PetIGA

A Framework for High Performance Isogeometric Analysis

Lisandro Dalcin[1,3], Nathaniel Collier[2],
Adriano Côrtes[1], Philippe Vignal[1], Victor M. Calo[1]

[1]King Abdullah University of Science and Technology (KAUST)
Thuwal, Saudi Arabia

[2]Oak Ridge National Laboratory (ORNL),
Knoxville, United States

[3]Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
Santa Fe, Argentina

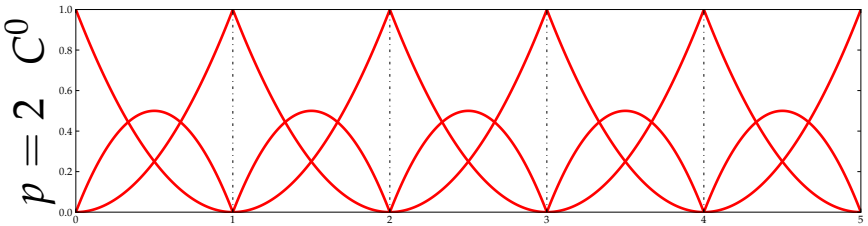
Celebrating 20 Years of Computational Science with PETSc
Tutorial and Conference
June 18, 2015

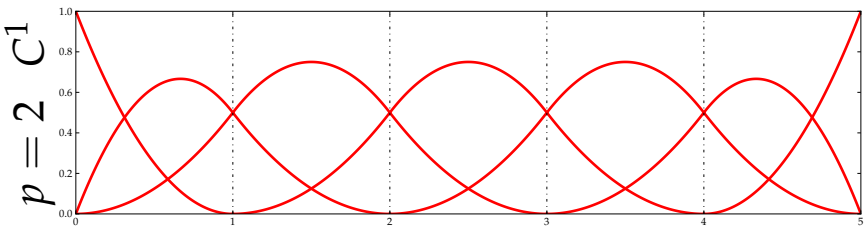
What is PetIGA?

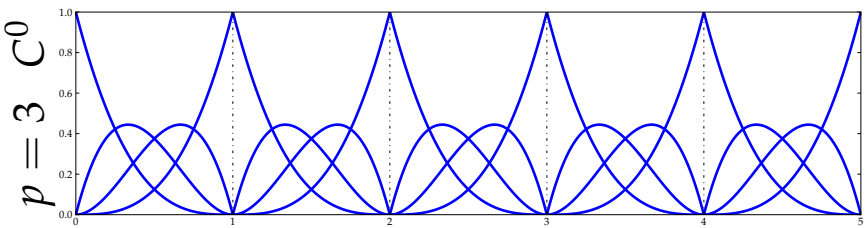
- An implementation of isogeometric analysis
- Built on top of PETSc preserving patterns and idioms
- Easy to get started (if you know PETSc in advance!)

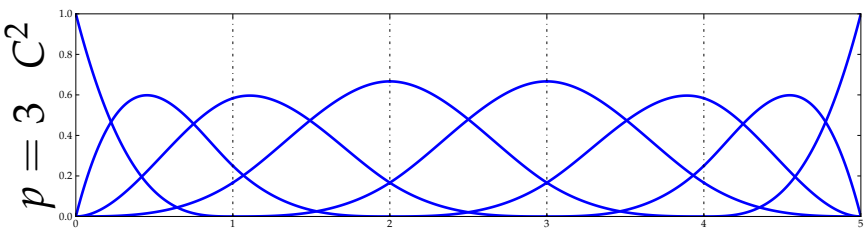
PetIGA = Pet (PETSc) + IGA (isogeometric analysis)

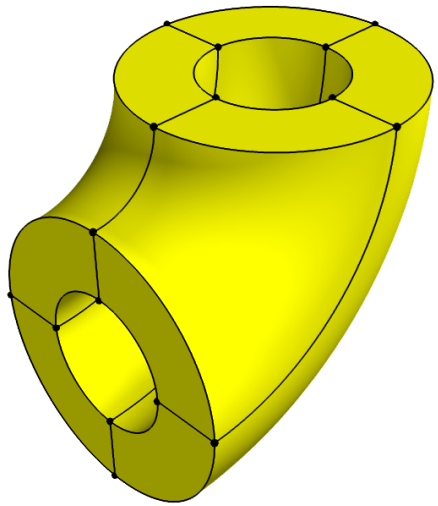
IGA: FEM + B-Spline/NURBS

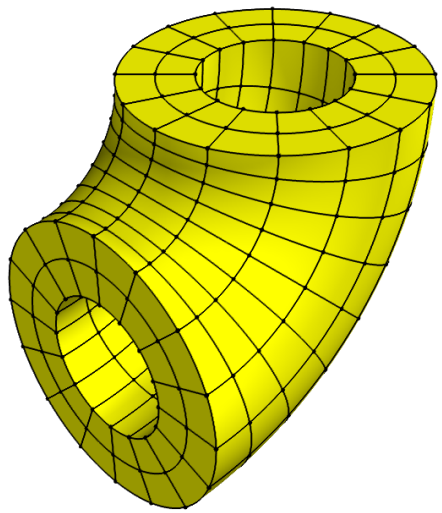


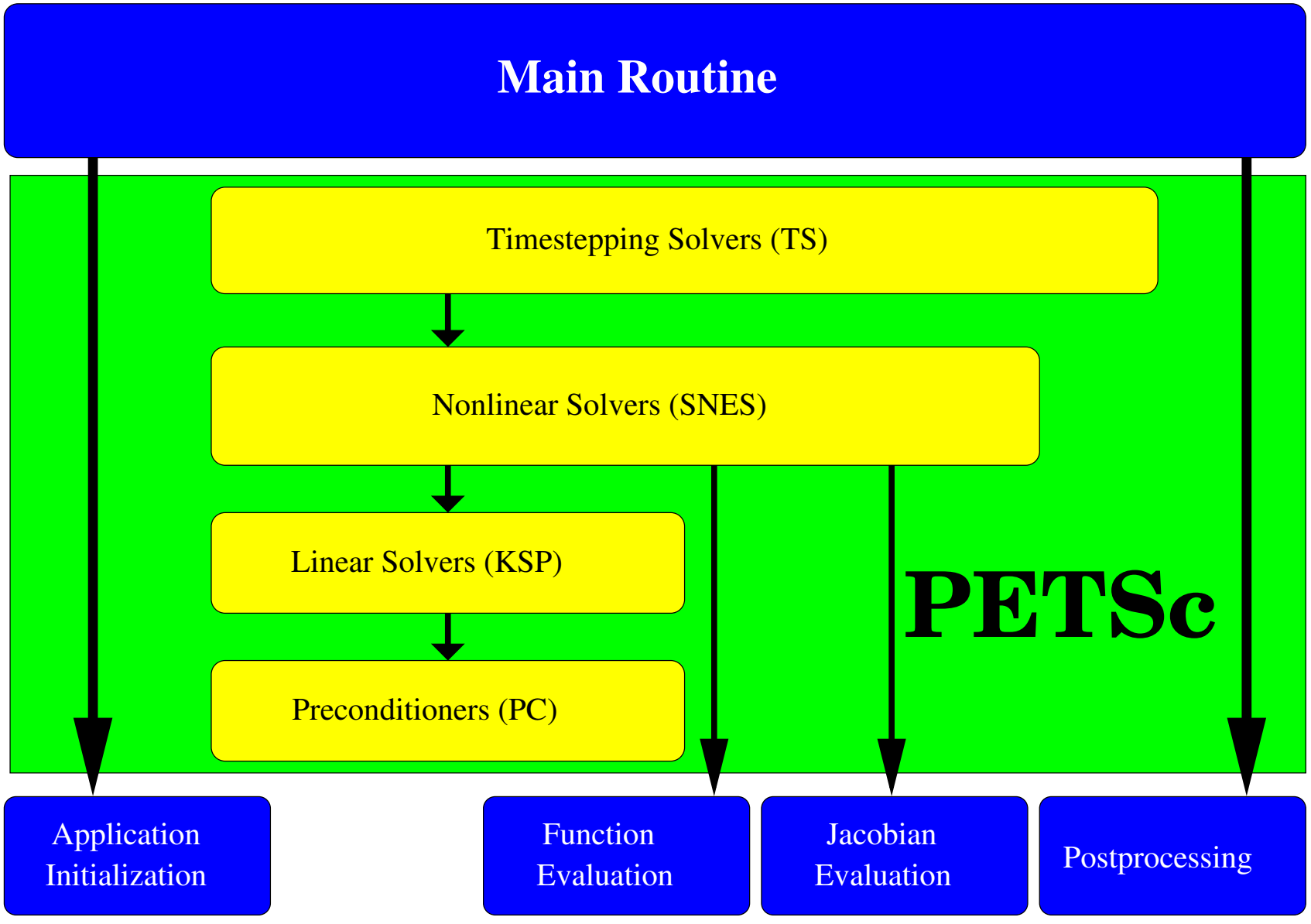








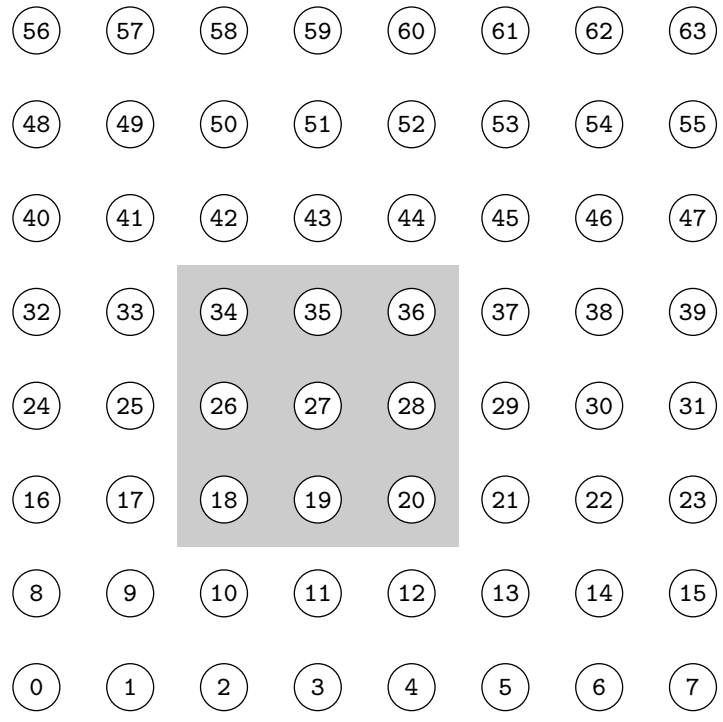



Main Routine
Timestepping Solvers (TS)
Nonlinear Solvers (SNES)
Linear Solvers (KSP)
Preconditioners (PC)
PETSc
Application Initialization
Function Evaluation
Jacobian Evaluation
Postprocessing

Parallel implementation

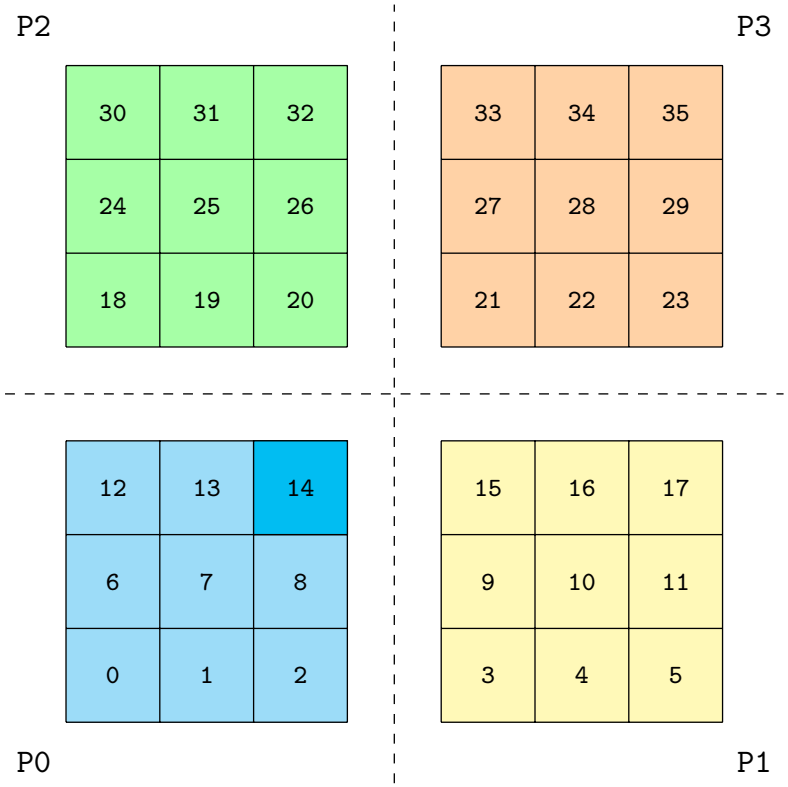
30	31	32	33	34	35
24	25	26	27	28	29
18	19	20	21	22	23
12	13	14	15	16	17
6	7	8	9	10	11
0	1	2	3	4	5

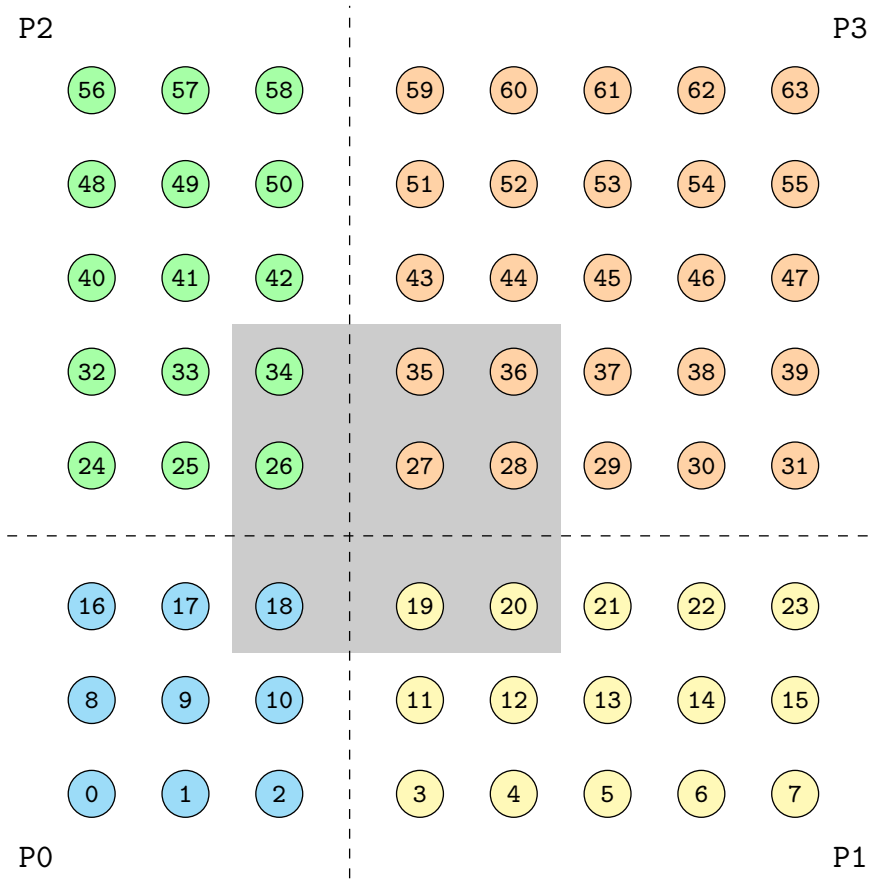
6×6 element grid



8×8 node grid
(quadratic C^1 space)

Parallel implementation

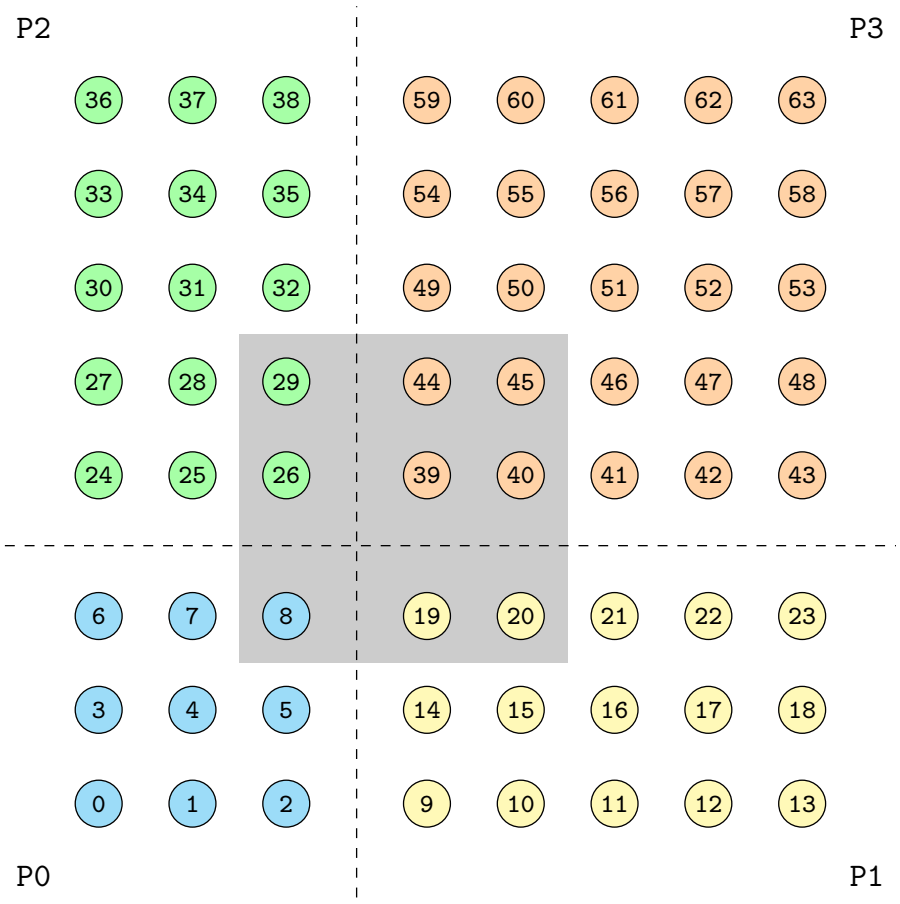




global node rid

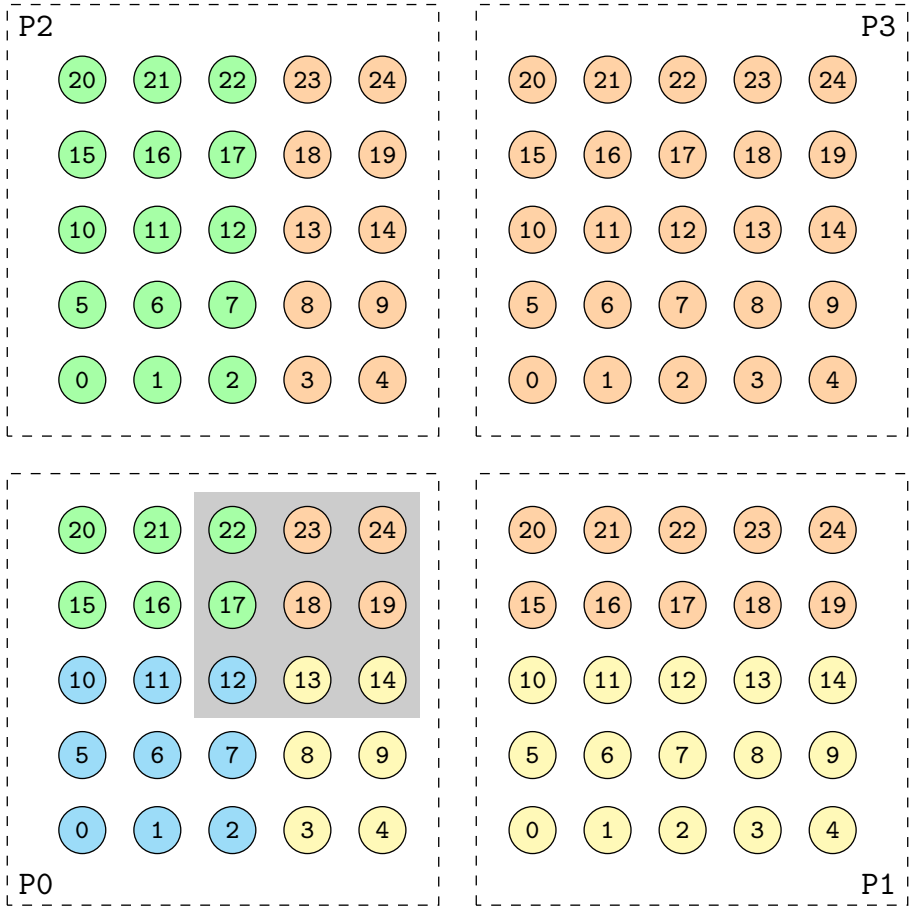
natural numbering

Parallel implementation



global node grid

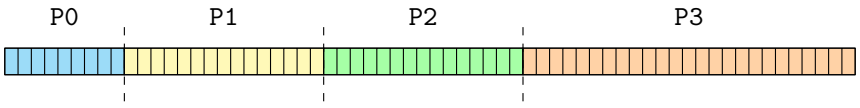
global numbering



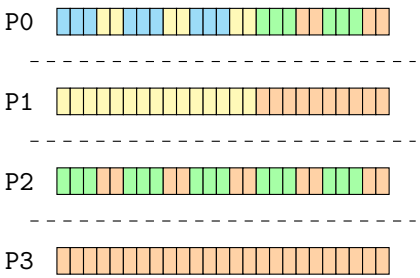
local node grids

local numbering

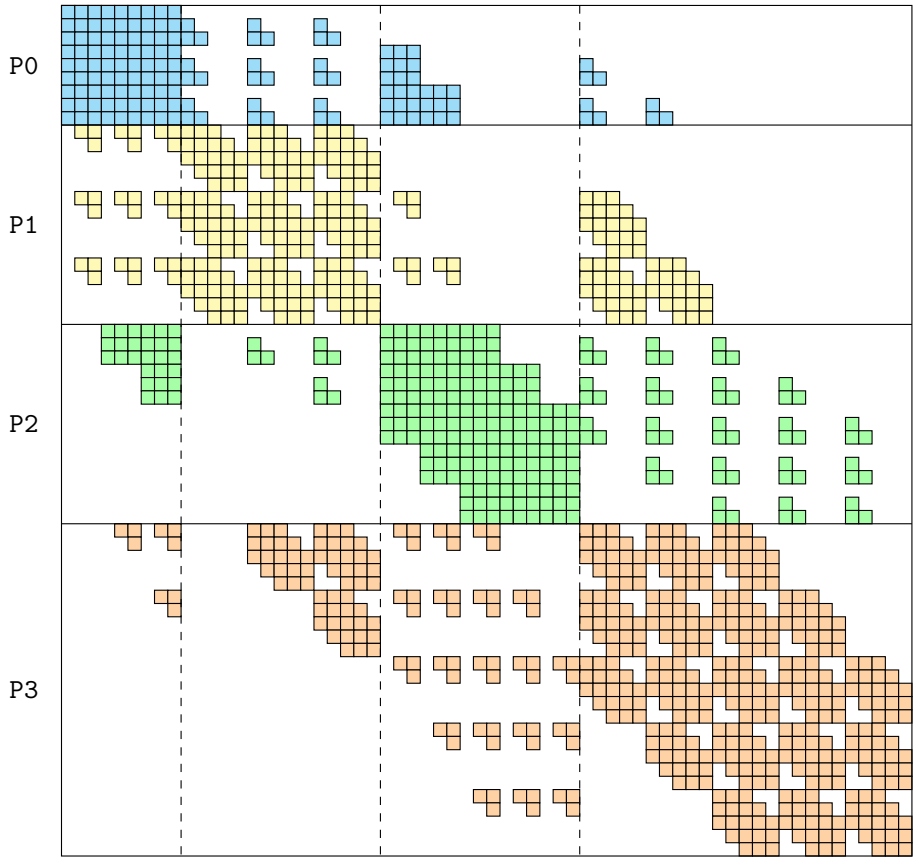
Parallel implementation



global vector



local vectors



global sparse matrix

Assembly

1: **function** FormFunction($\mathbf{U}$)
2: $\quad \mathbf{U}_\ell \longleftarrow \mathbf{U}$
3: $\quad$ **for each** element e in subpatch **do**
4: $\quad\quad \mathbf{U}_e \longleftarrow \mathbf{U}_\ell$
5: $\quad\quad \mathbf{G}_e \longleftarrow$ Geometry$[e]$
6: $\quad\quad$ **for each** quadrature point q in element e **do**
7: $\quad\quad\quad \mathbf{x}_q, w_q \longleftarrow$ Quadrature(q)
8: $\quad\quad\quad J_q, \{N_q\} =$ ShapeFuns$(\mathbf{x}_q, \mathbf{G}_e)$
9: $\quad\quad\quad \mathbf{F}_q =$ UserFunction$(\mathbf{x}_q, \{N_q\}, \mathbf{U}_e)$
10: $\quad\quad\quad J_q w_q \mathbf{F}_q \xrightarrow{+} \mathbf{F}_e$
11: $\quad\quad$ **end for**
12: $\quad\quad \mathbf{F}_e \xrightarrow{+} \mathbf{F}_{\text{stash}}$
13: $\quad$ **end for**
14: $\quad \mathbf{F}_{\text{stash}} \xrightarrow{+} \mathbf{F}$
15: $\quad$ return $\mathbf{F}$
16: **end function**

Demo: Bratu Problem

- Strong form

  Find $u : \bar{\Omega} \to \mathbb{R}$ such that

$$\begin{aligned} -\Delta u &= \lambda \exp(u), & \boldsymbol{x} &\in \Omega, \\ u &= 0, & \boldsymbol{x} &\in \partial\Omega. \end{aligned}$$

- Weak form (Galerkin)

  Find $u \in \mathcal{V}$ such that for all $w \in \mathcal{V}$

$$(\nabla w, \nabla u)_\Omega - (w, \lambda \exp(u))_\Omega = 0.$$

Demo: Bratu Problem

- Define some constants

```
#define dof 1 // scalar problem
#define dim 2 // two dimensions
```

- Initialize the IGA context

```
IGA iga;
IGACreate(PETSC_COMM_WORLD,&iga);
IGASetDof(iga,dof);
IGASetDim(iga,dim);
IGASetFromOptions(iga);
IGASetUp(iga);
```

Demo: Bratu Problem

- Set boundary conditions

```
for (dir=0; dir<dim; dir++)
  for (side=0; side<2; side++)
    IGASetBoundaryValue(iga,dir,side,0,0.0);
```

- Specify evaluation routines

```
Params params = { .lambda = 6.80 };
IGASetFormFunction(iga,Function,&params);
IGASetFormJacobian(iga,Jacobian,&params);
```

Demo: Bratu Problem

```c
typedef struct { double lambda; } Params;
#define dot(a,b) (a[0]*b[0]+a[1]*b[1])

int Function(IGAPoint p,const double U[],double F[],void *ctx)
{
  int     a,nen       = p->nen;
  double (*N0)        = (typeof(N0)) p->shape[0];
  double (*N1)[dim]   = (typeof(N1)) p->shape[1];
  double u,grad_u[dim],lambda = ((Params*)ctx)->lambda;
  IGAPointFormValue(p,U,&u);
  IGAPointFormGrad (p,U,grad_u);
  for (a=0; a<nen; a++)
    F[a] = dot(N1[a],grad_u) - lambda*exp(u)*N0[a];
  return 0;
}
```

Demo: Bratu Problem

```
int Jacobian(IGAPoint p,const double U[],double J[],void *ctx)
{
  int     a,b,nen     = p->nen;
  double (*N0)        = (typeof(N0)) p->shape[0];
  double (*N1)[dim]   = (typeof(N1)) p->shape[1];
  double u,lambda = ((Params*)ctx)->lambda;
  IGAPointFormValue(p,U,&u);
  for (a=0; a<nen; a++)
    for (b=0; b<nen; b++)
      J[a*nen+b] = dot(N1[a],N1[b]) -
                   lambda*exp(u)*N0[a]*N0[b];
  return 0;
}
```

Demo: Bratu Problem

- Initialize nonlinear solver context

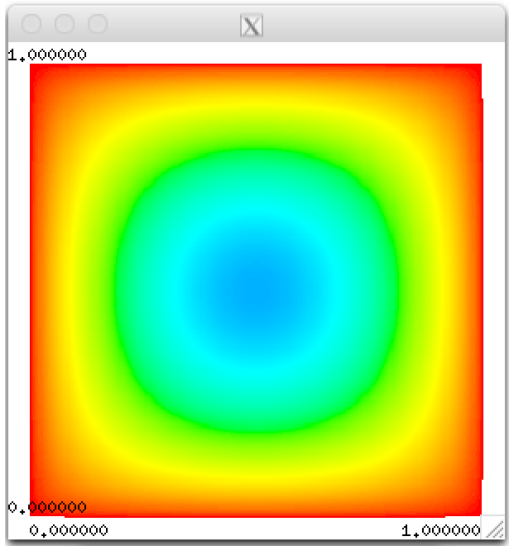
```
SNES snes; // nonlinear solver
IGACreateSNES(iga,&snes);
SNESSetFromOptions(snes);
```

- Solve nonlinear problem

```
Vec U; // solution vector
IGACreateVec(iga,&U);
SNESSolve(snes,NULL,U);
VecViewFromOptions(U,NULL,"-view");
```

Demo: Bratu Problem

```
$ mpiexec -n 4 ./Bratu -iga_elements 128 -iga_degree 2 -iga_view \
                       -ksp_type cg -sub_pc_type icc              \
                       -snes_monitor                              \
                       -view draw:x -draw_pause -1
IGA: dim=2 dof=1 order=2 geometry=0 rational=0 property=0
Axis 0: basis=BSPLINE[2,1] rule=LEGENDRE[3] periodic=0 nnp=130 nel=128
Axis 1: basis=BSPLINE[2,1] rule=LEGENDRE[3] periodic=0 nnp=130 nel=128
Partition - MPI: processors=[2,2,1] total=4
Partition - nnp: sum=16900 min=4096 max=4356 max/min=1.06348
Partition - nel: sum=16384 min=4096 max=4096 max/min=1
  0 SNES Function norm 5.266384548611e-02
  1 SNES Function norm 8.620220401724e-03
  2 SNES Function norm 2.054605014212e-03
  3 SNES Function norm 4.716226279209e-04
  4 SNES Function norm 8.916608064674e-05
  5 SNES Function norm 8.438014748365e-06
  6 SNES Function norm 1.155533195923e-07
  7 SNES Function norm 2.309601078808e-11
```



Numerical differentiation

- PETSc: Two built-in options (global evaluations)
  - Matrix-free Newton–Krylov (`-snes_mf`)
  - Coloring finite differences (`-snes_fd_color`)
- PetIGA: Local FD at quadrature points

$$\frac{\partial \mathbf{F}_q}{\partial \mathbf{U}_e}\hat{\mathbf{e}}_k \approx \frac{1}{\delta}\Big(\mathbf{F}_q(\mathbf{U}_e + \delta\hat{\mathbf{e}}_k) - \mathbf{F}_q(\mathbf{U}_e)\Big)$$

$$\delta = \eta\sqrt{1 + \|\mathbf{U}_e\|} \qquad \eta = \sqrt{\epsilon} \qquad \epsilon \approx 2.22 \times 10^{-16}$$

```
Params params = { .lambda = 6.80 };
IGASetFormFunction(iga,Residual,&params);
IGASetFormJacobian(iga,IGAFormJacobianFD,&params);
```

Numerical differentiation

				Time (seconds)		
P	N	p	k	Explicit	Coloring FD	Local FD
8	32^3	2	0	0.31	7.67	1.21
			1	0.33	7.79	1.24
		3	2	3.71	105.08	15.27
16	64^3	2	0	1.31	34.44	5.15
			1	1.35	34.84	5.26
		3	2	15.41	452.67	64.93
	128^3	1	0	0.52	10.35	1.72
		2	1	10.74	307.06	41.46
				baseline	20X - 30X	$\approx$ 4X

Numerical differentiation

Practical approach to nonlinear problems

1. Code residual function (Gauss point evaluation)
2. Use PetIGA local FD approach to approximate Jacobian
3. Code Jacobian function (Gauss point evaluation)
4. Check Jacobian correctness using matrix-free + LU
   - Approximate Jacobian with matrix-free (`-snes_mf_operator`)
   - Invert computed Jacobian as a preconditioner (`-pc_type lu`)

   If Jacobian is correct, KSP converges in one iteration

Geometry handling

Creation of initial (simple?) geometries is a nontrivial task

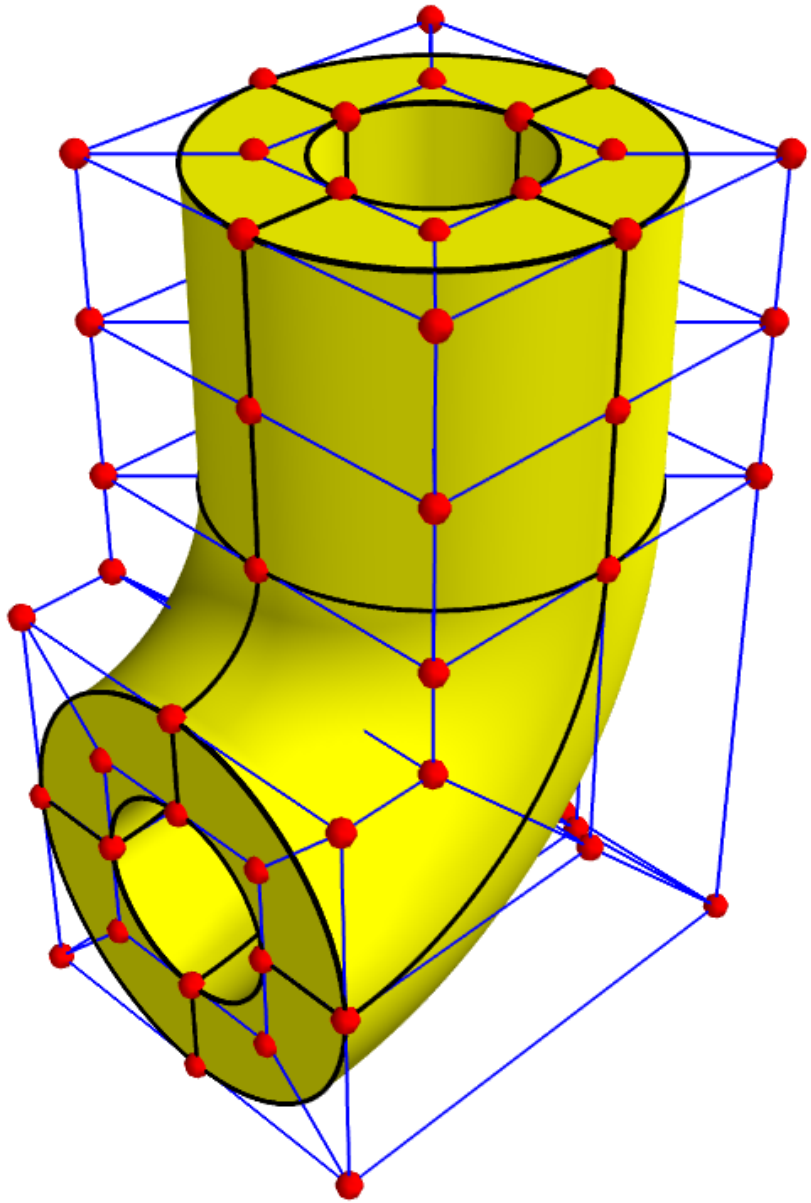
- Volume NURBS representations are cumbersome and are (mostly) developed manually
- Bridging the CAD/CAE gap is on going

Geometry handling made easier by:

1. Running in *parametric mode* if possible – avoid geometrical mapping cubes into cubes
2. Creating Python interfaces to low-level NURBS routines (knot insertion, degree elevation) – Python scripting is very flexible
3. Writing binary files that PETSc reads in parallel
   - No need to manually partition the domain
   - Sidesteps issue of parallel I/O

Geometry preparation

```
from igakit.cad import *

C0 = circle(radius=1)
C1 = circle(radius=2)

annulus = ruled(C0, C1)

pipe = extrude(annulus,
               displ=3.0, axis=2,
               ).reverse(2)

elbow = revolve(annulus,
                point=(3,0,0),
                axis=(0,-1,0),
                angle=Pi/2)

bentpipe = join(pipe, elbow, axis=2)
```



Solver Scalability

- Incompressible Navier–Stokes with VMS turbulence modeling
- 10 time steps $\times$ 2 Newton steps $\times$ 30 GMRES iterations
- B-spline space: $p = 2$, C^1; geometrical mapping: identity

Parallel efficiency, single node (Lonestar, TACC)

	Number of cores						
mesh	1	2	4	6	8	10	12
32^3	100%	98%	91%	84%	85%	77%	81%
64^3	100%	93%	85%	77%	·	79%	77%

Parallel efficiency, multiple node (Lonestar, TACC)

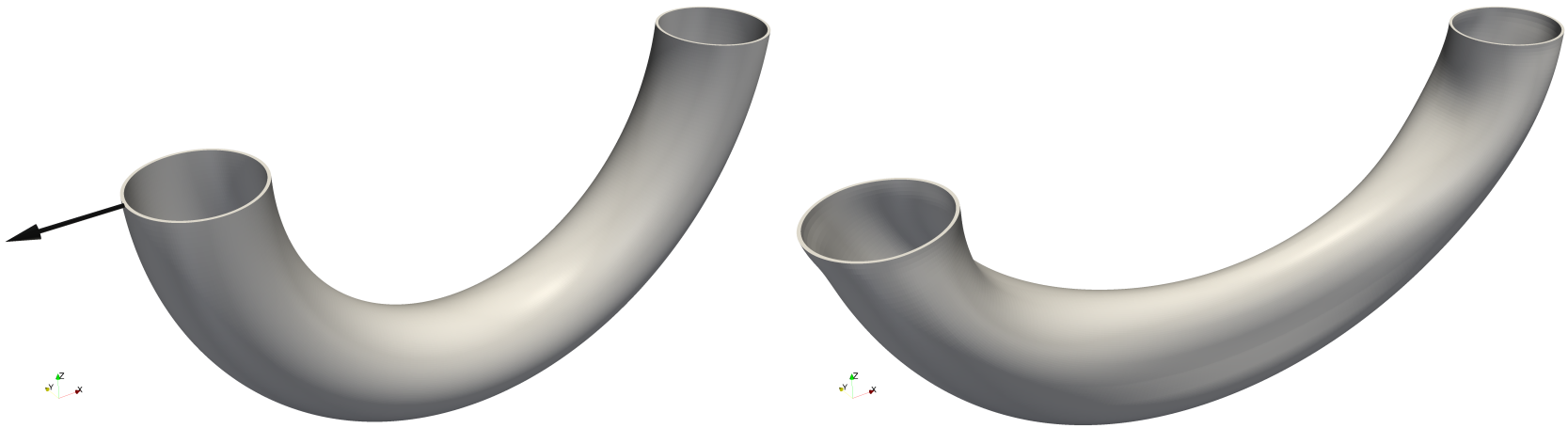
	Number of cores					
mesh	64	216	512	1000	1728	4104
120^3	100%	102%	100%	97%	87%	·
168^3	·	100%	90%	91%	93%	74%

Relevant Applications

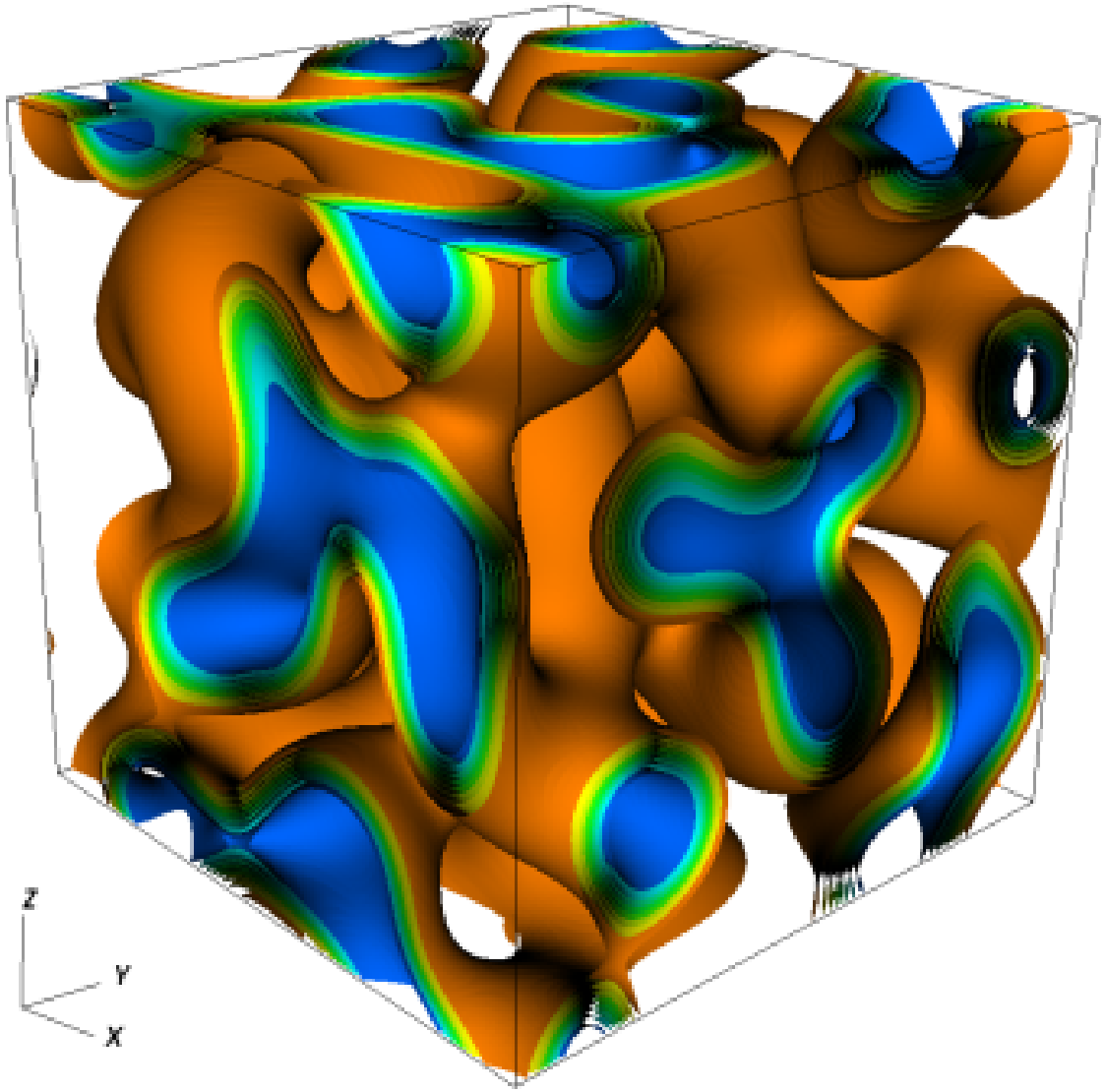
Framework used to address many 2D/3D applications

- Linear and Nonlinear Elasticity
- Cahn–Hilliard equations
- Navier–Stokes–Korteweg equations
- Phase-field Crystal equations
- Variational Multiscale for Navier–Stokes
- Diffusive Wave approximation to Shallow Water equations
- Pattern Formation (Advection-Diffusion-Reaction systems)

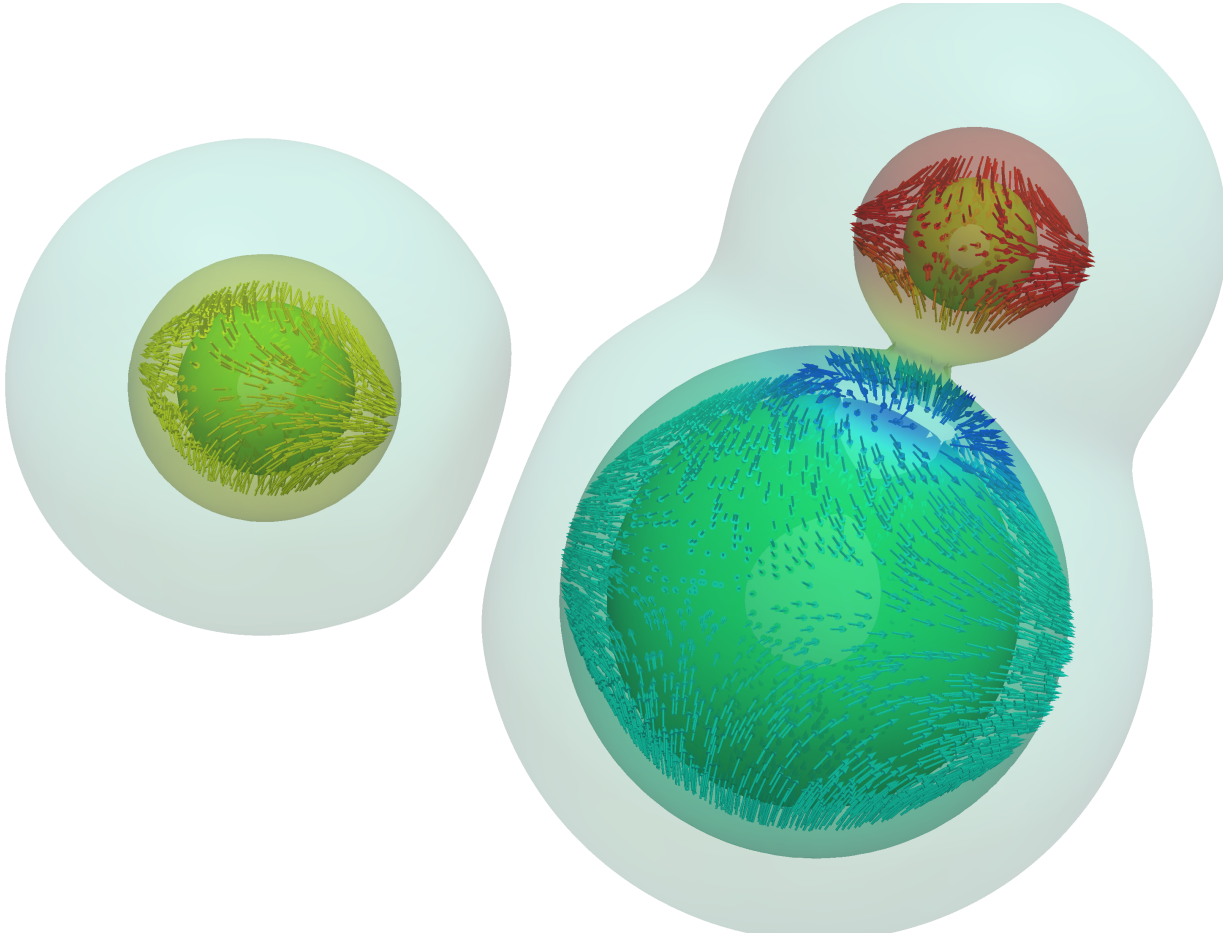
Applications: Hyper-Elasticity



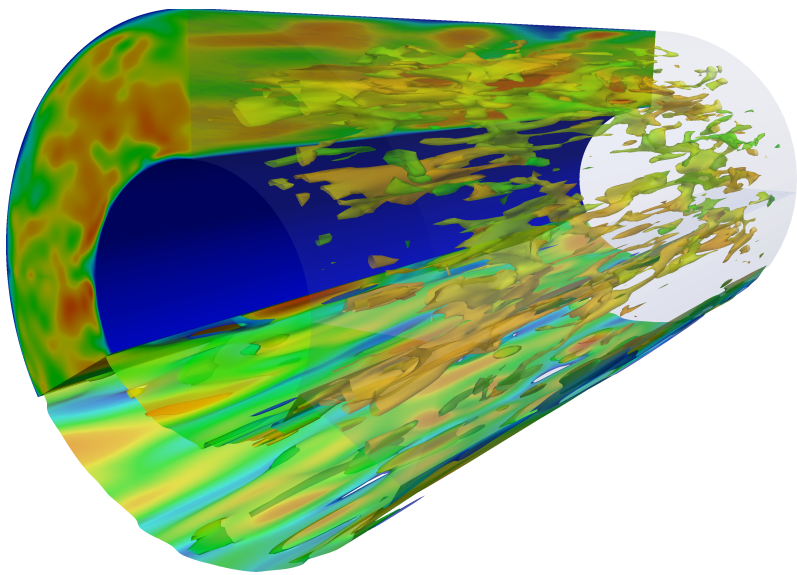
Applications: Cahn-Hilliard

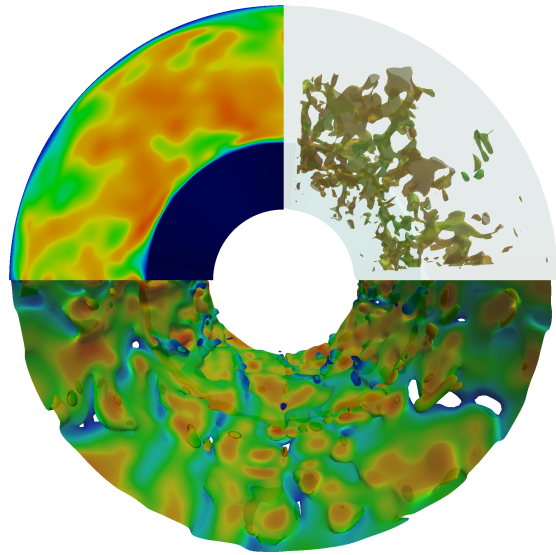


Applications: Navier–Stokes–Korteweg



Applications: Navier–Stokes + VMS





Limitations

- Dirichlet BCs on whole boundary faces
- Tensor-product, single-patch geometries
- Standard, tensor-product Gauss–Legendre quadrature
- Vector/mixed problems (same space for each component)

Source code:

- https://bitbucket.org/dalcinl/petiga
- https://bitbucket.org/dalcinl/igakit

Comments & Questions:

- dalcinl@gmail.com
- nathaniel.collier@gmail.com
- adrimacortes@gmail.com