Software Abstractions for High Performance Mesh-based Simulations

Connor Ward

November 25, 2022

1 Introduction

In scientific computing, the composition of appropriate software abstractions is essential for scientists to write portable and performant simulations in a productive way (the three P's). Having suitable abstraction layers allows for a separation of concerns whereby numericists can reason about their problem from a purely mathematical point-of-view and computer scientists can focus on optimising performance. Each discipline is presented with a particular interface from which the problems of interest can be expressed in the clearest possible way, facilitating rapid code development.

When it comes to writing software there are effectively three choices of approach. For many problems, generic library interfaces introduce too much overhead to be viable options for writing programs. Similarly, hand-written codes, though extremely fast, require a substantial effort to maintain and extend and the codebase can be very large. Code generation is an appealing solution to these problems. Given an appropriate abstraction, high-performance code can be automatically generated, compiled and run. This offers an advantage over library interfaces because problem-specific information can be exploited to generate faster code (e.g. commonly used operations can be memoized for fast lookups), and the task of actually writing the code is offloaded to a compiler rather than being hand-written. With a code generation framework, the key questions now become: "What is an appropriate abstraction for capturing all of the behaviour I wish to model?", and "What performance optimisations are nicely expressed at this layer of abstraction?"

In this work, we present `pyop3`, a library for the fast execution of mesh-based computations over some local stencil. In accordance with the principles described above, `pyop3` deals mainly in 3 abstractions: Firstly, the user interface is motivated by the fact that many operations relating to the solution of partial differential equations (PDEs) can be expressed as the operation of some 'local' kernel over a set of entities in the mesh where only functions with non-zero support on this entity are considered in the calculation. A classic example of this sort of calculation occurs with finite element assembly where the cells of the mesh are the iteration set and the kernel uses degrees-of-freedom (DoFs) from the cell and enclosing edges and vertices. This first abstraction layer therefore presents an interface to the user where they may straightforwardly express the operation of local kernels within loops over mesh entities, specifying the requisite restrictions for the data. The second abstraction, intended as an internal representation for the developer, describes the data layout and the third is a polyhedral loop model that is the target for the code generation.

The rest of this paper is laid out as follows: In Section **??** we review existing stencil libraries and mesh abstractions as well as strategies and details for code performance optimisation. Section 3 discusses the design of `pyop3`, Section 4 then reviews some possible extensions that might be pursued in future. Some concluding remarks are made in Section 5.

2 Background

In this section we review existing software abstractions for mesh computations and discuss common strategies for optimising performance.

2.1 Stencil languages

Given the ubiquity of stencil operations in simulations, a number of libraries exist providing convenient interfaces for stencil applications.

Ebb, Simit and Liszt follow the approach of providing a domain-specific language for the expression of stencil problems... They all provide high performance execution on GPUs as well as CPUs.

OP2 is another approach [21, 20]. Rather than using a domain-specific language, OP2 provides a simple API to the user for specifying the problem. The key entities in the OP2 data model are: sets, data on sets, mappings between sets, and operations applied over these sets. Having provided these inputs, the OP2 compiler is then called and transforms to source code to a high performance implementation of the traversal for a specific architecture.

Another library for the application of stencil operations, specifically high-order matrix-free kernels for the finite element method (FEM), is libCEED.

2.1.1 PyOP2

PyOP2 is a domain-specific language for expressing computations over unstructured meshes. It is the direct precursor, and inspiration for, `pyop3`.

It distinguishes itself from OP2, a library depending on the same abstractions, by using run-time code generation instead of static analysis and transformation of the source code.

PyOP2's mesh abstraction is formed out of the following main components:

In PyOP2, data is defined on *sets* and these are related to one another using *mappings*. Importantly, this abstraction does not contain any concept of the underlying mesh and instead all of the required information is encoded in the maps.

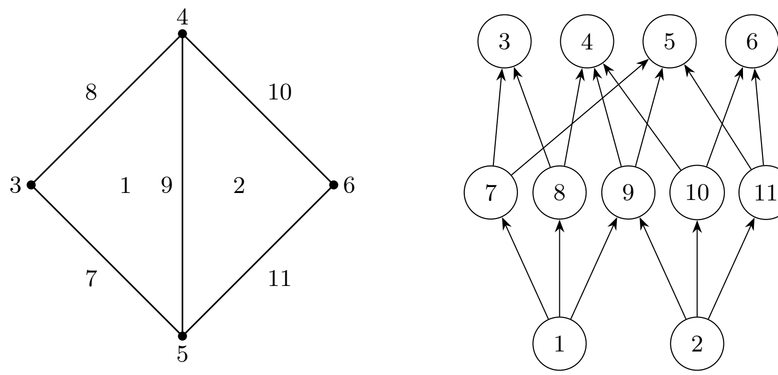


Figure 1: An example mesh and its Hasse diagram representation.

Computations over the mesh are expressed as the execution of some local kernel over all entities of some iteration set via a construct called a parallel loop, or *parloop*. The kernel is written using Loopy, a library for expressing array-based computations in a platform-generic language [15]. This intermediate representation allows for interplay between the local kernels enabling optimisations such as inter-element vectorisation [27].

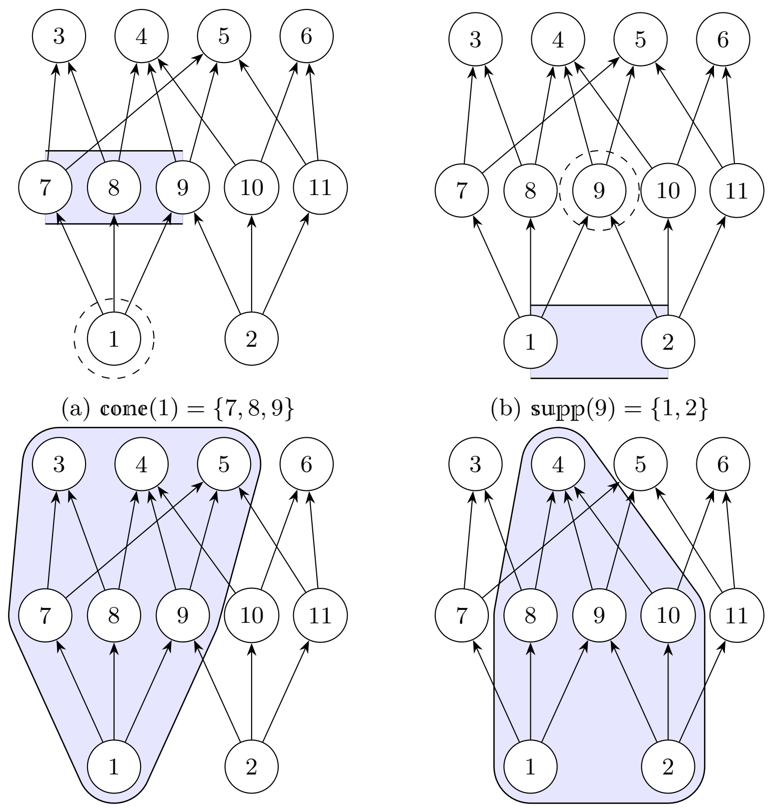
At present, PyOP2 only works on distributed memory, CPU-only systems (although some work has been done to permit execution on GPUs [16]). During the execution of a parloop, each rank works independently on some partition of the mesh. To avoid excessive communication between ranks, each rank has a narrow *halo* region that overlaps with neighbouring ranks that is executed redundantly. The halos are split into *owned*, *exec*, and *non-exec* regions to indicate the data's origin and the communication direction between the neighbouring processes.

2.2 Mesh representations

In software, a mesh is typically represented by a collection of sets of entities (e.g. cells or faces), coupled with adjacency relations between these sets. Possible abstractions capturing this behaviour include databases (ebb, moab) or hypergraphs (simit). In this work we focus on DMPlex, the unstructured mesh abstraction used in PETSc. In contrast with Ebb, Simit or Liszt, DMPlex is a more general purpose mesh abstraction and so has a more substantial feature set.

2.2.1 DMPlex

In DMPlex, the mesh is represented as a *CW-complex*, an object from algebraic topology that describes some topological space. In such a complex, all topological entities (e.g. cells, vertices) are simply referred to as *points* and the connectivity of the mesh can be expressed as the edges of a directed acyclic



(a) $\mathbf{cone}(1) = \{7, 8, 9\}$

(b) $\mathbf{supp}(9) = \{1, 2\}$

(c) $\mathbf{cl}(1) = \{1, 3, 4, 5, 7, 8, 9\}$

(d) $\mathbf{st}(4) = \{4, 8, 9, 10, 1, 2\}$

Figure 2: The possible DMPlex restriction queries (applied to the Hasse diagram from Figure 1).

graph (DAG) with the vertices being the points of the mesh. More specifically, the points and relations form a partially-ordered set (poset) such that the mesh can be visualised using a Hasse diagram (Figure 1).

It is important to note that DMPlex works for arbitrary dimension.

Stencil queries are natural to express at this level. For instance, the classical finite element request of "give me all of the DoFs that have local support" is simply expressed as the closure of a given cell. Another example useful for finite volume calculations: "what are my neighbouring cells?" is supp(cone(c)). One can also do clever patch things.

With DMPlex, parallel vectors are created by associating a DMPlex mesh with a `PetscSection`. A `PetscSection` is a simple object that tabulates offsets such that entries in an array may be addressed.

2.3 Methods for performance optimisation

In this section, we review some of the common bottlenecks in massively parallel simulation codes and describe some general ways for quantifying and improving performance. In particular, we focus on challenges for maximising parallel efficiency and the importance of the roofline model for choosing appropriate optimisations.

2.3.1 Achieving parallel efficiency

In order to run massive simulations, codes must be able to exploit the vast amounts of parallelism afforded to them by modern supercomputers. With the building of ever larger and more parallel machines (especially since we are at the "dawn of exascale"), this is becoming both more important to get right and more challenging to do so.

To quantify a code's effectiveness in parallel, one typically measures its *parallel efficiency* under either *strong-* or *weak-scaling*:

Strong-scaling Strong-scaling describes the behaviour of a code as the number of processes increases for a problem of *fixed size*. In a strong-scaling investigation, perfect efficiency (unity) would be achieved if the time-to-solution on p processors was p times smaller than the time-to-solution for a single process. A code would be considered to have 'good' strong-scaling if it retained high efficiency (e.g. 80%) at small problem sizes (e.g. 5000 DoFs per process for FEM).

If the efficiency is low, this suggests that there are sizeable portions of the code that are getting run in serial on each process, rather than being shared across all processes. To improve efficiency, therefore, one should focus on either reducing this overhead or distributing the work more effectively between processes.

Weak-scaling Weak-scaling differs from strong-scaling in that, rather than describing the decrease in time-to-solution for a problem of fixed size, it describes the behaviour of the code over a *range of problem sizes*, where the size of the problem scales linearly with the number of processors. In this case, perfect efficiency is achieved if the time-to-solution remains fixed with increasing parallelism. For a code to have 'good' weak-scaling, it would need to have a high efficiency (e.g. 80%) even when run on very large clusters.

If a code has poor weak-scaling, this suggests that there are algorithmic problems regarding how the problem is distributed among processors. For example, a parallel algorithm that required frequent all-to-all broadcast messages would have poor weak-scaling because this would increase in cost with the number of processors.

Taken together, these two metrics provide a relatively good indicator as to the suitability of a code for running on massively parallel computers. More informative measures of performance that take into account things such as convergence rates also exist [6], but we eschew such approaches here because they fall under the remit of the design of the stencil itself, which is not the focus for this work.

2.3.2 Maximising floating-point throughput

Once we have a code that scales well, the next step is to optimise performance for a single process. In order to do this, one should first profile the code and compare its performance to the theoretical limits of the hardware using a *roofline plot* [30]. A roofline plot provides both a good termination criterion for the optimisation process - you can't go faster than the hardware! - and also guides the developer as to what the performance bottlenecks might be and thus which optimisations might be usefully applied.

To begin with, one needs to measure the performance-critical kernels in the application to determine their floating-point throughput (the higher the better) and *arithmetic intensity*. Arithmetic intensity is a ratio of the number of floating-point operations (FLOPs) performed per byte of memory accessed and it indicates whether or not a kernel is likely to be *memory-bound*, where maximum throughput is limited by memory access speeds, or *compute-bound*, where it is limited by the speed of the chip itself. In general, operations such as square roots and divisions typically require a large number of FLOPs, yielding a high arithmetic intensity, while simple streaming access to an array (e.g. `z[i] = x[i] + y[i]`) read a large amount of data compared to the number of FLOPs performed and have a correspondingly low arithmetic intensity.

Having recorded the floating-point throughput and arithmetic intensity, the kernels can now be added to a roofline plot. An example plot is shown in Figure 3. To the left of the plot, where arithmetic intensity is low, throughput is dependent upon the rate at which data can be supplied to the chip and so the ceilings are given by the bandwidths of the various levels in the memory hierarchy. By contrast, to the right the arithmetic intensity is high and so

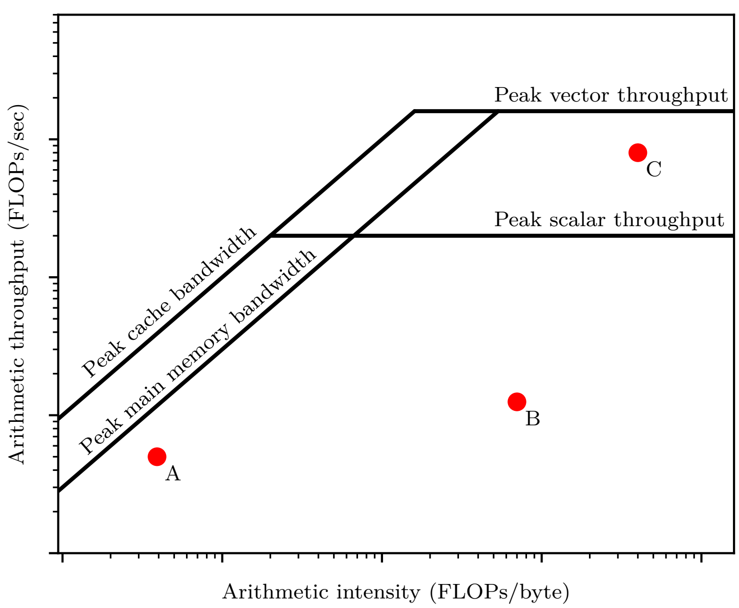


Figure 3: A simplified roofline model. The points A, B and C represent computational kernels with different performance characteristics.

the performance limits are prescribed by the peak floating-point throughput of the chip. Two lines are shown to demonstrate the fact that vectorisation can increase performance over standard scalar operations.

Looking at the example kernels shown in red in the figure, we can demonstrate the utility of roofline plots in the optimisation effort:

- Kernel A has low arithmetic intensity and so the maximum achievable throughput is well below theoretical peak. It lies fairly close to the main memory bandwidth ceiling and so fruitful optimisation efforts could include: modifying the data access patterns to better utilise cache memory, or making algorithmic modifications to increase the arithmetic intensity.

- Kernel B has high arithmetic and so should be able to achieve close to peak performance. It currently sits well below the theoretical limits and is therefore an excellent candidate for further optimisation. Since the code cannot be memory-bound, optimisation efforts should focus on compute-type optimisations such as loop unrolling, rather than data ones.

- Lastly, kernel C already achieves very close to peak throughput. It is not a good candidate for optimisation since the potential for performance improvements is low.

Within the context of stencil computations, the key observation that can be made regarding the classification of kernels into being either compute- or memory-bound is that, in the main, only memory-bound optimisations are worthwhile pursuing. This is because, if compute-bound, the vast majority of the code runtime will be spent inside the stencil computation, and hence all of the 'hot loops' that impact performance will be found inside it. On the other hand, if we are memory-bound, then most of the time will be spent inside the packing and unpacking code of the stencil library and there are genuine opportunities for optimisation. Therefore, in general, optimisations for stencil languages should focus on improving memory accesses.

We remark briefly that notable exceptions to this rule are inter-element vectorisation and GPU offloading. In both cases this would be implemented in a stencil library by, instead of looping over the iteration set one at a time, looping over multiple entities simulatenously, one per vector lane/GPU thread. This can improve the performance of compute-bound codes by allowing the chip to execute multiple operations per clock cycle.

In the case of inter-element vectorisation, in Figure 3 this would correspond to shifting the kernel from under the "Peak scalar throughput" ceiling to the "Peak vector throughput" one. Such an approach has been implemented in (a branch of) PyOP2, and demonstrated to be performant [27]. Similarly, some preliminary work on adding GPU offloading support to PyOP2 is ongoing [16].

2.4 Optimisations for stencil computations

2.4.1 Locality optimisations

From the roofline in Figure 3 one can see that, for a memory-bound code (low arithmetic intensity), dramatic speedups may be achieved by changing the level in the memory hierarchy at which the kernel operates. In this simplified figure this corresponds to being limited by "Peak cache bandwidth" instead of "Peak main memory bandwidth" (in reality chips have multiple layers of cache memory with different capacities and performance characteristics).

Cache levels have a small capacity, and so in order to exploit the faster data accesses the *working-set size* of the problem must be reduced to fit inside a particular cache level. The working-set size describes the amount of data that must be on hand to perform a computation. If this data volume exceeds the capacity of a cache level then memory performance will be driven by the cost of retrieving data from the next-fastest, next-smallest level of the memory hierarchy. Memory access speeds can differ by an order of magnitude between levels and so the working-set size is a primary consideration for memory-bound applications. One of the key ways to reduce the working-set size is to maximise *data locality*.

To try and minimise the number of accesses made to main memory, caches assume that application data exhibit both *spatial locality* and *temporal locality*. Spatial locality refers to the fact that if a particular piece of data is used by the application, then it is likely that neighbouring data will also be needed. Caches therefore load data in contiguous chunks termed *cache lines* so data adjacent to the target also get loaded into the cache. Temporal locality in caches is the assumption that loaded data may be needed multiple times. To exploit this, cache lines persist in the cache until eviction by new data.

To optimise performance, one must make *data locality* optimisations, that is, optimisations that ensure greater utilisation of the additional entries loaded per cache line as well as trying to avoid repeated loads of the same data if repeated accesses are required.

Common data locality optimisations for stencil-like applications include *tiling*, where a multi-dimensional iteration domain is subdivided into small *tiles* to exploit data reuse between the stencils [13, 22], and *kernel fusion*, where separate stencils are executed in a single loop to take advantage of the data shared between them [8]. *Time tiling* is a combination of the two where the iteration domain is tiled and then multiple stencils are applied in turn over each tile [19].. In the same way as kernel fusion this aims to exploit temporal locality by reusing data between the stencils.

For unstructured meshes, a common optimisation is to renumber the mesh entities such that all entities in the stencil are 'close', hence making use of spatial locality. It also helps with temporal locality since DoFs on faces will be reused between iterations. The reverse Cuthill-McKee (RCM) algorithm is a common 'cache-oblivious' way of reordering an unstructured mesh [7, 17].

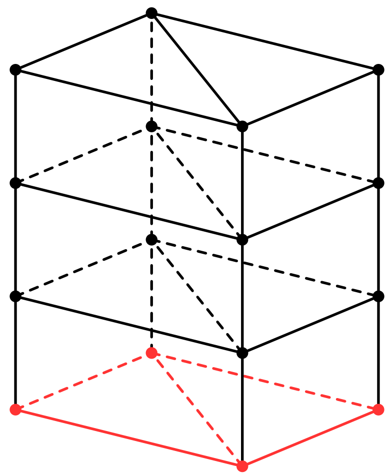


Figure 4: A sample extruded mesh formed by uniformly extruding a base mesh of two triangles (highlighted in red).

2.4.2 Exploiting mesh structure

As well as reordering data to reduce the *effective* working-set size, one can sometimes exploit the structure of the mesh itself to reduce the actual working-set size by reducing the volume of data needed per computation.

Meshes can (usually) be classified into one of two types: *structured* or *unstructured*. The fundamental difference between the two is that structured grids can determine their neighbouring entities without needing to tabulate everything individually. This means that data accesses have the form `x[f(i)]` instead of something like `x[map[i]]` that is required for unstructured meshes.

Structured meshes are known to have faster access to data than unstructured meshes for the following reasons:

- *Beneficial data layout*
  Since the data layout of a structured mesh is completely regular, we get a higher rate of cache hits as we iterate over the mesh. This is because there will be data reuse on the face where the two cells meet and also the hardware prefetcher will load cache lines for us.

- *Less memory is required per data access*
  Since lookup tables are not required for locating data, the total volume of data required from main memory is reduced. This reduces the working-set size of the problem.

- *Smaller working-set size*
  We need less data in order to compute a single stencil. If this volume exceeds the size of a particular level in the cache hierarchy then the next

one down needs to get used and this will have a bandwidth that is orders-of-magnitude slower.

- *Aids software prefetching*
  If the compiler can see that data is accessed in a particular pattern, it can emit prefetch instructions such that data will already be in the cache.

These benefits have motivated the design of *partially-structured* meshes, where only portions of the mesh are structured. Examples include: block-structured meshes, regularly refined meshes and extruded.

An example extruded mesh is shown in Figure 4. Users start with an unstructured 'base' mesh - here two triangles - that is *extruded* to produce layers of triangular prisms. The mesh has 'partial structure' in that the data layout is regular up each column, but irregular for the base mesh. As the number of layers increases, the cost of accessing extruded meshes has been shown to approach that of structured meshes [5].

Of the reasons for improved performance described above, we would like to emphasise that the first of these, the "beneficial data layout", is by far the most important, and in particular that the absence of indirection maps reducing the data volume has only a small effect.

The reason for this is that the difference in data volume between the meshes usually at most 25%. To demonstrate, consider a stencil code looping over the cells of a structured mesh where, for each cell, there are p points accessed and d DoFs per point. The total (minimum) amount of memory accessed is then given by

$$D_S = n_c \cdot p \cdot d \cdot 8 \cdot 2,$$

where n_c is the number of cells, the 8 comes from the fact that each DoF is 8 bytes, and the 2 is assuming that we read from one array and write to another. In the unstructured case, the data volume is the same as before plus the size of the indirection map:

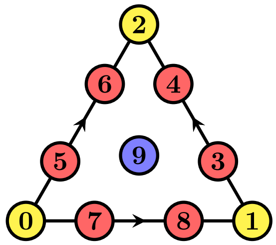
$$D_U = D_S + n_c \cdot p \cdot 4.$$

A factor of 4 is required instead of 8 because the maps are typically 4 byte integers. As a fraction of the structured case, the extra data required by the unstructured mesh is therefore given by

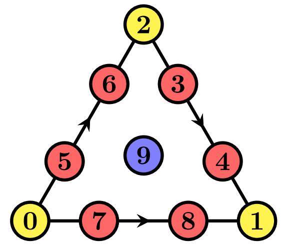
$$\frac{D_U - D_S}{D_S} = \frac{1}{4d},$$

which, since $d \geq 1$, bounds the extra data volume at 25% of the structured case.

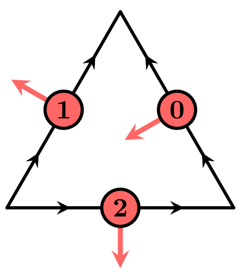
Taking into consideration that this represents a reasonable worst-case example - stencils frequently admit more than two data structures and often have more than one DoF per point - the performance benefits of direct addressing are not very dramatic compared with the possible order of magnitude improvements that one can get from a good data ordering. Though we do address adding support for partially-structured meshes in Section 4.1, initial work on `pyop3` does not focus on it.



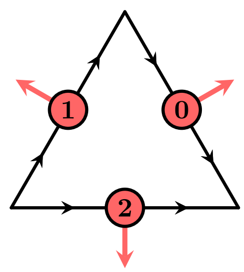
(a) Reference Lagrange finite element with polynomial degree 3.



(b) Degree 3 Lagrange finite element with an edge flipped.



(c) Reference Raviart-Thomas finite element.



(d) Raviart-Thomas element with an edge flipped.

Figure 5: The effect of edge flips on both scalar- and vector-valued finite elements on triangles. Cell, edge and vertex DoFs are shown in blue, red and yellow respectively.

2.5 Orienting degrees-of-freedom

When writing the local kernel for a stencil computation, the process is considerably simplified by assuming that the data being passed in from the iteration engine (here `pyop3`) is in some canonical ordering. Unfortunately, guaranteeing such an ordering a priori is often difficult as entities in the stencil may have different relative orientations. To give an example, with an arbitrary global numbering of vertices, it is possible for a mesh to contain cells with 'flipped' edges relative to their canonical orientation. If multiple DoFs are stored along the edge then this will result in reading them in the wrong order, breaking the code. An example of this for triangles is shown in Figure 5. Figure 5a shows the canonical ordering for a degree 3 Lagrange finite element. Note how flipping an edge (Figure 5b) results in a permutation of the stencil DoFs.

The situation is further complicated when the DoFs stored at each node are vector-valued, for example with $H(\mathrm{div})$ and $H(\mathrm{curl})$ conforming elements in FEM. With vector DoFs it can happen that, as well as possibly being out of order, the loaded vectors can 'point' in the wrong direction, requiring the application of some transformation to achieve a canonical representation. For example, in Figures 5c and 5d one can see that the effect of flipping an edge inverts the direction of the vector DoFs, so they must be multiplied by -1 to return to canonical. Things get even more difficult in 3D because vector-valued DoFs on facets can be defined relative to the two tangent vectors of the face. The transformation back to canonical therefore can require rotations (via 2×2 matrices) as well as flips.

2.5.1 A partial solution: Mesh renumbering

The approach used to 'fix' the orientation issue used in most finite element libraries is to exploit the fact that, for a variety of cell types, it is possible to determine a global numbering of the vertices of the mesh such that all cells have the same orientation. In particular this has been shown to work for simplices [24] and quadrilaterals [1, 11]. It is also possible to number unstructured hexahedral meshes in this manner, but the algorithm cannot be performed in parallel and it will not work for all possible meshes [1]. In particular, if a subset of the mesh forms a Möbius strip then it is not orientable.

The mesh renumbering approach is also unsuitable for other cell types (e.g. pyramids) or *mixed mesh* methods.

It is clear that a renumbering strategy is not a sufficiently general solution to the orientation problem. Therefore the stencil library abstraction needs to permit for different orientations.

2.5.2 The next step: Permuting scalar DoFs

As shown in Figures 5a and 5b, loading scalar data for cells with flipped edges will result in the data being packed in the wrong order. To avoid this, the DoFs need to be *permuted* prior to packing. This permutation can actually be precomputed and form part of the indirection maps used to address the stencils.

2.5.3 All the way there: Transforming vector DoFs

The above approach works for all scalar-valued function spaces, but is insufficient for vector-valued ones due to the need for transformations of the DoF values themselves. For example, as described above, the flip shown in Figures 5c and 5d mean that the DoFs need to be scaled by -1 prior to packing. It is not possible to store non-permutation transformations inside an indirection map and so the stencil application needs to implement a separate stage to resolve these orientation transformations. This is the approach used by Basix [26, 25], part of the FEniCSx finite element software suite [18, 2]. However, to our knowledge, this is not performed by any existing general purpose stencil library.

3 Implementation

As discussed in Section 2.1, existing stencil libraries may be classified according to whether or not they are aware of the mesh topology. A library that is not 'mesh-aware', for example PyOP2, can be more challenging to program

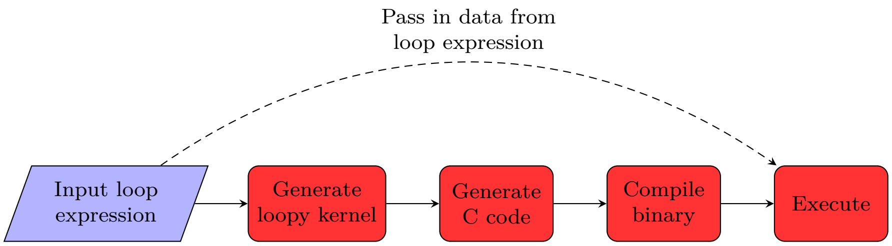


Figure 6: The code generation and execution pipeline for a `pyop3` loop expression.

```
do_loop(
  c := mesh.cells.index,
  kernel(dat0[closure(c)], dat1[closure(c)])
)
```

Listing 1: `pyop3` code declaring and executing a loop expression for finite element-type computations (since `closure` is used).

```
void do_loop(int ncells, double *dat0, double *dat1, int *map0) {
  double t0[CLOSURE_SIZE], t1[CLOSURE_SIZE];

  for (int c=0; c<ncells; ++c) {
    // Pack temporaries
    for (int p=0; p<CLOSURE_SIZE; ++p) {
      t0[p] = dat0[map0[c*CLOSURE_SIZE+p]];
      t1[p] = 0.0;
    }
    // Do the local computation
    kernel(t0, t1);
    // Now scatter the results
    for (int p=0; p<CLOSURE_SIZE; ++p) {
      dat1[map0[c*CLOSURE_SIZE+p]] += t1[p];
    }
  }
}
```

Listing 2: Simplified version of code that would be generated by `pyop3` for the loop expression from Section 3.1. `kernel` has access descriptors `READ` and `INC`, explaining the differing treatment for `t0` and `t1`. `CLOSURE_SIZE` is an integer constant and would be known at compile-time.

in because responsibility for reasoning about the topology, including orientations, is passed to the user who has to construct the appropriate indirection maps to represent their mesh. Composition of indirection maps is also difficult because of the loss of topological information. Such difficulties can be avoided with a mesh-aware stencil library, but (usually) at the expense of relying upon a custom mesh abstraction implementation. Writing mesh codes by hand is a non-trivial task requiring a significant amount of developer effort to maintain and introduce new features. An in-house implementation will also never be able to compete with the feature set of a more general purpose mesh library (e.g. I/O, parallel decomposition, adaptive refinement) and will also suffer from poor interoperability with other packages.

In this work we attempt to bridge the gap between mesh-aware and mesh-oblivious stencil libraries by writing a new stencil library, `pyop3`, that combines the advantages provided by mesh-aware frameworks with DMPlex, a mature, external mesh implementation (Section 2.2.1).

`pyop3` is, somewhat obviously, heavily inspired by and based upon PyOP2, and hence much of its design represents either an incremental improvement on PyOP2, or is in fact directly lifted from it. This report will make clear what parts of the design of `pyop3` are novel contributions and which are derived from PyOP2.

3.1 Interface

`pyop3` is implemented as a domain-specific language (DSL) embedded in Python. As part of a larger script users create *loop expressions*, prescribing the loops, local computations and stencil data access patterns in a manner resembling the algorithm's pseudocode. These loop expressions are then parsed, compiled and executed by the `pyop3` backend.

To give an example, the syntax for a typical FEM residual assembly, where one loops over cells and computes using data in the cell's closure, would look something like the code shown in Listing 1. Here the function `do_loop` declares and then executes the loop expression. All loop expressions consist of: 1) an iteration set, here `mesh.cells`, and 2) a sequence of instructions to execute, here simply the single local kernel `kernel`. `kernel` has type `LocalKernel` and consists of a *loopy kernel* (Section 3.1.1) augmented with *access descriptors* for each of its arguments. Possible access descriptors are `READ`, `WRITE`, `RW`, `INC`, `MIN` and `MAX` and they are required to ensure that `pyop3` generates the right packing/unpacking code. An example for a kernel with access descriptors `READ` and `INC` is shown in Listing 2. Lastly, the `dat{0,1}[closure(c)]` instructions indicate that `kernel` should be passed two contiguous arrays, each containing the DoFs associated with the closure of the current cell. This last part is where the difference between the mesh-oblivious PyOP2 and the mesh-aware `pyop3` is stark. In PyOP2 these closure maps would need to be computed in advance and passed in to the loop expression whereas `pyop3`, being mesh-aware, is capable of reasoning about the mesh and automatically computing the right indirection maps.

Note that in this example, by using `do_loop`, we declare the loop expression and then *immediately execute it*. This is not always desired behaviour as, if the same loop expression is executed multiple times, one may want to have a *persistent* loop expression to avoid some overhead. Such expressions can easily be created by calling `expr = loop(...)`, and then executed with Python call syntax: `expr(*args, **kwargs)`. This is discussed in more detail in Section 3.4.

3.1.1 Code generation

Having declared a loop expression, `pyop3` executes it by first lowering the expression through a sequence of intermediate representations, compiling to a binary, then running the code.

The target intermediate representation of `pyop3` is loopy [15], a polyhedral model inspired Python code generation library capable of generating code for multiple backends including CPUs and GPUs. With loopy, the main entry point is, not dissimilarly to `pyop3`, the declaration of a `LoopKernel` via the command `loopy.make_kernel`. Once a `LoopKernel` has been created, loopy also provides a wealth of different code transformations such as loop tiling, vectorisation and loop-invariant code motion.

To construct such a kernel, the user needs to specify *domains*, effectively loop extents, *instructions* (e.g. `x[i] += y[i]`) and *arguments* representing kernel inputs. For example, a simple loopy kernel could be created as follows:

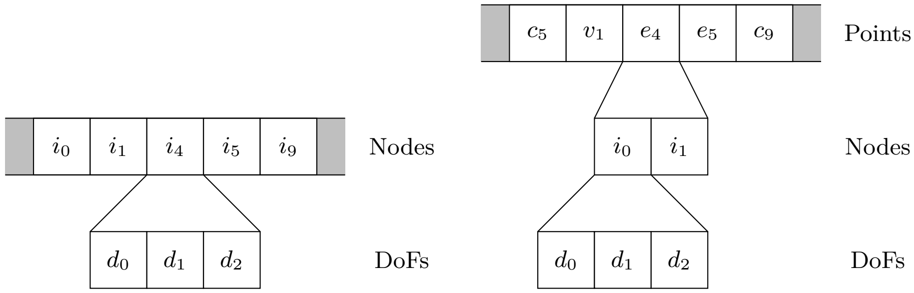
```
knl = loopy.make_kernel(
  "{ [i]: 0 <= i < n }",  # domains
  "y[i] = 2*x[i]",        # instructions
  [                       # arguments
    loopy.GlobalArg("x"),
    loopy.GlobalArg("y"),
    loopy.ValueArg("n"),
  ],
)
```

The role of `pyop3` in lowering to such a representation is therefore to determine the correct number and size of the iteration domains, and resolving the complex data access patterns required for multi-dimensional mesh data in order to emit the correct instructions.

Once `pyop3` has generated an appropriate loopy kernel, loopy is called upon to generate code in a low-level device-specific language (e.g. C for CPUs). This code is then compiled by `pyop3` and executed. A high-level overview of the code generation pathway is shown in Figure 6 and an example of the sort of C code that would be emitted is shown in Listing 2.

3.1.2 Data types

Just like PyOP2, `pyop3` has three main data types: `Globals`, values shared across all processors; `Dats`, vectors storing data associated with mesh points; and



(a) A typical PyOP2 `Dat` data layout for a `DataSet` with `dim (3,)`. Note that the nodes are unordered, since we are on an unstructured mesh, but that the DoFs are ordered.

(b) The equivalent `pyop3` data layout that is aware of the topology of the mesh. Note that the number of nodes per entity is not constant - here we indicate that there are 2 nodes per edge but this may differ for cells and vertices.

Figure 7: Comparing the DoF layouts for a vector-valued `Dat` between PyOP2 (left) and `pyop3` (right).

`Mats`, matrices (usually sparse) representing interactions between mesh points.

The distribution of these parallel objects is discussed in Section 3.3.

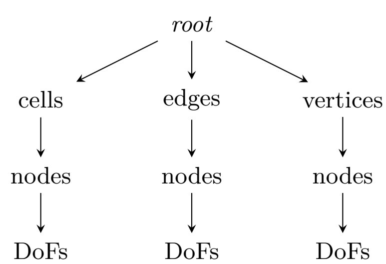
3.2 A new abstraction for mesh data layouts

The additional flexibility `pyop3` has over PyOP2 is thanks to its novel abstraction for describing data layouts.

In PyOP2, data layouts are prescribed by associating data (i.e. DoFs) with *sets*. More precisely, they are described using a `DataSet`, which is formed by associating a `Set` with some *local shape* (a tuple), termed its `dim`. An example of this is shown in Figure 7a. The `Set` here consists of the nodes in the mesh and the `dim` is `(3,)`, thus every node in the mesh stores 3 entries.

The fundamental idea behind `pyop3` is the observation that such a two-layered data layout is insufficient for fully describing the complexities of data living on an unstructured mesh; in particular, *topological information is lost*. Typical `Dats` in PyOP2 use the *nodes* of the mesh to form the underlying `Set`. Recalling from Section 2.2 that the nodes of a mesh are associated with a particular topological entity (i.e. cells, edges, etc), using nodes as a basis for describing a data layout means that information about the topological entities these nodes come from is lost. We claim that this 'premature flattening' of the data layout is disadvantageous, and so `pyop3` aims for a more complete description of it.

In order to do so, `pyop3` replaces the two-layered `DataSet` abstraction with a hierarchical, tree-based data layout. Each layer of the hierarchy (e.g. topological entities or nodes) is represented by a `MultiAxis` object, and to each `MultiAxis` is associated one or more `AxisParts`. `AxisParts` correspond to a particular 'class' of entity stored in the `MultiAxis` and, for instance, allow `pyop3`



(a) An example data layout tree. Each node in the tree, excluding *root*, corresponds to a `pyop3` `AxisPart`.

```
root = (
  MultiAxis()
  .add_part(AxisPart(ncells, id="cells"))
  .add_part(AxisPart(nedges, id="edges"))
  .add_part(AxisPart(nverts, id="verts"))
  .add_subaxis("cells", AxisPart(ncnodes, id="cnodes"))
  .add_subaxis("edges", AxisPart(nenodes, id="enodes"))
  .add_subaxis("verts", AxisPart(nvnodes, id="vnodes"))
  .add_subaxis("cnodes", AxisPart(ncdofs))
  .add_subaxis("enodes", AxisPart(nedofs))
  .add_subaxis("vnodes", AxisPart(nvdofs))
)
```

(b) `pyop3` code for constructing the tree structure shown above. `ncells`, `ncnodes` etc are integers and correspond to the number of entries for a given `AxisPart`.

Figure 8: Example hierarchical data layout for a `Dat` in `pyop3`.

to differentiate between the cells, edges and vertices of the mesh. An example of such a data layout is shown in Figure 7b. Compared with the equivalent PyOP2 data layout in Figure 7a, one can immediately see how topological information is preserved via the addition of an extra “Points” `MultiAxis`. Since the mesh will have been renumbered to improve data locality (Section 2.4.1), entries from different `AxisParts` (cells, edges and vertices) need to be interleaved.

To construct the hierarchy, it is permitted to attach a new `MultiAxis` to each `AxisPart`. The interface for this, and the resulting tree structure, is shown in Figure 8.

The hierarchical data layout system described here is similar to that used in Taichi [12], though Taichi does not support interleaving components of an axis as we do here. It also has no support for distributed memory parallelism (see Section 3.3).

3.2.1 Addressing the inhomogeneous

One major challenge presented by this new abstraction is that axes are no longer homogeneous. Previously, in PyOP2, one could stride over a `Dat` with steps matching the size of its `dim`. This is no longer possible since the strides for, say, the “Points” `MultiAxis` in Figure 7b are not constant because the number of nodes stored can differ between topological entities. Addressing individual topological entities therefore requires careful thought and in `pyop3` is separated into two phases: 1) entries in the data layout are addressed with *typed multi-indices*, and 2) these multi-indices are composed with *layout functions* to determine the correct offset into the data.

To start with the former, a typed multi-index is an object of the form $(t^0_{i_0}, t^1_{i_2}, \dots, t^n_{i_n})$ with t^x indicating the ‘type’ of index i_y. The type is required to select the correct `AxisPart` from a particular `MultiAxis`, and the index indicates which entry of that type is to be selected. To give an example, in order to correctly address the d_0 entry from Figure 7b, the multi-index (e_4, i_0, d_0). Similarly, for the code shown in Listing 1, `mesh.cells.index` is a multi-index selecting all cells in the mesh. `closure(c)` is in fact a *map* and represents a collection of multi-indices.

Provided with a multi-index, we can now use the appropriate layout functions to determine the right memory address (offset) for the data. In `pyop3`, layout functions are simply functions that take one or more indices and return an offset value. For example, in the case of axes with constant strides, a layout function would be some affine function of the form `offset = i*stride + step` (with `i` the input index). If the axis has non-constant strides then the layout function instead needs to be some sort of lookup table (i.e. `offset = offsets[i]`).

Layout functions are associated with individual `AxisParts` and so the process of determining the correct offset for a given multi-index is as follows:

1. Each ‘type’ in the multi-index is used to select a particular `AxisPart` in the hierarchy.

```
do_loop(
  f := mesh.interior_facets.index,
  kernel(
    dat0[closure(support(f))],
    dat1[closure(support(f))]
  )
)
```

Listing 3: Example `pyop3` code for computations over the cells incident on an interior facet.

2. The index part of the multi-index entry is then passed to the layout function of the selected `AxisPart` to compute an offset value.

3. The offsets for each level of the hierarchy are added together, yielding the final memory address.

3.2.2 Maps

As previously mentioned, `closure(c)` from Listing 1 is an example of a *map*, a function from one multi-index to many multi-indices. With a mesh of triangles, `closure(c)` would correspond to a function like:

$$(c_i,) \to \big[(c_i,),\ (e_{j_0},),\ (e_{j_1},),\ (e_{j_2},),\ (v_{j_3},),\ (v_{j_4},),\ (v_{j_5},)\big].$$

In other words, `closure(c)` yields, for every cell, 7 multi-indices pointing to the cell and the 6 incident edges and vertices.

In an identical way to layout functions, maps can be implemented either using affine functions (i.e. `e_0 = c_0*step + start`), or lookup tables (i.e. `e0 = map[c0]`) depending on whether or not there is structure to exploit in the data layout. This enables, for example, the use of structured meshes without needing to incur the data volume/memory bandwidth cost of tabulating a lookup table.

The approach just described enables arbitrary map composition because maps now relate mesh points to mesh points, rather than mesh points to nodes as is done in PyOP2. This is also the approach done by DMPlex.

Could include a commutativity diagram to make this clearer.

An example of this map composition is shown in Listing 3. `pyop3` makes it easy to describe more complex stencils such as for interior facet-type computations where the stencil consists of the closure of the cells incident on a facet, or, in DMPlex terminology, $\mathbb{cl}(\mathbb{supp}(p))$.

Note that thus far we have only discussed maps in the context of mapping between multi-indices containing just a single typed index. It is entirely possible to map between multi-indices with two or more entries, but we require that any 'parent' indices of the map outputs are the same as those for the map input. This is sufficient for unstructured meshes but not for partially-structured meshes (see Section 4.1.3). We similarly require that maps only relate multi-indices of the same size.

These could be demonstrated with an example.

3.2.3 Raggedness

There are occasions where one needs a data structure where the extent of an inner dimension depends on an outer one. This occurs for example in variable layer extruded meshes (Section 4.1) - the extent of the inner dimension (the columns) is dependent upon the mesh point in the base mesh. To get this to work, `pyop3` needs to generate code that resembles:

```
for (int i=0; i<N; ++i)
  for (int j=0; j<nlayers[i]; ++j)
    ...
```

Note how the inner loop extent is dependent upon the outer one via the `nlayers` array.

In `pyop3`, such a 'ragged' data structure can be initialised in the following way:

```
# store nlayers as an integer Dat
nlayers = Dat(MultiAxis(AxisPart(N)), dtype=int)
# nlayers is used as the extent of the 'inner' axis
root = (
  MultiAxis()
  .add_part(AxisPart(N, id="outer"))
  .add_subaxis("outer", AxisPart(nlayers))
)
```

Instead of using a constant integer value to prescribe the extent of the inner `AxisPart`, a `Dat`, `nlayers`, is used instead.

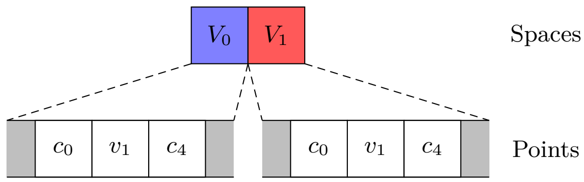
We can also use the same technique to generate code for maps with *variable arity*. An example of this would be for $\mathtt{st}(v)$ with v a vertex since the number of incident edges on a vertex can vary. This is useful for patch-based computations (see Section 4.2).

Should provide more demo code

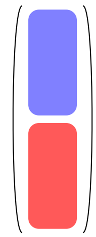
3.2.4 Sparsity

Although not yet implemented, it should be entirely possible to implement sparse data structures by making small extensions to the existing abstraction. If we consider a sparse matrix compressed with compressed-sparse-row (CSR) format, the data layout is described using two arrays: the row and column indices. This is very similar to our existing solution for ragged arrays above except that we assume that the internal dimension is logically dense, and hence do not need to specify column indices.

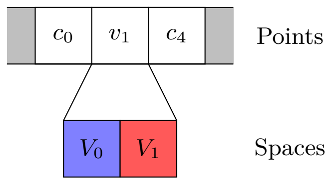
Would be served well by an example



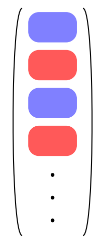
(a) A typical data layout for a 'mixed' system with the spaces V_0 and V_1 forming the 'outer' axis.



(b) The resulting block-structured vector.



(c) A transformed data layout where the "Spaces" and "Points" axes have been swapped.



(d) The resulting interleaved vector.

Figure 9: A possible data layout transformation for a 'mixed' system permitted by `pyop3`. The entries V_0 and V_1 represent the spaces of the mixed system and the "Points" axis is representative of the mesh. Note that additional subaxes for, for example, nodes and DoFs would be permitted.

3.2.5 Orienting degrees-of-freedom

Having a hierarchical, 'mesh-aware' data layout makes it straightforward to correctly handle orientations when packing stencils. In `pyop3` we can augment maps to return orientation information along with each multi-index. Using the triangles from Figure 5 as an example, the edges of the 'canonical' triangles would have orientations $(0, 0, 0)$, whilst the edges of the triangles with one flipped edge would have orientations $(0, 0, -1)$.

To handle the DoF transformations required for correctly orienting things, `pyop3` can apply a different transformation on each `AxisPart`. For instance if the nodes are out of order then `pyop3` can apply a permutation to the entries of the `AxisPart` (though in practice this would likely be done when preparing the maps), and if the DoFs require a transformation then this can be applied to the corresponding `AxisPart`.

A demo would help

3.2.6 Data layout transformations

With this decomposition of data layouts in a flexible, declarative hierarchy, it is now relatively easy to make *data layout transformations* to improve data locality (Section 2.4.1). Some of the possible optimisations include:

Swapping axes pyop3 allows the user to swap a pair of `MultiAxes` such that the 'inner' axis becomes the 'outer' and vice versa. This can improve data locality if different entries from the outer axis are accessed in the same iteration.

An example of this is shown in Figure 9. Typically a 'mixed' system like this - here formed of V_0 and V_1 - stores data in a blocked format (Figures 9a and 9b). This means that the DoFs corresponding to the same mesh point for V_0 and V_1 are very far apart in memory. If they are both used in the local computation then this constitutes poor data locality.

To rectify the situation, we can swap the axes such that the mixed components are stored per mesh point, adjacent in memory. This is shown in Figures 9c and 9d.

Reordering data within axes Once the axes have been ordered in the most advantageous way, we can now begin to rearrange the entries in a `MultiAxis` to maximise locality. In the context of meshes, these reorderings could correspond to, for example, an RCM renumbering of the mesh entities or a different ordering of the elements up the columns of an extruded mesh. As well as to improve data locality, there are circumstances where the user may wish to prescribe a custom data layout specific to their problem. For example, certain fieldsplit preconditioners may require a subset of DoFs to work, and performance could be improved by storing this subset as a contiguous block of memory.

To implement this reordering, pyop3 simply requires that a different layout function be associated with the respective `AxisParts`. Layout functions map entities in an axis to their offsets in memory and so using a different one gives the user total freedom over where data is stored. This approach is roughly equivalent to permuting a `PetscSection` in PETSc, though in our case we permit layout functions to be affine functions as well as tabulations.

3.2.7 Other locality optimisations

pyop3's abstraction also enables code transformations such as vectorisation and time-tiling (Section 2.4.1). Such optimisations are not data layout transformations, but transformations of the *iteration set*. To enact such transformations, the iteration over the set needs to be broken into a nest of loops. This would look something like:

```c
for (int i=0; i<NOUTER; ++i) {
  for (int j=0; j<NINNER; ++j) {
    int k = i*NINNER + j;
    ...
  }
}
```

In the case of vectorisation, `NINNER` would correspond to the length of the vector lanes of the CPU. If time-tiling, the size of tiles should be chosen such that data required for the fused kernels remain in cache between their invocations.

3.2.8 Enabling new research

In addition to the benefits espoused above, `pyop3`'s data layout abstraction should enable a number of new mathematical methods that would be very hard to impossible to implement in PyOP2. These include:

- **p-adaptivity** In order to reduce the errors in a simulation, one may vary the polynomial degree of particular cells in a process known as p-adaptivity. This introduces difficulties for stencil codes because: a) multiple local kernels are needed, one for each polynomial degree; and b) there are 'hanging' nodes at the boundaries between cells of differing degrees.

  Provide example

  Problem (a) is trivial to resolve in `pyop3`. Rather than having a `MultiAxis` that is composed only of cells, edges and vertices (each a distinct `AxisPart`), additional `AxisParts` can be added such that mesh points of different degree are associated with a unique `AxisPart`. This would mean that points would no longer be addressed with multi-indices containing typed indices of the form c_i, e_i and v_i, but instead be addressed with c_i^d, e_i^d and v_i^d, where d corresponds to the polynomial degree of the entity.

  Problem (b) is more challenging to solve and requires the addition of *constraints* to the abstraction[1]. Adding constraints would also enable *adaptive mesh refinement*. Support for constraints has already been implemented for DMPlex [14].

- **Mixed meshes** A mixed mesh is a mesh containing multiple different types of cell (e.g. triangles and squares). Iterating over such a mesh poses the same fundamental problem as p-adaptivity: different local kernels are required depending on the cell type. Since `pyop3` is 'mesh-aware' and can reason about the different classes of mesh points, this problem becomes trivial to solve, though care is still needed when handling orientations.

- **Particle-in-cell methods** Particle-in-cell methods are a type of numerical method where the cells of a mesh are associated with a number of, possibly advecting, particles. Since the number of particles differs between cells, a variable arity map is required to address them. This is addressed in Section 3.2.3.

3.3 Parallel design

At present, `pyop3` implements an identical approach to distributed computing as PyOP2: `Globals`, `Dats` and `Mats` are distributed between processes using only MPI parallelism. Hybrid parallel models with shared memory (e.g. OpenMP) are not supported.

To begin with, distribution of `Globals` is trivial. They represent globally consistent values and so consensus between processes is reached via all-to-all broadcasts.

[1] This is the approach used by libCEED: `https://libceed.org/en/latest/libCEEDapi/#finite-element-operator-decomposition`.

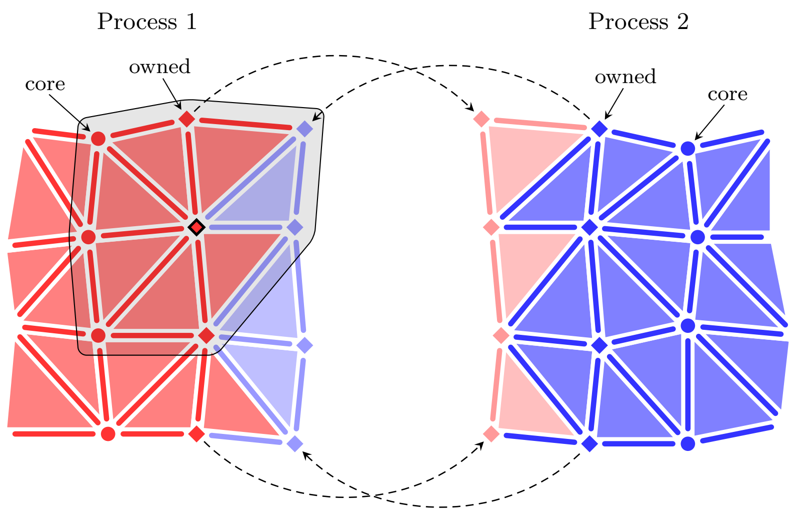


Figure 10: An example of a mesh distributed between two processes. Points ‘belonging’ to processes 1 and 2 are shown in red and blue respectively. The mesh is needed for vertex patches (shaded) and so the size of the overlap is chosen such that all required DoFs are stored locally. ‘Core’ vertices are stored as circles and ‘owned’ as diamonds. The direction of halo exchanges is indicated by the arrows.

3.3.1 Dats

`Dats`, by comparison, are distributed in a much more complicated way. Every DoF stored by a `Dat` 'belongs' to a single process but it may be needed for a stencil on a process storing adjacent points. Each process therefore stores a set of *halo* DoFs that belong to other processes. To ensure that computations are always done with up-to-date values, *halo exchanges* are triggered whenever one wishes to read halo data (writing to halos is fine).

Since halo values only exist on the outside of the mesh volume allocated to a particular process, points in the iteration set are classified as either *core* and *owned*. Core points are those where a computation can be executed without requiring an up-to-date set of halo entries. Conversely, owned points are those where an up-to-date halo is required. Such points exist only near the boundaries of the distributed mesh. Classifying the points into these categories lets `pyop3` interleave computation and communication: core points are be computed over while the halo exchanges required for owned points are in-flight.

An example distributed mesh is shown in Figure 10. Points 'belonging' to a process are shown in red and blue respectively and the halo values are shown in the appropriate colour for each.

In this example, the stencil getting used in the iteration is the closure of the patch of cells around a vertex (i.e. $\mathbb{cl}(\mathbb{st}(v))$). Since this constitutes a fairly large stencil, the halos have to be correspondingly larger to contain all of the values required by the vertices belonging to the process.

In the figure, core vertices are shown as circles and owned as diamonds. One can see that stencils involving the core vertices can be computed without the need for a halo exchange because all points in the stencil are the same colour (i.e. on the same process) as the vertex itself. Similarly, the patches around owned vertices contain at least one mesh point belonging to the other process and hence a halo exchange is required.

In this example we have chosen to demonstrate mesh partioning using halos of the smallest possible size. This is frequently desirable - smaller halos means a reduced volume data is transferred - but not universally so. If the cost of computing the stencils is smaller than the cost of the halo exchanges one may want a larger halo to reduce the number of messages sent between processes at the cost of performing some redundant computations inside the wider halo region. This gains in terms of reduced data movement, but leads to each process Such a strategy is not currently implemented in PyOP2 or `pyop3`, but it has been pursued in the past [19]. DMPlex has support for specifying meshes with arbitrary overlap (i.e. halo depth).

3.3.2 Mats

`pyop3` currently relies on PETSc to provide routines for matrix insertion[2]. In

[2]We would like to have a more unified abstraction for `Globals`, `Dats` and `Mats` such that this reliance on PETSc goes away, but this is very preliminary work and not discussed here.

parallel, PETSc distributes the data by partitioning the rows between processes[3].

3.3.3 Weak-scaling

Weak-scaling (Section 2.3.1) is an appropriate metric to evaluate the parallel communication design patterns described above. Firedrake, and therefore PyOP2 and PETSc by extension, has been demonstrated to have good weak-scaling performance [23]. Since `pyop3` reuses concepts from PyOP2, we expect similar behaviour. Strong-scaling performance is addressed in Section 3.4.

3.4 Avoiding Python overhead

This could probably be turned into an entire chapter where I also discuss my contributions to speed up Firedrake (e.g. assemble, assign, freezing halos).

Python is the language of choice for `pyop3` for a number of compelling reasons:

- It is a mature language with a wealth of pre-existing libraries so interoperability with other applications is streamlined.
- Its dynamically typed and interpreted nature allows users to prototype code very quickly.
- The syntax is very flexible, making it easy to develop embedded domain-specific languages that are concise and readable.

The main argument levelled against using Python for scientific computing applications is that it is much slower than a compiled language like C or Fortran. However, this issue is generally not important to code generation frameworks like `pyop3` and Firedrake. The performance critical parts of the code - the 'hot loops' - are just-in-time compiled to machine code and so performance is roughly the same as hand-written code. The fact that the rest of the library is written in Python does not matter as only a tiny fraction of the programs runtime is spent executing it. The performance hotspots should all be compiled code.

Unfortunately, there is one significant occasion where this claim falls down, and our choice of Python causes trouble: in the strong-scaling limit (Section 2.3.1). In this limit the problem occupying the 'hot loops' is not sufficiently large to dominate the program runtime and so a larger fraction of time is spent executing Python code. Compared with codes written in compiled languages, the problem size at which performance begins to degrade is larger, resulting in worse strong-scaling efficiency. This problem has been explicitly observed for Firedrake [6][4].

The solution to the issue is simple: `pyop3` needs to spend as little time as possible executing Python code. This can realistically be accomplished in two ways. Firstly, one could accelerate the new hot loops by rewriting them in a

[3] `https://petsc.org/release/docs/manual/mat/#parallel-aij-sparse-matrices`

[4] The results shown in this paper are exaggerated. We found that it was possible to substantially improve strong-scaling performance with a few minor code modifications.

compiled language. This could be done manually using a compiled language bound to Python (e.g. Cython[5]), or possibly with code generation. Alternatively, the library could apply judicious caching in order to minimise the number of times a function is called. Of these the former is somewhat trivial and will not be discussed here. We will instead focus on achieving performant caching solutions in `pyop3`.

3.4.1 Caching loop expressions

As mentioned in Section 3.1, one can use either the function `do_loop(...)` to instantiate and then immediately execute a loop expression, or the function `loop(...)` to create a persistent expression that can be called multiple times. While more concise, the former is not suitable for executing identical loop expressions repeatedly due to extra Python overhead involved with instantiating the expression. The latter naturally avoids this overhead by only instantiating the expression once.

Having persistent loop expressions introduces a couple of new complications. In particular, loop expressions are instantiated with reference to specific data structures (e.g. `mesh`, `dat0` and `dat1` in Listing 2). This causes trouble in two ways:

1. *Executing identical loop expressions with different data structures requires separate instantiations*

   This is an inconvenient expense for Firedrake's `assemble` function for example, where one can assemble the same input multiple times but write to a different output.

2. *Caching loop expressions causes memory leaks*

   One may wish to cache loop expressions to facilitate reuse. However, the data structures referenced by the expression have a large memory footprint and if the loop expressions remain in the cache then they will never be removed by the Python garbage collector, resulting in leaking significant amounts memory.

To resolve issue 1, users are able to pass keyword arguments when calling `pyop3` loop expressions to swap out data structures on the fly (e.g. `expr(output=mynewoutput)`). Since this approach requires references to the other data structures to persist, issue 2 is then resolved by using *weak references* to the data structures. Weak references in Python are references that do not increase the reference count of the object. This means that memory leaks cannot occur even when caching expressions since the data structures will still be cleaned up when they go out of scope.

[5] `https://cython.org/`

4 Future work

In this section we present a number of possible extensions to `pyop3`.

4.1 Direct addressing for partially-structured meshes

Depending upon the application, certain simulations use meshes that possess 'partial structure'. That is, meshes that possess both unstructured components and structured components. In general, the structure found in these meshes can be classified as either *refinement* or *extrusion*.

A refined mesh is a mesh where some unstructured 'coarse' mesh is refined by replacing cells of the mesh with multiple, smaller cells. Edges and vertices are also inserted to keep appropriate connectivity, though *hanging nodes* may occur if the refinement is non-conforming (see Section **??**). An example refined mesh is shown in Figure 13a.

By contrast, an extruded mesh is created by taking an unstructured 'base' mesh and extruding it into some number of layers. This results in a mesh composed of columns (e.g. Figure 4).

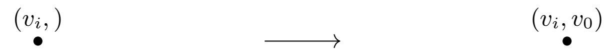
For refined meshes the partial structure comes from having a finite number of possible refinement patterns. Given a point in the refined mesh and a refinement pattern, it should be possible to address stencils without needing a lookup table for every single point as one can reason about the connectivity. For extruded meshes the partial structure exists within the columns - each layer can be addressed directly using offsets given a starting point at the bottom cell.

The benefit to using partially structured meshes is that the memory volume of the simulation can be reduced which, in a memory-bound computation, will directly lead to speedup. This is discussed in detail in Section 2.4.2 where we observed that the savings in memory volume are actually not that great, with a best case of 25%. Since the potential benefits are limited, we have not implemented meshes with partial structure in `pyop3` yet. This section exists to demonstrate that our implementation does not prohibit making such optimisations in the future, and that in fact they would be relatively simple to implement as a consequence of our mesh-aware data layout.

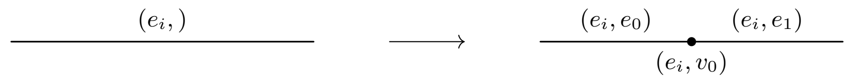
As an aside, it is important to note that, in order to be able to extract any memory savings from this approach, both the stencils (a.k.a. maps) *and* the layout functions must be expressible without the need for lookup tables.

4.1.1 A unifying abstraction: mesh transformations

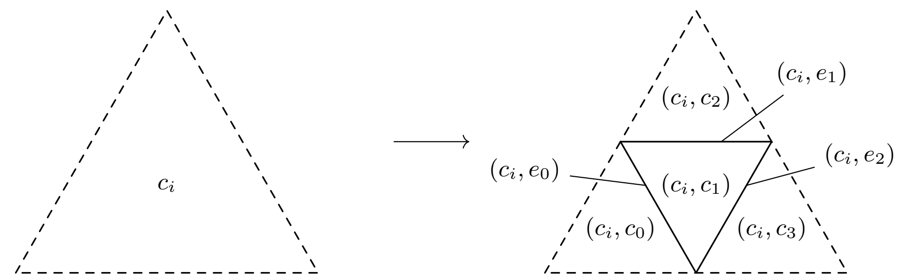
To handle refinement and extrusion, DMPlex has a convenient way of unifying the two. Termed *mesh transformations*, the points in the input mesh are modified via some *production rule*, resulting in a new, transformed, mesh. Some example production rules are shown in Figures 11 (refinement) and 12 (extrusion). It is important to note that the production rule does not produce a 'complete' cell - frequently only cell interiors without edges or edges without



(a) Vertex refinement - a no-op.



(b) Edge refinement.



(c) A possible refinement of a triangle. Note that no vertices are produced by the transformation and that the dashed lines indicate the production of a cell interior but not all its edges.

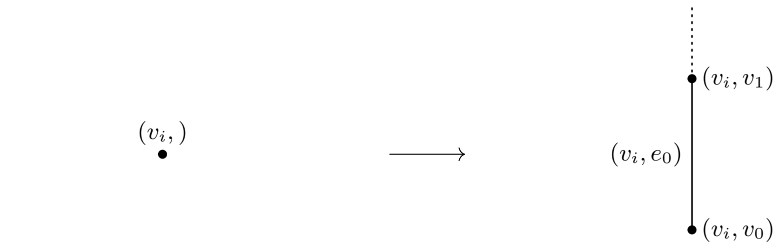
Figure 11: Example refinement transformations.

vertices are produced in the transformation. This is necessary because it constrains each point in the transformed mesh to only have a single parent, making reasoning about structure much easier.

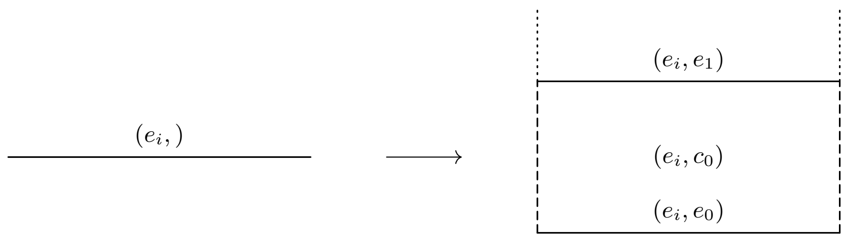
Note that there are a great many more production rules that are not shown here. We have not included refinement rules for 3D polytopes (e.g. tetrahedra) and quadrilaterals are skipped. Also, in some cases there are multiple ways to refine a point, for example a triangle can be 'green' refined by connecting one vertex with the midpoint of the opposite edge [4]. Lastly, it should also be remarked that some transformations are naturally parametric. For example extruding a mesh requires the number of layers to insert. Likewise we could consider refining an edge, say, into 3 or more segments rather than just 2 (Figure 11b).

4.1.2 Implementation: Overview

To implement mesh transformations in a structure preserving way, we simply require that the mesh points produced from the transformation produce a new subaxis in the data layout. This is most simply demonstrated for extruded meshes. If we consider the extruded mesh and data layout shown in Figure 14, the 'base' mesh is formed of 2 edges (e_0 and e_1) and 3 vertices (v_0, v_1 and v_2). From Figure 12 we see that, under extrusion, vertices produce points like $(v_0, e_0, v_1, \dots)$ and that edges produce points like $(e_0, c_0, e_1, \dots)$. These production rules exactly match the subaxes shown in Figure 14b.



(a) Vertex extrusion.



(b) Edge extrusion. Note that no vertices are produced in the transformation as they would be produced by the vertices incident on the initial edge.

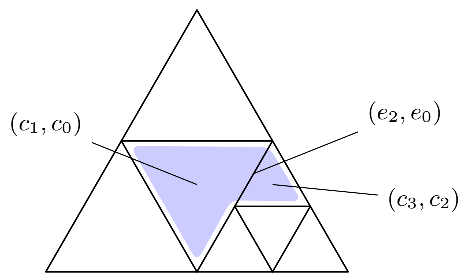
Figure 12: Example extrusion transformations. The dotted lines indicate that the transformation may produce more than a single layer.

The principle benefit of codifying the distinction between the base points and those up each of the columns in separate axes is that we can now use separate layout functions (see Section 3.2) to handle the addressing for each. The base mesh is unstructured - and so an indirection map is required to address its axis - but the points up each of the columns are structured and can be addressed using some affine indexing function (i.e. `offset = start + i*step`).

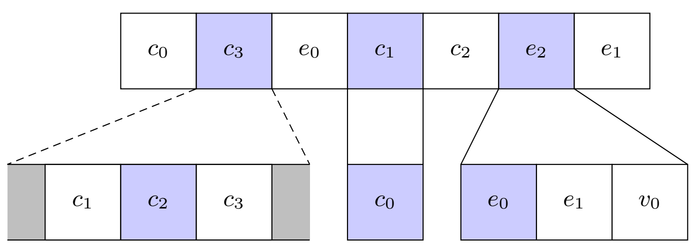
4.1.3 Implementation: Rethinking maps

As described in Section 3.2.2, a map is a function that accepts a multi-index and returns a collection of multi-indices. For meshes without partial structure it is sufficient to limit the length of these multi-indices to 1 - any 'parent' point, often non-existing, will always be the same for both the input point and all of the outputs. Inconveniently, for partially structured meshes, this assumption no longer holds and parent points may differ. This is illustrated in Figure 14: $\mathtt{st}((v_1, e_0))$ is $\left[(v_1, e_0), (e_0, c_0), (e_1, c_0)\right]$ - the parent points, here corresponding to the 'base' mesh points, are different.

As discussed above, in order to achieve any performance gains via memory volume reduction both the maps and the layout functions must be expressible without resorting to a global tabulation. In the extruded case just described, this really means that one cannot store the full multi-indices in a lookup table. Instead, the information available to `pyop3` is as follows: 1) the $\mathtt{st}$ of a vertical

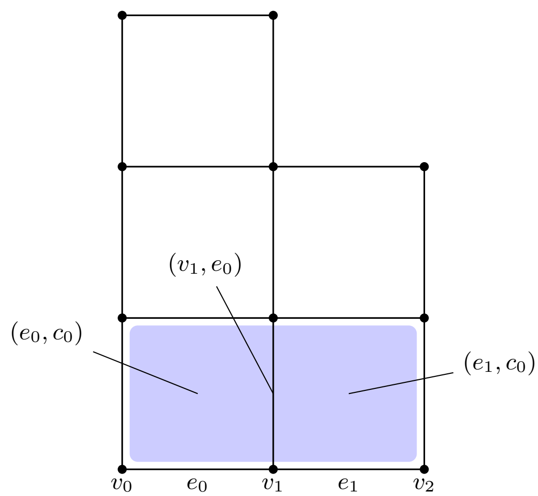


(a) An example of a stencil - $\mathtt{st}((e_2, e_0))$ - over a refined mesh. Note that the unrefined cell (c_1, c_0) is still indexed with two indices. We say that it has been refined using the identity transformation.

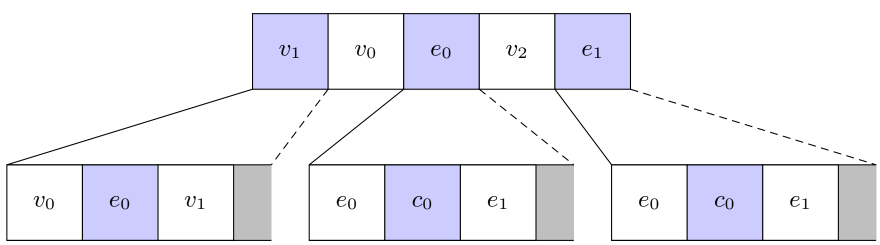


(b) Example data layout for the refined mesh shown above. Note that the base mesh in unstructured which is why the top axis is unordered.

Figure 13: Example data layout and stencil for a refined mesh.



(a) An example of a stencil - $\mathfrak{st}((v_1, e_0))$ - applied to an extruded mesh formed by extruding a 'base' mesh consisting of 2 edges (e_0 and e_1) and 3 vertices (v_0, v_1 and v_2). Note that the mesh shown here has 'variable layers' to emphasise that such a mesh would be supported by our abstraction.



(b) Example data layout for the extruded mesh shown above. Points in the stencil are highlighted in blue. Note that the points in the 'base' mesh are not ordered since it represents an unstructured mesh.

Figure 14: Example data layout and stencil for an extruded mesh.

edge contains cells 'belonging' to adjacent base edges, and 2) the adjacency relationships between base entities (i.e. we know that v_1 is incident on e_0 and e_1). Using these pieces of information it is possible to reconstruct the full set of complete multi-indices required to address the data correctly.

4.1.4 Implementation: Interfacing with DMPlex

One major shortcoming of the existing extruded mesh implementation in PyOP2 is that an extruded mesh is not in fact a DMPlex instance. Instead it uses a DMPlex to represent the unstructured base mesh and then uses custom code to handle the extrusion up the columns. With such an implementation, as far as DMPlex is concerned, an extruded mesh is merely a mesh with 'very big' cells (i.e. storing all the DoFs for each column).

This approach means that extruded meshes are special-cased throughout PyOP2 and Firedrake, requiring specialised implementations for all manner of operations including: preconditioner application, multigrid and I/O. Indeed there are places in Firedrake where, due to the extra burden of developing a custom implementation, there is no support for extruded meshes at all. One example of such a case is the fact that one cannot have an extruded `VertexOnlyMesh`.

`pyop3` takes a different approach to describing partial structure in meshes that we believe should avoid the need for a proliferation of custom implementations. Our approach revolves around the idea that a mesh should always be represented by a single DMPlex instance, from which any structure can be inferred. By choosing to sit directly on top of DMPlex, all code for unstructured meshes should now work for both cases.

To obtain a hierarchical data layout similar to that shown in Figure 14b, `pyop3` would inspect the *labels* of the DMPlex. These labels are integers associated with each mesh point.

The critical point here is that the *labels of the input mesh points are automatically passed to the transformed points*. This means that one can, for example, uniquely label each point in the input mesh and then extrude it and this will result in a mesh where all of the transformed points up the column 'know' the base point to which it belongs. The same approach naturally works for refined meshes, uniquely labelling the coarse points prior to refinement allows `pyop3` to reconstruct the right data layout by inspecting the labels.

Since labels persist when writing to disk, we believe that, with minimal code, the hierarchical data structures could be reconstructed via analysis of these labels.

4.2 Patch-based multigrid smoothers

It has been demonstrated that geometric multigrid with a smoother stage involving the direct solution of a 'local' finite element problem is effective for many problem [28, 3, 9]. These 'local' problems, called *patches*, are in fact subdomains of the entire mesh taken via some composition of DMPlex restrictions. Examples include *vertex-star* patches, the DoFs defined on a vertex and entities in its st, and *Vanka* patches, the same but taking the cl of the vertex-star to capture a larger patch. The idea behind these patches is that a local finite element problem is solved using them and this contributes an update, via either the additive or multiplicative Schwartz methods, to the current guess.

This abstraction has been implemented via contributions to Firedrake, PETSc and PyOP2 and is called PCPATCH [10]. To run, the 'outer loop' over patches and the updates (either additive or multiplicative) are performed by PETSc. Callbacks registered in Firedrake are used to construct the local problem. Since the problem is defined entirely using PETSc types, one can utilise any of the possible solver strategies provided by it. In particular, matrix-free solver implementations are natively supported.

Firedrake also supports an alternative backend for applying patch preconditioners called *TinyASM* [29]. TinyASM, at setup time, precomputes the matrix inverses for each patch so the local solve can be done very efficiently without needing to use PETSc objects, which are specialised towards much larger linear systems.

Both of these existing approaches have a number of drawbacks. As just mentioned, solving linear systems in PETSc can be inefficient for patches as one needs to solve lots of small problems, rather than a single large one. This is solved by TinyASM, but their approach is unsuitable for high order methods because it requires computing a large number of dense inverses which can cause a machine to run out of memory. Also, both systems require a significant amount of hand-coding for specific patches and reasoning about numberings etc.

We also run into problems when dealing with sparsity-preserving discretisations at high-order. Matrix-explicit implementations are unsuitable because the per-patch matrices, though sparse, are very large and can fill up a machine's memory. Also, matrix-free implementations won't work because each cell returns a dense block and the sparsity is lost. To resolve, we would like to be able to construct sparse matrices 'on-the-fly' for each patch. To make this efficient we would need to memoize the different potential sparsity patterns - you get different patterns depending on the number of incident edges on a vertex for example.

In `pyop3`, we would like to simplify these implementation considerations by raising the level of abstraction. The `pyop3` interface (Section 3) is already flexible enough to permit the sorts of loops that patch smoothing requires. For example, a Vanka patch (closure of a vertex-star) could be expressed as follows:

```
loop(v := mesh.vertices.index, [
  loop(p := star(v).index, [
    assemble_jacobian(dat1[closure(p)], dat2[closure(p)], "mat"),
    assemble_residual(dat3[closure(p)], "vec"),
  ]),
  solve_and_update("mat", "vec", dat4[v]),
])
```

Note that here we use the strings `"mat"` and `"vec"` to identify the loop temporaries. This is syntactic sugar and if we were to want to specify nondefault behaviour for these objects, for instance memoizing the sparsity patterns or using a pre-computed inverse, then we could instead instantiate specific `LoopTemporary` objects.

5 Conclusions

References

[1] Rainer Agelek et al. "On Orienting Edges of Unstructured Two- and Three-Dimensional Meshes". In: *ACM Transactions on Mathematical Software* 44.1 (July 24, 2017), pp. 1–22. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3061708. URL: https://dl.acm.org/doi/10.1145/3061708 (visited on 07/01/2021).

[2] Martin S. Alnæs et al. "The FEniCS Project Version 1.5". In: *Archive of Numerical Software* 3.100 (2015). DOI: 10.11588/ans.2015.100.20553.

[3] Douglas N Arnold, Richard S Falk, and R Winther. "PRECONDITIONING IN H (Div) AND APPLICATIONS". In: (1997), p. 28.

[4] Randolph E. Bank and Andrew H. Sherman. *A Refinement Algorithm and Dynamic Data Structure for Finite Element Meshes.*

[5] Gheorghe-Teodor Bercea et al. "A Structure-Exploiting Numbering Algorithm for Finite Elements Onextruded Meshes, and Its Performance Evaluation in Firedrake". In: *Geoscientific Model Development* 9.10 (Oct. 27, 2016), pp. 3803–3815. ISSN: 1991-9603. DOI: 10.5194/gmd-9-3803-2016. URL: https://gmd.copernicus.org/articles/9/3803/2016/ (visited on 10/12/2021).

[6] Justin Chang et al. "Comparative Study of Finite Element Methods Using the Time-Accuracy-Size(TAS) Spectrum Analysis". In: *SIAM Journal on Scientific Computing* 40.6 (Jan. 2018), pp. C779–C802. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/18M1172260. URL: https://epubs.siam.org/doi/10.1137/18M1172260 (visited on 11/26/2020).

[7] E. Cuthill and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices". In: *Proceedings of the 1969 24th National Conference On -*. The 1969 24th National Conference. Not Known: ACM Press, 1969, pp. 157–172. DOI: 10.1145/800195.805928. URL: http://portal.acm.org/citation.cfm?doid=800195.805928 (visited on 10/24/2022).

[8] Alain Darte. "On the Complexity of Loop Fusion". In: *Parallel Computing* (2000), p. 19.

[9] Patrick E. Farrell, Lawrence Mitchell, and Florian Wechsung. "An Augmented Lagrangian Preconditioner for the 3D Stationary Incompressible Navier–Stokes Equations at High Reynolds Number". In: *SIAM Journal on Scientific Computing* 41.5 (Jan. 2019), A3073–A3096. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/18M1219370. URL: https://epubs.siam.org/doi/10.1137/18M1219370 (visited on 10/11/2022).

[10] Patrick E. Farrell et al. “PCPATCH: Software for the Topological Construction of Multigrid Relaxation Methods”. In: *ACM Transactions on Mathematical Software* 47.3 (June 25, 2021), pp. 1–22. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3445791. URL: https://dl.acm.org/doi/10.1145/3445791 (visited on 10/28/2021).

[11] M. Homolya and D. A. Ham. “A Parallel Edge Orientation Algorithm for Quadrilateral Meshes”. In: *SIAM Journal on Scientific Computing* 38.5 (Jan. 2016), S48–S61. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/15M1021325. URL: http://epubs.siam.org/doi/10.1137/15M1021325 (visited on 06/29/2021).

[12] Yuanming Hu et al. “Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures”. In: *ACM Transactions on Graphics* 38.6 (Dec. 31, 2019), pp. 1–16. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3355089.3356506. URL: https://dl.acm.org/doi/10.1145/3355089.3356506 (visited on 09/15/2022).

[13] François Irigoin and Rémi Triolet. “Supernode Partitioning”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1988, pp. 319–329.

[14] Tobin Isaac and Matthew G. Knepley. “Support for Non-conformal Meshes in PETSc’s DMPlex Interface”. Aug. 10, 2015. arXiv: 1508.02470 [cs]. URL: http://arxiv.org/abs/1508.02470 (visited on 06/04/2021).

[15] Andreas Klöckner. “Loo.Py: Transformation-Based Code Generation for GPUs and CPUs”. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY’14* (2014), pp. 82–87. DOI: 10.1145/2627373.2627387. arXiv: 1405.7470. URL: http://arxiv.org/abs/1405.7470 (visited on 10/14/2020).

[16] Kaushik Kulkarni and Andreas Kloeckner. “UFL to GPU: Generating near Roofline Actions Kernels”. 2021. DOI: 10.6084/m9.figshare.14495301. URL: http://mscroggs.github.io/fenics2021/talks/kulkarni.html.

[17] Michael Lange et al. “Efficient Mesh Management in Firedrake Using PETSc DMPlex”. In: *SIAM Journal on Scientific Computing* 38.5 (Jan. 2016), S143–S155. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/15M1026092. URL: http://epubs.siam.org/doi/10.1137/15M1026092 (visited on 12/08/2020).

[18] Anders Logg, Kent-Andre Mardal, and Garth Wells, eds. *Automated Solution of Differential Equations by the Finite Element Method*. Vol. 84. Lecture Notes in Computational Science and Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-23098-1 978-3-642-23099-8. DOI: 10.1007/978-3-642-23099-8. URL: http://link.springer.com/10.1007/978-3-642-23099-8 (visited on 03/09/2020).

[19] Fabio Luporini et al. *Automated Tiling of Unstructured Mesh Computations with Application to Seismological Modelling*. June 19, 2019. arXiv: 1708.03183 [physics]. URL: http://arxiv.org/abs/1708.03183 (visited on 09/06/2022).

[20] G.R. Mudalige et al. "Design and Initial Performance of a High-Level Unstructured Mesh Framework on Heterogeneous Parallel Systems". In: *Parallel Computing* 39.11 (Nov. 2013), pp. 669–692. ISSN: 01678191. DOI: 10.1016/j.parco.2013.09.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167819113001166 (visited on 10/12/2020).

[21] G.R. Mudalige et al. "OP2: An Active Library Framework for Solving Unstructured Mesh-Based Applications on Multi-Core and Many-Core Architectures". In: *2012 Innovative Parallel Computing (InPar)*. 2012 Innovative Parallel Computing (InPar). San Jose, CA, USA: IEEE, May 2012, pp. 1–12. ISBN: 978-1-4673-2633-9 978-1-4673-2632-2 978-1-4673-2631-5. DOI: 10.1109/InPar.2012.6339594. URL: http://ieeexplore.ieee.org/document/6339594/ (visited on 01/28/2021).

[22] J. Ramanujam and P. Sadayappan. "Tiling Multidimensional Iteration Spaces for Multicomputers". In: *Journal of Parallel and Distributed Computing* 16.2 (Oct. 1992), pp. 108–120. ISSN: 07437315. DOI: 10.1016/0743-7315(92)90027-K. URL: https://linkinghub.elsevier.com/retrieve/pii/074373159290027K (visited on 11/24/2022).

[23] Florian Rathgeber et al. "Firedrake: Automating the Finite Element Method by Composing Abstractions". In: *ACM Transactions on Mathematical Software* 43.3 (Dec. 2016), pp. 1–27. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/2998441. URL: https://dl.acm.org/doi/10.1145/2998441 (visited on 10/05/2020).

[24] Marie E. Rognes, Robert C. Kirby, and Anders Logg. "Efficient Assembly of H(Div) and H(Curl) Conforming Finite Elements". In: *SIAM Journal on Scientific Computing* 31.6 (Jan. 2010), pp. 4130–4151. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/08073901X. URL: http://epubs.siam.org/doi/10.1137/08073901X (visited on 06/29/2021).

[25] Matthew W. Scroggs et al. "Basix: A Runtime Finite Element Basis Evaluation Library". In: *Journal of Open Source Software* 7.73 (May 25, 2022), p. 3982. ISSN: 2475-9066. DOI: 10.21105/joss.03982. URL: https://joss.theoj.org/papers/10.21105/joss.03982 (visited on 10/17/2022).

[26] Matthew W. Scroggs et al. "Construction of Arbitrary Order Finite Element Degree-of-Freedom Maps on Polygonal and Polyhedral Cell Meshes". Mar. 23, 2021. arXiv: 2102.11901 [cs, math]. URL: http://arxiv.org/abs/2102.11901 (visited on 06/29/2021).

[27] Tianjiao Sun et al. "A Study of Vectorization for Matrix-Free Finite Element Methods". In: *The International Journal of High Performance Computing Applications* 34.6 (Nov. 2020), pp. 629–644. ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342020945005. URL: http://journals.sagepub.com/doi/10.1177/1094342020945005 (visited on 10/13/2020).

[28] S.P Vanka. "Block-Implicit Multigrid Solution of Navier-Stokes Equations in Primitive Variables". In: *Journal of Computational Physics* 65.1 (July 1986), pp. 138–158. ISSN: 00219991. DOI: 10.1016/0021-9991(86)90008-2. URL: https://linkinghub.elsevier.com/retrieve/pii/0021999186900082 (visited on 11/09/2022).

[29] Florian Wechsung. *TinyASM: A Block-Jacobi Implementation for PETSc and Firedrake Focussed on Efficiently Inverting and Then Applying Small Dense Matrices.* URL: https://github.com/florianwechsung/TinyASM.

[30] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1498765.1498785. URL: https://dl.acm.org/doi/10.1145/1498765.1498785 (visited on 09/20/2022).