Firedrake: a High-level, Portable Finite Element Computation Framework

Florian Rathgeber\(^1\), Lawrence Mitchell\(^1\), David Ham\(^1,2\), Andrew McRae\(^2\), Fabio Luporini\(^1\), Gheorghe-teodor Bercea\(^1\), Paul Kelly\(^1\)

Slides: http://kynan.github.io/m2op-2014

\(^1\) Department of Computing, Imperial College London \(^2\) Department of Mathematics, Imperial College London
Computational Science is hard

Unless you break it down with the right abstractions

Many-core hardware has brought a paradigm shift to CSE, scientific software needs to keep up
Computational Science is hard

Unless you break it down with the right abstractions

Many-core hardware has brought a paradigm shift to CSE, scientific software needs to keep up

The Solution

High-level structure

- Goal: producing high level interfaces to numerical computing
- **PyOP2**: a high-level interface to unstructured mesh based computations
  \textit{Efficiently execute kernels over an unstructured grid in parallel}
- **Firedrake**: a performance-portable finite-element computation framework
  \textit{Drive FE computations from a high-level problem specification}

Low-level operations

- Separating the low-level implementation from the high-level problem specification
- Generate platform-specific implementations from a common source instead of hand-coding them
- Runtime code generation and JIT compilation open space for compiler-driven optimizations and performance portability
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 the final result.

**PyOP2**

A domain-specific language embedded in Python for parallel computations on unstructured meshes or graphs.

**Unstructured mesh**

**OP2 Sets:**
- nodes = [0, 1, 2, 3, 4, 5, 6, 7, 8]
- elements = [0, 1, 2, 3, 4, 5, 6, 7, 8]

**OP2 Map elements-nodes:**
- elem_nodes = [[0, 1, 2], [1, 3, 2], ...]

**OP2 Dat on nodes:**
- coords = [..., [.5,.5], [.5,-.25], [1,.25], ...]
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 the final result.

**PyOP2**

A domain-specific language embedded in Python for parallel computations on unstructured meshes or graphs.

**Unstructured mesh**

**OP2 Sets:**
- nodes = [0, 1, 2, 3, 4, 5, 6, 7, 8]
- elements = [0, 1, 2, 3, 4, 5, 6, 7, 8]

**OP2 Map elements-nodes:**
- elem_nodes = [[0, 1, 2], [1, 3, 2], ...]

**OP2 Dat on nodes:**
- coords = [..., [.5,.5], [.5,-.25], [1,.25], ...]

**PyOP2 Data Model**

**Mesh topology**

- Sets – cells, vertices, etc
- Maps – connectivity between entities in different sets

**Data**

- Dats – Defined on sets (hold pressure, temperature, etc)

**Kernels**

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

**Linear algebra**

- Sparsities defined by mappings
- Matrix data on sparsities
- Kernels compute a local matrix – PyOP2 handles global assembly
PyOP2
Kernels and Parallel Loops

Parallel loop syntax

\[
\text{op2.par_loop}(\text{kernel}, \text{iteration_set}, \text{kernel_arg1(access_mode, mapping[index])}, ..., \text{kernel_argN(access_mode, mapping[index])})
\]

PyOP2 program for computing the midpoint of a triangle

```python
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(""
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;
}"")

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

Future work: define kernels as Python functions
PyOP2 Architecture

- Parallel scheduling: partitioning, staging and coloring
- Runtime code generation and JIT compilation

User code

PyOP2 core

Code generation

just-in-time (JIT) compile kernels + marshaling code

PyOP2 Lib & Runtime Core
colouring, parallel scheduling

Lin. algebra
PETSc/Cusp

Backends

MPI

CPU seq.
CPU OpenMP
OpenCL
CUDA

PyOpenCL (JIT)
PyCUDA (JIT)
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 Partitioning, Staging & Coloring

Key optimizations performed by PyOP2 runtime core

- *Partitioning* for on-chip memory (shared memory / cache)
- *Coloring* to avoid data races on updates to the same memory location

Example
Parallel computation executing a kernel over the edges of the mesh:

```python
# Sets of nodes and edges
nodes = op2.Set(N)  # N = number of nodes
edges = op2.Set(M)  # M = number of edges

# Mapping from edges to nodes
edge_to_node_map = op2.Map(edges, nodes, 2, ...)

# Data defined on nodes
u = op2.Dat(nodes, ...)

# Kernel executing over set of edges, computing on nodal data
op2.par_loop(kernel, edges,
             u(edge_to_node_map[:,], op2.INC))
```
edges

shared / staging memory

vertices
- Mesh partitioned among processors
- Computations on boundaries require up-to-date halo data
- Enforce constraint on local mesh numbering for efficient comp-comm overlap
- Entities that do not touch the boundary can be computed while halo data exchange is in flight
- Halo exchange is automatic and happens only if halo is "dirty"

Local mesh entities partitioned into four sections

- **Core**: Entities owned by this processor which can be processed without accessing halo data.
- **Owned**: Entities owned by this processor which access halo data when processed.
- **Exec halo**: Off-processor entities redundantly executed over because they touch owned entities.
- **Non-exec halo**: Off-processor entities which are not processed, but read when computing the exec halo.
Finite-element computations with Firedrake
The weak form of the Helmholtz equation:

\[ \int_{\Omega} \nabla v \cdot \nabla u - \lambda v u \, dV = \int_{\Omega} v f \, dV \]

\[ Ax = b \]
The weak form of the Helmholtz equation

\[ \int_{\Omega} \nabla v \cdot \nabla u - \lambda v u \, dV = \int_{\Omega} v f \, dV \]

And its (almost) literal translation to Python with UFL

UFL: embedded domain-specific language (eDSL) for weak forms of partial differential equations (PDEs)

```python
e = FiniteElement('CG', 'triangle', 1)
v = TestFunction(e)
u = TrialFunction(e)
f =Coefficient(e)
lmbda = 1
a = (dot(grad(v), grad(u)) - lmbda * v * u) * dx
L = v * f * dx
```

UFL is the Unified Form Language from the FEniCS project.
Helmholtz
local
assembly
kernel
generated by
FFC

Helmholtz equation
\[ \int_\Omega \nabla v \cdot \nabla u - \lambda vu \, dV = \int_\Omega vf \, dV \]

UFL expression
\[
a = (\text{dot} (\text{grad} (v), \text{grad} (u)) - \text{lmbda} * v * u) * \text{dx}
\]

Generated C code

```c
void kernel(double A[1][1], double *x[2],
            int j, int k) {
    // FE0 - Shape functions
    // Dij - Shape function derivatives
    // Kij - Jacobian inverse / determinant
    // W3 - Quadrature weights
    // det - Jacobian determinant
    for (unsigned int ip = 0; ip < 3; ip++) {
        A[0][0] += (FE0[ip][j] * FE0[ip][k] * (-1.0)
                  + (((K00 * D10[ip][j] + K10 * D01[ip][j]))
                    *((K00 * D10[ip][k] + K10 * D01[ip][k]))
                    + (((K01 * D10[ip][j] + K11 * D01[ip][j]))
                        *((K01 * D10[ip][k] + K11 * D01[ip][k])))*W3[ip]*det;
    }
}
```

The FEniCS Form Compiler FFC compiles UFL forms to low-level code.
The Firedrake/PyOP2 tool chain

**Firedrake Interface**
- Geometry, (non)linear solves
- PETSc[4py] (KSP, SNES, DMPlex)
- Meshes, matrices, vectors

**Unified Form Language (UFL)**
- Problem definition in FEM weak form
- FFC Form Compiler
- FIAT
- Local assembly kernels, data dependencies

**PyOP2 Interface**
- Parallel loops: kernels executed over mesh
- Parallel scheduling, code generation
- MPI
- CPU (OpenMP/OpenCL)
- GPU (PyCUDA/PyOpenCL)
- Future arch.

**Firedrake**
- Performance-portable Finite-element computation framework

**PyOP2**
- Parallel unstructured mesh computation framework

Explicitly parallel hardware-specific implementation
Two-layered abstraction: Separation of concerns

**Expert for each layer**

- **Domain specialist:** mathematical model using FEM
- **Numerical analyst:** generation of FEM kernels
- **Domain specialist:** mathematical model on unstructured grid
- **Parallel programming expert:** hardware architectures, optimization

---

**Firedrake Interface**

- Geometry, (non)linear solves
- PETSc[4py] (KSP, SNES, DMPlex)
- Meshes, matrices, vectors

**Unified Form Language (UFL)**

- Problem definition in FEM weak form
- FFC Form Compiler
- FIAT
- Local assembly kernels, data dependencies

**PyOP2 Interface**

- Geometry, (non)linear solves
- parallel loops for assembly
- PETSc[4py] (KSP, SNES, DMPlex)

**Parallel scheduling, code generation**

- Explicitly parallel hardware-specific implementation
- Parallel loops: kernels executed over mesh

**MPI**

- CPU (OpenMP/OpenCL)
- GPU (PyCUDA/PyOpenCL)
- Future arch.
Firedrake architecture

- High-level Python interface (mostly) compatible to FEniCS' DOLFIN
- Purely a system for reasoning about variational forms
- Unified form language (UFL) to describe weak forms of PDEs
- FEniCS Form Compiler (FFC) translates forms into assembly kernels
- PyOP2 as the parallel execution layer for assembly kernels
  - responsible for storage, transfer and communication of data
  - backend independent
  - performance portable
  - no code changes required when switching backend
- PETSc used for
  - meshes (DMplex)
  - nonlinear solves (SNES)
  - linear solves (KSP, PC)
- *No parallel code*: parallelism handled by PyOP2 + PETSc
Firedrake architecture

- High-level Python interface (mostly) compatible to FEniCS' DOLFIN
- Purely a system for reasoning about variational forms
- Unified form language (UFL) to describe weak forms of PDEs
- FEniCS Form Compiler (FFC) translates forms into assembly kernels
- PyOP2 as the parallel execution layer for assembly kernels
  - responsible for storage, transfer and communication of data
  - backend independent
  - performance portable
  - no code changes required when switching backend
- PETSc used for
  - meshes (DMPllex)
  - nonlinear solves (SNES)
  - linear solves (KSP, PC)
- *No parallel code:* parallelism handled by PyOP2 + PETSc

Firedrake concepts

- **Function:** field defined on a set of degrees of freedom (DoFs), data stored as PyOP2 Dat
- **FunctionSpace:** Characterized by a family and and degree of FE basis functions, defined 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:

\[
\int_{\Omega} \nabla v \cdot \nabla u - \lambda vu \, dV = \int_{\Omega} vf \, dV
\]

```python
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)
```
Finite element assembly and solve in Firedrake

- Unified interface: Firedrake always solves nonlinear problems in residual form $F(u; v) = 0$ using Newton-like methods (provided by PETSc SNES).
- SNES requires two callbacks to evaluate residual and Jacobian:
  - evaluate residual: `assemble(F, tensor=F_tensor)`
  - evaluate Jacobian: `assemble(J, tensor=J_tensor, bcs=bc)`
- If Jacobian not provided by the user, Firedrake uses automatic differentiation:
  
  $J = ufl.derivative(F, u)$

- Transform linear problem with bilinear form $a$, linear form $L$ into residual form:
  
  $J = a$
  $F = ufl.action(J, u) - L$

  Jacobian known to be $a$, **always** solved in a single Newton (nonlinear) iteration
**Finite element assembly and solve in Firedrake**

- Unified interface: Firedrake always solves nonlinear problems in residual form $F(u; v) = 0$ using Newton-like methods (provided by PETSc SNES)
- SNES requires two callbacks to evaluate residual and Jacobian:
  - evaluate residual: `assemble(F, tensor=F_tensor)`
  - evaluate Jacobian: `assemble(J, tensor=J_tensor, bcs=bc)`
- If Jacobian not provided by the user, Firedrake uses automatic differentiation:

  ```python
  J = ufl.derivative(F, u)
  ```

- Transform linear problem with bilinear form $a$, linear form $L$ into residual form:

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

  Jacobian known to be $a$, **always** solved in a single Newton (nonlinear) iteration

```python
def solve(problem, solution, bcs=None, J=None, solver_parameters=None)
```

1. If problem is linear, transform into residual form
2. If no Jacobian provided, compute Jacobian by automatic differentiation
3. Set up PETSc SNES solver (parameters user configurable)
4. Assign residual and Jacobian forms for SNES callbacks
5. Solve nonlinear problem. For each nonlinear iteration:
   a) assemble Jacobian matrix
   b) assemble residual vector
   c) solve linear system using PETSc KSP
Assembling linear and bilinear forms: the assemble call

- Unified interface: assemble a UFL form into a global tensor
  - bilinear form: assemble matrix (optionally with boundary conditions)
  - linear form: assemble vector (optionally with boundary conditions)
  - functional: assemble scalar value
- UFL form may contain one or more integrals over cells, interior and exterior facets
- Each integral: local assembly kernel performing numerical quadrature
- Kernels generated by FFC and executed as PyOP2 parallel loops
  - Firedrake builds PyOP2 parallel loop call, using FFC-generated kernel
  - iterate over cells (for cell integrals) or facets (interior/exterior facet integrals)
  - output tensor depends on rank of the form (PyOP2 Mat, Dat or Global)
  - input arguments: coordinate field and any coefficients present in the form
Assembling linear and bilinear forms: the assemble call

- Unified interface: assemble a UFL form into a global tensor
  - bilinear form: assemble matrix (optionally with boundary conditions)
  - linear form: assemble vector (optionally with boundary conditions)
  - functional: assemble scalar value
- UFL form may contain one or more integrals over cells, interior and exterior facets
- Each integral: local assembly kernel performing numerical quadrature
- Kernels generated by FFC and executed as PyOP2 parallel loops
  - Firedrake builds PyOP2 parallel loop call, using FFC-generated kernel
  - iterate over cells (for cell integrals) or facets (interior/exterior facet integrals)
  - output tensor depends on rank of the form (PyOP2 Mat, Dat or Global)
  - input arguments: coordinate field and any coefficients present in the form

Boundary conditions

- Applied in a way that preserves symmetry of the operator. For each boundary node:
  1. Set matrix row and column to 0
  2. Set matrix diagonal to 1
  3. Modify the right-hand side vector with boundary value
- Leverage PETSc to avoid costly zeroing of CSR columns
  - on assembly, set row/column indices of boundary values to negative values
  - instruct PETSc to drop contributions, leaving a 0 in the assembled matrix
Summary and additional features

Summary

- Two-layer abstraction for FEM computation from high-level descriptions
- PyOP2: a high-level interface to unstructured mesh based methods
  
  *Efficiently execute kernels over an unstructured grid in parallel*

- Firedrake: a performance-portable finite-element computation framework
  
  *Drive FE computations from a high-level problem specification*

- Decoupling of Firedrake (FEM) and PyOP2 (parallelisation) layers
- Target-specific runtime code generation and JIT compilation
- Performance portability for unstructured mesh applications: FEM, non-FEM or combinations
- Extensible framework beyond FEM computations (e.g. image processing)

Firedrake features not covered

- Building meshes using PETSc DMPlex
- Communication-computation overlap when running MPI-parallel
- Using fieldsplit preconditioners for mixed problems
- Solving PDEs on extruded (semi-structured) meshes
- Solving PDEs on immersed manifolds
- Automatic optimization of generated assembly kernels with COFFEE
- ...
Thank you!

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

Resources

- **PyOP2** https://github.com/OP2/PyOP2
- **Firedrake** https://github.com/firedrakeproject/firedrake
- **UFL** https://bitbucket.org/mapdes/ufl
- **FFC** https://bitbucket.org/mapdes/ffc

This talk is available at http://kynan.github.io/m2op-2014 (source)

Slides created with remark