High performance mesh abstractions

Connor Ward

November 4, 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 low-level performance. Each discipline is presented with a particular interface from which the problems of interest can be expressed in the clearest possible way. This makes the application of particular optimisations straightforward as problem-specific information can be explicitly enumerated, rather than requiring inference to determine.

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? 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. 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 equation (PDE)s 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.

Having motivated the need for a mesh traversal abstraction, we now move on to motivating the need for a *high-performance* mesh traversal abstraction. In other words, are there reasonable use-cases where this traversal constitutes a substantial fraction of the overall program runtime?

To begin, the wall-clock time of a program, assuming good algorithmic/parallel design, is usually limited by some combination of the maximum throughput of the processor and the cost of moving data to and from said processor. Given that the principle role for `pyop3`is to marshall data for the kernel we focus on codes where the cost of moving data is the bottleneck. Any effort on minimising the number of floating-point operations (FLOPs) would be wasted as such optimisations would only be useful inside the 'hot' loops of the kernel. Also, hardware developments have seen a general trend where computing power is increasing at a more rapid rate than memory access speed. As such, the bottleneck in an increasing number of codes is going to be the memory accesses.

In this work, we present `pyop3`...

The rest of this paper is laid out as follows: ...

2 Background

In this chapter we review: existing software abstractions for mesh computations, common strategies for optimising performance, and mesh-specific optimisations.

2.1 Existing software abstractions

2.1.1 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 [12, 11]. 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.

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.

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 [7]. This intermediate representation allows for interplay between the local kernels enabling optimisations such as inter-element vectorisation [15].

At present, PyOP2 only works on distributed memory, CPU-only systems (although some work has been done to permit execution on GPUs [8]). 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.

2.1.2 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.

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 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

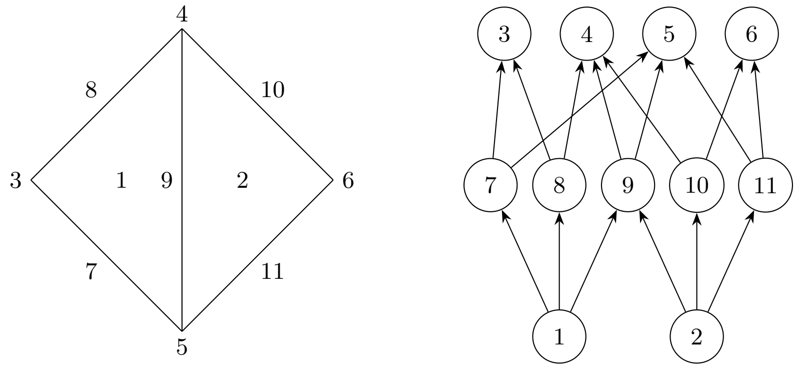
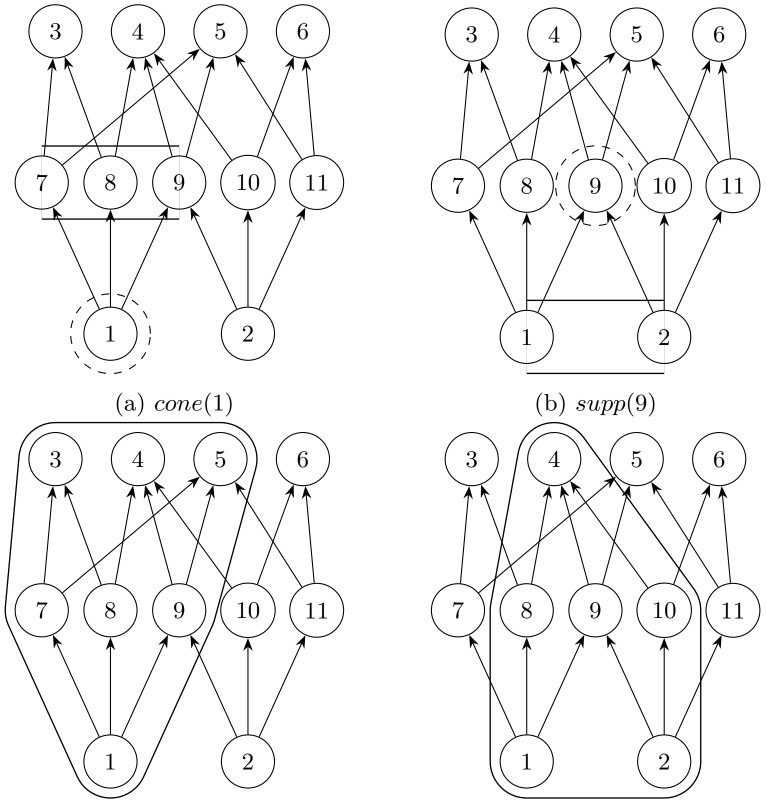


Figure 1: ...



(a) *cone*(1)

(b) *supp*(9)

(c) *cl*(1)

(d) *star*(4)

Figure 2: ...

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.

2.2 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. The aim is to motivate design choices made in `pyop3`.

2.2.1 Optimisation methodology

Premature optimisation is a classic trap software developers fall into when they attempt to accelerate their codes. It is surprisingly easy to find oneself making assumptions about the locations and causes of bottlenecks, and then wasting a considerable amount of time on optimising code that only constitutes a small fraction of the overall wall-clock time. To avoid this scenario, developers should follow the steps below:

1. **Find a realistic problem to try and accelerate.**

   Performance optimisation is often guided by toy problems that are simple to write and debug. Such problems, though, may end up having significantly different performance characteristics from a realistic simulation that a user would actually run. This can lead to premature optimisation as the wrong hotspots will be targeted. As a relevant example, one could work hard to optimise data movement for the application of low-order matrix-free stencils since the code will be memory-bound (see section 2.2.3) and so data movement would be the predominant cost. However, matrix-free methods are usually only applied at high-order [2], at which point the code will be compute-bound and the hotspots will be different.

2. **Identify the hotspots.** ...

3. **Create a performance model.** ...

4. **Optimise the code.** ...

5. **Repeat?** ...

2.2.2 Achieving parallel efficiency

In order to run massive simulations, codes must be able to exploit the massive 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 A strong-scaling investigation would solve the same problem but on an increasing number of processors and measure the time-to-solution. 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. If strong-scaling efficiency is poor, this suggests that there are sizeable portions of the code that are getting run in serial on each process, rather than being divided up. 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).

Weak-scaling Weak-scaling differs from strong-scaling by, rather than measuring the decrease in time-to-solution for a problem of fixed size, recording the time-to-solution for 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. 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. 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.

Taken together, these two metrics provide a relatively good indicator as to the suitability of a code for running on massively parallel computers, which is naturally essential for simulation codes to be effective. More informative measures of performance that take into account things such as convergence rates also exist [3], but we eschew such measures here because they fall under the remit of the design of the stencil itself, rather than the stencil *iteration*, which is the focus for this work.

2.2.3 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. Doing this 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.

As a first step, one should profile the performance-critical kernels in the application to determine their floating-point throughput (the higher the better)

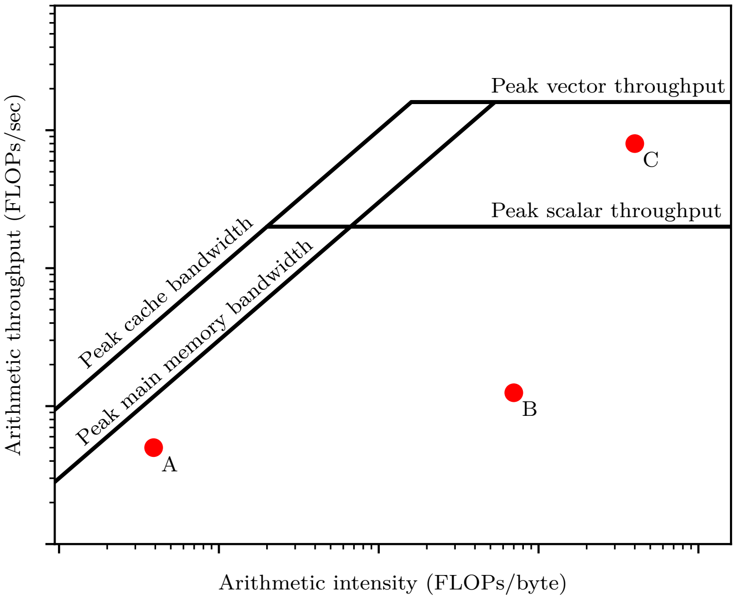


Figure 3: Simplified roofline model. The points A, B and C represent computational kernels with different performance characteristics.

and their arithmetic intensity (AI). Arithmetic intensity is a ratio of the number of FLOPs performed per byte of memory accessed. To simplify, a kernel with low arithmetic intensity reads a lot of data from memory but the processor does not need to perform many calculations, and a kernel with high arithmetic intensity performs a lot of calculations per byte of memory accessed. An example of a kernel with low AI would be `z[i] = x[i] + y[i];`; 1 FLOP (addition) is performed for every 3 data accesses (2 reads and 1 write). Assuming that `x`, `y` and `z` are 8 bytes each, this gives this operation an AI of 1/24. In contrast, a kernel with a high AI might include operations of the form `y[i] = sqrt(x[i]);`. Operations such as square roots and division typically require a large number of FLOPs, increasing the AI.

The key insight that AI gives us is that some applications are limited by the rate at which *memory is accessed*, and some applications are limited by the rate at which *chips perform FLOPs*. These are typically referred to as *memory-bound* and *compute-bound* respectively. It is worthwhile to note that there is trend in modern chips for floating-point throughput to increase at a faster rate than memory bandwidth, and so more and more codes are becoming memory-bound.

To determine which regime a kernel belongs to, it can be added to a *roofline* plot similar to the one shown in Figure 3 [16]. In a roofline plot, the performance limits of the machine are indicated by the solid lines. To the right, where AI is high, throughput is limited by the speed of the chip and so the line is flat. To the left, though, the AI is low and so throughput is dependent upon the rate at which data can be supplied to the chip. The plot shows two lines for the possible bandwidths to reflect the fact that different problems will have different working-set sizes and so the bandwidth needs to match the appropriate level in the memory hierarchy. Similarly, two lines are shown for peak throughput since vectorisation will usually increase the throughput by a factor of 4 or 8.

Looking at Figure 3, we can see how plotting kernels on this can guide the optimisation effort. To use the example points (shown in red):

- Kernel *A* has low AI and so the maximum achievable throughput is unfortunately well below theoretical peak. It lies fairly close to the peak if streaming data from main memory and so possible optimisations include: Reducing the working-set size to fit within cache memory - for example via tiling - making the new bottleneck the cost of streaming data from the cache rather than main memory. Reducing the volume of data that needs to be streamed; this would increase the AI.

- Kernel *B* has high AI and so should be able to achieve close to peak performance. However, 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.

2.3 Optimisations for stencil computations

Depending on the local kernel, stencil application can be either compute- or memory-bound. If the kernel has high AI, then the iteration will be compute-bound and throughput-oriented optimisations will be worthwhile. By contrast, if the kernel has low AI, then it will be memory-bound and data movement optimisations will be the ones to yield results. The key thing to observe is that, if compute-bound, the local stencil is doing a lot of work and hence all of the 'hot loops' that drive 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 language. Therefore, in general, optimisations for stencil languages should focus on improving memory accesses.

We remark briefly that one notable exception to this is vectorisation. One may wish to apply *inter-element* vectorisation at the level of the stencil language, that is, to loop over multiple entities in the iteration set (e.g. cells) at the same time, one per vector lane. Such an approach has been implemented in (a branch of) PyOP2, and demonstrated to be performant [15].

2.3.1 Locality optimisations

From the roofline in Figure 3 one can see that, for a memory-bound code (low AI), 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 practice there are several layers of cache and this represents a simplified picture. 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.

Whenever a piece of data is loaded into the cache, the hardware attempts to take advantage of the *spatial locality* of the data and loads in adjacent entries as a *cache line*, under the assumption that these will be required in subsequent computations. The hardware also makes the reasonable assumption of *temporal locality*, that data loaded into the cache may be used multiple times, and so cache lines persist and only the least recently used is evicted and replaced.

To demonstrate the impact of these hardware optimisations, consider an example where the data layout is so poor that only one entry from each cache line is ever used and where data is never reused. A cache line is typically 64 bytes so for double precision floats (8 bytes) this increases the *effective working-set size* by a factor of 8. This difference can cause us to exceed the working-set restrictions for a particular cache level and restrict the effective memory bandwidth to the lower level in the memory hierarchy, a difference that frequently incurs an order-of-magnitude hit to performance.

To avoid this, 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 the iteration domain is subdivided into small *tiles* to exploit data reuse between the faces of stencils; and *kernel fusion*, where separate stencils are executed in a single loop to take advantage of the data shared between them. *Time tiling* is a combination of the two and has been shown to lead to moderate improvements in performance for unstructured mesh stencils [10].

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 [4, 9].

2.3.2 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.

Meshes can (usually) be classified into one of two types: *structured* or *un-*

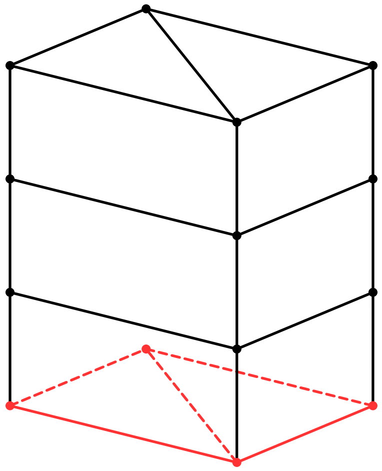


Figure 4

structured. The fundamental difference between the two is that structured grids can determine their neighbouring entities (i.e. perform the queries from Figure 2) via mathematical relations. This means that data accesses have the form `x[f(i)]`. By contrast, unstructured meshes have no structure they can exploit to determine their neighbours and hence need to tabulate these adjacency relations, resulting in accesses of the form `x[map[i]]`.

Structured meshes are known to have faster access to data than unstructured meshes. This is 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.

- *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. Note that this is unlikely to be much of a problem.

- *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 of a partially-structured 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 up the columns is regular, but addressing the base mesh is not. Such meshes are useful for simulations where the aspect ratio is very uneven. For example modelling the ‘thick’ shell of the atmosphere or ocean.

Though exploiting partial structure of meshes does speed up codes, we would like to emphasise that, in fact, the differences in performance between structured and unstructured meshes is not actually that great. 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 correct data ordering and iteration prescription. Though we do address adding support for partially-structured meshes in Section 4.1, initial work on `pyop3` targets considerations of data layout instead.

2.4 Orienting degrees-of-freedom

One significant challenge stencil applications face when applied to PDEs is on agreeing on a consistent orientation of the DoFs for shared entities.

To demonstrate the issue, consider the DoF arrangements shown in Figure 5. Figures 5a and 5c show the reference DoF arrangements for an arbitrary space on triangles, with scalar or vector DoFs on the edges respectively. Likewise, Figures 5b and 5d show the resulting DoF arrangements for the same spaces but with one of the edges flipped. In either flipped case, were the DoFs to be naively packed into a local temporary and passed to a local kernel, the results would be incorrect as the DoFs would be passed in in the wrong order.

For many cell types, the orientation problem can be avoided through renumbering the mesh such that adjacent entities agree on the orientations of any shared facets or edges. In particular this has been shown to work for simplices [**rognesEfficientAssemblyMathrmdiv2010**], as well as quadrilaterals and (some) hexes [1, 5]. ...Hexes are annoying because they cannot be oriented in parallel and some meshes (e.g. Mo(..)bius strip) are not orientable. But. We want them for tokamaks (cite).

However, for more complicated cell types with more complex symmetry groups (e.g. hexes or pyramids), the issue of orientation cannot be avoided by a simple renumbering and DoF transformations are needed to be able to collect the DoFs in a suitable reference order. For scalar-valued DoFs, one simply needs to permute the order in which DoFs are loaded into the local temporary. This can easily be done in advance and be encapsulated by, for example, a PyOP2 `Map`. This approach is insufficient for vector-valued DoFs though as components may still be pointing in the wrong direction. This is demonstrated in Figure 5d where one can see that simply permuting the DoFs on the flipped edge would not be enough. One also needs to multiply the values by -1 in order to get the vectors pointing in the right direction. The situation is further complicated in 3D where one could have two tangent vectors per DoF on each face, requiring the application of a 2×2 rotation matrix to reach consensus.

The general solution to orienting DoFs for stencil application is therefore as follows: First, one loads the (permuted) DoFs associated with a particular entity, along with a bitarray encoding the entity's orientation. Then, one can apply appropriate transformations to the loaded DoFs such that they can be correctly passed through to the local kernel. This is the approach used by Basix [14, 13], part of the FEniCSx finite element software suite []. However, to our knowledge, this is not performed by any existing stencil library.

3 Implementation

As discussed in Section 2.1.1, existing stencil languages 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 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. By contrast, a 'mesh-aware' library does not have these problems but it has to use its own custom mesh implementation. This increases the burden on the maintainers of the software and the mesh implementations will, without substantial development effort, suffer from both lack of features (e.g. I/O, parallel decomposition, adaptive refinement) and poor interoperabil-

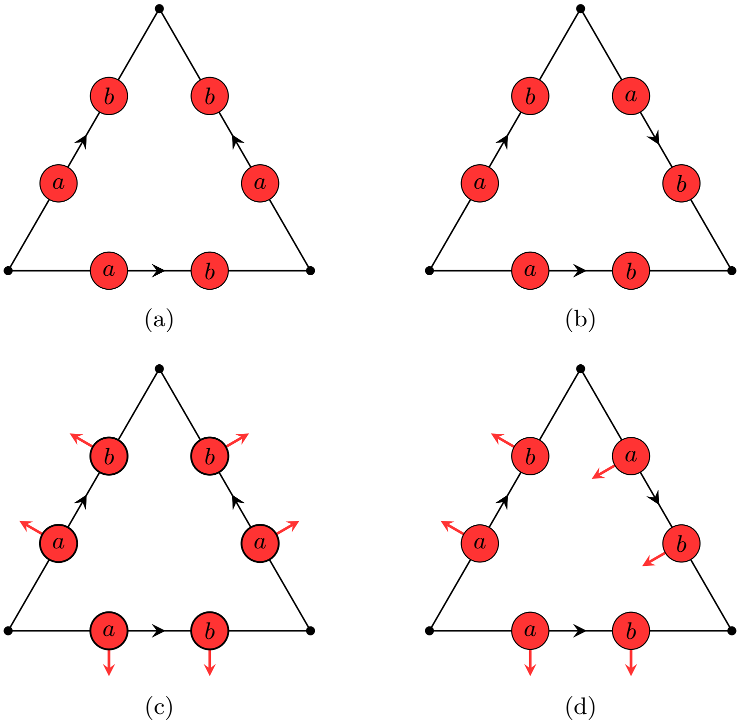


Figure 5

ity with other packages.

In this work we attempt to bridge this gap by writing a new stencil language, pyop3, that combines the advantages provided by 'mesh-aware' frameworks with a mature, external mesh implementation (DMPlex). 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.

In pyop3, users declare the iterations to be performed, the local operations to apply, and the stencil patterns for each data structure in a manner that is close to the mathematics/pseudocode. As 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:

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

Unlike PyOP2, pyop3 permits arbitrary composition of DMPlex restriction operations. One can, for example, easily describe stencils over interior facets where the stencil is composed of *the closure of the cells incident on the facet*, or, in DMPlex terminology, $\mathbf{cl}(\mathbf{supp}(p))$:

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

3.1 Data types

Like PyOP2, pyop3 has three main data types: Globals, Dats and Mats. These may be understand as data types aware of zero, one or two meshes respectively.

A Global represents some, possibly vector-valued, quantity that is shared across all processes. It has no awareness of the mesh whatsoever.

By contrast, a Dat is a vector associated with a given mesh, storing some set of values at each mesh point. It is uniquely associated with a single mesh and, since the mesh is distributed between processes, only a subset of the overall data structure is present on any single process.

Lastly, a Mat can be thought of as representing the 'overlap' between two (possibly the same) meshes. Abstractly, they are used to compute quantities that depend on the interaction between different mesh points. This can include the test and trial functions from FEM, or an interpolation matrix to interpolate values between different meshes. Since solving PDEs typically makes use of functions with *local support* (i.e. functions whose value are zero except in some

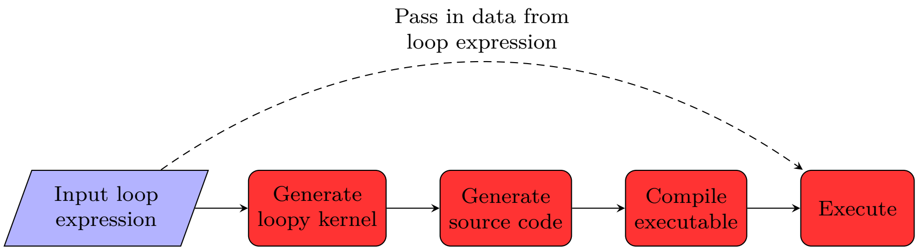


Figure 6

small region of the mesh), these matrices are typically very *sparse* as non-zeros are only present in the regions where these supports overlap.

For the FEM, where the cell integrals are defined using the closure operation, the overlap between test and trial functions is assumed to be the combination of the closures of each cell.

3.2 Just-in-time compilation

The execution pipeline is shown in Figure 6. First, and this is the main work done by `pyop3`, an input *loop expression* is converted to a loopy intermediate representation. Then, this intermediate representation is lowered to a low-level language such as C which is then just-in-time compiled to a shared library that can be executed using data provided by the original loop expression (e.g. here that means `mesh`, `dat0` and `dat1`).

Note that in this example, by using `do_loop`, we declare the loop expression and then *immediately* execute it. It is frequently desirable, for reasons of efficiency, to have a *persistent* loop expression which one can create by running `expr = loop(...)`. This is discussed in more detail in Section 3.5.

In the example above, `kernel` is an externally provided loopy kernel augmented with *access descriptors* (`READ`, `WRITE`, `RW`, `INC`, `MIN` or `MAX`) that allows `pyop3` to emit the correct packing/unpacking code. In this case, if we assume the access descriptors for `kernel` are `READ` and `INC`, then we would generate C code resembling that shown in Listing 1.

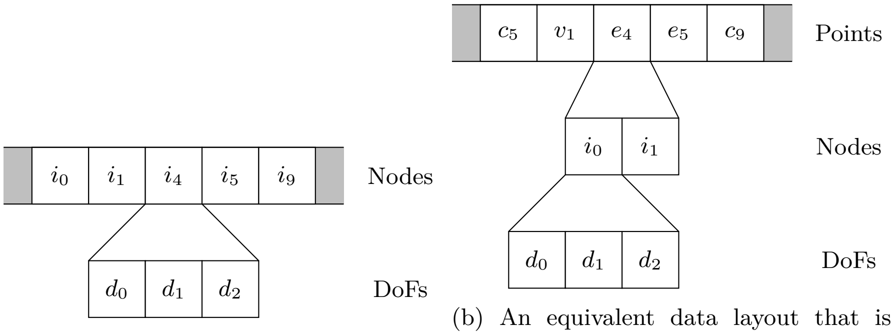
3.3 A new abstraction for mesh data layouts

The additional flexibility `pyop3` has over its precursor 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 with a `DataSet`, which is formed by associating a `Set` with some *local shape* (a tuple), termed its `dim`. To demonstrate, consider a PyOP2 `Dat` originating from some vector element discretisation applied over a mesh. At each node in the discretisation, recalling that there need not be just one node per topological entity, this `Dat` would store DoFs with some non-scalar `dim`, say, `(3,)`. Such a layout is shown in Figure 7a.

```
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 1: Simplified version of code that would be generated by `pyop3` where `kernel` has access descriptors `READ` and `INC`. `CLOSURE_SIZE` is an integer constant and would be known at compile-time.



(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) An equivalent 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, leaving cells and vertices unspecified.

Figure 7

There are a number of advantages to this approach: Firstly, code generation is very straightforward as packing/unpacking is as straightforward as iterating over the nodes with a fixed size inner loop over the DoFs. Also, this approach maps naturally to PETSc's notion of a *blocked* matrix (`Mat`) or vector (`Vec`), since a block describes contiguous data that can be addressed all together.

Despite these advantages, however, this simple data layout model *loses topological information.* PyOP2 `DataSets` store data per node, but they do not know which topological entities the nodes originated from. To counter this shortcoming, in `pyop3` we introduce a hierarchical model for describing data layout that can record, among other things, the topological entities that a given node comes from. This can be seen in Figure 7b. Rather than just having nodes and DoFs per node, we can represent the same vector-valued `Dat` using a 3-level data structure containing: mesh points, nodes per point, and DoFs per node.

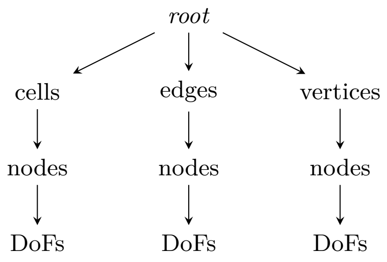
In `pyop3`, we describe these multi-dimensional, inhomogeneous data structures with a `MultiAxis`. A `MultiAxis` represents a single layer of the hierarchy and is simply a container for some positive number of `AxisPart` objects. Each `AxisPart` describes one particular class of entities in a given layer of the hierarchy and stores information such as the number of entries and how they might be addressed. The 'type' property of the typed multi-index is simply used as an identifier for the appropriate `AxisPart`. This is similar to the interface presented by Taichi [6].

To construct a hierarchical data layout, each `AxisPart` can optionally be given a sub-axis (`MultiAxis`). To demonstrate, the hierarchical layout shown in Figure 7b could be constructed using the code in Figure 8b. The tree itself is shown in Figure 8a.

Having constructed such a data layout, we address it via the use of a *typed multi-index.* This is an object of the form $((t_1, i_1), (t_2, i_2), \ldots, (t_n, i_n))$ where t_x (the 'type') indicates the correct `AxisPart` to in the hierarchy. For each `AxisPart`, and given an index into it (i_x), we use *layout functions* to determine its location in memory and these offsets are added together to get the final address. A layout function is a function that takes in an index and returns an offset. In the case of axes with constant stride this simply takes the form `off = iX * stride`, but for non-constant strides it would resemble `off = offsets[iX]`.

One major challenge presented by this new layout is that axes are no longer homogeneous. In Figure 7b for instance, not all points have the same number of nodes per point. This means that one can no longer stride over the axis by a constant value, and instead a lookup table must be used.

This approach is advantageous because it is much more natural to reason about mesh operations at the level of mesh points. DMPlex restrictions naturally map points to points instead of points to nodes, making map composition tractable. The approach also facilitates: orienting DoFs (Section 3.3.3), data layout optimisations (Section 3.3.4), and extruded and other partially-structured meshes (Section 4.1).



(a)

```
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)

Figure 8

3.3.1 Maps

When we address some data, the provided data structure is associated with a particular multi-index. When we directly address data structures, for example by doing the following:

```
loop(c := mesh.cells.index, kernel(dat0[c]))
```

Then the multi-index getting used, `c`, is simply $[(C, i)]$. This is a multi-index with only a single entry which targets all entries in the selected `AxisPart`, in this case all cells in the mesh. We remark that only the outermost axes need be indexed - the inner axes (here nodes-per-cell and DoFs-per-node) are automatically included as full slices.

Since computing stencils requires the addressing of adjacent mesh points, we use *maps* to describe which multi-indexes are required. To make things clear, a map is defined as a *function that accepts a multi-index and returns multiple multi-indexes*:

$$((t_1, i_1),) \to ((u_1^1, j_1^1),),\ ((u_1^2, j_1^2),),\ \ldots,\ ((u_1^m, j_1^m),).$$

As an example, assuming a triangular mesh, the code

```
loop(c := mesh.cells.index, kernel(dat[closure(c)]))
```

uses the cl DMPlex restriction operation to yield a map of the form

$$((C, c_0),) \to ((C, c_0),),\ ((E, e_0),),\ ((E, e_1),),\ ((E, e_2),),\ ((V, v_0),),\ ((V, v_1),),\ ((V, v_2),).$$

In an analogous way to layout functions, maps can be implemented either using index functions (i.e. $e_0 = f(c_0)$), or lookup tables (`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 memory bandwidth cost of tabulating a lookup table.

The approach just described enables arbitrary map composition because maps now work in line with how DMPlex handles restrictions, namely functions between mesh points, rather than between points and nodes.

Note that at present we restrict maps to only apply to multi-indexes of length 1. This is sufficient for unstructured meshes but a necessity for partially-structured meshes. This is discussed in detail in Section 4.1.

3.3.2 Raggedness and sparsity

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 - 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:

```
nlayers = Dat(MultiAxis(AxisPart(N)), dtype=int)
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, another MultiAxis-using data structure is used instead. Having extents also use MultiAxes is advantageous as the code generation procedure can be shared.

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

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 except that we assume that the internal dimension is logically dense, and hence do not need to specify column indices.

It should be noted however that we are assuming that the matrix is local to a single processor. Parallel sparse matrices are considerably more challenging to implement and so we defer the work to PETSc (see Section 3.4).

3.3.3 Orienting degrees-of-freedom

Having a hierarchical, ‘mesh-aware’ data layout makes it much more straightforward to correctly handle orientations when packing stencils. The way it works

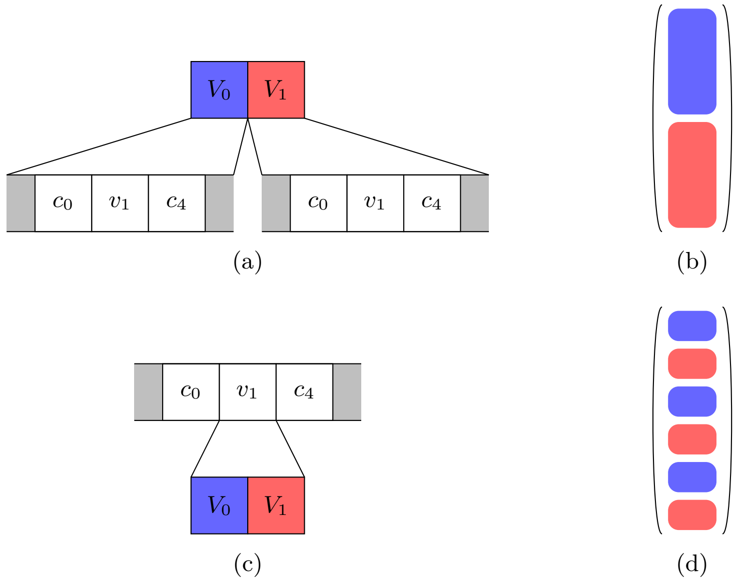


Figure 9

is as follows: 1) The DMPlex restrictions return orientation information of the mesh entities as well as the multi-index, 2) Each axis below the one selected by the map is transformed according to some predefined rule, using the orientation as a selector for the transformation.

As an example, we refer back to Figure 5d. Noting that the data layout will decompose into points, nodes-per-point and DoFs-per-node, the transformation to canonical layout is done in two steps: permuting the nodes on the edge and flipping the individual DoFs such that they point in the right direction. These operations map naturally to the different sub-axes of the data layout - the permutation applies to nodes and so can apply to the node axis, and the reflection applies to each DoFs individually and so can be applied to the DoFs axis.

3.3.4 Data layout transformations

With this decomposition of data layouts in a flexible, declarative hierarchy, it is now relatively straightforward to reason about making *data layout transformations* to improve properties such as the effective working-set size (Section 2.3.1).

Some of the possible optimisations include:

Swapping axes `pyop3` makes it straightforward to swap a pair of `MultiAxis` such that the ‘inner’ axis becomes the ‘outer’ and vice versa.

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, when applicable, permute the axes such that the mixed components are stored per mesh point, adjacent in memory. An example of 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. In addition to improving locality, one can also perform these reorderings in order to allow for efficient subset queries. Certain preconditioners require access to subsets of the mesh entities and the data layout can be modified so that the relevant DoFs are contiguous in memory.

To implement this reordering, `pyop3` simply requires that a different layout function be given to the respective `AxisParts`.

3.3.5 Other locality optimisations

`pyop3`'s abstraction also enables code transformations such as vectorisation and tiling (Section 2.3.1). Such optimisations are not data layout transformations, but transformations of the iteration set (i.e. the multi-index). In both cases, the flat iteration over some axis needs to be transformed to a set of nested loops of the form

```
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, and if tiling it would be tailored to its cache sizes.

It is valuable to note that, for unstructured meshes, tiling on its own is redundant as the amount of data shared between successive loops and prefetched by the hardware is already maximised by having an appropriate mesh numbering. The optimisation only becomes valuable when combined with kernel fusion to produce time tiling. Then the size of tiles should be chosen such that data required for both loops remains in cache between kernel invocations.

3.3.6 Enabling new research

In addition to the performance benefits espoused above, this new data layout abstraction should enable one to implement a number of new mathematical methods heretofore impossible to implement in PyOP2:

- **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. It is tricky to automate a stencil code for looping over the mesh because: a) multiple local kernels are needed, one for each degree, and b) there are 'hanging' DoFs at the boundaries between cells of differing degrees. 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`. Problem (b) is more challenging to solve and requires the addition of *constraints* to the abstraction (Section 4.3).

- **Mixed meshes** A mixed mesh is a mesh composed of multiple different types of polytope (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 polytope type. Since `pyop3` is 'mesh-aware' and can reason about the different classes of mesh points, this problem becomes trivial.

- **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 (Section 3.3.2).

3.4 Parallel design

3.5 Avoiding Python overhead

We have chosen to write `pyop3` in Python...

4 Future work

4.1 Direct addressing for partially-structured meshes

Under such a prescription, a map is therefore a *function between multi-indexes* of the form

$$\begin{pmatrix}(t_1, i_1)\\(t_2, i_2)\\\dots\\(t_n, i_n)\end{pmatrix} \to \begin{pmatrix}(u^1_1, j^1_1)\\(u^1_2, j^1_2)\\\dots\\(u^1_{k_1}, j^1_{k_1})\end{pmatrix}, \begin{pmatrix}(u^2_1, j^2_1)\\(u^2_2, j^2_2)\\\dots\\(u^2_{k_2}, j^2_{k_2})\end{pmatrix}, \dots, \begin{pmatrix}(u^m_1, j^m_1)\\(u^m_2, j^m_2)\\\dots\\(u^m_{k_m}, j^m_{k_m})\end{pmatrix}.$$

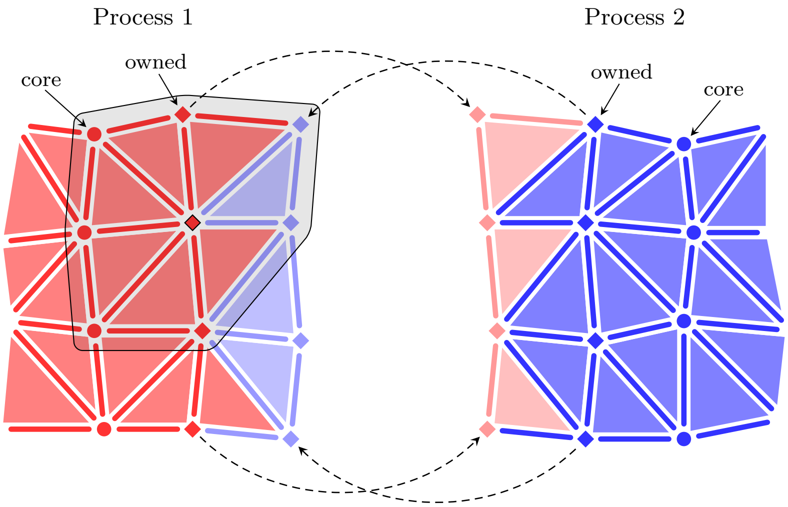
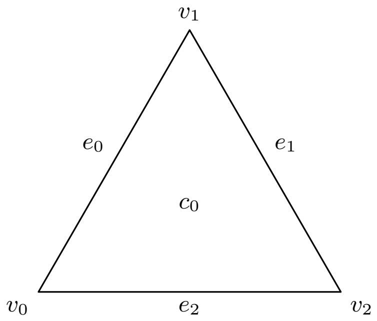
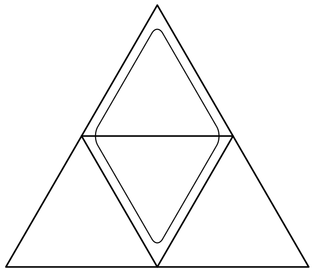


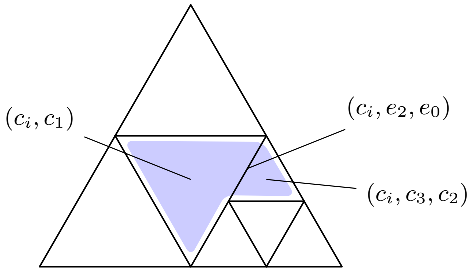
Figure 10: An example mesh distributed between two processes (red and blue). The mesh is intended for vertex patches (shaded) and so the overlap is chosen such that all required DoFs are stored locally. 'Core' vertices are stored as circles and 'owned' as diamonds. The direction of halo exchanges is indicated by the arrows.

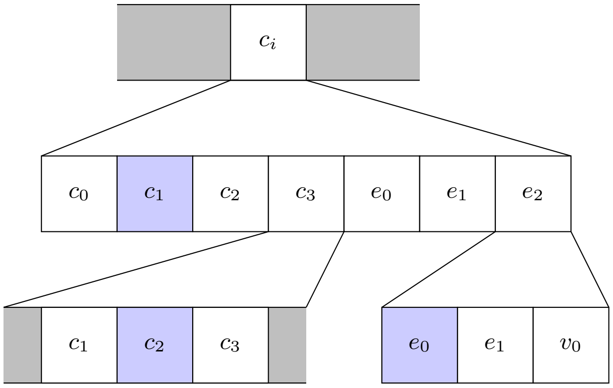
4.1.1 Extruded meshes

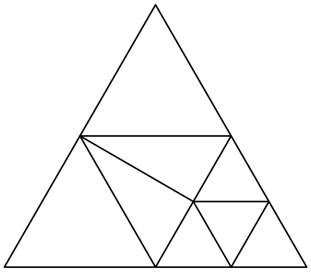
4.1.2 Regular mesh refinement

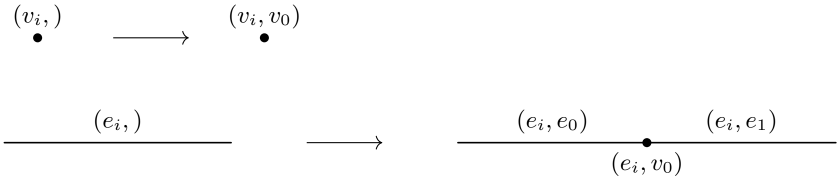


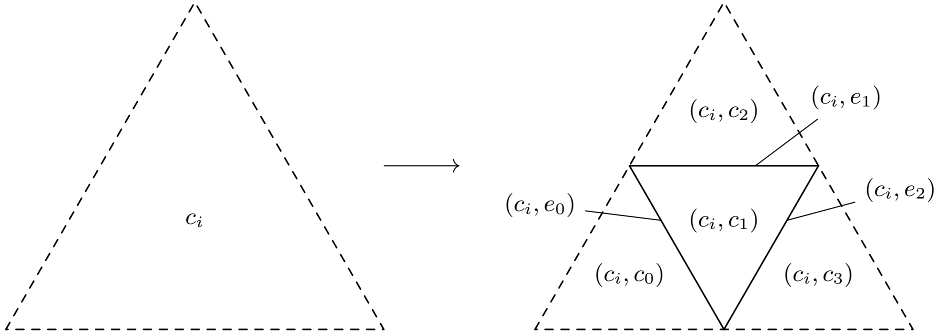



(c_i, c_1)
(c_i, e_2, e_0)
(c_i, c_3, c_2)


c_i
c_0
c_1
c_2
c_3
e_0
e_1
e_2
c_1
c_2
c_3
e_0
e_1
v_0




$(v_i,)$
(v_i, v_0)
$(e_i,)$
(e_i, e_0)
(e_i, e_1)
(e_i, v_0)



4.2 Patch preconditioners

```
loop(v := mesh.vertices, [
  loop(p := star(v), [
    kernel(dat1[closure(p)], dat2[closure(p)], "mat"),
    kernel(dat3[closure(p)], "vec"),
  ]),
  solve("mat", "vec", dat4[v]),
])
```

4.3 Constraints

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] Jed Brown et al. “Performance Portable Solid Mechanics via Matrix-Free p-Multigrid”. Apr. 4, 2022. arXiv: 2204.01722 [cs, math]. URL: http://arxiv.org/abs/2204.01722 (visited on 04/06/2022).

[3] 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).

[4] 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).

[5] 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).

[6] 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).

[7] 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).

[8] Kaushik Kulkarni and Andreas Kloeckner. “UFL to GPU: Generating near Roofline Actions Kernels”. In: *Proceedings of FEniCS 2021, Online, 22–26 March.* Ed. by Igor Baratta et al. 2021, p. 302. DOI: 10.6084/m9.figshare.14495301. URL: http://mscroggs.github.io/fenics2021/talks/kulkarni.html.

[9] 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).

[10] 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).

[11] 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).

[12] 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).

[13] 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).

[14] 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).

[15] 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).

[16] 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).