Solving partial differential equations efficiently and productively with Firedrake and PyOP2 - and how we got there

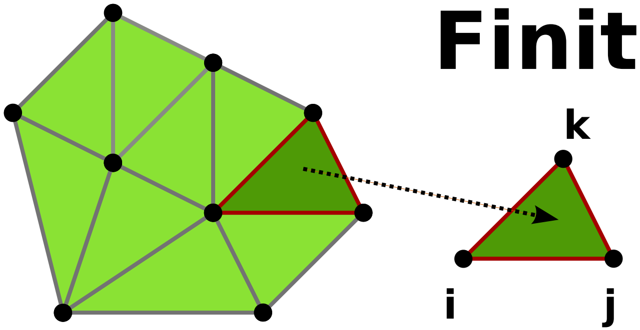
http://firedrakeproject.org

Florian Rathgeber[1], Lawrence Mitchell[1], David Ham[1,2], Michael Lange[3], Andrew McRae[2], Fabio Luporini[1], Gheorghe-teodor Bercea[1], Paul Kelly[1]

Slides: http://kynan.github.io/FiredrakeSeminar2014

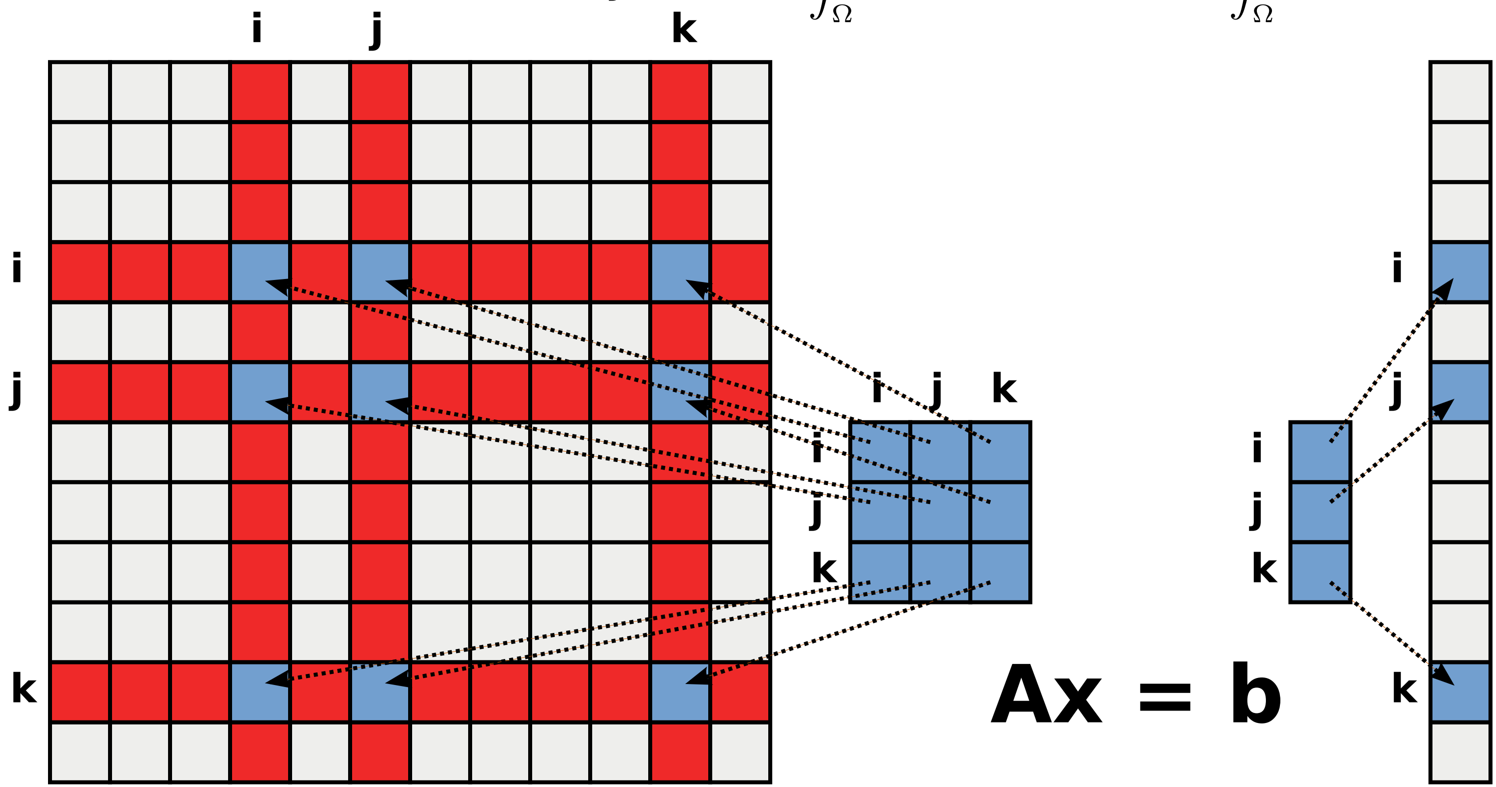
[1] Department of Computing, Imperial College London [2] Department of Mathematics, Imperial College London
[3] Department of Earth Science & Engineering, Imperial College London

Finite-element assembly



The weak form of the Helmholtz equation:

$$\int_\Omega \nabla v \cdot \nabla u - \lambda v u \; \mathrm{d}V = \int_\Omega v f \; \mathrm{d}V$$



Solving the Helmholtz equation in Python using Firedrake

$$\int_\Omega \nabla v \cdot \nabla u - \lambda v u \, dV = \int_\Omega v f \, dV$$

```
from firedrake import *

# Read a mesh and define a function space
mesh = Mesh('filename')
V = FunctionSpace(mesh, "Lagrange", 1)

# Define forcing function for right-hand side
f = Expression("- (lmbda + 2*(n**2)*pi**2) * sin(X[0]*pi*n) * sin(X[1]*pi*n)",
               lmbda=1, n=8)

# Set up the Finite-element weak forms
u = TrialFunction(V)
v = TestFunction(V)

lmbda = 1
a = (dot(grad(v), grad(u)) - lmbda * v * u) * dx
L = v * f * dx

# Solve the resulting finite-element equation
p = Function(V)
solve(a == L, p)
```

Unified Form Language (UFL) from the FEniCS project to describe weak form of PDE

FEniCS project

> *"The FEniCS Project is a collection of free software for automated, efficient solution of differential equations.* — *fenicsproject.org*

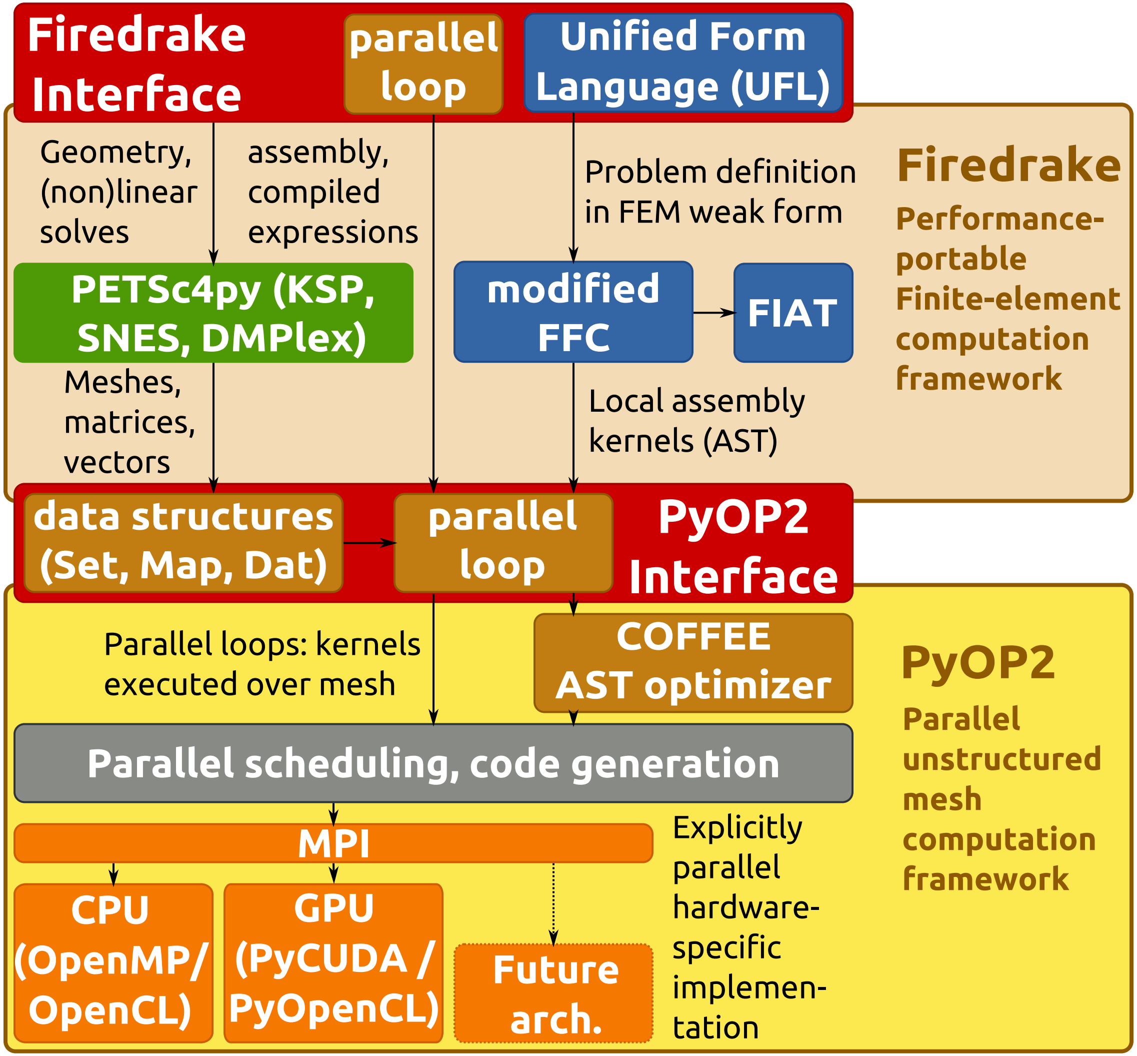
FEniCS project

> *The FEniCS Project is a collection of free software for automated, efficient solution of differential equations.* — *fenicsproject.org*

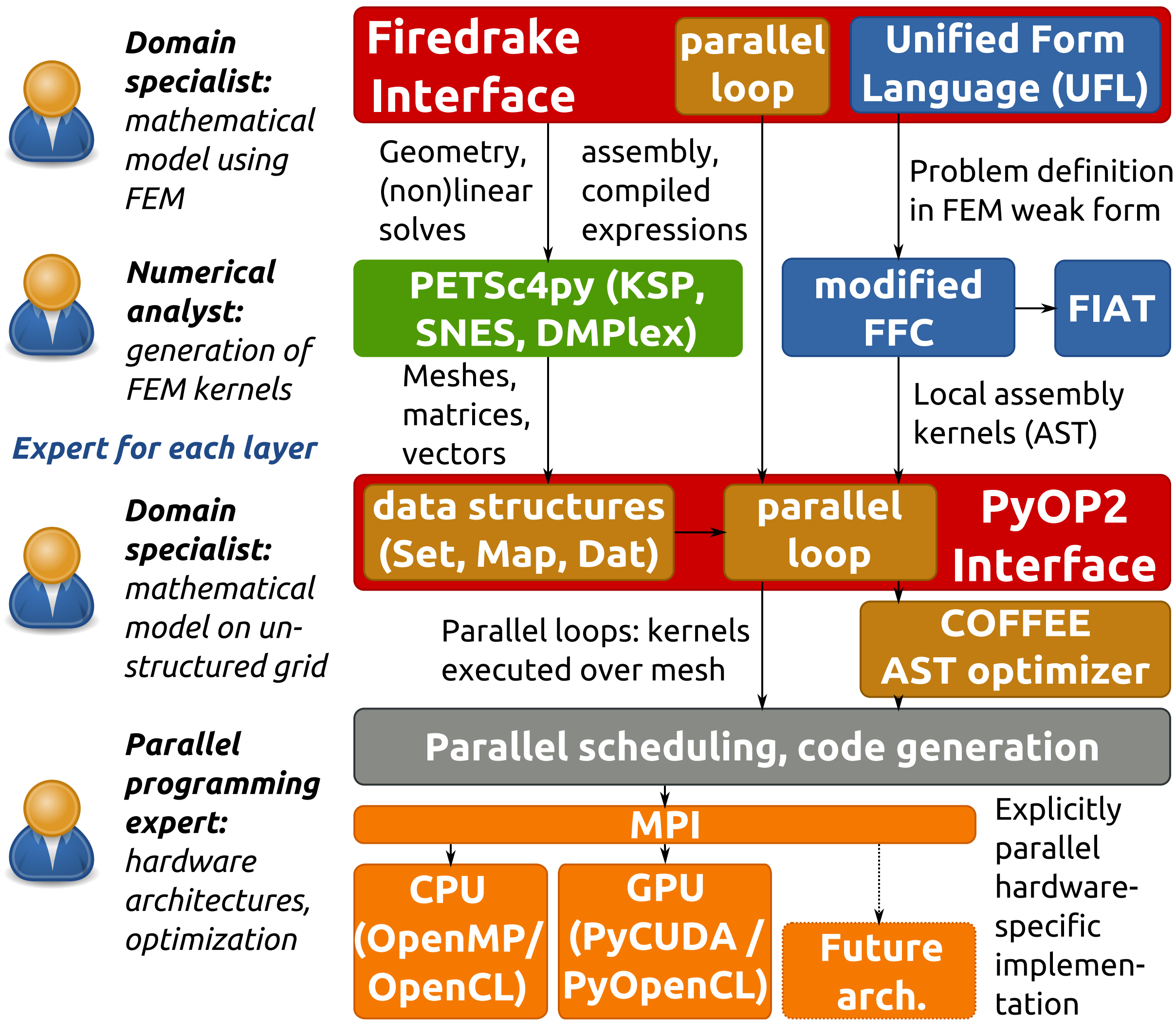
Firedrake

> *Firedrake is an automated system for the portable solution of partial differential equations using the finite element method (FEM).* — *firedrakeproject.org*

The Firedrake/PyOP2 tool chain



Two-layered abstraction: Separation of concerns



Parallel computations on unstructured meshes with PyOP2

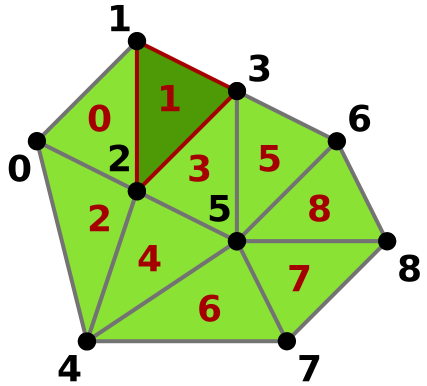
Scientific computations on unstructured meshes

- Independent *local operations* for each element of the mesh described by a *kernel*.
- *Reductions* aggregate contributions from local operations to produce final result.

PyOP2

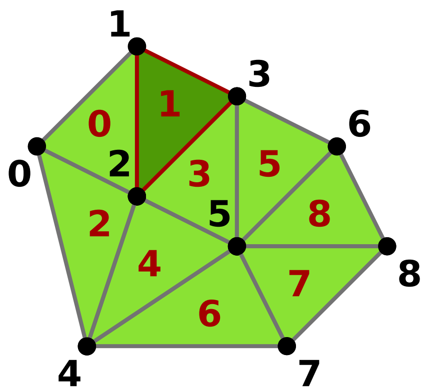
- Domain-specific language embedded in Python for data parallel computations
- Efficiently executes kernels in parallel over unstructured meshes or graphs
- Portable programmes for different architectures without code change
- Efficiency through runtime code generation and just-in-time (JIT) compilation

Unstructured mesh



PyOP2 Sets:
nodes (9 entities: 0-8)
elements (9 entities: 0-8)
PyOP2 Map elements-nodes:
elem_nodes = [[0, 1, 2], [1, 3, 2], ...]
PyOP2 Dat on nodes:
coords = [..., [.5,.5], [.5,-.25], [1,.25], ...]

PyOP2 Data Model



PyOP2 Sets:
nodes (9 entities: 0-8)
elements (9 entities: 0-8)
PyOP2 Map elements-nodes:
elem_nodes = [[0, 1, 2], [1, 3, 2], ...]
PyOP2 Dat on nodes:
coords = [..., [.5,.5], [.5,-.25], [1,.25], ...]

Mesh topology

- `Sets` – Mesh entities and data DOFs
- `Maps` – Define connectivity between entities in different `Sets`

Data

- `Dats` – Defined on `Sets` (hold data, completely abstracted vector)
- `Globals` – not associated to a `Set` (reduction variables, parameters)
- `Consts` – Global, read-only data

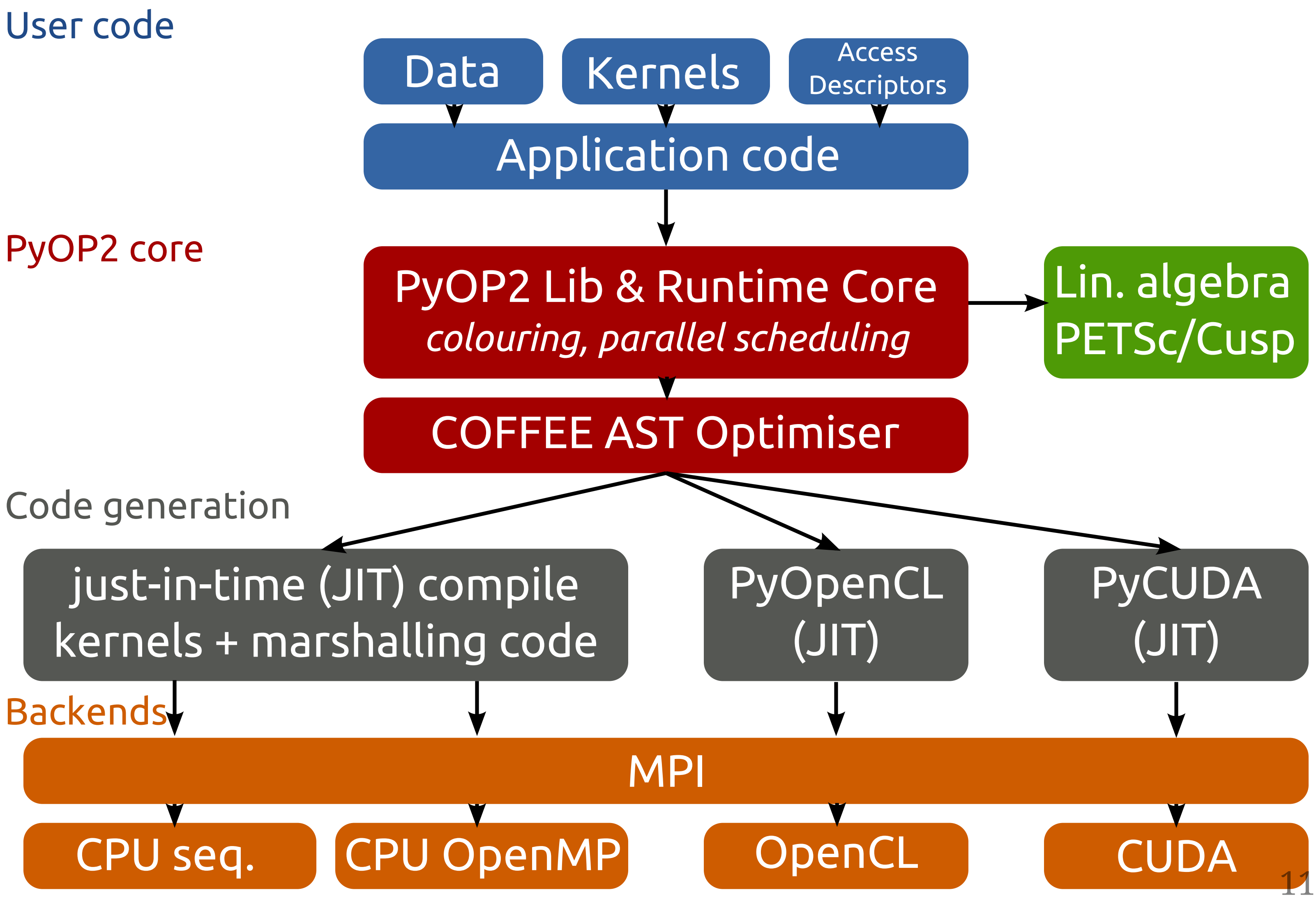
Kernels / parallel loops

- Executed in parallel on a `Set` through a parallel loop
- Read / write / increment data accessed via maps

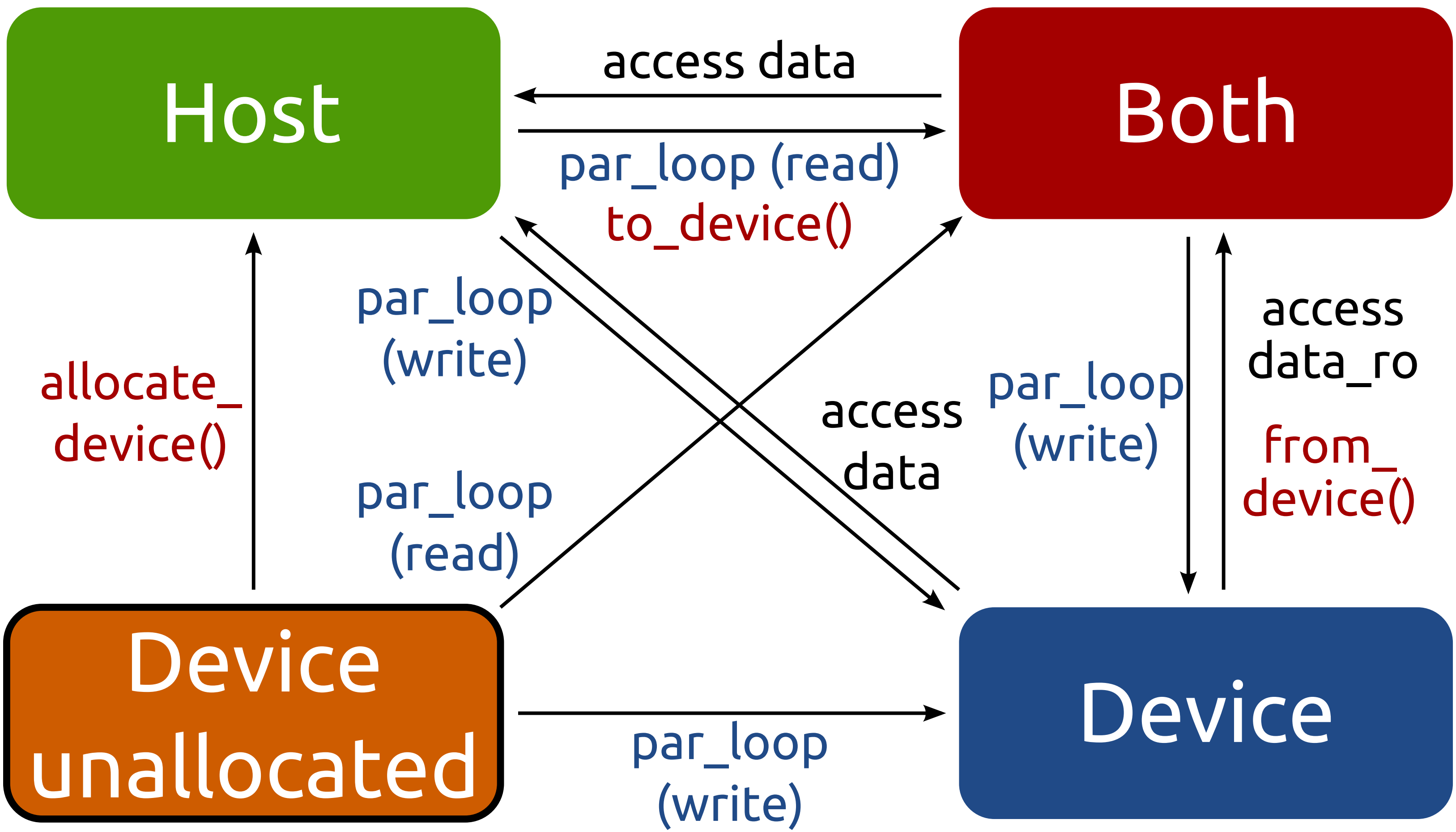
Linear algebra

- `Sparsity` patterns defined by `Maps`
- `Mat` – Matrix data on sparsities
- Kernels compute local matrix – PyOP2 handles global assembly

PyOP2 Architecture



PyOP2 Device Data Management



PyOP2 Kernels & Parallel Loops

Kernels:

- "local view" of the data
- sequential semantics

Parallel loop:

- use access descriptors to generate marshalling code
- pass "right data" to kernel for each iteration set element

Kernel for computing the midpoint of a triangle

```
void midpoint(double p[2], double *coords[2]) {
  p[0] = (coords[0][0] + coords[1][0] + coords[2][0]) / 3.0;
  p[1] = (coords[0][1] + coords[1][1] + coords[2][1]) / 3.0;
}
```

PyOP2 programme for computing midpoints over the mesh

```
from pyop2 import op2
op2.init()

vertices = op2.Set(num_vertices)
cells = op2.Set(num_cells)

cell2vertex = op2.Map(cells, vertices, 3, [...])

coordinates = op2.Dat(vertices ** 2, [...], dtype=float)
midpoints = op2.Dat(cells ** 2, dtype=float)

midpoint = op2.Kernel(kernel_code, "midpoint")

op2.par_loop(midpoint, cells,
             midpoints(op2.WRITE),
             coordinates(op2.READ, cell2vertex))
```

Kernels as abstract syntax tree (AST), C string or Python function (not currently compiled!)

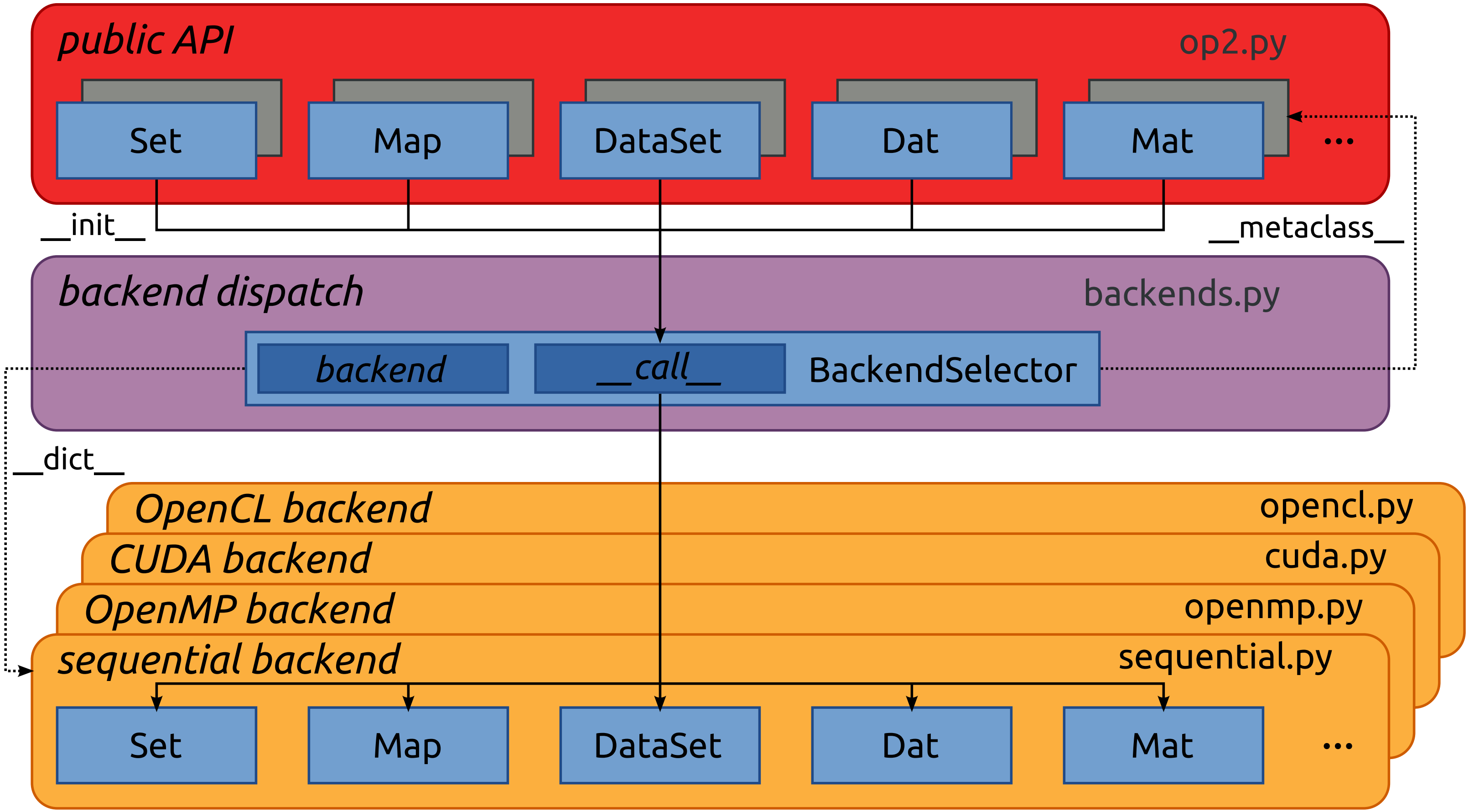
Generated sequential code calling the midpoint kernel

```
// Kernel provided by the user
static inline void midpoint(double p[2], double *coords[2]) {
  p[0] = (coords[0][0] + coords[1][0] + coords[2][0]) / 3.0;
  p[1] = (coords[0][1] + coords[1][1] + coords[2][1]) / 3.0;
}

// Generated marshaling code executing the sequential loop
void wrap_midpoint(int start, int end,
                   double *arg0_0, double *arg1_0, int *arg1_0_map0_0) {
  double *arg1_0_vec[3];
  for ( int n = start; n < end; n++ ) {
    int i = n;
    arg1_0_vec[0] = arg1_0 + (arg1_0_map0_0[i * 3 + 0])* 2;
    arg1_0_vec[1] = arg1_0 + (arg1_0_map0_0[i * 3 + 1])* 2;
    arg1_0_vec[2] = arg1_0 + (arg1_0_map0_0[i * 3 + 2])* 2;
    midpoint(arg0_0 + i * 2, arg1_0_vec);  // call user kernel (inline)
  }
}
```

Generated OpenMP code calling the midpoint kernel

```
// Kernel provided by the user
static inline void midpoint(double p[2], double *coords[2]) {
  p[0] = (coords[0][0] + coords[1][0] + coords[2][0]) / 3.0;
  p[1] = (coords[0][1] + coords[1][1] + coords[2][1]) / 3.0;
}

// Generated marshaling code executing the parallel loop
void wrap_midpoint(int boffset, int nblocks,
                   int *blkmap, int *offset, int *nelems,
                   double *arg0_0, double *arg1_0, int *arg1_0_map0_0) {
  #pragma omp parallel shared(boffset, nblocks, nelems, blkmap) {
    int tid = omp_get_thread_num();
    double *arg1_0_vec[3];
    #pragma omp for schedule(static)
    for ( int __b = boffset; __b < boffset + nblocks; __b++ ) {
      int bid = blkmap[__b];
      int nelem = nelems[bid];
      int efirst = offset[bid];
      for (int n = efirst; n < efirst+ nelem; n++ ) {
        int i = n;
        arg1_0_vec[0] = arg1_0 + (arg1_0_map0_0[i * 3 + 0])* 2;
        arg1_0_vec[1] = arg1_0 + (arg1_0_map0_0[i * 3 + 1])* 2;
        arg1_0_vec[2] = arg1_0 + (arg1_0_map0_0[i * 3 + 2])* 2;
        midpoint(arg0_0 + i * 2, arg1_0_vec);  // call user kernel (inline)
      }
    }
  }
}
```

PyOP2 Backend Selection



Why OP2 is not enough

- Static analysis at compile time: "Synthesis is easy, analysis is hard!"
- No object introspection, attributes needs to be explicit in code
- User code compiled for a specific backend, linked against runtime library

`adt_calc` kernel in the OP2 Airfoil example application:

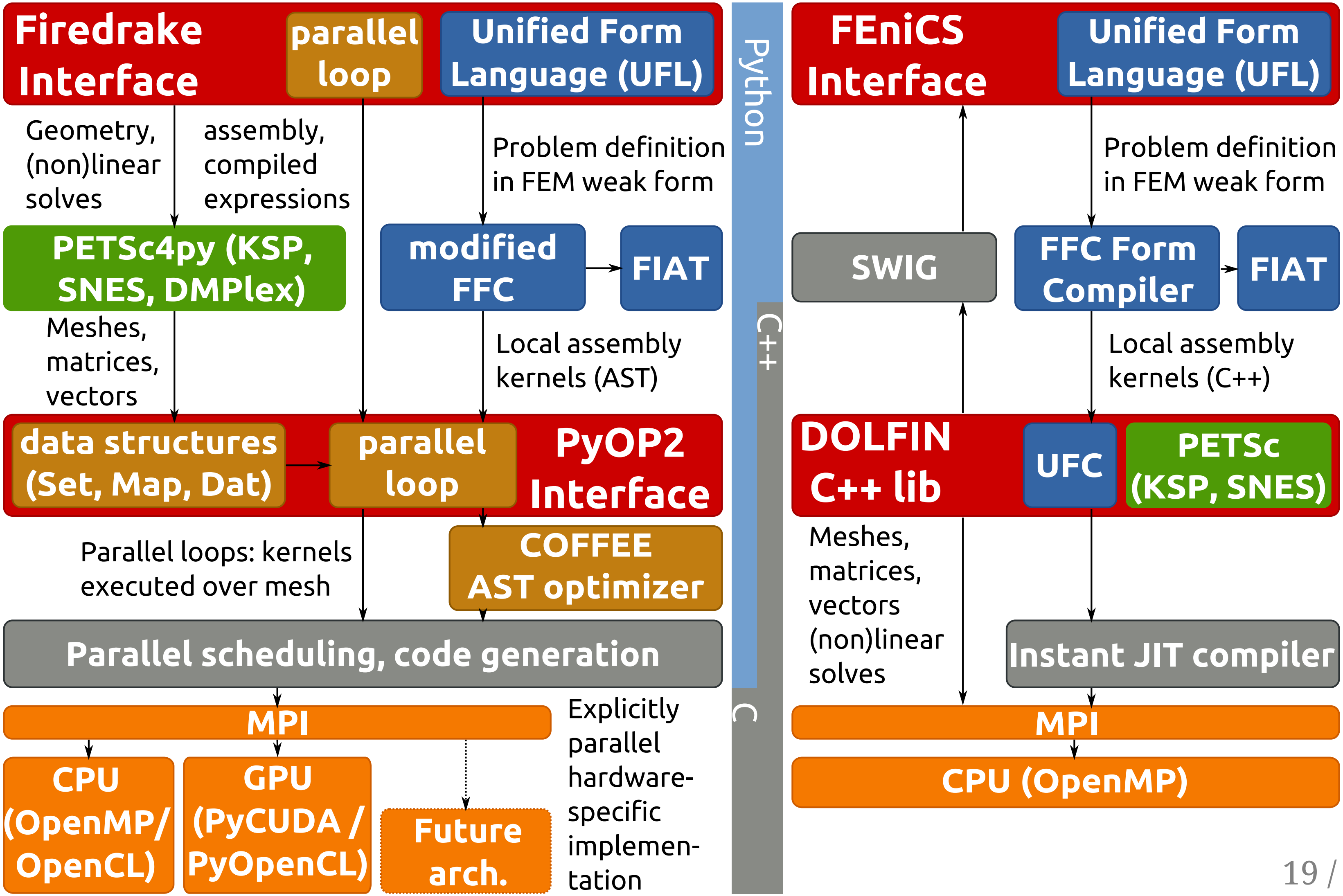
```
op_par_loop(adt_calc,"adt_calc",cells,
            op_arg_dat(p_x,   0,pcell, 2,"double",OP_READ ),
            op_arg_dat(p_x,   1,pcell, 2,"double",OP_READ ),
            op_arg_dat(p_x,   2,pcell, 2,"double",OP_READ ),
            op_arg_dat(p_x,   3,pcell, 2,"double",OP_READ ),
            op_arg_dat(p_q,  -1,OP_ID, 4,"double",OP_READ ),
            op_arg_dat(p_adt,-1,OP_ID, 1,"double",OP_WRITE));
```

`adt_calc` kernel in the PyOP2 Airfoil example application:

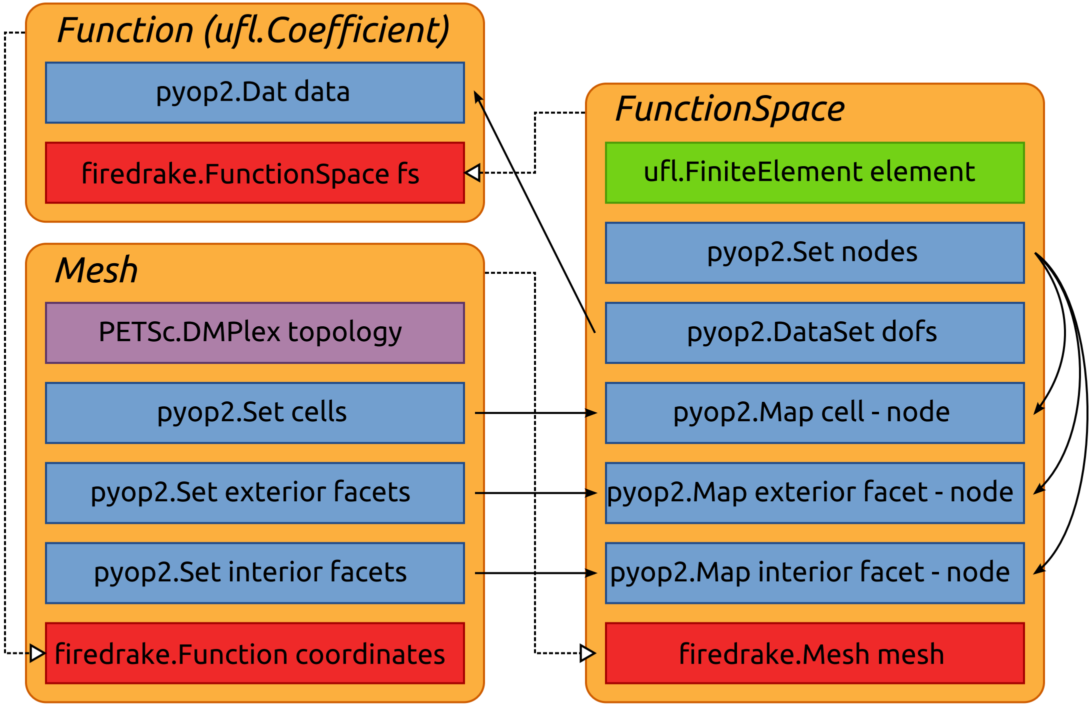
```
op2.par_loop(adt_calc, cells,
             p_x(op2.READ, pcell),
             p_q(op2.READ),
             p_adt(op2.WRITE))
```

Finite-element computations with Firedrake

Firedrake vs. DOLFIN/FEniCS tool chains



Firedrake concepts



Function

Field defined on a set of degrees of freedom (DoFs), data stored as PyOP2 `Dat`

FunctionSpace

Characterized by a family and degree of FE basis functions, defines DOFs for function and relationship to mesh entities

Mesh

Defines abstract topology by sets of entities and maps between them (PyOP2 data structures)

Driving Finite-element Computations in Firedrake

Solving the Helmholtz equation in Python using Firedrake:

$$\int_\Omega \nabla v \cdot \nabla u - \lambda v u \, dV = \int_\Omega v f \, dV$$

```
from firedrake import *

# Read a mesh and define a function space
mesh = Mesh('filename')
V = FunctionSpace(mesh, "Lagrange", 1)

# Define forcing function for right-hand side
f = Expression("- (lmbda + 2*(n**2)*pi**2) * sin(X[0]*pi*n) * sin(X[1]*pi*n)",
               lmbda=1, n=8)

# Set up the Finite-element weak forms
u = TrialFunction(V)
v = TestFunction(V)

lmbda = 1
a = (dot(grad(v), grad(u)) - lmbda * v * u) * dx
L = v * f * dx

# Solve the resulting finite-element equation
p = Function(V)
solve(a == L, p)
```

Behind the scenes of the solve call

- Firedrake always solves nonlinear problems in resdiual form `F(u;v) = 0`
- Transform linear problem into residual form:

```
J = a
F = ufl.action(J, u) - L
```

  - Jacobian known to be `a`
  - **Always** solved in a single Newton (nonlinear) iteration

- Use Newton-like methods from PETSc SNES (optimised C library)
- PETSc SNES requires two callbacks to evaluate residual and Jacobian:
  - implemented as Python functiones (supported by petsc4py)
  - evaluate residual by assembling residual form

```
assemble(F, tensor=F_tensor)
```

  - evaluate Jacobian by assembling Jacobian form

```
assemble(J, tensor=J_tensor, bcs=bcs)
```

  - `assemble` invokes PyOP2 with kernels generated from `F` and `J`

Applying boundary conditions

- Always preserve symmetry of the operator
- Avoid costly search of CSR structure to zero rows/columns
- Zeroing during assembly, but requires boundary DOFs:
  - negative row/column indices for boundary DOFs during addto
  - instructs PETSc to drop entry, leaving 0 in assembled matrix

Applying boundary conditions

- Always preserve symmetry of the operator
- Avoid costly search of CSR structure to zero rows/columns
- Zeroing during assembly, but requires boundary DOFs:
    - negative row/column indices for boundary DOFs during addto
    - instructs PETSc to drop entry, leaving 0 in assembled matrix

Preassembly

```
A = assemble(a)
b = assemble(L)
solve(A, p, b, bcs=bcs)
```

Applying boundary conditions

- Always preserve symmetry of the operator
- Avoid costly search of CSR structure to zero rows/columns
- Zeroing during assembly, but requires boundary DOFs:
    - negative row/column indices for boundary DOFs during addto
    - instructs PETSc to drop entry, leaving 0 in assembled matrix

Preassembly

```
A = assemble(a)  # A unassembled, A.thunk(bcs) not yet called
b = assemble(L)
solve(A, p, b, bcs=bcs)
```

Applying boundary conditions

- Always preserve symmetry of the operator
- Avoid costly search of CSR structure to zero rows/columns
- Zeroing during assembly, but requires boundary DOFs:
  - negative row/column indices for boundary DOFs during addto
  - instructs PETSc to drop entry, leaving 0 in assembled matrix

Preassembly

```
A = assemble(a)  # A unassembled, A.thunk(bcs) not yet called
b = assemble(L)
solve(A, p, b, bcs=bcs)  # A.thunk(bcs) called, A assembled
```

Applying boundary conditions

- Always preserve symmetry of the operator
- Avoid costly search of CSR structure to zero rows/columns
- Zeroing during assembly, but requires boundary DOFs:
    - negative row/column indices for boundary DOFs during addto
    - instructs PETSc to drop entry, leaving 0 in assembled matrix

Preassembly

```
A = assemble(a)  # A unassembled, A.thunk(bcs) not yet called
b = assemble(L)
solve(A, p, b, bcs=bcs)  # A.thunk(bcs) called, A assembled
# ...
solve(A, p, b, bcs=bcs)  # bcs consistent, no need to reassemble
```

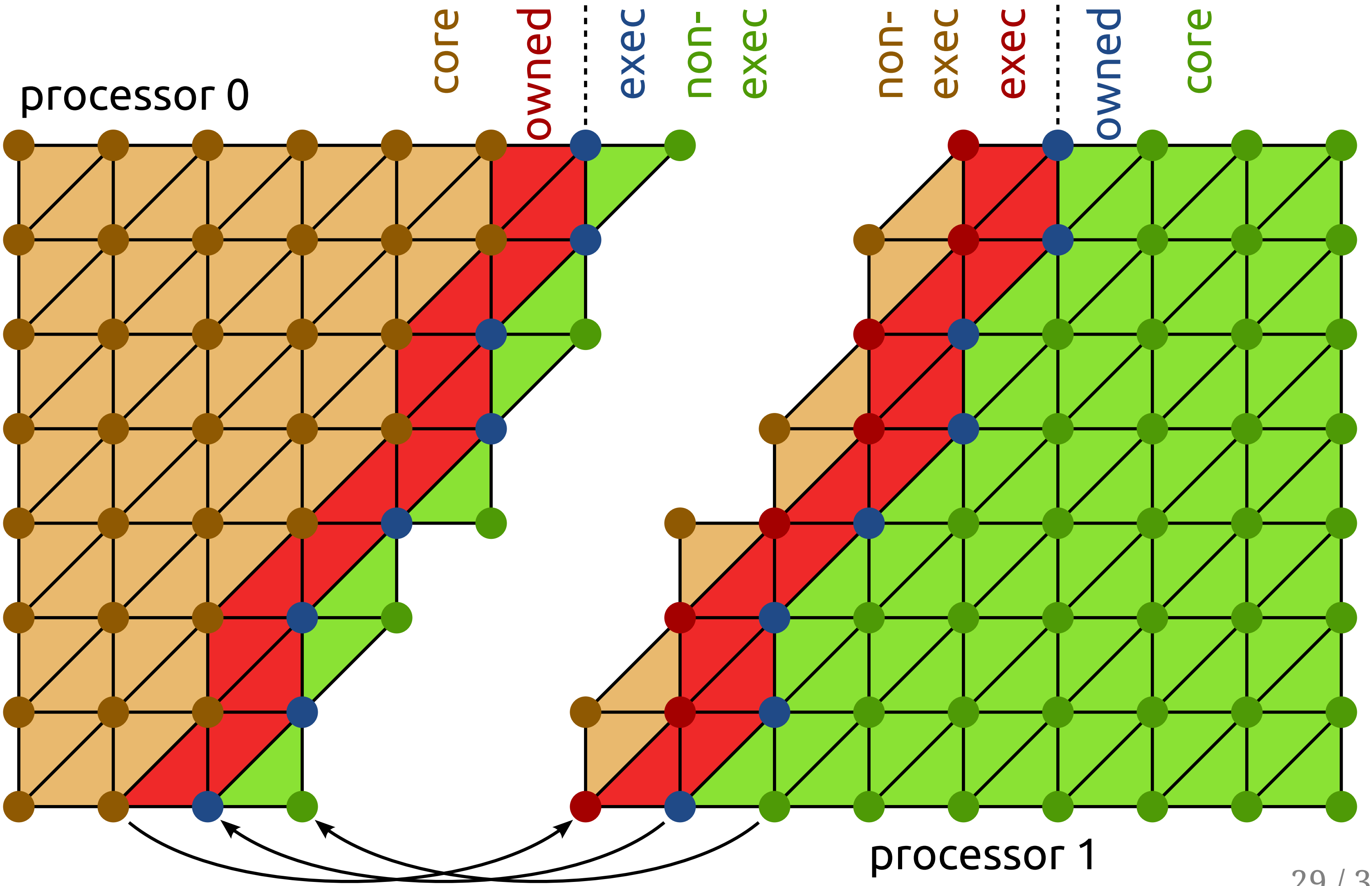
Applying boundary conditions

- Always preserve symmetry of the operator
- Avoid costly search of CSR structure to zero rows/columns
- Zeroing during assembly, but requires boundary DOFs:
  - negative row/column indices for boundary DOFs during addto
  - instructs PETSc to drop entry, leaving 0 in assembled matrix

Preassembly

```
A = assemble(a)  # A unassembled, A.thunk(bcs) not yet called
b = assemble(L)
solve(A, p, b, bcs=bcs)  # A.thunk(bcs) called, A assembled
# ...
solve(A, p, b, bcs=bcs)  # bcs consistent, no need to reassemble
# ...
solve(A, p, b, bcs=bcs2)  # bcs differ, reassemble, call A.thunk(bcs2)
```

Distributed Parallel Computations with MPI



How do Firedrake/PyOP2 achieve good performance?

- No computationally expensive computations (inner loops) in pure Python
- Call optimised libraries where possible (PETSc)
- Expensive library code implemented in Cython (sparsity builder)
- Kernel application over the mesh in natively generated code
- Python is not just glue!

Caching

- Firedrake
  - Assembled operators
  - Function spaces cached on meshes
  - FFC-generated kernel code
- PyOP2
  - `Maps` cached on `Sets`
  - Sparsity patterns
  - JIT-compiled code
- Only data isn't cached (`Function`/`Dat`)

Benchmarks

ARCHER: Cray XC30 with Aries interconnect (Dragonfly topology)

- 2x 12-core Intel Xeon E5-2697 @ 2.70GHz (Ivy Bridge)
- 64GB RAM

Compilers

- GNU Compilers 4.8.2
- Cray MPICH 6.3.1
- Compiler flags: -O3 -mavx

Software

- DOLFIN 30bbd31 (August 22 2014)
- Firedrake c8ed154 (September 25 2014)
- PyOP2 f67fd39 (September 24 2014)

Problem setup

- DOLFIN + Firedrake: RCM mesh reordering enabled
- DOLFIN: quadrature with optimisations enabled
- Firedrake: quadrature with COFFEE loop-invariant code motion, alignment and padding

Explicit Wave Equation

```
from firedrake import *
mesh = Mesh("wave_tank.msh")

V = FunctionSpace(mesh, 'Lagrange', 1)
p = Function(V, name="p")
phi = Function(V, name="phi")

u = TrialFunction(V)
v = TestFunction(V)

p_in = Constant(0.0)
bc = DirichletBC(V, p_in, 1)  # for y=0

T = 10.
dt = 0.001
t = 0

b = assemble(rhs)
dphi = 0.5 * dtc * p
dp = dtc * Ml * b

while t <= T:
    p_in.assign(sin(2*pi*5*t))
    phi -= dphi
    assemble(rhs, tensor=b)
    p += dp
    bc.apply(p)
    phi -= dphi
    t += dt
```

2nd order PDE:

$$\frac{\partial^2 \phi}{\partial t^2} - \nabla^2 \phi = 0$$

Formulation with 1st order time derivatives:

$$\frac{\partial \phi}{\partial t} = -p$$

$$\frac{\partial p}{\partial t} + \nabla^2 \phi = 0$$

Explicit Wave Equation Strong Scaling on UK National Supercomputer ARCHER

Cahn-Hilliard Equation Strong Scaling on UK National Supercomputer ARCHER

Summary and additional features

Summary

- Two-layer abstraction for FEM computation from high-level descriptions
  - Firedrake: a performance-portable finite-element computation framework
    Drive FE computations from a high-level problem specification
  - PyOP2: a high-level interface to unstructured mesh based methods
    Efficiently execute kernels over an unstructured grid in parallel
- Decoupling of Firedrake (FEM) and PyOP2 (parallelisation) layers
- Firedrake concepts implemented with PyOP2/PETSc constructs
- Portability for unstructured mesh applications: FEM, non-FEM or combinations
- Extensible framework beyond FEM computations (e.g. image processing)

Summary and additional features

Summary

- Two-layer abstraction for FEM computation from high-level descriptions
  - Firedrake: a performance-portable finite-element computation framework
    Drive FE computations from a high-level problem specification
  - PyOP2: a high-level interface to unstructured mesh based methods
    Efficiently execute kernels over an unstructured grid in parallel
- Decoupling of Firedrake (FEM) and PyOP2 (parallelisation) layers
- Firedrake concepts implemented with PyOP2/PETSc constructs
- Portability for unstructured mesh applications: FEM, non-FEM or combinations
- Extensible framework beyond FEM computations (e.g. image processing)

Preview: Firedrake features not covered

- Automatic optimization of generated assembly kernels with COFFEE (Fabio)
- Solving PDEs on extruded (semi-structured) meshes (Doru + Andrew)
- Building meshes using PETSc DMPlex (Michael)
- Using fieldsplit preconditioners for mixed problems
- Solving PDEs on immersed manifolds
- …

Thank you!

Contact: Florian Rathgeber, @frathgeber, f.rathgeber@imperial.ac.uk

Resources

- **PyOP2** https://github.com/OP2/PyOP2
  - *PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes* Florian Rathgeber, Graham R. Markall, Lawrence Mitchell, Nicholas Loriant, David A. Ham, Carlo Bertolli, Paul H.J. Kelly, WOLFHPC 2012
  - *Performance-Portable Finite Element Assembly Using PyOP2 and FEniCS* Graham R. Markall, Florian Rathgeber, Lawrence Mitchell, Nicolas Loriant, Carlo Bertolli, David A. Ham, Paul H. J. Kelly , ISC 2013
- **Firedrake** https://github.com/firedrakeproject/firedrake
  - *COFFEE: an Optimizing Compiler for Finite Element Local Assembly* Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, Paul H. J. Kelly, submitted
- **UFL** https://bitbucket.org/mapdes/ufl
- **FFC** https://bitbucket.org/mapdes/ffc

This talk is available at http://kynan.github.io/FiredrakeSeminar2014 (source)