Efficient multigrid solvers for mixed finite element discretisations in NWP models

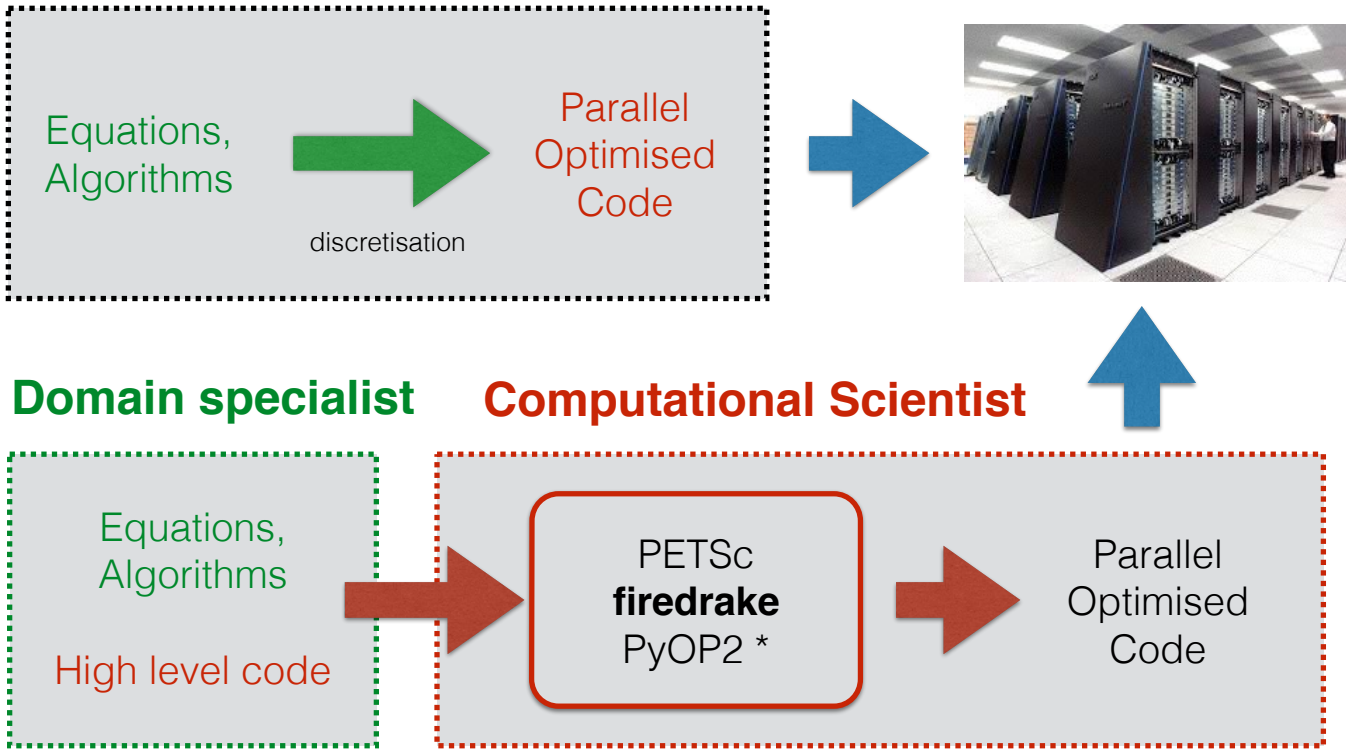
Colin Cotter[†], David Ham[†], Lawrence Mitchell[†],
Eike Hermann Müller[*], Robert Scheichl[*]

[*]University of Bath, [†]Imperial College London

16th ECMWF Workshop on High Performance Computing in Meteorology Oct 2014

Motivation

Separation of concerns and high-level abstractions



Use case: complex **iterative solver** for pressure correction eqn.

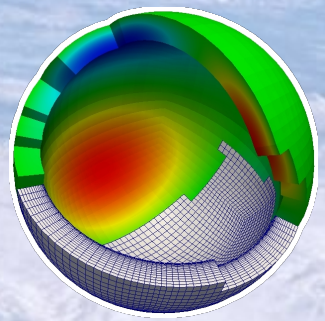
* All credit for developing firedrake/PyOP2 goes to David Ham and his group at Imperial, I'm giving this talk as a "user"

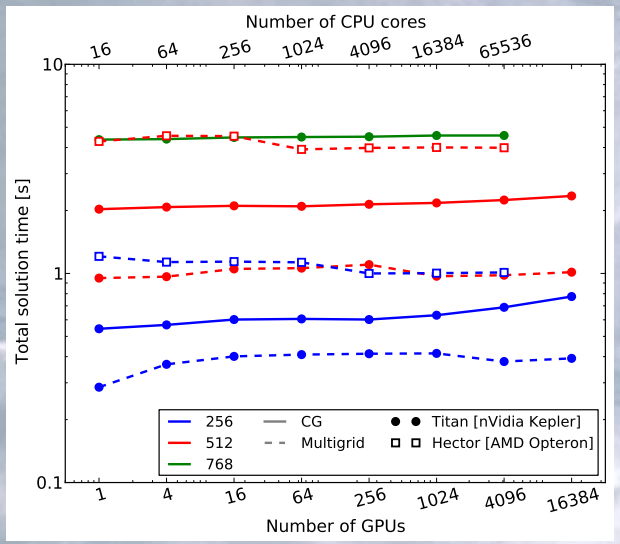
Motivation

Computational challenge in numerical forecasts:

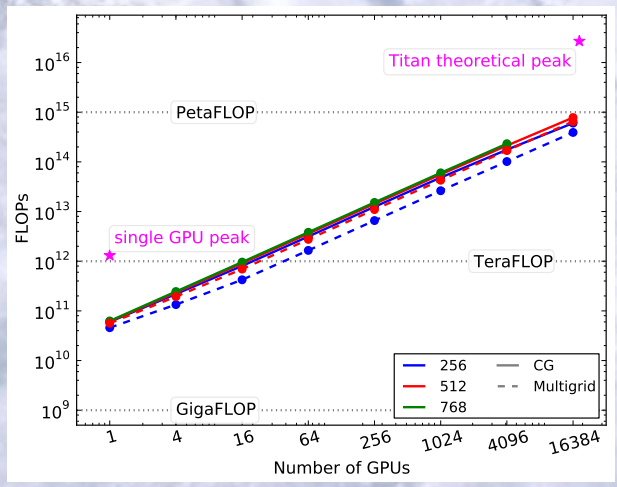
Solver for elliptic pressure correction equation

(algorithmic + parallel) scalability & performance





Weak scaling, total solution time



Absolute performance

- $0.5 \cdot 10^{12}$ dofs on 16384 GPUs of TITAN, 0.78 PFLOPs, 20%-50% BW
- Finite volume discretisation, simpler geometry

Motivation

Challenges

- Finite element discretisation (Met Office next generation dyn. core), unstructured grids ⇒ more complicated
- Duplicate effort for CPU, GPU (Xeon Phi,...) implementation & optimisation, reinvent the wheel?
- Mix two issues:
  - algorithmic (domain specialist/numerical analyst)
  - parallelisation & low level optimisation (computational scientist)

Goal

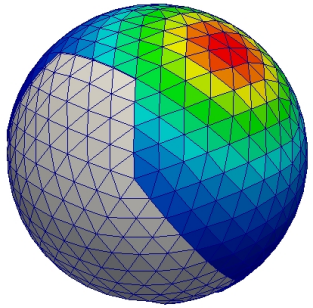
- **Separation of concerns** (algorithmic ↔ computational)
- ⇒ **rapid algorithm development** and testing at high abstraction level
- **Performance portability**
- **Reuse** existing, tested and optimised tools and libraries

⇒ **Choice of correct abstraction(s)**

This talk

Iterative solver for **Helmholtz equation**:

$$\phi + \omega\nabla\left(\phi^* \boldsymbol{u}\right) = r_\phi$$
$$-\boldsymbol{u} - \omega\nabla\phi = \boldsymbol{r_u}$$



- Mixed finite element discretisation, icosahedral grid
- Matrix-free multigrid preconditioner for pressure correction

Performance portable **firedrake/PyOP2 toolchain**

[Rathgeber et al., (2012 & 2014)]

Python ⇒ automatic generation of optimised C code

Collaboration between

- Numerical algorithm developers (Colin, Rob, Eike)
- Computational scientists, Library developers (David, Lawrence, {PETSc,PyOP2,firedrake} teams)

Shallow water equations

Model system: shallow water equations

$$\frac{\partial \phi}{\partial t} + \nabla (\phi \boldsymbol{u}) = 0$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla) \boldsymbol{u} + g \nabla \phi = 0$$

Implicit timestepping
$\Rightarrow$ **linear system** for velocity $\boldsymbol{u}$ and height perturbation ϕ

$$\phi + \omega \nabla (\phi^* \boldsymbol{u}) = r_\phi$$

$$-\boldsymbol{u} - \omega \nabla \phi = \boldsymbol{r_u}$$

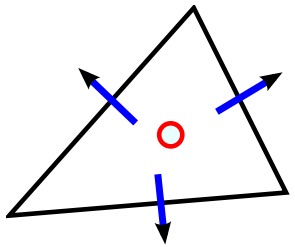
- reference state $\phi^* \equiv 1$
- $\omega = c_h \Delta t = \sqrt{g \phi^*} \Delta t$

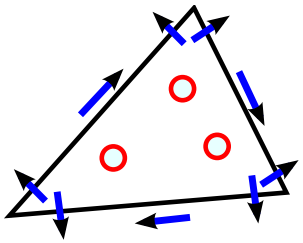
Mixed system

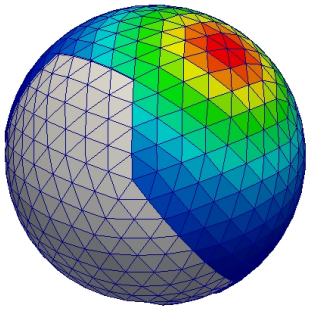
Finite element discretisation: $\phi \in \mathbb{V}_2$, $\boldsymbol{u} \in \mathbb{V}_1$

$\Rightarrow$ **Matrix system** for dof-vectors $\boldsymbol{\Phi} \in \mathbb{R}^{n_{\mathbb{V}_2}}$, $\boldsymbol{U} \in \mathbb{R}^{n_{\mathbb{V}_1}}$

$$A \begin{pmatrix} \boldsymbol{\Phi} \\ \boldsymbol{U} \end{pmatrix} = \begin{pmatrix} M_\phi & \omega B \\ \omega B^T & -M_u \end{pmatrix} \begin{pmatrix} \boldsymbol{\Phi} \\ \boldsymbol{U} \end{pmatrix} = \begin{pmatrix} \boldsymbol{R}_\phi \\ \boldsymbol{R}_u \end{pmatrix}$$







Mixed finite elements

- $DG_0 + RT_1$ (lowest order)
- $DG_1 + BDFM_1$ (higher order)

$$(M_\phi)_{ij} \equiv \int_\Omega \beta_i(x)\beta_j(x)\, dx, \quad (M_u)_{ef} \equiv \int_\Omega \boldsymbol{v}_e(x) \cdot \boldsymbol{v}_f(x)\, dx \qquad \text{(mass matrix)}$$

$$B_{ie} \equiv \int_\Omega \nabla \boldsymbol{v}_e(x)\beta_i(x)\, dx \qquad \text{(derivative matrix)}$$

Iterative solver

Iterative solver

1. **Outer iteration** (mixed system):
   GMRES, Richardson, BiCGStab, ...

   Preconditioner:

$$P \equiv \begin{pmatrix} M_\phi & \omega B \\ \omega B^T & -M_u^* \end{pmatrix}$$

lumped velocity mass matrix

$$P^{-1} = \begin{pmatrix} \mathbb{1} & 0 \\ (M_u^*)^{-1}\omega B^T & \mathbb{1} \end{pmatrix} \begin{pmatrix} H^{-1} & 0 \\ 0 & -(M_u^*)^{-1} \end{pmatrix} \begin{pmatrix} \mathbb{1} & \omega B (M_u^*)^{-1} \\ 0 & \mathbb{1} \end{pmatrix}$$

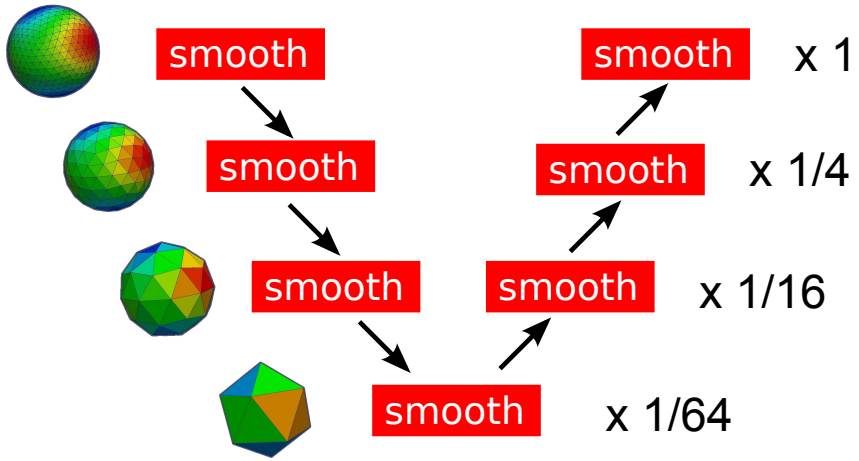
   Elliptic positive definite **Helmholtz operator**

$$H = M_\phi + \omega^2 B \left(M_u^*\right)^{-1} B^T$$

2. **Inner iteration** (pressure system $H\phi' = R'_\phi$):
   CG with geometric multigrid preconditioner

Multigrid

Multigrid V-cycle



Computational bottleneck

Smoother and operator application

$$\phi' \mapsto \phi' + \rho_{\text{relax}} D_H^{-1} \left(R'_\phi - H\phi' \right) \qquad H = M_\phi + \omega^2 B \left(M_u^* \right)^{-1} B^T$$

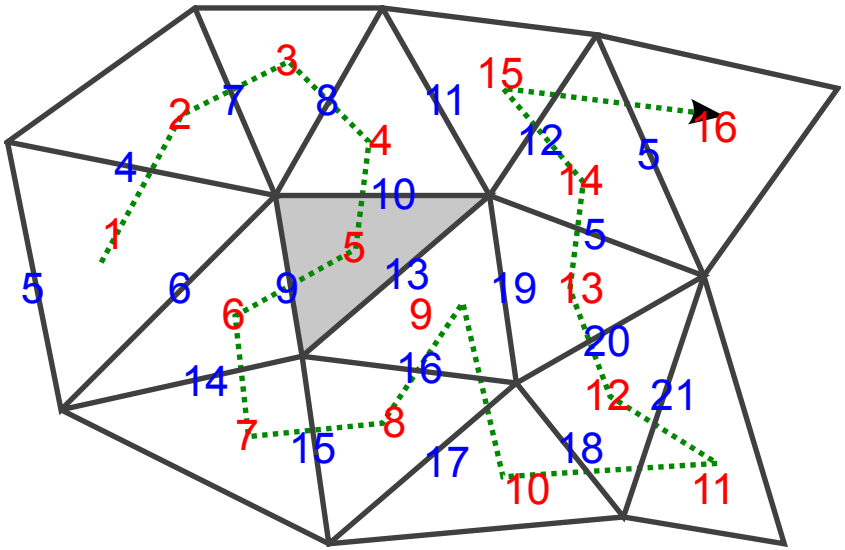
Intrinsic length scale

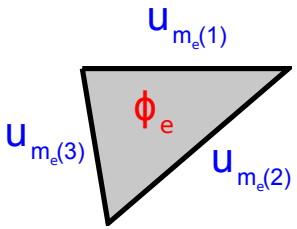
$$\Lambda/\Delta x = c_s \Delta t/\Delta x \sim \nu_{\text{CFL}} = \mathcal{O}(10) \text{ grid spacings} \quad (\forall \text{ resolutions } \Delta x)$$

$\Rightarrow$ small number of multigrid levels/coarse smoother sufficient

Grid iteration in FEM codes

Computational bottlenecks in finite element codes





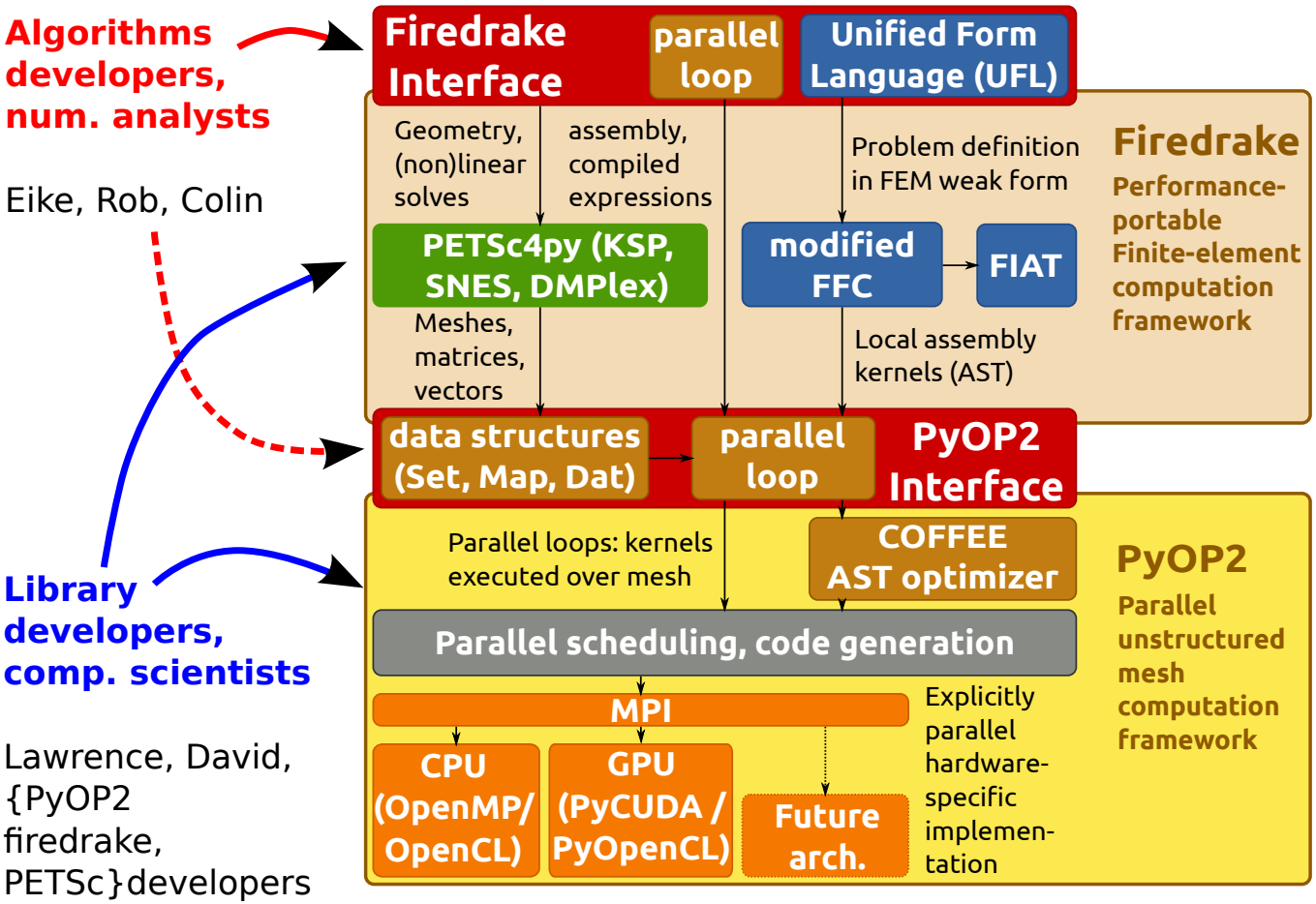
Example: Weak divergence

$\phi = \nabla \cdot \boldsymbol{u} \rightarrow \boldsymbol{\Phi} = B\boldsymbol{U}$

$$\phi_e \mapsto \phi_e + b_1^{(e)} u_{m_e(1)} + b_2^{(e)} u_{m_e(2)} + b_3^{(e)} u_{m_e(3)}$$

- Loop over topological entities (e.g. cells). In each cell
  - Access local dofs (e.g. pressure and velocity)
  - Execute local kernel
  - Write data back
- Manage halo exchanges, thread parallelism

Firedrake/PyOP2 framework



[Florian Rathgeber, PhD thesis, Imperical College (2014)]

UFL

Unified Form Language [Alnæs et al. (2014)]
Example: Bilinear form

$$a(u,v) = \int_\Omega u(x)v(x)\,dx$$

Python code

```
from firedrake import *
mesh = UnitSquareMesh(8,8)
V = FunctionSpace(mesh,'CG',1)
u = Function(V)
v = Function(V)
a_uv = assemble(u*v*dx)
```

Generated code

C-Code generated by Firedrake/PyOP2

Kernel (below)
execution over grid (right)

OpenMP backend

trunk — vim — 80×25

```
1  static inline void form_cell_integral_0_otherwise (
2    double A[1], double **vertex_coordinates,
3    double **w0, double **w1 , double **w2) {
4    double J[4];
5    compute_jacobian_triangle_2d(J, vertex_coordinates);
6    double K[4];
7    double detJ;
8    compute_jacobian_inverse_triangle_2d(K, detJ, J);
9    const double det = fabs(detJ);
10   static const double W3[3]  = {0.166666666666667, 0.166666666666667,
11                                 0.166666666666667};
12   static const double FE0[3][3]  = {
13   {0.666666666666667, 0.166666666666667, 0.166666666666667},
14   {0.166666666666667, 0.166666666666667, 0.666666666666667},
15   {0.166666666666667, 0.666666666666667, 0.166666666666667}};
16   #pragma pyop2 integration
17   for (int ip  = 0; ip < 3; ++ip) {
18     double F0  = 0.0; double F1  = 0.0;
19     for (int r  = 0; r < 3; ++r) {
20       F0 += (w1[r][0] * FE0[ip][r]);
21       F1 += (w2[r][0] * FE0[ip][r]);
22     }
23     A[0] += (det * W3[ip] * F1 * F0);
24   }
```

trunk — vim — 80×46

```
1  void wrap_form_cell_integral_0_otherwise(
2    int boffset, int nblocks, int *blkmap, int *offset, int *nelems,
3    double *arg0_0,
4    double *arg1_0, int *arg1_0_map0_0, double *arg2_0, int *arg2_0_map0_0,
5    double *arg3_0, int *arg3_0_map0_0, double *arg4_0, int *arg4_0_map0_0) {
6    #pragma omp parallel shared(boffset, nblocks, nelems, blkmap)
7    {
8      int tid = omp_get_thread_num();
9      double arg0_0_l0[1][1];
10     for ( int i = 0; i < 1; i++ ) arg0_0_l0[0][i] = (double)0;
11     double *arg1_0_vec[6]; double *arg2_0_vec[6];
12     double *arg3_0_vec[3]; double *arg4_0_vec[3];
13     #pragma omp for schedule(static)
14     for ( int __b = boffset; __b < boffset + nblocks; __b++ ) {
15       int bid = blkmap[__b]; int nelem = nelems[bid];
16       int efirst = offset[bid];
17       for (int n = efirst; n < efirst+ nelem; n++ ) {
18         int i = n;
19         arg1_0_vec[0] = arg1_0 + (arg1_0_map0_0[i * 3 + 0])* 2;
20         arg1_0_vec[1] = arg1_0 + (arg1_0_map0_0[i * 3 + 1])* 2;
21         arg1_0_vec[2] = arg1_0 + (arg1_0_map0_0[i * 3 + 2])* 2;
22         arg1_0_vec[3] = arg1_0 + (arg1_0_map0_0[i * 3 + 0])* 2 + 1;
23         arg1_0_vec[4] = arg1_0 + (arg1_0_map0_0[i * 3 + 1])* 2 + 1;
24         arg1_0_vec[5] = arg1_0 + (arg1_0_map0_0[i * 3 + 2])* 2 + 1;
25         arg2_0_vec[0] = arg2_0 + (arg2_0_map0_0[i * 3 + 0])* 2;
26         arg2_0_vec[1] = arg2_0 + (arg2_0_map0_0[i * 3 + 1])* 2;
27         arg2_0_vec[2] = arg2_0 + (arg2_0_map0_0[i * 3 + 2])* 2;
28         arg2_0_vec[3] = arg2_0 + (arg2_0_map0_0[i * 3 + 0])* 2 + 1;
29         arg2_0_vec[4] = arg2_0 + (arg2_0_map0_0[i * 3 + 1])* 2 + 1;
30         arg2_0_vec[5] = arg2_0 + (arg2_0_map0_0[i * 3 + 2])* 2 + 1;
31         arg3_0_vec[0] = arg3_0 + (arg3_0_map0_0[i * 3 + 0])* 1;
32         arg3_0_vec[1] = arg3_0 + (arg3_0_map0_0[i * 3 + 1])* 1;
33         arg3_0_vec[2] = arg3_0 + (arg3_0_map0_0[i * 3 + 2])* 1;
34         arg4_0_vec[0] = arg4_0 + (arg4_0_map0_0[i * 3 + 0])* 1;
35         arg4_0_vec[1] = arg4_0 + (arg4_0_map0_0[i * 3 + 1])* 1;
36         arg4_0_vec[2] = arg4_0 + (arg4_0_map0_0[i * 3 + 2])* 1;
37         form_cell_integral_0_otherwise(arg0_0_l0[0],
38                                        arg1_0_vec, arg2_0_vec,
39                                        arg3_0_vec, arg4_0_vec);
40       }
41     }
42     #pragma omp critical
43     for ( int i = 0; i < 1; i++ ) arg0_0[i] += arg0_0_l0[0][i];
44   }
45 }
-- INSERT --
```

Putting it all together

≈ 1,600 lines of highly modular python code (incl. comments)

1. Use **UFL** wherever possible

$$\int \phi(\nabla\cdot\boldsymbol{u})\, dx \mapsto \texttt{assemble(phi*div(u)*dx)} \qquad \text{(weak divergence operator)}$$

2. Use **PETSc** for iterative solvers

   provide operators/preconditioners as callbacks

3. Escape hatch: Write **direct PyOP2 parallel loops**

   lumped mass matrix division: $\boldsymbol{U} \mapsto (M_u^*)^{-1}\,\boldsymbol{U}$

```
kernel_code = '''void lumped_mass_divide(double **m,double **U) {
  for (int i=0; i<4; ++i) U[i][0] /= m[0][i];
}'''
kernel = op2.Kernel(kernel_code,"lumped_mass_divide")

  op2.par_loop(kernel,facetset,
               m.dat(op2.READ,self.facet2dof_map_facets),
               u.dat(op2.RW,self.facet2dof_map_BDFM1))
```

Helmholtz operator

Helmholtz operator $H\phi = M_\phi \phi + \omega^2 B (M_u^*)^{-1} B^T \phi$

Operator application

```
class Operator(object):
  [...]
  def apply(self,phi):
    psi = TestFunction(V_pressure)
    w = TestFunction(V_velocity)
    B_phi = assemble(div(w)*phi*dx)
    self.velocity_mass_matrix.divide(B_phi)
    BT_B_phi = assemble(psi*div(B_phi)*dx)
    M_phi = assemble(psi*phi*dx)
    return assemble(M_phi + self.omega**2*BT_B_phi)
```

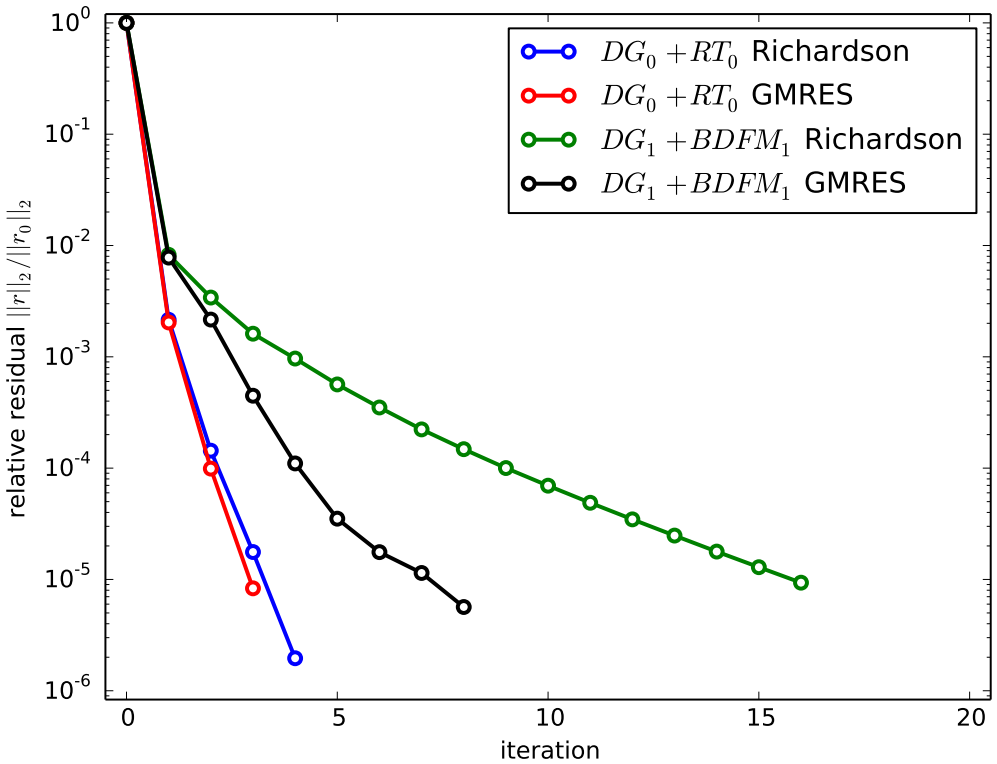
Interface with PETSc

```
petsc_op = PETSc.Mat().create()
petsc_op.setPythonContext(Operator(omega))
petsc_op.setUp()
ksp = PETSc.KSP()
ksp.create()
ksp.setOperators(petsc_op)
```

Algorithmic performance

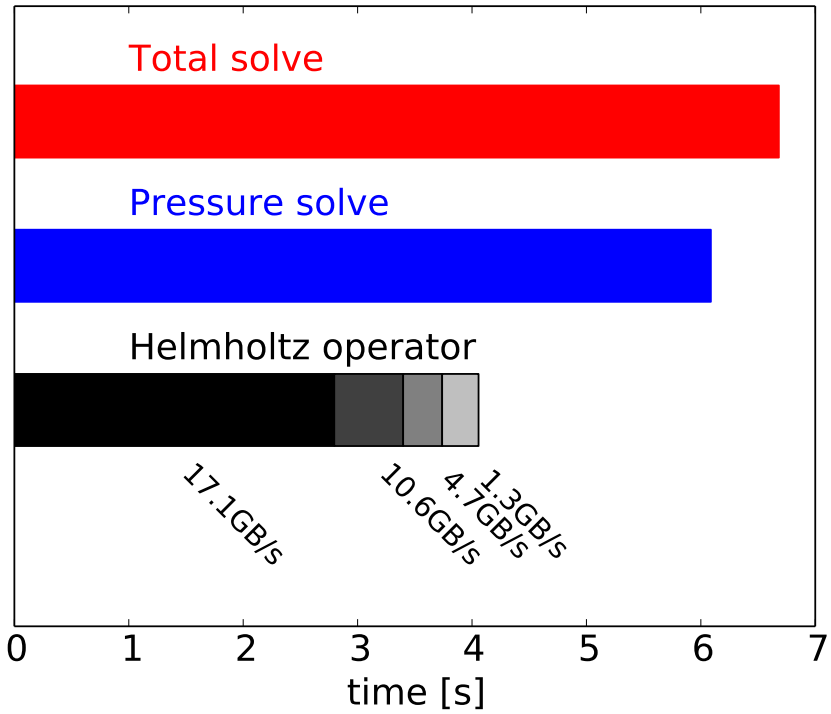
Convergence history

- Inner Solve: 3 CG iterations with multigrid preconditioner
- Icosahedral grid, 327,680 cells
- 4 multigrid levels



Efficiency

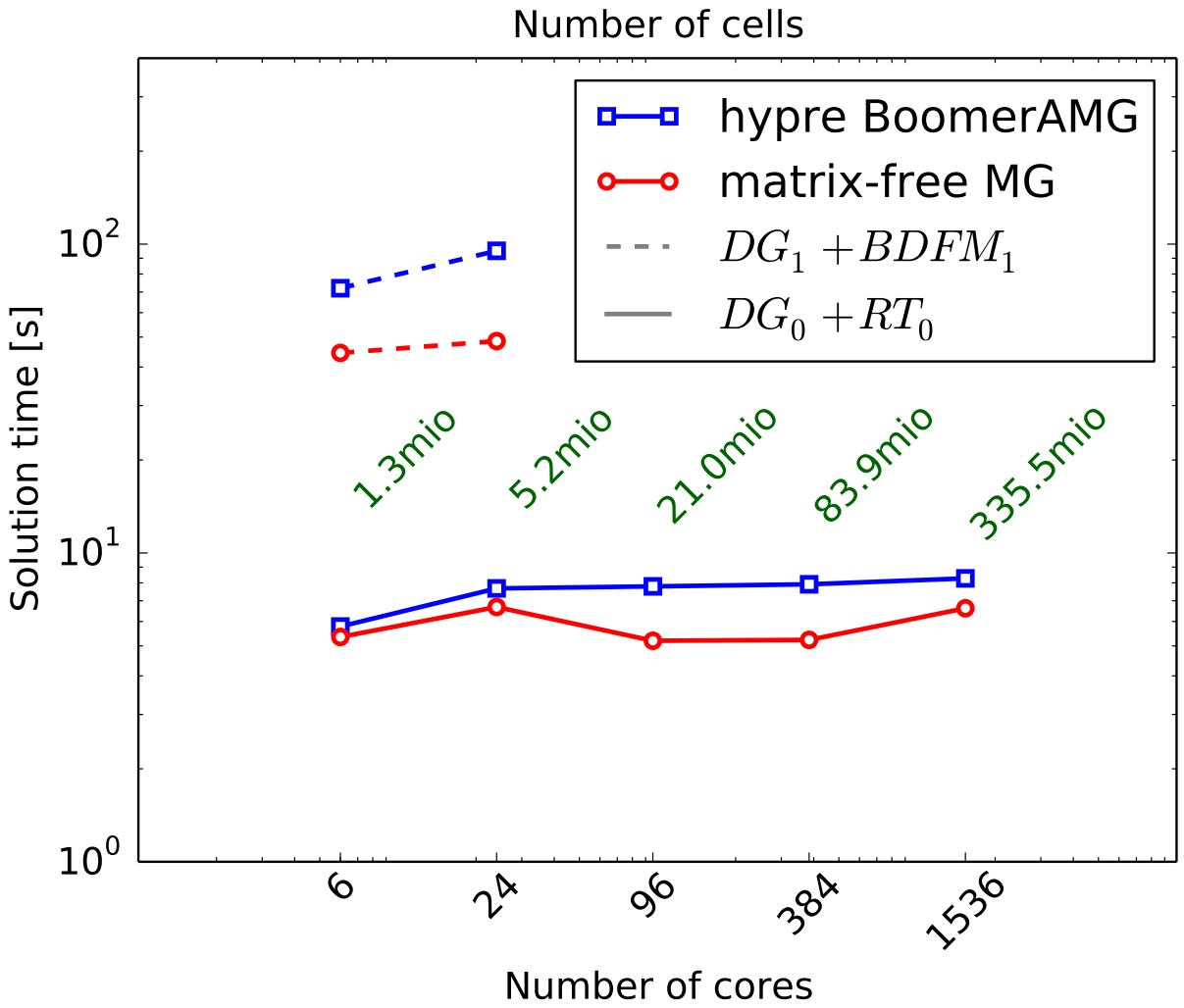
Single node run on ARCHER, lowest order, $5.2 \cdot 10^6$ cells [preliminary]



STREAM triad: 74.1GB/s per node $\Rightarrow$ up to $\approx$ 23% of peak BW

Parallel scalability

Weak scaling on ARCHER [preliminary]



Conclusion

Summary

- Matrix-free multigrid-preconditioner for pressure correction equation
- Implementation in firedrake/PyOP2/PETSc framework:
  - Performance portability
  - Correct abstractions $\Rightarrow$ Separation of concerns

Outlook

- Test on GPU backend
- Extend to 3d: Regular vertical grid $\Rightarrow$ Improved caching $\Rightarrow$ Improved memory BW
- Improve and extend parallel scaling
- Full 3d dynamical core (Colin Cotter)

References

The **firedrake** project: http://firedrakeproject.org/

- **F. Rathgeber et al.**: *Firedrake: automating the finite element method by composing abstractions.* in preparation
- **F. Rathgeber et al.**: *PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes.* HPC, Networking Storage and Analysis, SC Companion, p 1116-1123, Los Alamitos, CA, USA, 2012. IEEE Computer Society
- **E. Müller et al.**: *Petascale elliptic solvers for anisotropic PDEs on GPU clusters.* Submitted to Parallel Computing (2014) [arxiv:1402.3545]
- **E. Müller, R. Scheichl**: *Massively parallel solvers for elliptic PDEs in Numerical Weather- and Climate Prediction.* QJRMS (2013) [arxiv:1307.2036]

Helmholtz solver code on github

https://github.com/firedrakeproject/firedrake-helmholtzsolver