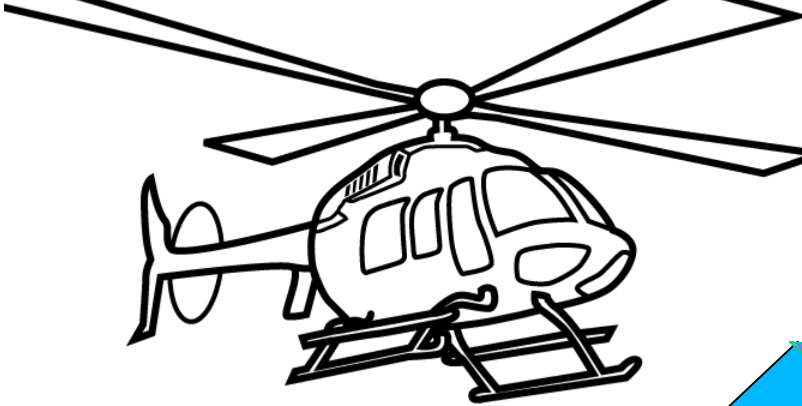
Imperial College London

Delivering and generalising domain-specific program optimisations

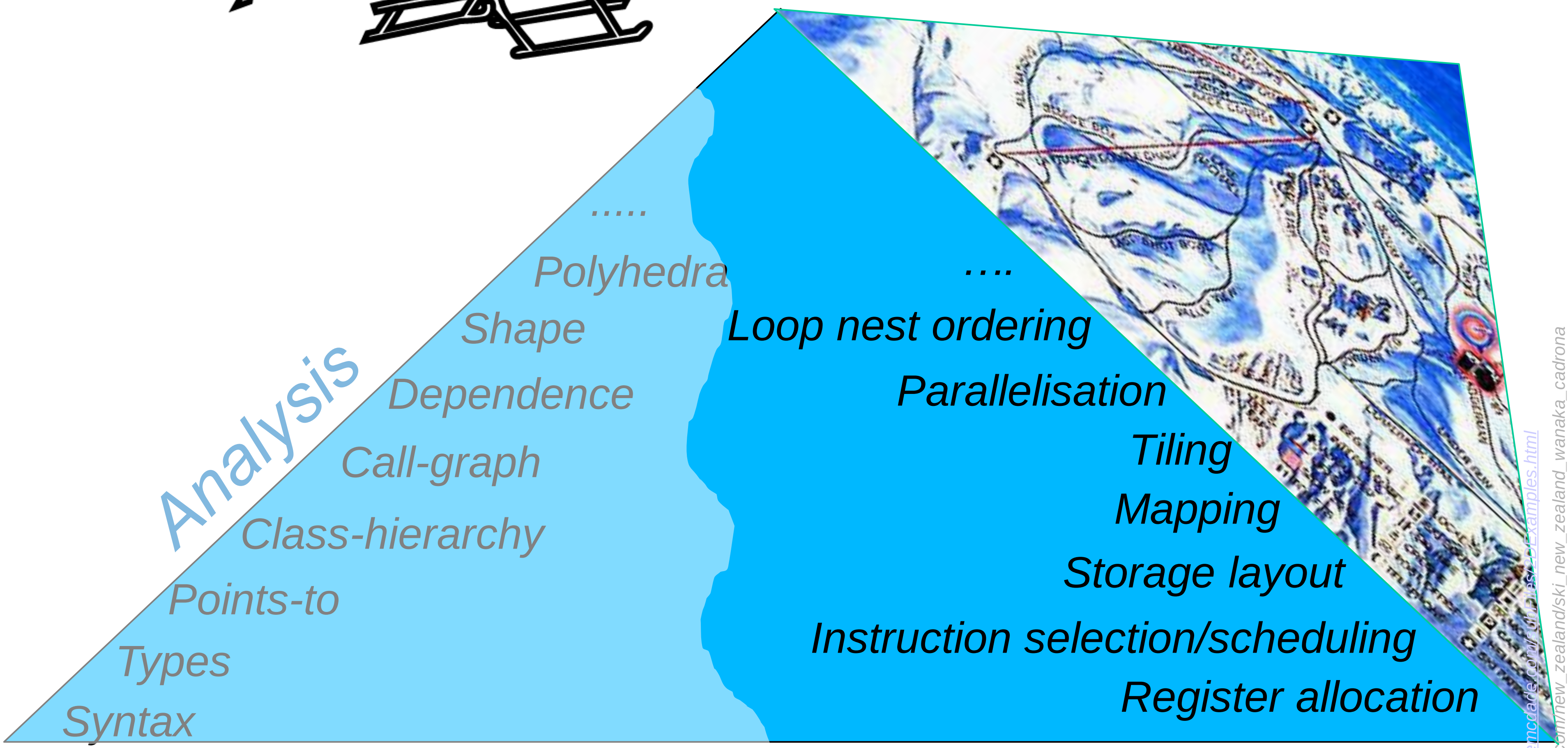
Paul H J Kelly
Group Leader, Software Performance Optimisation
Co-Director, Centre for Computational Methods in Science and Engineering
Department of Computing, Imperial College London

Joint work with :

David Ham (Imperial Computing/Maths/Grantham Inst for Climate Change)
Gerard Gorman, Michael Lange (Imperial Earth Science Engineering – Applied Modelling and Computation Group)
Mike Giles, Gihan Mudalige, Istvan Reguly (Mathematical Inst, Oxford)
Doru Bercea, Fabio Luporini, Graham Markall, Lawrence Mitchell, Florian Rathgeber, Francis Russell, George Rokos, Paul Colea (Software Perf Opt Group, Imperial Computing)
Spencer Sherwin (Aeronautics, Imperial), Chris Cantwell (Cardio-mathematics group, Mathematics, Imperial)
Michelle Mills Strout, Chris Krieger, Cathie Olschanowsky (Colorado State University)
Carlo Bertolli (IBM Research), Ram Ramanujam (Louisiana State University)
Doru Thom Popovici, Franz Franchetti (CMU), Karl Wilkinson (Capetown), Chris–Kriton Skylaris (Southampton)



Compilation is like skiing



- Analysis is not always the interesting part....
- It's more fun the higher you start!

http://www.nikkiemcdade.com/subFiles/2DExamples.html

http://www.ginz.com/new_zealand/ski_new_zealand_wanaka_cadrona

Imperial College London

Have your cake and eat it too

This talk is about the following idea:

- can we simultaneously
  - raise the level at which programmers can reason about code,
  - provide the compiler with a model of the computation that enables it to generate faster code than you could reasonably write by hand?

What we are doing….

Targetting MPI, OpenMP, OpenCL, Dataflow/ FPGA, from HPC to mobile, embedded and wearable

Technologies

- *Vectorisation, parametric polyhedral tiling*
- *Tiling for unstructured-mesh stencils*
- *Lazy, data-driven compute-communicate*
- *Runtime code generation*
- *Multicore graph worklists*
- *Massive common sub-expressions*
- *Optimisation of composite transforms*

Contexts

- *Finite difference*
- *Finite-volume*
- *Finite-element*
- *Real-time 3D scene understanding*
- *Adaptive-mesh CFD*
- *Unsteady CFD - higher-order flux-reconstruction*
- *Ab-initio computational chemistry (ONETEP)*

Projects

- ***PyOP2/OP2*** *Unstructured-mesh stencils*
- ***Firedrake*** *Finite-element assembly*
- ***SLAMBench*** *Dense SLAM – 3D vision*
- ***PRAgMaTIc*** *Dynamic mesh adaptation*
- ***GiMMiK*** *Small-matrix multiplication*
- ***TINTL*** *Fourier interpolation*

Applications

- *Aeroengine turbo-machinery*
- *Weather and climate*
- *Domestic robotics, augmented reality*
- *Tidal turbines*
- *Formula-1, UAVs*
- *Solar energy, drug design*

This talk

- What compilers can do
- What stops the compiler from doing what it can do
- What you might hope the compiler might do

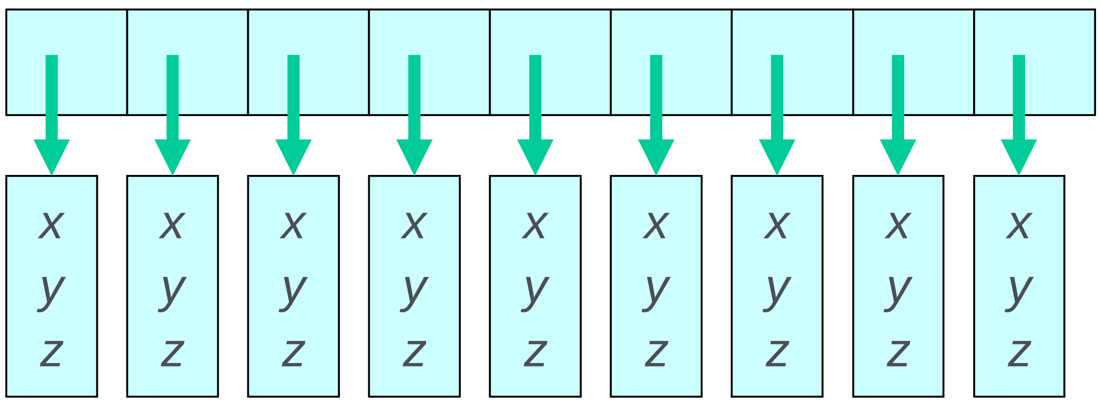
- Domain-specific optimisations
- Getting the abstraction right
- Delivering

Easy parallelism

Example:

```
for (i=0; i<N; ++i) {
  points[i]->x += 1;
}
```

- Can the iterations of this loop be executed in parallel?



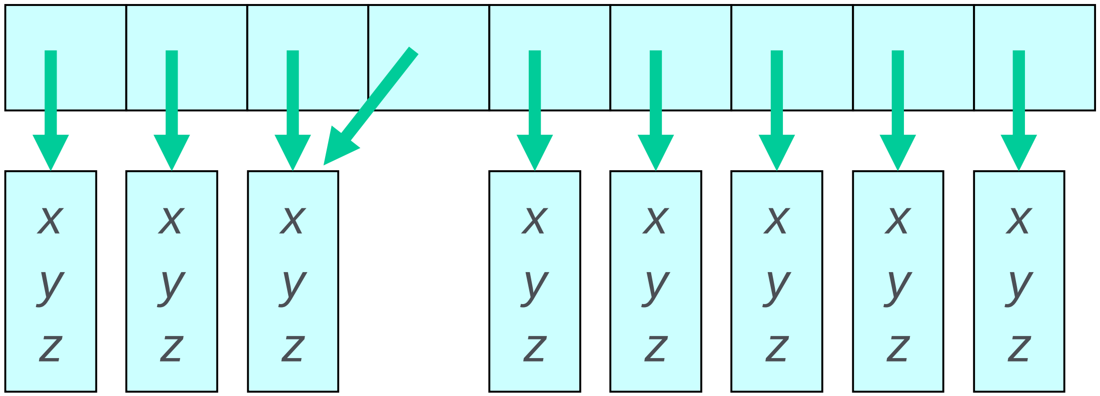
- No problem: each iteration is independent

Easy parallelism

Example:

```
for (i=0; i<N; ++i) {
  points[i]->x += 1;
}
```

- Can the iterations of this loop be executed in parallel?



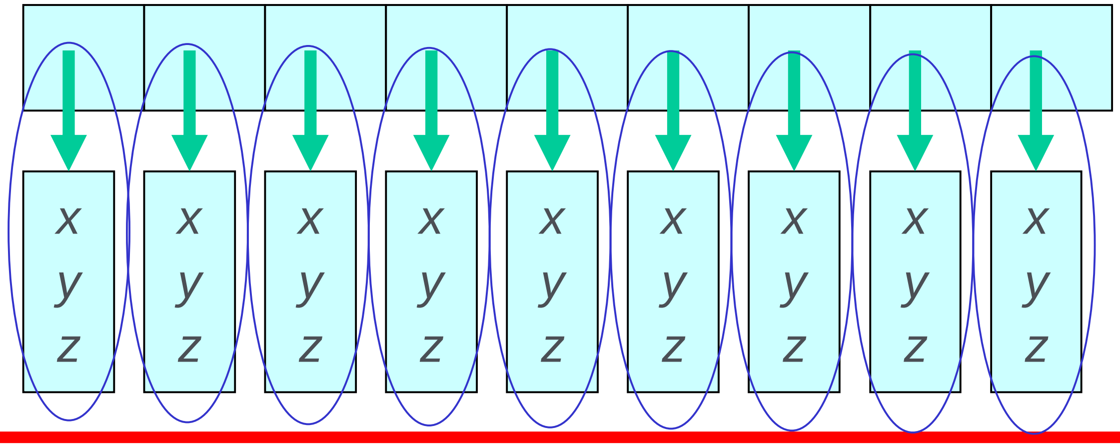
- Oh no: not all the iterations are independent!
  - You want to re-use piece of code in different contexts
  - Whether it's parallel depends on context!

Easy parallelism

Example:

```
for (i=0; i<N; ++i) {
  points[i]->x += 1;
}
```

- Can the iterations of this loop be executed in parallel?



Sergio Almeida's 1998 PhD thesis:

"Balloon types" ensure that each cell is reached only by its owner pointer – see also ownership in Rust

Points-to analysis

2006 PhD thesis work of David Pearce,
(based on Andersen'94)

```
int *f(int *p) {
  return p;
}
int g() {
  int x,y,*p,*q,**r,**s;
  s=&p;
  if(...) p=&x;
  else p=&y;
  r=s;
  q=f(*r);
}
```

(1) $f_* \supseteq f_p$

(2) $g_s \supseteq \{g_p\}$

(3) $g_p \supseteq \{g_x\}$

(4) $g_p \supseteq \{g_y\}$

(5) $g_r \supseteq g_s$

(6) $f_p \supseteq *g_r$

(7) $g_q \supseteq f_*$

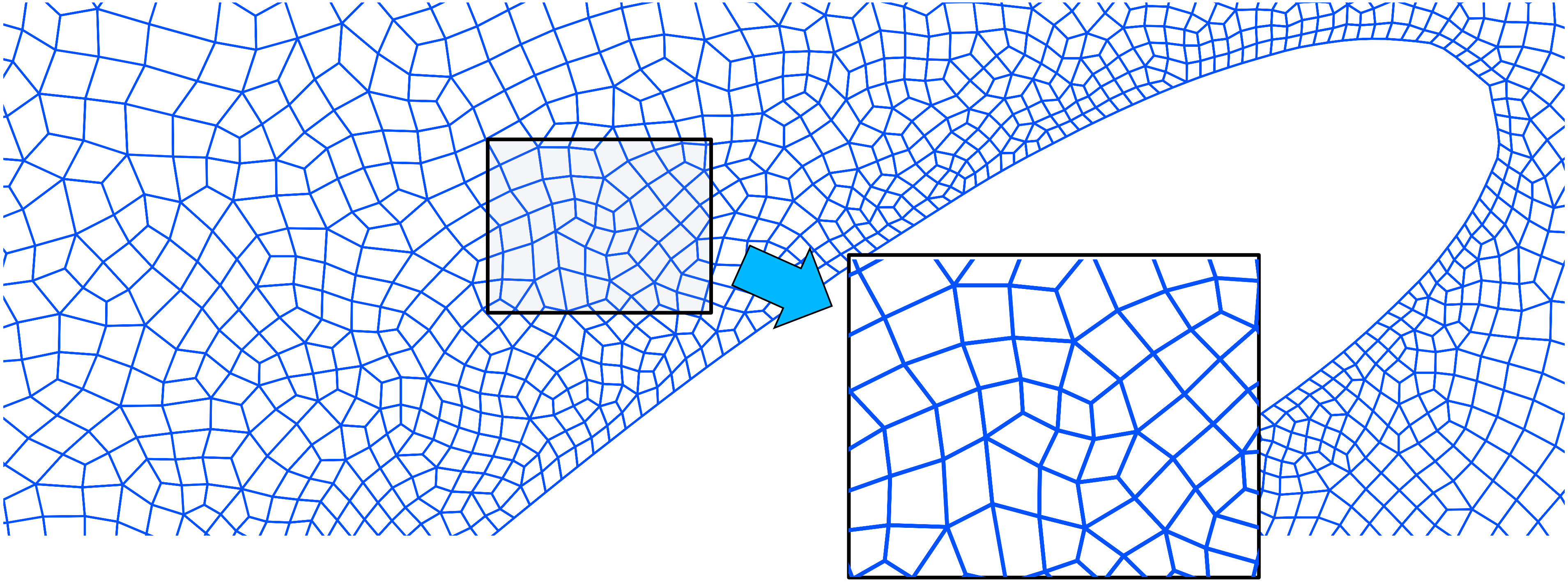
Variable s of function g might point to variable p of function g

R might point to anything s might point to

f's p might point to anything r might point to

q might point to anything f returns

- Goal: for each pointer variable (p,q,r,s), find the set of objects it might point to at runtime

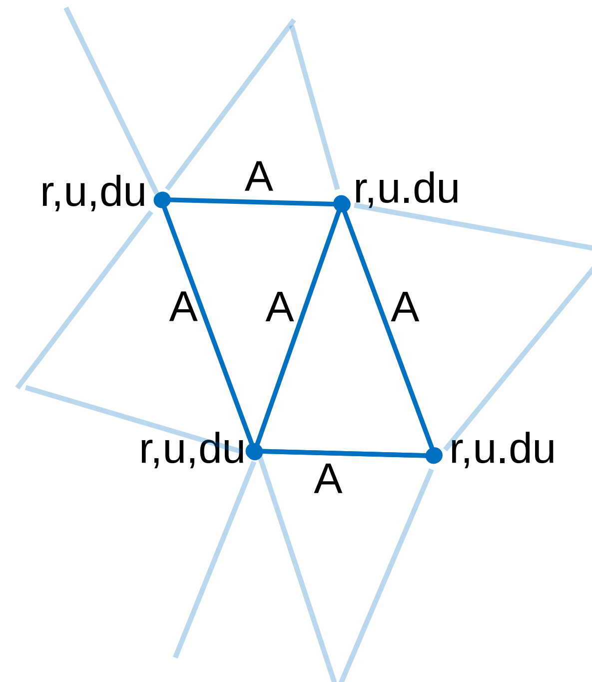


- Unstructured meshes require pointers/indirection because adjacency lists have to be represented explicitly
- A controlled form of pointers (actually a general graph)

- **OP2** is a C++ and Fortran library for parallel loops over the mesh implemented by source-to-source transformation
- **PyOP2** is an major extension implemented in Python using runtime code generation

- Generates highly-optimised CUDA, OpenMP and MPI code

```
# declare sets, maps, and datasets
nodes = op2.Set(nnode)
edges = op2.Set(nedge)

ppedge = op2.Map(edges, nodes, 2, pp)

p_A = op2.Dat(edges, data=A)
p_r = op2.Dat(nodes, data=r)
p_u = op2.Dat(nodes, data=u)
p_du = op2.Dat(nodes, data=du)

# global variables and constants declarations
alpha = op2.Const(1, data=1.0, np.float32)
beta = op2.Global(1, data=1.0, np.float32)
```

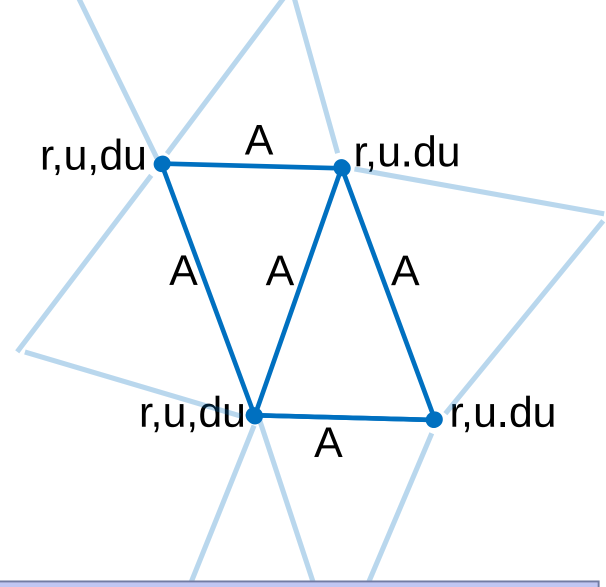


https://github.com/OP2/PyOP2/blob/master/demo/jacobi.py

Example – Jacobi solver

PyOP2: "decoupled access-execute"

- Parallel loops, over sets (nodes, edges etc)
- Access descriptors specify how to pass data to and from the C kernel
- The kernel operates only on local data




```
for iter in xrange(0, NITER):
    u_sum = op2.Global(1, data=0.0, np.float32)
    u_max = op2.Global(1, data=0.0, np.float32)
    op2.par_loop(res, edges,
                 p_A(op2.READ),
                 p_u(op2.READ, ppedge[1]),
                 p_du(op2.INC, ppedge[0]),
                 beta(op2.READ))

    op2.par_loop(update, nodes,
                 p_r(op2.READ),
                 p_du(op2.RW),
                 p_u(op2.INC),
                 u_sum(op2.INC),
                 u_max(op2.MAX))
```

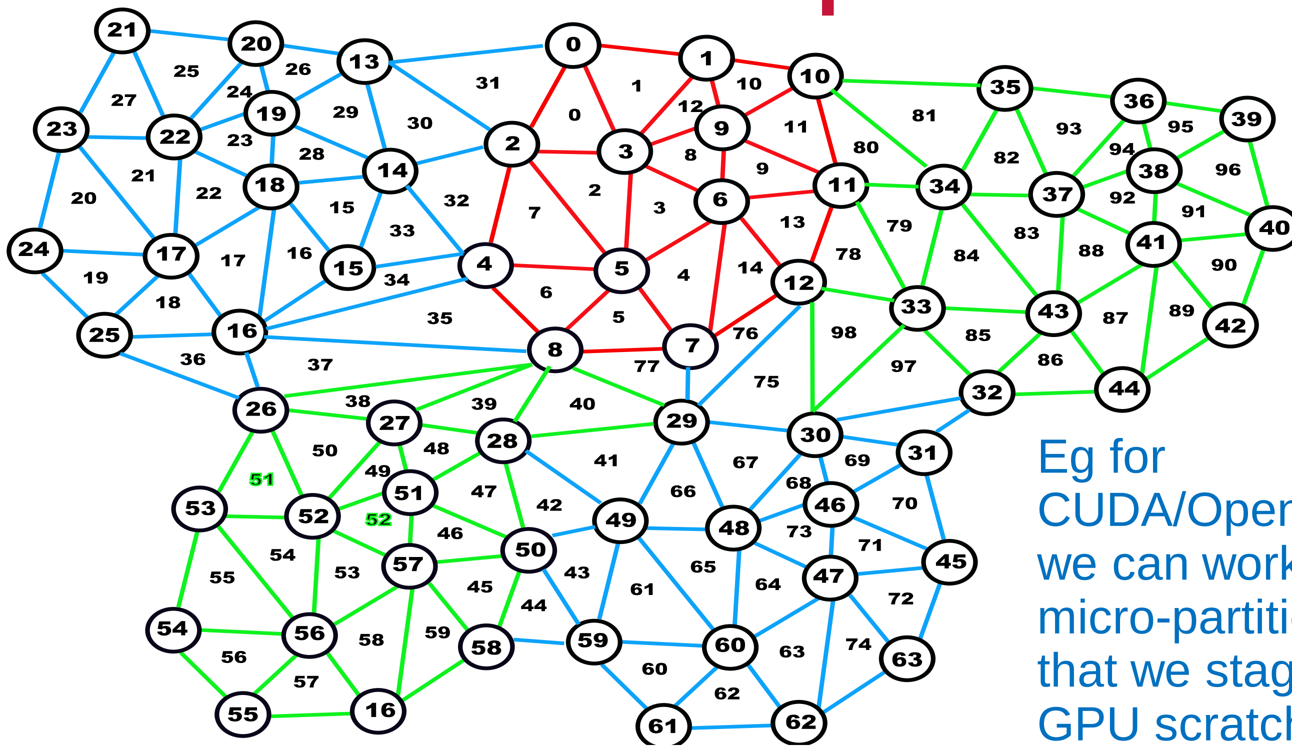
Access descriptors specify how to feed the kernel from the mesh

```
void res(float *A, float *u, float *du,
         const float *beta) {
  *du += (*beta) * (*A) * (*u);
}
```

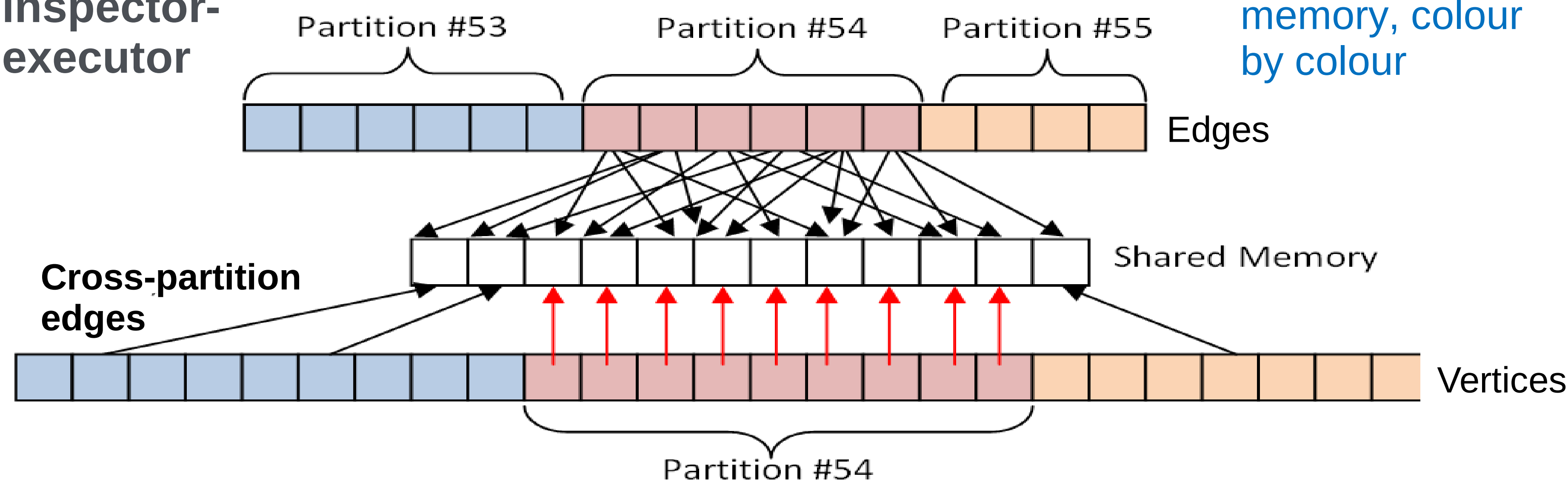
```
void update(float *r, float *du, float *u, float
       *u_sum, float *u_max) {
  *u += *du + alpha * (*r);
  *du = 0.0f;
  *u_sum += (*u) * (*u);
  *u_max = *u_max > *u ? *u_max : *u;
}
```

Code generation for indirect loops in OP2

- Supports diverse code generation schemes
- For MPI, OpenMP, GPU, and in prototype form for FPGA
- Key idea: **inspector-executor**

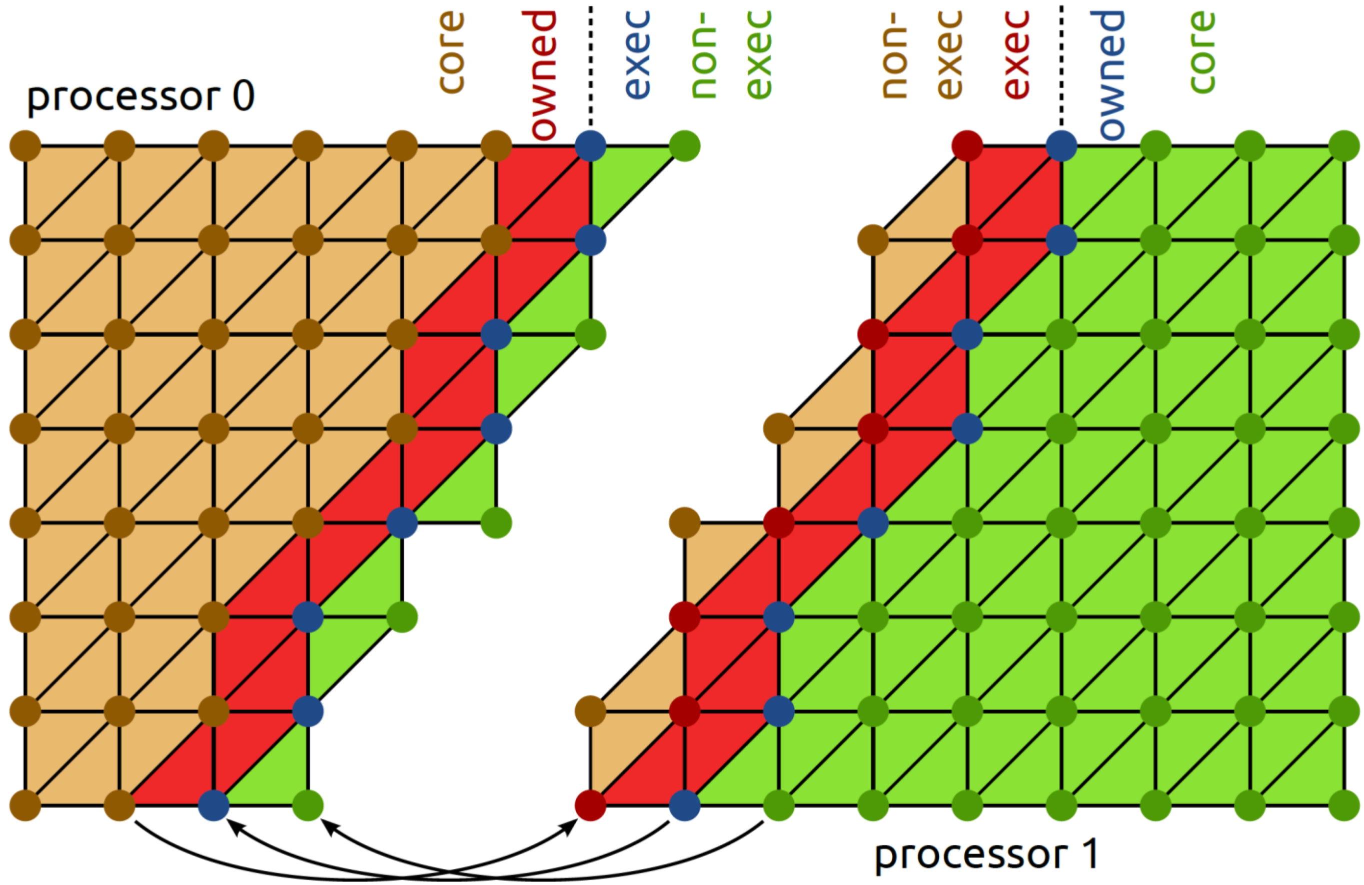


Eg for CUDA/OpenCL we can work with micro-partitions that we stage into GPU scratchpad memory, colour by colour



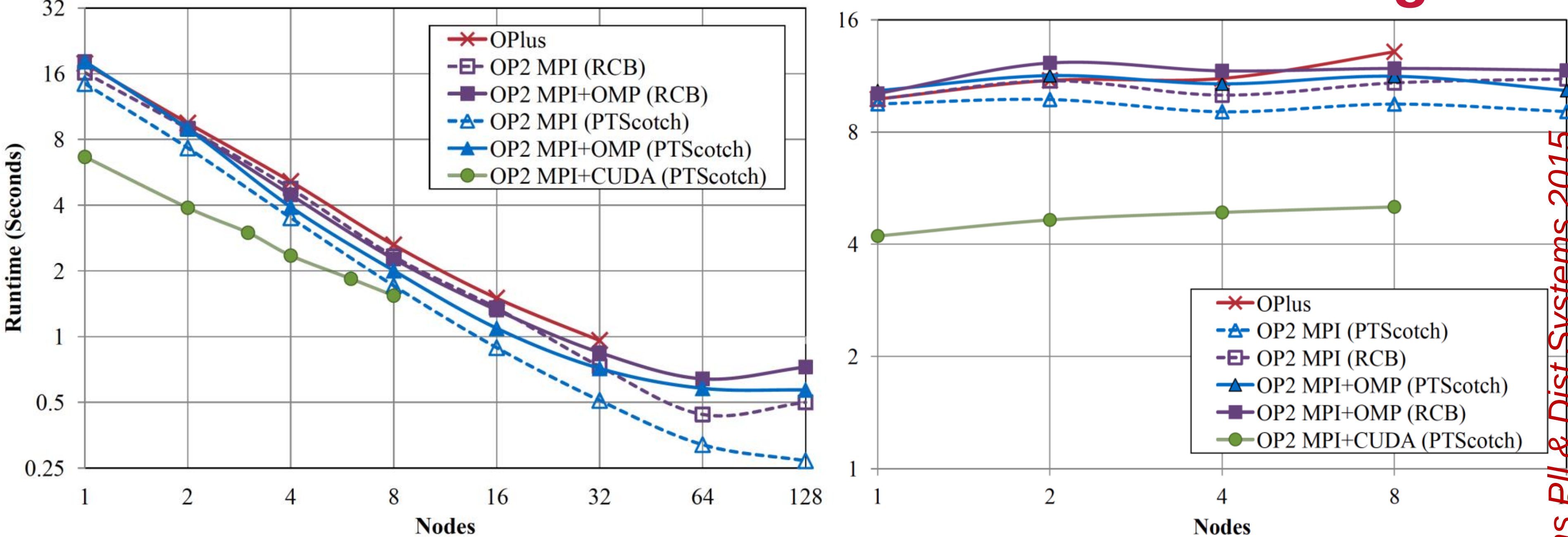
Code generation for indirect loops in PyOP2

- For MPI we precompute partitions & haloes
- Derived from PyOP2 access descriptors, implemented using PetSC DMPlex
- At partition boundaries, the entities (vertices, edges, cells) form layered halo region



- **Core:** entities owned which can be processed without accessing halo data.
- **Owned:** entities owned which access halo data when processed
- **Exec halo:** off-processor entities which are 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

HYDRA: Full-scale industrial CFD using OP2



(a) Strong Scaling (2.5M edges)

(b) Weak Scaling (0.5M edges per node)

- ***Unmodified Fortran OP2 source code exploits inter-node parallelism using MPI, and intra-node parallelism using OpenMP and CUDA***
- ***Application is a proprietary, full-scale, in-production fluids dynamics package***
- ***Developed by Rolls Royce plc and used for simulation of aeroplane engines***

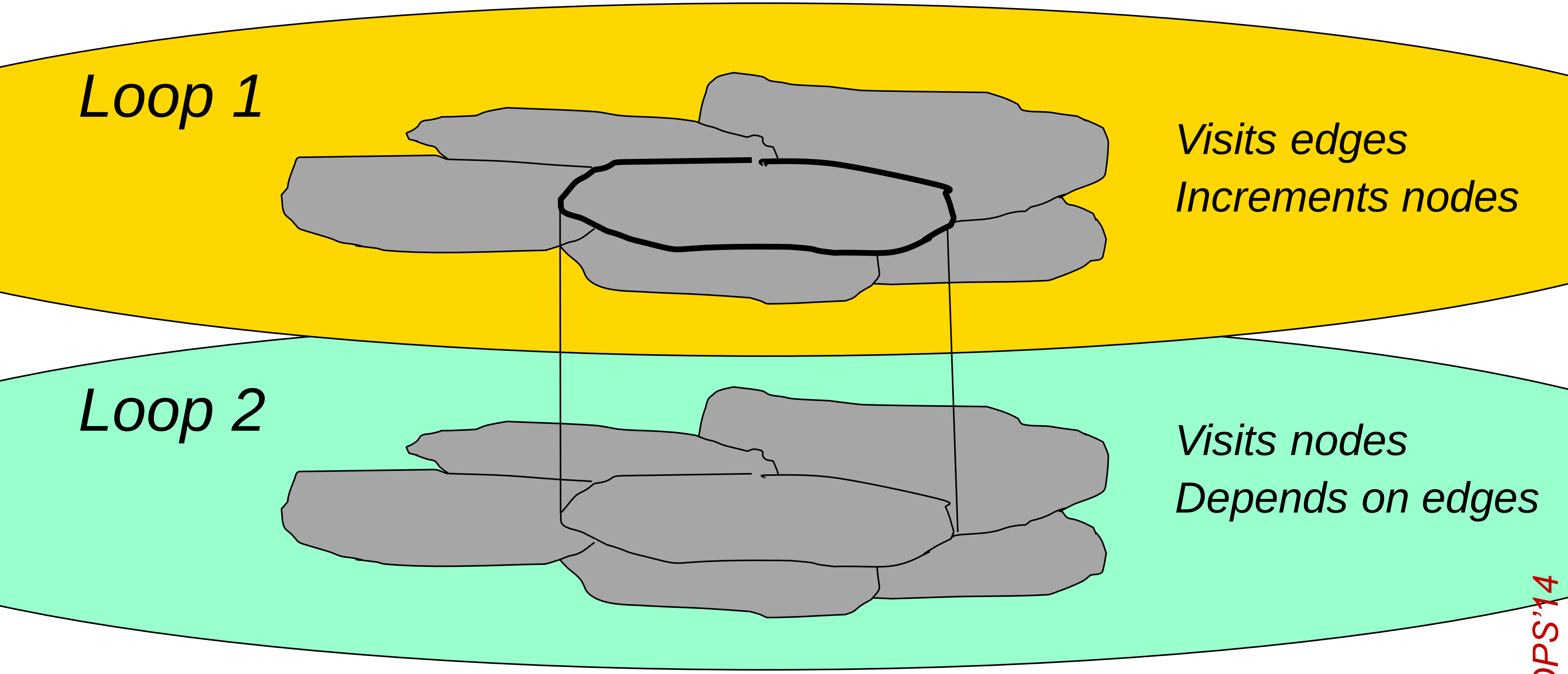
(joint work with Mike Giles, Istvan Reguly, Gihan Mudalige at Oxford)

- ***"Performance portability"***

HECToR (Cray XE6)	Jade (NVIDIA GPU Cluster)
2×16-core AMD Opteron 6276 (Interlagos)2.3GHz	2×Tesla K20m + Intel Xeon E5-1650 3.2GHz
32GB	5GB/GPU (ECC on)
128	8
Cray Gemini	FDR InfiniBand
CLE 3.1.29	Red Hat Linux Enterprise 6.3
Cray MPI 8.1.4	PGI 13.3, ICC 13.0.1, OpenMPI 1.6.4
-O3 -h fp3 -h ipa5	-O2 -xAVX -arch=sm_35 -use_fast_math

Reguly, Mudalige et al, IEEE Trans Pll & Dist Systems 2015

Sparse split tiling on an unstructured mesh, for locality



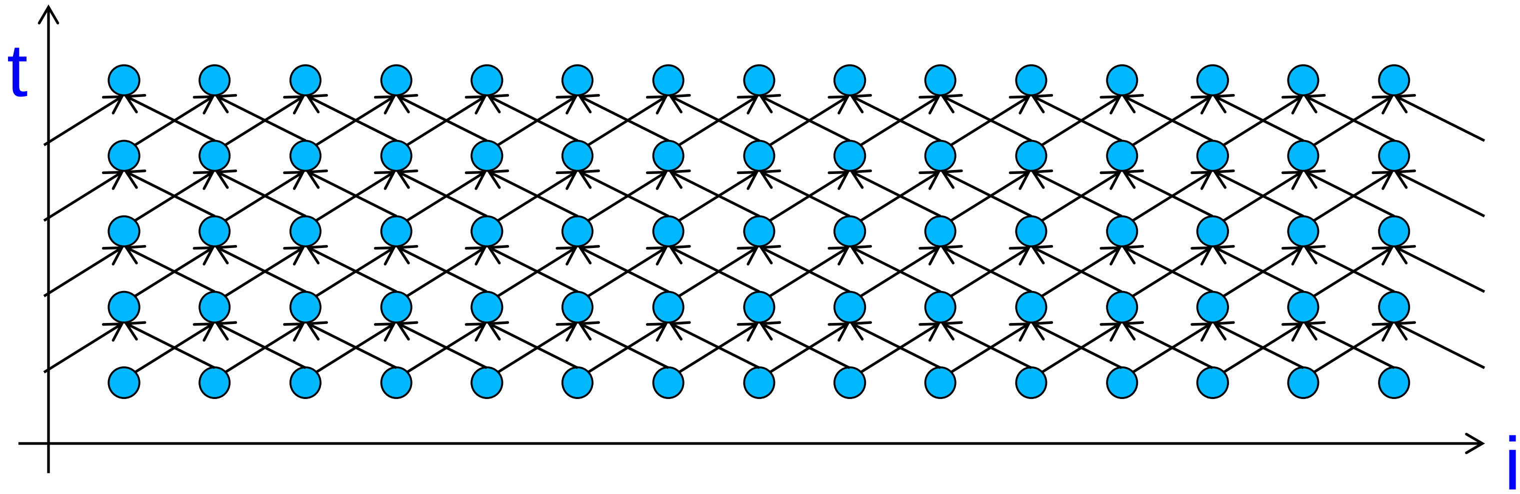
- How can we fuse two loops, when there is a “halo” dependence?
- I.e. load a block of mesh and do the iterations of loop 1, then the iterations of loop 2, before moving to the next block
- If we could, we could dramatically improve the memory access behaviour!

Strout, Luporini et al, IPDPS’14

Tiling a *structured* mesh for locality

- To understand sparse split tiling, we need to first understand *split* tiling
- Consider a 1D stencil loop, iterated a number of times

```
for (t=0; t<N; ++t)
  for (i=1; i<M-1; ++i)
    U[t+1][i] = U[t][i-1] + U[t][i+1]
```

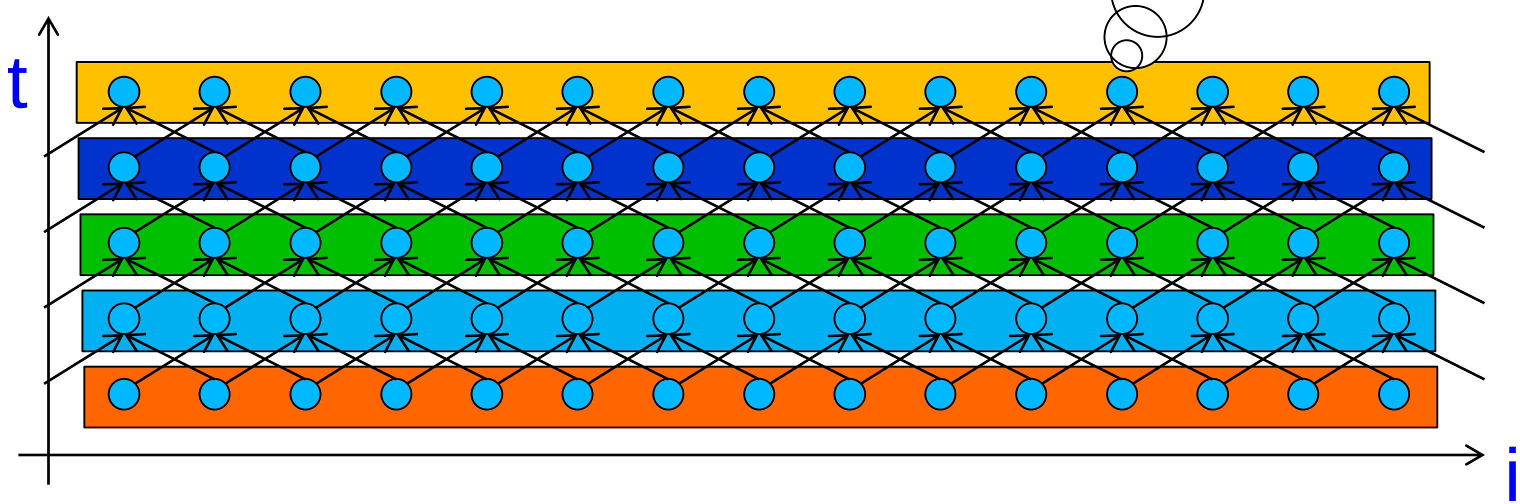


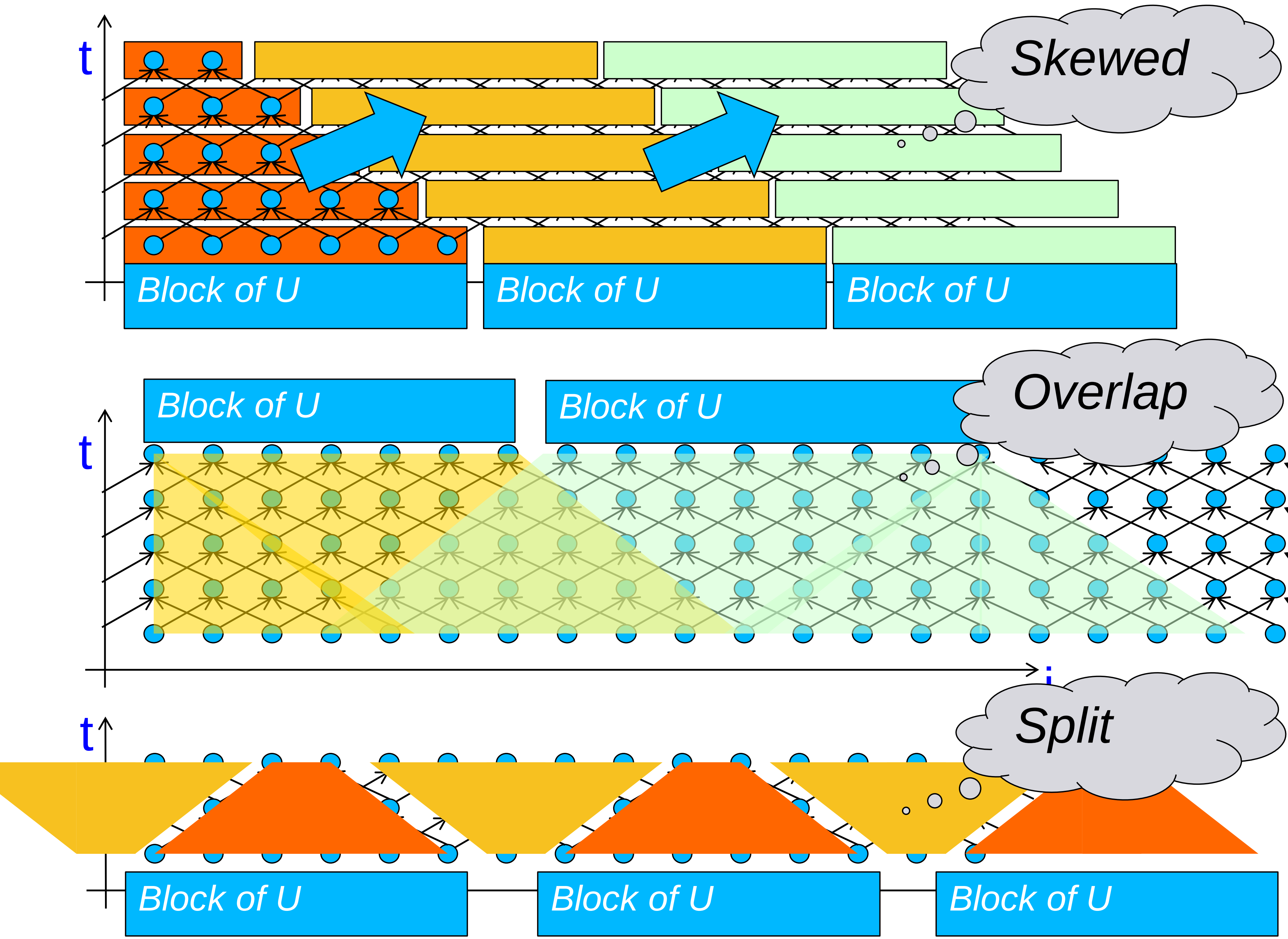
Tiling a *structured* mesh for locality

- To understand sparse split tiling, we need to first understand *split* tiling
- Consider a 1D stencil loop … number of times

```
for (t=0; t<N; ++t)
  for (i=1; i<M-1; ++i)
    U[t+1][i] = U[t][i-1] + U[t
```

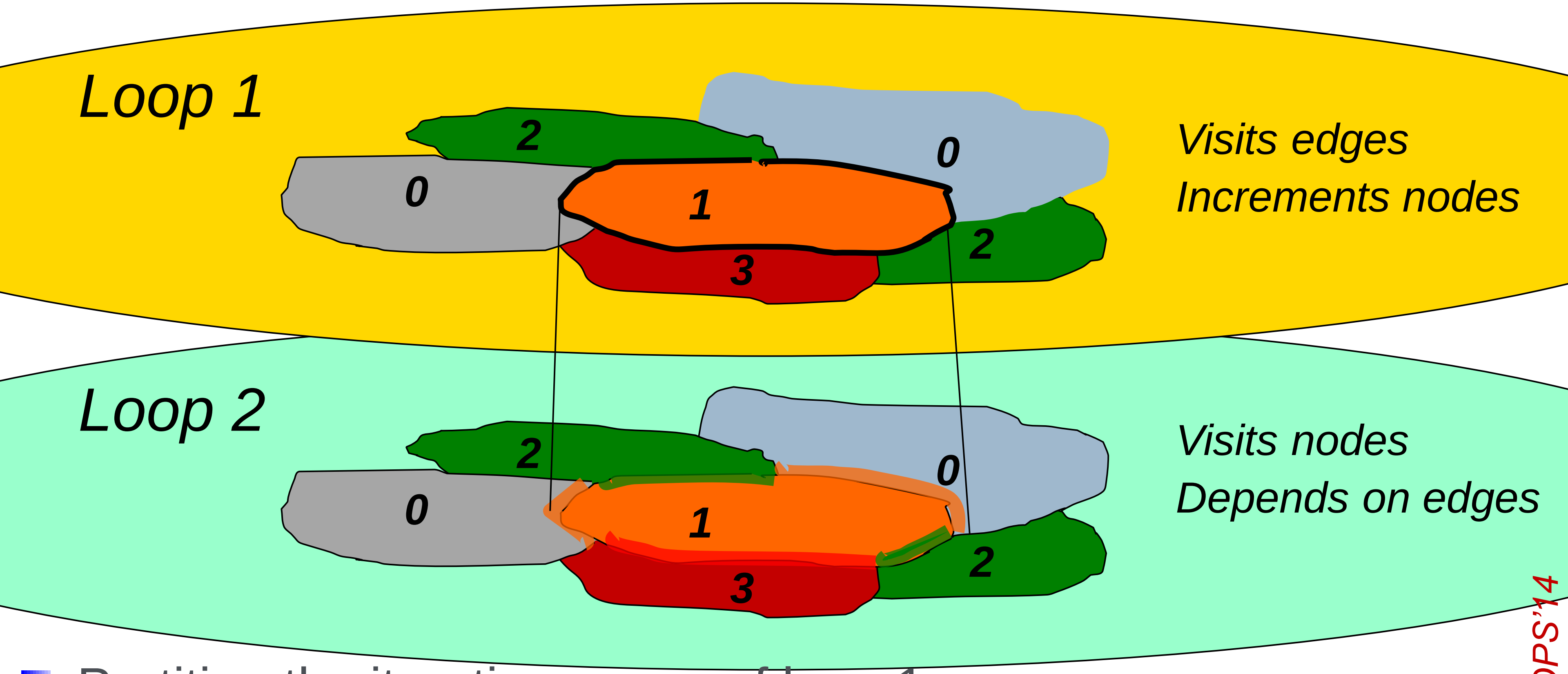
Lots of parallelism – but lots of *data movement*





t
Skewed
Block of U
Block of U
Block of U
t
Overlap
Block of U
Block of U
i
t
Split
Block of U
Block of U
Block of U

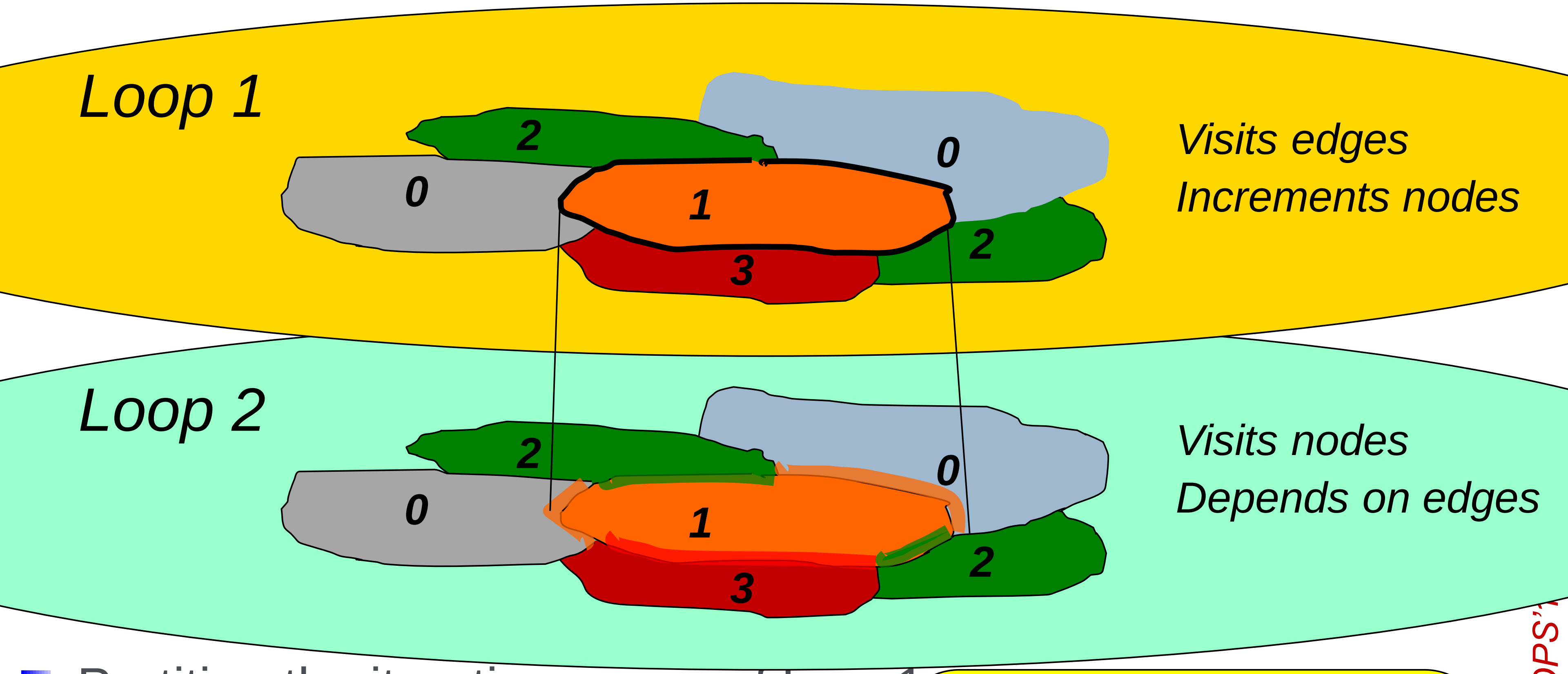
Sparse split tiling



- Partition the iteration space of loop 1
- Colour the partitions, execute the colours in order
- Project the tiles, using the knowledge that colour n can use data produced by colour n-1
- Thus, the tile coloured #1 *grows* where it meets colour #0
- And *shrinks* where it meets colours #2 and #3

Strout, Luporini et al, IPDPS'14

Sparse split tiling

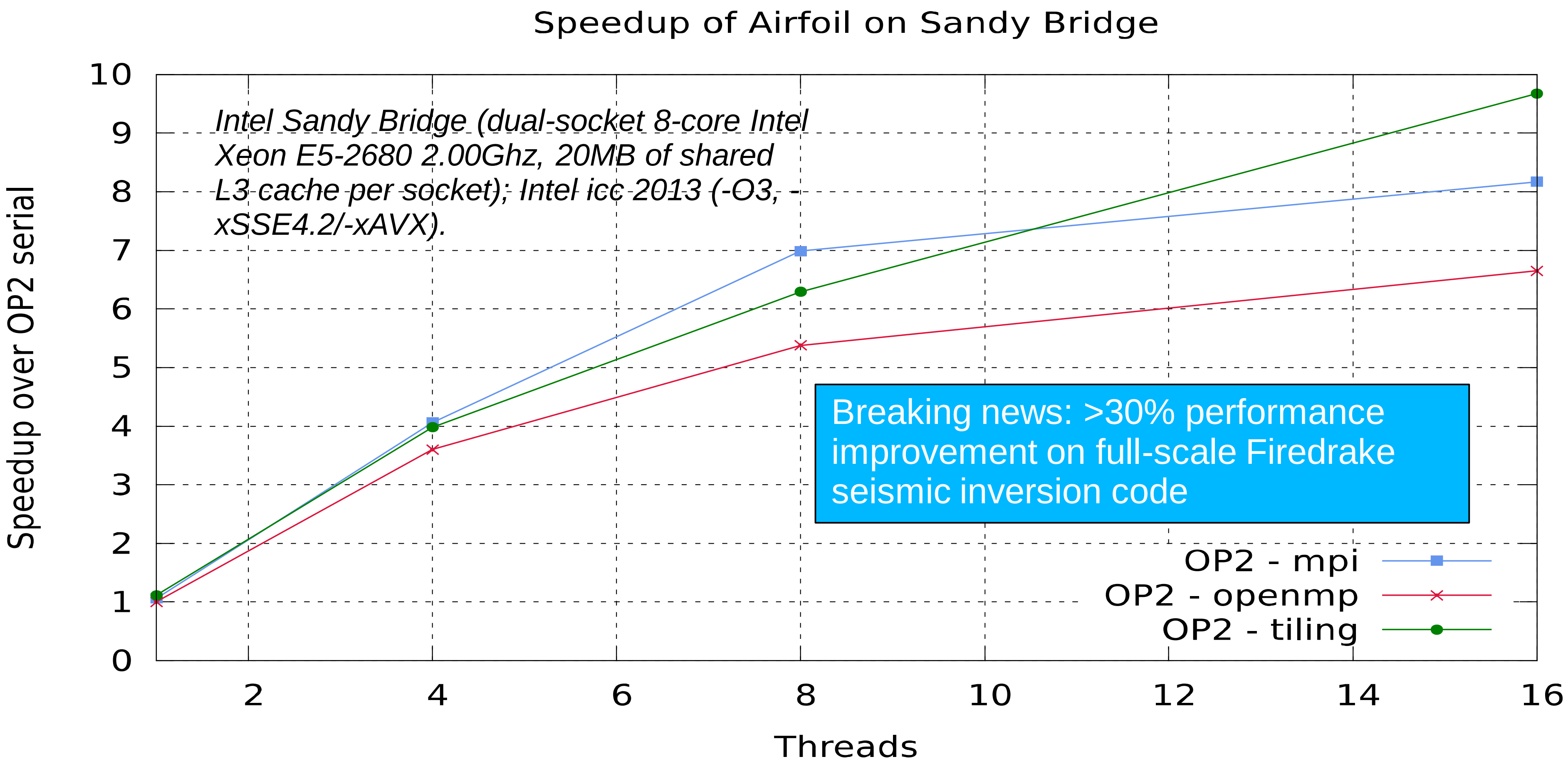


- Partition the iteration space of loop 1
- Colour the partitions
- Project the tiles, using the knowledg
  data produced by colour n-1
- Thus, the tile coloured #1 grows wh
- And shrinks where it meets colours #2 and #3

Inspector-executor: derive tasks and task graph from the mesh, ***at runtime***

Strout, Luporini et al, IPDPS'1

OP2 loop fusion in practice

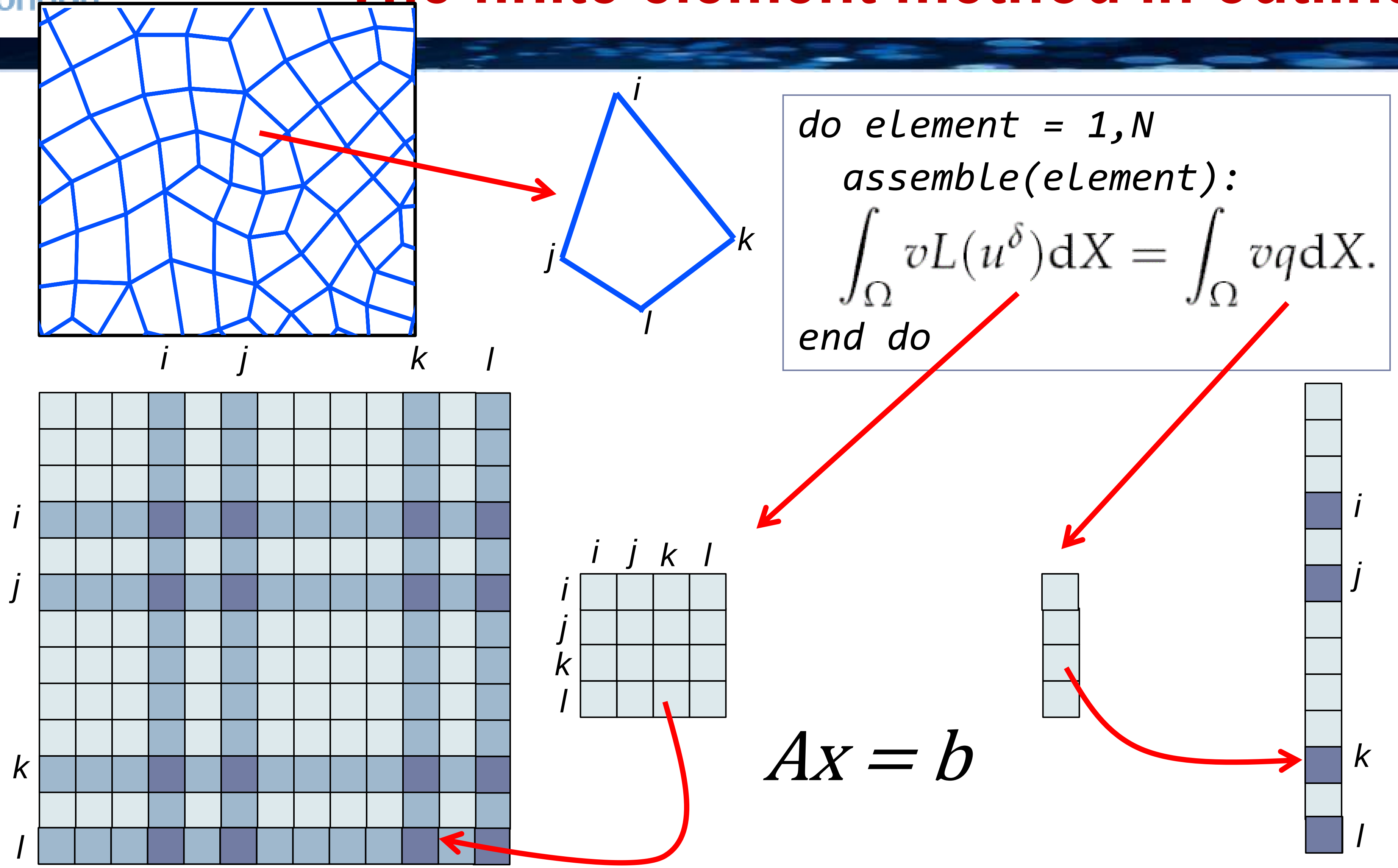


- *Mesh size = 1.5M edges*
- *# Loop chain = 6 loops*
- *No inspector/plans overhead*
- *Airfoil test problem*
- *Unstructured-mesh finite-volume*

- **Where did the domain-specific advantage come from?**
  - OP2's access descriptors provide precise dependence iteration-to-iteration information
    - Could easily be delivered in a lambda-based parallel loop framework
  - We "know" that we will iterate many times over the same mesh – so it's worth investing in an expensive "inspector-executor" scheme
  - We capture chains of loops over the mesh
    - We *could* get our compiler to find adjacent loops
    - We *could* extend the OP2 API with "loop chains"
  - What we actually do?
    - We delay evaluation of parallel loops
    - We build a chain (DAG) of parallel loops at runtime
    - We generate code at runtime for the traces that occur

Strout, Luporini et al IPDPS 2014
Luporini PhD thesis, forthcoming

The finite element method in outline




```
do element = 1,N
  assemble(element):
```

$$\int_{\Omega} vL(u^{\delta})\mathrm{d}X = \int_{\Omega} vq\mathrm{d}X.$$

```
end do
```

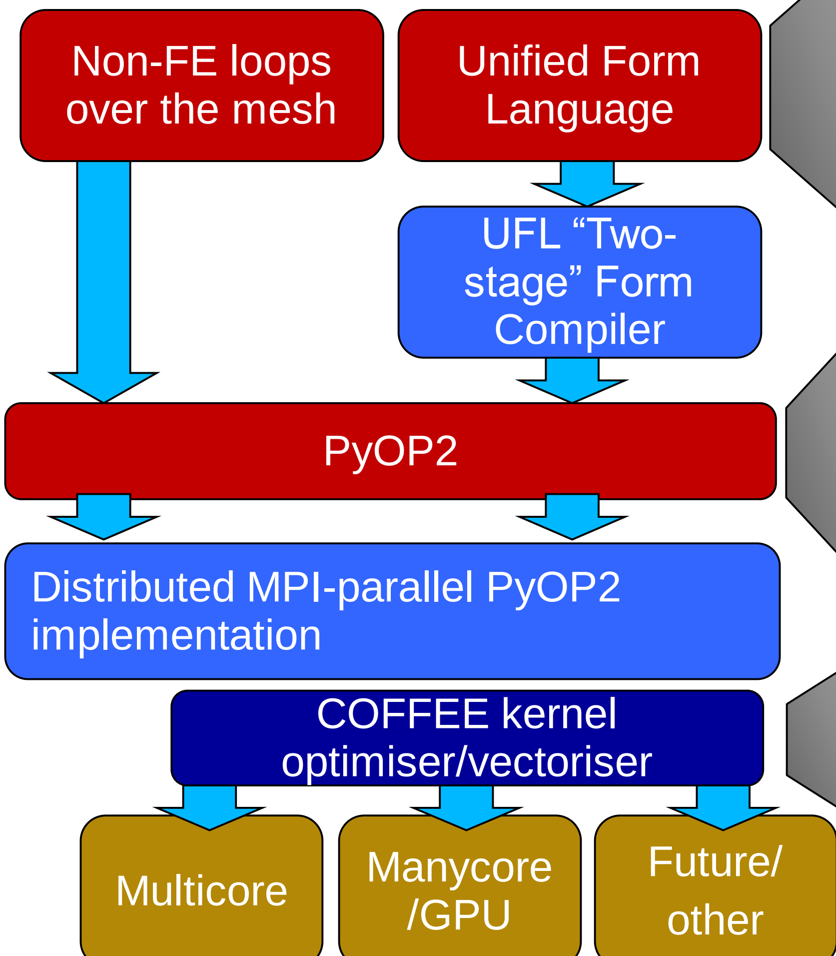
- Key data structures: Mesh, dense local assembly matrices, sparse global system matrix, and RHS vector

Multilayered abstractions for FE

- Local assembly:
  - Specified using the FEniCS project’s DSL, UFL (the “Unified Form Language”)
  - Computes local assembly matrix
  - Key operation is evaluation of expressions over basis function representation of the element
- Mesh traversal:
  - *PyOP2*
  - *Loops over the mesh*
  - *Key is orchestration of data movement*
- Solver:
  - Interfaces to standard solvers, such as PetSc

Firedrake: a finite-element framework

- An alternative implementation of the FEniCS language
- Using PyOP2 as an intermediate representation of parallel loops
- All embedded in Python using runtime code generation



- The FEniCS project’s UFL – DSL for finite element discretisation
- Compiler generates PyOP2 kernels and access descriptors

- Stencil DSL for *unstructured-mesh*
- Explicit *access descriptors* characterise access footprint of kernels

- Domain-specific loop optimizer
- For finite-element assembly and similar loop nests
- Vectorisation and flop-minimisation

Rathgeber, Ham, Mitchell et al, ACM TOMS 2016

- **The advection-diffusion problem:**

$$\frac{\partial T}{\partial t} = \underbrace{D\nabla^2 T}_{\text{Diffusion}} - \underbrace{\mathbf{u}\cdot\nabla T}_{\text{Advection}}$$

- Weak form:

$$\int_\Omega q\frac{\partial T}{\partial t}\,\mathrm{d}X = \int_{\partial\Omega} q(\nabla T - \mathbf{u}T)\cdot\mathbf{n}\,\mathrm{d}s - \int_\Omega \nabla q\cdot\nabla T\,\mathrm{d}X + \int_\Omega \nabla q\cdot\mathbf{u}T\,\mathrm{d}X$$

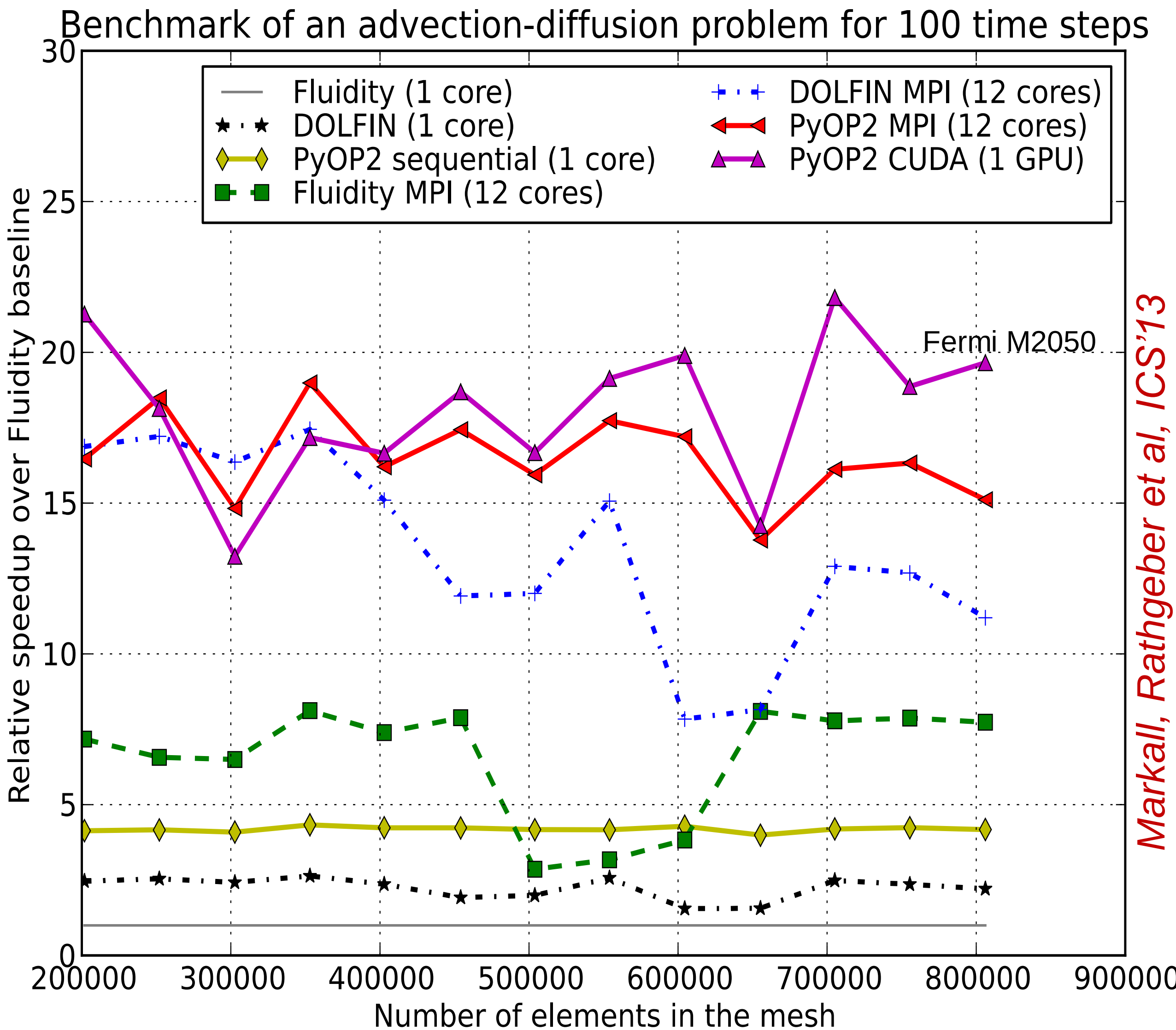
- This is the entire specification for a solver for an advection-diffusion test problem
- Same model implemented in FEniCS/Dolfin, and also in Fluidity – hand-coded Fortran

```
t=state.scalar_fields["Tracer"]         # Extract fields
u=state.vector_fields["Velocity"]       # from Fluidity

p=TrialFunction(t)                      # Setup test and
q=TestFunction(t)                       # trial functions

M=p*q*dx                                # Mass matrix
d=-dt*dfsvty*dot(grad(q),grad(p))*dx    # Diffusion term
D=M-0.5*d                               # Diffusion matrix

adv = (q*t+dt*dot(grad(q),u)*t)*dx      # Advection RHS
diff = action(M+0.5*d,t)                # Diffusion RHS

solve(M == adv, t)                      # Solve advection
solve(D == diff, t)                     # Solve diffusion
```

Firedrake – single-node performance

- Here we compare performance against two production codes solving the same problem on the same mesh:
  - Fluidity: Fortran/C++
  - DOLFIN: the FEniCS project's implementation of UFL

These results are preliminary and are presented for discussion purposes – see Rathgeber, Ham, Mitchell et al, http://arxiv.org/abs/1501.01809 for more systematic and up to date evaluation



- Graph shows speedup over Fluidity on one core of a 12-core Westmere node

- **Where did the domain-specific advantage come from?**
  - UFL (the Unified Form Language, inherited from the FEniCS Project)
    - Delivers spectacular expressive power
    - Reduces scope for coding errors
    - Supports flexible exploration of different models, different PDEs, different solution schemes
  - Building on PyOP2
    - Handles MPI, OpenMP, CUDA, OpenCL
    - Completely transparently
    - PyOP2 uses runtime code generation
    - So we don't need to do static analysis
    - **So the layers above can freely exploit unlimited abstraction**

Rathgeber, Ham, Mitchell et al, http://arxiv.org/abs/1501.01809

COFFEE: Optimisation of kernels

```
void helmholtz(double A[3][3], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W[3] = {...}
  static const double X_D10[3][3] = {{...}}
  static const double X_D01[3][3] = {{...}}

  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      for (int k = 0; k<3; k++)
        A[j][k] += ((Y[i][k]*Y[i][j]+
          +((K1*X_D10[i][k]+K3*X_D01[i][k])*(K1*X_D10[i][j]+K3*X_D01[i][j]))+
          +((K0*X_D10[i][k]+K2*X_D01[i][k])*(K0*X_D10[i][j]+K2*X_D01[i][j])))*
          *det*W[i]);
}
```

- Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange p = 1 elements.
- The local assembly operation computes a small dense submatrix
- These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE

Luporini, Varbenescu et al, ACM TACO/HiPEAC 2015

COFFEE: Optimisation of kernels

```
void helmholtz(double A[3][3], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W[3] = {...}
  static const double X_D10[3][3] = {{...}}
  static const double X_D01[3][3] = {{...}}

  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      for (int k = 0; k<3; k++)
        A[j][k] += ((Y[i][k]*Y[i][j]+
          +((K1*X_D10[i][k]+K3*X_D01[i][k])*(K1*X_D10[i][j]+K3*X_D01[i][j]))+
          +((K0*X_D10[i][k]+K2*X_D01[i][k])*(K0*X_D10[i][j]+K2*X_D01[i][j])))
          *det*W[i]);
}
```

- Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange p = 1 elements.
- The local assembly operation computes a small dense submatrix
- These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE

Luporini, Varbenescu et al, AC TACO/HiPEAC 2015

COFFEE: Optimisation of kernels

```
void helmholtz(double A[3][4], double **coords) {
 #define ALIGN __attribute__((aligned(32)))
 // K, det = Compute Jacobian (coords)

 static const double W[3] ALIGN = {...}
 static const double X_D10[3][4] ALIGN = {{...}}
 static const double X_D01[3][4] ALIGN = {{...}}

 for (int i = 0; i<3; i++) {
  double LI_0[4] ALIGN;
  double LI_1[4] ALIGN;
  for (int r = 0; r<4; r++) {
   LI_0[r] = ((K1*X_D10[i][r])+(K3*X_D01[i][r]));
   LI_1[r] = ((K0*X_D10[i][r])+(K2*X_D01[i][r]));
  }
  for (int j = 0; j<3; j++)
   #pragma vector aligned
   for (int k = 0; k<4; k++)
    A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j]+LI_1[k]*LI_1[j])*det*W[i]);
 }
}
```

- Local assembly code for the Helmholtz problem after application of
  - padding,
  - data alignment,
  - Loop-invariant code motion
- In this example, sub-expressions invariant to j are identical to those invariant to k, so they can be precomputed once in the r loop

Luporini, Varbenescu et al, AC TACO/HiPEAC 2015

Imperial College London

Kernels are often a lot more complicated

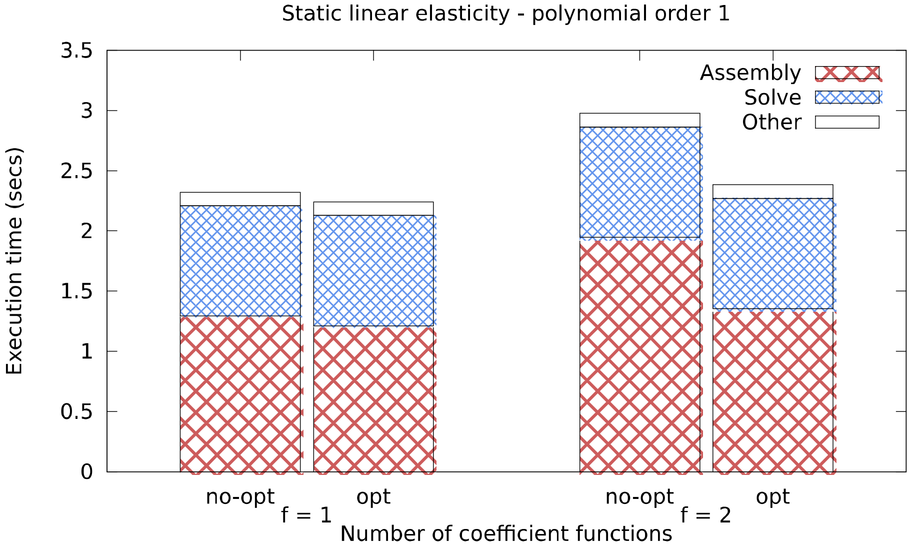
```
void burgers(double A[12][12], double **coords, double **w) {
 // K, det = Compute Jacobian (coords)

 static const double W[5] = {...}
 static const double X1_D001[5][12] = {{...}}
 static const double X2_D001[5][12] = {{...}}
 //11 other basis functions definitions.
 ...
 for (int i = 0; i<5; i++) {
  double F0 = 0.0;
  //10 other declarations (F1, F2,...)
  ...
  for (int r = 0; r<12; r++) {
   F0 += (w[r][0]*X1_D100[i][r]);
   //10 analogous statements (F1, F2, ...)
   ...
  }
  for (int j = 0; j<12; j++)
   for (int k = 0; k<12; k++)
    A[j][k] += (..(K5*F9)+(K8*F10))*Y1[i][j])+
     +(((K0*X1_D100[i][k])+(K3*X1_D010[i][k])+(K6*X1_D001[i][k]))*Y2[i][j]))*F11)+
     +(..((K2*X2_D100[i][k])+...+(K8*X2_D001[i][k]))*((K2*X2_D100[i][j])+...+(K8*X2_D001[i][j]))..)+
     + <roughly a hundred sum/muls go here>)..)*
     *det*W[i]);
 }
}
```

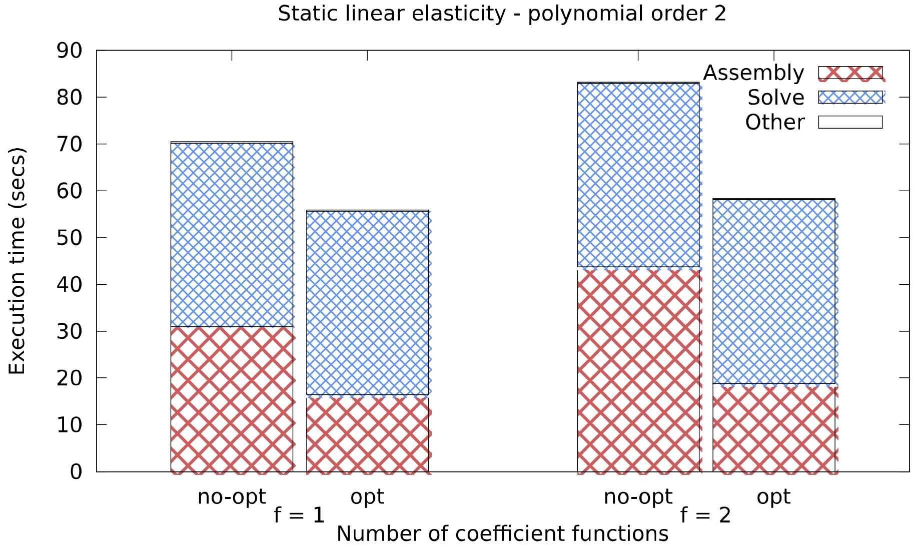
- Local assembly code generated by Firedrake for a Burgers problem on a 3D tetrahedral mesh using Lagrange p = 1 elements
- Somewhat more complicated!
- Examples like this motivate more complex transformations
- Including loop fission

Luporini, Varbenescu et al, AC TACO/HiPEAC 2015

Imperial College London

COFFEE: Performance impact





Luporini, Varbenescu et al, AC TACO/HiPEAC 2015

- Fairly serious, realistic example: static linear elasticity, p=2 tetrahedral mesh, 196608 elements
- Including both assembly time and solve time
- Single core of Intel Sandy Bridge
- Compared with Firedrake loop nest compiled with Intel's icc compiler version 13.1
- At low p, matrix insertion overheads dominate assembly time
- At higher p, and with more coefficient functions (f=2), we get up to 1.47x overall application speedup

- **Where did the domain-specific advantage come from?**
  - Finite-element assembly kernels have complex structure
  - With rich loop-invariant expression structure
  - And simple dependence structure
  - COFFEE generates C code that we feed to the best available compiler
  - COFFEE's transformations make this code run faster
  - COFFEE does not use any semantic information not available to the C compiler
    - But it does make better decisions
    - For the loops we're interested in
    - **For the linear operators arising in finite-element assembly we can show that it's possible to *minimise* the inner-loop flop count**

Luporini, Ham, Kelly: An algorithm for the optimization of finite element integration loops, in revision

Conclusions (but wait…)

- Pointers lead to the compiler making conservative decisions
  - Idea: capture the key data structures at a higher level of abstraction
    - Let the tools "own the data" – and control its distribution
    - "inspector-executor" – take time to derive a schedule from the specific mesh at runtime
- Your compiler doesn't know things that you know
  - That you will iterate over the mesh *many* times without changing it
  - That the graph is easily-partitionable and colourable
- Your compiler won't do optimisations that *we* know are good for your code
  - Policy vs mechanism – good for *your* code might not be good *in general*
- Runtime code generation is liberating
  - **We do not try to do static analysis on client code**
  - **We encourage client code to use powerful abstractions**

Challenge 1/3: Domain-specific optimisations

- Where do DSO opportunities come from?
  - Domain semantics (eg in SPIRAL)
  - Domain expertise (eg we know that inspector-executor will pay off)
  - Domain idiosyncracies (eg for GLICM)
  - Transforming at the right representation
    - Eg fusing linear algebra ops instead of loops
  - Data abstraction (eg AoS vs SoA)
    - Or whether to build the global system matrix (or instead to use a matrix-free or local-assembly scheme)

  - **How can we engage with the application specialists to expose and automate domain-specific optimisations?**

Challenge: 2/3: PyOP2 layer

- The key idea in OP2/PyOP2 is ***access descriptors***
- OP2's access descriptors are ***declarative specifications*** of how each loop iteration is connected to the abstract mesh
- The kernels do not access the mesh
- The implementation is responsible for connecting the kernel to the data
- The implementation is free to select layout, stage data, schedule loops
  - We can map from data to iterations

- **What *would* a programming abstraction for data locality look like?**

Challenge: 3/3: Firedrake layer

- Dramatically raised level of abstraction
- But we still can match or exceed hand-coded, in-production code
- Costs of abstraction are eliminated by dynamic generation of code specialised to context
- Domain-specific optimisations can yield big speedups over the best available general-purpose compilers

- **The real payoff lies in supporting the users in navigating freely to the best way to model their problem**

- **How can the *barriers to adoption* of DSLs be overcome?**

Imperial College London

Acknowledgements

Partly funded by

- NERC Doctoral Training Grant (NE/G523512/1)
- EPSRC “MAPDES” project (EP/I00677X/1)
- EPSRC “PSL” project (EP/I006761/1)
- Rolls Royce and the TSB through the SILOET programme
- EPSRC “PAMELA” Programme Grant (EP/K008730/1)
- EPSRC “PRISM” Platform Grant (EP/I006761/1)
- EPSRC “Custom Computing” Platform Grant (EP/I012036/1)
- AMD, Codeplay, Maxeler Technologies

- Code:
  - http://www.firedrakeproject.org/
  - http://op2.github.io/PyOP2/