High-Performance Implementations for High-Order Finite-Element Discretizations of PDEs

Martin Kronbichler

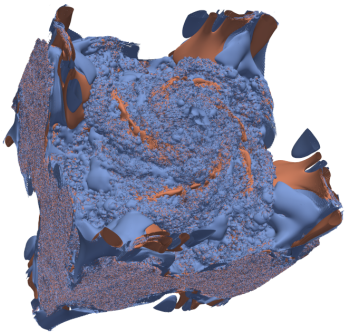
Institute of Mathematics, University of Augsburg, Germany

Collaboration with Momme Allalen, Niklas Fehn, Katharina Kormann, Karl Ljungkvist, Peter Munch, Dmytro Sashko

NHR PerfLab Seminar

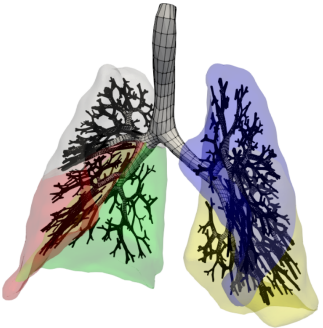
Application areas

Application: Turbulent flows



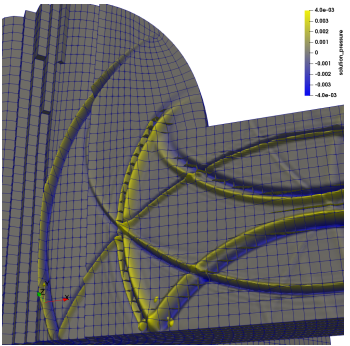
- Incompressible Navier-Stokes
- Fast solution of Poisson equation
- Multigrid

Application: Coupled flows



- Simulation of flow in biomedical settings
- Navier-Stokes coupled to transport
- Poisson + multigrid

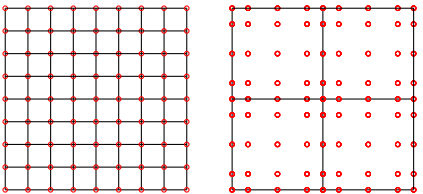
Application: Acoustics



- Acoustic wave equation
- Explicit time stepping for DG

Efficient discretizations

- Desire 1: Low number of degrees of freedom $\rightarrow$ high accuracy per unknown
- Desire 2: Methods applicable to general geometries
- Challenge: High-Reynolds number incompressible flows develop fine-scale features
  - Need to track solution over long times
  - Need accurate “dispersive” behavior
- Solution: High-order finite element methods
  - Geometrically flexible by use unstructured meshes
  - Hexahedral meshes preferred
  - High-order methods add unknowns inside elements
  - Range of attractive degrees: $p = 3, \ldots, 7$
  - Discontinuous Galerkin especially attractive: upwind fluxes



degree 1, 8^2 mesh degree 4, 2^2 mesh

Stability of discretization

- Stable schemes for fluid dynamics need problem-adapted discretizations
- Advanced mathematical ingredients to ensure stability
- As an example, for incompressible Navier–Stokes must fulfill
  - inf–sup condition (e.g. Taylor–Hood finite element)
  - point-wise divergence-free velocity field needed for energy stability and pressure robustness[1]
- For $H(\text{div})$ velocity field, L_2 conforming **discontinuous Galerkin discretization** of pressure is the **natural space**

[1] Fehn, Kronbichler, Lehrenfeld, Lube, Schroeder, High-order DG solvers for underresolved turbulent incompressible flows: A comparison of L^2 and $H(\text{div})$ methods, Int. J. Numer. Meth. Fluids 91, 2019

Outline

Accelerating FEM with matrix-free methods

Enhancing data locality in conjugate gradient solver

Algorithms for adaptive meshes

Performance of multigrid solver

Summary

Outline

Accelerating FEM with matrix-free methods

Enhancing data locality in conjugate gradient solver

Algorithms for adaptive meshes

Performance of multigrid solver

Summary

Developing fast finite element solvers

- Complex iterative solvers, block preconditioners, multigrid
- Usually 60–95 % of time spent in matrix-vector products
- Large structured/unstructured meshes and adaptivity: traditionally solved with sparse matrices at 0.16–0.25 Flop/Byte
- **Bottleneck:** memory-limited, only 1–5 % of arithmetic peak
- **Goal**: change algorithm by reading less from memory and computing more on the fly

Matrix-based algorithm in finite elements

- Loop through cells and assemble from cell matrices into global sparse matrix
- Apply action of sparse matrix in iterative solver (CG, GMRES, . . .)
- Build preconditioner from sparse matrix
- Traditionally considered necessary for unstructured, adaptively refined meshes

Matrix-based vs matrix-free algorithms

Matrix-based algorithm in finite elements

- Loop through cells and assemble from cell matrices into global sparse matrix
- Apply action of sparse matrix in iterative solver (CG, GMRES, . . .)
- Build preconditioner from sparse matrix
- Traditionally considered necessary for unstructured, adaptively refined meshes

Matrix-free algorithm

- Apply action of discretized operator within iterative solver on the fly
- Classical approach: Assume locally structured mesh, coefficients constant or slowly varying
  - Leads to set of a few **stencils**, similar optimization steps as for FDM
  - Research by U. Rüde and co-workers
- My choice: **Compute integrals** underlying FEM operator with fast integration, tailored to unstructured meshes & variable coefficients, but preconditioner selection limited
  - Matrix diagonal feasible $\rightarrow$ multigrid with special smoothers feasible
  - ILU on original matrix not feasible because performance will not be any better

Matrix-free operator evaluation by fast integration

Matrix-vector product

matrix-based:

$$\begin{cases} A = \sum_{e=1}^{N_{\text{el}}} G_e^{\mathsf{T}} A_e G_e & \text{(assembly)} \\ v = Au & \text{(matrix-vector product within iterative solver)} \end{cases}$$

matrix-free:

$$v = \sum_{e=1}^{N_{\text{el}}} G_e^{\mathsf{T}} A_e (G_e u)$$

implication: assembly facilities within iterative solvers

Matrix-free evaluation of FEM Laplacian

- $v = 0$
- loop over elements $e = 1, \ldots, N_{\text{el}}$
  - (i) Extract local vector values: $u_e = G_e u$
  - (ii) Apply operation locally by integration: $v_e = A_e u_e$, **do not form** A_e, compute its action by FEM integrals, $v_e = S_e^{\mathsf{T}} D_e S_e u_u$
  - (iii) Sum results from (ii) into the global solution vector: $v = v + G_e^{\mathsf{T}} v_e$

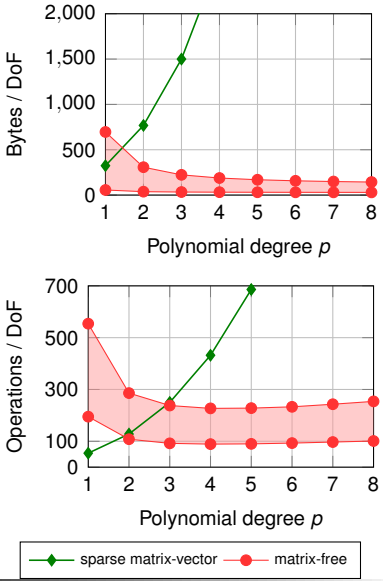
M. Kronbichler, K. Kormann, A generic interface for parallel finite element operator application. *Comput. Fluids* 63:135–147, 2012

M. Kronbichler, K. Kormann, Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM TOMS* 45(3), 29, 2019

Included in `deal.II` finite element library, `www.dealii.org`

Opportunities of matrix-free algorithms

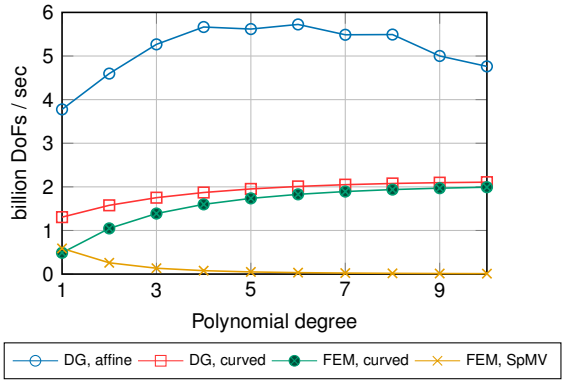
- **Sparse matrix**
  - memory: 8 bytes entry + 4 bytes for index, i.e., 12 byte per nnz + 8 byte per row + vectors
  - 1 multiply + 1 add / nnz
- High-order methods get increasingly expensive due to denser coupling
- **Matrix-free**
  - vectors + affine geometry (lower bound) versus separate Jacobian at each quadrature point (upper bound)
  - sum factorization for $\mathcal{O}(p)$ complexity per DoF
- Work for matrix-free goes down from $p = 1$ to $p = 3$ due to fewer DoFs shared by several elements



Matrix-free vs. matrix-based methods

- Performance of matrix-vector product essential for iterative solvers
- Sparse matrices unsuitable for higher orders $p \geq 2$ on modern hardware due to **memory-bandwidth** limit
- Matrix-free algorithm successful in trading computations for less memory transfer
  - Software: Specify operation at quadrature points
  - Combine with reference cell interpolation matrices
  - Indirect access into vector entries for continuous FEM

Throughput of matrix-vector product (unknowns processed per second) of 3D Laplacian



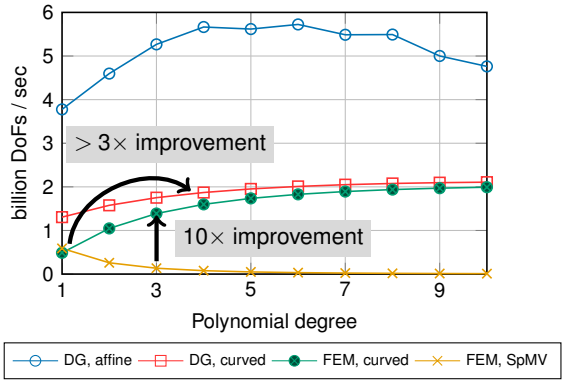
System: 1 node of 2×24 cores of Intel Xeon Platinum 8174 (Skylake)
Memory bw: 205 GB/s, arithmetic peak 3.5 TFlop/s

Kronbichler, Kormann: A generic interface for parallel cell-based finite element operator application. *Comput Fluids* 63:135–147, 2012
Kronbichler, Wall: A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers. *SISC* 40(5):A3423–48, 2018
Kronbichler, Kormann: Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM TOMS* 45(3):29/1–40, 2019

Matrix-free vs. matrix-based methods

- Performance of matrix-vector product essential for iterative solvers
- Sparse matrices unsuitable for higher orders $p \geq 2$ on modern hardware due to **memory-bandwidth** limit
- Matrix-free algorithm successful in trading computations for less memory transfer
  - Software: Specify operation at quadrature points
  - Combine with reference cell interpolation matrices
  - Indirect access into vector entries for continuous FEM

Throughput of matrix-vector product (unknowns processed per second) of 3D Laplacian



System: 1 node of 2 × 24 cores of Intel Xeon Platinum 8174 (Skylake)
Memory bw: 205 GB/s, arithmetic peak 3.5 TFlop/s

Kronbichler, Kormann: A generic interface for parallel cell-based finite element operator application. *Comput Fluids* 63:135–147, 2012
Kronbichler, Wall: A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers. *SISC* 40(5):A3423–48, 2018
Kronbichler, Kormann: Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM TOMS* 45(3):29/1–40, 2019

Explanation of performance difference: Modeling

Matrix-based

- **Ideal memory access matrix-based:** A single load to sparse matrix (12 bytes per DP entry), a single load to source, a single store
- **Balance:** 0.16 FLOPs/Byte

Matrix-free

- **Ideal memory access matrix-free:** A single load to source, a single store (or load+store with read-for-ownership)
- **Arithmetics:** Around 120–250 operations per DoF
- **Balance:** 1.5–10 FLOPs/Byte
- **Caches** beneficial because they can hold neighboring data that is re-used

Matrix-based must only consider memory, matrix-free must consider both compute and memory access!

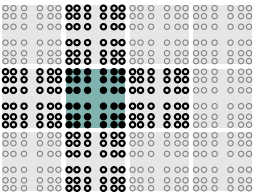
Data access patterns in DG methods

Typical DG discretization with matrix-free computations involves access akin to a block-finite difference stencil

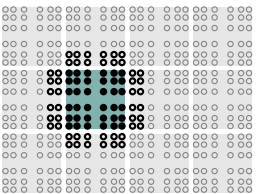
- Choice 1 (spectral elements): Minimize arithmetic operations (diagonal mass matrix)
  - Access all $(p+1)^d$ unknowns on neighbors
  - Interpolation matrix from neighbor points to values on face
  - Around 170–200 Flop/DoF
- Choice 2: Basis with minimal non-zero values across faces $\phi_i(0) \neq 0$ and $\phi_i'(0) \neq 0$ (Hermite)
  - Less access into neighbor
  - More direct use of values on faces
  - More arithmetic operations, 190–250 Flop/DoF
- In figure, highlighted area (in black/green) is interleaved with computation of integrals

Choice 2 gives (much) better performance

Data access spectral $p = 5$



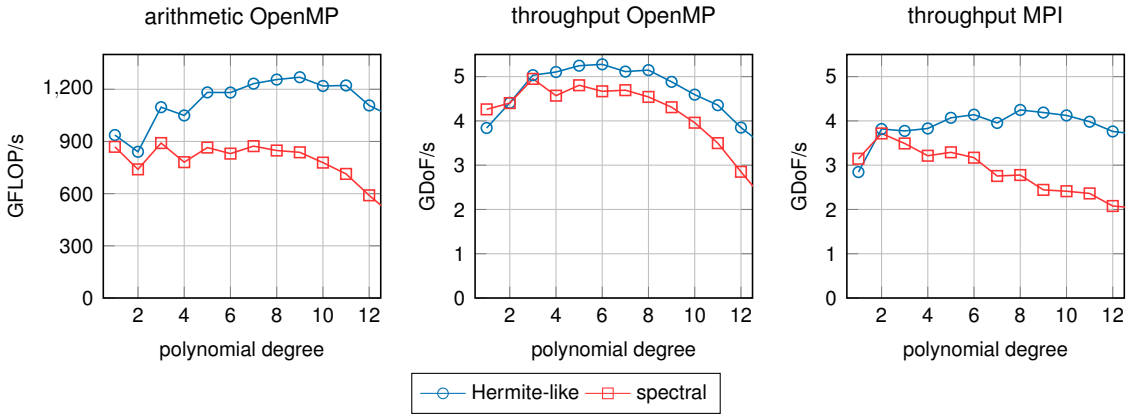
Data access Hermite-like $p = 5$



○ only read
● read + write

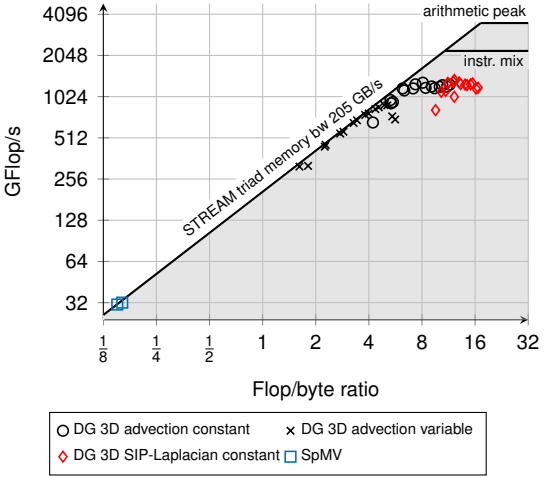
Node-level performance for DG Laplacian (including RFO transfer)

- Evaluate on 2×24 cores of Intel Skylake Platinum, 3D Laplacian, affine mesh
- Compact Hermite-like basis faster than spectral despite more work $\rightarrow$ data access more important than arithmetic work
- Compare: Simple copy from input to output (no RFO) gives 11 GDoF/s



Roofline evaluation: Modeled data transfer

- Minimize arithmetic operations
  - Sum factorization
  - Avoid full derivative matrices $S \otimes D, D \otimes S$ by transforming to collocation space with $I \otimes D_{\text{co}}, D_{\text{co}} \otimes I$
  - Symmetry: even-odd decomposition
  - Vectorization
- Data access optimizations
  - Use single loop to compute all data (DG cell + face integrals)
  - Grid traversal: neighbor data hits cache
  - On CPUs: re-compute metric terms on the fly from node positions
  - Use lower/intermediate order geometry



System: 1 node of 2×24 cores of Intel Xeon Platinum 8174 (Skylake)

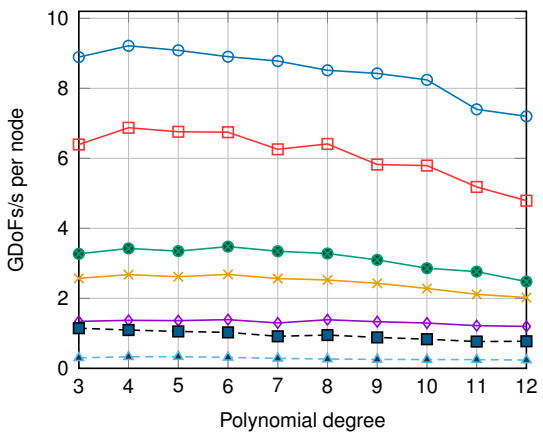
Kronbichler, Kormann: A generic interface for parallel cell-based finite element operator application. *Comput Fluids* 63:135–147, 2012
Kronbichler, Kormann: Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM TOMS* 45(3):29/1–40, 2019

Architectural comparison: hardware

- **Ice Lake 72C**: 2× 36 core Intel Xeon Platinum 8360Y, nominal power 250 W, released in 2021 (**Fritz cluster**)
- **Skylake 48C**: 2× 24 core Intel Xeon Platinum 8174, nominal power 205 W, released in 2017 (**SuperMUC-NG**)
- **Broadwell 40C**: 2× 20 core Intel Xeon E5-2698 v4, nominal power 135 W, released in 2016
- **Haswell 28C**: 2× 14 core Intel Xeon E5-2697 v3, nominal power 145 W, released in 2014 (**SuperMUC Phase 2**)
- **Haswell 16C**: 2× 8 core Intel Xeon E5-2630 v3, nominal power 85 W, released in 2014
- **Sandy Bridge 16C**: 2× 8 core Intel Xeon E5-2680, nominal power 130 W, released in 2012 (**SuperMUC Phase 1**)
- **Opteron 16C**: 2× 8 core AMD Opteron 6128, nominal power 80 W, released in 2010

Architectural comparison: performance and energy efficiency

Matrix-vector product of 3D Laplacian with DG-SIP, OpenMP parallelization



Outline

Accelerating FEM with matrix-free methods

Enhancing data locality in conjugate gradient solver

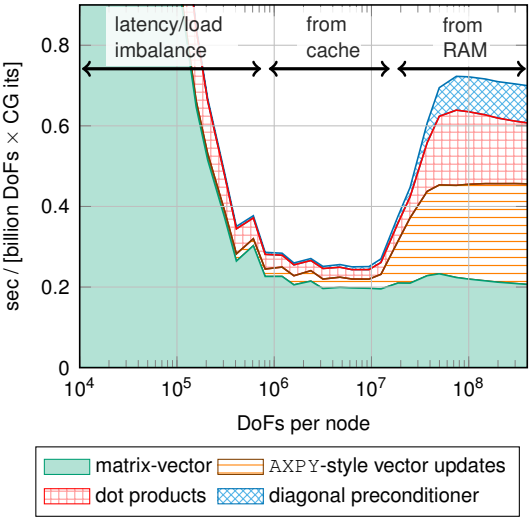
Algorithms for adaptive meshes

Performance of multigrid solver

Summary

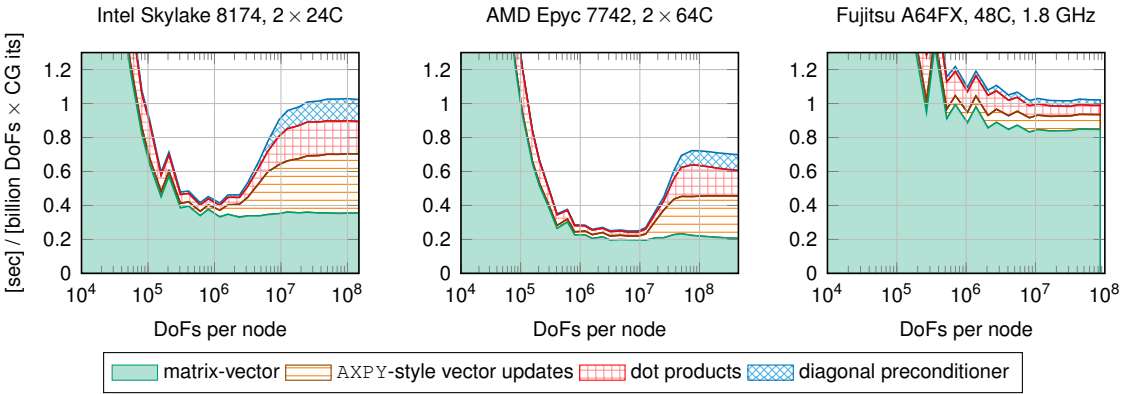
Beyond the matrix-vector product

- CEED benchmark problem BP4
  - 3D Poisson, deformed geometry, continuous elements
  - Conjugate gradient + diagonal preconditioner
  - Metric terms computed on the fly from tri-quadratic geometry
- 1 node of dual-socket AMD Epyc 7742
  - bandwidth from RAM: 400 GB/s theory, $\sim$ 300 GB/s achievable
  - 4.6 TFlop/s
- Matrix-vector product no longer dominant operation for large sizes
- Vector operations take significant share of time when operated from RAM



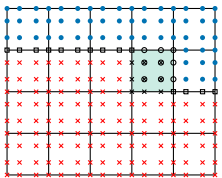
Behavior dependent on hardware?

- BP4 problem, Poisson with $p = 5, q = 7$, time of one iteration with CG
- Intel Xeon Skylake, 48 cores, 205 GB/s, 3.5 TFlop/s at 2.3 GHz, 2×205W
- AMD Epyc Zen 2, 128 cores, 295 GB/s, 4.6 TFlop/s at 2.25 GHz, 2×225W
- Fugaku's Fujitsu A64FX, 48 cores, 830 GB/s, 2.8 TFlop/s at 1.8 GHz, ~ 130W



Reduce impact of vector operations in CG

- Idea: Combine arithmetic intensive matrix-vector product with memory intensive vector operations
- Fuse loops with repeated vector access (e.g. AXPY and dot product)
- Classical conjugate gradient contains several barriers that prevent effective fusion
- Variant on the right: Redundant computation of some information, several application of preconditioner (diagonal = cheap, no long-range dependency)
- Run vector updates before $\mathbf{Ap}_k$ first touches vector entries, dot products after $\mathbf{Ap}_k$ last touches vector entries
  - Access 87% of vector entries **only once per CG iteration**



while not converged **do**

if k even **then**

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \alpha_{k-1}\mathbf{p}_{k-1} + \frac{\alpha_{k-2}}{\beta_{k-2}}\left(\mathbf{p}_{k-1} - \mathbf{M}^{-1}\mathbf{r}_{k-1}\right)$$

end if

$$\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_{k-1}\mathbf{v}_{k-1}$$

$$\mathbf{p}_k = \mathbf{M}^{-1}\mathbf{r}_k + \beta_{k-1}\mathbf{p}_{k-1}$$

fused loop
3.83 read
2.5 write

$$\mathbf{v}_k = \mathbf{A}\mathbf{p}_k$$

$$\gamma_k = \mathbf{r}_k^T\mathbf{r}_k$$
$$a_k = \mathbf{p}_k^T\mathbf{v}_k$$
$$b_k = \mathbf{r}_k^T\mathbf{v}_k$$
$$c_k = \mathbf{v}_k^T\mathbf{v}_k$$
$$d_k = \mathbf{r}_k^T\mathbf{M}^{-1}\mathbf{r}_k$$
$$e_k = \mathbf{r}_k^T\mathbf{M}^{-1}\mathbf{v}_k$$
$$f_k = \mathbf{v}_k^T\mathbf{M}^{-1}\mathbf{v}_k$$

fused loop
3.3 read
0 write

$$\alpha_k = \frac{d_k}{a_k}$$

if $\sqrt{\gamma_k - 2\alpha_k b_k + \alpha_k^2 c_k} < \varepsilon$ **then**

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k$$

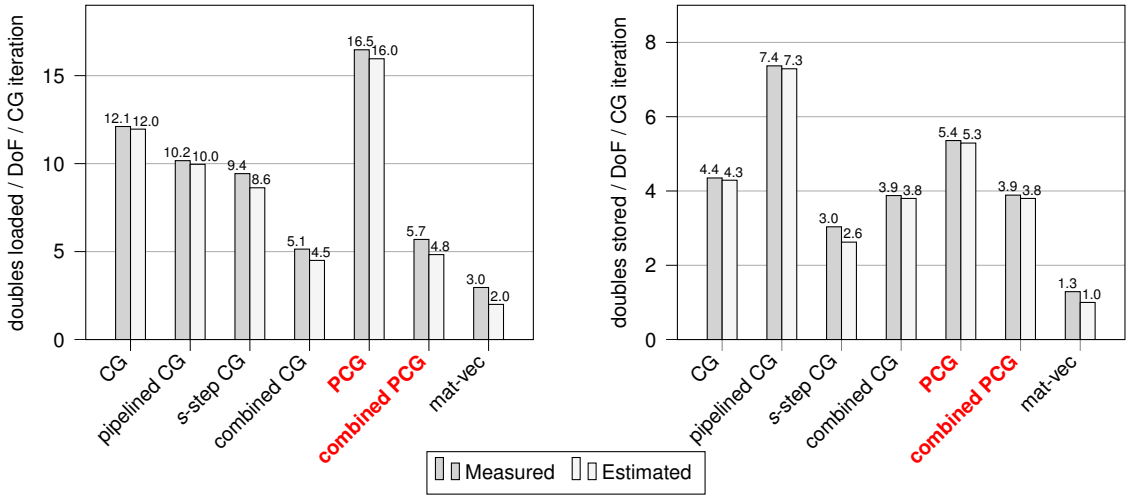
end if

$$\beta_k = \frac{d_k - 2\alpha_k e_k + \alpha_k^2 f_k}{d_k}$$

end while

Improvement of combined CG algorithm

- Analyze load and store behavior of CG variants



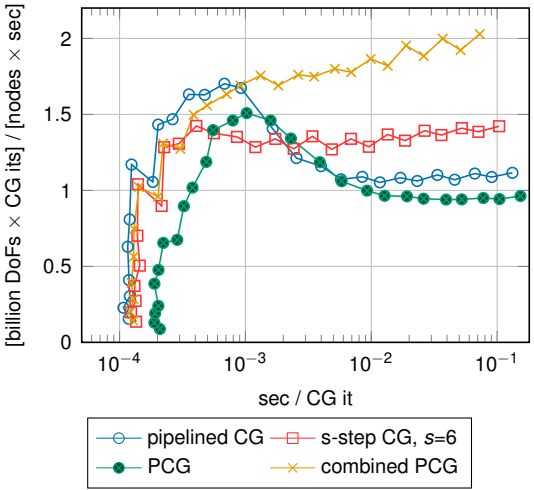
Kronbichler, Sashko, Munch: Enhancing data locality of the conjugate gradient method for high-order matrix-free finite-element implementations. *IJHPCA*, 2022

Summary key ideas

- Classical wavefront/diamond blocking difficult for high-order schemes due to wide stencil $\rightarrow$ especially with MPI, additional transfer ruins possible performance gains
  - Initial test: matrix power kernel with $k = 3$ steps runs with $\sim$20% lower throughput than single matrix-vector product
  - Reason: data locality of evaluation of integrals destroys data locality
- Proposed method interleaves vector operations before or after a single matrix-vector into matrix-vector product
  - Utilize that operator evaluation not completely memory limited
  - Reduce memory access before and after loop

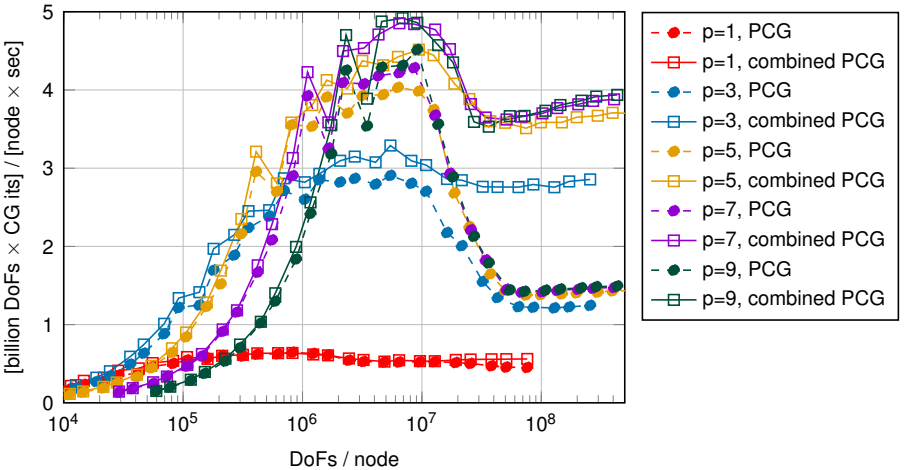
Comparison of throughput on 512 nodes

- BP4: vector-valued Poisson
- Reduction of memory transfer increases throughput by almost 2× on 512 nodes
- New combined CG method runs more quickly also near the scaling limit due to a single `MPI_Allreduce`
- Scaling limit on 512 nodes similar to pipelined CG and *s*-step CG (without preconditioner)



Behavior for different polynomial degrees

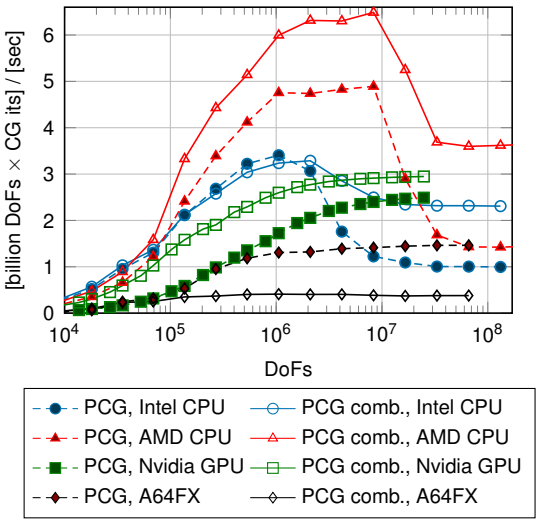
Run BP4 on 2×64 core AMD Epyc 7742 for degrees $p = 1, 3, 5, 7, 9$



Conclusion: New methods are $2-3\times$ faster on CPUs with caches!

Cross-platform comparison for BP5

- 3D Poisson problem, GLL quadrature, $p = 5, q = 6$
- Intel CPU: Xeon Skylake 8174, 48 cores
- AMD CPU: Epyc 7742, 128 cores
- NVIDIA V100 GPU
  - Note: US CEED group reaches 15–20% higher throughput for BP5 with plain CG
  - Combined CG run globally on vectors, no overlap into mat-vec
- Fujitsu A74FX, 48 cores
  - Note: combined CG performs really badly (vectorization dot products, latency from context shift (mat-vec vs. vector blocks)?



Node-level performance of Chebyshev smoother on Skylake

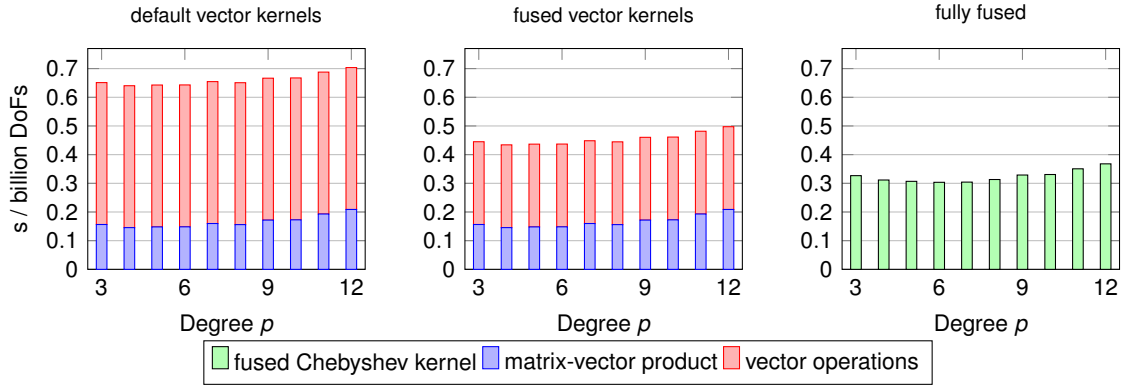
Chebyshev smoother in multigrid at iteration j:

$$t^{(j)} = \rho^{(j)} t^{(j-1)} + \theta^{(j)} P^{-1} \left(A u^{(j-1)} - b \right),$$

$$u^{(j)} = u^{(j-1)} - t^{(j)},$$

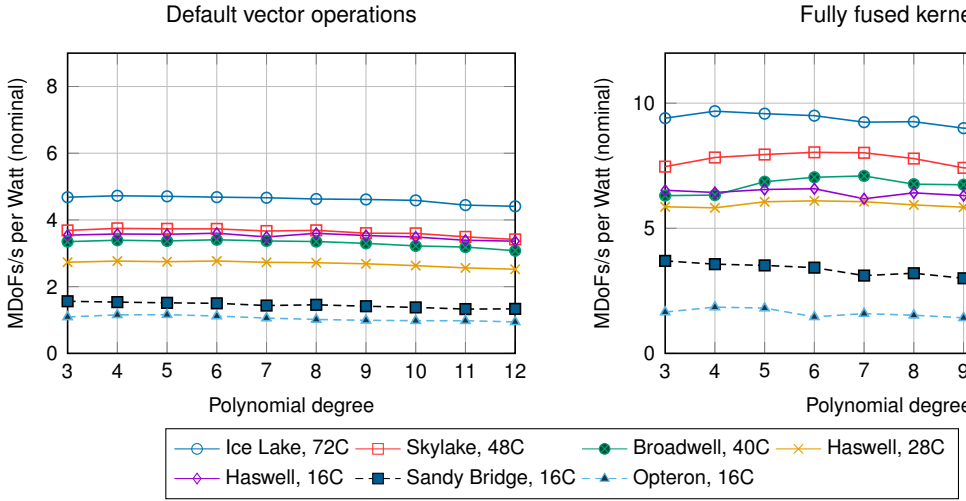
Inner preconditioner: $P = \mathrm{diag}(A)$

- **Default vector kernel:** One vector update at a time, `daxpy` style
- **Fused vector kernel:** Separate mat-vec, all vector updates within single loop
- **Fully fused:** Apply vector updates within loop over cells in DG operator



Power efficiency: default vs merged vector operations

Architectural comparison of one Chebyshev smoother iteration on 3D Laplacian with DG-SIP, OpenMP parallelization



Outline

Accelerating FEM with matrix-free methods

Enhancing data locality in conjugate gradient solver

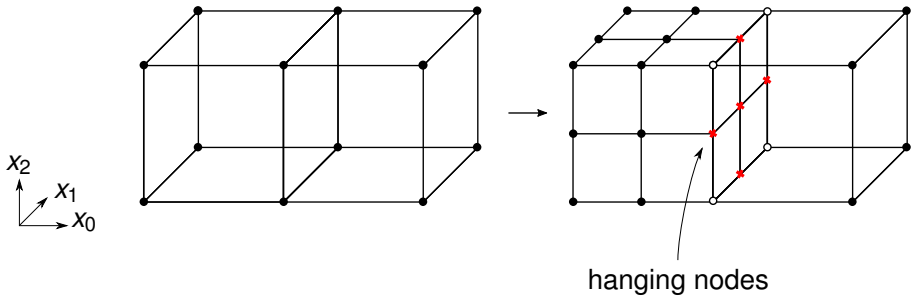
Algorithms for adaptive meshes

Performance of multigrid solver

Summary

Better algorithms for hanging nodes

Example: 2 coarse cells; scalar, linear Lagrange elements ($p = 1$); non-conformal refinement

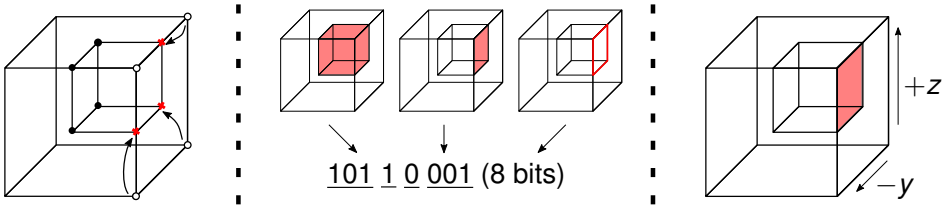


- Task: guarantee H^1 conformity
- Traditional algorithm in deal.II via general-purpose sparse matrix: $x_i = C_{ij} x_j + b_i$

... locally dense $\mathcal{O}(k^{2(d-1)})$

Shephard: Linear multipoint constraints applied via transformation as part of a direct stiffness assembly process. IJNME, 1984

Efficient algorithm for hanging node constraints



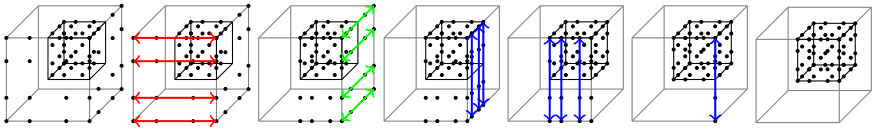
1) update DoF map 2) encode refinement config. 3) inplace interpolation

- Idea: Use sum-factorization algorithms for interpolation from coarse DoFs to refined basis representation
- Challenge: 137 refinement configurations, need to select appropriate algorithm

P.F. Fischer et al: Spectral element methods for transitional flows in complex geometries. J. Sci. Comput., 2002
K. Ljungkvist: Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes. In SpringSim (HPC), 2017

Implementation on the CPU

- Step 1: split application of constraints into general-purpose constraints (e.g. Dirichlet, no-normal flux) and hanging nodes $\rightarrow \mathscr{C}_e^{\text{HN}} \circ \mathscr{C}_e^{\text{GP}} \circ \mathscr{G}_e$
- Step 2: Merge general-purpose constraints with gather of unknowns on cell, $\mathscr{C}_e^{\text{GP}} \circ \mathscr{G}_e$, as proposed in [2]
- Inplace interpolation on each object in sequence:

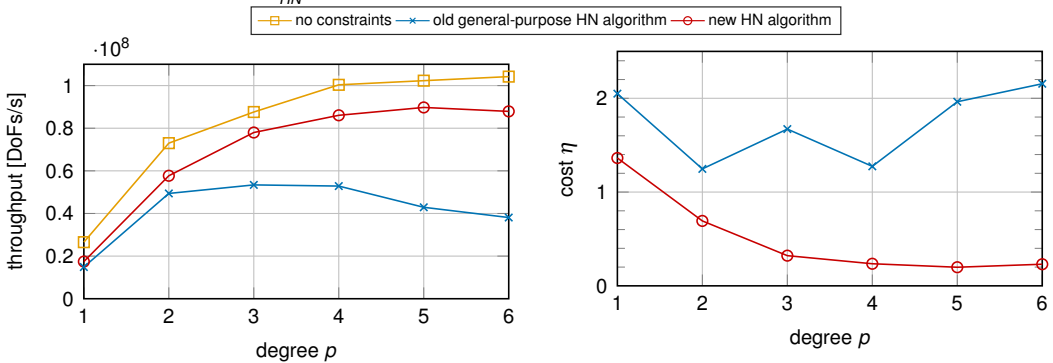


- Requirements:
  1. Determine which objects are constrained
  2. Determine which 1D interpolation to use
- Costs: $\mathscr{O}((d-1)k^d)$
- Done on GPU with masking and appropriate access to array
- Downside on CPU: several `if` statements and control logic

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

Result: serial (shell)

Cost definition: $\eta_1 \sim \frac{T_{HN} - T_{NO}}{T_{HN}} - 1^3$

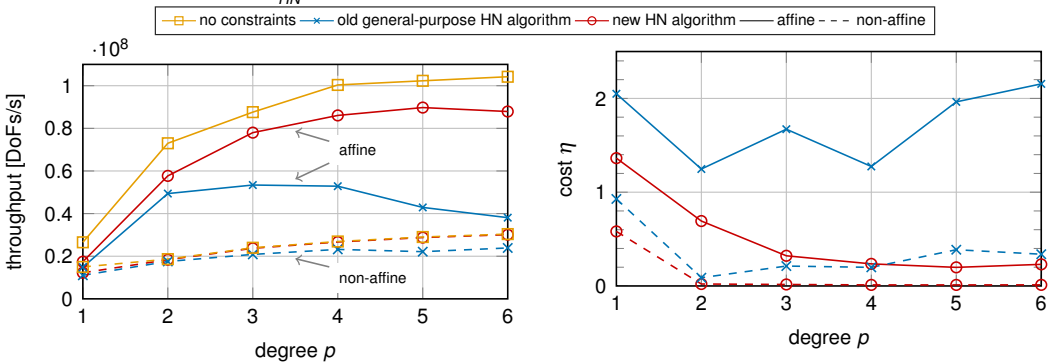


- Implication for parallel simulations: less load imbalance
- Non-affine constraint application can be hidden behind loading the metric terms

[3] P. Munch, K. Ljungkvist, M. Kronbichler: Efficient Application of Hanging-Node Constraints for Matrix-Free High-Order FEM Computations on CPU and GPU, *Proceedings ISC High Performance 2022 (LNCS 13289)*

Result: serial (shell)

Cost definition: $\eta_1 \sim \frac{T_{HN} - T_{NO}}{T_{HN}} - 1$[3]

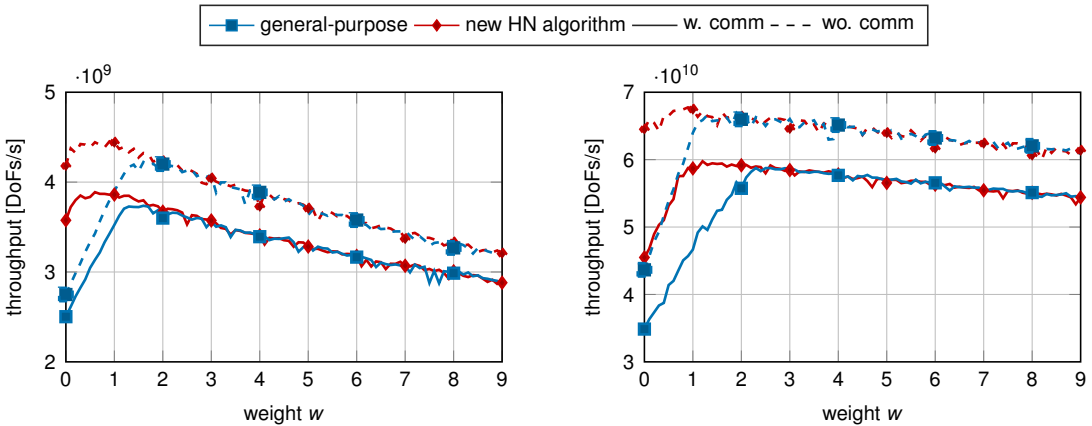


- Implication for parallel simulations: less load imbalance
- Non-affine constraint application can be hidden behind loading the metric terms

[3] P. Munch, K. Ljungkvist, M. Kronbichler: Efficient Application of Hanging-Node Constraints for Matrix-Free High-Order FEM Computations on CPU and GPU, *Proceedings ISC High Performance 2022 (LNCS 13289)*

Performance in parallel

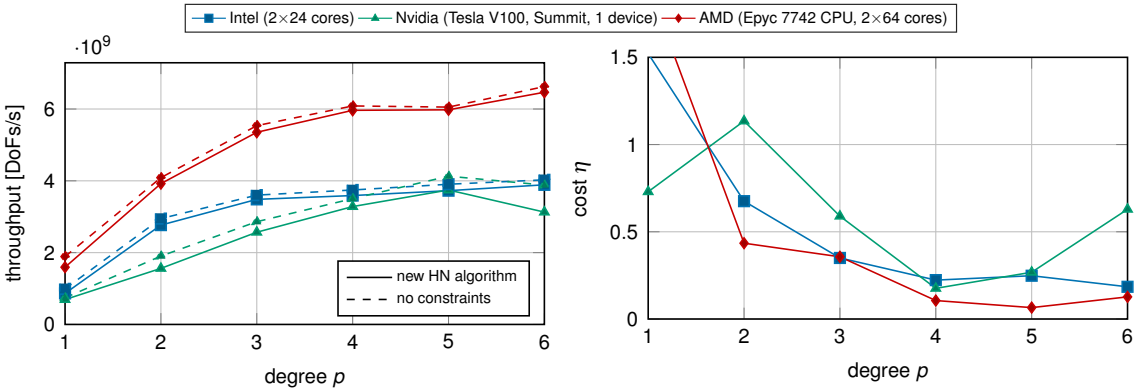
- Even with optimized sum factorization, hanging-node constraints need more time
- To optimize performance, weight cells with hanging nodes with factor $w > 0$ as $1 + w$



1 node (48 cores)

16 nodes (768 cores)

Results and cross-platform validation (shell, parallel)



- GPU overhead looks higher, but is actually lower, because we execute the HN algorithm on all cells for the GPU

Outline

Accelerating FEM with matrix-free methods

Enhancing data locality in conjugate gradient solver

Algorithms for adaptive meshes

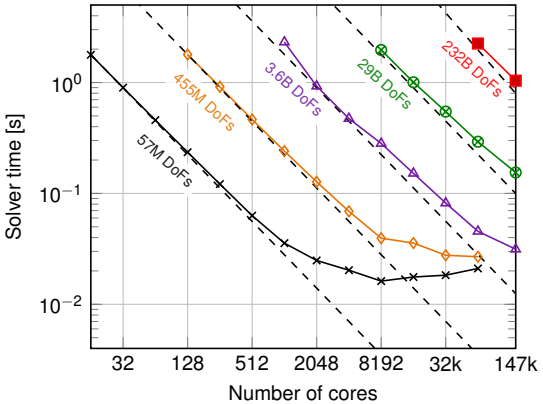
Performance of multigrid solver

Summary

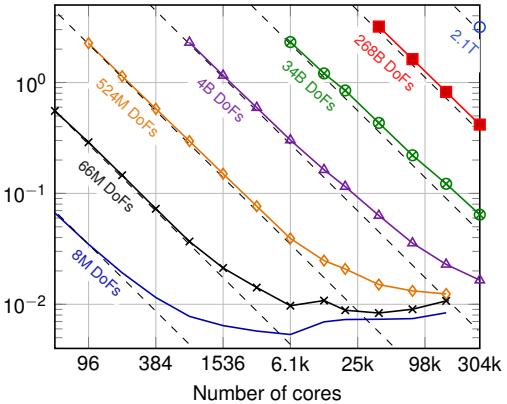
Scaling of multigrid for the Poisson equation

Geometric multigrid with deal.II and matrix-free implementations, reduce residual by 10^{-3}

Continuous FEM, degree $p = 3$, Intel Xeon Sandy Bridge (SuperMUC) from 2013, up to 9,216 nodes



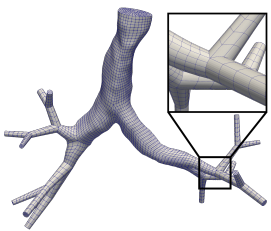
Discontinuous Galerkin, $p = 4$, Intel Xeon Skylake (SuperMUC-NG) from 2019, up to 6,336 nodes



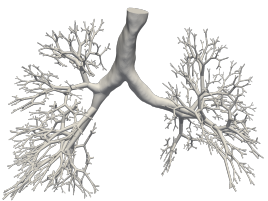
$\sim 1.2\times$ throughput per core (despite DG vs FEM), $2\times$ lower scaling limit

Multigrid on complicated meshes

- Simulation of air flow in human respiratory system
- Much more complicated geometry
- All-hex mesh, deformed from STL file
- Adaptive mesh with hanging nodes
- Polynomial degree $p = 3$
- # CG iterations with hybrid MG: 7

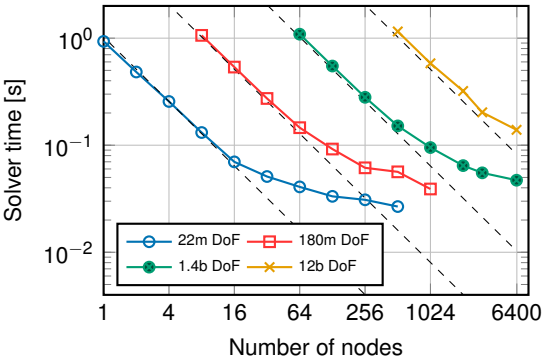


Exemplary mesh



Computed geometry

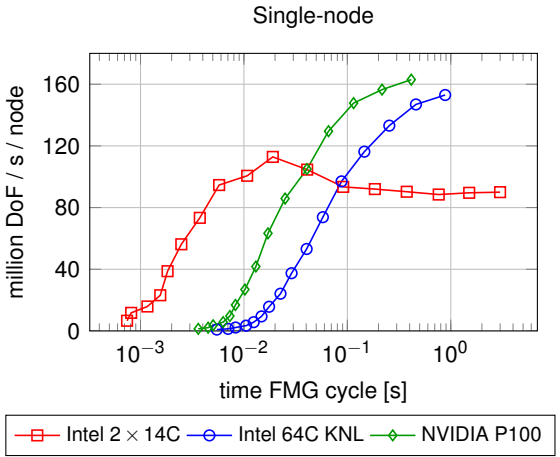
Simulation of Poisson with tolerance 10^{-3} on SuperMUC-NG machine (Intel Xeon Platinum 8174, 48 cores / node)

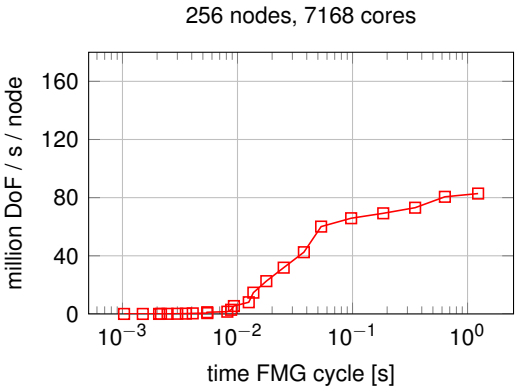


- Scaling limit 3× higher vs simple geometry
- Largest computations on 6,400 nodes subject to noise (dynamic load balancing)

Matrix-free algorithms are fast on all HPC architectures

Geometric multigrid with full multigrid cycle, Chebyshev (5,6) smoother, $\mathcal{Q}_4$ elements

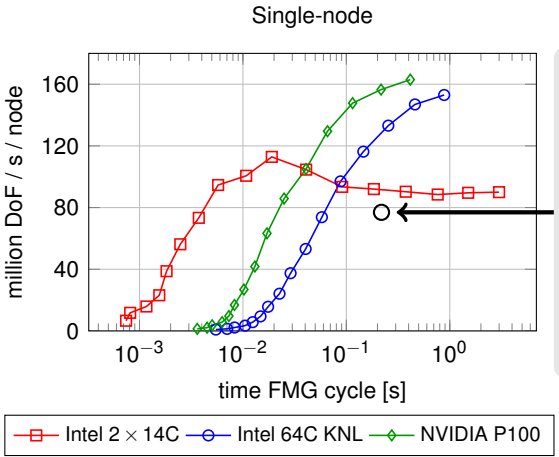




Kronbichler, Wall, A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers, *SISC* 40:A3423–48, 2018
Kronbichler, Ljungkvist, Multigrid for matrix-free high-order finite element computations on graphics processors, *ACM TOPC*, 6(1), 2019

Matrix-free algorithms are fast on all HPC architectures

Geometric multigrid with full multigrid cycle, Chebyshev (5,6) smoother, $\mathcal{Q}_4$ elements



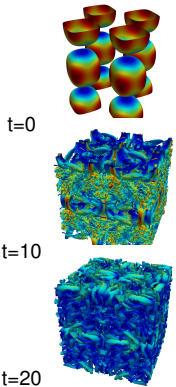
One matrix-vector product with SpMV for statically condensed finite elements with $\mathcal{Q}_4$ elements, 17m DoF, 79 million DoF/s on Broadwell

Matrix-free **solves a linear system in less time than one SpMV**!

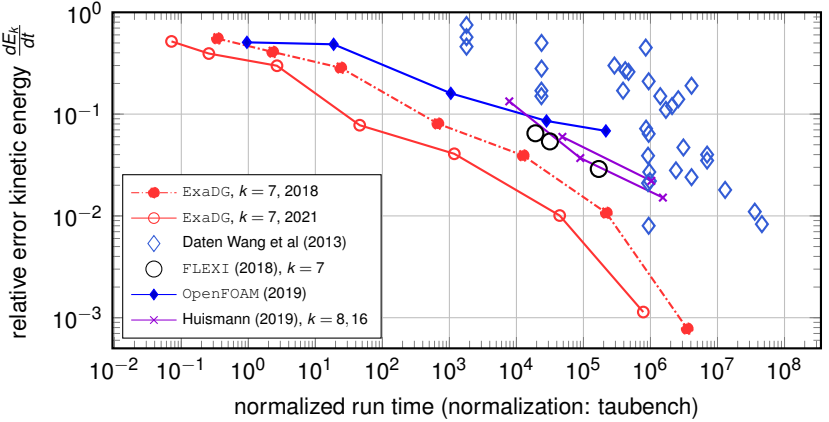
Kronbichler, Wall, A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers, *SISC* 40:A3423–48, 2018
Kronbichler, Ljungkvist, Multigrid for matrix-free high-order finite element computations on graphics processors, *ACM TOPC*, 6(1), 2019

Impact of matrix-free algorithms on CFD application

3D Taylor–Green vortex at Re = 1600: iso-contours of q-criterion (value 0.1) colored by velocity magnitude



Result of my research group around **10x faster** than all results of Wang et al. (2013), normalized run time



Wang, Fidkowski et al., High-order CFD methods: current status and perspective, *Int. J. Numer. Meth. Fluids* 72(8), 2013

Fehn, Wall, Kronbichler, Efficiency of high-performance discontinuous Galerkin spectral element methods for under-resolved turbulent incompressible flows, *Int. J. Numer. Meth. Fluids* 88, 2018

Outline

Accelerating FEM with matrix-free methods

Enhancing data locality in conjugate gradient solver

Algorithms for adaptive meshes

Performance of multigrid solver

Summary

Summary

- Fast matrix-free methods for high-order elements
- Scalable multigrid infrastructure
- Reach 20–30% of arithmetic peak of Intel/AMD CPUs, $> 85\%$ of memory bandwidth
- Memory bandwidth bottlenecks $\rightarrow$ compute more, store (even) less
- Matrix-vector product no longer dominant
  - Fuse vector operations into matrix-vector product for conjugate gradient or multigrid smoothers
  - Fuse vector operations into operator evaluation for explicit time stepping