Efficient, large-scale simulations on complex geometries

using a hybrid multigrid solver and a fully distributed triangulation

Peter Munch[1,2] and Martin Kronbichler[2]

[1] Institute of Materials Research, Materials Mechanics, Helmholtz-Zentrum Geesthacht, Germany

[2] Institute for Computational Mechanics, Technical University of Munich, Germany

August 6, 2019

Peter Munch (@peterrum)

- Ph.D. candidate at TUM and employee at HZG
- Supervised by Martin Kronbichler
- Research interests:
  - efficient and hardware-aware algorithms (incl. cache analysis)
  - **matrix free**, shared memory, **large-scale simulations with MPI**
  - **iterative solvers (hybrid multigrid)**
  - applications: **fluid mechanics**, solid mechanics, material science, and plasma physics
- For this workshop:
  - discuss the newly developed fully distributed triangulation
  - provide help regarding parallelization, vectorization, matrix free, …

https://www.lnm.mw.tum.de/staff/peter-muench/

Helmholtz-Zentrum Geesthacht
Zentrum für Material- und Küstenforschung

Part 1:

Problem definition and motivation

Complex geometry

Definition (1)

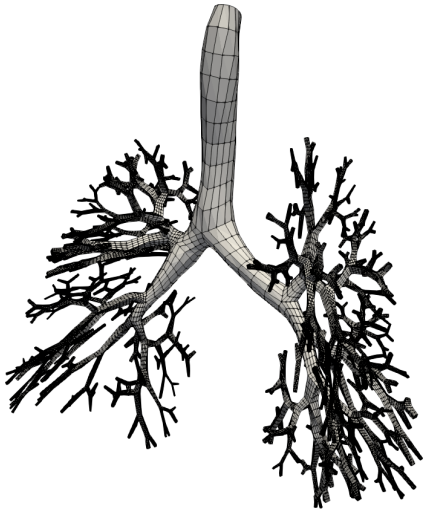
A geometry is complex if it can be meshed only with a significant amount of cells.

Definition (2)

A geometry is complex if it can be meshed only with a significant amount of coarse cells[1]*.*

[1] **Note:** Meshes generated by black-bock mesh-generation software might also have a large amount of coarse cells. The algorithms presented here can be applied also for those meshes.

Example

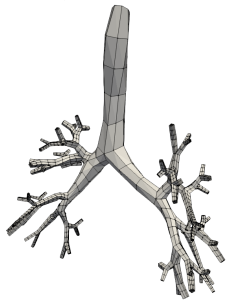
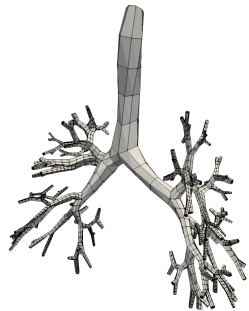
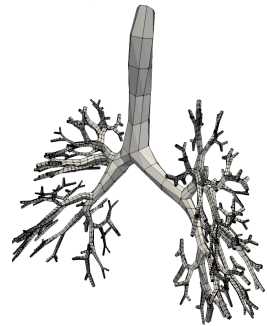


- mesh model of the first 11 airway generations of a patient-specific lung geometry
- 84,864 coarse cells + single global refinement

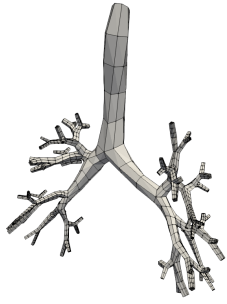
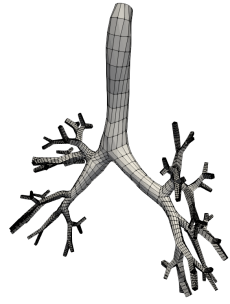
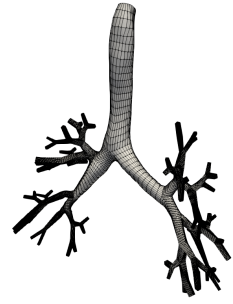
- goal: to understand the ventilation in the entire lung, i.e., oxygen transport and mixture between fresh and consumed air

C. J. Roth, K. M. Förster, A. Hilgendorff, B. Ertl-Wagner, W. A. Wall, and A. W. Flemmer, Gas exchange mechanisms in preterm infants on HFOV - a computational approach, Sci. Rep., vol. 8, pp. 13008, 2018.

Example (cont.)

	7 generations	8 generations	9 generations
			
coarse-grid cells:	4,236	9,396	16,032

Example (cont.)

7 generations with	(coarse grid)	1 refinement	2 refinements
			
cells:	4,236	33,888	271,104

using transfinite interpolation

Example (cont.)

Simulations with up to **16 lung generations with at least a million coarse cells** in total are needed to be able to retrieve biologically meaningful results.

Problem statement

Solving the Poisson equation[2]:

- with `FE_Q(k)` and `FE_DGQ(k)` with $k > 1$
- huge coarse grid ($\gg 10,000$ coarse-grid cells)
- small number of refinements (≤ 2 refinements)
- no adaptivity (no hanging nodes)

requires:

1. an efficient matrix-free hybrid multigrid solver and ▷ Part 2/3
2. a “fully distributed” triangulation. ▷ Part 4

[2] Solving the Poisson equation is a substep of the dual-splitting projection scheme for solving the Navier–Stokes equations and of the solution of the Vlasov-Poisson equation (plasma physics).

Part 2:

Matrix-free geometric multigrid methods (recap)[3]

[3] see also: Martin Kronbichler and Wolfgang A Wall. A Performance Comparison of Continuous and Discontinuous Galerkin Methods with Fast Multigrid Solvers. SIAM Journal on Scientific Computing, 40(5):A3423A3448, jan 2018. ISSN 1064-8275. doi: 10.1137/16M110455X.

Multigrid (V-cycle)

Solve the system of linear equations $\mathcal{A}(\boldsymbol{x}) = \boldsymbol{b}$:

- pre-smoothing:

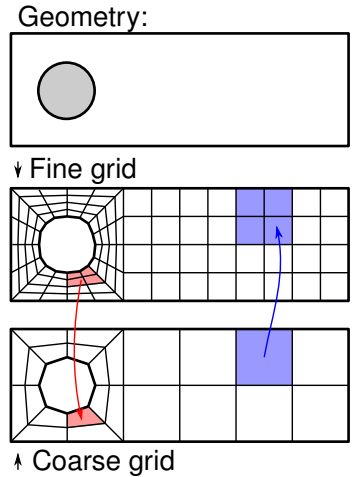
$$\boldsymbol{x} \leftarrow \mathcal{S}(\boldsymbol{x})$$

- recursive coarse-grid correction:

$$\mathcal{A}_c(\boldsymbol{v}) = \mathcal{R}(\boldsymbol{b} - \mathcal{A}(\boldsymbol{x})) \quad \text{and} \quad \boldsymbol{x} \leftarrow \boldsymbol{x} + \mathcal{P}(\boldsymbol{v})$$

- post-smoothing:

$$\boldsymbol{x} \leftarrow \mathcal{S}(\boldsymbol{x})$$



needed: smoother ($\mathcal{S}$), intergrid transfer operator ($\mathcal{I}$, $\mathcal{P}$, $\mathcal{R}$), coarse-grid solver

Matrix-free smoothers

Chebyshev smoother with point Jacobi inner preconditioner:

$$\boldsymbol{u}_{j+1} = \boldsymbol{u}_j + \sigma_j(\boldsymbol{u}_j - \boldsymbol{u}_{j-1}) + \theta_j \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{u}_j)$$

with the main building block:

Matrix-free matrix-vector product

$$\boldsymbol{v} = \boldsymbol{A}\boldsymbol{u} = \left(\sum_c \Pi_c^T \boldsymbol{A}_c \Pi_c\right)\boldsymbol{u} = \sum_c \Pi_c^T \boldsymbol{A}_c \Pi_c \boldsymbol{u} = \boxed{\sum_c \Pi_c^T \circ \mathcal{A}_c \circ \Pi_c(\boldsymbol{u}) = \boldsymbol{v}}$$

- reduction of data to be loaded (+mapping)
- reduction of work, using sum factorization within $\mathcal{S}$ in $\mathcal{A}_c = \mathcal{S}^T \circ \mathcal{Q}_c \circ \mathcal{S}$
- vectorization over elements (enabled by AoS↔SoA-functionality of Π_c)
- improved cache usage

Kronbichler, Kormann, A generic interface for parallel finite element operator application. *Comput. Fluids* 63:135–147, 2012
Kronbichler, Kormann, Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *arXiv:1711.03590*, 2017

Matrix-free smoothers (cont.)

Example: cell contribution of Laplace operator

Listing 1: $v = \sum_c \Pi_c^T \circ \mathcal{S}^T \circ \mathcal{Q}_c \circ \mathcal{S} \circ \Pi_c(u)$

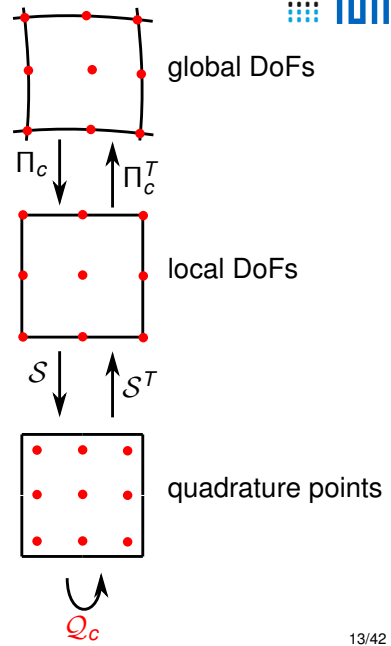
```
data.cell_loop(this, cell_loop, dst, src);
```

Listing 2: $\Pi_c^T \circ \mathcal{S}^T \circ \mathcal{Q}_c \circ \mathcal{S} \circ \Pi_c(u)$

```
void cell_loop(data, dst, src, range) const {
  FEEvalCell fe_eval(data);

  for(auto cell = range.first; cell < range.second; ++cell) {
    fe_eval.reinit(cell);
    fe_eval.gather_evaluate(src, false, true);
    this->do_cell_integral(fe_eval);
    fe_eval.integrate_scatter(false, true, dst);
  }
}
```

Listing 3: $\mathcal{Q}_c(u_q) \rightarrow$ “kernel”

```
void do_cell_integral(fe_eval) const {
  for(unsigned int q = 0; q < fe_eval.n_q_points; ++q)
    fe_eval.submit_gradient(fe_eval.get_gradient(q), q);
}
```



Matrix-free smoothers (cont.)

Example: cell contribution of Laplace operator

```
namespace MF::Util {
  class Kernel{
    Kernel(/*Flags*/){}
    virtual void do_cell_integral(fe_eval) const {};
    virtual void do_face_integral(fe_eval_m, fe_eval_p) const {}; //DG
    virtual void do_boundary_integral(fe_eval) const {};          //DG
  }

  template<typename Kernel>
  class LinearOperator {
    LinearOperator(Kernel kernel) : kernel(kernel) {}
    void vmult(dst, src);

    Kernel & kernel;
  }
}
```

```
class LaplaceKernel : public MF::Util::Kernel {
  LaplaceKernel : MF::Util::Kernel(/*flags*/){}

  void do_cell_integral(fe_eval) const {
    for(unsigned int q = 0; q < fe_eval.n_q_points; ++q)
      fe_eval.submit_gradient(fe_eval.get_gradient(q), q);
  }
}
```

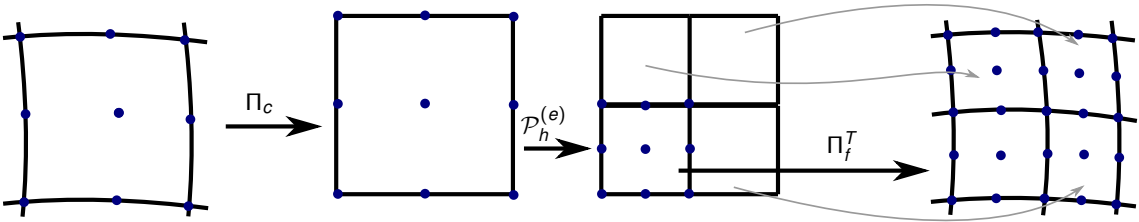
New infrastructure → PR????:

- DG/CG
- multigrid levels
- ECL/FCL
- scalar/vectorial
- constraints

Matrix-free intergrid operators

Due to quad-/oct-tree refinement, the global prolongation/restriction operation can be reformulated as a local prolongation/restriction operation of each coarse cell c onto its 4/8 children f [4]:

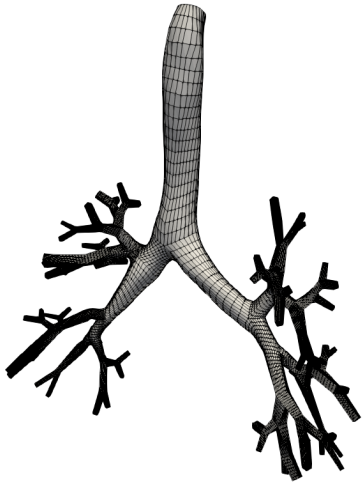
$$\boxed{\boldsymbol{x}_f = \mathcal{P}_h(\boldsymbol{x}_c) = \sum_{(c,f)} \Pi_f^T \circ \mathcal{P}_h^{(e)} \circ \Pi_c(\boldsymbol{x}_c)} \quad \text{and} \quad \boldsymbol{x}_c = \mathcal{R}_h(\boldsymbol{x}_f) = \mathcal{P}_h^T(\boldsymbol{x}_f)$$



▷ matrix-free via sum factorization, for Q and DG_Q!

[4] Note: in the case of Q, degrees of freedom shared by multiple cells have to be weighted by Π.

Back to the example: 7 generations with `FE_DGQ(5)`:



Number of DoFs on coarse grid:

$$\approx 4,236 \times (5+1)^3 = 914,976$$

This coarse problem is not suited for:

- plain conjugate gradient methods (CG)
- algebraic multigrid (AMG)

Remedy: hybrid multigrid solver!

Definition (3)

A geometry is complex if the computation time is dominated by solving the coarse-grid problem (i.e. $t_c \gg t_f$) when only h-coarsening is exploited.

Part 3:

Hybrid multigrid solver[5]

[5] developed in my Master's thesis "An efficient hybrid multigrid solver for high-order discontinuous Galerkin methods" 2018, supervised by Martin Kronbichler and Niklas Fehn (online available: https://mediatum.ub.tum.de/node?id=1514962)

Definition

A hybrid multigrid solver exploits the structure of the problem as much as possible by both geometrical and polynomial coarsening to decrease the size of the coarse-grid problem. The resulting minimal coarse-grid problem can be solved by AMG in a black-box fashion.

Observation

In the case of discontinuous Galerkin methods (DG), the problem size can be decreased even more by switching to continuous basis.

Multigrid levels

Each multigrid level can be identified by the following triple:

$$(\texttt{level}, \texttt{degree}, \texttt{FE}) \in \mathbb{N}_0 \times \mathbb{N} \times \{\texttt{Q}, \texttt{DGQ}\}$$

with following transition functions:

$$\begin{aligned}
(\texttt{level}, \texttt{degree}, \texttt{FE}) &\xrightarrow{h} (\texttt{level-1}, \texttt{degree}, \texttt{FE}) \\
(\texttt{level}, \texttt{degree}, \texttt{FE}) &\xrightarrow{p} (\texttt{level}, \textbf{next}(\texttt{degree}), \texttt{FE}) \\
(\texttt{level}, \texttt{degree}, \texttt{DGQ}) &\xrightarrow{c} (\texttt{level}, \texttt{degree}, \texttt{Q}) \\
(0, 1, \texttt{Q}) &\to \bot
\end{aligned}$$

Multigrid levels (cont.)

Each multigrid level can be identified by the following triple:

$$(\texttt{level}, \texttt{degree}, \texttt{FE}) \in \mathbb{N}_0 \times \mathbb{N} \times \{\texttt{Q}, \texttt{DGQ}\}$$

with the **p-coarsening strategies next()**:

$$\textcircled{1}\ k_{i-1} = 1 \qquad \textcircled{2}\ k_{i-1} = \max(1, \lfloor k_i/2 \rfloor) \qquad \textcircled{3}\ k_{i-1} = \max(1, k_i - 1)$$

Observation: bisection (strategy ②) is the most efficient; it is cheap and its iteration numbers are only weakly dependent on degree.

Multigrid levels (cont.)

Each multigrid level can be identified by the following triple:

$$(\texttt{level}, \texttt{degree}, \texttt{FE}) \in \mathbb{N}_0 \times \mathbb{N} \times \{\texttt{Q}, \texttt{DGQ}\}$$

Example:
GMG:

$$\texttt{(2,5,DGQ)} \xrightarrow{h} \texttt{(1,5,DGQ)} \xrightarrow{h} \texttt{(0,5,DGQ)}$$

HMG:

$$\texttt{(2,5,DGQ)} \xrightarrow{c} \texttt{(2,5,Q)} \xrightarrow{p} \texttt{(2,2,Q)} \xrightarrow{p} \texttt{(2,1,Q)} \xrightarrow{h} \texttt{(1,1,Q)} \xrightarrow{h} \texttt{(0,1,Q)}$$

with:

+ 3 additional explicit multigrid levels
+ reduction of the size of the coarse problem by $\approx 6^3 = 216$ for 3D
+ linear continuous elements suited very well for AMG (e.g. ML and MueLu).

Implementation details: transfer

Based on the observation that[6]:

$$(\texttt{level},\ \underbrace{\texttt{degree, FE}}_{\text{def. of DoFHandler}}) \in \mathbb{N}_0 \times \mathbb{N} \times \{\texttt{Q}, \texttt{DGQ}\}$$

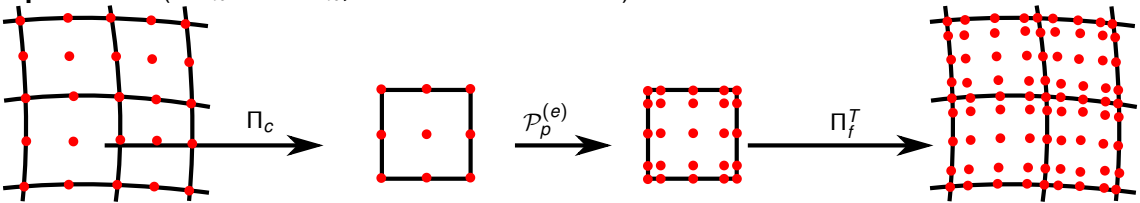
the following `deal.II`-specific implementation can be derived:

- construct a `DoFHandler` for each (`degree`, `FE`)-pair and
- define new multigrid transfer operators $\mathcal{P}_\square$, $\mathcal{R}_\square$, and $\mathcal{I}_\square$ with $\square \in \{h, p, c\}$ working on `Vector`s living on different `DoFHandler`s.

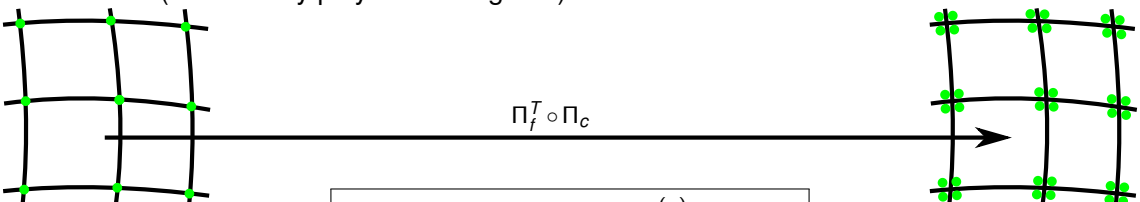
[6] `DoFHanlder<dim> dof_handler(tria); dof_handler.distribute(FE_Q(degree));`

Implementation details: transfer (cont.)

▷ **p-transfer** (for Q and DGQ; with sum factorization):



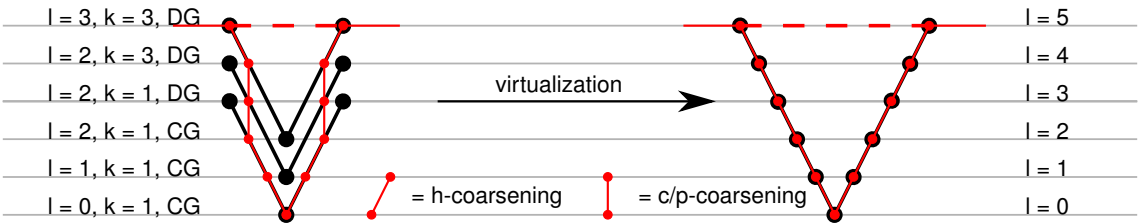
▷ **c-transfer** (for arbitrary polynomial degrees):



$$\boldsymbol{x}_f = \mathcal{P}_\square(\boldsymbol{x}_c) = \sum_{(c,f)} \Pi_f^T \circ \mathcal{P}_\square^{(e)} \circ \Pi_c(\boldsymbol{x}_c)$$

Implementation details: transfer (cont.)

Virtualization:



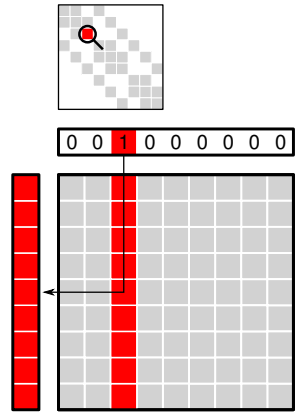
Implementation details: construction of the coarse-grid system matrix

Requirement: AMG needs a matrix representation of the operator.

Is there a way to create a matrix
with a matrix-free framework?

The i-th column of the local element matrix can be determined via matrix-free operator evaluations:

$$\text{column}_i(A_K) = \mathcal{A}_K(e_i)$$



A sequence of calculation of single columns leads to the full (element) matrix.

Remarks:

- matrix-free complexity: $d \cdot (k+1)^{2 \cdot d+1}$ vs. naïve: $d \cdot (k+1)^{3 \cdot d}$
- perfect vectorization

Implementation details: construction of system matrix

```
namespace MF::Util {
  // Kernel and LinearOperator ...

  template<typename Kernel> class SparseMatrixOperator {
    void init_system_matrix(system_matrix) const;
    void calculate_system_matrix(system_matrix) const;
  }

  template<typename Kernel> class DiagonalOperator {
    void calculate_diagonal(diagonal) const;
  }

  template<typename Kernel> class BlockDiagonalOperator {
    void calculate_block_diagonal_matrices(do_invert) const;
    void apply_block_diagonal(dst, src) const;
  }

  template<typename Kernel> class NonLinearOperator {
    void set_solution_linearization(vector) const;
    void evaluate_residual(dst, src) const;
  }

  // ... and many more
}
```

New infrastructure → PR????:

- sparse matrix (!)
- diagonal
- block diagonal
- (matrix-free) block diagonal
- non-linear operator
- tested against each other

Conclusions

Matrix-free hybrid multigrid methods are efficient for large coarse grids ***if*** *the size of the coarse problem can be reduced to a minimum, i.e., enough* ***explicit multigrid levels*** *can be constructed via geometrical and polynomial coarsening.*

This conclusion is in conflict with the observation that in the case of complex geometries only moderate number of refinements (≤ 2) and moderate polynomial degrees (≤ 5) are feasible.

Future work:

- create a pull request (containing the hybrid multigrid algorithm and the extended basis operator)
- extend the hybrid multigrid algorithm to h-adaptivity (with my master student) and hp-adaptivity (with @marcfehling)

Part 4:

Fully distributed triangulation

Observation

For *complex* geometries, `parallel::distributed::Triangulation` is not optimal:

- storing the coarse grid by every process is too memory consuming
- non-optimal partitioning: discontinuous partitions

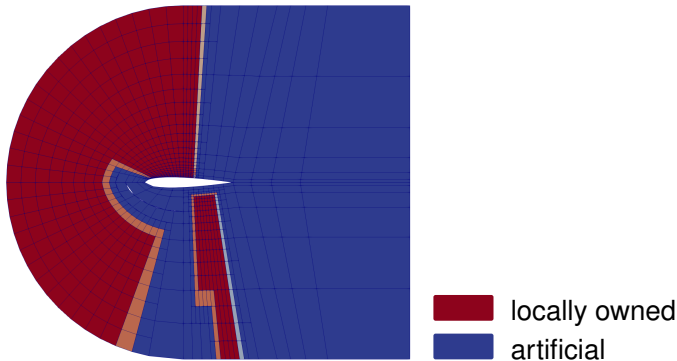


Figure: NACA 0012 airfoil (with 156 coarse cells, `P:D:T`, by Elias Dejene)

Observation (cont.)

For *complex* geometries, `parallel::distributed::Triangulation` is not optimal:

- storing the coarse grid by every process is too memory consuming
- non-optimal partitioning: discontinuous partitions

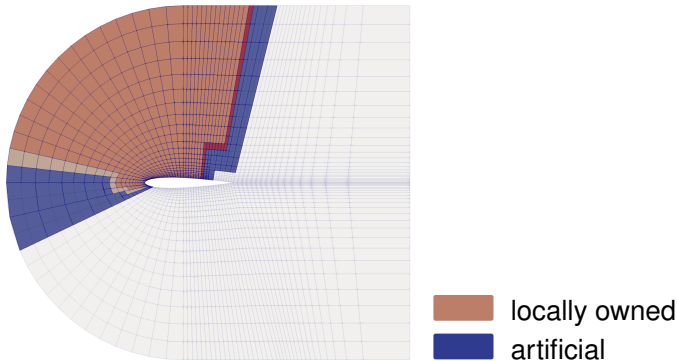


Figure: NACA 0012 airfoil (with 156 coarse cells, `P:F:T`, by Elias Dejene)

Problem definition

PR #8418 proposes the new `parallel::fullydistributed::Triangulation`, which only stores the **locally relevant cells** and allows more flexibility regarding **partitioning** (e.g. METIS, ParMETIS, Zoltan, p4est, etc.).

Definition

Locally relevant cells are cells:

- that are locally owned by the current process or are ghost cells on a level, or
- one of their children is locally relevant.

Problem definition (cont.)

Further requirements:

- ☑ extract information from `dealii::Triangulation` and from `P:D:T`
- ☑ static mesh
- ☑ hanging nodes
- ☑ geometric multigrid
- ☑ periodicity (partly implemented)
- ☑ 2D/3D and **1D (also in parallel!)**
- ☑ I/O

- ☒ adaptive mesh refinement (AMR)

Construction data

The creation of `P:F:T` is centered around `P:F:ConstructionData`.

Definition

The struct `P:F:ConstructionData` contains:

- *a locally relevant coarse-grid triangulation (local cells and one layer of ghost cells)*
  - *vertices*
  - *cells with material and manifold IDs*
  - *boundary IDs*
- *a mapping of the locally relevant coarse grid into the global coarse-grid triangulation*
- *information about cell refinement and cell owner (subdomain ID)*

Construction data (cont.)

The creation of `P:F:T` is centered around `P:F:ConstructionData`.

Two-step process:

- create `P:F:ConstructionData`
- create `P:F:Triangulation` using it

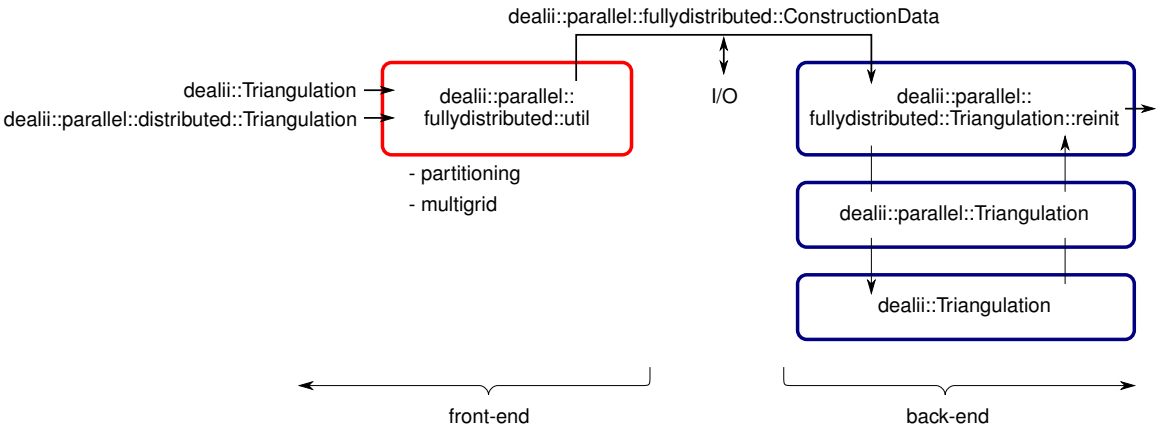
```
#include <deal.II/distributed/tria.h>

using namespace parallel::fullydistributed;

// create a construction data
ConsturcionData<dim> data;
// ... and fill it (TODO: by user)

// create triangulation
Triangulation<dim> tria(comm);
tria.reinit(data); // by using the construction data
```

Creating `P:F:ConstructionData` with `P:F:Utilities`:

```
ConstructionData<dim, spacedim> copy_from_triangulation(const Triangulation<dim, spacedim> &tria);
ConstructionData<dim, spacedim> create_and_partition(std::function<void(Triangulation<dim, spacedim> &)> func);
ConstructionData<dim, spacedim> deserialize(std::string file_name);
```

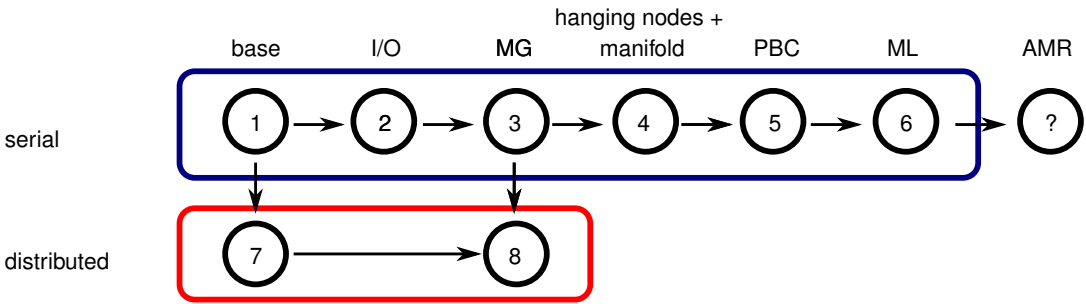
Construction data (cont.)

The creation of `P:F:T` is centered around `P:F:ConstructionData`.



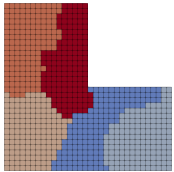
Tutorials

You can find a set of tutorials on `https://github.com/peterrum/dealii-pft`:



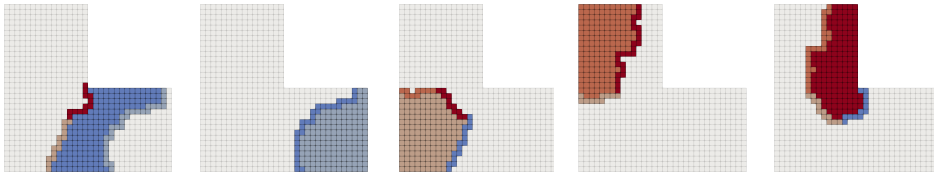
Step 1: serial triangulation

▷ closes PR#3956 ping @tjhei



colored serial mesh

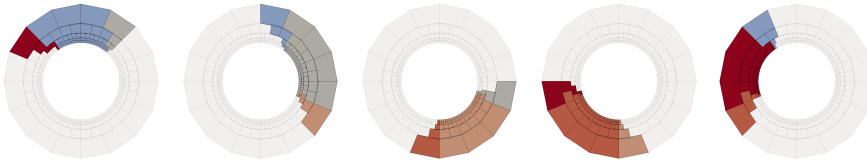
partitioned mesh



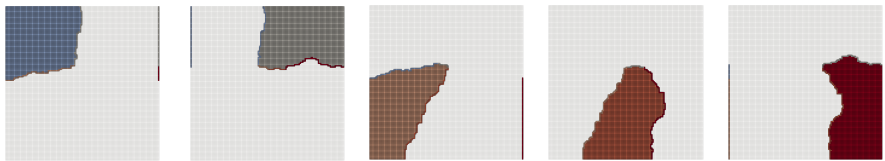
Step 2: serialization/deserialization → boost

Step 3/4: multigrid, hanging nodes, manifolds

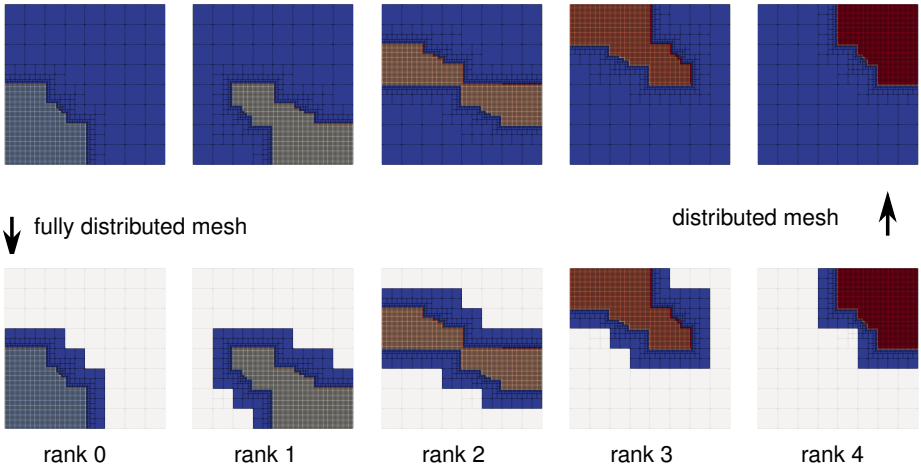
▷ see also deal.II step-1



Step 5: periodic faces



Step 7/8: `P:D:T`


fully distributed mesh
distributed mesh
rank 0
rank 1
rank 2
rank 3
rank 4

Step 6: Multi-level partitioning: master-slave

1. create a serial triangulation by a subset of master processes (e.g. one per node):

```
auto construction_data =
  parallel::fullydistributed::Utilities::create_and_partition<dim, spacedim>(
    [&](dealii::Triangulation<dim, spacedim> & tria) mutable {
      GridGenerator::subdivided_hyper_cube(tria, n_subdivisions);
      tria.refine_global(n_refinements);
    }, tria_pft, additional_data);
```

2. partition the mesh using e.g. METIS, taking node-locality into account:

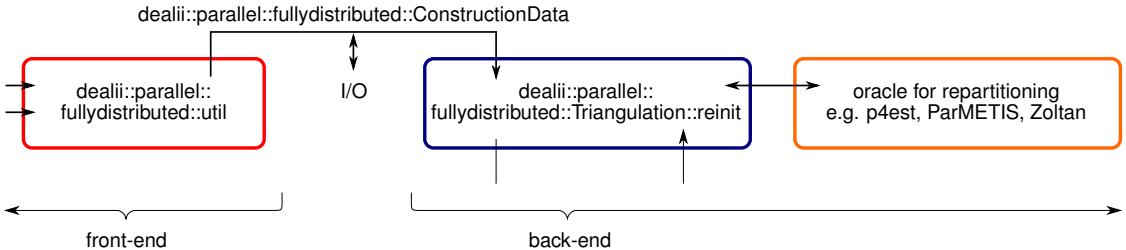
```
void shared_partition_triangulation(dealii::Triangulation<dim> &tria,
                                    AdditionalData              additional_data);
```

3. distribute the `P:F:T` mesh information

Conclusions

Future work:

- **different front-ends**: e.g. `create_and_partition` also for `P:D:T`
- **adaptive mesh AMR:** ▷ using `ConsensusAlgorithm` (PR #8300)



- any features missing?