PyOP2: A performance portable unstructured mesh framework

Graham Markall, Florian Rathgeber, Nicolas Loriant, Georghe-Teodor Bercea, David Ham, Paul Kelly – Imperial College London
Lawrence Mitchell – EPCC, Edinburgh
Mike Giles, Gihan Mudalige – Oxford University
Istvan Reguly – Pazmany Peter Catholic University

- **Performance portability:** platform-agnostic performance without source code changes
- **It is essential for performance portability that both a kernel and its call site are generated**
  - GPU: Kernel call, shared memory staging
  - CPU: AVX vectorisation, data movement

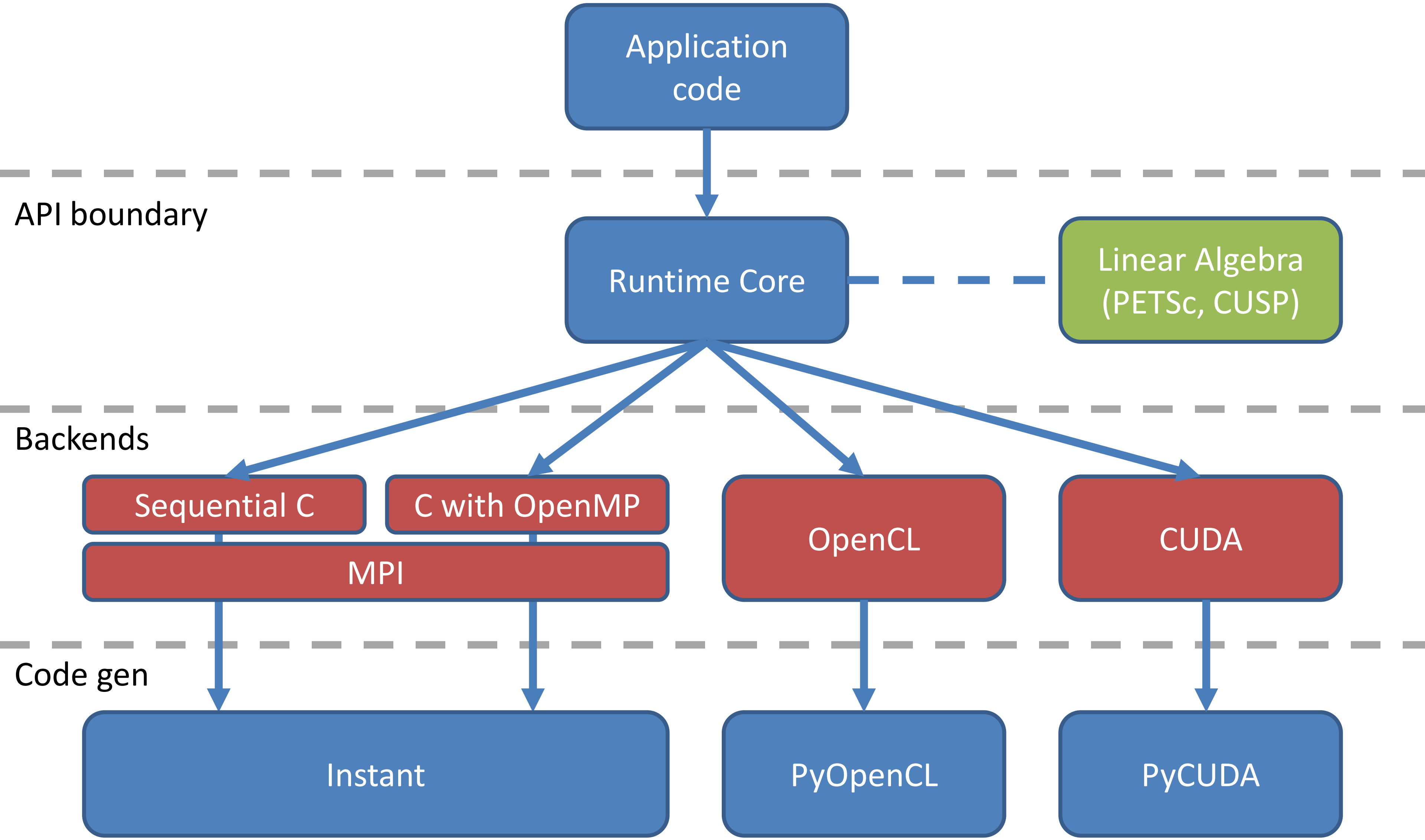
PyOP2

- Driving application: finite element assembly
- Hardware-specific performance optimisations in the form compiler breaks modularity

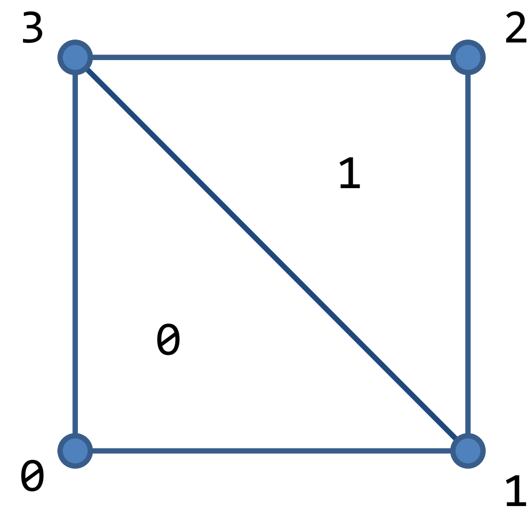
- Based on OP2 – static-compiled C++ API

- Python re-implementation
  - JIT Compilation
  - Linear algebra
  - Iteration spaces

PyOP2 Overview



Data declarations




```
dofs      = op2.Set(4)
cells     = op2.Set(2)
cell_dof  = op2.Map(cells, dofs, 3,
                    [ 0, 1, 3, 2, 3, 1 ])

dof_vals  = op2.Dat(dofs, 1,
                    [0.0, 0.0, 0.0, 0.0])
cell_vals = op2.Dat(cells, 1, [1.0, 2.0])

sparsity = op2.Sparsity([(cell_dof, cell_dof)])
mat      = op2.Mat(sparsity)
```

Kernel and parallel loop

```
user_kernel = op2.Kernel(“”””
void kernel(double *dof_val, double *cell_val) {
  for (int i=0; i<3; i++)
    dof_val[i] += *cell_val;
}”””””, “kernel”)

op2.par_loop(user_kernel, cells,
             dof_vals(cell_dof, op2.INC),
             cell_vals(op2.IdentityMap, op2.READ))
```

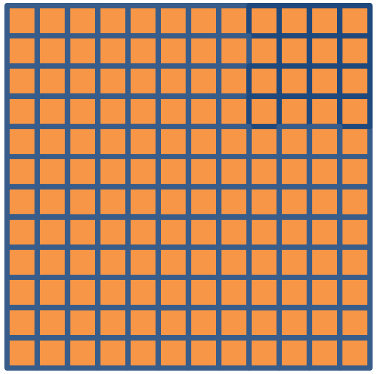
Iteration spaces – Design + API

- Entry-to-thread mapping should be handled by the runtime - **not** the user kernel
- Define user kernel in terms of one matrix entry

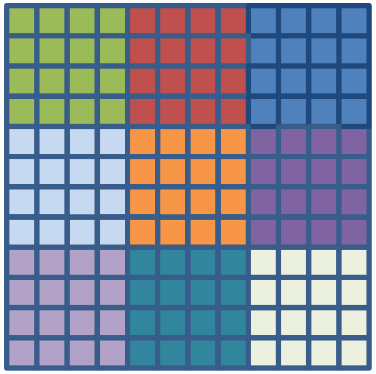
```
op2.par_loop(kernel, cells(3,3),
             mat(cell_dof[op2.i[0]], cell_dof[op2.i[1]]),
             *args)

op2.par_loop(kernel, cells(12,12),
             mat(cell_dof[op2.i[0]], cell_dof[op2.i[1]]),
             *args)
```

Iteration spaces - motivation

144 entries
Multiple matrices
Per thread



144 entries
1 thread per tile
What should tile size be?



```
void user_kernel(...) {
for (ele=TID/4; ele<n; ele<n/4)
  for (i=0; i<12; i++)
    for (j=0; j<12; j++)
      A[i,j] += ...
}
```

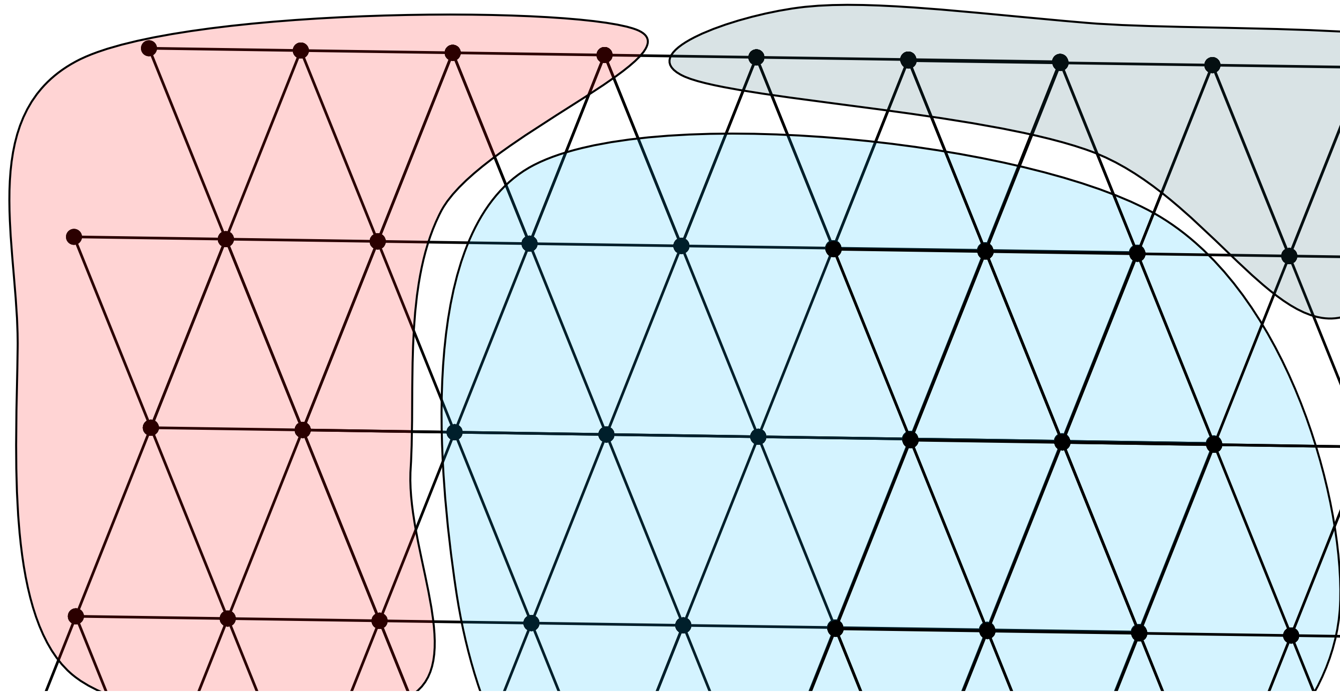
```
void user_kernel(...) {
for (ele=TID/9; ele+=NT/9; ele<n)
  patch_i = TID%3;
  patch_j = (TID%9)/3;
  for (i=0; i<4; i++)
    for (j=0; j<4; j++)
      A[patch_i*4+i, patch_j*4+j]
        += ...
}
```

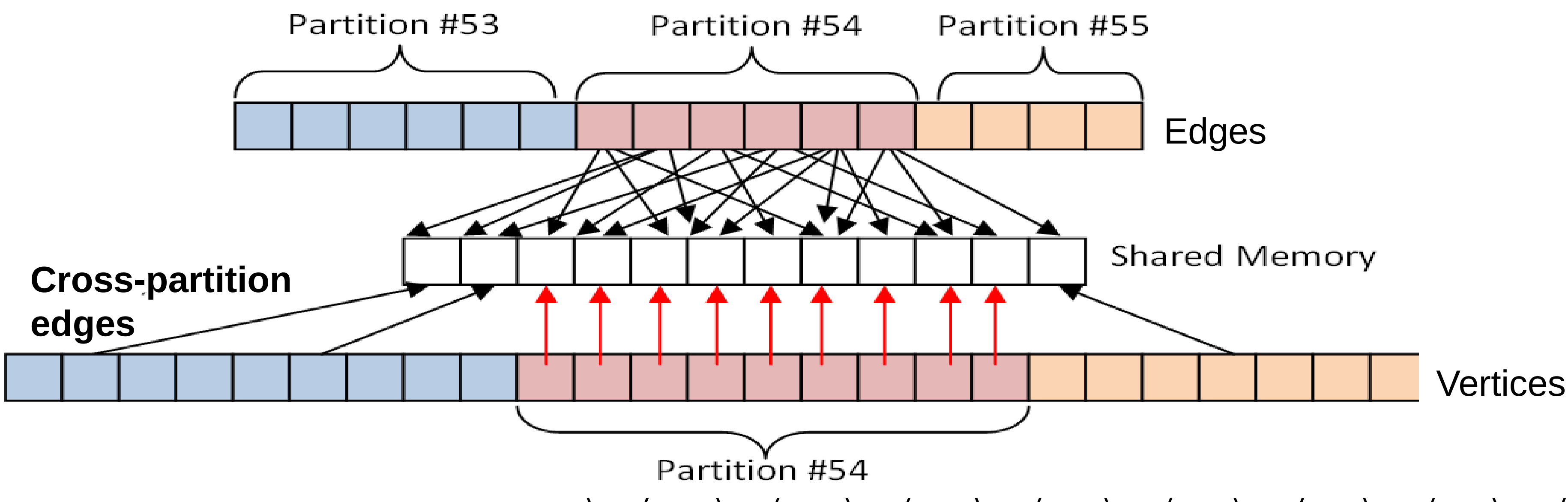
Iteration spaces – code generation

```
user_kernel(..., int i, int j) { A[i,j] += ... }

for (ele=TID/3; ele+=NT/3; ele<n)
  patch_i = TID%3;
  patch_j = (TID%9)/3;
  for (i=0; i<4; i++)
    for (j=0; j<4; j++)
      ki = patch_i*4 + i; kj = patch_j*4 + j;
      user_kernel(..., ki, kj);
      addto(matrix, ki, kj, ele)

for (ele=TID; ele+=NT; ele<n)
  for (i=0; i<12; i++)
    for (j=0; j<12; j++)
      user_kernel(..., i, j)
      addto(matrix, i, j, ele)
```

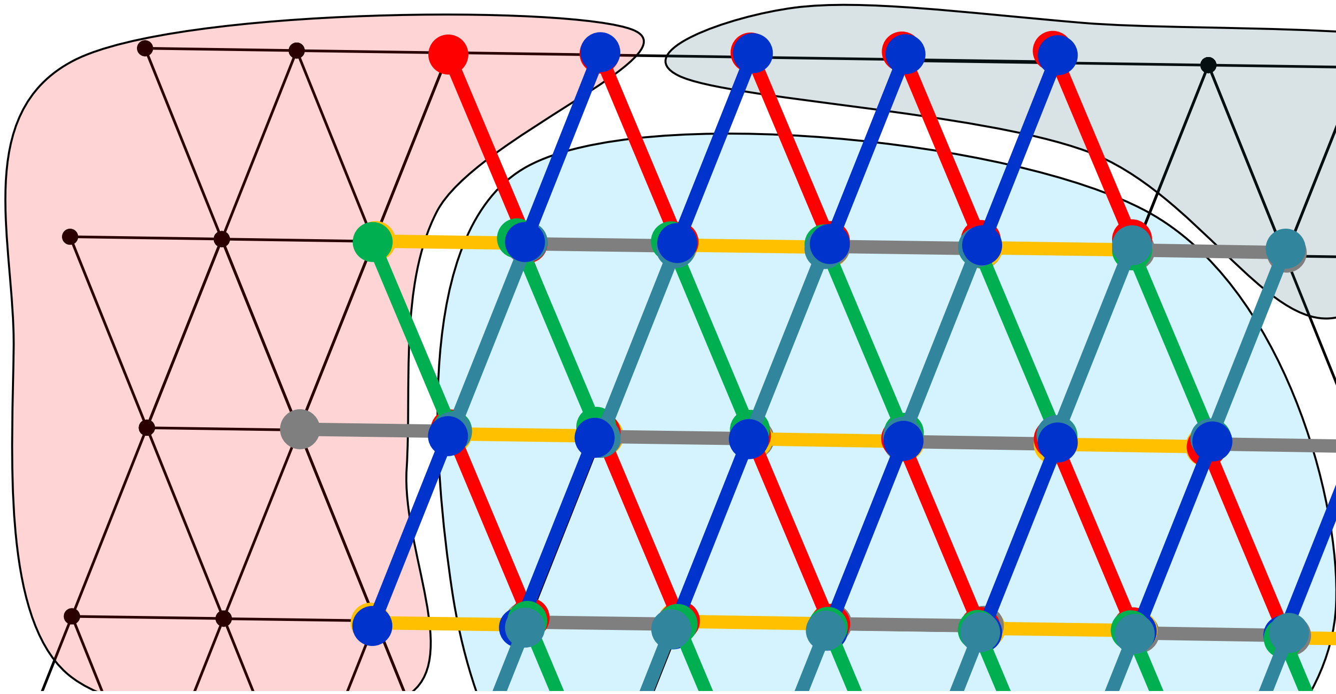
Parallel Execution

- Two key optimisations:
- **Partitioning**
- Colouring

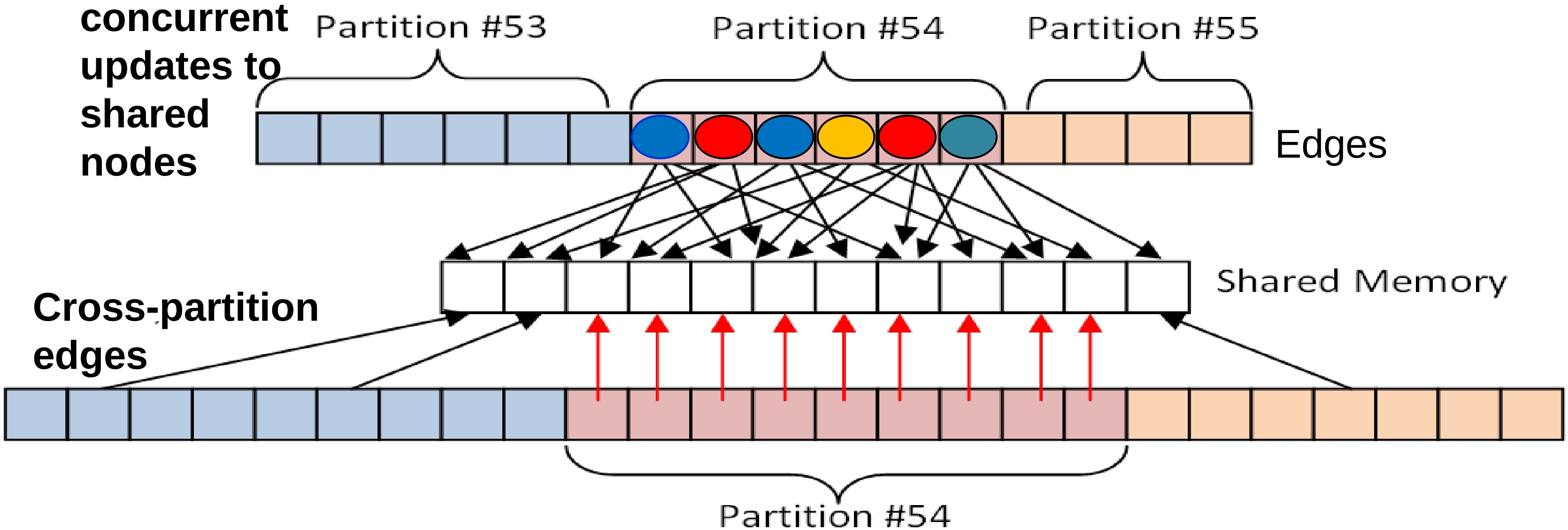




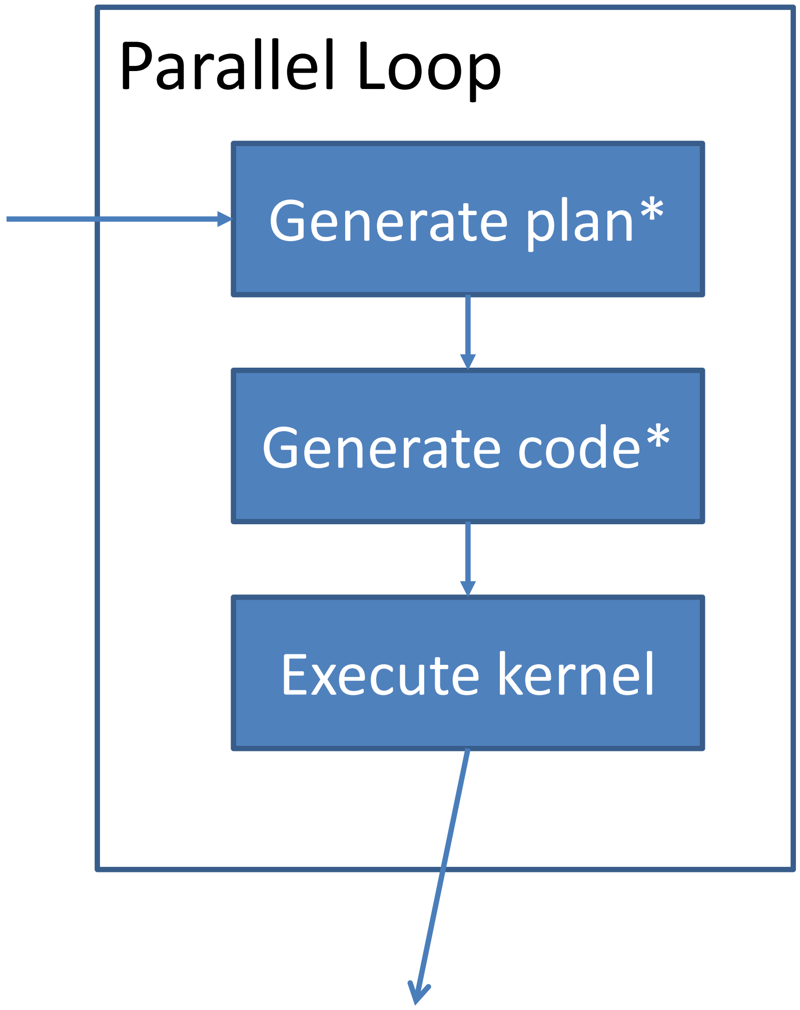
Parallel Execution



- Two key optimisations:
- Partitioning
- **Colouring**
  - **Elements of the edge set are coloured to avoid races due to concurrent updates to shared nodes**



Parallel execution



* Cached items

Summary

- PyOP2 takes control of the data layout,
- Generating data movement code, and
- Using freedom to manage the iteration space,
- it provides performance portability for unstructured mesh applications

In the future, will allow:

- AVX vectorisation for CPU
- Multi-GPU support with CUDA+MPI

Spare/unused slides

Colouring

```
__device__ user_kernel(args...) { ... }

__global__ wrap_user_kernel__(args) {
  for (partition=0; partition<np; partition++) {
  /* Stage in data for partition */
    for (col=0; col<ncol; col++) {
      for (i=0; itspace_i; i++)
        for (j=0; itspace_j; j++)
          user_kernel(..., i, j);
    }
  /* Stage out data for partition */
}
```

```
for col in xrange(plan.ncolors):
    # PyCUDA kernel launch
    fun.prepared_async_call(grid_size, block_size,
                            stream, *arglist,
                            shared_size=shared_size)
```

API

- Data declarations:
  - **Sets**: vertices, edges, cells etc.
  - **Dats**: data on sets – pressure, velocity
  - **Maps**: represent connectivity – cells → vertices
  - **Sparsities**: matrix structure
  - **Mats**: matrix data
- Parallel execution:
  - **Kernel** definition
  - **Parallel loop** invocation

Data declarations

- Runtime free to manage the data structures
- User is prevented - freed – from having to manage data
- Numpy array wrapping – can get accessor when necessary

Kernel and parallel loop

- Kernels computation for a single set element
- Par loop traverses set in any order
- Dat arguments accessed:
  - Directly, with the identity map
  - Indirectly, through a map
  - READ, WRITE, RW
  - INC, MAX, MIN

CUDA/OpenCL Execution

- Coalescing
- Little opportunity on unstructured meshes
- Staging into shared memory used instead

Parallel Execution

- Two key optimisations:
- Partitioning
- **Colouring**
  - At two levels