Research Paper

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2023, Vol. 37(2) 61–81
© The Author(s) 2022
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/10943420221107880
journals.sagepub.com/home/hpc
SAGE

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

Martin Kronbichler[1,2], Dmytro Sashko[3] 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 Guermond 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

[1]Institute for Computational Mechanics, Technical University of Munich, Germany
[2]Department of Mathematics, University of Augsburg, Germany
[3]School of Mechanical and Mining Engineering, The University of Queensland, Saint Lucia, QLD, Australia
[4]Institute 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.

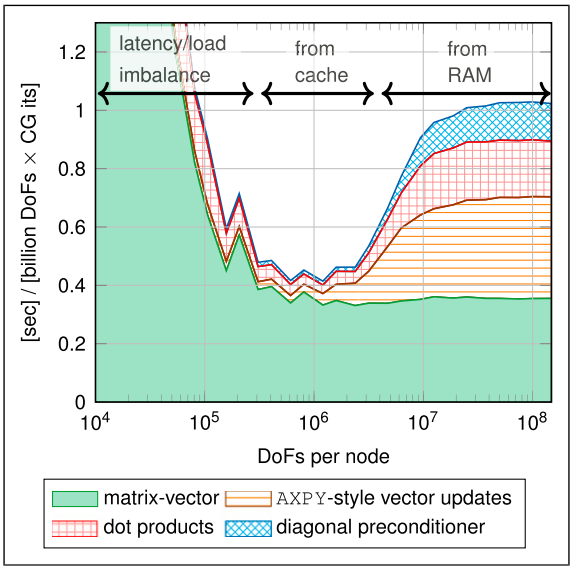

Figure 1. Breakdown of times per CG iteration in the Center of Efficient Exascale Discretizations (CEED) benchmark problem BP4 (Fischer et al., 2020) with finite elements of degree $p = 5$ on 2 × 24 cores of Intel Xeon Platinum 8174. See Section 3.1 below for a more detailed explanation of the steps.

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 $\Omega \subset \mathbb{R}^3$

$$-\nabla^2 \boldsymbol{u} = \boldsymbol{f} \tag{1}$$

with the vector field $\boldsymbol{u}(\boldsymbol{x}) = (u_1(\boldsymbol{x}), u_2(\boldsymbol{x}), u_3(\boldsymbol{x})) \in (H^1(\Omega))^3$ and a forcing $\mathbf{f} \in (L^2(\Omega))^3$. On the domain boundary $\partial\Omega$, Dirichlet boundary conditions $\boldsymbol{u} = \boldsymbol{g}$ 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, $e = 1, \ldots, n_{\text{cells}}$. On a hexahedral element Ω_e, the solution interpolation is given by

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

Here, $\boldsymbol{u}_e = [u_{e,j}]_j$ denotes the vector of unknown coefficients on Ω_e in an expansion with a polynomial basis $\{\varphi_j, j = 1, \ldots, 3(p+1)^3\}$. 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

$$\mathbf{A}\mathbf{u} = \mathbf{b} \tag{3}$$

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

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

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

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

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 $n_{\text{q}} = (p+2)^3$ points $\widehat{\boldsymbol{x}}_q$ per cell and the associated quadrature weight w_q. The integrals are transformed to reference coordinates $\widehat{\boldsymbol{x}}$ via a polynomial mapping $\boldsymbol{x}(\widehat{\boldsymbol{x}})$ 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 $[\boldsymbol{J}_e(\widehat{\boldsymbol{x}})]_{ij} = \frac{\partial x_i}{\partial \widehat{x}_j}$. The local result $\boldsymbol{A}_e\boldsymbol{u}_e$ is obtained by evaluating equation (5) for all test functions φ_i, $i = 1, \ldots, 3(p+1)^3$.

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 $\boldsymbol{P}_e$ 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 $\boldsymbol{P}_e$ 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 $\boldsymbol{J}_{e,q}^{-1}$ 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, $\boldsymbol{J}_{e,q}^{-1}(w_q \det \boldsymbol{J}_{e,q})\boldsymbol{J}_{e,q}^{-T}$ ("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

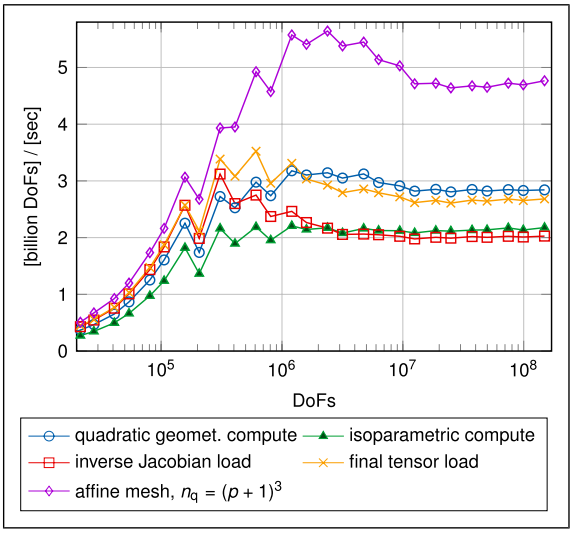

Figure 2. Comparison of different implementations of matrix-free operator evaluation for polynomial degree $p = 5$ on 2×24 cores of Intel Xeon Platinum 8174.

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 $\boldsymbol{J}^{-1}$ throughout the whole mesh and using $n_q = (p+1)^d$ 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 $\boldsymbol{u}$, $\boldsymbol{v}$, 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 examined in detail, for example, in Kronbichler and Wall (2018).

Algorithm 1 Preconditioned conjugate gradient method.

$r_0 = b - \mathbf{A}x_0$, $z_0 = M^{-1}r_0$, $p_0 = z_0$, $e_0 = r_0^T z_0$
$k = 0$
while not converged **do**
  $v_k = A p_k$
  $\alpha_k = \frac{e_k}{p_k^T v_k}$ | 1st region:r:2
  $x_{k+1} = x_k + \alpha_k \mathbf{p}_k$ | 2nd region:r:4/w:2
  $r_{k+1} = r_k - \alpha_k \mathbf{v}_k$
  if $\sqrt{\gamma_{k+1}} = \|r_{k+1}\| < \varepsilon$ **then**
    break
  end if
  $z_{k+1} = M^{-1} r_{k+1}$
  $e_{k+1} = r_{k+1}^T z_{k+1}$ | 3rd region:r:2
  $\beta_k = \frac{e_{k+1}}{e_k}$
  $p_{k+1} = z_{k+1} + \beta_k p_k$ | 4th region: r:2/w:1
  $k = k+1$
end while

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 $\boldsymbol{u}$, 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 $\|\boldsymbol{r}_k\|$, 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 ($\boldsymbol{y} = a\boldsymbol{x} + \boldsymbol{y}$),
- 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 $\|\boldsymbol{r}_{k+1}\|$ already during the computation of $\boldsymbol{r}_{k+1}$, 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 $\boldsymbol{x}_{k+1}$ and $\boldsymbol{r}_{k+1}$ depends on $\boldsymbol{p}_k$, $\boldsymbol{r}_k$, $\boldsymbol{v}_k$, $\boldsymbol{x}_k$, and α_k. The latter itself depends on $\boldsymbol{p}_k$ and $\boldsymbol{v}_k$ and requires a full vector sweep through them. If the size of the vectors $\boldsymbol{p}_k$ and $\boldsymbol{v}_k$ 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 $\boldsymbol{p}_{k+1}$ in the fourth region, the vector entries of $\boldsymbol{p}_k$ and $\boldsymbol{z}_{k+1}$ 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:	$\mathbf{q}_k = \mathbf{A}\mathbf{w}_k$	
3:	$\beta_k = \gamma_{k-1}/\gamma_{k-2}$	
4:	$\alpha_k = \gamma_{k-1}/\left(\delta_{k-1} - \beta_k \frac{\gamma_{k-1}}{\alpha_{k-1}}\right)$	
5:	$\mathrm{p}_k = \mathrm{r}_k + \beta_k \mathrm{p}_{k-1}$	r:7/w:6
6:	$\mathrm{x}_k = \mathrm{x}_k + \alpha_k \mathrm{p}_k$	
7:	$\mathrm{s}_k = \mathrm{w}_k + \beta_k \mathrm{s}_{k-1}$	
8:	$\mathrm{r}_k = \mathrm{r}_{k-1} - \alpha_k \mathrm{s}_k$	
9:	$\mathrm{z}_k = \mathrm{q}_k + \beta_k \mathrm{z}_{k-1}$	
10:	$\mathrm{w}_k = \mathrm{w}_{k-1} - \alpha_k \mathrm{z}_k$	
11:	$\gamma_k = \mathrm{r}_k^{\mathrm{T}} \mathrm{r}_k$	
12:	$\delta_k = \mathrm{w}_k^{\mathrm{T}} \mathrm{r}_k$	
13:	**end while**	

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

1:	**while** not converged **do**	
2:	$\mathrm{T}_k = [\mathrm{r}_k, \mathrm{A}\mathrm{r}_k, \ldots, \mathrm{A}^s \mathrm{r}_k]$	
3:	$\mathrm{B}_k = -\mathrm{W}_{k-1}^{-1}(\mathrm{Q}_k^{\mathrm{T}} \mathrm{P}_{k-1})$	r:2s/w:0
4:	$\mathrm{P}_k = \mathrm{R}_k + \mathrm{P}_{k-1} \mathrm{B}_k$	r:2s+1/w:1
5:	$\mathrm{W}_k = \mathrm{Q}_k^{\mathrm{T}} \mathrm{P}_k$	
6:	$\mathrm{g}_k = \mathrm{P}_k^{\mathrm{T}} \mathrm{r}_k$	
7:	$\mathrm{a}_k = \mathrm{W}_k^{-1} \mathrm{g}_k$	
8:	$\mathrm{x}_k = \mathrm{x}_{k-1} + \mathrm{P}_k \mathrm{a}_k$	r:s+1/w:1
9:	$\mathrm{r}_k = b - \mathrm{A}_k \mathrm{x}_k$	r:2/w:1
10:	$\gamma_k = \mathrm{r}_k^{\mathrm{T}} \mathrm{r}_k$	
11:	**end while**	

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 $\boldsymbol{v}_k$ and $\boldsymbol{z}_{k+1}$ use the same memory). This number can be slightly reduced to 9 by moving the computation of $\boldsymbol{x}_{k+1}$ to the 4th region to reuse reads of $\boldsymbol{p}_k$.

3.2. Alternative CG methods

For a simpler comparison, we now consider plain conjugate gradient algorithms without preconditioner. The basic version (Algorithm 1 with $\boldsymbol{z}_{k+1} = \boldsymbol{r}_{k+1}$ and $\boldsymbol{M}^{-1} = \boldsymbol{I}$) 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 $\boldsymbol{x}$ every other iteration (before and after the update of $\boldsymbol{p}$).

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, $\boldsymbol{P}_k$ instead of $\boldsymbol{p}_k$ and $\boldsymbol{R}_k$ instead of $\boldsymbol{r}_k$. 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 $[\mathbf{A}\mathbf{r}_k, \ldots, \mathbf{A}^s\mathbf{r}_k]$ 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 ($\boldsymbol{z}_k = \boldsymbol{r}_k$ and $\boldsymbol{M}^{-1} = \boldsymbol{I}$) and aim to perform the computation of contributions to β_k before finalizing the computation of α_k and $\boldsymbol{r}_{k+1}$. We therefore expand $\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}$ into

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

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 $\boldsymbol{r}_k$ during the summation compared to the computation of $\mathbf{v}_k^T \mathbf{p}_k$ alone, this is compensated by computing $\boldsymbol{r}_{k+1}$ at the same time as using the respective entry for $\boldsymbol{p}_{k+1}$. In addition, the fused scheduling uses $\boldsymbol{p}_k$ for both $\boldsymbol{x}_{k+1}$ and $\boldsymbol{p}_{k+1}$. 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 $\boldsymbol{x}_{k+1}$ only every other iteration, reusing the content of the vector $\boldsymbol{p}_{k-1}$ and $\boldsymbol{r}_{k-1}$ 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:	$\mathbf{v}_k = \mathbf{A}\mathbf{p}_k$	
11:	$a_k = \mathbf{p}_k^T \mathbf{v}_k$	"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:	$\alpha_k = \frac{\gamma_k}{a_k}$	
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 $\boldsymbol{z}_{k+1} = \boldsymbol{M}^{-1}\boldsymbol{r}_{k+1}$, 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

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

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

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 $\boldsymbol{r}_k$ and the result of the matrix-vector product $\boldsymbol{v}_k$, all scalars of the current conjugate gradient iteration can be computed using one reduction region. The vector $\boldsymbol{z}_{k+1}$ is no longer stored explicitly, since we assume that the application of the preconditioner is cheaper than the read and write of $\boldsymbol{z}_{k+1}$. 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 $\boldsymbol{x}_k$ 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 $\boldsymbol{r}_k$ and one for $\boldsymbol{p}_k$ as well as one for $\boldsymbol{x}_k$ 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.

Algorithm 5 Preconditioned conjugate gradient method with merged vector operations.

```
1:  k = 0, α_0 = β_0 = 0, r_0 = b − Ax_0, p_0 = v_0 = 0
2:  while not converged do
3:    k = k + 1
4:    if k > 1 odd then                          "pre" region:r:3.83/w:2.5
5:      x_k = x_{k−2} + α_{k−1} p_{k−1}
6:      + (α_{k−2}/β_{k−2}) (p_{k−1} − M^{−1} r_{k−1})
7:    end if
8:    r_k = r_{k−1} − α_{k−1} v_{k−1}
9:    p_k = M^{−1} r_k + β_{k−1} p_{k−1}
10:   v_k = A p_k
11:   γ_k = r_k^T r_k                             "post" region:r:3. 3/w:0
12:   a_k = p_k^T v_k
13:   b_k = r_k^T v_k
14:   c_k = v_k^T v_k
15:   d_k = r_k^T M^{−1} r_k
16:   e_k = r_k^T M^{−1} v_k
17:   f_k = v_k^T M^{−1} v_k
18:   α_k = d_k / a_k
19:   if sqrt(γ_k − 2α_k b_k + α_k^2 c_k) < ε then
20:     if k odd then
21:       x_{k+1} = x_k + α_k p_k
22:     else
23:       x_{k+1} = x_{k−1} + α_k p_k + (α_{k−1}/β_{k−1}) (p_k − M^{−1} r_k)
24:     end if
25:     break
26:   end if
27:   β_k = (d_k − 2α_k e_k + α_k^2 f_k) / d_k
28: end while
```

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

- 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 $\boldsymbol{r}_k$ and $\boldsymbol{v}_k$ from caches or registers when $\boldsymbol{M}^{-1}\boldsymbol{r}_k$ and $\boldsymbol{M}^{-1}\boldsymbol{v}_k$ are computed.
- The memory access induced by the preconditioner is assumed to be less expensive than the aggregated store and load of $\boldsymbol{z}_{k+1}$ in Algorithm 1.

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 $\boldsymbol{M}^{-1}\boldsymbol{r}_{k+1}$ 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 $\boldsymbol{p}$, $\boldsymbol{r}$, $\boldsymbol{v}$ in caches during the "post" stages, leading to a single memory read of the vectors $\boldsymbol{p}$, $\boldsymbol{r}$, $\boldsymbol{v}$, and $\boldsymbol{x}$ 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 $\boldsymbol{u}$ and produces the destination vector $\boldsymbol{v}$. The cell-wise nature of our matrix-free algorithm, involves a loop through each cell of the mesh that

- reads all unknowns $\boldsymbol{u}_e = \boldsymbol{P}_e\boldsymbol{u}$ 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, $i \in \{1, \ldots, n\}$, 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 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 $\boldsymbol{u}$ 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 $\boldsymbol{v}$ as a "post" operation.

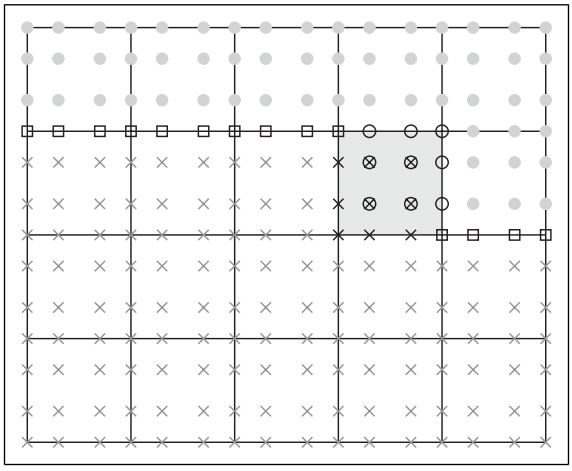

Figure 3. Illustration of data dependencies in matrix-free operator evaluation with degree $p = 3$ for a lexicographic loop through the cells starting from the bottom left. Each symbol represents an unknown. Gray crosses denote unknowns where the result of the operator evaluation is complete before the 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.

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 $\boldsymbol{u}$ 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 $\boldsymbol{v}$ 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

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

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

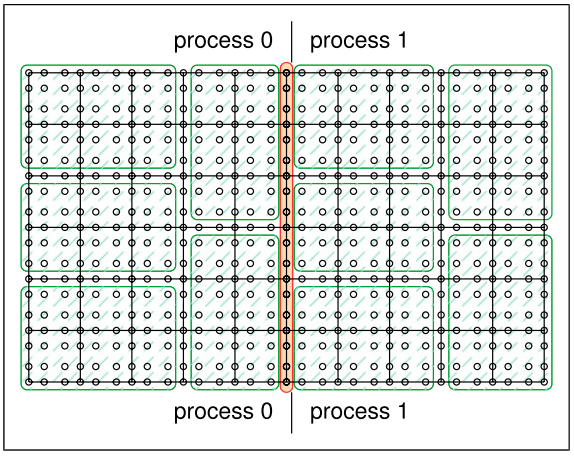

Figure 4. Illustration of the numbering of degrees of freedom for a 2D setup with polynomial degree $p = 3$. Cells are grouped together into batches of 6 cells and the interior unknowns are numbered first (highlighted by green-shaded boxes). The second set of numbers are unknowns located on more than 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_Isend 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 $\boldsymbol{A}$ 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

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

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

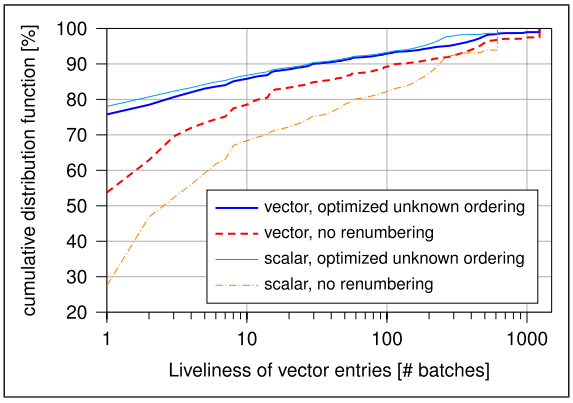

Figure 5. Liveliness of data in vector ranges for the 3D vector and scalar Laplacians with polynomial degree $p = 5$ on 40 MPI processes. The vector Laplacian involves 297 million DoFs subdivided into 1229 cell batches on each MPI process, the scalar Laplacian 99 million DoFs with 615 cell batches.

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

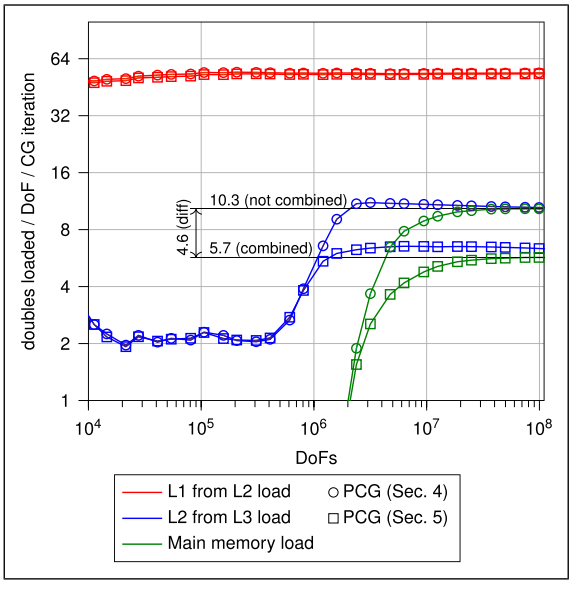

Figure 6. Comparison of measured memory transfer for 2 × 20 cores of Intel Xeon Gold 6230 for standard and combined versions of Algorithm 5.

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 $3.\overline{3}$ 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	–	–	$3.8\overline{3}$	3.5

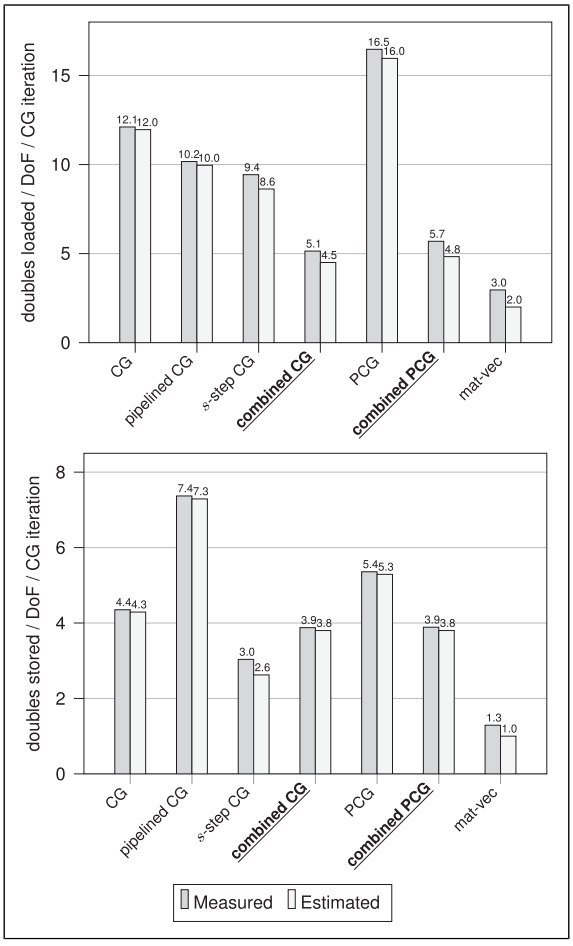

Figure 7. Comparison of measured and estimated memory transfer for various methods on 2 × 20 cores of Intel Xeon Gold 6230 and 10^8 DoFs, using the measured values of the matrix-vector multiplication as a baseline transfer.

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 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.

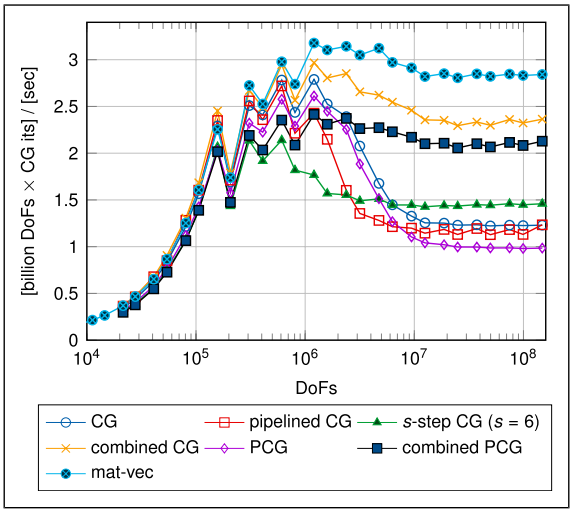

Figure 8. Throughput over the problem size on a single Intel Xeon Platinum 8174 node for the different CG variants.

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

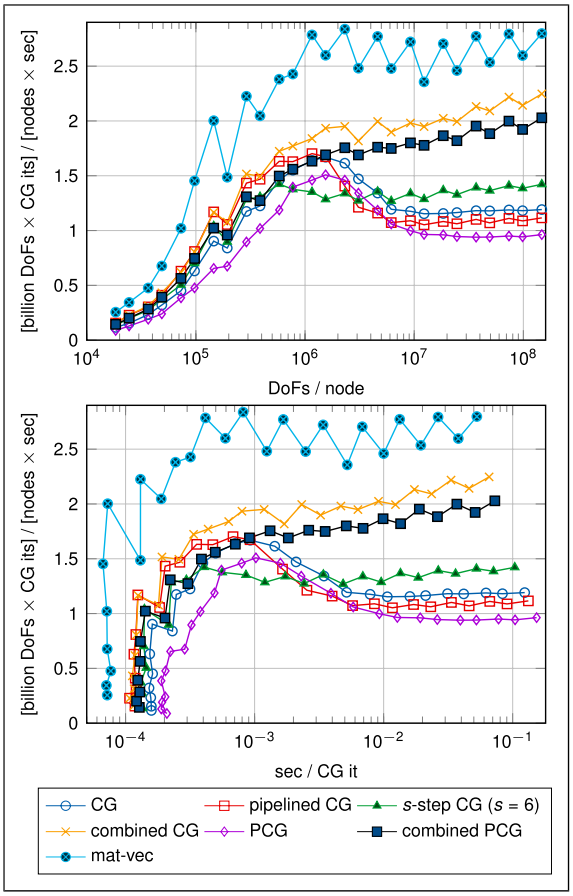

Figure 9. Throughput over the problem size per node (top) and throughput over latency (bottom) on 512 nodes of Intel Xeon Platinum 8174 (24,576 MPI ranks) for various formulations.

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 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.

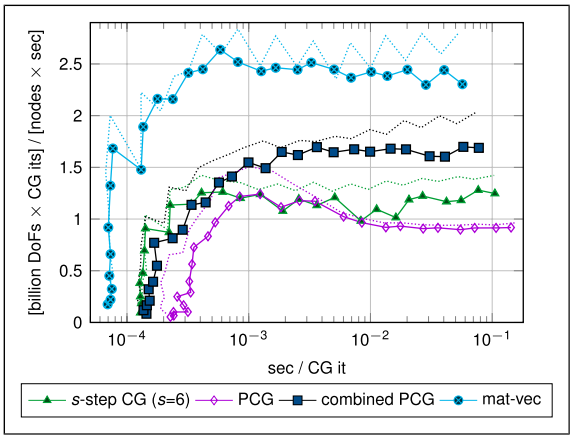

Figure 10. Throughput over the problem size per node (top) and throughput over latency (bottom) on 3072 nodes of Intel Xeon Platinum 8174 (147,456 MPI ranks) for various formulations. The dotted lines show the scaled throughput on 512 nodes (see also Figure 9).

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.

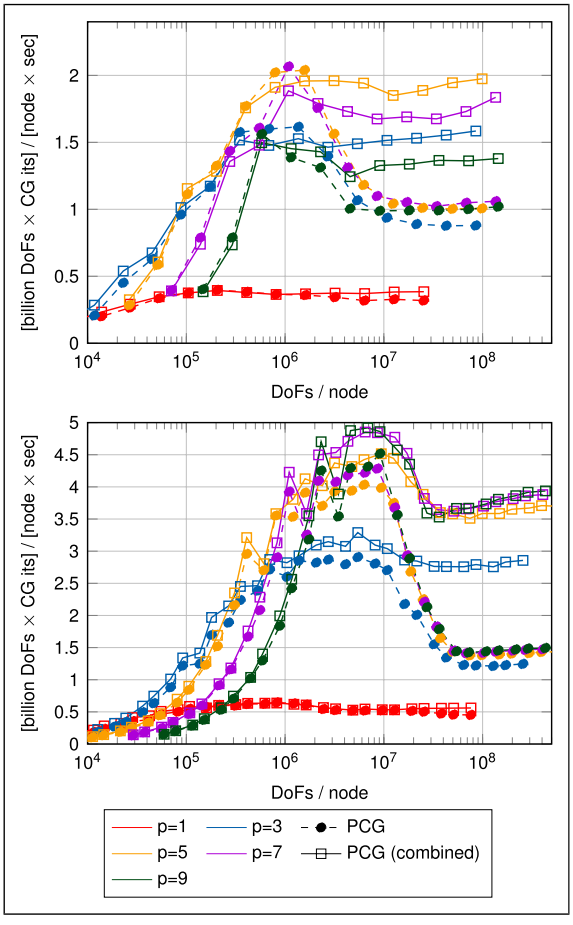

Figure 11. Throughput over the problem size on four nodes of Intel Xeon Platinum 8174 (top) and on one node of 2 × 64 core Advanced Micro Devices Inc. (AMD) Epyc 7742 (bottom) for the BP4 benchmark for different polynomial degrees p, all with quadratic geometry representation and $n_q = (p + 2)^3$ quadrature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

6.3.1. Variation of the polynomial degree

The results obtained for the polynomial degree $p = 5$ above are transferable also to other polynomials 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× faster when loading the inverse Jacobians and even 2.27× 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,

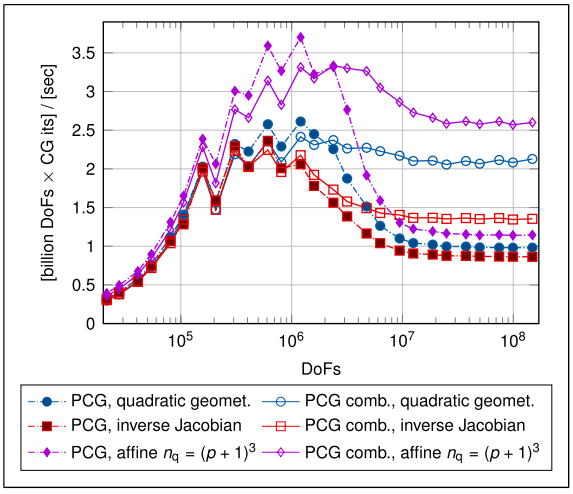

Figure 12. Comparison of throughput of BP4 benchmark with basic preconditioned CG algorithm and the proposed combined variant with different implementations of matrix-free operator evaluation for polynomial degree $p = 5$ on 2 × 24 cores of Intel Xeon Platinum 8174.

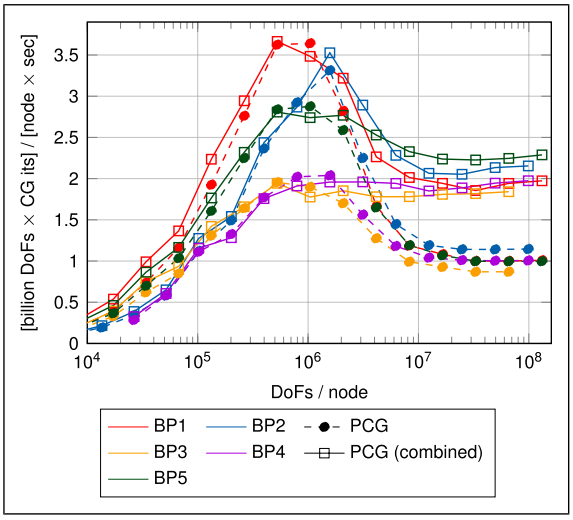

Figure 13. Throughput over the problem size for the standard preconditioned CG scheme and the proposed improved version on #3 nodes of Intel Xeon Platinum 8174 for the CEED benchmark problems BP1 (scalar mass matrix), BP2 (vector-valued mass matrix), BP3 (scalar Laplace matrix), BP4 (vector-valued Laplace matrix), and BP5 (scalar Laplace matrix, collocation setting with Gauss–Lobatto quadrature on $n_q = (p + 1)^3$ 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× compared to the baseline CG solver (compared to 2 GDoFs/sec and 2× 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 Summit[3] (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

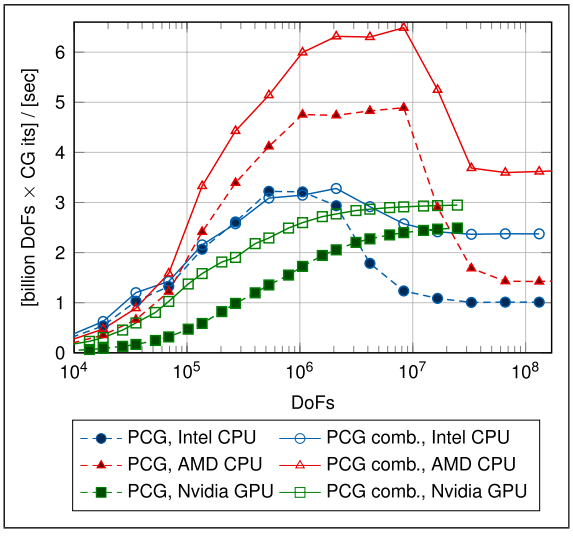

Figure 14. BP5: Throughput over the problem size on a single node for the basic preconditioned CG method and the proposed combined variant.

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 JI, 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, Tutlyaeva 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, Hoefler 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 Tissot 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.

Guermond 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 intratile 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

Patera 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, Jüngel 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.