points.

Minimizing the solution time by splitting the spatial computation domain into partitions for parallel computation – so that each process only works on a part of the mesh cells, called *locally owned cells* – is a key aspect to enable large systems, since most of the computational time is spent here. However, extensive memory usage and poor parallel scalability of other parts of the code might become a bottleneck as well. For instance, gathering data from all processes to be able to run the grain-tracking algorithm, as done in [17,36], is only feasible for small number of processes but is not an option at a larger scale, limiting the maximum problem sizes per process that can be solved. This implies that we need a grain-tracking algorithm that returns the minimal number of order parameters but is also computationally and memory-wise cheap to apply even if it is not executed at each time step.

Hardware

All experiments are performed, unless stated otherwise, on a dual-socket 20-core *Intel Cascade Lake Xeon Gold 6230* system (2.6 TFLOP/s, 180 GB/s, AVX-512) with up to 8 compute nodes. To give a broader performance perspective, we also report experiments from a dual-socket *AMD EPYC 7713* processor with 64 cores per socket. It has an aggregated bandwidth of 320–340 GB/s and only supports AVX2, but the clock frequency and core count are higher (4.6 TFLOP/s). The parallel-scaling experiments are executed on a dual-socket 24-core *Intel Xeon Platinum 8174* (Skylake) system of the supercomputer SuperMUC-NG,³ (achieved STREAM triad memory throughput of 205 GB/s) with up to 1024 compute nodes (49k processes).

4. Solver components

Section 3 identifies the solver components needed for scalable large-scale simulations of solid-state-sintering processes: fast operator evaluation of the Jacobian and the residual as well as a balance between the costs and effectiveness of the preconditioner. Both ingredients need to be able to deal with a large and dynamic number of order parameters, i.e., components. A fully parallel implementation of the grain-tracking algorithm enables a low number of order parameters also for large simulations, where replicating information is unfeasible. In the following, we present the algorithmic realization of these steps.

Algorithm 1: Cell loop considering *all* vector blocks. The conversion of the number of components into a constant expression is done at a central place before looping over all cells.

```

1 if  $n_{\text{blocks}} = n_{\text{static}}^{\text{blocks}}$  then
2   for cell  $\in$  cells do
3     for  $b = 1$  to  $n_{\text{blocks}}$  do
4       | read from block  $b$  of source vector
5       | perform cell integral  $\rightarrow$  action of  $\mathcal{A}_c$  in (8)
6       for  $b = 1$  to  $n_{\text{blocks}}$  do
7         | write to block  $b$  of destination vector
8 else
9   | not shown

```

4.1. Dealing with large and dynamic number of components

During sintering, grains build necks with neighboring particles, increase and/or decrease in size, and might disappear. The grain-tracking algorithm assigns and reassigns the grains to order parameters. Typical values of the number of order parameters are between 8 and 14. Depending on the topology of grains, the number of order parameters can be dynamically reduced or has to be increased. A natural choice is to assign each order parameter to a component of a vectorial finite element. For solid-state sintering, this implies vectorial elements with two components for the Cahn–Hilliard system and one component for each order parameter. Such a dynamic behavior is in contrast to the fixed number of components in most vectorial problems solved with finite-element models.

In order to simplify the workflow in the context of a general-purpose FEM library, we do not actually work with vectorial elements but with multivectors defined upon scalar finite elements, which we manually combine on the cell⁴/quadrature-point level. Each component corresponds to a vector in the multivector, which simplifies adding or removing blocks during remapping. In the following, we use the term *block* to denote an individual vector related to a scalar function space within the multivector and denote the vector of all unknowns as block vector. To accelerate computations of integrals on the mesh cells, we use the C++ template mechanism for generating separate code for different numbers of components to provide the compiler with optimization opportunities via known loop bounds and data-structure sizes. From compute kernels precompiled up to a known maximal possible number of components, the right kernel is chosen at runtime. The resulting procedure is summarized in Algorithm 1. The memory consumption is $\mathcal{O}(N_c N_p)$ and the computational complexity is $\mathcal{O}(N_c^2 N_p)$, since coupling between order parameters is considered on the cell level also when a cell is not cut by the boundary of a grain.

In the context of large-scale finite-difference implementations [8, 10], it is common to exploit the local support of grains [37], i.e., $\eta_i(\mathbf{x}) > 0$ only for limited \mathbf{x} . For this purpose, the indices of relevant grains are stored for each cell. In practice, the number of possible relevant grains is fixed and limited, e.g., to $c_{\text{max}} = 6$. Since $c_{\text{max}} \ll c$ is generally much lower than the number of order parameters and – per definition – constant, the memory consumption is $\mathcal{O}(N_p)$ and the computational complexity is $\mathcal{O}(N_p)$ with a larger constant of proportionality.

It is also possible to adopt this approach in the context of FEM if one realizes that the data structure used for existing finite-difference codes is – in a nutshell – a compressed-row-storage-format object with rows being the cell/vertex/DoF indices, storing a fixed number of the relevant column indices (grain indices) and associated values (η_i). In the context of computational solution of the density functional theory based on FEM [38,39], such a data structure was successfully used in the form of *sparse block vectors*.

³ <https://top500.org/system/179566/> received on December 11, 2022.

⁴ We use the terms *cell* and *element* interchangeably.

Algorithm 2: Cell loop considering only vector blocks *relevant for the current cell*. The conversion of the number of components into a constant expression has to be performed for each cell.

```

1 for cell ∈ cells do
2   blocks ← relevant blocks of cell
3   for b ∈ blocks do
4     read from block b of source vector
5   if |blocks| = nstaticblocks then
6     perform cell integral → action of Ae in (8)
7   else
8     not shown
9   for b ∈ blocks do
10    write to block b of destination vector

```

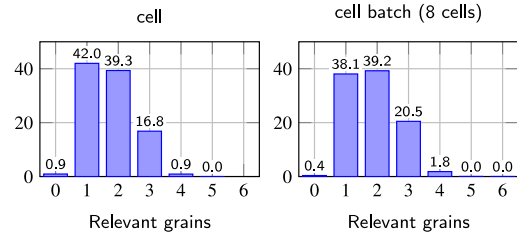


Fig. 5. 51 particles: distribution [%] of cells and cell batches with n relevant grains at $T = 500$ (see Section 6).

Despite of the appeal of sparse block vectors with N_p blocks regarding memory consumption, their usage within an implicit (non)linear solver is challenging. For instance, sparse block vectors imply different and frequently changing sparsity patterns of the matrix of each block, which makes the setup of preconditioners more expensive and parallelization more challenging if established linear-algebra libraries should be used. However, processing all particles of one order parameter implies a natural parallelization across particles under the assumption that the grain-tracking algorithm is able to reduce the number of order parameters to a reasonable value. Therefore, we defer the investigation of such data structures to future work and, instead, focus on a novel simplified approach: we allocate (dense) block vectors in memory but limit the arithmetic work on cells to the relevant blocks in the block vector, including the coupling in cell integrals. This is achieved by storing the relevant order parameters for each cell. Obviously, this approach still implies a memory consumption of $\mathcal{O}(N_c N_p)$ and an associated overhead throughout the remainder of the solver (e.g., non-relevant entries of the vectors have to be zeroed and communicated), but the computational effort can be significantly lowered. In the current work, we use the—heuristic—criterion $\eta_i > 10^{-5}$ to determine whether a grain is relevant within a cell. Fig. 5 shows the distribution of the resulting number of components on cells. The value ranges from 0 to 6 with 81% of the cells only containing 1 or 2 grains, which indicates a significant increase of efficacy. We present the corresponding performance comparison in Section 6. The resulting algorithm is shown in Algorithm 2 indicating that the number of components needs to be translated into a constant expression for each cell.

4.2. Fast Jacobian and residual evaluation

A widespread optimization of iterative linear solvers for modern hardware is to not assemble the final global matrix but implement the action of the linear(ized) operator through a loop over cells and compute the FEM integrals on the fly in a matrix-free way. This approach

has been established in the high-order spectral-element community. By now, it is commonly used in the computational fluid dynamics formulations [40–42] and has been also applied, e.g., in the context of solid mechanics [43,44], material science [14] and computational plasma physics [45]. Depending on the polynomial degree and the underlying quadrature formula, the matrix-free operator evaluation is beneficial for accelerating the actual matrix–vector product or only for reducing the setup costs of the Jacobian. The latter might provide, depending on iteration counts, a better overall runtime also when the actual evaluation is more expensive.

Due to the implementation similarity between the matrix-free evaluation of the Jacobian and the evaluation of the residual, we introduce the concepts of high-performance evaluation of an arbitrary operator $\mathcal{A}(x)$. The overall structure is [46–48]:

$$v = \mathcal{A}(x) = \sum_e \underbrace{G_e^T \circ \underbrace{S_e^T \circ Q_e \circ S_e}_{v_e = \mathcal{A}_e(x_e)} \circ G_e}_e. \quad (8)$$

For each cell e , the operator G_e gathers the values pertaining to the local FEM solution expansion from the source vector x and applies constraints (like hanging-node constraints [49] related to AMR). The operator S_e computes values or gradients associated to the vector x at the quadrature points. These quantities are processed by a quadrature-point operation Q_e ; the result is integrated and summed into the vector v by applying S_e^T and G_e^T . In the literature, specialized implementations for GPUs [48,50–52] and CPUs [45–47,50,53] for operations as expressed in (8) have been presented, including the use of the structure in interpolation matrices. For CPUs, it is an option to vectorize across elements [46,47], i.e., evaluate \mathcal{A}_e for multiple cells in different lanes of vector execution units by the same instructions. This necessitates the data to be laid out in a struct-of-arrays fashion. The necessary permutations to support unstructured meshes can be done, e.g., by G_e , while looping through all elements [46]. On modern CPUs, the most common number of lanes N_{SIMD} is either 4 (AVX) or 8 (AVX512) for double-precision floating-point numbers, implying that batches of 4 or 8 cells are processed at once.

For multicomponent FEM, the indices within G_e are the same for each block, allowing to process blocks one by one with the same index data. Furthermore, S_e and S_e^T can be executed for each component individually. The operator Q_e determines the quantities to be provided at quadrature points, encoding the actual physics via the weak forms (5) and (7). In the case of the residual (5) and the Jacobian (7), we need the values and the gradients of c , μ , η_i and have to multiply the results by both the value and the gradient of the respective test function. When evaluating the operators in (8) sequentially, the amount of temporary data is $N_{\text{SIMD}}(N_c + 2)(d + 1)(p + 1)^d$ data fields, with p the polynomial degree and d the spatial dimension. The performance crucially depends on the ability to keep the temporary results between the steps of Eq. (8) on a cell accessible quickly. Hence, the higher numbers of components require larger cache capacities to remain fast.

At each quadrature point, the values and gradients of c , μ , η_i are coupled via Q_e . From a performance point of view, the two limiting factors are possible register spills for high number of components and $\mathcal{O}(N_c^2)$ arithmetic operations. Apart from constants, this is the same complexity as for a matrix-based implementation, since coupling between all components arises in both cases. The crucial difference is that a matrix-based code involves the coupling over the whole stencil of a point. Furthermore, matrices are large objects in memory with low data reuse, implying that data needs to be loaded via the slow main memory, whereas the matrix-free evaluation only experiences the complexity on a single point and on cached data. In the ideal case, the time complexity is thus $\mathcal{O}(N_p N_c)$ if the computation can be hidden behind the global memory access. In the following, we concentrate on linear elements ($p = 1$) in 3D, as common in the literature [12,26,30] for simulating sintering problems.

Fig. 6 shows the time and throughput of the computation of the residual and the application of the Jacobian. Here, the throughput is

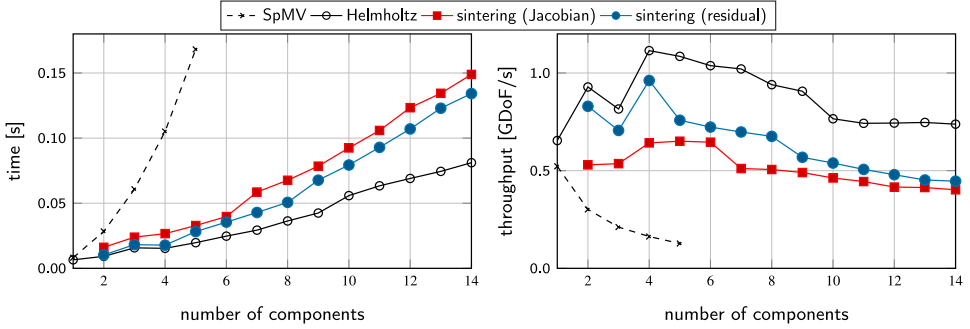


Fig. 6. Comparison of the time and throughput (given in $\text{GDoF/s}=10^9\text{DoF/s}$) of the application of sparse matrix (SpMV), of the vector Helmholtz operator, and of the generic sintering operator. 4.3 million DoFs per vector component for different number of components. Run on 1 compute node with Intel Xeon 6230 (40 cores).

computed as the ratio between the number of degrees of freedom for all vector components and the runtime/wall time of the experiment in seconds, in short, the number of DoFs processed per second. We use Algorithm 1 on a uniformly refined Cartesian mesh with 4.3 million DoFs per vector component. As a reference, we also list the timings for the evaluation of the vector Helmholtz operator $(v_i, u_j) + (\nabla v_i, \nabla u_j)$. Since this operator involves the same operations S_e and S_e^T , the difference illustrates the cost of the physics-based Q_e . The residual evaluation of the sintering operator involves around 50% more floating-point operations on average, with the execution being 30% slower than the one of the vector Helmholtz operator. Evaluating the Jacobian gives a 19% lower throughput than the residual because of additional floating-point operations from linearization and additional data access for loading a given linearization point, as we precompute the values of c , μ , η_i and the gradients of c , μ at the quadrature points. We also show the times of the multiplication with a sparse matrix in which all components are coupled. It is clear that the time of the sparse matrix-vector multiplication increases quadratically with the number of components, as does the memory consumption, making this approach unfeasible for large number of components. Note that sparse matrices involve additional costs for the assembly and the creation of the pattern of non-zero entries.

The experiments also show a decrease in throughput with increasing the number of components for the matrix-free case, which indicates a quadratic complexity with low constant of proportionality. These results motivate the development of algorithmic variants that reduce the number of components a particular cell needs to work on, e.g., by using Algorithm 2. The central ingredient for this algorithm is to track the relevant grains *per cell batch*. Since the batching of cells is done a priori, changes in the simulation lead to a slight increase in the number of relevant grains per cell batch, compared to the one of the relevant grains per cell, as illustrated in Fig. 5.

Table 1 illustrates the performance metrics of the underlying experiments. One can see that the write access to main memory, normalized per unknown, increases only marginally with increasing number of components, indicating that all relevant data needed for S_e/Q_e fit into cache. The data read per DoF decreases with increasing number of components, reflecting the fact that other data (e.g., indices and metric terms) can be shared between components and the relative amount of data related to the linearization point, where the Cahn-Hilliard part dominates, decreases. The number of floating-point operations increases as 73–78 and 86–99 FLOP per quadrature point as the number of components increases in the cases of residual and Jacobian evaluation, respectively. As a summary, the results are inserted into a graphical representation of the roofline performance model [55] for the residual evaluation in Fig. 7. With increasing the number of components, the arithmetic intensity increases (11.5–13.1 FLOP/byte)

Table 1

Comparison of measured throughput, read/write memory access and arithmetic work for evaluation of the residual and the Jacobian of generic sintering operator for different number of components (see Fig. 6). As a reference, the vector Helmholtz operator involves 416 FLOP/DoF. The hardware performance counters were accessed with the LIKWID tool [54].

| N_c+2 | Sintering (residual) | | | | Sintering (Jacobian) | | | |
|---------|----------------------|-----|-----|-----|----------------------|------|-----|-----|
| | D/s | r/D | w/D | F/D | D/s | r/D | w/D | F/D |
| 2 | 0.83 | 5.3 | 1.3 | 608 | 0.53 | 37.7 | 1.6 | 690 |
| 3 | 0.71 | 4.9 | 1.5 | 591 | 0.54 | 29.4 | 1.7 | 688 |
| 4 | 0.96 | 4.7 | 1.5 | 579 | 0.64 | 25.1 | 1.7 | 704 |
| 5 | 0.76 | 4.6 | 1.6 | 579 | 0.65 | 22.6 | 1.8 | 711 |
| 6 | 0.72 | 4.6 | 1.6 | 579 | 0.65 | 20.9 | 1.8 | 717 |
| 7 | 0.70 | 4.5 | 1.6 | 582 | 0.51 | 19.7 | 1.9 | 723 |
| 8 | 0.68 | 4.4 | 1.7 | 585 | 0.51 | 18.7 | 1.9 | 730 |
| 9 | 0.57 | 4.4 | 1.7 | 590 | 0.49 | 18.1 | 1.9 | 756 |
| 10 | 0.54 | 4.4 | 1.7 | 595 | 0.46 | 17.5 | 2.0 | 763 |
| 11 | 0.51 | 4.3 | 1.7 | 601 | 0.44 | 17.0 | 2.0 | 771 |
| 12 | 0.48 | 4.2 | 1.7 | 607 | 0.42 | 16.5 | 2.1 | 779 |
| 13 | 0.45 | 4.2 | 1.7 | 614 | 0.41 | 16.2 | 2.1 | 786 |
| 14 | 0.45 | 4.2 | 1.8 | 620 | 0.40 | 15.9 | 2.1 | 794 |

D/s: throughput in [GDoFs/s]; r/D: read data per DoF in [Double/DoF]; w/D: written data per DoF in [Double/DoF], F/D: work [FLOP/DoF]. Due to the usage of Gauss-Legendre quadrature implying approx. 8 quadrature points per vertex, F/D needs to be divided by 8 to obtain the number of FLOP per quadrature point and component. The numbers represent the average of 100 repetitions of the experiment.

and the obtained performance decreases (556–277 GFLOP/s). As a result, 27%–12% of the obtainable hardware bandwidth limit is reached. Albeit the gap appears to be significant, the result can be explained by limits not included in this simplistic roofline model, most prominently by the relatively high share of non-floating point operations for linear shape functions, such as unstructured gather/scatter access and related integer operations. For comparison, a sparse matrix-vector product (SpMV) would reach a much lower performance of 45 GFLOP/s due to an arithmetic intensity ≈ 0.25 FLOP/byte.

In order to reduce the impact of a higher number of components, the following optimization strategies are developed:

- We do not explicitly compute individual terms of the free energy (3) or mobility (4) but instead apply them directly to vectors. This allows us to transform equations like

$$\sum_{j,i \neq j} \frac{\partial^2 f}{\partial \eta_i \partial \eta_j} u_j = \sum_{j,i \neq j} (\eta_i \eta_j) u_j = \eta_i \sum_{j,i \neq j} \eta_j u_j = \eta_i (\alpha - \eta_i u_i),$$

with the precomputed factor $\alpha = \sum_i \eta_i u_i$. Note that some scaling factors are dropped for the sake of simplicity.

- Other factors, e.g., $\sum_i \eta_i^2$ and $\sum_i \eta_i^3$, can be precomputed and reused for each grain.
- Off-diagonal block entries are processed in pairs, and symmetry is exploited, e.g., $\sum_{i=1}^g \sum_{j=1, i \neq j}^g \eta_i \eta_j = 2 \sum_{i=1}^g \sum_{j=1}^{i-1} \eta_i \eta_j$.

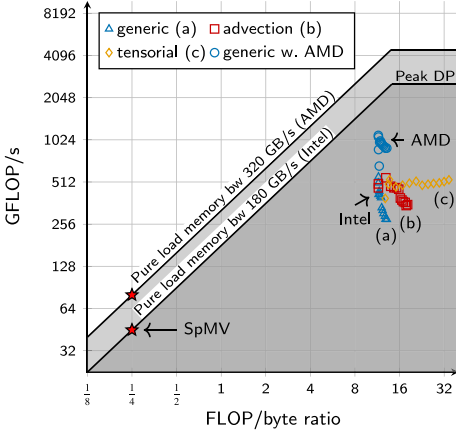


Fig. 7. Roofline performance model of the evaluation of the residual of the generic operator (Table 1) as well as the advection and tensorial operator (Table 8) for different number of components. On the Intel system, the bandwidth and the number of floating-point operations are extracted from hardware-performance counters. On the AMD system, where direct measurements have not been possible, the memory access of the Intel system is assumed as a model.

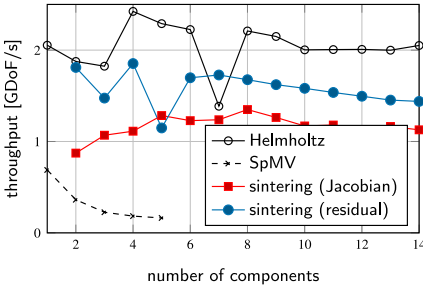


Fig. 8. Comparison of the throughput of application of the sparse matrix, of the vector Helmholtz operator, and of the generic sintering operator on the AMD hardware.

- Those coupling terms that cannot be reformulated and, as a consequence, introduce a quadratic complexity are explicitly optimized.

Further examples are discussed in Section 7 for tensorial mobility, which requires more transformations due to a higher arithmetic load. In particular, that section contains an example for the last bullet point.

Remark. We have experimented with advanced techniques to increase the throughput of the cell integrals, namely (1) interleaving the evaluation/integration with the loop over the quadrature points by performing quadrature in two-dimensional layers at a time [47] and (2) implementing the arithmetic work for 2^d cells at once. These techniques allow, e.g., to decrease the size of the working set by additional cache blocking and to reduce the cost of indirect addressing when accessing the DoFs on cells. The preliminary results are promising with speedups between 50% and 100%, allowing for a reduction of the gap to the hardware bandwidth limit. Detailed investigations on these options and alternative vectorization strategies, e.g., vectorization over components [53], are deferred to future work.

Behavior on AMD and GPU hardware

The above-mentioned performance bottlenecks in terms of the number of components are partly due to the specific microarchitecture of the chosen Intel hardware. In order to better illustrate the performance capabilities on the expected trajectory of computer hardware evolution, we performed a node-level performance experiment on a more recent AMD Epyc 7713 processor. According to Fig. 8, the throughput is on average $2.3\times$ higher than the Intel results from Fig. 6, which is related to the higher bandwidth and compute performance ($1.8\times$ each) as well as to larger-bandwidth caches. Our code utilizes around 50% of the available memory bandwidth for a low number of components and around 40% for a higher number of components, as indicated by the roofline performance model (Fig. 7). This shows that the proposed algorithms also work, as expected, on alternative CPU-based hardware, however, the dependency on the number of components is less prominent due to a more balanced cache system of the AMD hardware.

The present work focuses on CPU implementations. However, fast (matrix-free) operator evaluation is also attractive on GPUs. Refs. [44, 51] discuss optimizations for some standard operators. For the sintering operator, the need to compute significantly larger amount of data at the quadrature points limits the applicability of current algorithms, since many implementations rely on keeping most intermediate data in registers rather than in caches used for CPUs. However, we believe that the novel algorithm design to organize work along 2D layers, as described above, combined with moving the remaining data into shared memory, as proposed for high orders in [56], could be beneficial.

Application in Jacobian-free methods

We conclude this subsection by discussing an efficient implementation of a Jacobian-free method (6) in the case of the fast operator evaluation (8) for the residual. In order to evaluate a Jacobian-free operator n times, we need to evaluate the residual $n + 1$ times. During the first iteration, the residual at the linearization point $F(\mathbf{u})$ has to be evaluated once, which can be reused in the subsequent operations. Within the actual Jacobian-free evaluation, the linearization point has to be perturbed, the residual is evaluated, and the finite difference needs to be taken. We propose to merge the two vector-update steps into the loop over cells and perform them on ranges of the vectors only when the value is needed for the cell integral (*pre*) and once contributions from all cells have been added to an index (*post*), as also has been done to accelerate conjugate-gradient solvers [57] and additive Schwarz solvers [58]. In order to minimize the memory footprint, we perturb the vector containing the linearization point and revert the modifications afterwards. The resulting *pre* and *post* operations are as follows:

$$\underline{\text{pre}}: u_i^{\text{lin}} \leftarrow u_i^{\text{lin}} + \beta p_i, \quad \underline{\text{post}}: \begin{cases} v_i \leftarrow (v_i + r_i^{\text{lin}})/\beta \\ u_i^{\text{lin}} \leftarrow u_i^{\text{lin}} - \beta p_i, \end{cases}$$

which are run interleaved with $v \leftarrow \mathcal{F}(u^{\text{lin}})$. This implies 6 additional floating-point operations and – under the assumption that the vector entries are still in cache – one extra write operation per DoF besides those in Table 1. In order to obtain the parameter β , our current implementation computes the l_2 -norm of the source vector, implying, in addition, one global vector reduction step with the associated data access before the cell loop.

4.3. Block preconditioner

In the following, we propose a preconditioner of the Jacobian (7). The preconditioner is needed when we apply the Jacobian directly or via a finite-difference approach. In the current work, we consider incomplete LU factorizations that are of block-Jacobi type across MPI processes, in short, ILU-based block preconditioners. In order to compute an ILU, we need access to the entries of the underlying sparse matrix. To avoid the cost for memory access, we propose to set up ILU for the Cahn–Hilliard block and a single Allen–Cahn block, applying the

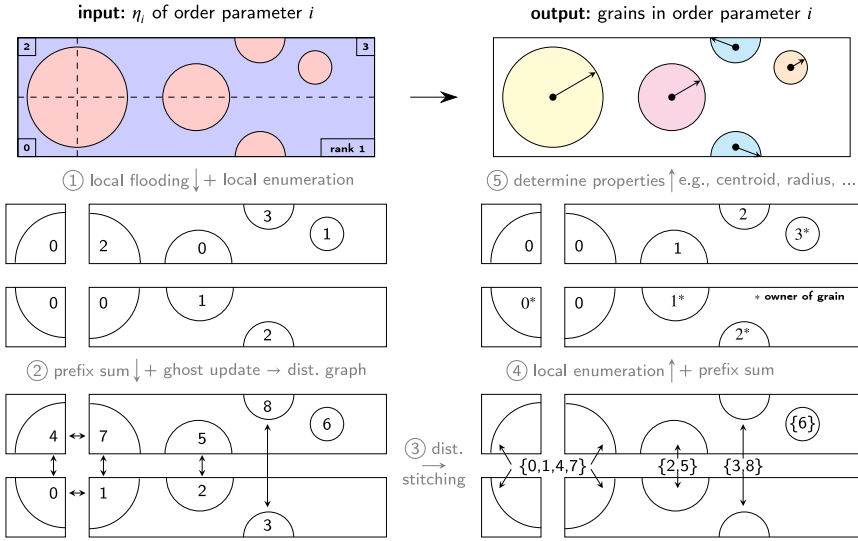


Fig. 9. Visualization of the fully distributed determination of grains within the same order parameter, starting with a discrete concentration field η_i . The results are (1) the grain index of each locally owned cell and (2) grain properties, e.g., centroid and bounding radius. In the example, the mesh is partitioned among four processes. For the sake of simplicity, the mesh is not shown and isocontours in η_i are perfectly circular. Grain indices are constant within cells. Steps ③ and ④ can be interpreted as a distributed connected-component algorithm that returns, for each node, a component number.

latter to each block, as indicated in the rightmost panel of Fig. 3. The setup is of constant complexity $\mathcal{O}(N_p)$, and the application naturally has a linear complexity $\mathcal{O}(N_c N_p)$. The runtime cost of the preconditioner applications can be reduced if the application is *batched* in a matrix-multiplicator form, thus only loading the entries of the factorized matrix once. To be able to reuse the ILU instance for each Allen-Cahn block, we take the maximum of the term $(v_i, L\partial^2 f / \partial \eta_i^2 u_i)$ in (7c) over all order parameters.

Remark. We use an ILU for the two-by-two Cahn-Hilliard block. For better performance with larger time steps or higher diffusivity, physics-based preconditioners, like the one in [59], might reduce the iteration counts. We defer the investigation of such preconditioners to future work, since the Cahn-Hilliard block is independent of the number of order parameters and, in our experiments, is not the bottleneck ($2 \ll N_c$).

4.4. Fully distributed grain tracking and remapping

In the following, we propose a fully distributed version of the grain-tracking algorithm, which we need to guarantee a non-conflicting assignment of the order parameters according to the invariant described in Fig. 2. The algorithm is inspired by reference [17], which presents the grain-tracking implementation in MOOSE. The procedure (1) detects grains in each order parameter, based on the discrete values of η_i , (2) checks for conflicts within each order parameter, and (3) reassigns the grains with conflicts to existing or new order parameters, which involves also the copying of data in the solution vectors.

Our implementation is heavily graph-based and works only on locally owned cells, allowing the algorithm to scale to large problem sizes.

4.4.1. Grain detection

For each order parameter η_i , each process runs a flooding algorithm [60] on locally owned cells to identify all agglomerations of cells with $\eta_i > \eta_{lim}$ ($\eta_{lim} = 0.01$ in our simulations). An agglomeration either forms a whole grain itself or is a part of a grain if it is located at internal boundaries between the neighboring processes. The challenging task is

to match agglomerations related to grains stretched over multiple processes. For this purpose, we propose the following algorithm, which is inspired by the unique enumeration of DoFs in the context of FEM [61] and is visualized in 5 steps in Fig. 9.

Initially, we locally enumerate all identified agglomerations and give them globally unique indices via a parallel prefix sum (steps 1 and 2 in Fig. 9). Once each agglomeration has a unique index, we communicate that information by a ghost-value update so that processes know the agglomeration index of each ghost cell. This information is enough to build a distributed graph with nodes being agglomerations and edges being connections determined based on the information from the ghost cells. By determining *all connected components*, we get all agglomerations (and cells) that make up a grain (distributed stitching, step 3 in Fig. 9). Finally, we give each grain a unique index and determine properties, like centroid or bounding radius and—optionally—force or torque (see Section 7.2), via parallel reduction operations (steps 4 and 5 in Fig. 9).

Note: The described algorithm also works for systems with periodic boundary conditions if it is possible to access ghost values across periodic boundaries (see the blue grain in Fig. 9). In addition, special care has to be taken during determination of properties, e.g., the grain centroids.

4.4.2. Grain tracking

After the grains have been detected for the current configuration, they need to be matched to those from the previous one. This is required to ensure no new grains have appeared, which would be an abnormal, not physically admissible behavior in the case of sintering. In order to determine, for a grain, the closest grain from the previous configuration, we build an R-tree, which allows a fast query.

4.4.3. Grain reassignment

Once all the grains have been identified and matched between the current and previous configurations, they are checked for collisions in a safety region $r'_i = r_i + 0.05r_i$. If the safety regions of any two grains $i \neq j$ in the same order parameter overlap, we need to reassign one of

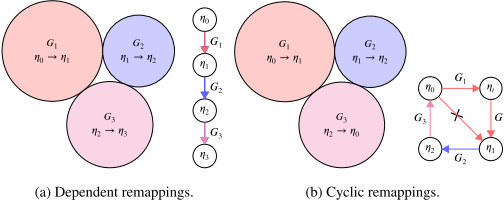


Fig. 10. Special remapping cases. Grain packings and remapping graphs.

them, since they might come into conflict in the next time steps. For reassigning, we use two algorithms.

A *greedy algorithm* checks, for a conflicting grain pair from order parameter i , whether any grain from all other order parameters $i \neq j$ would have a conflict with one of the two grains being currently under investigation. If not, one of the conflicting grains can be moved there, resolving the conflict. If no such order parameter could be found, the conflicting grain has to be assigned to a newly created order parameter or a more involved algorithm has to be used. In order to check whether a grain is in conflict with any grain in an order parameter, we use an R-tree data structure to accelerate the query.

Alternatively to the greedy algorithm, we use an algorithm based on *graph coloring*. In this case, grains are the nodes and conflicts are the edges. The graph coloring gives the minimum number of colors, which we interpret as the new order parameters. This approach, however, does not guarantee that grains keep their current order parameter so that remapping might become overly expensive even in the case of slight topological changes.

4.4.4. Grain remapping

The previous step has assigned grain G_i to an order parameter j : $\eta_j \rightarrow \eta_{j'}$ with possibly $j \neq j'$ so that grains in the same order parameter do not have any conflicts. Now, values corresponding to grain i in the current and previous solution vectors have to be transferred from the block corresponding to order parameter j to the one of j' , without overwriting the values in the destination vector that still need to be moved.

Let us consider the two configurations in Fig. 10, each aiming to reassign three grains. While the remapping can be done straightforwardly with the sequence $(G_3: \eta_2 \rightarrow \eta_3)$, $(G_2: \eta_1 \rightarrow \eta_2)$, $(G_1: \eta_0 \rightarrow \eta_1)$ in the first case, this is not possible in the second case due to cyclic dependencies. Here, we need to introduce a temporary vector and can run the remapping, e.g., with the following sequence: $(G_1: \eta_0 \rightarrow \eta_t)$, $(G_3: \eta_2 \rightarrow \eta_0)$, $(G_2: \eta_1 \rightarrow \eta_2)$, $(G_1: \eta_t \rightarrow \eta_1)$, where η_t corresponds to the temporary vector.

In order to automatically detect and resolve possible remapping issues, we construct a directed graph containing all necessary remappings. We run a depth-first search algorithm to determine all connected nodes, i.e., grain-remapping operations that need to be serialized. If a cluster of connected nodes does not contain cycles, we can schedule the reassignment in topological order. However, if a cycle is detected, a dummy node corresponding to the temporal vector is introduced to break the cyclic dependency. For performance purposes, the described remapping procedure is applied within a single cell loop, based on cached resolved graphs.

Note: In our implementation, we are performing the graph/tree algorithms in the cases of grain stitching, matching, reassignment and remapping redundantly on each process, using either `boost.graph/boost.geometry` [62] or `Zoltan` from the `Trilinos` library [63]. This implies that 0D grain information has to be replicated on each process. An extension to distributed graphs is possible but is deferred to future work to enable packings with more than 10k grains.

4.5. Solver configuration

We conclude this section by summarizing the ingredients of our solver. We refer to the configuration of the solver as *default configuration* if the Jacobian is evaluated in a matrix-free way and all order parameters are considered on the cell level (Algorithm 1). When different variants are investigated, these are labeled accordingly.

4.5.1. Nonlinear and linear solver

In order to solve the nonlinear system (5), we use a Newton solver with cubic line search from the `NOX` package from the `Trilinos` library [63]. This nonlinear solver is run until the l_2 -norm of the residual has been decreased by $\epsilon_{\text{nonlin}} = 10^{-5}$ and the Jacobian is solved with `GMRES` with a rather coarse relative tolerance of $\epsilon_{\text{lin}} = 10^{-2}$, in line with Brown [35] (see also Supplementary Material S3.2). The choice $\epsilon_{\text{nonlin}} = 10^{-5}$ as default tolerance for our nonlinear solver is the result of experiments over a wide range of tolerances and benchmark applications, ensuring sufficient accuracy in the mass conservation and the free energy of the discretized solution. A more detailed discussion on this matter with the elementary assessment of the conservation error in view of [64] can be found in Supplementary Materials S3.1 and S3.3. The preconditioner for `GMRES` is set up at the beginning of each time step once and is reinitialized between Newton iterations only if the number of linear iterations at a particular single Newton step exceeds the value of 50. We use the `ILU` implementation from the `Ifpack` package from the `Trilinos` library [63].

4.5.2. Time stepping

We use `BDF2` with adaptive time steps. We increase the time-step size by 20% if the number of nonlinear iterations is less than 5 and the accumulated number of linear iterations is less than 100, i.e., less than 20 linear iterations per nonlinear iteration on average. We consider a time step as failed if the nonlinear solver needs more than 10 iterations or the linear solver more than 100 iterations in total. If the nonlinear solver has failed, we decrease the time-step size by 50% and rerun the time step, based on the old converged solution. The described time-stepping heuristic is conservative, since time-step sizes are not increased for challenging configurations, reducing the number of time steps that need to be repeated. Nonetheless, the obtained time-step sizes are quite large overall, leading to significant pressure on the linear solver and its preconditioner.

4.5.3. Initial mesh generation

The initial geometry is defined as a list of spherical particles. The definition of potentially overlapping particles can be supplied for the phase-field simulations from the preliminary `DEM` early-stage-sintering analysis [4]. The computational domain is then constructed based on a bounding box over all particles with a boundary padding $0.5r_{\text{max}}$, where r_{max} is the radius of the largest particle in the packing.

We choose the diffuse interface thickness a priori as $w = 0.1r_{\text{avg}} \dots 0.15r_{\text{avg}}$, where r_{avg} is the average radius of particles in the packing. The choice of w then defines the free-energy properties $A = (12\gamma_s - 7\gamma_{gb})/w$, $B = \gamma_{gb}/w$, $\kappa_c = 3w(2\gamma_s - \gamma_{gb})/4$, $\kappa_\eta = 3w\gamma_{gb}/4$, based on a limit case analysis [24,30,31] with the physical surface γ_s and the grain boundary energies γ_{gb} , both usually given by the physics of the material of interest. Depending on the problem size and the desirable accuracy in capturing the interface motion, the thickness of the latter is discretized by 1–4 cells, giving the finest mesh size h_c desired only at the interface itself. The mesh is obtained by locally refining a coarse quad-/hex-only mesh. This mesh is constructed in such a way that cells have a good aspect ratio and the value h_c is approximately obtained in each direction by local refinement. For the latter, we use a forest-of-trees approach where cells are recursively replaced by 2^d child cells and rely on `p4est` [61,65]. The proposed strategy generates meshes with on average less than 10k and 100k scalar DoFs per particle for 2D and 3D simulations, respectively.

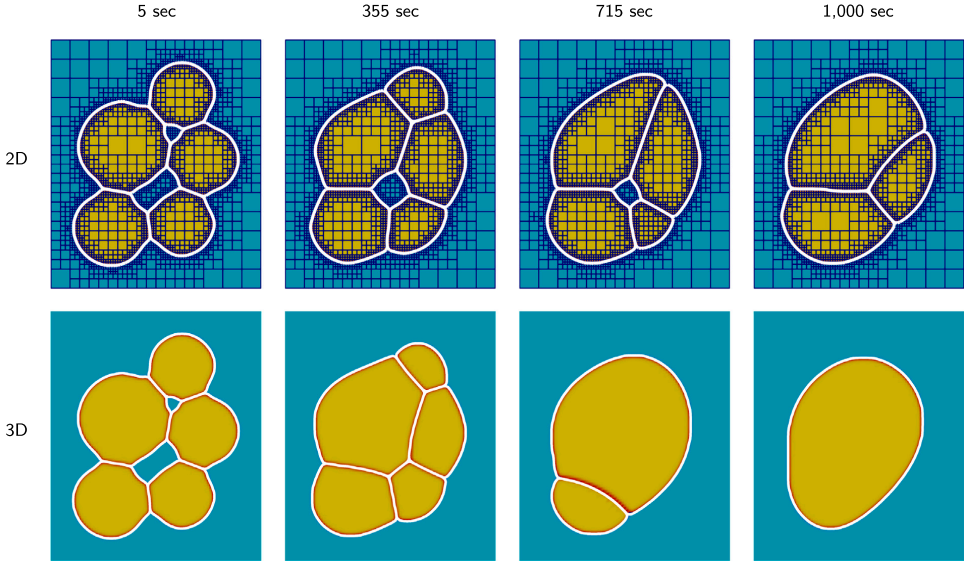


Fig. 11. 5 particles: microstructures obtained with MOOSE and with our code during simulations of the 5-particle sintering. The plots show $\sum_i \eta_i^2$ computed with MOOSE, and the white lines display the isosurfaces constructed for each $\eta_i = 0.5$ with `hpsint`.

4.5.4. Adaptive mesh refinement

The AMR algorithm is triggered every 10th time step or when the mesh quality has deteriorated in relation to the values of η_i ; we only allow $|\max(\eta_i) - \min(\eta_i)| < 0.5$ for all cells. We coarsen cells if they are not close to the boundary of a particle, which is identified by $0.05 < \eta_i < 0.95$. At boundaries themselves, we keep the mesh fine throughout the simulation. The number of maximum/minimum refinements is a runtime parameter and is chosen depending on the material properties.

4.5.5. Grain tracking

We run the grain-tracking algorithm every 25 s of the simulated time or according to the same quality criterion as in the case of AMR described in Section 4.5.4. The initial distribution of grains over the order parameters is performed by running the graph-colorization algorithm that is also used at the reassignment step (see Section 4.4.3).

5. Numerical results

In the following, we present numerical results obtained with the developed solver, mainly to verify it from the sintering-physics point of view. We keep the discussion short and regard this section as an introduction into the setups of the performance analyses in Section 6.

5.1. 5-Particle case

We first analyze the 5-particle geometry depicted in Fig. 1; the centers and radii of the particles are listed in Table 2. The cubic computational domain of size $42 \times 50 \times 27$ in 3D is generated as described in Section 4.5.3. The free energy constants are defined as $A = 16$, $B = 1$ and the energy barriers are set to $\kappa_c = 1.0$, $\kappa_f = 0.5$. These values render the diffuse interface thickness $w \approx 1.0$. The following diffusion mobilities are chosen: $M_{vo} = 10^{-2}$, $M_{va} = 10^{-10}$, $M_s = 4.0$, $M_{gb} = 0.4$. The grain-boundary mobility is set to $L = 1.0$. This is a common set of parameters used in multiple works [22,25,30]. The packing is analyzed in 2D and 3D for the period up to $t_{\text{end}} = 1,000$ seconds. For this benchmark, we use a tighter tolerance $\epsilon_{\text{lin}} = 10^{-5}$

Table 2

5 particles: initial locations and radii of the particles.

| ID | x | y | z | r |
|----|--------|--------|---|-----|
| 1 | 7.5 | 7.5 | 0 | 7.5 |
| 2 | 10 | 23.81 | 0 | 9.0 |
| 3 | 21.464 | 8.5 | 0 | 6.5 |
| 4 | 25.8 | 21.285 | 0 | 7.0 |
| 5 | 21.583 | 34.109 | 0 | 6.5 |

for the linear solver, matching the default value in MOOSE, in order to make sure that the coarser value of 10^{-2} , which we normally use as stated in Section 4.5.1, does not degrade the performance of MOOSE and, that way, ensures the comparison of the two codes. Note that no units are given except for time, since the geometry, energy and mobility properties for the current problem are defined as dimensionless.

The aim of these simulations is to compare the results with an alternative, well-established implementation [26,30,66] of the same phase-field model available in project `Crow`,⁵ which is based on the MOOSE framework [11]. `Crow` reuses most of the parts of the phase-field module of MOOSE, including the kernels that provide the mobility and free-energy terms as materials required by this particular sintering model. We carefully tuned the AMR settings, absolute and relative tolerances, iteration thresholds, implicit time-integration properties of both solvers such that the numbers of DoFs and of linear and nonlinear iterations are comparable in both codes. Fig. 11 shows meshes at different times in the 2D case. Other settings were set to be optimal for the MOOSE solver according to its documentation.⁶ For instance, the preconditioned JFNK solution strategy provided by SNES from PETSc [34] was used and the parallel ILU implementation provided by the `Euclid` library from `hypre` [67] was chosen as preconditioner for the GMRES linear solver.

⁵ <https://github.com/SudiptaBiswas/Crow>.

⁶ <https://mooseframework.inl.gov/source/systems/NonlinearSystem.html>, https://mooseframework.inl.gov/modules/phase_field/Solving.html.

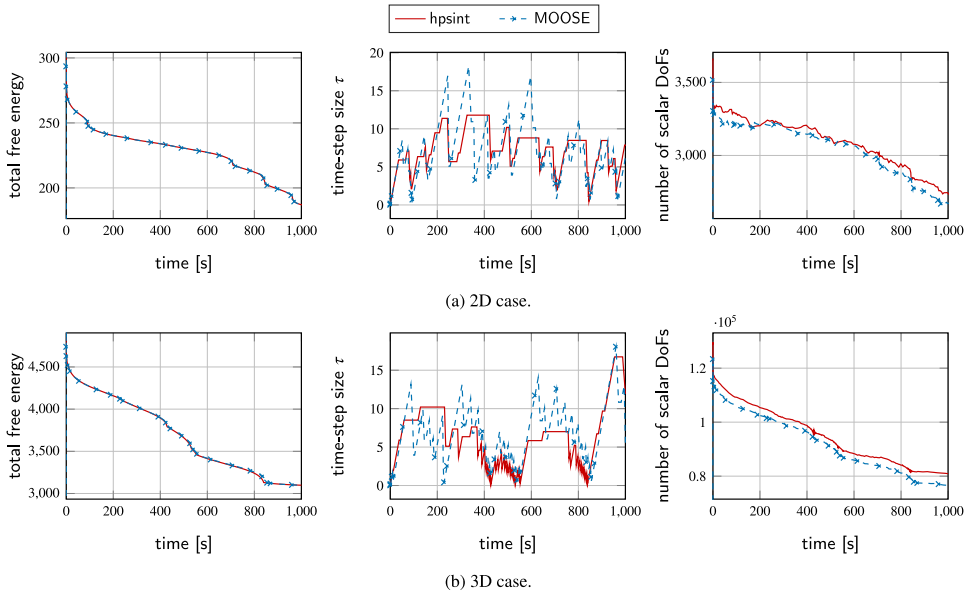


Fig. 12. 5 particles: comparison of total free energy, time-step size, and number of scalar DoFs over the simulation time.

Table 3
5 particles: performance metrics of the 2D and 3D simulations in hpsint and MOOSE. Scalar DoFs are given.

| | 2D | | 3D | |
|----------------|--------|-------|---------|---------|
| | hpsint | MOOSE | hpsint | MOOSE |
| Wall time [s] | 164.8 | 704.4 | 1072 | 69,898 |
| max τ | 12.22 | 18.22 | 16.74 | 18.05 |
| # time steps | 190 | 223 | 303 | 241 |
| # DoFs initial | 3604 | 3504 | 128,700 | 123,209 |
| # DoFs final | 2751 | 2673 | 80,909 | 76,697 |

As can be seen in Fig. 11, the microstructures obtained for both solvers are identical for the 2D and 3D cases. For this purpose, the quantity $\sum_i \eta_i^2$ obtained in MOOSE is compared with the isolines constructed for each $\eta_i = 0.5$. For the 3D case, the corresponding plots are shown at the cross-section plane $z = 0$. As expected and previously demonstrated in [12], the 3D case exhibits faster microstructure evolution than the 2D simulations for the same material parameters. This is due to the added driving force from a second curvature term in 3D and also due to qualitative topological differences: for comparable geometries, those pores that are closed in 2D are usually open in 3D.

Fig. 12 shows the plots of the total free energy (2). The curves obtained in both codes reveal a gradual reduction of the total free energy. The comparison of its components (bulk and interfacial) along with some other model metrics is shown in Supplementary Material S3.1. Additionally, Fig. 12 presents the evolution of the total number of DoFs and the time-step size during the numerical simulations. In distinct contrast to MOOSE, our code does not always attempt to increase τ , since we target a predefined number of linear and nonlinear iterations; once that limit is exceeded, the time-step size is not enlarged. Confirming the design of the experiment, the numbers of DoFs in both codes are found to vary similarly.

The solver-related metrics⁷ are summarized in Table 3. In particular, we show timings for the MOOSE-based and our implementation. The wall times for both codes were obtained in separate runs with output disabled. A speedup of 4.3/65.3 is visible for 2D and 3D. The numbers are primarily intended to show the significance of the proposed optimizations, as the comparison to the MOOSE code has to remain qualitative: even though its settings have been tuned to provide the maximum performance, we believe that better performance might be possible for developers with deeper insight into MOOSE.⁸

5.2. 332-Particle case

As a second validation benchmark, we compare the results of our simulations with those presented in [7]. The authors of that paper kindly provided us the initial packings used for their numerical analysis. The original focus of the discussion in [7] was on the influence of the rigid body motions on the sintered microstructures. Even though our code also implements the advection terms (see Section 7.2), we intentionally perform the comparison of results without advection and, thus, rigid body motions. Due to this reason, we run the sintering simulation of the packing containing 332 particles, for which a number

⁷ The simulations were executed on a single node of the cluster at Helmholtz-Zentrum Hereon (dual-socket 24-core 2.1 GHz Intel Xeon Scalable Platinum 8160 processor; Skylake), using 24 processes for 2D and 48 for 3D.

⁸ A small benchmark code based on deal.II, which evaluates (8) for the residual without exploiting the structure of the shape functions (tensor-product, same for each component, ...) and does not vectorize over cells, results in a $\approx 30\times$ lower throughput. The code path is not specialized for the evaluation of the residual but for easy-to-read and generic assembly of a sparse matrix, which normally has to be computed less frequently so that performance optimizations are not as crucial as if it would be done in each linear iteration. Since LibMESH [68], the FEM backend of MOOSE, evaluates the residual similarly to the implementation of the slow path in deal.II, we are confident that the presented performance comparison with MOOSE is qualitatively reasonable.

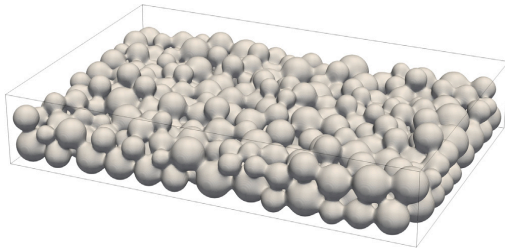


Fig. 13. 332 particles: the final configuration after sintering for 3.47 h.

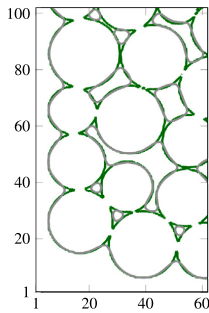


Fig. 14. 332 particles: contour lines on the $z = 10 \mu\text{m}$ plane taken at $t = 0$ (green) and 3.47 h (gray).

of metrics is available in [7], referred to as “Case 3” in that publication. The numerical implementation in [7] is based on finite differences on a uniform grid and uses explicit time integration.

We use the same material parameters: the free-energy constants are set to $A = 32$, $B = 8$, $\kappa_c = 0.4$, $\kappa_\eta = 0.2$, and the diffusion mobilities are defined as $M_{vo} = 10^{-2}$, $M_{va} = 10^{-3}$, $M_s = 10.0$, $M_{gb} = 1.0$. The energy properties lead to the diffuse interface thickness $w \approx 0.18$. The same grain-boundary mobility $L = 100$ as in [7] is used, ensuring that the grain-growth effects do not dominate in the regime examined. Similarly to the original publication, we also solve the dimensionless form of the phase-field equations, using the same scaling parameters: $l = 5 \mu\text{m}$, $M_0 = 10^{-12} \text{ cm}^2/\text{s}$, and $t_{\text{ref}} = l^2/M_0 = 2.5 \times 10^5 \text{ s}$. Given the physical sintering time $t_{\text{physical}} = 3.47 \text{ h}$, the simulation time is $t_{\text{end}} = 0.05$. The initial time-step size is chosen as $\tau_{\text{initial}} = 10^{-3}$.

The computational domain is defined by the bounding box with dimensions of $-1.5 < x < 41.5$, $-1.5 < y < 61.5$, $-1.5 < z < 13$ such that the distance between each particle and the domain boundary is at least 1. The coarse mesh consists of $86 \times 126 \times 29$ cubic cells, and we perform two local-refinement steps to obtain the minimal cell size $h_e = 0.125$, which is comparable to the grid spacing $\Delta x = 0.1$ from [7]. This results in $\approx 6 - 7$ million scalar DoFs in our case in contrast to the ≈ 34 million points ($420 \times 620 \times 130$ cells) of the uniform mesh in [7]. A typical simulation of this test case with our code needs only 14 time steps when using the time-stepping strategy described in Section 4.5.2 and, on 16 nodes of SuperMUC-NG, runs in about 5 min (not counting the time required for generating the output).

In this particular simulation, we employ the conventional surface-mobility term [22] instead of the one from (4) in order to achieve a better agreement with the original results from [7]. To alleviate the arising convergence issue due to the non-smooth coefficient, we apply the Jacobian-free formulation (6).

In order to postprocess and visualize the results, the physical domain is used. Fig. 13 presents the final configuration of the packing,

Table 4

Numbers of order parameters, cells and scalar DoFs for 51–10,245-particle packings.

| N_{grains} | 51 | 102 | 212 | 316 | 603 | 1370 | 3760 | 6140 | 10,245 |
|--|-----|-----|------|------|------|------|-------|-------|--------|
| N_{op} | 9 | 10 | 10 | 11 | 10 | 11 | 11 | 11 | 11 |
| N_{cells} [$\times 1\text{e}6$] | 3.0 | 5.7 | 10.4 | 18.8 | 34.0 | 53.5 | 153.2 | 325.5 | 589.1 |
| N_{DoFs} [$\times 1\text{e}6$] | 3.4 | 6.4 | 11.7 | 20.9 | 37.9 | 59.5 | 170.2 | 360.8 | 651.7 |

and Fig. 14 shows the microstructure view at the cross-section plane $z = 10 \mu\text{m}$. Visually, both images are in close agreement with the analogous Figs. 7(a) and (c) presented in [7], without noticeable differences.

To perform a quantitative comparison, we also compute the microstructure metrics

$$\text{solid-volume fraction} \quad \int_{\Omega} c \, d\Omega, \quad (9a)$$

$$\text{surface area} \quad \sum_i \int_{\Gamma_i} 1 \, d\Gamma_i \, w, \quad \Gamma_i = \{p \in \Omega \mid \eta_i(p) = 0.5\}, \quad (9b)$$

$$\text{grain-boundary area} \quad \sum_i \int_{\Gamma_i} 0.5 \, d\hat{\Gamma}_i \, w, \quad \Gamma_i = \{p \in \Gamma_i \mid \eta_i \eta_j > 0.14 \, \exists j \neq i\} \quad (9c)$$

in the control volume $5 < x < 195 \mu\text{m}$, $5 < y < 295 \mu\text{m}$, $2.5 < z < 35 \mu\text{m}$ as in [7]; the corresponding graphs are shown in Fig. 15. The curves do not match perfectly. For the solid-volume fraction, the mismatch is, in fact, negligible given the scale of the y -axis. For the remaining two quantities, the difference is more tangible but still does not exceed 3% for the surface and 10% for the grain-boundary areas, respectively, and, is most probably related to the details in the implementation of the postprocessors, which extract the isosurfaces. Despite of the differences in the values, the slopes of the curves are in a very good agreement, meaning that the dynamics of the sintering processes in both simulations are the same.

6. Performance

In the following, we analyze the performance of the implementation of our solver in detail from a holistic point of view. This is needed because of the multi-faceted challenges of the solution of sintering processes, in which it is not enough to optimize and analyze the (matrix-free) linear operator evaluation, as we did in Section 4.2. For example, the preconditioner choices implied by the matrix-free solver design need to be assessed in terms of the total solution time.

This section is divided into two parts. We start by studying the performance of the solver for a moderate number of grains (51; far left in Fig. 16), where the focus is on the influence of different variations of the solver. Next, we analyze the parallel scalability of the code by increasing the number of particles up to 10k and using up to 50k processes on a supercomputer.

All packings considered in the current section are cubic and shown in Fig. 16. Bounding boxes of different sizes are used to extract smaller packings containing 51...6,140 particles from the largest one having 10,245 grains within a control volume of size $1399 \times 1400 \times 1318 \mu\text{m}^3$. The latter has been obtained by the preliminary DEM simulations performed with the package Yade [69], following the procedure proposed in [4], which was designed to deliver an isotropic initial configuration with irregular, realistic distributions of particles having relatively low porosity. The particle-size distribution of the largest packing is shown in Fig. 16.

In contrast to Section 5.1, larger powder particles are considered in this section. For this reason, the energy parameters are defined as $A = 4.35$, $B = 0.15$, $\kappa_c = 9.0$, $\kappa_\eta = 1.79$. These values yield a thicker diffuse interface with $w = 4$ for the surface and grain-boundary free-energy values $\gamma_s = 1.8$ and $\gamma_{gb} = 0.6$. The diffusion mobilities are defined as $M_{vo} = 10^{-2}$, $M_{va} = 10^{-3}$, $M_s = 4.0$, $M_{gb} = 0.4$, and the grain-boundary mobility is set to $L = 1.0$. The initial number of divisions per interface thickness is chosen as $\zeta = 3$. The initial numbers of scalar

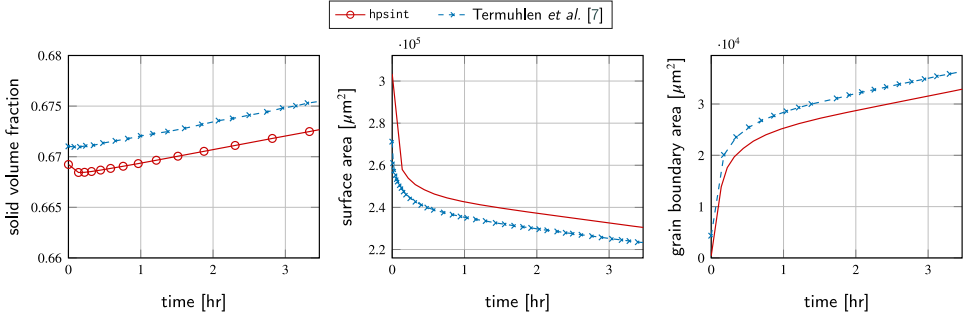


Fig. 15. 332 particles: comparison of relevant microstructure metrics according to (9).

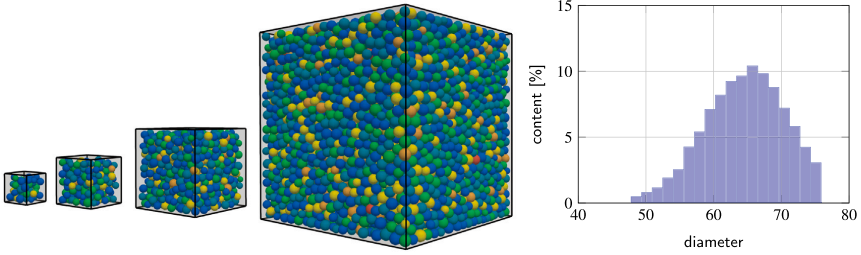


Fig. 16. Left: Particle packings considered for the performance study in Section 6: 51, 212, 1037, 10,245 particles (not shown: 102, 316, 603, 3076, 6140 particles), with colors indicating the order parameters. Right: Distribution of particle diameters for the packing containing 10,245 particles.

Table 5
51 particles (up to $t = 500$): Comparison of different solver configurations.

| Configuration | τ_{\max} | N_T | \bar{N}_N | \bar{N}_L | \bar{N}_R | N_T^f | N_L^f | T | \bar{T}_J | $\bar{T}_{P,\text{setup}}$ | $\bar{T}_{P,\text{apply}}$ | \bar{T}_R |
|--------------------------|---------------|-----------|-------------|-------------|-------------|-------------|-------------|--------------|--------------|----------------------------|----------------------------|--------------|
| Default | 19.78 | 51 | 4.6 | 25.3 | 1.2 | 0.0% | 0.0% | 463.2 | 0.027 | 1.279 | 0.013 | 0.028 |
| JF (Jacobian-free) | 19.78 | 51 | 4.6 | 25.7 | 1.2 | 0.0% | 0.0% | 444.7 | 0.023 | 1.277 | 0.013 | 0.028 |
| cut-off | 19.78 | 51 | 4.6 | 25.4 | 1.2 | 0.0% | 0.0% | 353.9 | 0.009 | 1.269 | 0.013 | 0.009 |
| JF + cut-off | 19.78 | 51 | 4.6 | 25.3 | 1.2 | 0.0% | 0.0% | 356.1 | 0.008 | 1.272 | 0.013 | 0.009 |
| tensorial | 16.48 | 73 | 4.4 | 16.4 | 1.2 | 4.1% | 5.7% | 551.4 | 0.040 | 1.253 | 0.014 | 0.036 |
| tensorial + JF | 19.78 | 59 | 4.5 | 19.5 | 1.2 | 1.7% | 2.2% | 450.8 | 0.028 | 1.239 | 0.014 | 0.032 |
| tensorial + cut-off | 16.48 | 60 | 4.6 | 19.8 | 1.2 | 0.0% | 0.0% | 355.3 | 0.010 | 1.192 | 0.013 | 0.010 |
| advection + JF | 19.78 | 56 | 4.4 | 24.4 | 1.2 | 1.8% | 2.7% | 523.0 | 0.025 | 1.262 | 0.014 | 0.065 |
| advection + JF + cut-off | 19.78 | 56 | 4.5 | 24.4 | 1.2 | 1.8% | 2.6% | 434.1 | 0.010 | 1.238 | 0.014 | 0.025 |

τ_{\max} : maximal time-step size, N_T : number of time steps, \bar{N}_N : average number of nonlinear iterations per time step, \bar{N}_L : number of linear iterations per nonlinear iteration, \bar{N}_R : number of residual evaluations per nonlinear iteration, N_T^f : fraction of repeated time steps, N_L^f : fraction of linear iterations of repeated time steps, T : total runtime in seconds, \bar{T}_J : average time for application of Jacobian, $\bar{T}_{P,\text{setup}}$: average setup time of the preconditioner, $\bar{T}_{P,\text{apply}}$: average time for application of the preconditioner, \bar{T}_R : average time for evaluation of the residual

degrees of freedom generated for each of the packings by using such settings are shown in Table 4.

For time marching, the BDF2 scheme is used with the initial time-step size $\tau = 0.1$ and its growth is limited by the maximum value $\tau_{\max} = 100$.

6.1. 51 Particles

We start with the investigation of a 51-particle packing. All experiments are run on 4 Intel Cascade Lake Xeon Gold 6230 nodes with a total of 160 processes. In this section, each experiment has been run once and we average the results over at least 50 time steps, 230 nonlinear iterations, and 5200 linear iterations.

6.1.1. Default configuration

The line “default” in Table 5 shows the solver statistics obtained over a complete simulation up to $t = 500$ (τ_{\max} , N_T , \bar{N}_N , \bar{N}_L , \bar{N}_R , T , \bar{T}_J , $\bar{T}_{P,\text{setup}}$, $\bar{T}_{P,\text{apply}}$, \bar{T}_R ; see Section 3). The maximum time-step size achieved is 19.78. The average numbers of nonlinear and linear iterations are about 4.6 and 25.3, respectively. These numbers lie in the expected range, given the control parameters of the adaptive time stepping according to Section 4.5.2.

Similarly, Table 7 shows the data for $t = 15,000$. One can see that the nonlinear solver fails to converge for certain time steps, requiring a second attempt with decreased time-step size. However, their number and the resulting wasted (non)linear iterations are rather small and do not exceed 6%.

For the long simulation, Fig. 17 shows the time share of different parts of the code. 80% of the time are spent on solving the Jacobian. From this linear solver process, 46% are spent on applying the

Table 6

51 particles (up to $t = 500$): Comparison of different preconditioners. The default configuration uses ILU + sILU (max).

| Preconditioner name | τ_{\max} | N_T | \bar{N}_N | \bar{N}_L | \bar{N}_R | N_T^f | N_L^f | T | \bar{T}_J | $\bar{T}_{P,setup}$ | $\bar{T}_{P,apply}$ | \bar{T}_R |
|---------------------|---------------|-----------|-------------|-------------|-------------|-------------|-------------|--------------|--------------|---------------------|---------------------|--------------|
| Default | 19.78 | 51 | 4.6 | 25.3 | 1.2 | 0.0% | 0.0% | 449.6 | 0.024 | 1.265 | 0.013 | 0.026 |
| ILU ^a | 23.74 | 56 | 4.4 | 22.3 | 1.2 | 1.8% | 6.3% | 2495.0 | 0.022 | 22.446 | 0.140 | 0.024 |
| ILU + ILU | 23.74 | 54 | 4.5 | 21.9 | 1.2 | 1.9% | 6.3% | 2141.0 | 0.026 | 19.037 | 0.128 | 0.028 |
| ILU + bILU | 19.78 | 50 | 4.4 | 24.7 | 1.2 | 0.0% | 0.0% | 530.8 | 0.025 | 2.890 | 0.019 | 0.027 |
| ILU + sILU (none) | 5.52 | 112 | 4.3 | 29.9 | 1.2 | 0.0% | 0.0% | 1009.0 | 0.024 | 1.047 | 0.013 | 0.026 |
| ILU + sILU (avg) | 19.78 | 53 | 4.5 | 24.8 | 1.2 | 1.9% | 2.8% | 478.9 | 0.027 | 1.315 | 0.014 | 0.028 |

^a 5 compute nodes used due to increased memory consumption of the sparse matrix needed to set up the preconditioner.

Table 7

51 particles (up to $t = 15,000$): Comparison of different preconditioners. The default configuration uses ILU + sILU (max).

| Preconditioner name | τ_{\max} | N_T | \bar{N}_N | \bar{N}_L | \bar{N}_R | N_T^f | N_L^f | T | \bar{T}_J | $\bar{T}_{P,setup}$ | $\bar{T}_{P,apply}$ | \bar{T}_R |
|---------------------|---------------|-------------|-------------|-------------|-------------|-------------|-------------|-----------------|--------------|---------------------|---------------------|--------------|
| Default | 25.52 | 1043 | 4.8 | 26.0 | 1.2 | 3.5% | 4.6% | 10 240.0 | 0.027 | 1.117 | 0.014 | 0.027 |
| ILU + bILU | 27.43 | 990 | 5.3 | 27.4 | 1.5 | 2.5% | 5.3% | 12 710.0 | 0.025 | 2.785 | 0.019 | 0.029 |
| ILU + sILU (avg) | 24.57 | 1113 | 4.7 | 25.8 | 1.2 | 3.7% | 4.7% | 10 790.0 | 0.028 | 1.133 | 0.014 | 0.029 |

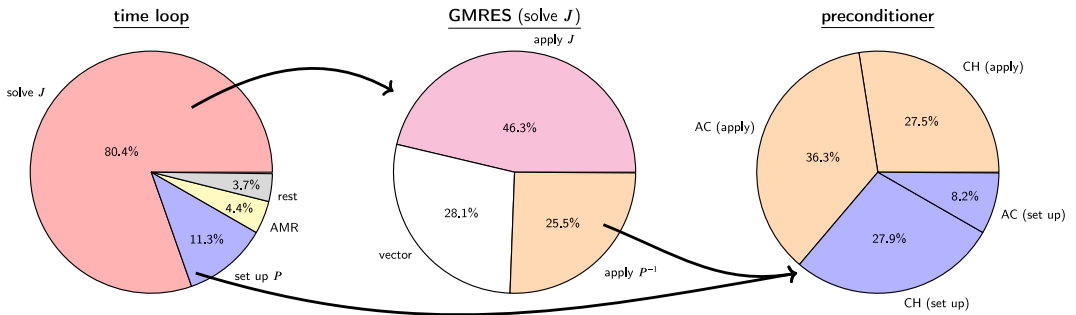


Fig. 17. 51 particles (up to $t = 15,000$): time share of different parts of the code.

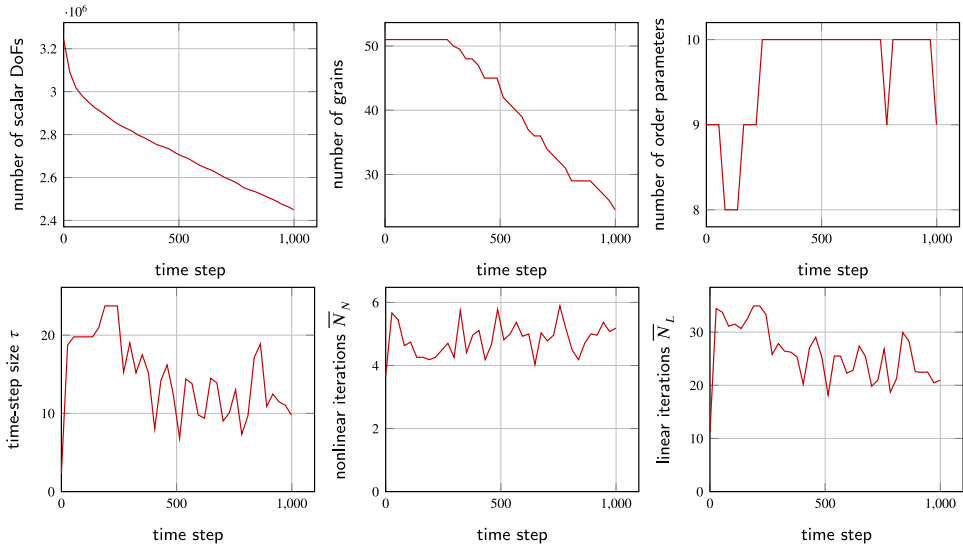


Fig. 18. 51 particles (up to $t = 15,000$): number of scalar DoFs, number of grains, number of order parameters, time-step size τ , nonlinear iterations \bar{N}_N , and linear iterations \bar{N}_L over time. The values are averaged over ranges of 30 time steps.

Jacobian, 26% on the application of the preconditioner and the rest on vector operations within GMRES. About 11% of the time is spent on setting up the preconditioner, which is dominated by the setup of the preconditioner for the 2×2 Cahn–Hilliard block. Fig. 18 shows relevant quantities (number of DoFs, number of grains, number of order parameters, time-step size, number of nonlinear and linear iterations) over time. The number of DoFs is decreasing during the simulation, which is correlated with the disappearance of grains. The number of order parameters does not exceed 10 due to the developed grain-tracking algorithm. On average, the number of order parameters is 9.5. The number of nonlinear and linear iterations is kept at a constant value by our strategy to increase/decrease time-step sizes and resetting the preconditioner. The time-step size shows a strong variation over time, primarily driven by the high dynamics when new necks are forming between particles.

6.1.2. Alternative preconditioners

In the following, we compare the proposed preconditioner (ILU + sILU (max)) with the following alternatives, motivated by the block sparsity pattern in Fig. 3. (1) ILU: set up ILU for the complete Jacobian; (2) ILU + ILU: set up ILU for the Cahn–Hilliard block and the Allen–Cahn block; (3) ILU + bILU: set up ILU for the Cahn–Hilliard block and for each scalar Allen–Cahn block, ignoring the coupling between the Allen–Cahn blocks. (4) In addition to taking the maximum of $(v_i, L\partial^2 f / \partial \eta_i^2 u_i)$ in ILU + sILU (max), we also consider ignoring this problematic term (none) and taking the average value over all order parameters (avg).

Table 6 evaluates the solver efficiency – for a simulation up to time $t = 500$ – for the different preconditioners. We note that the choice of the preconditioner does not influence the accuracy of the solution, which is primarily controlled by the nonlinear tolerance. It is clear that ILU and ILU+ILU are inherently expensive due to quadratic complexities in both setup costs and memory consumption. Considering only the blocks on the diagonal of the Allen–Cahn block (ILU+bILU) improves the situation without significantly affecting the convergence of the overall solver. In order to reduce the setup costs, the reuse of the preconditioner of one Allen–Cahn block helps. Our observation is that ILU+sILU (avg) and ILU+sILU (max) are able to cut down the setup costs of the preconditioner by a factor of two and the costs of applying by 30%, since the ILU instance has to be loaded only once due to the batched application.⁹ Overall, the performance can be improved by 20%. ILU+sILU (none), however, results in a significant drop in applicable time-step sizes, leading to an increase in the required overall solution time. For the best three preconditioners, Table 7 shows the results obtained for simulations up to $t = 15,000$, indicating that the scalar preconditioners are also applicable to later stages of sintering simulations and that the proposed variant ILU + sILU (max) is indeed the fastest overall.

6.1.3. Matrix-free vs. Jacobian-free evaluation

Table 5 (row “JF”) also evaluates the efficiency of the Jacobian-free approach. The number of linear iterations is growing moderately. In total, a matrix-free approach that evaluates the Jacobian exactly and a Jacobian-free approach result in similar solution times, with slight advantages in favor of the Jacobian-free approach due to lower costs of residual evaluation without loss of accuracy. This observation allows users to skip explicit linearization of the weak form as well as to avoid the implementation and its cumbersome performance optimization. Nevertheless, a fast parameter-free approach (we still need to identify β for the Jacobian-free case, see Section 4.2) might be crucial and useful in practice.

⁹ Please note that the costs of setting up the Cahn–Hilliard 2×2 block are now dominating.

6.1.4. Relevant grains on cells

Table 5 (rows “cut-off” and “JF + cut-off”) shows the impact of considering only relevant grains on a cell level. In the case of matrix-free and Jacobian-free operator evaluations, the costs can be cut down by a factor of 3.0 and 2.9, respectively. Considering that the operator evaluation only takes half of the overall runtime, the solver speedup is rather modest at 30% or 25%, respectively. These results are a strong motivation for adopting the cut-off strategy to other parts of the code in future work. For example, the design of a suitable variation of the chosen preconditioner is not straightforward without manually rewriting sparse matrix kernels. We refer interested readers to Supplementary Material S3.4 for the results of experiments regarding the influence of the cut-off tolerance on the model accuracy.

6.2. Scaling up to 10k particles

We conclude this section by presenting scaling results. For this purpose, we run packings with 51, 102, 212, 316, 603, 1037, 3076, 6140, and 10,245 particles on 8, 16, 32, 64, 128, 256, 512, and 1024 compute nodes on SuperMUC-NG. In each case, we perform the simulation for a fixed number of 10 time steps: the grain tracking is triggered once, at the first step. We run all experiments three times and report the best timings. The statistical distribution is close to the minimal time within a few percent.

Fig. 19 shows the scaling behavior of the most important ingredients for our solver. Overall, it is clear that the solver (linear solver and setup of preconditioner) scales well up to 512 compute nodes. The performance drops significantly when going to 1k nodes, which is related to an unexpected spike of setup costs of ILU in Trilinos.

Alternative plotting of the data in Fig. 20 indicates excellent parallel efficiency over large ranges in the case of the linear solver: for sufficiently large problem sizes per process, one can increase the number of processes by a factor of 16–32 with only a loss of 25% in parallel efficiency. Furthermore, the lowest times to solutions are reached for about 10k DoFs per process.

The cost of the grain-tracking algorithm increases linearly with the number of processes. This is related to the fact that the graphs are gathered during stitching. Since the costs are rather small compared to other parts of the code, we defer the investigation of the grain-tracking algorithm to future work, for which we plan to adopt algorithms based on distributed graphs.

7. Extensions

So far, we have considered a basic phase-field approach for simulating solid-state-sintering processes. More involved physical models involve additional computations (see Section 2.1). Since we do not assemble matrices, where one would only pay the cost of additional computations during assembly and the cost of the application of the matrix is independent of the depth of the physical model as long as the sparsity pattern is not changing, additional computations could limit the throughput in a matrix-free implementation. Also the choice of the physical model might influence the preconditioner selection. In the following, we investigate these points for two common physical extensions: (1) tensorial mobility and (2) advection terms (rigid body motions).

7.1. Extension 1: tensorial mobility

In the basic model (1), the mass fluxes are defined by the scalar mobility term (4). Despite its simplicity, this term is not the best choice if a rigorous treatment of the surface and the grain-boundary fluxes is of primary interest, as shown in [23]. In this case, the tensorial form

$$\mathbf{M} = M_{v0}\phi\mathbf{I} + M_{va}(1 - \phi)\mathbf{I} + 4M_s c^2(1 - c)^2 \mathbf{T}_s + M_{gb} \sum_{i=1}^N \sum_{j \neq i}^N \eta_i \eta_j \mathbf{T}_{ij} \quad (10)$$

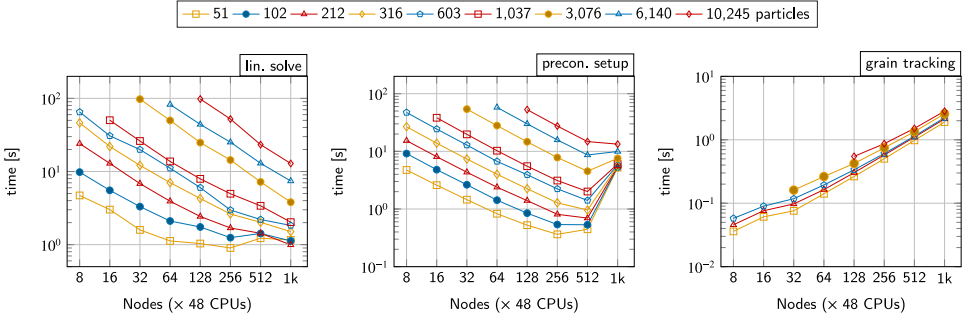


Fig. 19. Scaling of different and computationally most expensive parts of the solver for 51–10,245 particles and 10 time steps. Times are accumulated over all time steps and include both computations and communications. The data for “grain tracking” is only shown for a selected set of particles (51, 212, 603, 3076, 10,245 particles).

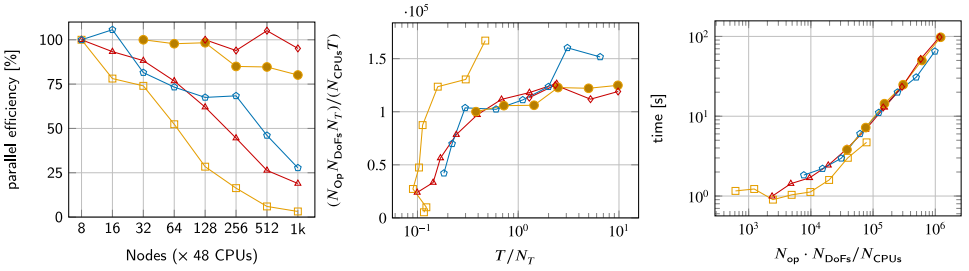


Fig. 20. Detailed scalability analysis of linear solver (see Fig. 19). The data is only shown for a selected set of particles (51, 212, 603, 3076, 10,245 particles).

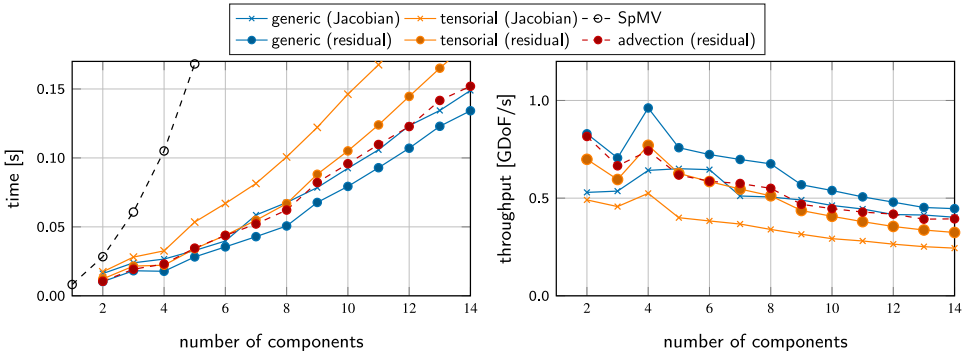


Fig. 21. Comparison of different sintering operator variants for different number of components: generic (Section 2.2), tensorial (Section 7.1), and advection operator (Section 7.2).

can be considered. This definition restrains the surface and the grain-boundary atomic fluxes to be strictly tangent to the corresponding diffusion paths, by introducing the following tensors:

$$\mathbf{T}_s = \mathbf{I} - \mathbf{n}_s \otimes \mathbf{n}_s,$$

$$\mathbf{T}_{ij} = \mathbf{I} - \mathbf{n}_{ij} \otimes \mathbf{n}_{ij},$$

where the unit normal vector to the grain surface is computed as

$$\mathbf{n}_s = \frac{\nabla c}{|\nabla c|}, \quad (11)$$

and the unit normal vector between the phases i and j is computed from the gradients of the corresponding order parameters as

$$\mathbf{n}_{ij} = \frac{\nabla \eta_i - \nabla \eta_j}{|\nabla \eta_i - \nabla \eta_j|}. \quad (12)$$

Here, symbol \otimes denotes a tensor product.

The definition of the Jacobian given by (7) also holds for the present case. Of course, the derivatives of the mobility differ, as detailed in Supplementary Material S2. Since the modified mobility only influences the rows of the Jacobian related to the Cahn–Hilliard block and does not influence the Allen–Cahn blocks, we can adopt the preconditioner from Section 4.3 also in the current context.

Table 8

Comparison of different sintering operator variants for different number of components (see Fig. 21). Table 1 contains the data for the generic implementation.

| N_c+2 | tensorial (residual) | | | | tensorial (Jacobian) | | | | advection (residual) | | | |
|---------|----------------------|-----|-----|------|----------------------|------|-----|------|----------------------|-----|-----|-----|
| | D/s | r/D | w/D | F/D | D/s | r/D | w/D | F/D | D/s | r/D | w/D | F/D |
| 2 | 0.70 | 5.3 | 1.3 | 706 | 0.49 | 14.0 | 1.5 | 1055 | 0.82 | 5.3 | 1.3 | 608 |
| 3 | 0.60 | 4.9 | 1.5 | 656 | 0.46 | 13.6 | 1.6 | 941 | 0.67 | 6.1 | 1.5 | 698 |
| 4 | 0.77 | 4.8 | 1.5 | 690 | 0.52 | 13.3 | 1.7 | 1024 | 0.74 | 5.6 | 1.6 | 739 |
| 5 | 0.63 | 4.6 | 1.6 | 753 | 0.40 | 13.2 | 1.7 | 1136 | 0.62 | 5.3 | 1.6 | 772 |
| 6 | 0.58 | 4.6 | 1.6 | 830 | 0.38 | 13.2 | 1.8 | 1278 | 0.59 | 5.1 | 1.6 | 794 |
| 7 | 0.55 | 4.5 | 1.6 | 918 | 0.37 | 13.1 | 1.8 | 1437 | 0.57 | 5.0 | 1.6 | 811 |
| 8 | 0.51 | 4.5 | 1.7 | 1012 | 0.34 | 13.0 | 1.8 | 1607 | 0.55 | 4.9 | 1.7 | 827 |
| 9 | 0.44 | 4.4 | 1.7 | 1111 | 0.32 | 13.0 | 1.9 | 1805 | 0.47 | 4.8 | 1.7 | 840 |
| 10 | 0.41 | 4.4 | 1.7 | 1212 | 0.29 | 13.0 | 1.9 | 1988 | 0.45 | 4.7 | 1.7 | 853 |
| 11 | 0.38 | 4.3 | 1.7 | 1316 | 0.28 | 13.0 | 2.0 | 2175 | 0.43 | 4.6 | 1.7 | 864 |
| 12 | 0.35 | 4.3 | 1.7 | 1421 | 0.26 | 12.9 | 2.0 | 2366 | 0.42 | 4.5 | 1.7 | 875 |
| 13 | 0.34 | 4.2 | 1.7 | 1528 | 0.25 | 12.9 | 2.1 | 2558 | 0.39 | 4.5 | 1.8 | 886 |
| 14 | 0.32 | 4.2 | 1.8 | 1636 | 0.24 | 12.9 | 2.1 | 2752 | 0.39 | 4.4 | 1.8 | 896 |

Additional challenges in comparison to the scalar-mobility case are: (1) in the application of the Jacobian, one needs the gradient of η_i at the linearization point in addition to those of c and μ , as required in the scalar case, and (2) more complex coupling terms need to be evaluated. We address the first point by precomputing the values of the linearization point at the quadrature points and by computing the gradients on the fly. In order to reduce the costs of the coupling, we use similar strategies as discussed in Section 4.2. In particular, we exploit the facts (1) that a tensor of the form $\mathbf{n}_c \otimes \mathbf{n}_b$ applied to a vector \mathbf{v} can be efficiently replaced by a dot product and a scalar scaling, $(\mathbf{n}_c \otimes \mathbf{n}_b)\mathbf{v} = \mathbf{n}_c(\mathbf{n}_b \cdot \mathbf{v})$, without the need to actually set up the tensor and (2) that in (11) and (12) no square root has to be computed, since it cancels out later on during subsequent multiplications.

Fig. 21 and Table 8 present timings, data volume, and work compared to the scalar-mobility case for the evaluation of the residual and of the Jacobian. One can observe a clear slowdown of up to 29 and 38%, respectively. This is caused by an increase of arithmetic operations by 7–127 and 35–244 FLOP per quadrature point for different numbers of components in the two cases, showing more pronouncedly the quadratic complexity. According to the roofline performance model of Fig. 7, high FLOP/s can be maintained with increasing number of components. In the case of the Jacobian, the measured read data volume decreases, as the gradients are computed on the fly. In total, the read and write data volumes are similar to the ones in Table 1, indicating that the compute-intensive operations are performed on cached data.

Table 5 gives statistics of a complete 51-particle simulation, showing an increase in the simulation time by 19% with an increase in the number of time steps and a decrease in the number of linear iterations. We note that the matrix-free and the Jacobian-free evaluation as well as the cut-off approach are applicable here without the need for any modifications.

7.2. Extension 2: advection terms

The system (1) is capable to capture the evolution of the microstructure during sintering, however, completely omits shrinkage mechanisms. In order to resolve this issue, Wang [22] added advection terms to system (1):

$$\frac{\partial c}{\partial t}(\mathbf{x}, t) = \nabla \cdot \left[M \nabla \frac{\delta F}{\delta c} \right] - \nabla \cdot (c \mathbf{v}), \quad (13a)$$

$$\frac{\partial \eta_i}{\partial t}(\mathbf{x}, t) = -L \frac{\delta F}{\delta \eta_i} - \nabla \cdot (\eta_i \mathbf{v}_i). \quad (13b)$$

Here, $\mathbf{v} = \sum_i \mathbf{v}_i$ and the advection velocity within individual particles \mathbf{v}_i consists of translational and rotational components:

$$\mathbf{v}_i(\mathbf{x}) = \begin{cases} \mathbf{v}_{t,i}(\mathbf{x}) + \mathbf{v}_{r,i}(\mathbf{x}), & \text{if inside (e.g., } \eta_i \geq 0.1) \\ 0, & \text{otherwise} \end{cases}$$

$$\mathbf{v}_{t,i}(\mathbf{x}) = \frac{m_t}{V_i} \mathbf{F}_i,$$

$$\mathbf{v}_{r,i}(\mathbf{x}) = \frac{m_r}{V_i} \mathbf{T}_i \times (\mathbf{x} - \mathbf{x}_{c_i}),$$

where m_t and m_r are constants characterizing the particle translation and rotation and where

$$V_i = \int_{\Omega} \eta_i d\Omega$$

is the particle volume. Vector \mathbf{x}_{c_i} denotes the mass center of the i th particle. The velocity components $\mathbf{v}_{t,i}$ and $\mathbf{v}_{r,i}$ are proportional to the force and torque [22], which are given by

$$\mathbf{F}_i = \int_{\Omega} d\mathbf{F}_i, \quad (14a)$$

$$\mathbf{T}_i = \int_{\Omega} (\mathbf{x} - \mathbf{x}_{c_i}) \times d\mathbf{F}_i. \quad (14b)$$

The effective local force density $d\mathbf{F}_i$ is the key component of the entire shrinkage mechanism and is related to the annihilation of the over-saturated vacancies at the grain boundaries:

$$d\mathbf{F}_i = k \sum_{i \neq j} (c - c_0) \langle \eta_i \eta_j \rangle [\nabla \eta_i - \nabla \eta_j] d\Omega.$$

The quantities V_i , \mathbf{F}_i , \mathbf{T}_i , \mathbf{x}_{c_i} can be considered as additional properties of the 0D representation, as discussed in Section 4.4. These quantities have to be gathered for each cell (batch) during the cell loop when multiple particles are assigned to the same order parameter.

The definition of advective velocities as proposed in [22] is far from optimal and introduces several drawbacks: the lack of rigorous physically based foundation, complicated calibration of the advection model parameters or severe size effects [70] if only the original Wang's forces are employed. In fact, the discussion of the rigid body motions in the phase-field sintering models [15,25,70] is a big topic by itself. Keeping the above-mentioned limitations in mind, we still implemented this particular extension due to its wide use and the fact that it can be applied for other non-local advection mechanisms. For instance, the staggered coupling of the phase-field and DEM approaches [71] also requires the reduced 0D modeling for its implementation and, thus, could benefit from the techniques described in the present publication.

The velocity terms in (13b) result in different Allen–Cahn blocks, potentially requiring different (ILU) preconditioners. Our experiments indicate that these terms can be dropped during preconditioning, allowing to work with a single ILU instance for all Allen–Cahn blocks, as proposed in Section 4.3.

Due to the non-local terms (14), the evaluation of the exact Jacobian would be both memory-intensive and computationally demanding. Therefore, we only consider the Jacobian-free implementation. Figs. 7, 21, Tables 5, and 8 describe the properties of the residual evaluation as stand-alone and in the context of the 51-particle sintering benchmark with and without cut-off. The increase in the simulation time can be

mainly contributed to the fact that the grain-tracking algorithm has to be performed at each time step. Also the number of operations during the residual evaluation increases by a fixed value of ~ 33 FLOP per quadrature point.

8. Conclusions and outlook

We have presented an efficient, adaptive, implicit finite-element solver for modeling solid-state-sintering processes by means of a well-established phase-field approach that is able to capture diffusive mass-transport, shrinkage, and grain-growth phenomena. Our implementation, which is freely available as `hpsint`, has been verified with reference data from the literature and successfully simulates packings with ten thousands of particles in high-performance-computing environments.

To enable such large-scale simulations, we have performed a holistic optimization of the solver on many levels by an interdisciplinary effort in order to minimize the time to solution. The proposed optimizations include a tailored block preconditioner, a distributed graph-based version of the grain-tracking algorithm, and the usage of fast matrix-free evaluation kernels. For the latter, the presented solver relies on the open-source library `deal.II` [20,21], particularly, on its state-of-the-art matrix-free framework [46,47]. Even though it is most efficient for higher-degree shape functions, the underlying algorithmic choices following the current trends of exascale algorithms ensure a high node-level performance also for the linear shape functions in the present case, with increasing advantage for larger numbers of components. We have extended the matrix-free algorithm to deal with varying number of vector components related to changing number of order parameters and to work only with locally-relevant components related to the local support of the phase field as well as presented low-level strategies to reduce the computational effort at quadrature points.

In addition to these fundamental advances, we have discussed possible optimizations that build upon the current developments and might allow an additional speedup of $2\times$ in the near future. This includes the usage of sparse block vectors [39], interleaving of evaluation and quadrature-point loops [47], and physics-based preconditioning [59].

CRediT authorship contribution statement

Peter Munch: Conceptualization, Investigation, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Vladimir Ivannikov:** Formal analysis, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Christian Cyron:** Conceptualization, Funding acquisition, Supervision, Writing – review & editing. **Martin Kronbichler:** Investigation, Methodology, Software, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The raw data required to reproduce these findings and the processed data required to reproduce these findings are available to download from <https://github.com/hpsint/hpsint.data>.

Acknowledgments

The authors acknowledge collaboration with Thomas Ebel, Daniel Paukner, Magdalena Schreter-Fleischhacker, and Regine Willumeit-Römer as well as the `deal.II` community. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (LRZ, www.lrz.de) through project id pn36li. This work was also funded by Helmholtz-Zentrum Hereon through I²B MgSinter project. We would like to credit Hui-Chia Yu, Robert Termuhlen, Xanthippi Chatzistavrou and Jason D. Nicholas for providing us the input data of their simulations.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at <https://doi.org/10.1016/j.commatsci.2023.112589>.

References

- [1] Huilong Zhu, Robert S. Averback, Molecular dynamics simulations of densification processes in nanocrystalline materials, *Mater. Sci. Eng. A* 204 (1) (1995) 96–100, Proceedings of the Symposium on Engineering of Nanostructured Materials.
- [2] Lifeng Ding, Ruslan L. Davidchack, Jingzhe Pan, A molecular dynamics study of sintering between nanoparticles, *Comput. Mater. Sci.* 45 (2) (2009) 247–256.
- [3] Ken-ichiro Mori, Finite element simulation of powder forming and sintering, *Comput. Methods Appl. Mech. Engrg.* 195 (48) (2006) 6737–6749.
- [4] Vladimir Ivannikov, Fritz Thomsen, Thomas Ebel, Regine Willumeit-Römer, Coupling the discrete element method and solid state diffusion equations for modeling of metallic powders sintering, *Comput. Part. Mech.* 10 (2) (2023) 185–207.
- [5] Christopher Gloeckle, Thomas Konkol, Olaf Jacobs, Wolfgang Limberg, Thomas Ebel, Ulrich A. Handge, Processing of highly filled polymer-metal feedstocks for fused filament fabrication and the production of metallic implants, *Materials* 13 (19) (2020).
- [6] Eshwara Phani Shubhakar Nidadavolu, Diana Krüger, Berit Zeller-Plumhoff, Domonkos Tolnai, Björn Wiese, Frank Feyerabend, Thomas Ebel, Regine Willumeit-Römer, Pore characterization of pm mg-0.6ca alloy and its degradation behavior under physiological conditions, *J. Magnes. Alloys* 9 (2) (2021) 686–703.
- [7] Robert Termuhlen, Xanthippi Chatzistavrou, Jason D. Nicholas, Hui-Chia Yu, Three-dimensional phase field sintering simulations accounting for the rigid-body motion of individual grains, *Comput. Mater. Sci.* 186 (2021) 109963.
- [8] Henrik Hierl, Johannes Hötzer, Marco Seiz, Andreas Reiter, Britta Nestler, Extreme scale phase-field simulation of sintering processes, in: 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (Scala), IEEE, 2019, pp. 25–32.
- [9] Johannes Hötzer, Andreas Reiter, Henrik Hierl, Philipp Steinmetz, Michael Selzer, Britta Nestler, The parallel multi-physics phase-field framework Pace3D, *J. Comput. Sci.* 26 (2018) 1–12.
- [10] Eisuke Miyoshi, Tomohiro Takaki, Munekazu Ohno, Yasushi Shibuta, Shinji Sakane, Takashi Shimokawabe, Takayuki Aoki, Ultra-large-scale phase-field simulation study of ideal grain growth, *NPJ Comput. Mater.* 3 (1) (2017) 25.
- [11] Derek Gaston, Chris Newman, Glen Hansen, Damien Lebrun-Grandié, Moose: A parallel computational framework for coupled systems of nonlinear equations, *Nucl. Eng. Des.* 239 (10) (2009) 1768–1778.
- [12] Ian Greenquist, Michael R. Tonks, Larry K. Agesen, Yongfeng Zhang, Development of a microstructural grand potential-based sintering model, *Comput. Mater. Sci.* 172 (2020) 109288.
- [13] Supriyo Ghosh, Christopher K. Newman, Marianne M. Francois, Tusas: A fully implicit parallel approach for coupled phase-field equations, *J. Comput. Phys.* 448 (2022) 110734.
- [14] Stephen DeWitt, Shiva Rudraraju, David Montiel, W. Beck Andrews, Katsuyo Thornton, PRISMS-PF: A general framework for phase-field modeling with a matrix-free finite element method, *npj Comput. Mater.* 6 (1) (2020) 1–12.
- [15] Vladimir Ivannikov, Fritz Thomsen, Thomas Ebel, Regine Willumeit-Römer, Capturing shrinkage and neck growth with phase field simulations of the solid state sintering, *Modelling Simul. Mater. Sci. Eng.* 29 (7) (2021) 075008.
- [16] Fritz Thomsen, Götz Hofmann, Thomas Ebel, Regine Willumeit-Römer, An elementary simulation model for neck growth and shrinkage during solid phase sintering, *Materialia* 3 (2018) 338–346.

- [17] Cody J. Permann, Michael R. Tonks, Bradley Fromm, Derek R. Gaston, Order parameter re-mapping algorithm for 3D phase field model of grain growth using FEM, *Comput. Mater. Sci.* 115 (2016) 18–25.
- [18] Phani Motamarri, Sambit Das, Shiva Rudraraju, Krishnendu Ghosh, Denis Davydov, Vikram Gavini, DFT-FE – a massively parallel adaptive finite-element code for large-scale density functional theory calculations, *Comput. Phys. Comm.* 246 (2020) 106853.
- [19] Nikhil Kodali, Gourab Panigrahi, Debashis Panda, Phani Motamarri, Fast hardware-aware matrix-free computations of higher-order finite-element discretized matrix multi-vector products, 2022, arXiv preprint arXiv:2208.07129.
- [20] Daniel Arndt, Wolfgang Bangerth, Marco Feder, Marc Fehling, Rene Gassmüller, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Perleter, Simon Sticko, Bruno Turcksin, David Wells, The deal.II library, version 9.4, *J. Numer. Math.* 30 (3) (2022) 231–246.
- [21] Daniel Arndt, Wolfgang Bangerth, Denis Davydov, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Jean-Paul Perleter, Bruno Turcksin, David Wells, The deal.II finite element library: Design, features, and insights, *Comput. Math. Appl.* 81 (2021) 407–422.
- [22] Yu U. Wang, Computer modeling and simulation of solid-state sintering: A phase field approach, *Acta Mater.* 54 (4) (2006) 953–961.
- [23] Jie Deng, A phase field model of sintering with direction-dependent diffusion, *Mater. Trans.* 53 (2) (2012) 385–389.
- [24] Karim Ahmed, Clarissa A. Yablinsky, A. Schulte, Todd Allen, Anter El-Azab, Phase field modeling of the effect of porosity on grain growth kinetics in polycrystalline ceramics, *Modelling Simul. Mater. Sci. Eng.* 21 (6) (2013) 065005.
- [25] Marco Seiz, Effect of rigid body motion in phase-field models of solid-state sintering, *Comput. Mater. Sci.* 215 (2022) 111756.
- [26] Sudipta Biswas, Daniel Schwen, Hao Wang, Maria A. Okuniewski, Vikas Tomar, Phase field modeling of sintering: Role of grain orientation and anisotropic properties, *Comput. Mater. Sci.* 148 (2018) 307–319.
- [27] Carl E. Krill III, Long-Qing Chen, Computer simulation of 3-D grain growth using a phase-field model, *Acta Mater.* 50 (12) (2002) 3059–3075.
- [28] Srikanth Vedantam, Prasad Patnaik, Efficient numerical algorithm for multiphase field simulations, *Phys. Rev. E* 73 (2006) 016703.
- [29] Qingcheng Yang, Yongxin Gao, Arkadz Kirshtein, Qiang Zhen, Chun Liu, A free-energy-based and interfacially consistent phase-field model for solid-state sintering without artificial void generation, *Comput. Mater. Sci.* 229 (2023) 112387.
- [30] Sudipta Biswas, Daniel Schwen, Vikas Tomar, Implementation of a phase field model for simulating evolution of two powder particles representing microstructural changes during sintering, *J. Mater. Sci.* 53 (8) (2018) 5799–5825.
- [31] K. Chockalingam, Varvara. G. Kouznetsova, Olaf van der Sluis, Marc G.D. Geers, 2D phase field modeling of sintering of silver nanoparticles, *Comput. Methods Appl. Mech. Engrg.* 312 (2016) 492–508.
- [32] Michael Pernice, Homer F. Walker, NITSOL: A Newton iterative solver for nonlinear systems, *SIAM J. Sci. Comput.* 19 (1) (1998) 302–318.
- [33] Peter N. Brown, Youcef Saad, Hybrid Krylov methods for nonlinear systems of equations, *SIAM J. Sci. Stat. Comput.* 11 (3) (1990) 450–481.
- [34] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William Gropp, et al., PETSc Users Manual, Argonne National Laboratory, 2019.
- [35] Jed Brown, Efficient nonlinear solvers for nodal high-order finite elements in 3D, *J. Sci. Comput.* 45 (1) (2010) 48–63.
- [36] Stephen DeWitt, Shiva Rudraraju, David Montiel, W. Beck Andrews, Katsuyo Thornton, PRISMS-PF: A general framework for phase-field modeling with a matrix-free finite element method, *npj Comput. Mater.* 6 (1) (2020) 29.
- [37] Seong Gyooun Kim, Dong Ik Kim, Won Tae Kim, Yong Bum Park, Computer simulations of two-dimensional and three-dimensional ideal grain growth, *Phys. Rev. E* 74 (6) (2006) 061605.
- [38] Denis Davydov, Timo Heister, Martin Kronbichler, Paul Steinmann, Matrix-free locally adaptive finite element solution of density-functional theory with nonorthogonal orbitals and multigrid preconditioning, *Phys. Status Solidi (b)* 255 (9) (2018) 1800069.
- [39] Denis Davydov, Martin Kronbichler, Algorithms and data structures for matrix-free finite element operators with MPI-parallel sparse multi-vectors, *ACM Trans. Parallel Comput. (TOPC)* 7 (3) (2020) 1–30.
- [40] Daniel Arndt, Niklas Fehn, Guido Kanschat, Katharina Kormann, Martin Kronbichler, Peter Munch, Wolfgang A. Wall, Julius Witte, ExaDG: High-order discontinuous Galerkin for the exa-scale, in: *Software for Exascale Computing-SPEXA 2016-2019*, Springer, 2020, pp. 189–224.
- [41] Michel O. Deville, Paul F. Fischer, Ernest H. Mund, *High-Order Methods for Incompressible Fluid Flow*, Cambridge University Press, 2002.
- [42] Martin Kronbichler, Ababacar Diagne, Hanna Holmgren, A fast massively parallel two-phase flow solver for microfluidic chip simulation, *Int. J. High Perform. Comput. Appl.* 32 (2) (2018) 266–287.
- [43] Denis Davydov, Jean-Paul Perleter, Daniel Arndt, Martin Kronbichler, Paul Steinmann, A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid, *Internat. J. Numer. Methods Engrg.* 121 (13) (2020) 2874–2895.
- [44] Jed Brown, Valeria Barra, Natalie Beams, Leila Ghaffari, Matthew Knepley, William Moses, Rezgar Shakeri, Karen Stengel, Jeremy L. Thompson, Junchao Zhang, Performance portable solid mechanics via matrix-free ρ -multigrid, 2022, arXiv preprint arXiv:2204.01722.
- [45] Peter Munch, Katharina Kormann, Martin Kronbichler, hyper.deal: An efficient, matrix-free finite-element library for high-dimensional partial differential equations, *ACM Trans. Math. Software* 47 (4) (2021) 33/1–34.
- [46] Martin Kronbichler, Katharina Kormann, A generic interface for parallel cell-based finite element operator application, *Comput. & Fluids* 63 (2012) 135–147.
- [47] Martin Kronbichler, Katharina Kormann, Fast matrix-free evaluation of discontinuous Galerkin finite element operators, *ACM Trans. Math. Softw.* 45 (3) (2019) 1–40.
- [48] Tzanio Kolev, Paul Fischer, Misun Min, Jack Dongarra, Jed Brown, Veselin Dobrev, Tim Warburton, Stanimire Tomov, Mark S. Shephard, Ahmad Abdelfattah, Valeria Barra, Natalie Beams, Jean-Sylvain Camier, Noel Chalmers, Yohann Doudout, Ali Karakus, Ian Karlin, Stefan Kerkemeier, Yu-Hsiang Lan, David Medina, Elia Merzari, Aleksandr Obabko, Will Pazner, Thilina Rathnayake, Cameron W. Smith, Lukas Spies, Kasia Swirydowicz, Jeremy Thompson, Ananias Tomboulides, Vladimir Tomov, Efficient exascale discretizations: High-order finite element methods, *Int. J. High Perform. Comput. Appl.* 35 (6) (2021) 527–552.
- [49] Peter Munch, Karl Ljungkvist, Martin Kronbichler, Efficient application of hanging-node constraints for matrix-free high-order FEM computations on CPU and GPU, in: *International Conference on High Performance Computing*, Springer, 2022, pp. 133–152.
- [50] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Veselin Dobrev, Yohann Doudout, Aaron Fisher, Tzanio Kolev, Will Pazner, Mark Stowell, Vladimir Tomov, Ido Akkerman, Johann Dahm, David Medina, Stefano Zampini, MFEM: A modular finite element methods library, *Comput. Math. Appl.* 81 (2021) 42–74.
- [51] Martin Kronbichler, Karl Ljungkvist, Multigrid for matrix-free high-order finite element computations on graphics processors, *ACM Trans. Parallel Comput.* 6 (1) (2019) 2:1–32.
- [52] Karl Ljungkvist, Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes, in: *SpringSim (HPC)*, 2017.
- [53] Steffen Müthing, Marian Piatkowski, Peter Bastian, High-performance implementation of matrix-free high-order discontinuous Galerkin methods, 2017, arXiv preprint arXiv:1711.10885.
- [54] Jan Treibig, Georg Hager, Gerhard Wellein, LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments, in: *Proceedings of PSTI2010*, San Diego CA, 2010, pp. 207–216.
- [55] Samuel Williams, Andrew Waterman, David Patterson, Roofline: an insightful visual performance model for multicore architectures, *Commun. ACM* 52 (4) (2009) 65–76.
- [56] Katarzyna Świrydowicz, Noel Chalmers, Ali Karakus, Timothy Warburton, Acceleration of tensor-product operations for high-order finite element methods, *Int. J. High Perform. Comput. Appl.* 33 (4) (2019) 735–757.
- [57] Martin Kronbichler, Dmytro Sashko, Peter Munch, Enhancing data locality of the conjugate gradient method for high-order matrix-free finite-element implementations, *Int. J. High Perform. Comput. Appl.* (2022) 10943420221107880.
- [58] Peter Munch, Martin Kronbichler, Cache-optimized and low-overhead implementations of additive Schwarz methods for high-order FEM multigrid computations, 2023, Manuscript.
- [59] Owe Axelsson, Petia Boyanova, Martin Kronbichler, Maya Neytcheva, Xunxun Wu, Numerical and computational efficiency of solvers for two-phase problems, *Comput. Math. Appl.* 65 (3) (2013) 301–314.
- [60] James D. Foley, Andries Van Dam, John F. Hughes, Steven K. Feiner, *Computer Graphics: Principles and Practice*, second ed., Addison-Wesley, 1990.
- [61] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, Martin Kronbichler, Algorithms and data structures for massively parallel generic adaptive finite element codes, *ACM Trans. Math. Softw.* 38 (2) (2012) 1–28.
- [62] Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, The Pearson Education, 2001.
- [63] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, et al., *An Overview of Trilinos*, Citeseer, 2003.

- [64] Xiaobing Feng, Steven Wise, Analysis of a Darcy–Cahn–Hilliard diffuse interface model for the Hele–Shaw flow and its fully discrete finite element approximation, *SIAM J. Numer. Anal.* 50 (3) (2012) 1320–1343.
- [65] Carsten Burstedde, Lucas C. Wilcox, Omar Ghattas, p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees, *SIAM J. Sci. Comput.* 33 (3) (2011) 1103–1133.
- [66] Sudipta Biswas, Daniel Schwen, Jogender Singh, Vikas Tomar, A study of the evolution of microstructure and consolidation kinetics during sintering using a phase field modeling based approach, *Extreme Mech. Lett.* 7 (2016) 78–89.
- [67] Robert D. Falgout, Ulrike Meier Yang, hypre: A library of high performance preconditioners, in: *Computational Science–ICCS 2002: International Conference Amsterdam, The Netherlands, April 21–24, 2002 Proceedings, Part III*, Springer, 2002, pp. 632–641.
- [68] Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, Graham F. Carey, libMesh: A C++ library for parallel adaptive mesh refinement/coarsening simulations, *Eng. Comput.* 22 (3–4) (2006) 237–254.
- [69] Vaclav Smilauer, et al., *Yade Documentation*, second ed., The Yade Project, 2015, <http://yade-dem.org/doc/>.
- [70] Marco Seiz, Henrik Hierl, Britta Nestler, An improved grand-potential phase-field model of solid-state sintering for many particles, *Modelling Simul. Mater. Sci. Eng.* 31 (5) (2023) 055006.
- [71] Kazunari Shinagawa, Simulation of grain growth and sintering process by combined phase-field/discrete-element method, *Acta Mater.* 66 (2014) 360–369.