An Execution Abstraction for Compact Computational Kernels on Unstructured Meshes

Connor J. Ward

May 10, 2024

Contents

Chapter 1

Introduction

A common criticism of domain specific languages is that they are superfluous since any code they generate could have been written in C, C++ or Fortran by hand instead. This is especially true for `pyop3` since it generates a simple subset of C code focussed on reading/writing from array-like data structures. This is the sort of code that is bread-and-butter for undergraduate courses on high-performance computing; classical optimisations like loop tiling and AoS-SoA are very simple to write by hand.

The counter point, of course, is that `pyop3` is not primarily intended to be used as a simulation front-end, but instead as an intermediate representation of some higher level abstraction. DSLs can more or less only express problems that are also expressible in all of their IRs. The only exception to this is if the abstractions provide an escape hatch where one can inject custom code to perform a particular task. Escape hatches, however, are difficult to produce and "hacky". Also, the cost of implementing one can be prohibitive. It would be preferable for problems to be fully expressible in all IRs.

The key contribution of `pyop3` is that it increases the number of representable states of a finite-element-like code (whilst also cutting down on boilerplate). This increased expressiveness is beneficial as it enables higher-level DSLs (e.g. UFL) to express more problems without having to resort to abstraction-breaking internal code changes. As a consequence, implementing novel numerical methods that would have previously been infeasible are now tractable.

The remainder of this paper is laid out as follows…

Chapter 2

Background

2.0.1 Inspector-executor model

[4]
[8] [7] [1]

2.1 Domain-specific languages

2.1.1 An example of a complicated stencil function: solving the Stokes equations using the finite element method

For a moderately complex stencil operation that we will refer to throughout this thesis we consider solving the Stokes equations using the finite element method (FEM) [6]. The Stokes equations are a linearisation of the Navier-Stokes equations and are used to describe fluid flow for laminar (slow and calm) media. For domain Ω they are given by

$$-\nu\Delta u + \nabla p = f \quad \text{in } \Omega, \tag{2.1}$$

$$\nabla \cdot u = 0 \quad \text{in } \Omega, \tag{2.2}$$

$$\tag{2.3}$$

where u is the fluid velocity, p the pressure, ν the viscosity and f is a known forcing term. We also prescribe Dirichlet boundary conditions for the velocity across the entire boundary

$$u = g \quad \text{on } \Gamma. \tag{2.4}$$

For the finite element method we seek the solution to the *variational*, or *weak*, formulation of

these equations. These are obtained by multiplying each equation by a suitable *test function* and integrating over the domain. For 2.1, with v as the test function and integrating by parts, this gives

$$\int \nu \nabla u : \nabla v \mathrm{d}\Omega - \int p \nabla \cdot v \mathrm{d}\Omega = \int f \cdot v \mathrm{d}\Omega \tag{2.5}$$

Note that the surface terms from the integration by parts can be dropped since v is defined to be zero at Dirichlet nodes.

For the second equation we simply get

$$\int q \nabla \cdot u \mathrm{d}\Omega = 0. \tag{2.6}$$

In order for these equations to be well-posed we require that the functions u, v, p and q be drawn from appropriate function spaces…

2.2 Related work

Chapter 3

Mesh-like data layouts

`pyop3` was created to provide a richer abstraction than PyOP2 for describing stencil-like operations over unstructured meshes. Most of the innovation in `pyop3` stems from its novel data model. Data structures associated with a mesh are created using more information about the mesh topology. This lays the groundwork for a much more expressive DSL since more of the semantics are captured/represented.

The semantics for data kept on a mesh are not accurately captured by existing array abstractions.

Classic existing abstractions include N-dimensional array, ragged arrays and struct-of-arrays.

To provide a motivating example, consider the mesh shown in fig. 3.1. Degree 3 Lagrange elements have been used and these have 1 DoF per vertex, 2 per edge and 1 per cell. DoFs are always stored contiguously per mesh point, and so the data layout for this mesh would look something like that shown in **??**. It is clear that, due to the variable step size for each mesh point, an N-dimensional array (with $N > 1$) is a poor fit for describing the layout. One could also view the data as just a flat array (figure ZZZ), but this loses the information about the mesh points.

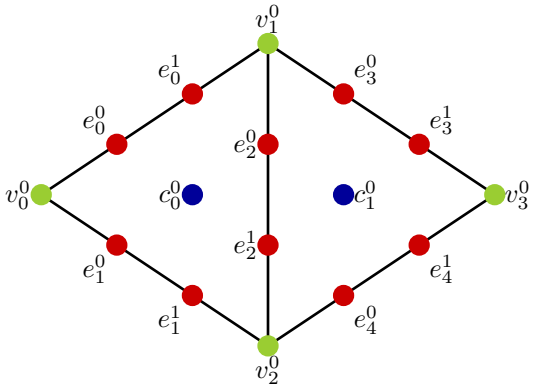


Figure 3.1: TODO

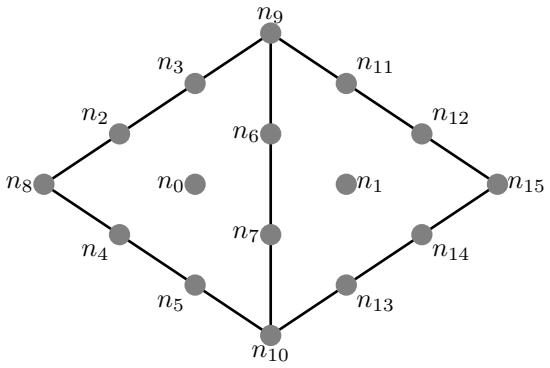


Figure 3.2: TODO

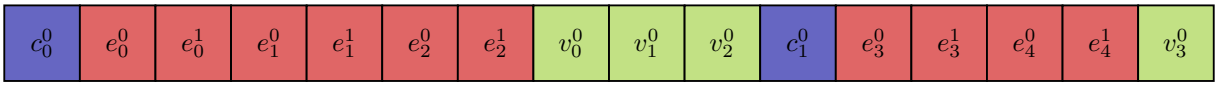


Figure 3.3: TODO

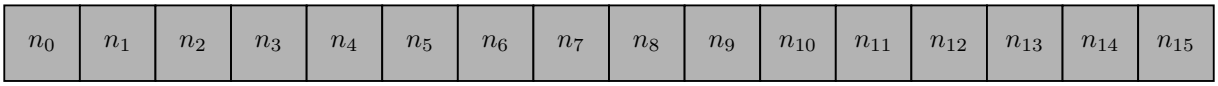


Figure 3.4: TODO

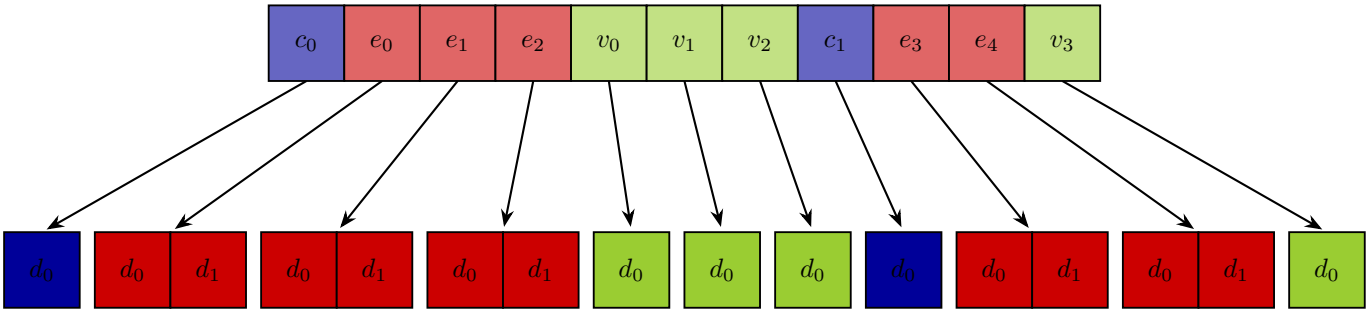
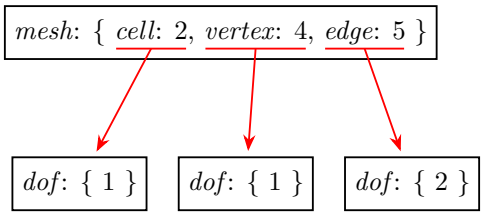


Figure 3.5: TODO



(a) TODO. For simplicity the component labels for the *dof* subaxes have been omitted.

```
axes = AxisTree.from_nest({
  Axis({"cell": 2, "vertex": 4, "edge": 5}, "mesh"): [
    Axis(1, "dof"),  # cell DoFs
    Axis(1, "dof"),  # vertex DoFs
    Axis(2, "dof"),  # edge DoFs
  ]
})
```

(b) TODO

Figure 3.6: The axis tree representing the data layout for mesh data corresponding to that shown in fig. 3.1. Note that the data has not been reordered here (see section 3.2).

We can therefore conclude that mesh data layouts require a new abstraction for comprehensively describing their semantics: *axis trees*.

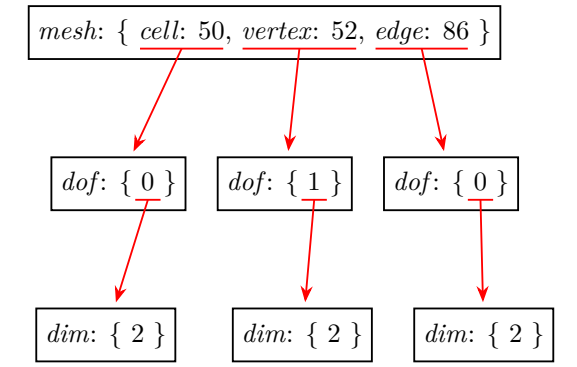
3.1 Axis trees

From **??** it can be observed that the data layout naturally decomposes into a tree-like structure. For every class of topological entity (i.e. vertex, edge or cell) there is a distinct number of DoFs associated with it.

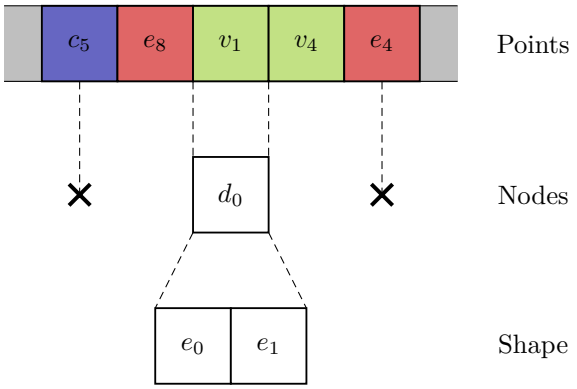
Typically, this structural information is discarded. `pyop3`, however, is capable of capturing this information through using the concept of an *axis tree*.

And axis tree is composed of a hierarchy of *axes*, and each axis has one or more *axis components*. Each axis may either be the *root* axis, with no parent, or it has a parent consisting of the 2-tuple (parent axis, parent component). In other words each subaxis is attached to a particular axis, component pair like, say, the cells of a mesh.

To uniquely identify axes and components, they are both equipped with a *label*. With these labels, one can uniquely describe a particular *path* going down the tree from root to leaf. To give an example from fig. 3.6a, one could select the DoFs associated with the edges by passing the path (as a mapping): `{ "mesh": "edge", "dof": None }`. The keys of the mapping are the axis labels and the values are the component labels. `None` is permissible for the `"dof"` axis because



(a) TODO



(b) TODO

Figure 3.7: TODO

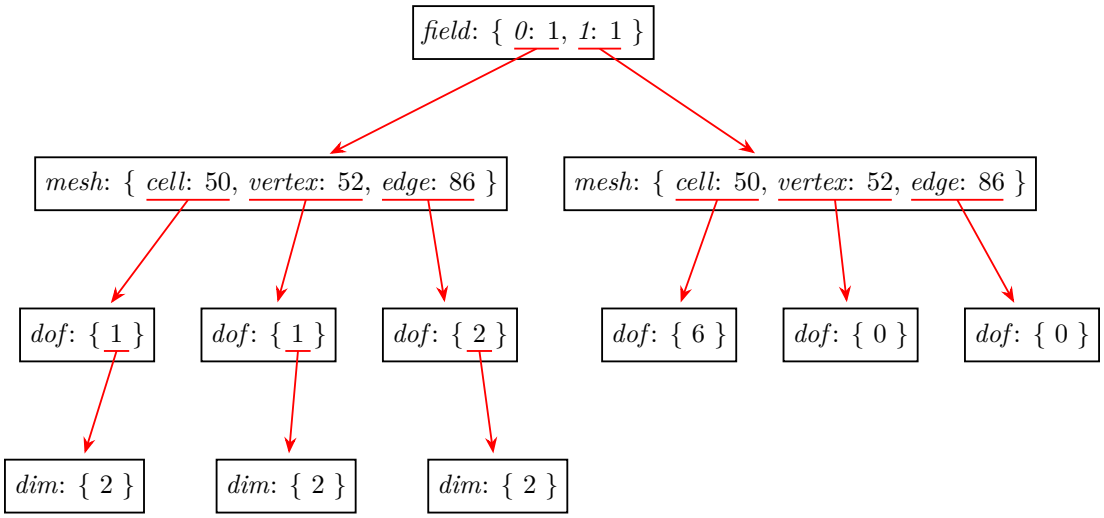
there is only a single component, and hence no ambiguity. Axis component labels must be unique within an axis, and axis labels must be unique within each possible path leading from root to leaf.

The notion of an *axis* has already been well established by numpy. If we consider a 3-dimensional numpy array with shape `(3, 4, 5)`, each dimension of the array is considered to be an axis. One can for instance change the order in which the array is traversed by specifying the axes via a `transpose` call (e.g. `numpy.transpose(array, (2, 0, 1))`).

3.1.1 Examples

Vector-valued function spaces

This approach naturally extends to tensor-valued function spaces, where the multiple inner axes may be provided to represent, for example, a small 3×3 matrix stored for every mesh point.



(a) TODO

Mixed function spaces

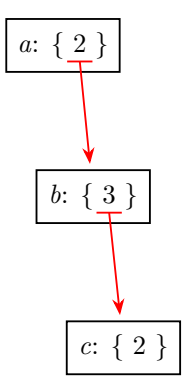
In exactly the same way as for vector-/tensor-valued function spaces, the order in which the axes are declared is flexible...

3.2 Renumbering for data locality

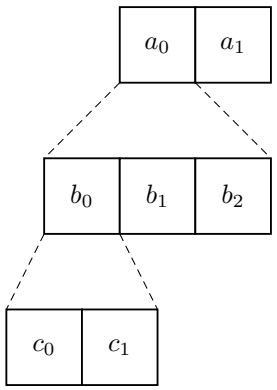
For memory-bound codes, performance is synonymous with data locality. In the case of stencil codes like finite element assembly, one should aim to arrange the data such that the data required for a single stencil calculation is contiguous in memory and can be read from memory into cache with only a single instruction.

For simulations involving unstructured meshes, data reorderings that provide perfect streaming access to memory are not possible and so renumbering strategies have been developed to try and maximise locality. For example, the data layout shown in fig. 3.3 approximates the strategy taken by PyOP2, cells are traversed according to some RCM ordering and the cell closures are packed next to the cell [5]. The is effective for finite element codes because finite element assembly (usually) involves iterating over cells and accessing the data in their closures.

In `pyop3`, we choose a simple approach and defer to PETSc to provide us with an appropriate RCM numbering for the points. This is communicated to the axis tree by giving an axis, in this case the `"mesh"` one, a `numbering` argument. This numbering consists of the flat indices of the axis and is exactly the object given to us from PETSc. This is not quite the case in parallel (see chapter 6).



(a) TODO



(b) TODO

Path	Layout function
$\{a, b, c\}$	$6i_a + 2i_b + i_c$

(c) TODO

Figure 3.9: TODO

3.3 Ragged arrays

3.4 Computing offsets

In the same way that the shape of a numpy array decribes how to stride over a flat array, axis trees are simply data layout descriptors that declare how one accesses an ultimately flat array. Indeed, in `pyop3` (flat) numpy arrays are used as the underlying data structure. It is the job of the axis tree to provide the right expression that can be evaluated giving the correct offset into the flat array.

In `pyop3`, axis trees are traversed to produce *layout functions*. These are symbolic expressions of zero or more indices that can be evaluated to give the correct offset into the underlying array. Layout functions, expressed in the symbolic maths package *pymbolic*[1], may either be evaluated given a set of indices or used during code generation.

To give a simple example, consider the axis tree and corresponding data layout shown in fig. 3.9. The tree shown here is equivalent to a numpy array with shape `(2, 3, 2)` with the numpy axes 0, 1 and 2 given the labels a, b and c respectively. Given a multi-index of the form (i_a, i_b, i_c) the correct offset into the array may be calculated with the layout function $6i_a + 2i_b + i_c$.

[1] https://documen.tician.de/pymbolic/index.html

3.4.1 The layout algorithm, step by step

The algorithm can be deconstructed into two stages:

1. Determine the right expression for describing the layout of each axis component separately. For the linear axis tree shown in fig. 3.9 this corresponds to determining the expressions $6i_a$, $2i_b$ and i_c.

2. Add the component-wise layout expressions together.

Of these, the former stage is by far the most complex and is the one that will be explained in more detail below.

In the following we will incrementally describe the algorithm for determining the right layout function for a given axis tree.

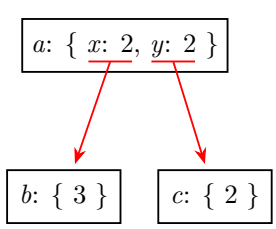
There are additional considerations in parallel that are discussed later in chapter 6.

Linear axis trees

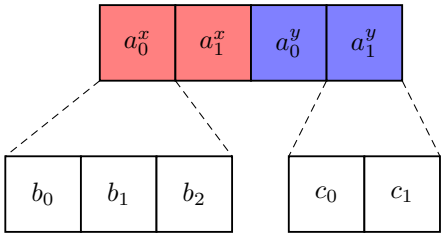
Algorithm 1 Algorithm for computing the layout functions of a linear (single component) axis tree such as that shown in fig. 3.9a. The function is initially invoked by passing the root axis of the tree.

```
def tabulate_layouts_linear(axis: Axis):
  layouts = {}

  # post-order traversal
  if has_subaxis(axis):
    subaxis = get_subaxis(axis)
    layouts |= tabulate_layouts_linear(subaxis)

  # layout expression for this axis
  if has_subaxis(axis):
    step = get_subaxis_size(axis)
  else:
    step = 1
  layouts[axis] = AxisVar(axis) * step

  return layouts
```

We begin our exposition with the simplest possible case: "linear" axis trees. A "linear" tree means that the axes in the tree are restricted to be single component. Such trees are directly equivalent to numpy-like N-dimensional arrays or *tensor* objects in many domain-specific programming languages. An example of such a tree and data layout is shown in fig. 3.9.



(a) TODO



(b) TODO

Path	Layout function
$\{(a, x), b\}$	$3i_a + i_b$
$\{(a, y), c\}$	$2i_a + i_c + 6$

(c) TODO

Figure 3.10: TODO

Pseudocode for determining the right layout function for a linear axis tree is shown in algorithm 1. The axis tree is traversed in a post-order fashion with subaxes handled first (the reason for this is made clear in section 3.4.1). At each axis, since we only require affine expressions, the layout function is simply the symbolic expression `AxisVar(axis) * step`, where `AxisVar(axis)` is a pymbolic `Variable` object and step is an integer corresponding to the size of the subtree as seen from that axis.

Multi-component axis trees

When multi-component axis trees are introduced, a number of things change: First, there are now multiple layout functions per axis. This is one per `(axis, component)` pair. Second, the affine indexing used in the linear case above must now also include offsets.

This is shown in fig. 3.10. The root axis of the axis tree now has two components, given the labels x and y, each with their own subaxis (labelled b and c). The layouts of the (a, x) part of the tree are effectively unchanged from the linear case, but the y component of axis a now clearly carries an offset. This is shown in the layout functions in fig. 3.10c.

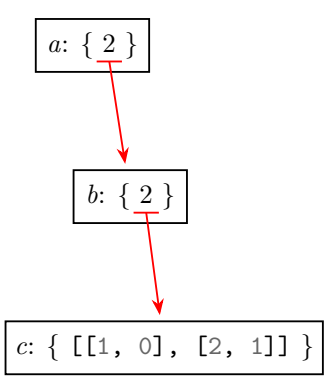
The modifications from algorithm 1 required to determine the right layout function for a multi-component axis tree are relatively straightforward. The modified algorithm is shown in algorithm 2 with the core changes labelled and highlighted in red. These core changes are:

A The post-order traversal must now be over *per-component* subaxes, so a loop over axis components is required.

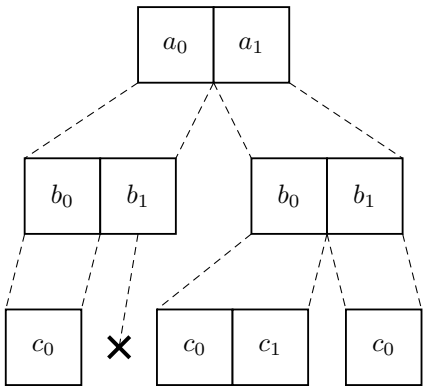
B The layout functions are now stored per `(axis, component)` pair, and an additional offset, named `start`, is added.

Algorithm 2 Algorithm for computing the layout functions of an axis tree where any of the contained axes may have multiple components.

```
def tabulate_layouts_multi_component(axis: Axis):
  layouts = {}

  # post-order traversal
  for component in axis.components:                          } A
    if has_subaxis(axis, component):
      subaxis = get_subaxis(axis, component)
      layouts |= tabulate_layouts_multi_component(subaxis)

  # layout expressions for this axis
  start = 0
  for component in axis.components:                          } B
    if has_subaxis(axis, component):
      step = get_subaxis_size(axis, component)
    else:
      step = 1
    layouts[(axis, component)] = AxisVar(axis) * step + start
    start += step

  return layouts
```



(a) TODO



(b) TODO

Path	Layout function
$\{a, b, c\}$	$[[0, 1], [1, 3]]_{i_a,i_b} + i_c$

(c) TODO

Figure 3.11: TODO

Ragged axis trees

3.5 Data structures

Thus far we have only discussed the *specification* of how data is stored in `pyop3` and not the actual implementation. For continuum mechanics problems one typically needs to have representations for scalars, vectors and matrices. In `pyop3`, recycling the terminology from PyOP2, we call scalars `Globals`, vectors `Dats` and matrices `Mats`. All of these data structures work in parallel, and their parallel implementation is deferred to chapter 6.

3.5.1 Scalars (`Globals`)

`Globals` are the simplest of `pyop3`'s data structures. They wrap a single scalar value, which may be of any data type (e.g. `int32`, `float64`, `complex128`) and thus have a trivial data layout, hence they have no need for axis trees. It is not valid to index into a `Global` (**??**).

3.5.2 Vectors (`Dats`)

Thus far, all of the data structures that we have encountered would be stored as `Dats`. `Dats` are constructed with a single axis tree that provides the information necessary to address the underlying flat array that carries the data. Having a single axis tree, `Dats` may be indexed using a single index (**??**).

Algorithm 3 Algorithm for computing the layout functions of an axis tree where any of the contained axes may be ragged.

```
def tabulate_layouts_ragged(axis: Axis):
  layouts = {}

  # post-order traversal
  for component in axis.components:
    if has_subaxis(axis, component):
      subaxis = get_subaxis(axis, component)
      sublayouts, subtree = tabulate_layouts_ragged(subaxis)
      layouts |= sublayouts

  # layout expressions for this axis
  start = 0
  for component in axis.components:
    if has_subaxis(axis, component):
      step = get_subaxis_size(axis, component)
    else:
      step = 1
    layouts[(axis, component)] = AxisVar(axis) * step + start
    start += step

  return layouts
```

Currently `Dats` use numpy arrays as the underlying data storage mechanism, but we intend to permit further array types to enable targeting accelerator architectures like CUDA GPUs.

3.5.3 Matrices (`Mats`)

`Mats` require 2 axis trees: one for the rows of the matrix and one for the columns. They rely on PETSc `Mat` objects for the underlying data storage. To improve performance one should preallocate the matrix by constructing a `Sparsity` object and doing a simulated run of all the loop expressions so that non-zeros are put in the right places.

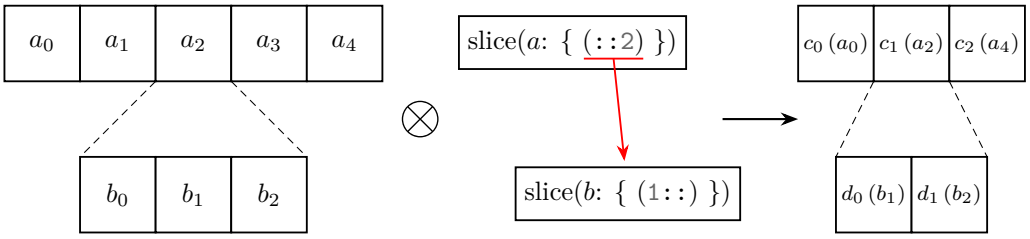
Since `Mats` have two axis trees, two indices are needed when indexing.

Chapter 4

Indexing

Axis trees provide us with a method for describing the data layout of global objects. In order to be able to execute compact stencils this is insufficient. We need a way to extract portions of the array such that they may be operated on independently. In `pyop3`, this extraction operation is referred to as *indexing*.

4.1 Fundamentals



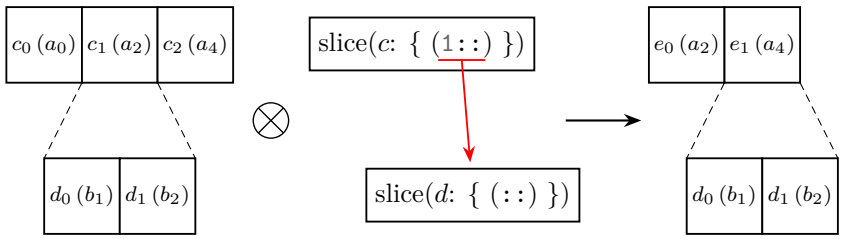
(a) Diagram of the data layout transformation. The original axis tree (left) is composed with an index tree (see section 4.1.1) (middle) to produce a new axis tree (right). The bracketed values in the resultant tree show the original indices that they map to.

Source path	Target path	Target expressions
$\{c, d\}$	$\{a, b\}$	$\{a\colon 2i_c, b\colon i_d + 1\}$

(b) The indexing information carried by the transformed axis tree that allows it to map back to the original unindexed tree.

Figure 4.1: The axis tree transformation resulting from indexing a linear axis tree with shape $(5, 3)$ with the slices `(::2)` and `(1::)` on axes a and b respectively. The resulting axis tree has shape $(3, 2)$ and different labels: c and d.

To begin with, consider indexing a linear array with shape `(10, 3)` (a matrix with 10 rows and 3 columns) with the slice `[::2, 1:]`. This is Python syntax for requesting access to every



(a) TODO

Source path	Target path	Index expression
$\{e, d\}$	$\{a, b\}$	$\{a\colon 2(i_e + 1), b\colon i_d + 1\}$

(b) TODO

Figure 4.2: TODO

other row of the first axis, and all but the first entry of the second axis. If this code were to be executed using a numpy array of those dimensions, a new array would be returned with shape `(5, 2)` containing only those elements. numpy is capable of reasoning about slices and so this new array would merely be a *view* of the same data; modifying the returned array would also modify the data in the original array.

`pyop3` takes inspiration from numpy to provide similar indexing routines for its arrays, though with support for many more indexing operations. Since `pyop3` uses code generation, any indexing transformations are necessarily lazy and symbolic, allowing for their insertion into the generated code.

We will use the numpy operation above as an initial example. An equivalent axis tree can be created... (needs diagram)

4.1.1 Index trees

Data layout transformations

4.1.2 Index composition

4.2 Loop expressions

The indexing routines demonstrated so far are not sufficient for `pyop3`'s purposes. If we consider the prototypical finite element assembly loop (**??**) we see that there is an outer loop over cells, and that the data is packed, or indexed, *relative* to the current cell.

In `pyop3`, these outer loops are described by the `loop(...)` construct. Calling `loop` creates a *loop expression*, which is a symbolic object representing the loop to be performed. The loop

```
loop(
  i := dat.axes.index(),
  dat[i].assign(666, eager=False),
)
```

(a) `pyop3` loop expression representing the operation of setting all elements of `dat` to 666. The keyword argument `eager=False` is an implementation detail required to enforce that the assignment is a symbolic rather than numeric operation.

```
for i in range(len(numpy_dat)):
  numpy_dat[i] = 666
```

(b) Python code equivalent to the loop expression shown in 4.3a where `dat` has been replaced by a numpy array (`numpy_dat`). For simplicity we assume that `numpy_dat` is 1-dimensional.

Figure 4.3: A comparison of a simple assignment loop written in `pyop3` and numpy/Python.

expression expects a *loop index*, and a collection of *statements*. The loop index represents the domain to be iterated over and it has an associated axis tree where each element of the axis tree will be visited. The statements may be further loops or arbitrary operations (see **??** for a more in-depth description).

To give an example, one of the simplest possible loops that one can write in `pyop3` is shown in fig. 4.3a. Here the loop index (`i`) is simply "all elements of `dat`" and there is a single statement that sets each entry in `dat` to the arbitrary value of 666. Note that the syntax of the loop expression is deliberately similar to that of Python (or other imperative programming languages).

The challenge here is how to represent the indexing operation `dat[i]`...

Furthermore, the indexed object is dependent upon the loop index: the target expressions of the axis tree must reference the loop index. In `pyop3` we say that it is *context-dependent*.

4.3 Maps

Maps differ from slices because they add additional shape. They have a from index

How to build a map.

Give closure as an example. Index tree.

4.3.1 Ragged maps

Ragged maps are also supported. e.g. support, star

4.3.2 Map composition

e.g. g(f(p))

Chapter 5

pyop3

the top level API: loop expressions, made of statements also kernels, access descriptors

5.0.1 Context-sensitive ...

context-sensitive things: maps and loops

5.1 Code generation

5.1.1 `pyop3` transformations

expand loop contexts

pack unpack transforms

5.1.2 Lowering to loopy

5.1.3 Loopy transformations

don't think we actually do any of these currently...

5.2 PETSc integration

also not sure that this is the right place for this section, maybe put inside an "other features" section?

Arguably this could even go after where we discuss parallel. Most of the content is focused on axis tree arrays so this is arguably a confusing distraction. Should probably have a section per chapter on PETsc matrices as there are relevant bits per chapter that apply.

Need to discuss raxes, caxes, and tabulating the rmap and cmap, they are materialised indexed things (like we have for temporaries).

Chapter 6

Parallelism

Just like Firedrake (e.g. [2]) and PETSc (e.g. ???), `pyop3` is designed to be run efficiently on even the world's largest supercomputers. Accordingly, `pyop3` is designed to work SPMD with MPI/distributed memory. As with Firedrake and PETSc, MPI is chosen as the sole parallel abstraction; hybrid models also using shared memory libraries like OpenMP (cite) are not used because the posited performance advantages are contentious [4] and would increase the complexity of the code.

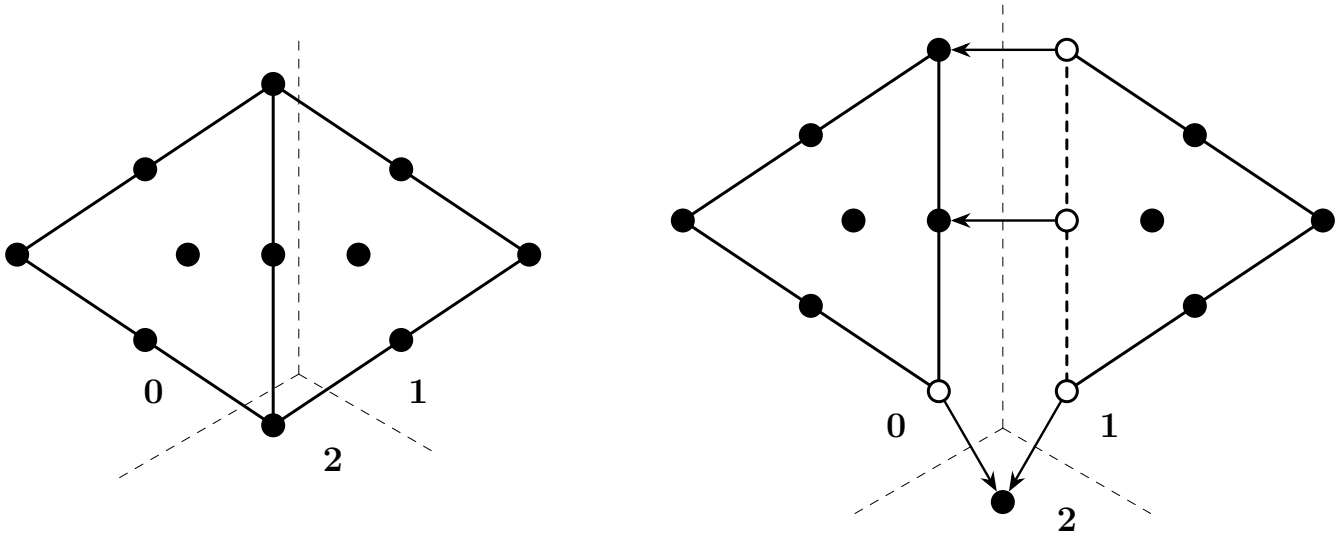
6.1 Message passing with star forests

Almost all message passing in `pyop3` is handled by star forests, specifically by PETSc star forests (`PetscSF`) [9].

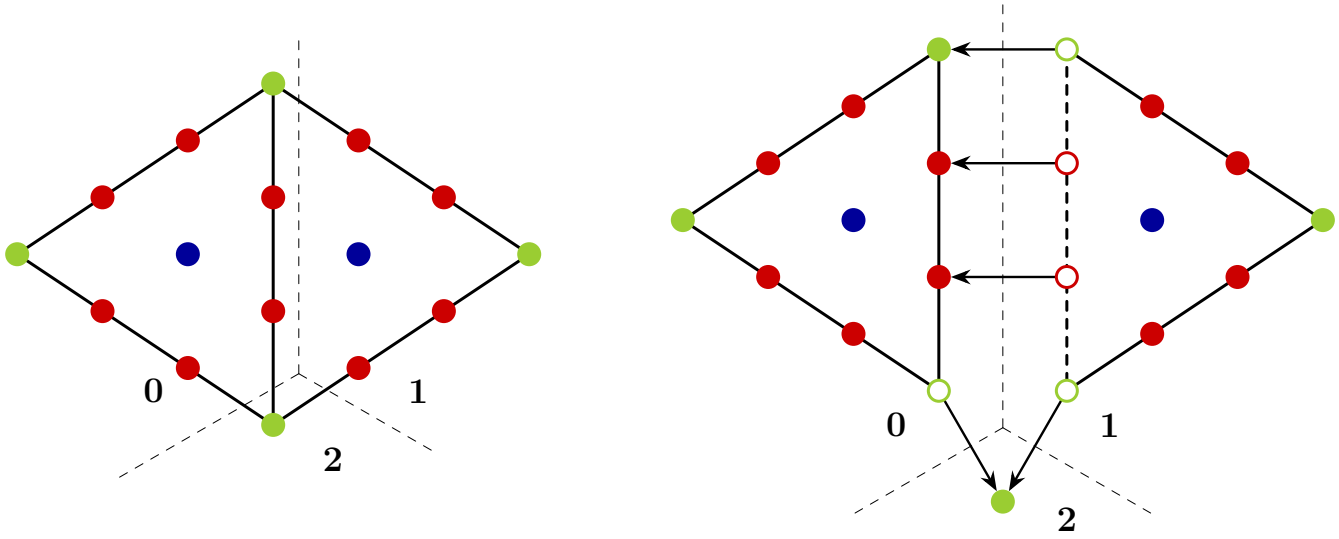
A star forest is defined as a collection of stars, where a star is defined as a tree with a single root and potentially many leaves. Star forests are effective for describing point-to-point MPI operations because they naturally encode the source and destination nodes as roots and leaves of the stars. They can flexibly describe a range of different communication patterns. For example, a value shared globally across n ranks can be represented as a star forest containing a single star with the root node on rank 0 and $n - 1$ leaves, 1 for each other rank. This is shown in Figure **??**. Star forests are also suitable for describing the overlap between parts of a distributed mesh. In this case, each star in the forest represents a single point (cell, edge, vertex) in the mesh with the root on the "owning" rank and leaves on the ranks where the point appears as a "ghost". An example of such a distribution is shown in Figure **??**.

Some terminology

- Owned Points are termed "owned" if they are present on a process and are not a leaf pointing to some other rank.



(a) TODO



(b) TODO

Figure 6.1: TODO

- Core Points are "core" if they are owned *and* are not part of (i.e. a root of) any star.

6.2 Overlapping computation and communication

In order to hide the often expensive latencies associated with halo exchanges, `pyop3` uses non-blocking MPI operations to interleave computation and communication. Since distributed meshes only need to communicate data at their boundary, and given the surface-area-to-volume ratio effect, the bulk of the required computation can happen without using any halo data. The algorithm for overlapping computation and communication therefore looks like this:

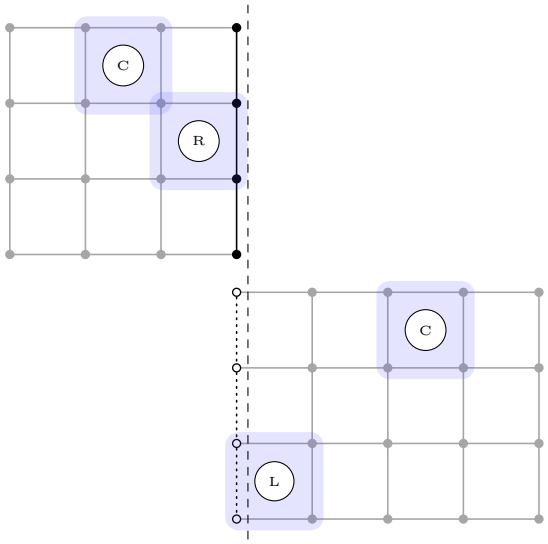
1. Initiate non-blocking halo exchanges.
2. Compute results for data that does not rely on the completion of these halo exchanges.
3. Block until the halo exchanges are complete.
4. Compute results for data that requires up-to-date halo data.

This interleaving approach is used in PyOP2 and has been reimplemented, with slight improvements, in `pyop3`.

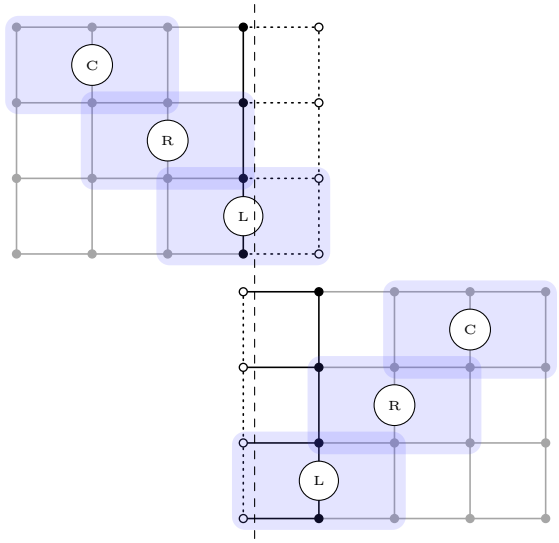
Although this interleaving approach may seem like the most sensible approach to this problem, it is worthwhile to note that there are subtle performance considerations that affect the effectiveness of the algorithm over a simpler blocking halo exchange approach. [3] showed that, in the (structured) finite difference setting, it is in fact often a better choice to use blocking exchanges because (a) the background thread running the non-blocking communication occasionally interrupts the stream of execution, and (b) looping over entries that touch halo data separately adversely affects data locality. With `pyop3` we have only implemented the non-blocking approach for now, though a comparison with blocking exchanges in the context of an unstructured mesh would be interesting to pursue in future.

6.2.1 Lazy communication

Coupled with the goal of "don't wait for data you don't need", `pyop3` also obeys the principle of "don't send data if you don't have to". `pyop3` associates with each parallel data structure two attributes: `leaves_valid` and `pending_reduction`. The former tracks whether or not leaves (ghost points) contain up-to-date values. The latter tracks, in a manner of speaking, the validity of the roots of the star forest. If the leaves of the forest were modified, `pending_reduction` stores the reduction operation that needs to be executed for the roots of the star forest to contain correct



(a) TODO



(b) TODO

Figure 6.2: TODO

values. As an example, were values to be incremented into the leaves[1], a `SUM` reduction would be required for owned values to be synchronised. If there is no pending reduction, the roots are considered to be valid.

The advantage to having these attributes is that they allow `pyop3` to only perform halo exchanges when absolutely necessary. Some pertinent cases include:

- If the array is being written to `op3.WRITE`, all prior writes may be discarded.
- If the array is being read from (`op3.READ`) and all values are already up-to-date, no exchange is necessary.
- If the array is being incremented into (`op3.INC`) multiple times in a row, no exchange is needed as the reductions commute.

One can further extend this by considering the access patterns of the arrays involved. If the iteration does not touch leaves in the star forest then this affects, access descriptor dependent, whether or not certain broadcasts or reduction are required. This is shown, alongside the rest in Algorithm ??.

PyOP2 is able to track leaf validity, but does not have a transparent solution for commuting reductions.

6.3 Performance results

[1]For this to be valid the leaves need to be zeroed beforehand.

Chapter 7

Firedrake integration

7.1 Packing

7.1.1 Tensor product cells

7.1.2 Hexahedral elements

Chapter 8

Summary

8.1 Future work

8.2 Conclusions

Bibliography

[1] Manuel Arenaz, Juan Touriño, and Ramón Doallo. "An Inspector-Executor Algorithm for Irregular Assignment Parallelization". In: *Parallel and Distributed Processing and Applications*. Ed. by Jiannong Cao et al. Red. by David Hutchison et al. Vol. 3358. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 4–15. ISBN: 978-3-540-24128-7 978-3-540-30566-8. DOI: 10.1007/978-3-540-30566-8_4. URL: http://link.springer.com/10.1007/978-3-540-30566-8_4 (visited on 11/27/2023).

[2] Jack D. Betteridge, Patrick E. Farrell, and David A. Ham. "Code Generation for Productive Portable Scalable Finite Element Simulation in Firedrake". Apr. 16, 2021. arXiv: 2104.08012 [cs]. URL: http://arxiv.org/abs/2104.08012 (visited on 06/22/2021).

[3] George Bisbas et al. *Automated MPI Code Generation for Scalable Finite-Difference Solvers*. Dec. 20, 2023. arXiv: 2312.13094 [cs]. URL: http://arxiv.org/abs/2312.13094 (visited on 01/03/2024). preprint.

[4] Matthew G Knepley et al. "Exascale Computing without Threads". In: (2015), p. 2.

[5] 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).

[6] Mats G. Larson and Fredrik Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Vol. 10. Texts in Computational Science and Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-33286-9 978-3-642-33287-6. DOI: 10.1007/978-3-642-33287-6. URL: http://link.springer.com/10.1007/978-3-642-33287-6 (visited on 03/10/2020).

[7] R. Mirchandaney et al. "Principles of Runtime Support for Parallel Processors". In: *Proceedings of the 2nd International Conference on Supercomputing - ICS '88*. The 2nd International Conference. St. Malo, France: ACM Press, 1988, pp. 140–152. ISBN: 978-0-89791-272-3. DOI: 10.1145/55364.55378. URL: http://portal.acm.org/citation.cfm?doid=55364.55378 (visited on 11/27/2023).

[8] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. "The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code". In: *Proceedings of the IEEE* 106.11 (Nov. 2018), pp. 1921–1934. ISSN: 1558-2256. DOI: 10.1109/JPROC.2018.2857721.

[9] Junchao Zhang et al. "The PetscSF Scalable Communication Layer". May 21, 2021. arXiv: 2102.13018 [cs]. URL: http://arxiv.org/abs/2102.13018 (visited on 11/11/2021).