An Execution Abstraction for Compact Computational Kernels on Unstructured Meshes

Connor J. Ward

May 22, 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]
[9] [8] [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) [7]. 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.

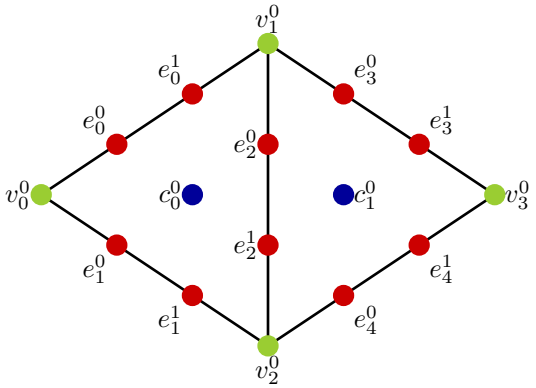


Figure 3.1: TODO

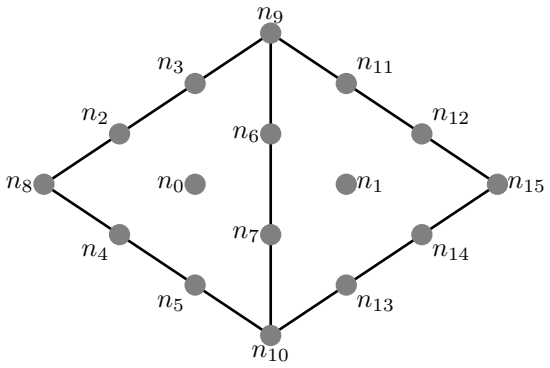


Figure 3.2: TODO

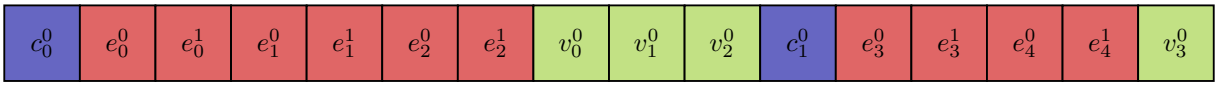


Figure 3.3: TODO

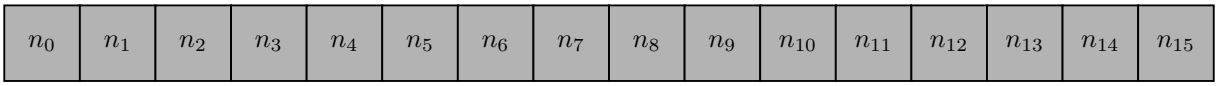


Figure 3.4: TODO

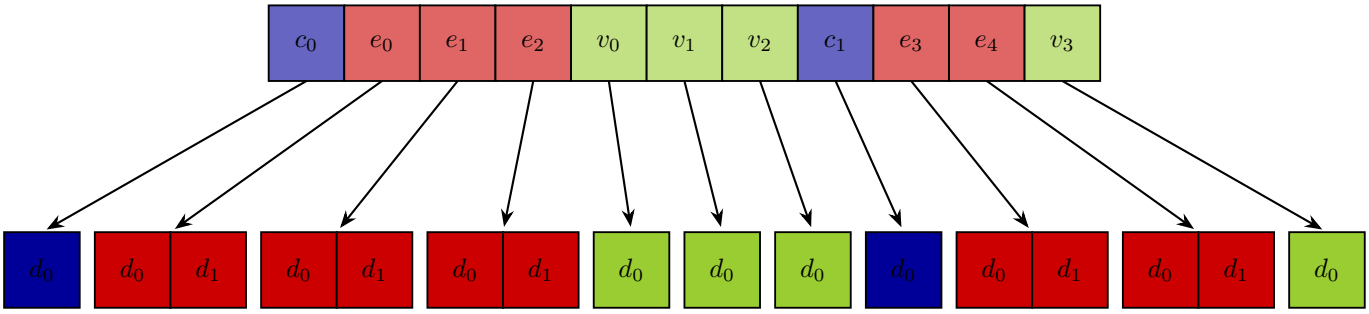
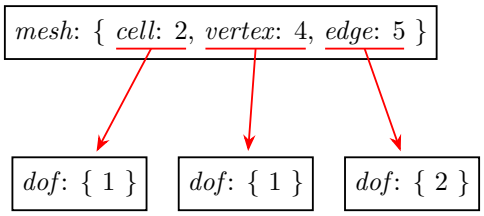


Figure 3.5: TODO



(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*.

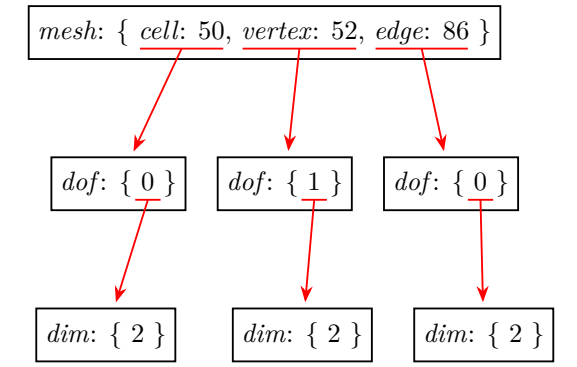
3.1 Axis trees

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.

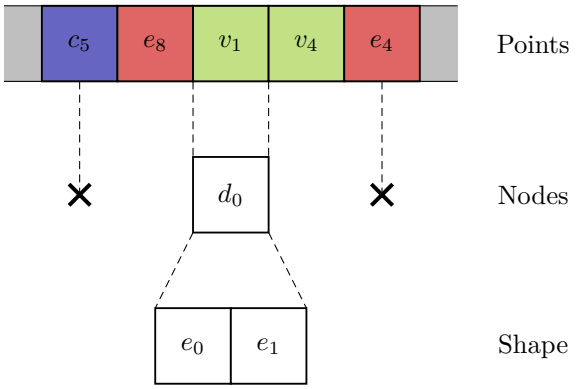
Typically, this structural information is discarded. `pyop3`, however, is capable of capturing this information through using the concept of an *axis tree*.

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

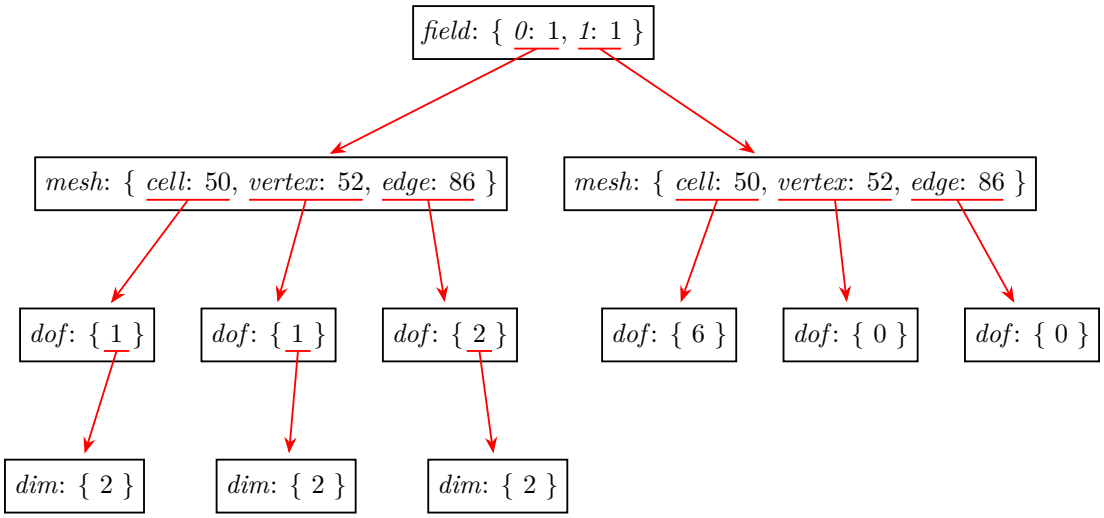
there is only a single component, and hence no ambiguity. Axis component labels must be unique within an axis, and axis labels must be unique within each possible path leading from root to leaf.

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

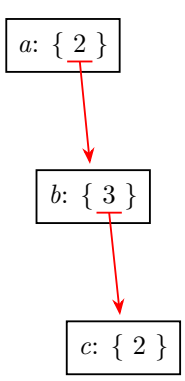
In exactly the same way as for vector-/tensor-valued function spaces, the order in which the axes are declared is flexible...

3.2 Renumbering for data locality

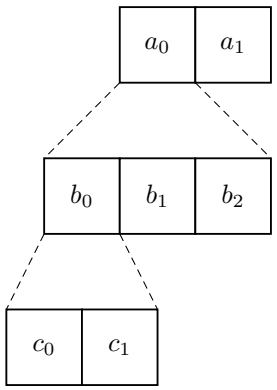
For memory-bound codes, performance is synonymous with data locality. In the case of stencil codes like finite element assembly, one should aim to arrange the data such that the data required for a single stencil calculation is contiguous in memory and can be read from memory into cache with only a single 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 [6]. 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.

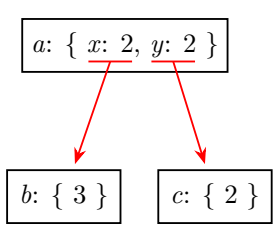
Linear axis trees

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.

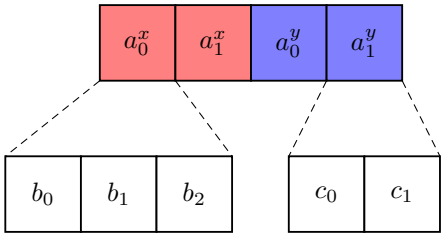
```
def tabulate_layouts_linear(axis: Axis):
  layouts = {}

  # 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.

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



(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

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.

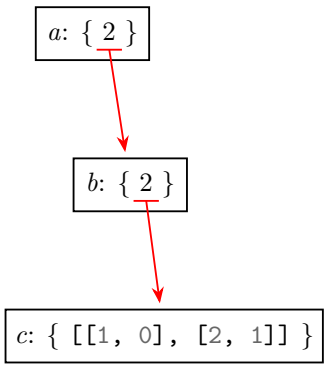
B The layout functions are now stored per `(axis, component)` pair, and an additional offset, named `start`, is added.

Ragged axis trees

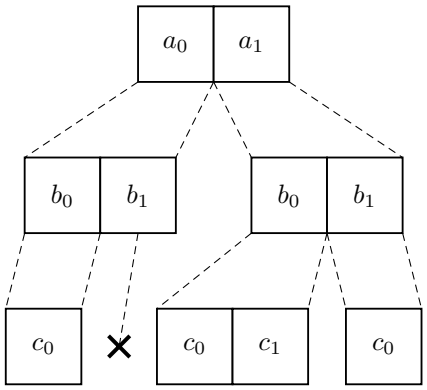
Algorithm 2 Algorithm for computing the layout functions of an axis tree where any of the contained axes may have multiple components.

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

  # post-order traversal
  for component in axis.components:
    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:
    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



(a) TODO



(b) TODO

B

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

(c) TODO

Figure 3.11: TODO

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

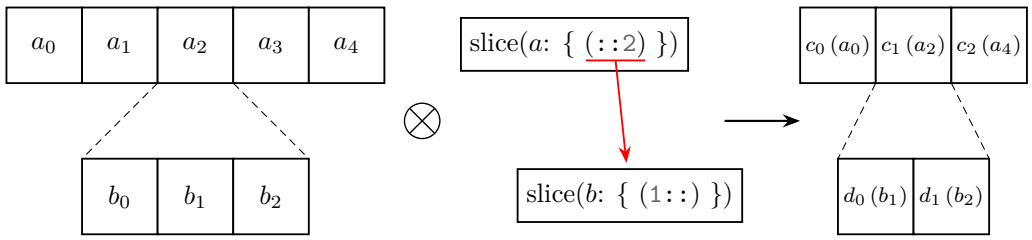
```
def tabulate_layouts_ragged(axis: Axis):
  layouts = {}

  # 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
```

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*. In many cases it is directly equivalent to the same terminology/operation as numpy.

4.1 Index trees



(a) Diagram of the data layout transformation. The original axis tree (left) is composed with an index tree (see section 4.1) (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
$\{c, d\}$	$\{a, b\}$	$\{i_a: 2i_c, i_b: i_d + 1\}$

(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.

In `pyop3`, indexing is accomplished via the use of *index trees*. Analogously to axis trees, index

trees consist of multiple *index* objects (equivalent to an axis), each of which has one or more *index components*. When an array is indexed, its associated axis tree is *transformed* via composition with an index tree. This composition operation yields a new *indexed* axis tree that understand how its entries map back to the original array. Finally this new axis tree is used to construct a new array object.

To illustrate this with a simple example, consider the indexing operation shown in fig. 4.1. It shows a slicing operation applied to a linear axis tree (left) with shape (5, 3) and with axes labelled a and b. The index tree (middle) is also linear and consists of two *slice* objects over the axes a and b respectively with the former taking every even entry in a (`::2`) and the latter taking all but the first entry in b (`1::`). The axis tree resulting from the composition of these trees is shown to the right. As shown in the figure, only the selected components of the original axis tree are present and the array has additionally received a new label numbering from zero: c and d. This resulting axis tree is in fact an *indexed* axis tree, which differs from an ordinary axis tree in a number of ways:

- It does not retabulate any layout functions. These are reused from the original, unindexed, axis tree.
- It knows about the fact that it is indexed and can map back to the original axis tree using *target paths* and *target expressions*.

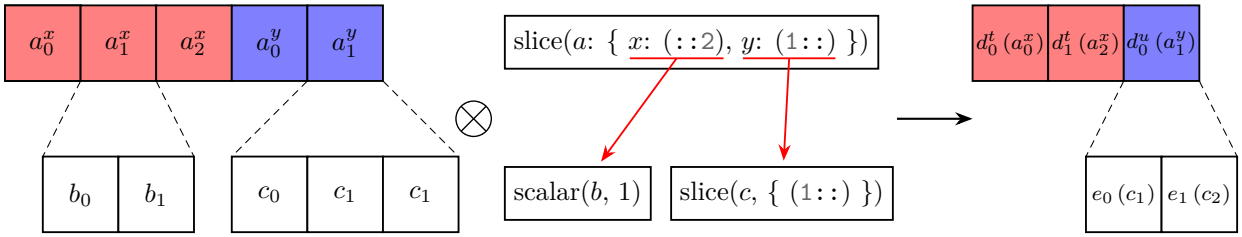
4.1.1 Target paths and expressions

When an axis is indexed, two pieces of information need to be retained for the indexed tree to be able to map back to the original array: (a) which component of the original axis tree is targeted, and (b) the correct expression taking index values from the indexed tree and mapping them to index values in the original tree. Of these the former is termed the *target paths* and the latter the *target expressions*. Due to the notion of mapping *from* the indexed axis tree back *to* the original axis tree, the indexed tree is also called the *source* axis tree and the original tree is the *target*.

For the indexing operation shown in fig. 4.1, the resulting target paths and target expressions are shown in fig. 4.3b. Since the tree is linear, only a single pair of path and expressions exist mapping the source axes c and d back to the target axes a and b. Regarding the target expressions, it can be seen that the index value for target axis a is given by an expression involving the index associated to source axis c: $i_a = 2i_c$. This encodes the information that source axis entry c_0 maps to a_0, c_1 maps to a_2 etc. This is clearly shown by the indexed tree in fig. 4.1a. The same principle applies to axes b and d, except that the different slicing pattern means that the target expression now includes an offset.

With the target expressions mapping from the indexed axis tree back to the unindexed one, and with the unindexed tree having layout functions that map from axis variables to offsets, we possess all of the information necessary for the indexed arrays to "know" what data they in fact contain. All one needs to do is to replace the axis variables in the layout functions with the target expressions from the indexed tree to get the right mapping from the source axis variables to the right offsets.

4.1.2 Multi-component index trees



(a) Diagram of the data layout transformation. ...

Source path	Target path	Target expressions

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

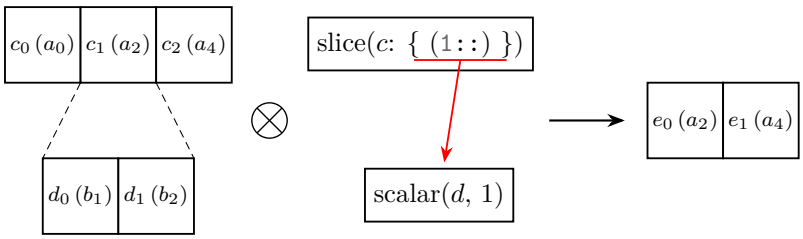
Figure 4.2: TODO The axis tree transformation resulting from indexing a linear axis tree with shape $(5, 3)$ with the slices (::2) and (1::) on axes a and b respectively. The resulting axis tree has shape $(3, 2)$ and different labels: c and d.

4.2 Index composition

4.3 Outer loops

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 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).



(a) The index composition operation. All but the first (`1::`) of axis c is selected with only the second entry in axis d.

Source path	Target path	Target expressions
$\{e\}$	$\{a, b\}$	$\{i_a\colon 2(i_e + 1), i_b\colon 2\}$

(b) Index expressions and paths relating the indexed "source" tree back to the original unindexed tree. Note how the index for axis d (i_d) is not present among the target expressions as it has been substituted for a 1.

Figure 4.3: The composition of an already indexed axis tree (from fig. 4.1) with another index tree. Since a scalar index is used, the axis tree "loses shape" and is transformed from one with shape (3, 2) to one with shape (2,). The resulting axis tree can still be mapped correctly back to the original unindexed axis tree.

To give an example, one of the simplest possible loops that one can write in `pyop3` is shown in fig. 4.4a. 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*.

4.4 Maps

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

How to build a map.

4.4.1 Ragged maps

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

```
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 walrus operator (`:=`) used here is a feature of Python 3.8 and above and is an *assignment expression*. In effect this passes `dat.axes.index()` as an argument to `loop` whilst also binding its value to the variable `i`. 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
```

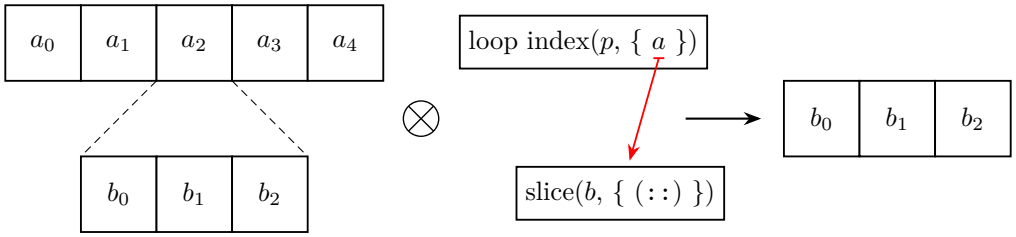
(b) Python code equivalent to the loop expression shown in 4.4a where `dat` has been replaced by a numpy array (`numpy_dat`). For simplicity we assume that `numpy_dat` is 1-dimensional.

Figure 4.4: A comparison of a simple assignment loop written in `pyop3` and numpy/Python.

4.4.2 Map composition

4.5 Data layout transformations

With `pyop3`'s axis trees it is straightforward to construct alternative data layouts for the same data. This is touched upon in section 3.1.1 for the cases of vector and mixed function spaces. Such alternative layouts can be very beneficial for improving the data access patterns of the data, but it presents a new problem: the packing and unpacking code must be different for the different data layouts. Conveniently, using index trees allows us to ignore the problem completely. Since one can think of index trees as being "proto" axis trees, it is possible to index differently laid out data structures using the same index tree and the resulting temporary will have the same shape as prescribed by the index tree.

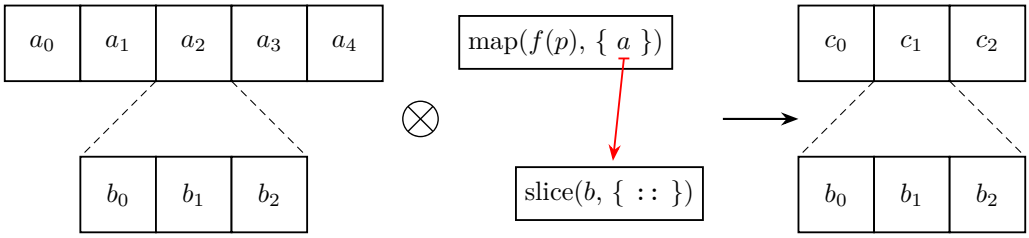


(a) The data layout transformation from the unindexed axis tree (left) to the indexed one (right). Note how the a axis is no longer present in the final axis tree as it has been fully indexed.

Source path	Target path	Target expressions
$\{b\}$	$\{a, b\}$	$\{i_a\colon i_a^p, i_b\colon i_b\}$

(b) The indexing information carried by the indexed axis tree (right). Note how the target expression for axis a is the *loop index* i_a^p. This means that the indexed axis tree cannot be interpreted without the outer loop p being present.

Figure 4.5: Index transformation equivalent to indexing a numpy array with shape `(5, 3)` with indices `[p, :]`, where `p` is an index coming from some outer loop. The resulting array has shape `(3,)` because the outermost loop has been fully indexed by `p`.

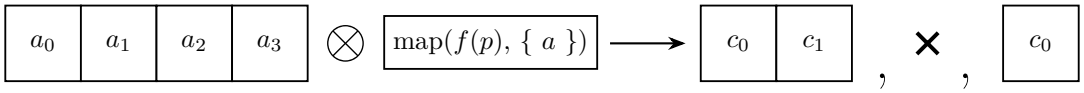


(a) The data layout transformation from the unindexed axis tree (left) to the indexed one (right). Note how the a axis has been replaced by the map axis c.

Source path	Target path	Target expressions
$\{c, b\}$	$\{a, b\}$	$\{i_a\colon f(i_a^p, i_c), i_b\colon i_b\}$

(b) The indexing information carried by the indexed axis tree (right). Using a map means that the index for axis a is an expression containing both the outer loop index (i_a^p) and an index over the shape coming from the map's arity (i_c).

Figure 4.6: Index transformation representing the packing of an axis tree with shape `(5, 3)` containing the entries referenced by the map $f(p)$, where p is some outer loop index. The map has arity 3, so the resulting array has shape `(3, 3)`.

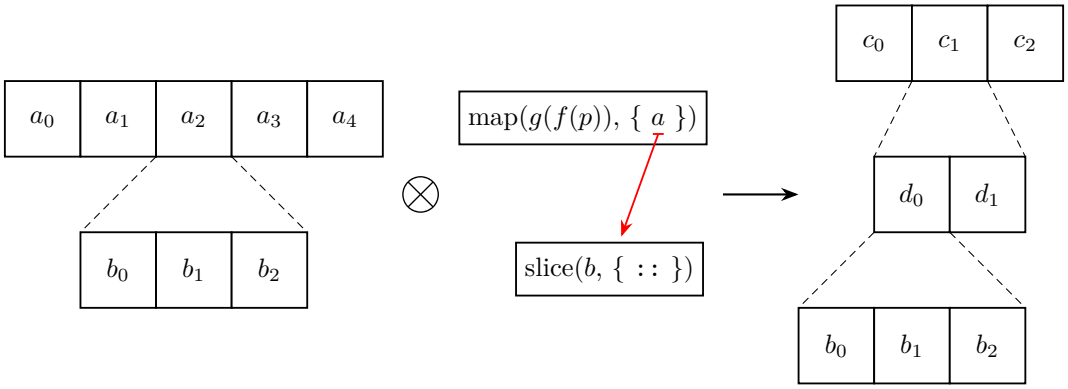


(a) TODO

Source path	Target path	Target expressions

(b) TODO

Figure 4.7: TODO

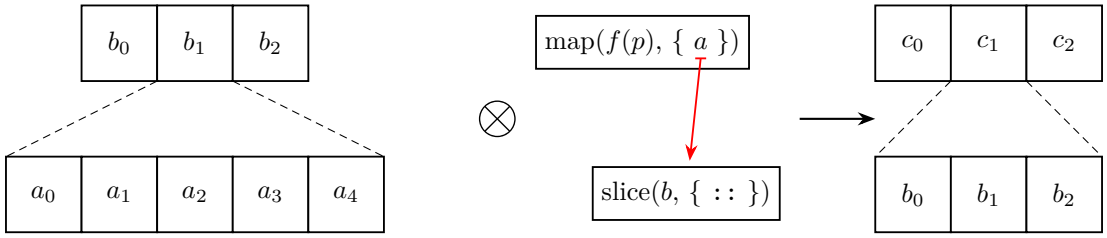


(a) TODO

Source path	Target path	Target expressions
$\{c, d, b\}$	$\{a, b\}$	$\{i_a{:}\ g(f(i_a^p, i_c), i_d),\ i_b{:}\ i_b\}$

(b) TODO

Figure 4.8: TODO



(a) The data layout transformation from the unindexed axis tree (left) to the indexed one (right). TODO

Source path	**Target path**	**Target expressions**
$\{c, b\}$	$\{a, b\}$	$\{i_a\colon f(i_a^p, i_c),\ i_b\colon i_b\}$

(b) The indexing information carried by the indexed axis tree (right). Note how the values here are entirely identical to those in fig. 4.6b. This is because the indexing operations are kept separate from any layout considerations.

Figure 4.9: Index transformation equivalent to fig. 4.6a apart from the fact that the data layout of the original axis tree has been transposed with axes a and b flipped. Despite this, the indexing transformation and resultant indexed tree are exactly the same as they were before.

Chapter 5

The execution model

Thus far we have established a new abstraction for mesh-like data structures, and an approach for symbolically representing smaller "packed" parts of them. In order for `pyop3` to be a usable library, rather than just an interesting abstraction to consider, three problems remain:

- How are the actual data structures stored in memory?
- How does one express operations to be executed?
- How are these operations executed?

These questions will be answered in this chapter, giving us a fully capable, serial-only, `pyop3` library.

5.1 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.

5.1.1 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` (**??**).

5.1.2 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 (**??**).

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 (see section 8.1).

5.1.3 Matrices (`Mats`)

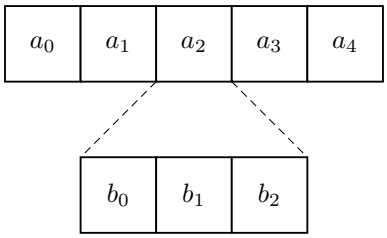
`Mats` require 2 axis trees: one for the rows of the matrix and one for the columns. They rely on PETSc `Mat` objects for the underlying data storage. To improve performance one should preallocate the matrix by constructing a `Sparsity` object and doing a simulated run of all the loop expressions so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

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

5.2 The domain-specific language

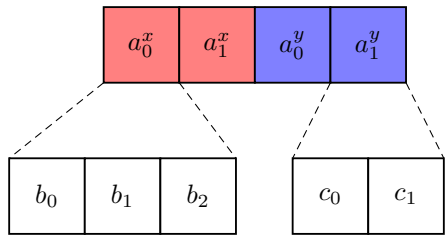
5.2.1 Loop expressions

```
loop(
  p := axes.index(),
  do_something(...),
)
```




```
for ia in range(5):
  for ib in range(3):
    do_something(...)
```

Figure 5.1: A simple example of generating code from a loop expression. The loop expression (left) loops over axis tree `axes` and performs some computation (`do_something`) for each point in the iteration. `axes` is defined to be a two-dimensional linear axis tree with shape `(5, 3)` and axis labels a and b (middle). Pseudocode for the code that is generated from this expression is shown to the right. Loop indices `ia` and `ib` correspond to looping over axes a and b respectively.



(b) TODO

```
loop(
  p := axes.index(), ...
)
```

(a) TODO

```
for ia in range(2):
  for ib in range(3):
    ...

for ia in range(2):
  for ic in range(2):
    ...
```

(c) TODO

Figure 5.2: TODO

Context-sensitive loops

5.2.2 Kernels

5.3 Code generation

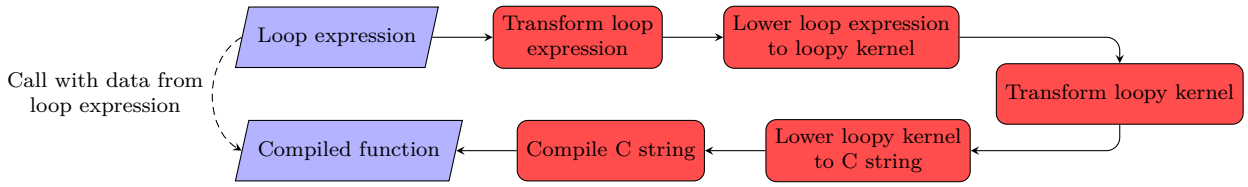


Figure 5.3: The code generation pipeline for the compilation of a loop expression into a callable function. The input ("Loop expression") and output ("Compiled function") are shown in blue whilst the intermediate processes are red. The dashed line from input expression to output function is included to represent the fact that the compiled function additionally requires data from the input loop expression in order to be usable.

To aid with the explanation, we will take a straightforward loop expression typical of finite element style codes and demonstrate the transformation and lowering stages of the compilation pipeline as they apply to it:

```
loop(
  p := a.index(),
  kernel(dat0[map0(p), :], dat1[p]),
)
```

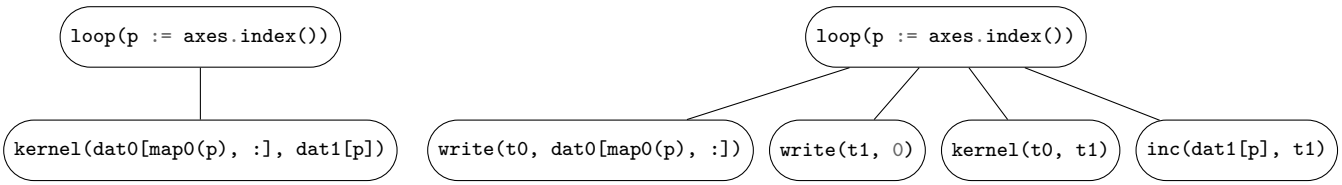
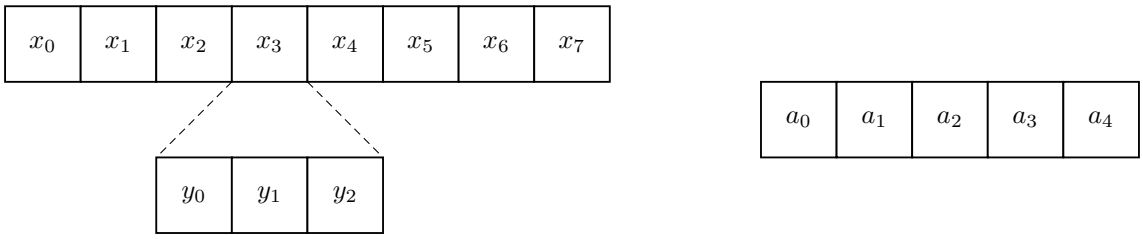


Figure 5.4: The expression tree transformation expanding implicit pack and unpack instructions for the example loop expression (section 5.3). The input loop expression is shown on the left and the output, expanded, loop expression is shown on the right. `kernel` has argument intents of `READ` and `INC` for its first and second argument respectively and so the transformed expression contains `write` and `inc` instructions where appropriate.

In this loop expression a number of terms require further explanation. Firstly, `dat0` and `dat1` are defined to be arrays with shape `(8, 3)` and `(5,)` respectively, with their axis trees appearing as follows:



The axis tree used to construct `dat1` (right) is the same as the one used in the loop index `p := a.index()`. We therefore expect that the generated loop will have the following structure (since axis a has 5 entries):

```
for ia in range(5):
  kernel(...)
```

`dat1` is entirely indexed by the loop index `p`, and so the indexed array `dat1[p]` only has size 1. The situation for `dat0` is more complicated. `map0` is a map from axis a to axis x with arity 2, and the slice notation ":" indicates a full slice over the inner axis y. The indexed object `dat0[map0(p), :]` passed through to `kernel` therefore has size 6.

Lastly, the local kernel (`kernel`) is defined to be some function taking two arguments with size 6 and intent `READ`, and size 1 and intent `INC` respectively.

Intent	Pack instruction	Unpack instruction
READ	`write(temporary, indexed)`	-
WRITE	-	`write(indexed, temporary)`
RW	`write(temporary, indexed)`	`write(indexed, temporary)`
INC	`write(temporary, 0)`	`inc(indexed, temporary)`
MIN_WRITE	-	`min(indexed, temporary)`
MIN_INC	`write(temporary, 0)`	`min(indexed, temporary)`
MAX_WRITE	-	`max(indexed, temporary)`
MAX_INC	`write(temporary, 0)`	`max(indexed, temporary)`

Table 5.1: Intent values supported by `pyop3` kernels and their corresponding pack/unpack instructions. In the instructions, the variable "`indexed`" is used to represent the indexed view of some piece of global data (e.g. `dat0[map0(p)]`) and the variable "`temporary`" is the temporary buffer for storing the materialised data. Table entries marked with a "-" indicate that no pack/unpack instruction is emitted for this intent.

```
--------------------------------------------------------------------------------
KERNEL: mykernel
--------------------------------------------------------------------------------
ARGUMENTS:
array_0: ArrayArg, type: np:dtype('float64'), shape: unknown in
array_1: ArrayArg, type: np:dtype('int64'), shape: unknown in
array_2: ArrayArg, type: np:dtype('float64'), shape: unknown in/out
--------------------------------------------------------------------------------
DOMAINS:
{ [i_0] : 0 <= i_0 <= 4 }
{ [i_1] : 0 <= i_1 <= 1 }
{ [i_2] : 0 <= i_2 <= 2 }
--------------------------------------------------------------------------------
TEMPORARIES:
t_0: type: np:dtype('float64'), shape: (6), dim_tags: (N0:stride:1)
t_1: type: np:dtype('float64'), shape: (1), dim_tags: (N0:stride:1)
--------------------------------------------------------------------------------
INSTRUCTIONS:
for i_0, i_1, i_2
    t_0[i_1*3 + i_2] = array_0[array_1[i_0*2 + i_1]*3 + i_2]
  end i_1, i_2
  t_1[0] = 0
  loopy_kernel(t_0, t_1)
  array_2[i_0] = array_2[i_0] + t_1[0]
end i_0
--------------------------------------------------------------------------------
```

Figure 5.5: Abbreviated textual representation of the loopy kernel generated for the example expression (section 5.3).

```
void mykernel(double const *__restrict__ array_0,
              int64_t const *__restrict__ array_1,
              double *__restrict__ array_2)
{
  double t_0[6];
  double t_1[1];

  for (int32_t i_0 = 0; i_0 <= 4; ++i_0)
  {
    for (int32_t i_2 = 0; i_2 <= 2; ++i_2)
      for (int32_t i_1 = 0; i_1 <= 1; ++i_1)
      {
        t_0[i_1 * 3 + i_2] = array_0[array_1[2 * i_0 + i_1] * 3 + i_2];
      }
    t_1[0] = 0.0;
    loopy_kernel(&(t_0[0]), &(t_1[0]));
    array_2[i_0] = array_2[i_0] + t_1[0];
  }
}
```

Figure 5.6: TODO

5.3.1 Loop expression transformations

5.3.2 Lowering loop expressions to loopy kernels

5.3.3 Compilation and execution of loopy kernels

Once at the level of a loopy kernel, the rest of the compilation becomes straightforward. Depending on the *target* attribute belonging to a kernel, loopy can generate an appropriate string of C code that `pyop3` writes to a file and compiles with a traditional C compiler (e.g. gcc). Once compiled, `pyop3` can load the function pointer from the shared object file, allowing it to be executed. This process is unchanged from PyOP2.

For our demo the generated C code can be seen in fig. 5.6. Unsurprisingly it looks very similar to the input loopy kernel (fig. 5.5).

Note Despite being included in the compilation flowchart (fig. 5.3), no transformations are currently applied at the level of the loopy kernel. Transformations at this level would include operations like intra-element vectorisation [10], or enabling the generated code to run on GPUs [5]. These things are considered to be future work (see section 8.1).

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.

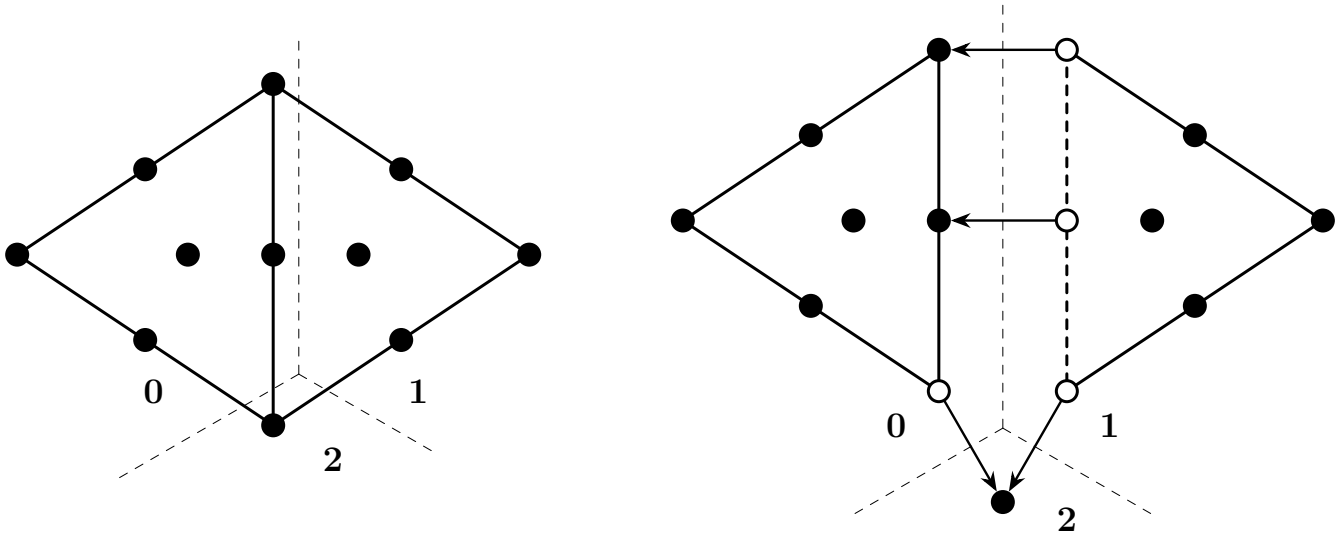
6.1 Message passing with star forests

Almost all message passing in `pyop3` is handled by star forests, specifically by PETSc star forests (`PetscSF`) [11].

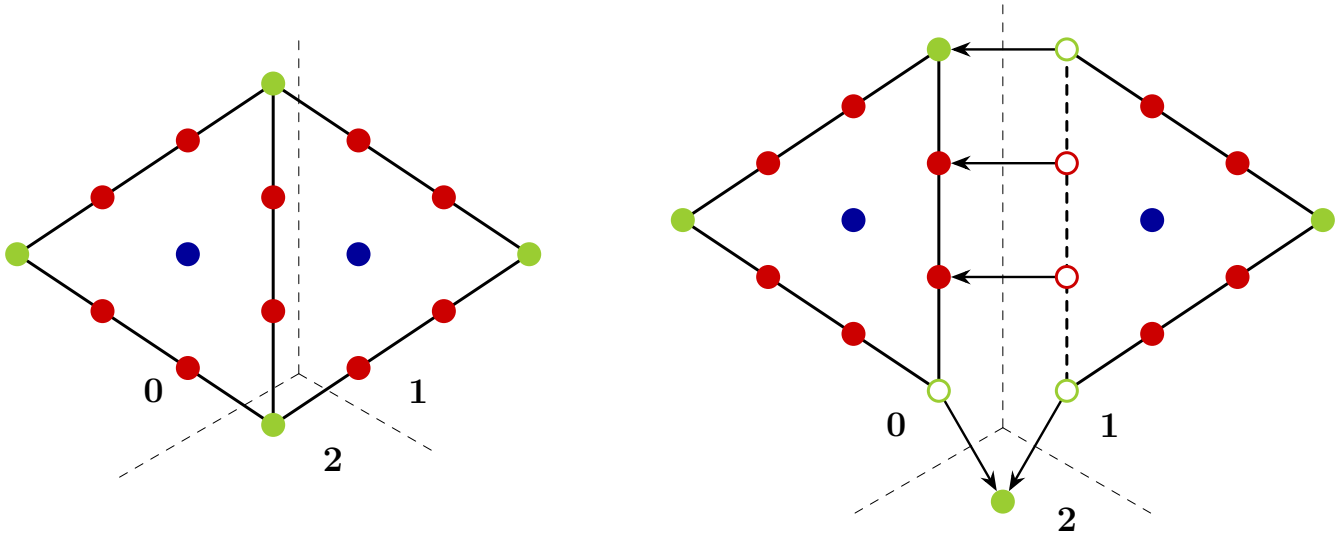
A star forest is defined as a collection of stars, where a star is defined as a tree with a single root and potentially many leaves. Star forests are effective for describing point-to-point MPI operations because they naturally encode the source and destination nodes as roots and leaves of the stars. 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:

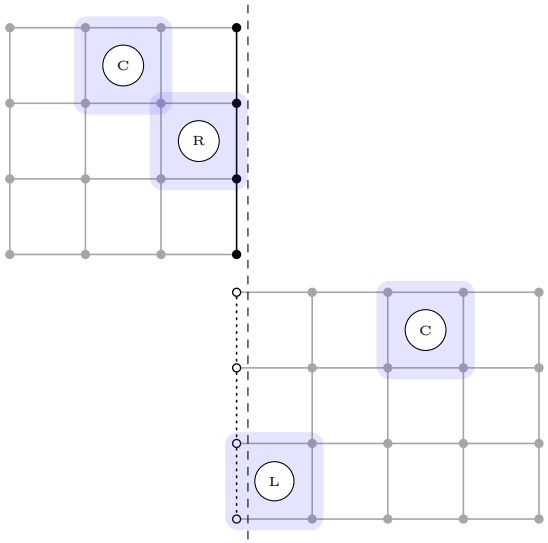
1. Initiate non-blocking halo exchanges.
2. Compute results for data that does not rely on the completion of these halo exchanges.
3. Block until the halo exchanges are complete.
4. Compute results for data that requires up-to-date halo data.

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

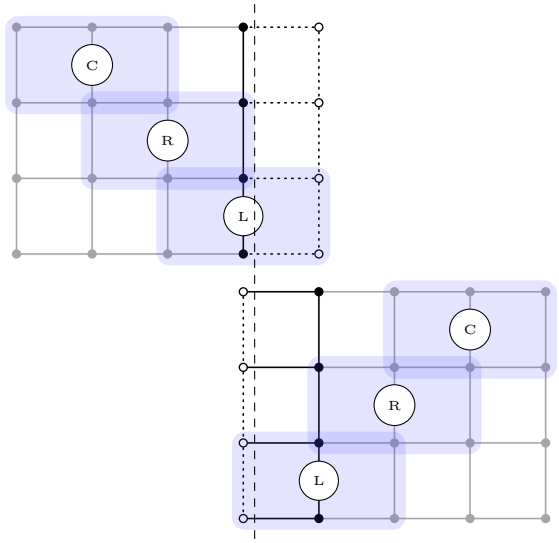
Although this interleaving approach may seem like the most sensible approach to this problem, it is worthwhile to note that there are subtle performance considerations that affect the effectiveness of the algorithm over a simpler blocking halo exchange approach. [3] showed that, in the (structured) finite 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] Kaushik Kulkarni and Andreas Kloeckner. "UFL to GPU: Generating near Roofline Actions Kernels". 2021. DOI: 10.6084/m9.figshare.14495301. URL: http://mscroggs.github.io/fenics2021/talks/kulkarni.html.

[6] 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).

[7] 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).

[8] 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).

[9] 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.

[10] 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).

[11] 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).