Supported by SPPEXA (German software for exascale computing project, DFG), project ExaDG
Supported by Bayerisches Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen (KONWIHR)

The State of Matrix-free Methods and HPC

Martin Kronbichler

Institute for Computational Mechanics
Technical University of Munich, Germany

May 26, 2020

Eighth deal.II Users and Developers Workshop

Outline

What kind of matrix-free methods are there in deal.II and why?

Use cases of matrix-free methods

Geometric multigrid

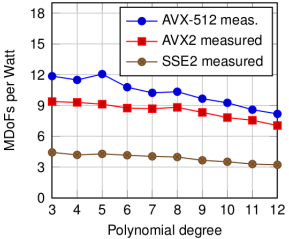
Today's big HPC theme: On the road towards exascale

HPC community is moving towards exascale (machine with 1 Exaflop/s $= 10^{18}$ floating point ops / sec)

We want to enable the user of deal.II to

- Select most efficient discretization parameters
  - Meshing (high-order curved, adaptive, . . .)
  - Polynomial degree
- Select most efficient iterative solver
  - $> 10^9$ spatial unknowns, millions of time steps $\rightarrow$ **optimal-complexity solvers** essential
  - Example: multigrid
- Select most efficient implementation for given hardware
  - Matrix-free vs matrix-based $\rightarrow$ **avoid memory wall**
  - Usage of most efficient instruction set – e.g. vectorization with AVX/AVX-512
  - Intel/AMD CPUs, NVIDIA GPUs, ARM SVE, . . .
  - Scalability to 10,000+ nodes

SuperMUC-NG supercomputer (top) (source: www.lrz.de) and efficiency of various instruction sets on Intel Skylake-X for DG operator evaluation (bottom)



Implement by ourselves or use external libraries?

External libraries

- Matrix-based linear algebra and preconditioners: PETSc, Trilinos, UMFPACK, cuSPARSE, cuSOLVERS
- Mesh partitioning: p4est, METIS
- ...

Programming frameworks & backends

- Distributed memory: MPI
- Threading (TBB; taskflow, ...?)
- CUDA for GPUs
- Vectorization: via compiler (intrinsics) or C++ standard library

Implemented inside deal.II

- Algorithms for finite elements and beyond
- Linear algebra infrastructure not available with adequate functionality or performance externally
- MPI-parallel vector `LinearAlgebra::distributed::Vector<Number>`
- SIMD abstraction class `VectorizedArray<Number>`
- Wrapper classes for unified interface to external libraries
- Help our users to concentrate on their application: separation of concerns

Outline

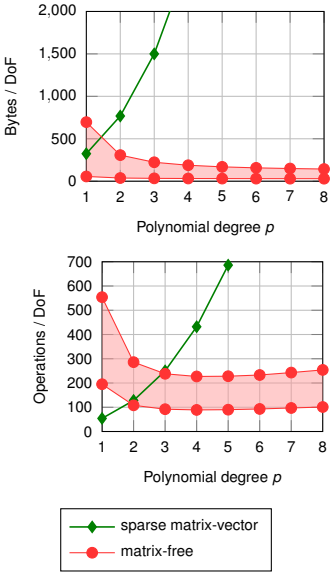
What kind of matrix-free methods are there in deal.II and why?

Use cases of matrix-free methods

Geometric multigrid

Matrix-free algorithms: hot topic in HPC

- What is limiting resource in FEM programs?
- Iterative solvers (or explicit time stepping) typically spend 60–95% of time in matrix-vector product
- Classical approach: **sparse matrix-vector product**
  - 2 arithmetic operations (mult, add) per matrix entry loaded from memory (8 byte for value + 4.x byte for index data): 0.16–0.25 Flop/Byte
  - Modern CPUs/GPUs can do 3–20 Flop/Byte
  - Performance limit is **memory bandwidth**
  - SpMV leaves resources unused
  - Memory wall: gap to memory widens over time
- Matrix-free algorithm: **transfer less, compute more**
  - Choice in deal.II: fast computation of FEM integrals
  - 1–8 Flop/Byte
  - Method of choice for higher polynomial degrees p
- Closely related to US CEED project



Matrix-free algorithm layout in deal.II

Matrix-vector product

matrix-based:

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

matrix-free:

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

implication: cell loop within iterative solvers, need optimized loops + vector access

Matrix-free evaluation of generic FEM operator

- $v = 0$
- loop over elements $e = 1, \ldots, N_{\text{el}}$
  - (i) Extract local vector values: $u_e = P_e u$
  - (ii) Apply operation locally by integration: $v_K = A_K u_K$ (**without** forming A_K)
  - (iii) Sum results from (ii) into the global solution vector: $v = v + P_e^{\mathsf{T}} v_e$

Design goals:

- Data locality for higher arithmetic intensity: single sweep through data
- Absolute performance in **unknowns per second** (DoFs/s), not maximal GFlop/s or GB/s

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

Matrix-vector product on cell by integration

Contribution of cell K to matrix-vector product: example for Laplacian

$$(A_K u_K)_j = \int_K \nabla_{\mathbf{x}} \phi_j \cdot \nabla_{\mathbf{x}} u^h d\mathbf{x} \approx \sum_q w_q \det J_q \nabla_{\mathbf{x}} \phi_j \cdot \nabla_{\mathbf{x}} u^h \Big|_{\mathbf{x}=\mathbf{x}_q}$$

$$= \sum_q \nabla_\xi \phi_j J_q^{-1} (w_q \det J_q) J_q^{-\mathsf{T}} \sum_i \nabla_\xi \phi_i u_{K,i} \Big|_{\mathbf{x}=\mathbf{x}_q}, \quad j = 1, \ldots, \text{cell\_dofs}$$

(a) Compute unit cell gradients $\nabla_\xi u^h = \sum (\nabla_\xi \phi_i) u_{K,i}$ at all quadrature points

(b) At each quadrature point, apply geometry $J_q^{-\mathsf{T}}$, multiply by quadrature weight and Jacobian determinant, apply geometry for test function J_q^{-1}

(c) Test by unit cell gradients of all basis functions and sum over quadrature points

Matrix notation:

$$v_K = A_K u_K$$
$$= S^\mathsf{T} W S u_K$$

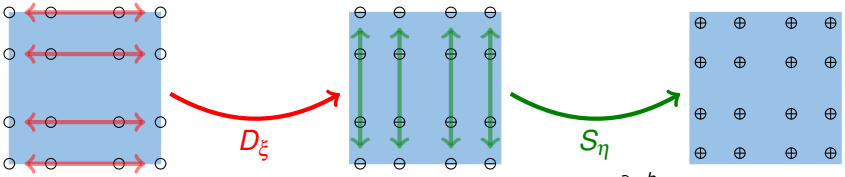
with

$S_{qi} = \nabla_\xi \phi_i \big|_{\xi_q}$

$W_{qq} = J_q^{-1} (w_q \det J_q) J_q^{-\mathsf{T}}$

Fast interpolation and integration: sum factorization via `FEEvaluation`

- Efficient evaluation of S and S^{T} matrices with structure $S_{3\mathrm{D}} = \begin{bmatrix} S_\zeta \otimes S_\eta \otimes D_\xi \\ S_\zeta \otimes D_\eta \otimes S_\xi \\ D_\zeta \otimes S_\eta \otimes S_\xi \end{bmatrix}$
- Ideas from spectral elements (1980s) for tensor product shape functions and tensor product quadrature
- Visualization of interpolation of $\frac{\partial u}{\partial \xi}$ with $\mathscr{Q}_3$ element (Lagrange basis) $S_\eta \otimes D_\xi$: successively apply 1D kernels



Vector values u_K on nodes

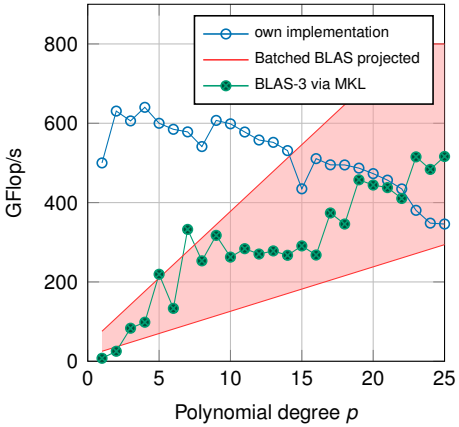
$\frac{\partial u^h}{\partial \xi}$ on quadrature points

Tensor-based evaluation reduces evaluation cost from 4^4 to 2×4^3

In general for degree p and dimension d: $\mathscr{O}((p+1)^{2d})$ to $\mathscr{O}(d(p+1)^{d+1})$

Sum factorization – implement ourselves or use performance libraries?

- Sum factorization consists of series of matrix-matrix multiplication of size $(p+1) \times (p+1)$ and $(p+1) \times (p+1)^2$ for polynomial degree p
- Standard BLAS optimized for large N
  - bad here due to function call overhead & rearrangement cost
- Series of mat-mat! Batched BLAS possible?
  - Combine small mat-mat of all elements
  - **Limit**: Data locality between mat-mat lost
  - Unclear how to best utilize data in caches (CPU) or registers (GPU)
- Own implementation with SIMD and unrolled loops (compiler via templated C++ code)



System: 2×14 Intel Broadwell 2690 v4, peak: 1.3 TFlop/s, 115 GB/s

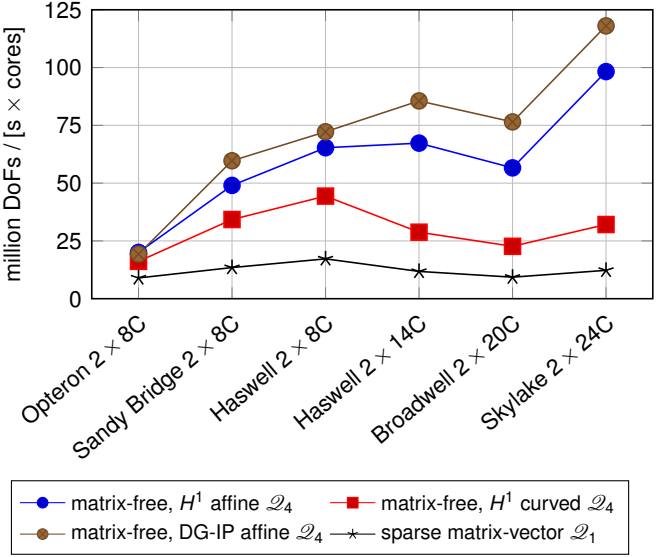
Own implementation 2–10 times faster in **interesting regime** $2 \leq p \leq 10$

Node-level performance over time

Track performance of operator evaluation **per core** over 10 years of hardware

- Baseline: sparse matrix-vector product, `FE_Q(1)`
- Continuous H^1 conforming elements, affine (arithmetic intensive) and high-order deformed (memory intensive), `FE_Q(4)`
- Discontinuous elements, affine, `FE_DGQ(4)`

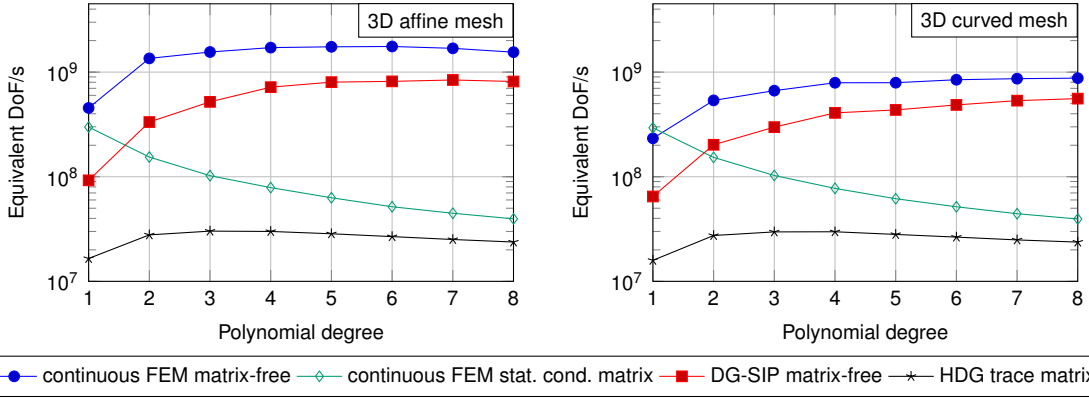
Matrix-free and SpMV within a factor 2 in 2010, **more than 10× apart today**!



Comparison versus matrix-based: matrix-free and high order wins

Continuous finite elements, DG-SIP, hybridizable discontinuous Galerkin (HDG) representing efficient sparse matrix-based scheme

Throughput of matrix-vector product measured using run time against $N_{\text{el}} k^3$ "equivalent" DoFs to make different discretizations comparable[1]



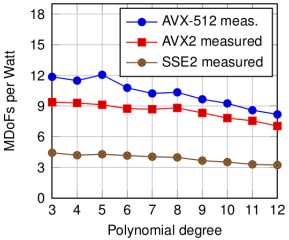
[1] Kronbichler, Wall, A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers, *SIAM J Sci Comput* 40:A3423–48, 2018

If you care about performance with $p \geq 2$, use matrix-free methods!

- Best efficiency (DoFs/s) of matrix-free operator evaluation for degrees $p = 3, 4, 5, 6$
  - **10**$\times$ faster than best matrix-based method (hybridization/static condensation) for same p
  - **3**$\times$ faster for high order than sparse mat-vec for linear FEM with same # DoFs
- step-37, step-48, step-50, step-59, step-64, step-67 tutorial programs
- **Current support** in deal.II not for all elements
  - Available: continuous `FE_Q` and discontinuous `FE_DGQ` elements, systems of these elements
  - Plan for next release: H(div) and H(curl) elements: some nodal form of `FE_RaviartThomas` and some `FE_NedelecNodal` (to be introduced)

Ingredients

- SIMD vectorization essential to use arithmetic power in CPU $\rightarrow$ better throughput / Watt
- `LinearAlgebra::distributed::Vector` rather than using PETSc or Trilinos to embed MPI-local index access & efficient ghost update ($\sim 2\times$ improvement)
- Evaluator class `FEEvaluation`



CPU code example: Cell term for Laplacian from step-37 tutorial

Evaluation of weak form $(\nabla\phi_j, \nabla u_h)_{\Omega_h}$ representing product $v = Au$

```
void cell(MatrixFree<dim> &data,
          Vector &v,
          const Vector &u,
          const std::pair<unsigned int,unsigned int> &range)
{
  FEEvaluation<dim,degree> eval (data);
  for (unsigned int cell=range.first; cell<range.second; ++cell)
    {
      eval.reinit (cell);                                  // set pointers to data
      eval.gather_evaluate(u,                              // read from source
           /*interpolate_values=*/ false,                  // sum factorization
           /*interpolate_gradients=*/ true);
      for (unsigned int q=0; q<eval.n_q_points; ++q)
        eval.submit_gradient (eval.get_gradient(q), q);// equation
      eval.integrate_scatter (false, true,                 // integrate, sum factoriz
                              v);                          // sum into result vector
    }
}
```

CPU versus GPU code

- CPU code most mature
- Basic support on GPUs available (work by Karl Ljungkvist, Bruno Turcksin, Daniel Arndt, Peter Munch)
- CPU and GPU use different implementations but provide similar interfaces
- GPU code reduces implementation to operation at quadrature point by functor with CUDA threads to parallelize over quadrature points:

```
template <int dim, int fe_degree>
__device__ void LaplaceOperatorQuad<dim, fe_degree>::
  operator()(CUDAWrappers::FEEvaluation<dim, fe_degree> *eval,
             const unsigned int                          q) const
{
  // test expression gradient(u) by nabla phi,
  // i.e., (nabla phi_i, nabla u)_K
  eval->submit_gradient(eval->get_gradient());
}
```

- step-64 tutorial program
- Could implement similar interface for CPU code: hide loop over `q`, iterator for index

Outline

What kind of matrix-free methods are there in deal.II and why?

Use cases of matrix-free methods

Geometric multigrid

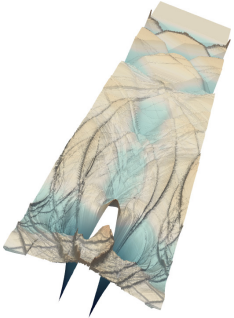
Using matrix-free algorithms in practice

- **Explicit time integration** only needs action of a right hand side / residual
- For well-conditioned matrices (mass, Helmholtz with nice parameters): **conjugate gradient solver**
- Many established preconditioners such as ILU, Gauss–Seidel, AMG, . . . not directly available in matrix-free context
  - I have used AMG by using matrix-free operator evaluation for mat-vec **and** matrix for hierarchy generation
- Matrix-free operator evaluation requires specific preconditioners
  - Active research topic
  - Most work in terms of **multigrid methods**
  - Previously: variants of Jacobi smoothers
  - Future: block smoothers via fast diagonalization method / approximate Kronecker inverses

Explicit time integration: the step-67 tutorial program

Fast integration infrastructure via `MatrixFree` and `FEEvaluation` straight-forwardly applicable to explicit time integration

- New tutorial program in 9.2 release
- Solves the Euler equations with a high-order discontinuous Galerkin scheme
- Attention mostly on wave-like phenomena because plain DG cannot deal with shocks
- Demonstrates matrix-free facilities for
  - systems of equations,
  - inverse mass matrix in DG, and
  - overlap of vector operations within `MatrixFree::cell_loop` by tracking dependencies $\rightarrow$ hides memory access cost behind computations
- Performance for degree $p = 5$ in 3D: 1252 million DoFs/s on 40 cores
  - Throughput around $3\times$ higher per core than other reported DG results (e.g. Flexi project)



step-67 cell integrator

Cell term $(\nabla \boldsymbol{v}, \boldsymbol{F}(\boldsymbol{w}_h))_K$: similar code as in Laplace case

```
void EulerOperator<dim, degree, n_points_1d>::local_apply_cell(...) const
{
  // evaluator for 'dim + 2' components
  FEEvaluation<dim, degree, n_points_1d, dim + 2, Number> eval(matrix_free);

  for (unsigned int cell = cell_range.first; cell < cell_range.second; ++cell)
    {
      eval.reinit(cell);
      eval.gather_evaluate(src, true, false);
      for (unsigned int q = 0; q < eval.n_q_points; ++q)
        {
          // weak form: (nabla v, euler_flux(w(x_q)))
          const auto w_q = eval.get_value(q);
          eval.submit_gradient(euler_flux<dim>(w_q), q);
        }
      eval.integrate_scatter(false, true, dst);
    }
}
```

step-67 inner face integrator

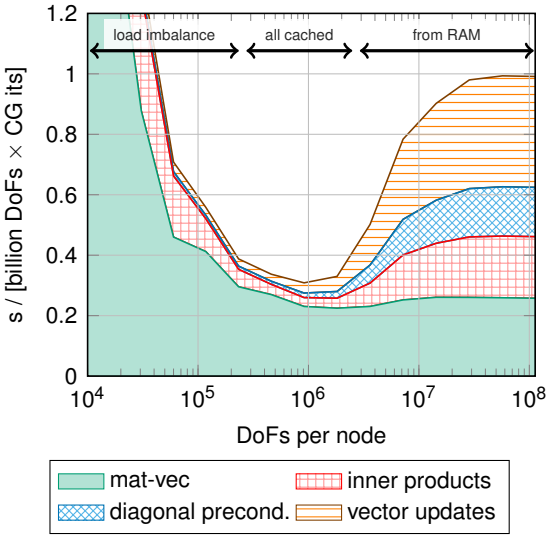
Interface term $\left\langle \boldsymbol{v}, \boldsymbol{n} \cdot \boldsymbol{F}^*(\boldsymbol{w}_h^-, \boldsymbol{w}_h^+) \right\rangle_F$: two evaluators from both elements adjacent to a face

```
void EulerOperator<dim, degree, n_points_1d>::local_apply_inner_face(...) const {
  FEFaceEvaluation<dim, degree, n_points_1d, dim + 2, Number> eval_m(mf, true);
  FEFaceEvaluation<dim, degree, n_points_1d, dim + 2, Number> eval_p(mf, false);

  for (unsigned int face = face_range.first; face < face_range.second; ++face)
    {
      eval_p.reinit(face); eval_p.gather_evaluate(src, true, false);
      eval_m.reinit(face); eval_m.gather_evaluate(src, true, false);
      for (unsigned int q = 0; q < eval_m.n_q_points; ++q)
        {
          // implement all point-wise work in separate function 'numerical_flux'
          const auto numerical_flux =
            euler_numerical_flux<dim>(eval_m.get_value(q), eval_p.get_value(q),
                                      eval_m.get_normal_vector(q));
          eval_m.submit_value(-numerical_flux, q);
          eval_p.submit_value(numerical_flux, q);
        }
      eval_p.integrate_scatter(true, false, dst);
      eval_m.integrate_scatter(true, false, dst);
    } }
```

Conjugate gradient solver, diagonal precondition: CEED benchmarks

- Benchmark problem BP5 of US exascale discretization initiative CEED
- 3D Poisson, deformed geometry, continuous elements
- conjugate gradient + diagonal preconditioner, GLL quadrature
- 1 node of dual-socket Intel Skylake
- Matrix-vector product **no longer dominant**
- Vector operations take significant time when operated from RAM

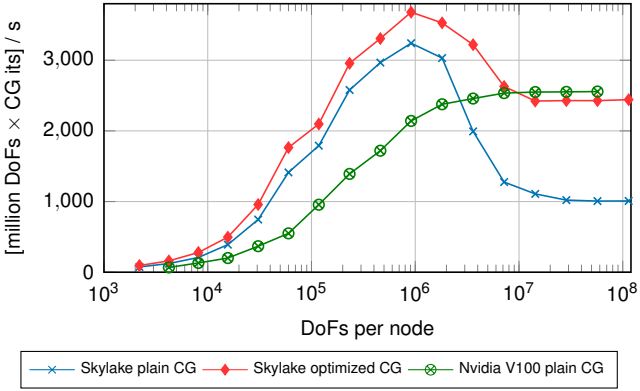


https://ceed.exascaleproject.org/bps/

Fischer, Min, Rathnayake, Dutta, Kolev, Dobrev, Camier, Kronbichler, Warburton, Świrydowicz, and Brown: Scalability of High-Performance PDE Solvers, *Int. J. High Perf. Comput. Appl.*, 2020

Solution: Fuse vector operations with mat-vec

- Solution: Data locality
- Vector operations of CG before/after touching DoFs in mat-vec via `MatrixFree::cell_loop`
- Overlap arithmetic intensive phase of mat-vec with memory-intensive vector operation



Achieved throughput 34M DoFs:

- Volta: 2560 MDoFs/s
- Intel Skylake 2 × 24C baseline: 1010 MDoFs/s
- Skylake optimized: 2420 MDoFs/s

Arithmetic:

- Volta: 586 GFlop/s
- Skylake optimized: 702 GFlop/s

Memory:

- Volta: 699 GB/s
- Skylake optimized: 191 GB/s

Note: Skylake (opt) and Volta run vastly different implementations! CPU code considerably more complex!

Outline

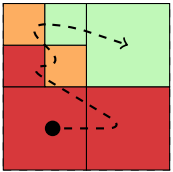
What kind of matrix-free methods are there in deal.II and why?

Use cases of matrix-free methods

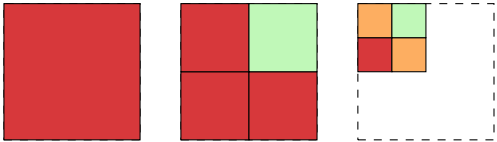
Geometric multigrid

Geometric multigrid in deal.II

- Level smoothers crucial ingredient
- Select methods where fast operator evaluation (matrix-free) is core component
  - Point-diagonal (Jacobi) or block-diagonal with tensor product inversion
  - Chebyshev iteration around these methods to improve efficiency: class `PreconditionChebyshev`
- In deal.II: flexible geometric multigrid infrastructure like *h*-MG, *p*-MG ($\rightarrow$ planned), adaptive meshes



Sample view of active mesh on 3 processors



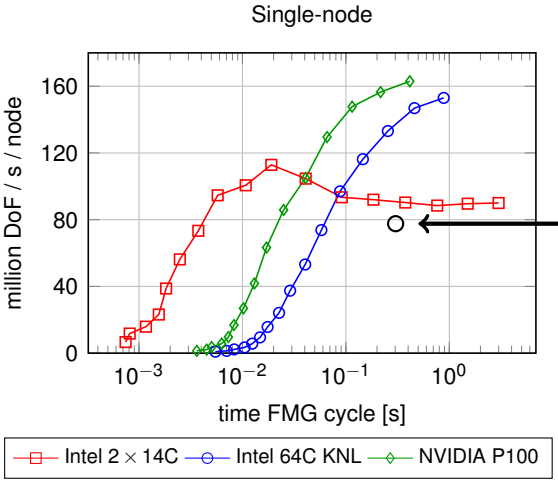
level $\ell = 0$ level $\ell = 1$ level $\ell = 2$

Clevenger, Heister, Kanschat, Kronbichler, A Flexible, Parallel, Adaptive Geometric Multigrid method for FEM, *arXiv*:1904.03317, 2019

Fehn, Munch, Wall, Kronbichler, Hybrid multigrid methods for high-order discontinuous Galerkin discretizations, *JCP*, 2020

Multigrid node-level comparison: CPU vs Xeon Phi vs GPU

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

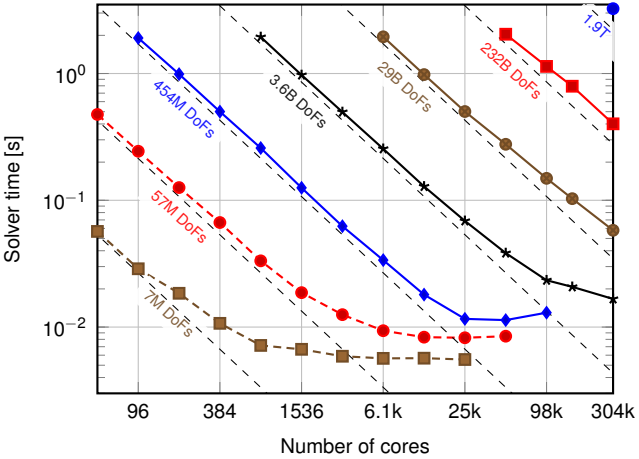
Matrix-free **solves a linear system faster than one SpMV**!

Sparse matrix **inefficient, non-HPC format** for $p \geq 3$!

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):2/1–2/29, 2019

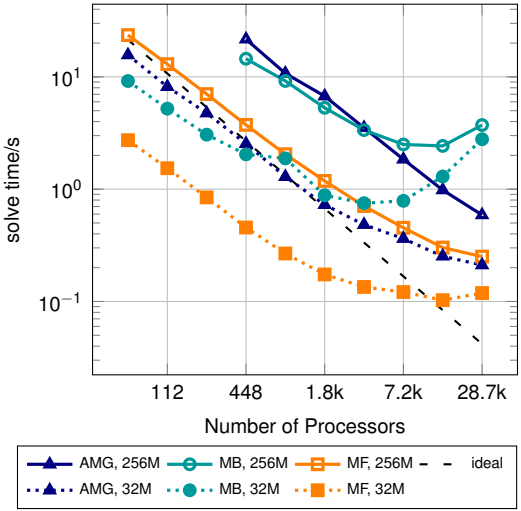
Scalability of multigrid: discontinuous elements

- Discontinuous elements
  `FE_DGQHermite<3>(5)`
- Conjugate gradient solver preconditioned by *ch* multigrid
  - DG: Chebyshev (6,6) iteration around block-Jacobi (fast diagonalization)
  - FEM: Chebyshev (6,6) around point Jacobi
  - Multigrid V-cycle in single precision
- 2 CG iterations (tolerance 10^{-3})
- Intel Xeon Platinum 8174, 48 cores per node, up to 6336 nodes (full SuperMUC-NG)



Arithmetic performance 1.9 trillion DoFs:
5.8 PFlop/s (5.5 PFlop/s in SP, 0.27 PFlop/s in DP)
180 GB/s per node (STREAM: 210 GB/s)

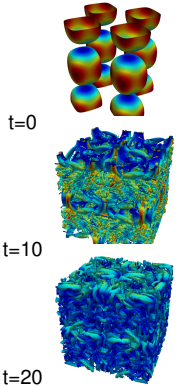
Comparison of matrix-free GMG, matrix-based GMG, and AMG



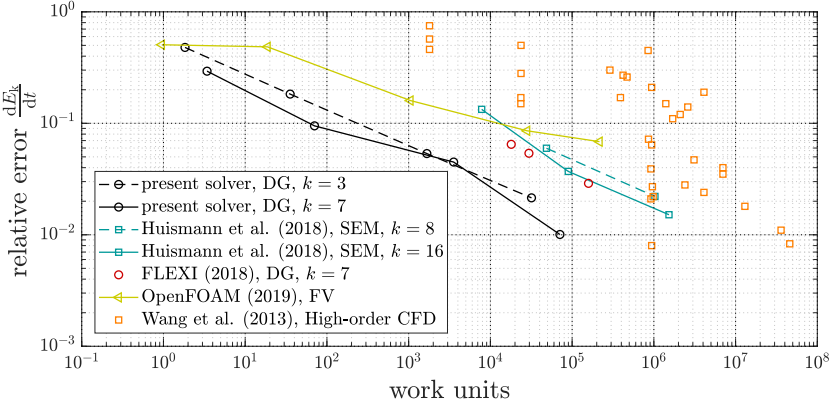
- Adaptively refined mesh in 3D for Fishera corner `hyper_L`
- Continuous $\mathcal{Q}_2$ elements
- Experiment: Timo Heister, Conrad Clevenger, step-50 tutorial program
- Matrix-free (MF) solves 8 times as many unknowns in same time as matrix-based AMG and GMG solvers
- Matrix-based (MB) GMG solver slows down for many cores because Trilinos/Epetra mat-vec includes barrier on all levels
  - Involves cores that have dropped out on coarse levels $\rightarrow$ slowdown
  - Our own parallel vector in MF scales **much better**: only point-to-point

Bringing it all together – a CFD application

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



Opportunities with tuned implementation: We are **one order of magnitude faster in normalized run time** than all results from Wang et al. (2013)

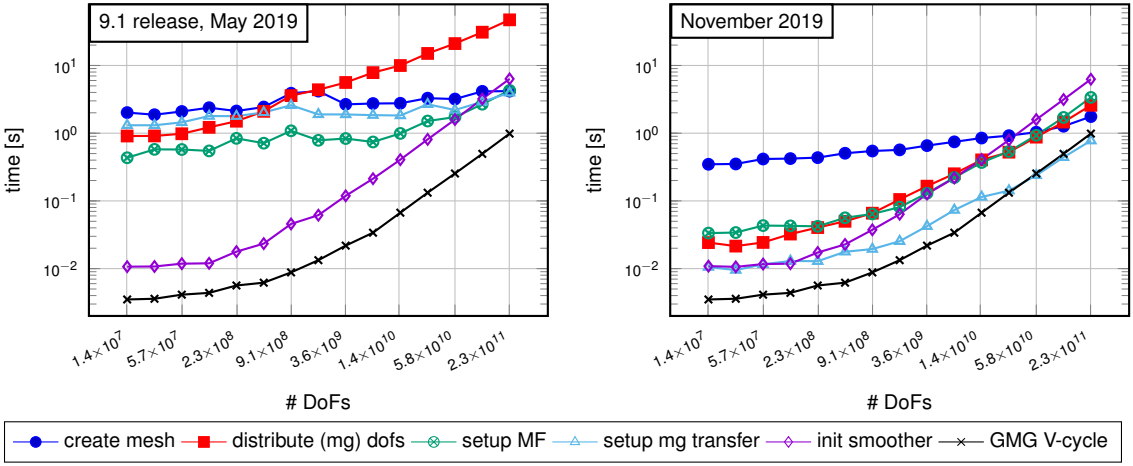


- Fehn, Wall, Kronbichler, Efficiency of high-performance discontinuous Galerkin spectral element methods for under-resolved turbulent incompressible flows, *Int J Numer Meth Fluids 88:32–54*, 2018 + recent updates
- Wang et al., High-order CFD methods: current status and perspective, *Int J Numer Meth Fluids 72:811–845*, 2013
- www.flexi-project.org
- Huismann, Stiller, Fröhlich, Scaling to the stars – a linearly scaling elliptic solver for *p*-multigrid, *J Comput Phys*, 2019

Recent focus: Improve scaling of initialization routines

Previous optimizations: Solver stage

At large scale, setup routines were expensive: example with $\mathscr{Q}_5$ elements on 1024 nodes / 49k MPI ranks



Summary and next steps

- Matrix-free algorithms from deal.II among leading high-order FEM algorithms, especially on Intel/AMD CPUs
- More work to be done for GPUs
  - Support for systems of equations, DG, no. integration points $\neq$ no. of shape functions
  - Collaboration with libCEED?
- Evaluate geometry on the fly to reduce memory access
- Work on support for Raviart–Thomas and Nédélec elements
- Bring fused vector operations for CG and Chebyshev into deal.II via specialized matrix-free operators
- Reduce cost of memory-heavy ghost exchange by MPI-3 shared memory concepts $\rightarrow$ plan for new vector class `LA::SharedMPI::Vector` (Peter Munch)
- Improve data locality on CPU codes by new cell-centric loops for DG (Peter Munch)
  - All face integrals around an element done together with element integral
- Support for ARM SVE in `VectorizedArray`: I'm in contact with Japanese partners to soon work on Fujitsu's A64FX processor