THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Building a Bridge Between IBAMR and deal.II

David Wells

University of North Carolina

Cardiovascular Modeling and Simulation Lab

June 18, 2021

fiddle: implementing the IB method with deal.II + IBAMR

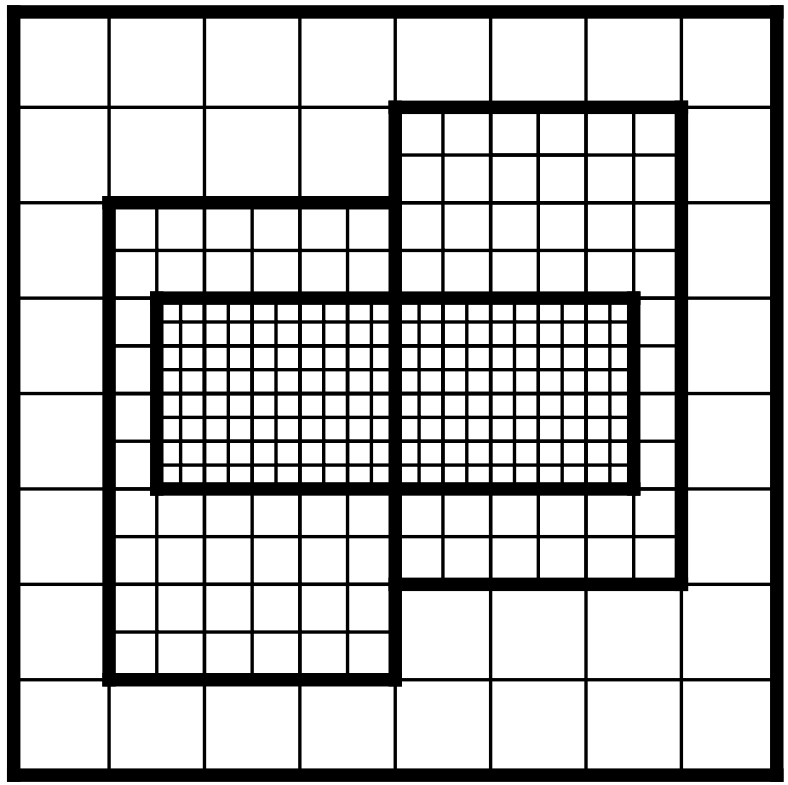
- *fiddle*: **f**our-chambered heart, **I**BAMR, **d**eal.II **l**ibrary: `https://github.com/drwells/fiddle`
- FSI with the immersed boundary (IB) method of Peskin
- Solid mechanics by Boffi et al
- One class (`fdl::IFEDMethod`) that hooks into IBAMR
- Modular components - use IBAMR's comprehensive support for IB kernels
- Primary difficulty - volumetric coupling in parallel means we need multiple data partitionings
- *This talk*: why deal.II works well for this problem

Collaborators

Discussions with many people have helped me figure out how to implement things in a useful way:

- Boyce Griffith
- Charles Puelz
- Jae Ho Lee
- Ben Vadala-Roth
- Marshall Davey
- Simone Rossi
- Margaret Anne Smith

SAMRAI: Patch-Based Finite Differences



- **S**tructured **A**daptive **M**esh **R**efinement **A**pplication **I**nfrastructure
- Finite differences on Cartesian grids: fast solvers
- Fine cells placed on structures and near important flow features
- Boundary conditions: simplified circuit-like models of the human body

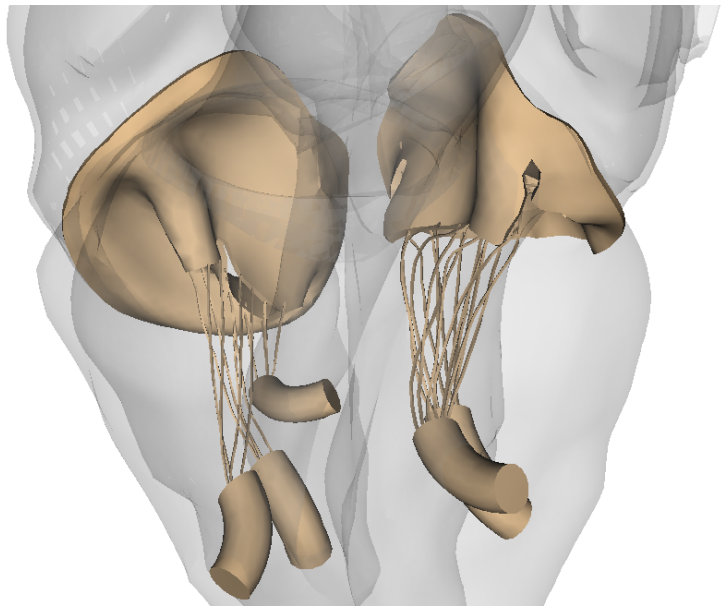
SAMRAI::find_active_cell_around_point()

```
template <class DoubleArray>
inline hier::Index<NDIM>
IndexUtilities::getCellIndex(
const DoubleArray& X, const double* const x_lower,
const double* const x_upper, const double* const dx,
const hier::Index<NDIM>& ilower, const hier::Index<NDIM>& iupper)
{
    hier::Index<NDIM> idx;
    for (unsigned int d = 0; d < NDIM; ++d)
    {
        double dX_lower = X[d] - x_lower[d];
        double dX_upper = X[d] - x_upper[d];
        if (std::abs(dX_lower) <= std::abs(dX_upper))
            idx(d) = ilower(d) + floor(dX_lower / dx[d]);
        else
            idx(d) = iupper(d) + floor(dX_upper / dx[d]) + 1;
    }
    return idx;
}
```

IBAMR: Immersed Boundary Adaptive Mesh Refinement

- `https://ibamr.github.io`
- Authors: Boyce Griffith (UNC), Amneet Bhalla (SDSU), me, a few others (Amneet has good students)
- Built on SAMRAI - fluids typically use MAC scheme
- Level set, Brinkman penalization, Stokes and Navier-Stokes, multiple IB formulations (CIB, IIM, rods, etc)

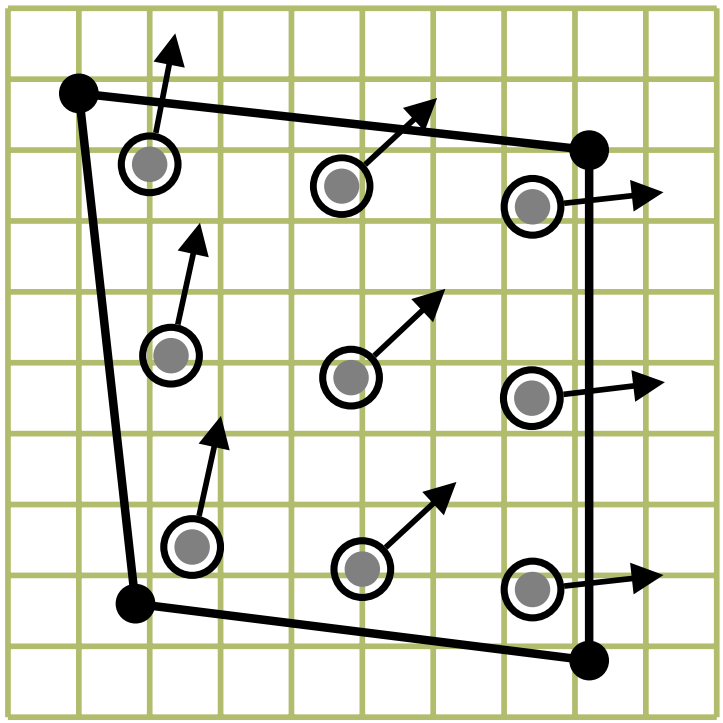
Incompressible Fluids, Incompressible Structures



- Navier-Stokes: finite differences on a Cartesian grid
- Finite strain elasticity: finite elements on an unstructured mesh
- Coupled with the immersed boundary (IB) method

The fluid and structure are assumed to have equal densities (neutral buoyancy). IFED means *immersed finite element-finite difference*.

The Immersed Finite Element-Finite Difference Method



- Solve for the fluid at all points (*including inside the structure*)
- Structure velocity given by fluid velocity
- Fluid force is given by the structure force density
- Coupling is done by regularized delta functions (δ_h)

Coupling Operators

In this talk I'll focus on the coupling operators, *force spreading*:

$$\mathbf{f}(\mathbf{x}, t) = \int_\Omega \mathbf{F}(\mathbf{X}, t)\delta(\mathbf{x} - \chi(\mathbf{X}, t))\,\mathrm{d}\boldsymbol{X},$$

$$f^1_{i,j+\frac{1}{2},k+\frac{1}{2}} = \sum_{(\boldsymbol{X}_q, w_q)\in\mathbb{Q}_q} F^1_q(t)\,\delta_h(\boldsymbol{x}_{i,j+\frac{1}{2},k+\frac{1}{2}} - \chi_h(\boldsymbol{X}_q, t))w_q,$$

and *velocity interpolation plus projection*:

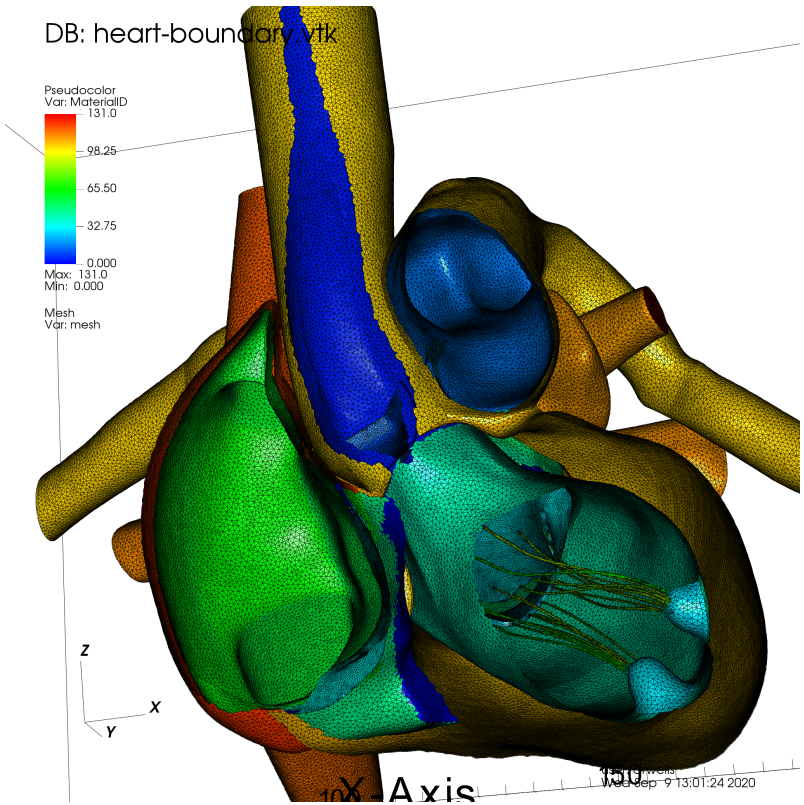
$$\mathbf{U}(\mathbf{X}, t) = \mathbf{u}(\chi(\mathbf{X}, t), t) = \int_\Omega \mathbf{u}(\mathbf{x}, t)\delta(\mathbf{x} - \chi(\mathbf{X}, t))\,\mathrm{d}\boldsymbol{x},$$

$$\mathbf{U}^{IB,1}(\boldsymbol{X}_q) = \sum_{i,j,k} u^1_{i,j+\frac{1}{2},k+\frac{1}{2}}\,\delta_h(\boldsymbol{x}_{i,j+\frac{1}{2},k+\frac{1}{2}} - \chi_h(\boldsymbol{X}_q, t))\Delta x^3$$

$$(\varphi, \mathbf{U}^1) = (\varphi, \mathbf{U}^{IB,1}), \forall\varphi \in X^h$$

Quadrature points must satisfy some point density condition - implemented by `fdl::QuadratureFamily`

What does deal.II do?



Necessary features:

- simplices (our geometries are hard, we need `TetWild`)
- matrix free solvers
- Scalable consensus algorithms like `some_to_some()`
- native tensor library for solid mechanics

What does deal.II do?

Very useful features:

- the one-to-many idiom: classes can set up a private `DoFHandler`
- `LA::d::Vector` is efficient, handles ghost data nicely
- `MappingFEField` (having separate mappings is great)
- support for `boost::rtree`
- tolerant of weird uses, abuses

Future work:

- Distributed rtree (perhaps arborx)
- AMR with tets
- More tet elements (higher-order `FE_Simplex_P_Bubbles`)

fiddle: four-chambered heart, IBAMR, deal.II library

Fundamentally we just need to do three things (in parallel):

- interpolate: SAMRAI to deal.II
- spread: deal.II to SAMRAI
- count quadrature points: deal.II to SAMRAI, for parallel partitioning and load balancing

Fiddle is essentially one class, `fdl::IFEDMethod`, that hooks into IBAMR and can do those three things (plus other things IBAMR requires).

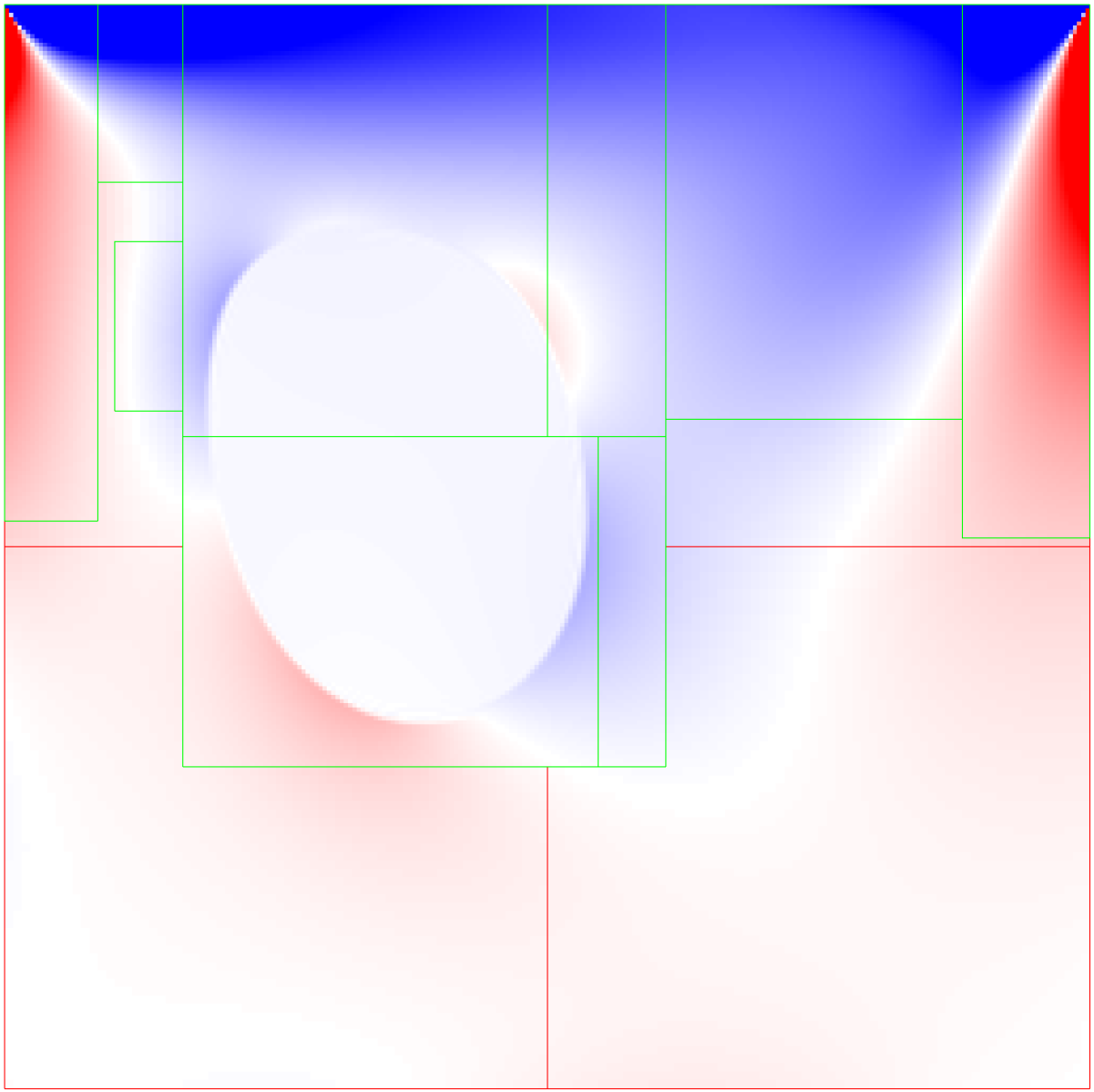
Parallel Partitioning with `fdl::OverlappingTriangulation`

To evaluate couplings, we must find all cells intersecting a SAMRAI patch

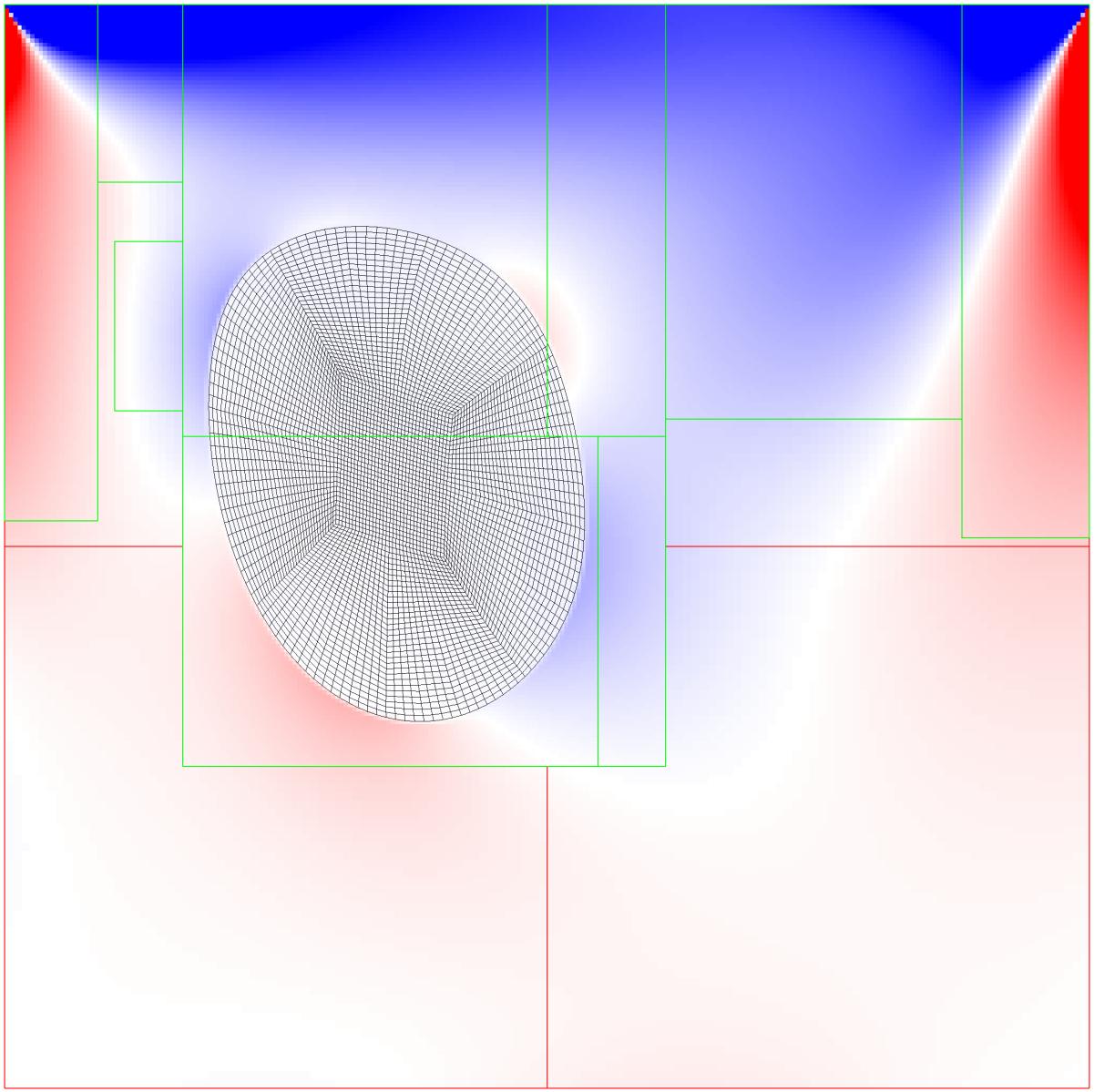
```
class OverlappingTriangulation<dim>
    : public dealii::Triangulation<dim, spacedim>
{
  void reinit(
    const p::s::Triangulation<dim, spacedim> &shared_tria,
    const IntersectionPredicate<dim, spacedim> &predicate);

  cell_iterator
  get_native_cell(const cell_iterator) const;
}
```

- both distributed (exists on all processors) and serial (no ghost cells). Aside from `reinit()` is completely serial - uses `Vector<double>`.
- Did you know serial Triangulations can use artifical cells?
- Alternative to `RemotePointEvaluation`, `find_active_cell_around_point()`, etc.

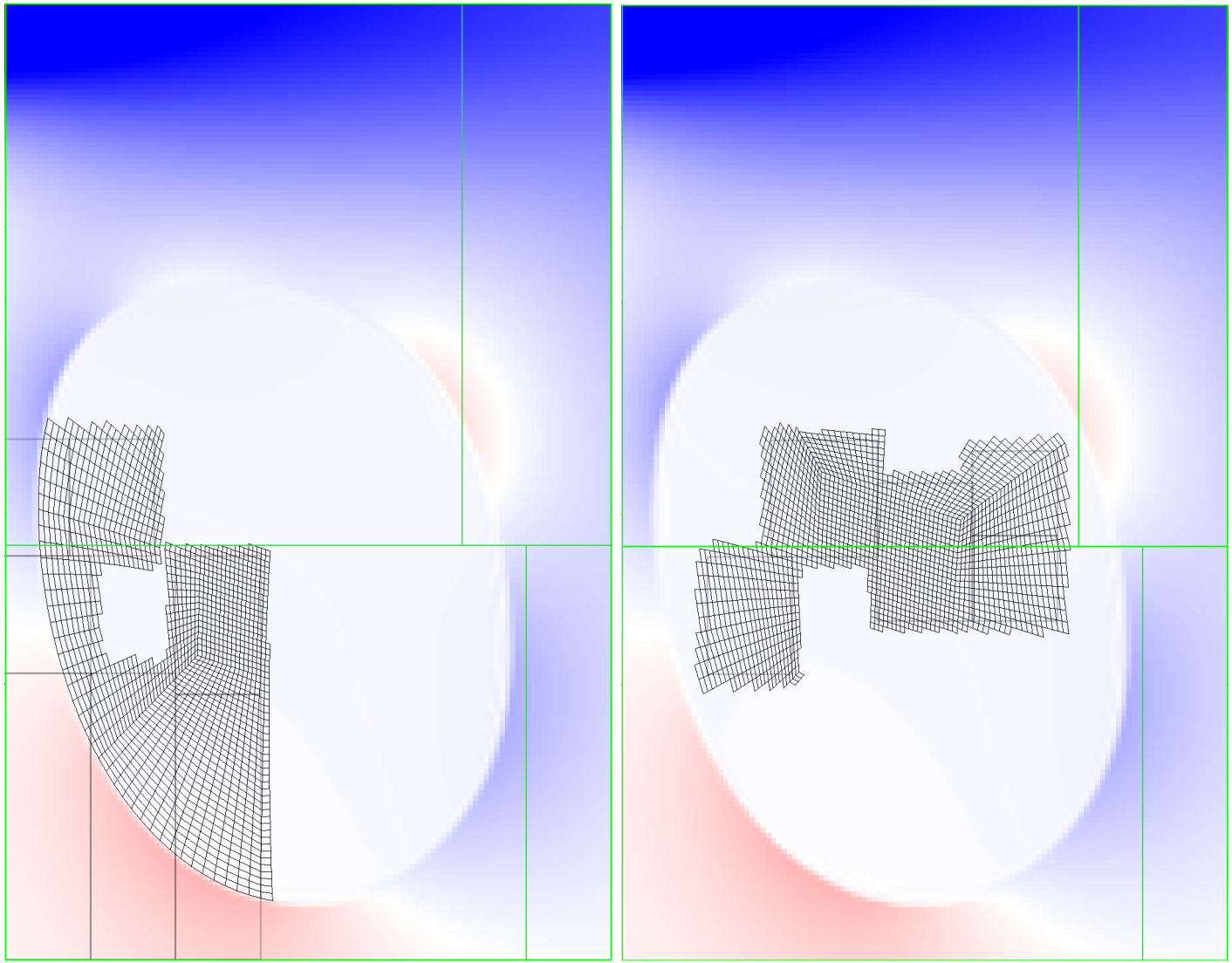
Parallel Partitioning with `fdl::OverlappingTriangulation`



Parallel Partitioning with `fdl::OverlappingTriangulation`



Parallel Partitioning with `fdl::OverlappingTriangulation`



Parallel Partitioning with `fdl::OverlappingTriangulation`

Outline of algorithm:

1. Compute bounding boxes for SAMRAI patches on the current processor. Put them in a `boost::rtree`.
2. Compute bounding boxes for deal.II cells on the current processor.
3. Gather *all* cell bounding boxes on the current processor.
4. Intersect the global bounding box list with the rtree.

DoF index translation is similar:

1. Given an `OverlappingTriangulation`, find the equivalent cells in the `p::s::T` (the native Triangulation).
2. Send requested (via `some_to_some`) active cell indices to owning (native) processors.
3. Send dofs on active cells (via `some_to_some`) back to owning (overlap) processors.
4. Create the mapping from purely local overlap dofs to distributed native dofs.

Other things in fiddle

- Nice API for specifying PK1 stresses, structural body forces, etc (pressure is WIP)
- `MechanicsValues`, like `FEValues` but for common solid mechanics quantities
- `PatchMap`, Iterate over `DoFHandler` iterators intersecting a given patch
- `QuadratureFamily`, adaptive quadrature to satisfy IB point density conditions
- `Scatter` class built on `LA::d::Vector` for moving data from native to overlap and vice versa

Let me know if you are interested in using it or pieces of it. If there is some interest in this approach we can move things into deal.II proper.

Thanks for listening!