

Handling non-matching methods between independent distributed grids: step-70

Luca Heltai¹, Bruno Blais²

¹International School for Advanced Studies

²Polytechnique Montréal



**POLYTECHNIQUE
MONTRÉAL**

TECHNOLOGICAL
UNIVERSITY



26 May 2020 – deal.II Online Workshops

Outline

1 Introduction

- Motivations
- Everything reduces to point evaluation
- We need a way to track points: particles

2 Fluid structure interaction

- Matching solids points into fluid cells
- Insert global particles
- Keeping track of ownership (LAC)

3 Putting it all together: Step-70

- Nitsche method
- A classical experiment
- Results: impellers in 2 and 3 dimensions

Plan

1 Introduction

- Motivations
- Everything reduces to point evaluation
- We need a way to track points: particles

2 Fluid structure interaction

- Matching solids points into fluid cells
- Insert global particles
- Keeping track of ownership (LAC)

3 Putting it all together: Step-70

- Nitsche method
- A classical experiment
- Results: impellers in 2 and 3 dimensions

Mixing

- ✓ Critical industrial application
- ✗ Often a severe bottleneck that leads to waste
- ✗ Costly and difficult to investigate experimentally
- ✗ Not trivial so simulate



Simulation of mixing

Rotation of the geometry prevents the use of ALE techniques. Requires the use of specialized techniques:

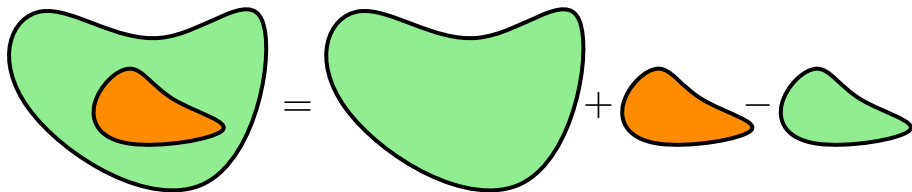
- ✗ Change of reference frame: **Severe limitations on the geometry**
- ✗ Sliding mesh: **Requires coupling the boundaries of two grids**
- ✓ Non-matching methods: immersed boundaries, fictitious domains, etc.

Non-matching methods can also be used for a much broader range of applications:

- ✓ Fluid-Structure interaction
- ✓ Overlapping geometries (e.g. Planetary mixers)



Graphical summary of non-matching methods



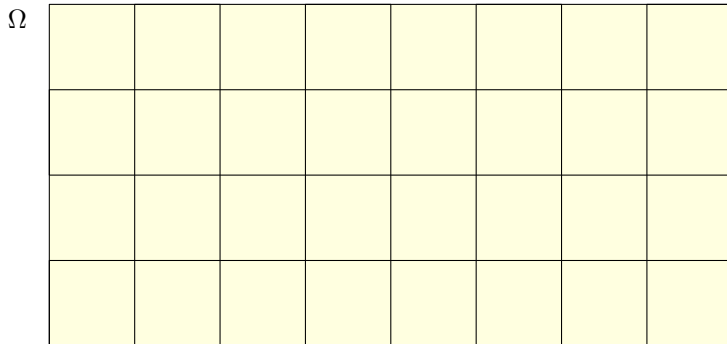
- ✓ Different properties on different domains
- ✓ Independent discretizations (FEM, BEM, FV, FD, etc.)
- ✓ Different physics on different domains (i.e., fluids and solids)
- ✓ Possibly different intrinsic scales
- ✗ We need a numerically efficient way to transfer information between the domains

We need a way to track points: particles

Particles are an ideal tool to keep track of the motion of our embedded domain:

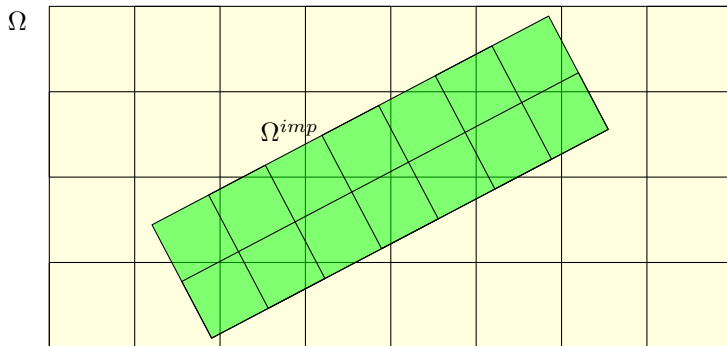
- ✓ They can readily move from processor to processor and can be exchanged using some-to-some communications.
- ✓ They already contain a very efficient mechanism to detect in which cell they lie and what is their reference location.
- ✓ An arbitrary amount of properties can be attached to them.

Quadrature (or support) points to particles



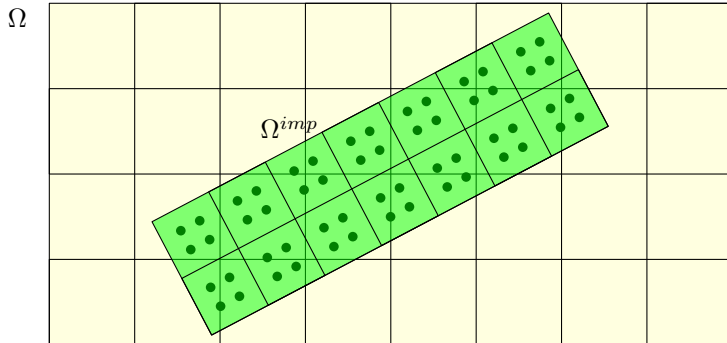
```
1  GridGenerator::generate_from_name_and_arguments(  
2    fluid_tria, par.name_of_fluid_grid,  
    par.arguments_for_fluid_grid);
```


Quadrature (or support) points to particles



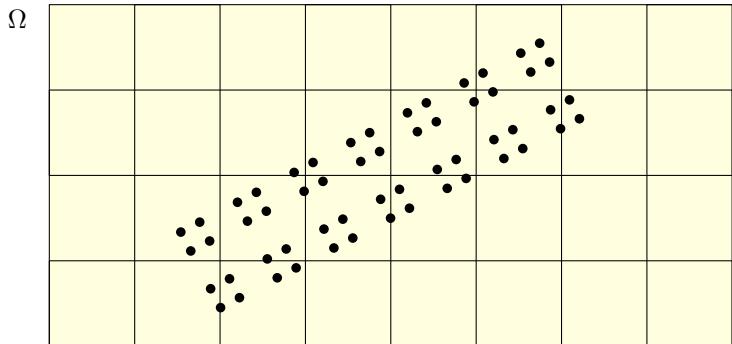
```
1 GridGenerator::generate_from_name_and_arguments(  
2   solid_tria, par.par_name_of_solid_grid,  
   par.arguments_for_solid_grid);
```

Quadrature (or support) points to particles



```
1  QGauss<dim> quadrature(fluid_fe->degree + 1);  
2  for (const auto &cell :  
      solid_dh.active_cell_iterators())  
3      if (cell->is_locally_owned())  
4          for (unsigned int q = 0; q < points.size(); ++q)  
5              quadrature_points_vec.emplace_back(points[q]);
```

Quadrature (or support) points to particles



```
1 solid_particle_handler.  
2   insert_global_particles(quadrature_points_vec,  
3                           global_fluid_bounding_boxes,  
4                           properties);
```

Plan

1 Introduction

- Motivations
- Everything reduces to point evaluation
- We need a way to track points: particles

2 Fluid structure interaction

- Matching solids points into fluid cells
- Insert global particles
- Keeping track of ownership (LAC)

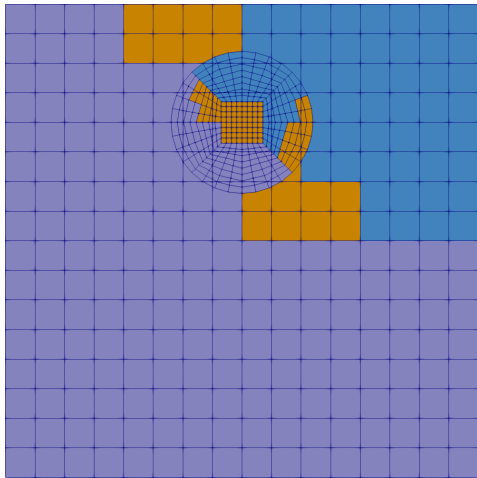
3 Putting it all together: Step-70

- Nitsche method
- A classical experiment
- Results: impellers in 2 and 3 dimensions

Insert global particles in distributed environments

Generating the particles of the embedded domain is quite challenging in parallel distributed environments:

- ✗ The two domains do not match
- ✗ The points of the embedded domain can lie in a cell which belongs to another processor



Requires a global insertion mechanism to create particles from an :

```
std::vector<Points<dim>>
```

where each processor owns such a (possibly non-empty) vector.

Insert global particles

```
1 particle_handler.  
2   insert_global_particles(points_vector,  
3                           global_fluid_bounding_boxes,  
4                           particle_properties);
```

- ✓ The vector containing the location of the particles created by the processor. These particles do not need to be located in a cell in that processor.

```
1 std::vector<Points<dim> > points_vector
```

- ✓ An (n_{proc}) size vector containing a vector of bounding boxes

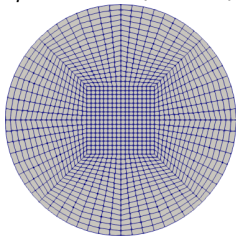
```
1 std::vector<std::vector<BoundingBox<spacedim>>>  
   global_bounding_boxes
```

- ✓ A vector of vectors of particle properties

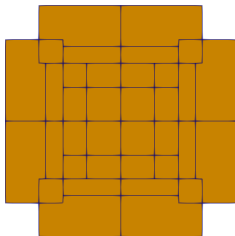
```
1 std::vector<std::vector<double> >  
   particle_properties
```

boost::rtree of Bounding boxes surrounding cells

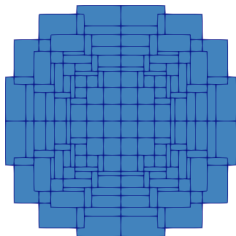
A `boost::rtree` of bounding boxes allows the identification of the cells in which the particles lie in $\mathcal{O}(n_{particles} \log(m_{boxes}))$ time.



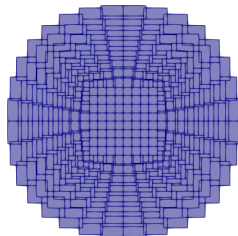
1 Level



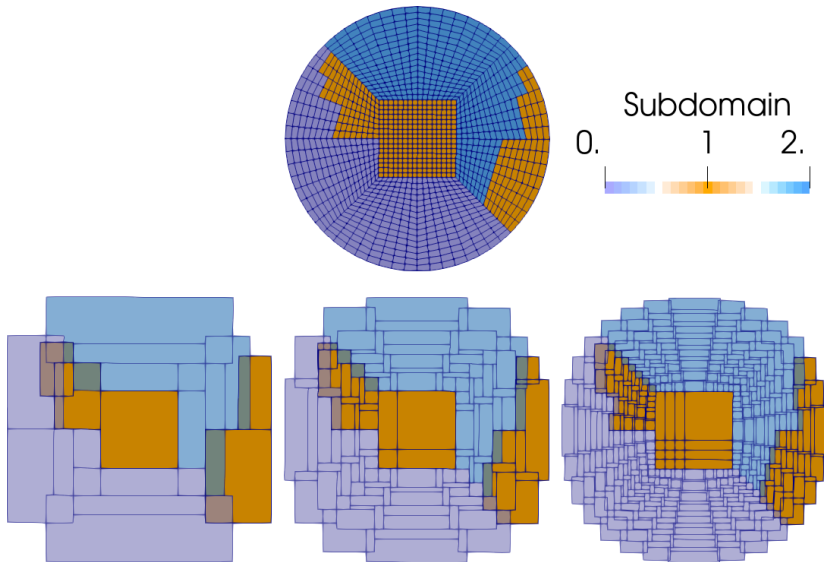
2 Levels



3 Levels



boost::rtree of Bounding boxes in parallel



Bounding boxes

Generating the bounding boxes is relatively easy.

```
1 std::vector<BoundingBox<spacedim>> all_boxes;
2 all_boxes.reserve(fluid_tria.n_locally_owned_active_cells());
3 for (const auto cell : fluid_tria.active_cell_iterators())
4     if (cell->is_locally_owned())
5         all_boxes.emplace_back(cell->bounding_box());
6
7 const auto tree = pack_rtree(all_boxes);
8 const auto local_boxes =
9     extract_rtree_level(tree,
10         par.fluid_rtree_extraction_level);
11
12 global_fluid_bounding_boxes =
13     Utilities::MPI::all_gather(mpi_communicator,
14         local_boxes);
```

and bounding boxes can now even be written and visualized!

```
1 BoundingBoxDataOut<dim> data_out;
2 data_out.build_patches(my_bounding_box);
```

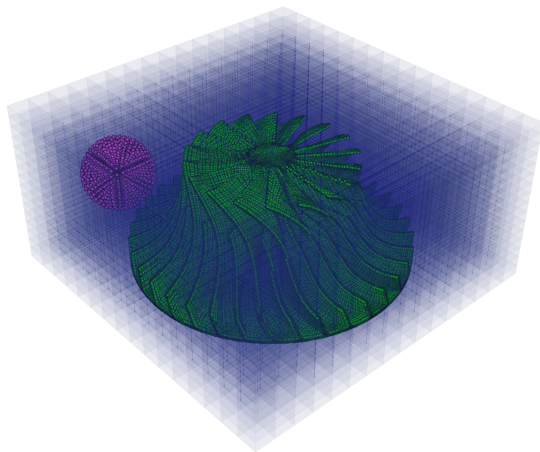
Insert global particles

Finally, the key steps in the global particle insertion:

- Gather the number of particles to be added for each processors (all_gather)
- Identify on which cell and on which processor the particle are located (distributed_compute_point_location)
- Gather the properties of the particles from the original owner of the points
- Generate the particles with their properties and unique id

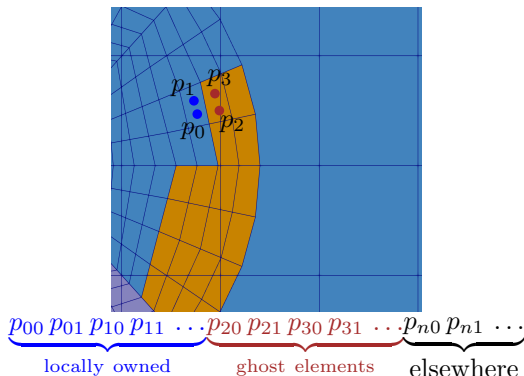
Outcome: Flexible particle insertion mechanisms

- ✓ Insertion at DOF support points
- ✓ Insertion at the quadrature points of a triangulation



As long as you have a triangulation, you can now use it to insert particles.

Keeping track of ownership: Linear algebra coupling



- We keep around two IndexSet objects, to identify locally owned particles coordinates (blue dots) ...
- ... and locally relevant particle coordinates (brown dots), i.e., coordinates of particles that fall within the current fluid domain, owned by someone else

Initializing ownership

Either based on where particles end up in the first step:

```
1  locally_owned_tracer_particle_coordinates =  
2      tracer_particle_handler.locally_relevant_ids().tensor_product  
3      complete_index_set(spacedim));
```

...or based on solid ownership (as in the previous image). Then at each step, keep track of locally relevant ones in the same way:

```
1  locally_relevant_tracer_particle_coordinates =  
2      tracer_particle_handler.locally_relevant_ids().tensor_product  
3      complete_index_set(spacedim));
```

And initialize a parallel MPI vector with it...

```
1  relevant_tracer_particle_displacements.reinit(  
2      locally_owned_tracer_particle_coordinates,  
3      locally_relevant_tracer_particle_coordinates,  
4      mpi_communicator);
```

...that can be used as a base for the solid equations.

Updating velocities

Interpolating from fields to particles:

```
1  Particles::Utilities::interpolate_field_on_particles(  
2      fluid_dh,  
3      tracer_particle_handler,  
4      locally_relevant_solution,  
5      tracer_particle_velocities,  
6      fluid_fe->component_mask(FEValuesExtractors::Vector  
7  
8  tracer_particle_velocities *= time_step;
```

...update the relevant IndexSet, and copy locally (read only!) the velocities of all the particles that fall within the locally owned fluid domain (MPI communication happens in the background):

```
1  relevant_tracer_particle_displacements =  
    tracer_particle_velocities;
```

...and transfer information back to the particles:

```
1  tracer_particle_handler.set_particle_positions(  
2      relevant_tracer_particle_displacements);
```

Plan

1 Introduction

- Motivations
- Everything reduces to point evaluation
- We need a way to track points: particles

2 Fluid structure interaction

- Matching solids points into fluid cells
- Insert global particles
- Keeping track of ownership (LAC)

3 Putting it all together: Step-70

- Nitsche method
- A classical experiment
- Results: impellers in 2 and 3 dimensions

Physics solved

In step-70, we solve the Stokes equations for a creeping flow (i.e. a flow where $\text{Re} \rightarrow 0$) and a no-slip boundary condition is applied on a moving *embedded domain* Γ associated with an impeller.

This leads to the following differential problem: given a sufficiently regular function \mathbf{g} on Γ , find the solution (\mathbf{u}, p) to

$$-\Delta \mathbf{u} + \nabla p = 0, \quad (1)$$

$$-\nabla \cdot \mathbf{u} = 0, \quad (2)$$

$$\mathbf{u} = \mathbf{g} \text{ in } \Gamma, \quad (3)$$

$$\mathbf{u} = 0 \text{ on } \partial\Omega. \quad (4)$$

Imposing the motion of the impeller

Two scenarios occur when enforcing conditions on the embedded domain:

- The geometrical dimension 'dim' of the embedded domain Γ is the same of the domain Ω ('spacedim'), In this case, the imposition of the Dirichlet boundary condition on Γ is done through a volumetric penalization.
- The embedded domain Γ has an intrinsic dimension 'dim' which is smaller than that of Ω ('spacedim'), thus its spacedim-dimensional measure is zero; for example it is a curve embedded in a two dimensional domain, or a surface embedded in a three-dimensional domain. In this case, the boundary condition is imposed weakly on Γ by applying the Nitsche method.

For both cases, a possible resulting formulation is:

$$(\nabla \mathbf{v}, \nabla \mathbf{u})_{\Omega} - (\operatorname{div} \mathbf{v}, p)_{\Omega} - (q, \operatorname{div} \mathbf{u})_{\Omega} + \beta(\mathbf{v}, \mathbf{u})_{\Gamma} = \beta(\mathbf{v}, \mathbf{g})_{\Gamma}.$$

Assembly of the Nitsche terms – Part I

Looping over all particles, cell-wise, and getting information from particles:

```
1  auto particle = solid_particle_handler.begin();
2  while (particle != solid_particle_handler.end())
3  {
4      const auto &cell =
5          particle->get_surrounding_cell(fluid_tria);
6      const auto pic =
7          solid_particle_handler.particles_in_cell(cell);
8      Assert(pic.begin() == particle,
9             ExcInternalError());
10     for (const auto &p : pic)
11     {
12         const auto &ref_q =
13             p.get_reference_location();
14         const auto &real_q = p.get_location();
15         const auto &JxW    = p.get_properties()[0];
16         ...
17     }
18 }
```

Assembly of the Nitsche terms – Part II

Assemble the penalization matrix:

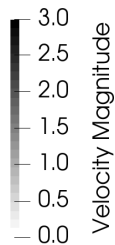
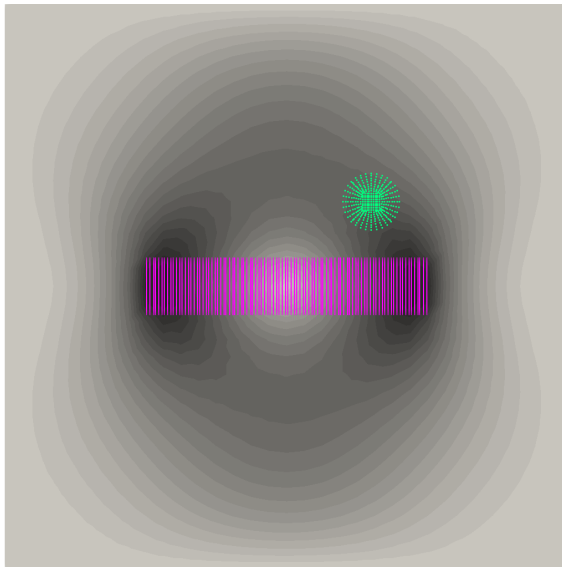
```
1  for (unsigned int i = 0; i < dofs_per_cell; ++i) {
2      const auto comp_i =
3          fluid_fe->system_to_component_index(i).first;
4      if (comp_i < spacedim) {
5          for (unsigned int j = 0; j < dofs_per_cell; ++j)
6              {
7                  const auto comp_j =
8                      fluid_fe->system_to_component_index(j).first;
9                  if (comp_i == comp_j)
10                     local_matrix(i, j) +=
11                         penalty_parameter * par.penalty_term *
12                         fluid_fe->shape_value(i, ref_q) *
13                         fluid_fe->shape_value(j, ref_q) * JxW;
14             }
15         local_rhs(i) += penalty_parameter * par.penalty_term *
16             solid_velocity.value(real_q, comp_i) *
17             fluid_fe->shape_value(i, ref_q) * JxW;
18     }
19 }
```

The problems: Mixing in the creeping flow regime

- ✓ Famous experiment by Sir G.I. Taylor using coloured dyes
- ✓ Multiple similar experiments on Youtube



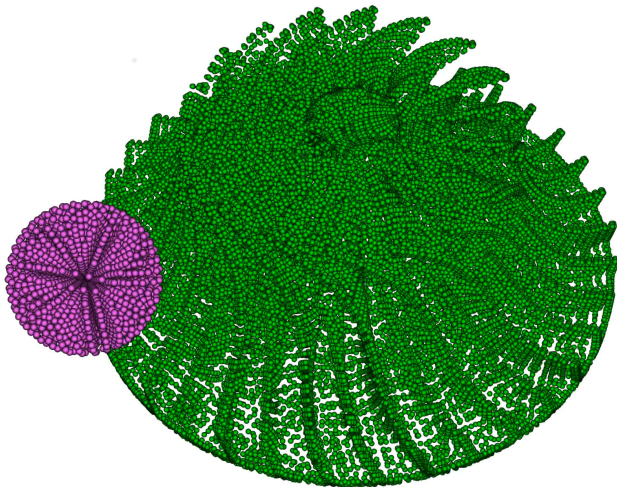
2D results



Impeller

Tracer Particles

3D results



Conclusions

Step-70 demonstrates a large variety of concepts

- ✓ Global insertion of particles
- ✓ Linear algebra coupling between particles and DOFS
- ✓ Non-matching methods
- ✓ Post-processing of particles using the new `Particles::DataOut`

It opens up a large range of possibilities for the solution of:

- ✓ Massively parallel immersed boundary problems
- ✓ Fluid structure interaction
- ✓ Chimera grids
- ✓ and many others...

It also was a very fun collaborative work!