An Execution Abstraction for Compact Computational Kernels on Unstructured Meshes

Connor J. Ward

June 26, 2024

Contents

Chapter 1

Introduction

1.1 The problem domain: calculations involving mesh iteration

1.1.1 A motivating example: solving the Stokes equations using the finite element method

As an introductory example to a calculation requiring iterating over a mesh, we consider solving the Stokes equations using the finite element method (FEM). Our exposition will focus on the aspects of the computation that are relevant for `pyop3`, for a more complete review of FEM we refer the reader to [8] and [21].

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 and boundary Γ, omitting any viscosity or forcing terms for simplicity, they are given by

$$-\Delta u + \nabla p = 0 \quad \text{in } \Omega, \tag{1.1a}$$

$$\nabla \cdot u = 0 \quad \text{in } \Omega, \tag{1.1b}$$

$$u = g \quad \text{on } \Gamma. \tag{1.1c}$$

with u the fluid velocity and p the pressure. We also prescribe Dirichlet boundary conditions for the velocity across the entire boundary, setting u to the value of function g. Since we have a coupled system of two variables (u and p), we refer to the Stokes system as being a *mixed* problem.

Deriving a weak formulation

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 eq. (1.1), using v and q as the test functions, drawn from function spaces $\hat{V}$ and Q respectively, and integrating by parts this gives

$$\int \nabla u : \nabla v \, \mathrm{d}\Omega - \int p \nabla \cdot v \, \mathrm{d}\Omega - \int (\nabla u \cdot n) \cdot v \, \mathrm{d}\Gamma - \int p n \cdot v \, \mathrm{d}\Gamma = 0 \qquad \forall \; v \in \hat{V} \tag{1.2a}$$

$$\int q \, \nabla \cdot u \, \mathrm{d}\Omega = 0 \qquad \forall \; q \in Q. \tag{1.2b}$$

From these weak forms it is now possible to classify the function spaces for u and p. For u, we already know that the space must be vector-valued, since it stores a velocity, and constrained to g on the boundary. Equation (1.2a) further shows us that u must have at least one weak derivative. We can therefore say that $u \in V$ where

$$V = \{ \; v \in [H^1(\Omega)]^d : v|_\Gamma = g \; \} \tag{1.3}$$

p is scalar-valued, no derivatives of p are present in the weak formulation, nor are any boundary conditions applied to it and so we can write that $p \in Q$ where

$$Q = \{ \; q \in L^2(\Omega) \; \} \tag{1.4}$$

Since the values of u at the boundary are already prescribed, the function space of the test function v is defined to be zero at those nodes

$$\hat{V} = \{ \; v \in [H^1(\Omega)]^d : v|_\Gamma = 0 \; \}.$$

This allows us to drop some terms from eq. (1.2a), allowing us to state the final problem as follows:

Find $(u, p) \in V \times Q$ such that

$$\int \nabla u : \nabla v \, \mathrm{d}\Omega - \int p \nabla \cdot v \, \mathrm{d}\Omega = 0 \qquad \forall \; v \in \hat{V} \tag{1.5a}$$

$$\int q \, \nabla \cdot u \, \mathrm{d}\Omega = 0 \qquad \forall \; q \in Q. \tag{1.5b}$$

Discretising the system of equations

In order to solve this weak formulation using the finite element method we discretise the function spaces in use by replacing them with a finite dimensional equivalent:

$$V \to V_h \subset V, \quad \hat{V} \to \hat{V}_h \subset \hat{V}, \quad Q \to Q_h \subset Q.$$

Each of these discrete spaces is spanned by a set of basis functions so any function can be expressed as a linear combination of the basis functions and their coefficients. For example, we can write the function $u_h \in V_h$ as

$$u_h = \Sigma_{i=1}^{N} \hat{u}_i \psi_i^{V_h}$$

for basis functions $\psi_i^{V_h}$ and coefficients $\hat{u}_i$.

Substituting these discrete function spaces back into eq. (1.5), and discarding the basis coefficients for the arbitrary functions v_h and q_h, we obtain the discrete problem:

Find $(\hat{u}, \hat{p})$ such that

$$\int \hat{u} \nabla \psi^{V_h} : \nabla \psi^{\hat{V}_h} \, \mathrm{d}\Omega - \int \hat{p} \psi^{Q} \nabla \cdot \psi^{\hat{V}_h} \, \mathrm{d}\Omega = 0 \qquad \forall \ \psi^{\hat{V}} \tag{1.6a}$$

$$\int \psi^{Q} \, \nabla \cdot \hat{u} \psi^{V_h} \, \mathrm{d}\Omega = 0 \qquad \forall \ \psi^{Q} \tag{1.6b}$$

This can be reformulated as the (saddle point) linear system

$$\left(\begin{array}{c|c} \int \nabla \psi^{V_h} : \nabla \psi^{\hat{V}_h} \, \mathrm{d}\Omega & -\int \psi^{Q} \nabla \cdot \psi^{\hat{V}_h} \, \mathrm{d}\Omega \\ \hline \int \psi^{Q} \, \nabla \cdot \psi^{V_h} \, \mathrm{d}\Omega & 0 \end{array} \right) \left(\begin{array}{c} \hat{u} \\ \hline \hat{p} \end{array} \right) = \left(\begin{array}{c} 0 \\ \hline 0 \end{array} \right) \tag{1.7}$$

Solving the Stokes equations using the finite element method therefore boils down to constructing, or *assembling*, the left-hand-side matrix and the, here trivial, right-hand-side vector before solving for the coefficients $\hat{u}$ and $\hat{p}$.

The assembly algorithm

In order to assemble such a system, the integrals must be evaluated numerically for each pair of basis functions in the two function spaces. In the finite element method this process can be done

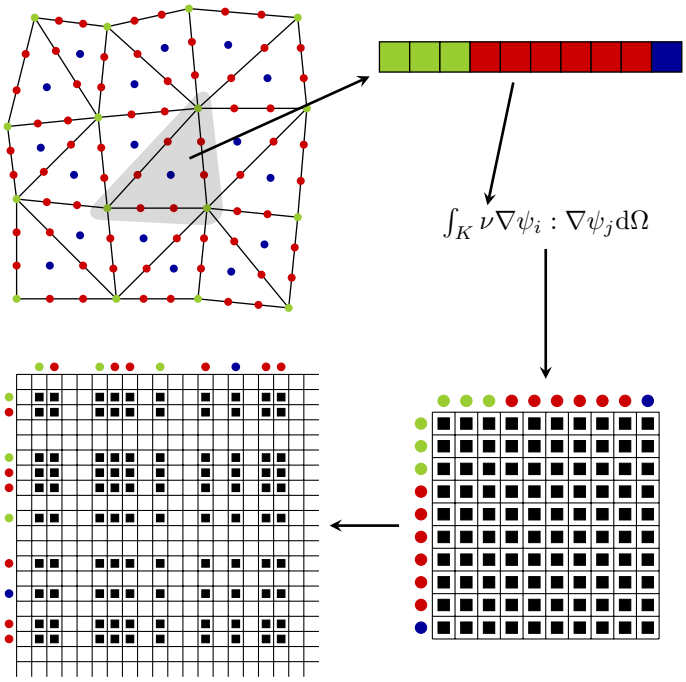
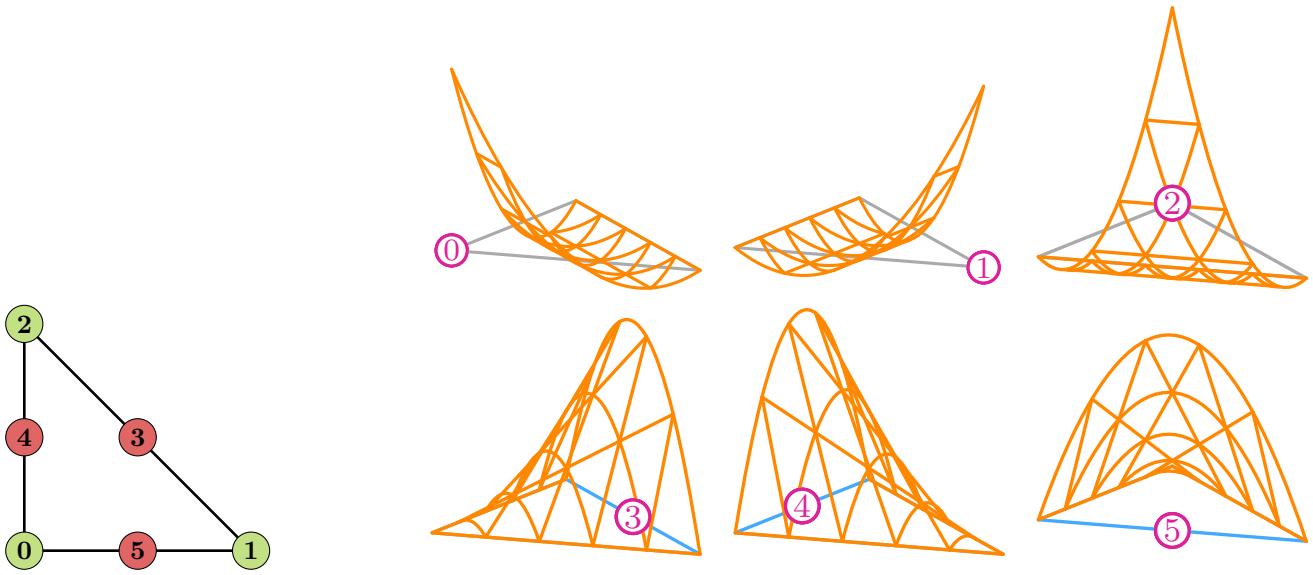


Figure 1.1: TODO

Algorithm 1 TODO

```
    FOR EACH cell IN mesh.cells:
      FOR EACH coefficient IN expression:
        collect the coefficients of basis functions that have non-
zero support over cell
      compute the integral numerically
      scatter the values of the computed integrals into the global matrix or vector
```



(a) P2, TODO (b) P2 basis functions, TODO

Figure 1.2: The P_2 (Lagrange, degree 2) finite element, [26]

efficiently because the basis functions are defined to have *local support*, that is, they are defined to be zero across almost the entire domain. This means that, instead of iterating over all pairs of basis functions, the cells of the mesh may be visited in turn and only the basis functions with non-zero support on that cell are computed with. These cell-wise contributions are then accumulated to form the global matrix. Since most of the basis functions have zero overlap the resultant matrix is *sparse*.

The basis functions are derived from a finite element definition. First formalised by Ciarlet [9], a finite element is the triple (K, P, N), where:

- K is a cell in the mesh with non-empty interior and piecewise smooth boundary,
- P is a finite-dimensional space of functions on K, and
- N is a set of linear functionals that form a basis for the dual space of P.

A simple example of a finite element, the degree 2 Lagrange element, is shown in fig. 1.2a. For this element K (the cell) is a triangle, P (the function space) is the space of order 2 polynomials, and N (the dual basis) is defined to be point evaluation at each of the nodes:

$$l_i(v) = v(x_i),$$

where l_i is the linear functional associated with node i, v is some function in P and x_i are the coordinates of the i-th node.

Figure 1.3: Scott-Vogelius element (degree 3?)

From these attributes, it is possible to determine a *nodal basis* for P by imposing that

$$l_i(\psi_j) = \delta_{ij} \quad i, j = 0, 1, \ldots, n_k.$$

In the case of the degree 2 Lagrange element this yields the basis functions

$$\begin{aligned}
\psi_0 &= 2x^2 + 4xy - 3x + 2y^2 - 3y + 1, \\
\psi_1 &= x(2x - 1), \\
\psi_2 &= y(2y - 1), \\
\psi_3 &= 4xy, \\
\psi_4 &= 4y(-x - y + 1), \\
\psi_5 &= 4x(-x - y + 1),
\end{aligned}$$

shown in fig. 1.2b.

Data structures for the finite element method

The choice of basis functions used by the function spaces has significant implications for the convergence and stability of the model. For the Stokes equations in 2D, a common choice of element pair, or *mixed* element, with properties matching the constraints given in eq. (1.3) and eq. (1.4) is the Scott-Vogelius element [25]. Shown in fig. 1.3, the element consists of a continuous vector-valued degree k Lagrange element for the velocity space, and a discontinuous Lagrange element of degree $k - 1$. Note that the Scott-Vogelius element is known to be inf-sup stable for degree ≥ 4 but we only show degree 3 here for brevity [13].

With the degree 3 mixed element in fig. 1.3, for a one-cell mesh one has 26 unknown basis function coefficients, or *degrees of freedom*: 20 for the velocity and 6 for the pressure. As each basis function yields a single unknown, the size of the linear system in eq. (1.7) is 26×26.

In eq. (1.7) the different function spaces have been partitioned to produce a block matrix system.

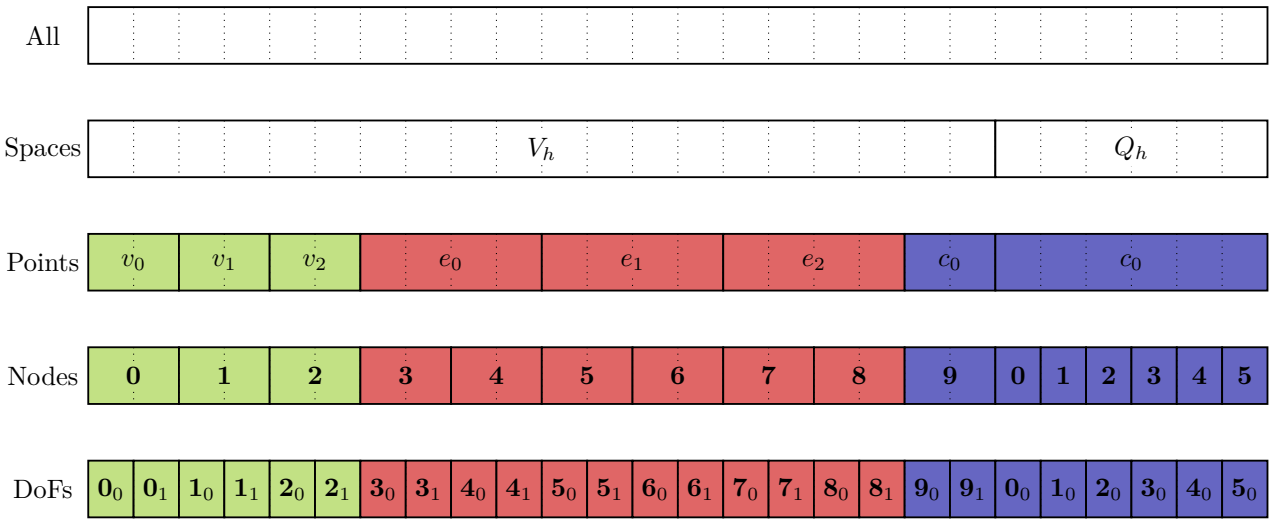


Figure 1.4: TODO

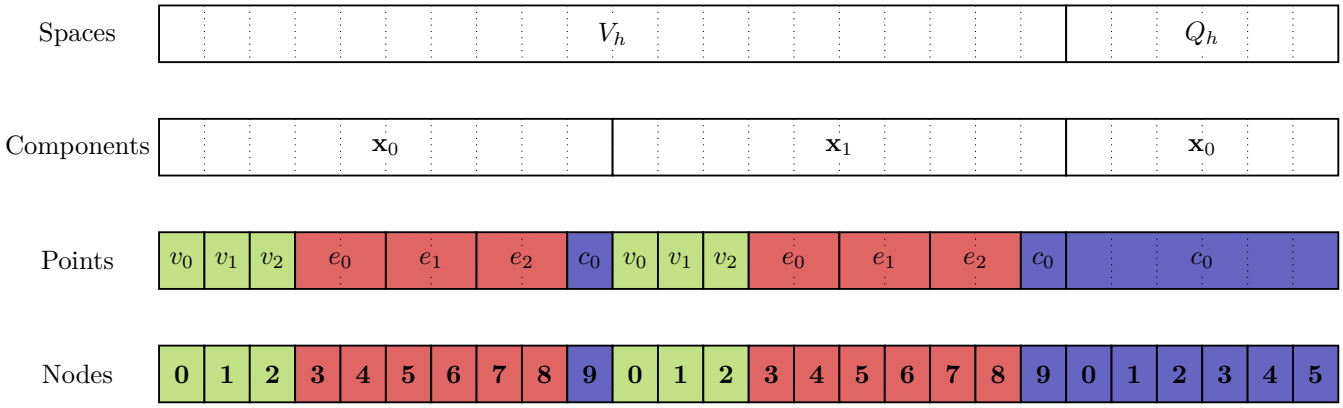


Figure 1.5: TODO

Naturally the choice of how to lay these values out in memory is arbitrary, but a common approach is to split them by function space, then by topological entity, and then by vector component (fig. 1.4)

1.2 Performance considerations

1.2.1 Distributed memory parallelism

1.2.2 Mesh renumbering

1.2.3 Data layout transformations

1.2.4 Interoperability with existing software

1.3 Execution models for unstructured meshes

At this point we have established, excluding the local kernels and global solve, the algorithms and data structures necessary to solve a finite element problem. Subsequently, we are interested in how to manifest these in software. Writing these codes by hand is prohibitively difficult: writing a performant and scalable simulation would take months or years of programmer effort and any changes to the partial differential equations (PDEs), discretisation or hardware might constitute a substantial rewrite. To counter this, numerous frameworks exist providing the building blocks from which a domain specialist, without expertise in high performance computing nor months of programmer time, might build a simulation. This creates a separation of concerns between the framework maintainers, who specialise in low-level optimisation, and the users, who can instead reason about the problem in terms of the mathematics.

In addition to the step-change in programmer productivity, high-level abstractions also facilitate advanced performance optimisations that would be very difficult to implement for a low-level code. Sometimes, high-level algorithmic changes (discretisation, solver, etc) are required to achieve acceptable performance on a given machine and having a high-level of abstraction means that tweaking these options is minimally invasive [6]. Further, having a high-level representation of the problem enables optimisations best expressed at the level of the mathematics that would otherwise be very challenging to implement (e.g. [15]).

[27] [22] [1]

For the unstructured mesh traversal operation we are interested in, we need an abstraction that:

- Expresses operations in terms of loops, compact kernels and restricted data structures,
- Supports the indirection mappings necessary for unstructured meshes (e.g. the map from cells to supported DoFs), and

- Is backend agnostic: the same code should be able to target different architectures with almost no changes.

Along with these hard requirements we also have a number of strongly desirable features. The software should:

- Be composable: support nested loops, multiple kernels and map composition,
- Interoperate with external packages (PETSc),
- Support distributed memory parallelism (i.e. MPI).

1.3.1 Existing software

A number of packages exist that already meet most or all of these requirements:

Liszt is a domain-specific language (DSL) embedded in Scala [10]. Mesh connectivity is expressed through built-in topological relations and mesh data is associated with specific topological entities. Liszt uses a custom mesh implementation with support for parallel partitioning and hence works in a distributed memory environment. Liszt is also capable of generating code for use in a multi-threaded or GPU context.

Simit is another DSL for mesh simulations [16]. It has a unique design where mesh data structures have a dual representation: they can either be viewed as a hypergraph or as a multi-dimensional tensor. This enables for both mesh-like queries to be applied to the data structures as well as enabling linear algebra operations to be expressed. Simit is capable of targeting both CPUs and GPUs without needing to change the input code, though distributed memory computing is not available.

Ebb is another DSL embedded in Lua [5]. It uses a relational database model to describe the mesh and has a 3-layer infrastructure that separates simulation code from data structure specification and different code generation targets. It has support for execution on GPUs but distributed memory computing is not available.

OP2 Unlike the frameworks mentioned above, OP2 is an *active library* that provides source-to-source translation from C, C++ or Fortran to target a range of different backends including OpenMP, CUDA and OpenCL [23]. OP2 uses a simplified model of a mesh where entities are represented as *sets*. One can store data on these sets and mappings exist between sets. Computations are termed *kernels* and are provided by the user. Distributed memory computing is possible and OP2 is even able to interleave computation and communication to provide improved scaling.

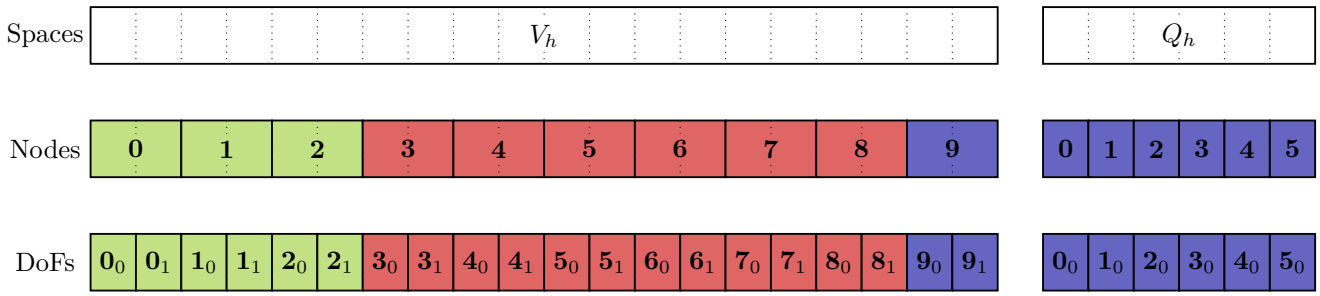


Figure 1.6: The data layout matching fig. 1.4 as it would be stored by PyOP2. Data for each function space (V_h and Q_h) are stored in separate arrays.

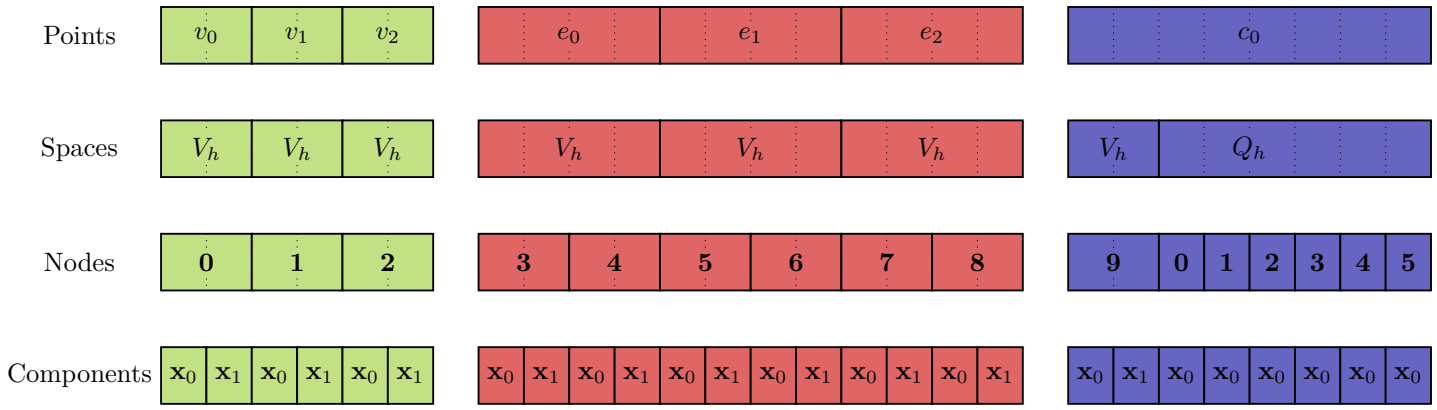


Figure 1.7: The closest possible data layout for fig. 1.4 for a library that associates unknowns with topological entities. Data for each topological entity are stored in separate arrays.

PyOP2 is a reimplementation of OP2 in Python [24]. The same core abstraction of sets, mappings and kernels is used but runtime code generation is used instead of source-to-source translation. PyOP2 currently only targets execution on CPUs though a proof-of-concept GPU extension has been created [19]. Distributed memory computing is supported and PyOP2 can also assemble sparse matrices with PETSc [4, 3, 2].

1.3.2 Data layout limitations

1.3.3 The missing abstraction

Clearly, there is something missing here. The designs of the existing libraries all require that one either use topological information in a simplified way - associating data with particular mesh entities only - or that one take ownership of the data, discarding topological information that is helpful for having a composable abstraction. To get around this difficulty we have developed a new abstraction for data layouts, termed *axis trees*, that bridges the gap between these worlds. Axis trees allow the user to describe complex data layouts of the sort shown in fig. 1.4 fully, without

needing to discard any of the topological information. As a convenient side benefit, expressing data layout transformations (**??**) becomes natural to do.

The axis tree abstraction is included in the new Python library `pyop3`. `pyop3` is a near-total rewrite of PyOP2 that aims to substantially improve its expressivity power and composability. It has support for distributed memory parallelism and integrates with PETSc.

1.4 Thesis outline

The remainder of this work is structured as follows…

Chapter 2

Foundations

`pyop3` was created to be a successor to PyOP2, and so it is instructive to review how PyOP2 works and identify any shortcomings. We will then review a number of libraries whose abstractions capture the missing behaviour.

2.1 PyOP2

Just like `pyop3`, PyOP2 is an execution model for the application of compact computational kernels over unstructured meshes [24]. It was introduced to provide the same abstractions as OP2 [23] but using runtime code generation instead of source-to-source translation. It is a core component of the Firedrake finite element framework [14].

2.1.1 Data structures

PyOP2 has no innate concept of what an unstructured mesh is. Instead, topological entities are treated as *sets*, with *mappings* between the different sets.

There are 3 different types of data structures defined in PyOP2: globally constant values, vectors and matrices. These are termed `Globals`, `Dats` and `Mats` respectively.

For more complex problems like the Stokes equations in section 1.1.1 the degrees-of-freedom (DoFs) are associated with multiple types of topological entity. In section 1.1.1 for example the unknowns are associated with the cell, edges and vertices. This means that one has to associate the DoFs for that function space with a distinct *node set*, rather than a set for a particular topological entity. As a consequence, the data structures do not know to what topological entity they refer and the library user must take responsibility for constructing the right maps from, say, cells to nodes.

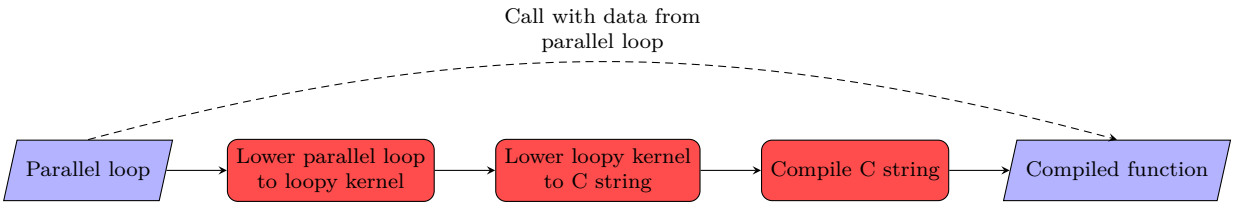


Figure 2.1: Simplified code generation pathway for a PyOP2 parallel loop.

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

(a) Python code to construct the kernel. Some arguments to `make_kernel` have been omitted for simplicity.

```
void loopy_kernel(double const *x,
                  double *y,
                  int64_t const n)
{
  for (int32_t i = 0; i <= -1 + n; ++i)
    y[i] = 2.0 * x[i];
}
```

(b) The generated C code.

Figure 2.2: An example loopy kernel. The kernel takes two array arguments, `x` and `y`, and sets the values in `y` to twice those in `x`. Both arrays have the same unknown length `n` which is also passed in to the kernel as an argument.

2.1.2 Loops

In order to apply kernels to these data structures, a *parallel loop* (`par_loop`) is constructed and executed. The loop takes as arguments a *local kernel*, *iteration set* and zero or more *arguments* that provide the data structures needed by the local kernel.

An example loop statement is shown in **??**.

2.1.3 Code generation

The code generation pipeline is summarised in fig. 2.1. Having constructed a parallel loop, PyOP2 executes it by first lowering the loop object through a sequence of intermediate representations before compiling and running the generated low-level code.

The first intermediate representation is loopy [17], a polyhedral model inspired Python code generation library.

With loopy, the main entry point is the declaration of a `LoopKernel`. To construct such a kernel, the user needs to specify *domains*, *instructions* and *arguments*. An example loopy kernel is shown in fig. 2.2a, with the generated C string shown in fig. 2.2b.

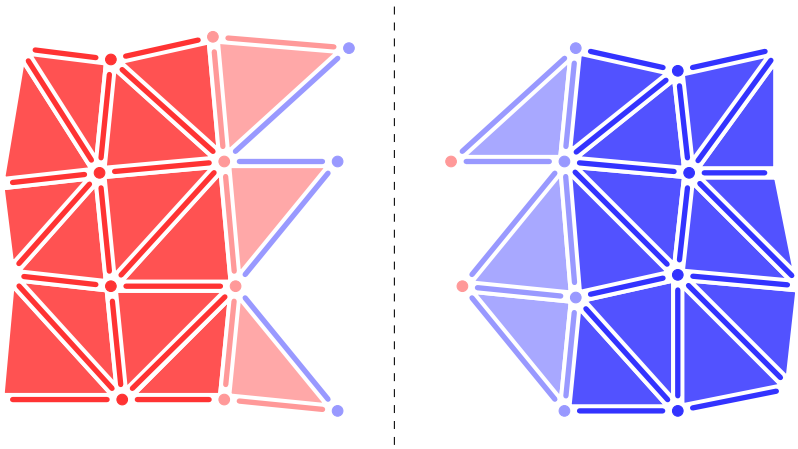


Figure 2.3: PyOP2 entity classes for a mesh distributed between 2 processes. Points belonging to process 1 (left) are shown in red and points belonging to process 2 (right) are shown in blue. *Ghost* points are indicated by points whose colour match the other process, *owned* points are faded and the rest are labelled *core*. We assume that one is only computing cell-wise integrals on the mesh and so the mesh overlap need only contain enough ghost points to ensure that all the cells have a complete closure.

loopy is capable of generating code for multiple backends including CPUs and GPUs and so PyOP2 targeting loopy should, in principle, allow the same PyOP2 code to target different architectures.

Once a `LoopKernel` has been created, loopy also provides a wealth of different code transformations such as loop tiling, vectorisation and loop-invariant code motion.

2.1.4 Parallel execution

Algorithm 2 The PyOP2 parallel loop execution algorithm to interleave computation and communication.

```
Trigger required halo exchanges
FOR EACH item IN iterset.core:
  compute(item)

Await halo exchanges
FOR EACH item IN iterset.owned:
  compute(item)
```

By keeping careful track of the parallel decomposition of sets, PyOP2 is capable of interleaving computation and communication when executing parallel loops. To do so each set is split into 3 parts:

- *Core*: Set elements that do not require any data from other processes during a parallel loop.

- *Owned*: Set elements that belong to the current process but do require data from other processes.

- *Ghost*: Set elements present on a process that belong to another process.

The number of *ghost* points is known to the mesh already, as it has knowledge of its overlap. To determine the *core* and *owned* partitions one loops over the cells of a mesh and inspecting all the points in the closure of the cell. If any of the points in the closure are *ghost*, then all other points in the closure are marked as *owned*. Any remaining points without a label are labelled *core*. An example partitioning of a distributed mesh into *core*, *owned* and *ghost* points is shown in fig. 2.3.

From this partitioning it is possible to interleave computation and communication. Computations over points marked *core* are not influenced by any ghost data and so can proceed before ghost data has been communicated. *Owned* points need to have up-to-date ghost data and so must wait for all communication to be completed before beginning. This is shown in algorithm 2.

2.1.5 Limitations

As mentioned in **??**, PyOP2 is negatively affected by a number of design choices that limit its suitability:

Poor composability Associating mesh data with *node sets* discards topological information and places a burden on the library user to keep track of the relations between the mesh topology and the nodes.

Inflexible interface Not all mesh operations are expressible as a single kernel executed within a single loop over entities. Algorithms for physics-based preconditioners such as hybridisation [12] and additive Schwartz methods [11] involve multiple kernels and nested loops and so implementing them required sui-generis additions to PyOP2 that are difficult to extend. It would be preferable to have an abstraction for mesh computations that was sufficiently flexible for these algorithms to be expressible.

`pyop3` aims to resolve both of these limitations with PyOP2 by rethinking the core abstractions. The design of a number of libraries were used as inspiration for approaches to solving these problems. These will be reviewed below.

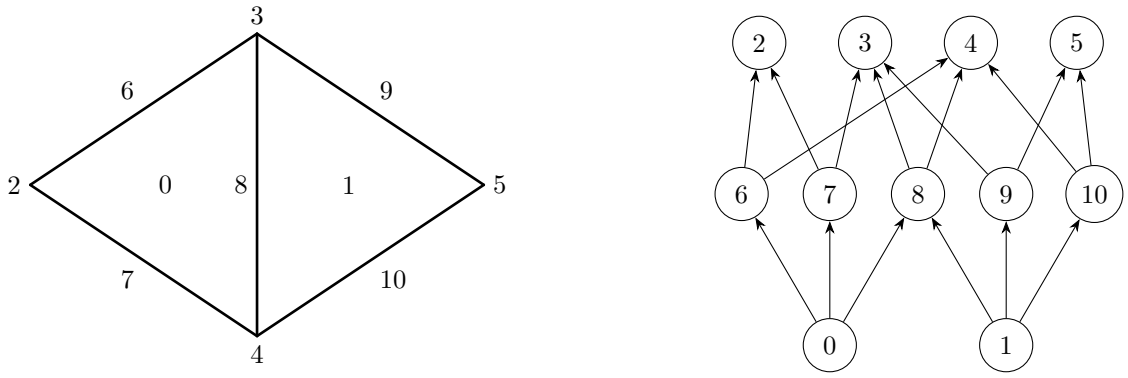


Figure 2.4: TODO, mention Hasse diagram

2.2 Relating unknowns to mesh topology: DMPlex

To tackle the problem of composability - sets discarding information about mesh topology - we look to DMPlex, an abstraction for unstructured meshes.

With DMPlex, an unstructured mesh is represented as a directed acyclic graph (DAG) (fig. 2.4). All topological entities are referred to as *points*, with each entity type belonging to a particular *strata* of the overall DAG. By treating all entities as points, DMPlex is capable of expressing unstructured meshes of any dimension.

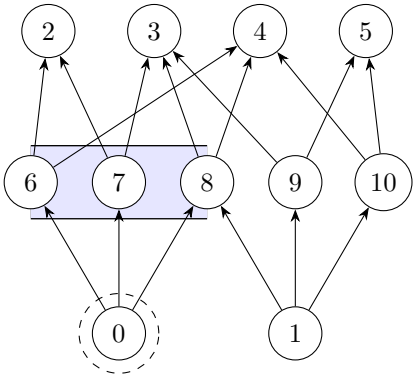
2.2.1 Representing data layouts

Algorithm 3 The tabulation algorithm that determines the right offsets from a `Section`. This code is executed during `PetscSectionSetUp()`.

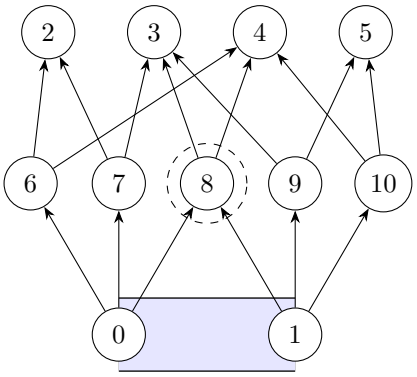
```
counter = 0
offset = 0
FOR EACH point IN chart
  renumbered_point = renumber(point)
  offsets[counter] = offset
  counter += 1
  offset += PetscSectionGetDof(renumbered_point)
```

In order to associate data with these mesh points, a user typical constructs a `Section` object. These are simple CSR-like data structures that associate mesh points with offsets into an array.

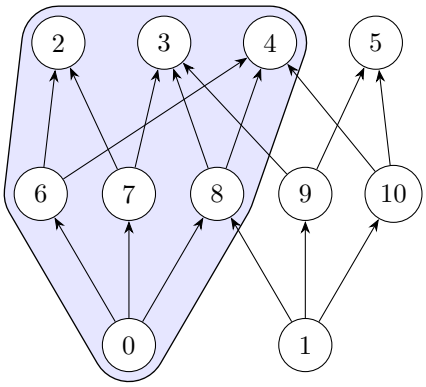
`Sections` are constructed by assigning a number of DoFs with each mesh point (fig. 2.6). Once the number of DoFs has been specified, calling `PetscSectionSetUp()` traverses the input points and accumulates the offset for each point. As a simple example, given the DoF count



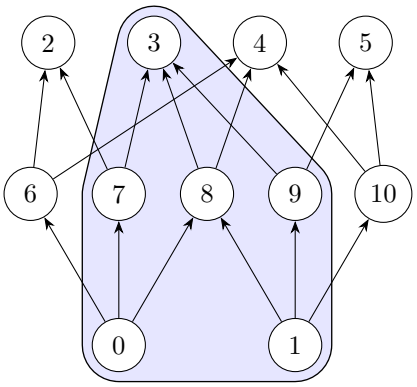
(a) $\mathbf{cone}(0) = \{6, 7, 8\}$



(b) $\mathbf{supp}(8) = \{0, 1\}$



(c) $\mathbf{cl}(0) = \{0, 6, 7, 8, 2, 3, 4\}$



(d) $\mathbf{st}(3) = \{3, 7, 8, 9, 0, 1\}$

Figure 2.5: The possible DMPlex covering queries applied to the Hasse diagram from Figure 2.4.

```
// set cell DoFs
DMPlexGetDepthStratum(dmplex, 2, &start, &end);
for (int cell=start; cell<end; cell++)
  PetscSectionSetDof(section, cell, 2);

// set edge DoFs
DMPlexGetDepthStratum(dmplex, 1, &start, &end);
for (int edge=start; edge<end; edge++)
  PetscSectionSetDof(section, edge, 4);

# set vertex DoFs
DMPlexGetDepthStratum(dmplex, 0, &start, &end);
for (int vertex=start; vertex<end; vertex++)
  PetscSectionSetDof(section, vertex, 2);

PetscSectionSetUp(section);
```

Figure 2.6: C code constructing an appropriate `Section` for a $[P_3]^2$ finite element (section 1.1.1). Some boilerplate code is omitted.

`[1, 0, 3, 2, 0, 1]` the `Section` would tabulate the following offsets: `[0, 1, 1, 4, 6, 6]` (algorithm 3).

`Sections` are also able to express the sorts of renumbering locality optimisations described in section 1.2.2. One simply provides the `Section` with a *permutation* that is accounted for during set up (algorithm 3).

Whilst a powerful tool for describing data layouts, `Sections` have a number of limitations:

- **They are fully ragged**
  `Sections` do not distinguish between different types of topological entity and so important structure cannot be represented. They only store DoF information per point in a completely unstructured way and are incapable of knowing, say, that every cell in the mesh stores exactly one DoF. This can prohibit the compiler from making certain optimisations (e.g. loop unrolling) that it would have been able to do were it to know of a constant loop extent. Additionally, this variable size increases memory pressure as redundant arrays of constant sizes need to be streamed through memory.

- **Shape information is lost**
  Though `Sections` allow one to directly associate DoFs with mesh entities of different dimension, they still lose information about the structure of the function space. For instance, to a `Section`, a point with a single vector-valued node is indistinguishable from a point with multiple nodes.

These limitations prevent `pyop3` from directly using `Sections` to describe its data layouts, but it takes a similar approach:

- All topological entities in the mesh should be equivalent,
- Interleaved points require one to tabulate an array of offsets (algorithm 3).

2.2.2 Parallel

Point SF, DoF SF (diagrams)

* see bullet points...

2.3 A language for structured data: numpy

2.3.1 N-dimensional arrays

The key abstraction introduced by numpy is the *N-dimensional array*, or `ndarray`.

Index operation	Example	Return value	Array return type
Single element indexing	`array[1]`	`"B"`	N/A
Slicing	`array[1:6:2]`	`["B", "D", "F"]`	View
Integer array indexing	`array[[0, 3, 4]]`	`["A", "D", "E"]`	Copy

Table 2.1: Common indexing operations for numpy arrays. The examples shown apply the index to the string array `["A", "B", "C", "D", "E", "F"]` (called `array` above). The array return type for single element indexing is marked as "N/A" because a string is returned instead of an array.

2.3.2 Indexing arrays

Broadcasting operations, that handle the entire array monolithically, are not adequate for many programs. numpy, therefore, allows for array *indexing*, where portions of the full array are extracted to yield a new array.

Some of the ways that a numpy array may be indexed are shown in table 2.1:

- **Single element indexing**

  Single element indexing takes an integer and simply returns the value stored at that point.

- **Slicing**

  Slices are a standard Python concept for describing ranges of indices and have the syntax `[start:stop:step]`. Omitting `start`, `stop` or `step` will default to 0, the end of the array, and 1 respectively. In the example shown in table 2.1 the slice `[1:6:2]` corresponds to asking for "the values in the array from index 1 (inclusive) to index 6 (exclusive), striding by 2".

- **Integer array indexing**

  Integer array indexing returns a new array containing values stored at the requested indices, for the example in table 2.1 this simply being 0, 3 and 4.

Although the examples provided are all for a 1-dimensional array, it is completely permissible to index N-dimensional arrays with a collection of these indexing operations, one per axis of the array. This collection of indices is termed a *multi-index*.

numpy draws a distinction between "basic" indexing, single element indexing and slicing, and "advanced" indexing like using integer arrays. For the former, the array returned from the indexing operation is a *view*, whereas for the latter a *copy* is returned. Alongside the obvious memory advantages, views are also preferable to copies because they are *composable*. One can take views of views repeatedly without triggering a copy, allowing for changes to the indexed array to be propagated back to the original. In `pyop3`, as well as generalising the indexing operations above

to axis trees, we overcome this shortcoming of numpy's advanced indexing such that views are always used regardless of the indexing method used.

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.

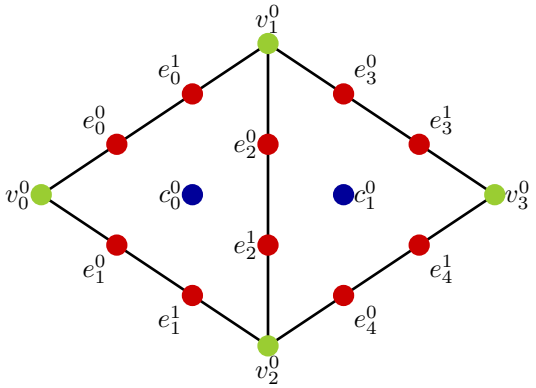


Figure 3.1: TODO

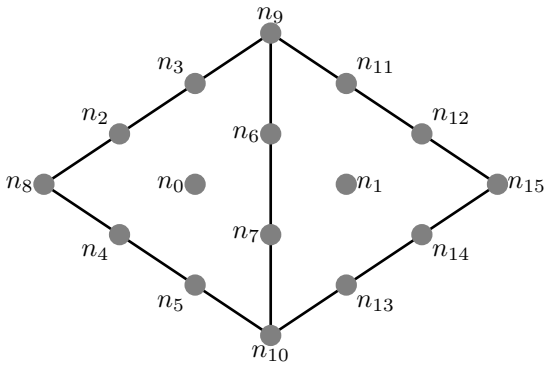


Figure 3.2: TODO

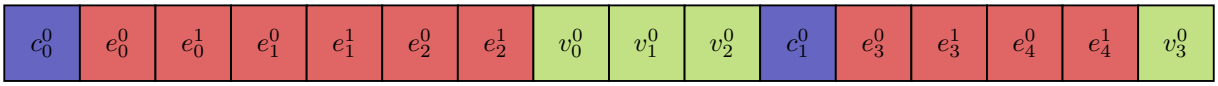


Figure 3.3: TODO

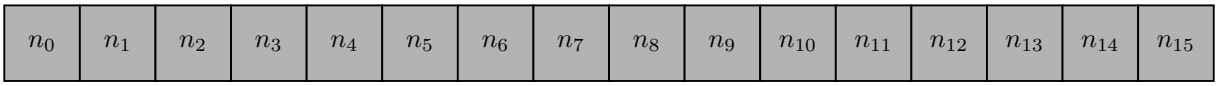


Figure 3.4: TODO

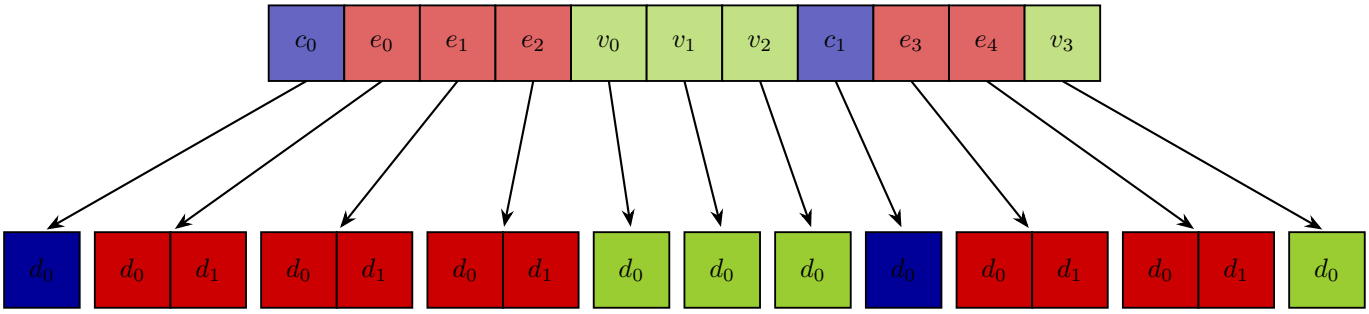
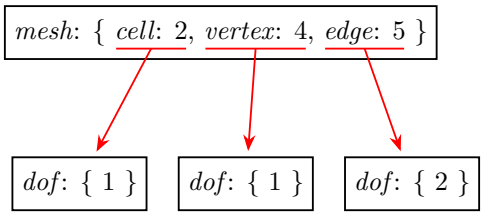


Figure 3.5: TODO



(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*.

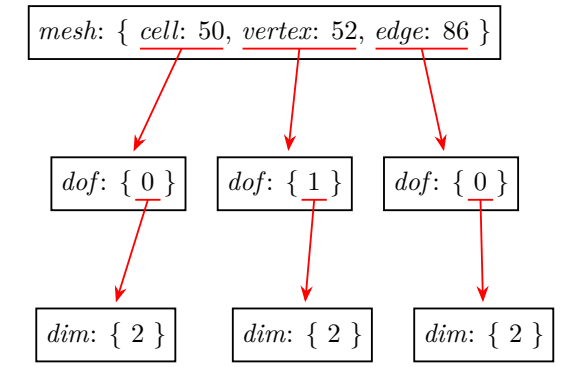
3.1 Axis trees

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.

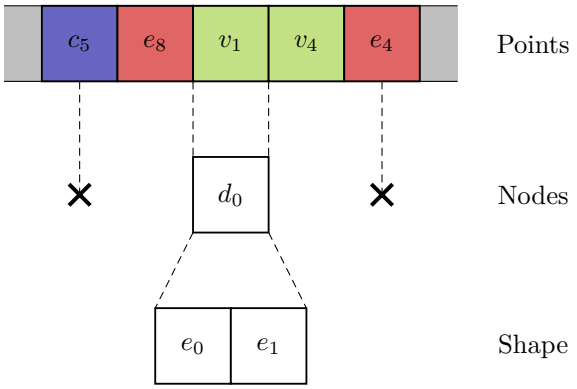
Typically, this structural information is discarded. `pyop3`, however, is capable of capturing this information through using the concept of an *axis tree*.

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



(a) TODO



(b) TODO

Figure 3.7: TODO

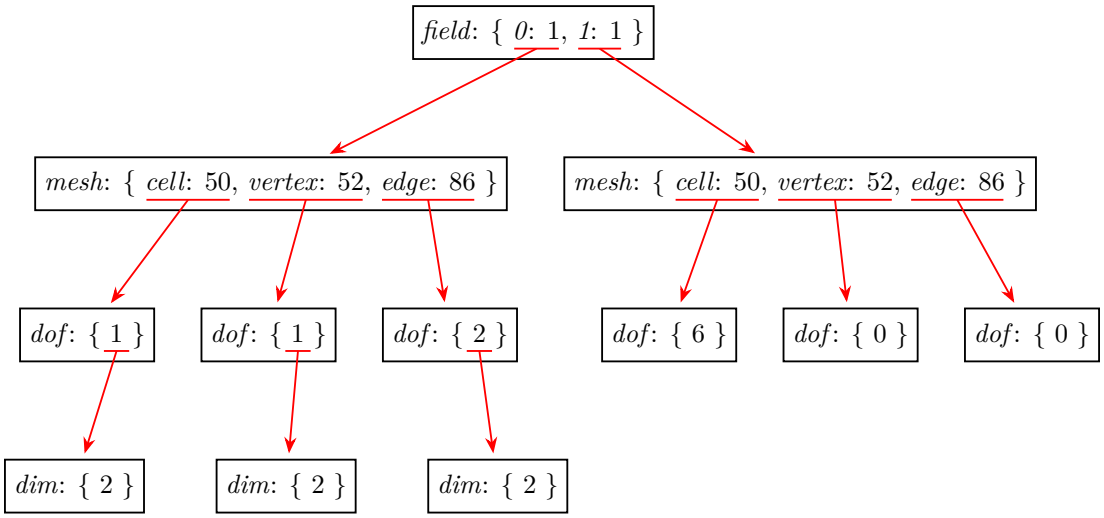
labels and the values are the component labels. `None` is permissible for the `"dof"` axis because there is only a single component, and hence no ambiguity. Axis component labels must be unique within an axis, and axis labels must be unique within each possible path leading from root to leaf.

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

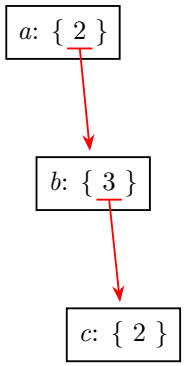
In exactly the same way as for vector-/tensor-valued function spaces, the order in which the axes are declared is flexible...

3.2 Renumbering for data locality

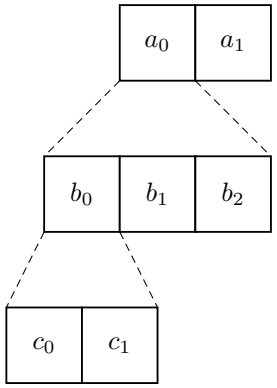
For memory-bound codes, performance is synonymous with data locality. In the case of stencil codes like finite element assembly, one should aim to arrange the data such that the data required for a single stencil calculation is contiguous in memory and can be read from memory into cache with only a single 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 [20]. 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.

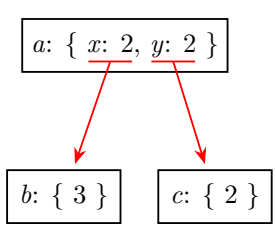
There are additional considerations in parallel that are discussed later in chapter 6.

Linear axis trees

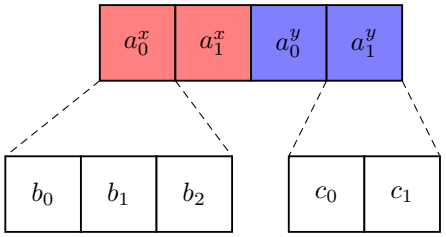
Algorithm 4 Algorithm for computing the layout functions of a linear (single component) axis tree such as that shown in fig. 3.9a. The function is initially invoked by passing the root axis of the tree.

```
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 4. 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.

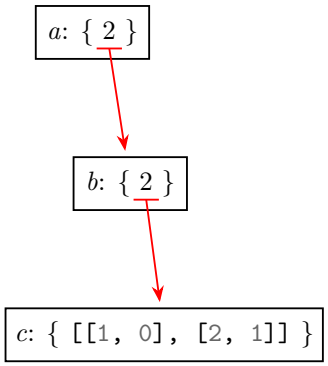
The modifications from algorithm 4 required to determine the right layout function for a multi-component axis tree are relatively straightforward. The modified algorithm is shown in algorithm 5 with the core changes labelled and highlighted in red. These core changes are:

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

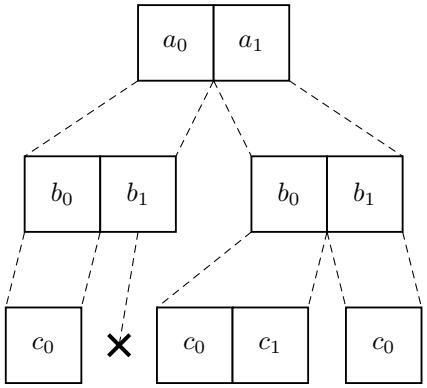
B The layout functions are now stored per `(axis, component)` pair, and an additional offset, named `start`, is added.

Algorithm 5 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                                                  } B
  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) TODO



(b) TODO

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

(c) TODO

Figure 3.11: TODO

Ragged axis trees

Algorithm 6 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

In array codes it is very rarely the case that the entire array is operated on as a single unit. Instead, what more commonly happens is the array is restricted to a smaller piece (e.g. a single value) so that it may be read or modified. This operation is almost universally referred to as *array indexing*. In this chapter we introduce the necessary abstractions and algorithms required to index axis trees in `pyop3`.

4.1 Index trees

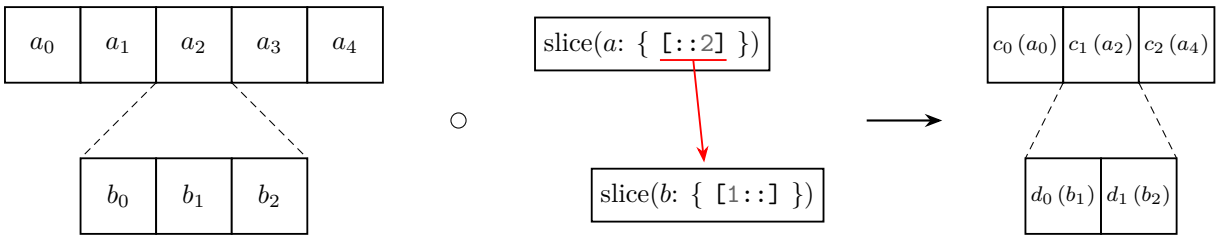
In `pyop3`, indexing is accomplished via the use of *index trees*. Analogously to axis trees, index trees consist of multiple *index* objects, each of which has one or more *index components*. When an axis tree is indexed, it is transformed via composition with an index tree:

$$\text{Axis tree} \quad \circ \quad \text{Index tree} \quad \rightarrow \quad \text{Indexed axis tree}$$

This composition operation yields a new *indexed* axis tree that understand how its entries map back to the original axis tree. In other words, it is a *view*.

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 with shape `(5, 3)` and axes labelled a and b. The index tree is also linear and consists of two *slice* objects over the axes a and b respectively with the former taking every other entry in a (`[::2]`) and the latter taking all but the first entry in b (`[1::]`). The indexed axis tree resulting from the composition of these trees is shown to the right: only the selected indices from axes a and b are present and the axes have been relabelled (arbitrarily) c and d.

In addition to having a new shape and new labels, the indexed axis tree also carries the information necessary to map back from the indexed shape (the *source*) to the original axis tree



(a) Diagram of the data layout transformation. The original axis tree (left) is composed with an index tree (middle) to produce a new, indexed, axis tree (right). The bracketed values in the final tree show the original array entries that they map to.

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

(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 slices `[::2]` and `[1::]` on axes a and b respectively. The resulting axis tree has shape `(3, 2)` and different labels: c and d.

(the *target*). This is done by associating two attributes with the indexed axis tree:

Target paths The target path is a map from the source tree to the axis labels of the target tree. It allows `pyop3` to know where the source axes came from so it can select the right layout functions (**??**) from the target tree. In fig. 4.1b, the target path shows that source axes c and d map back to a and b in the original array.

Target expressions For an indexed axis tree, the target expressions relate the source indices to the target indices as a distinct symbolic expression per target axis. In fig. 4.1b the two target expressions are shown to be $i_a := 2i_c$ and $i_b := i_d + 1$, telling us that c_m maps to a_{2m} and that d_n maps to b_{n+1} respectively.

With these two pieces of information, it is now possible to implement view-like semantics for indexed axis trees. One simply has to:

1. Use the target path to select the appropriate layout function from the target axis tree.

2. Modify the layout function by substituting the target indices in the function with the source indices as described by the target expressions attribute of the indexed tree.

3. Evaluate the offset using the new layout function (that is now a function of the source indices rather than the target indices).

Applying these steps using the indexing operation of fig. 4.1, we have:

1. The target path is $\{a, b\}$, so the selected layout function is $\text{offset}(i_a, i_b) = 3i_a + i_b$.
2. Substituting the target expressions in fig. 4.1b, we get the new layout function $\text{offset}(i_c, i_d) = 6i_c + i_d + 1$.
3. The original axis tree may now be addressed using the source indices i_c and i_d.

4.1.1 Indexed axis tree construction

Algorithm 7 Algorithm that constructs the necessary components to build an indexed axis tree by visiting the nodes of an index tree.

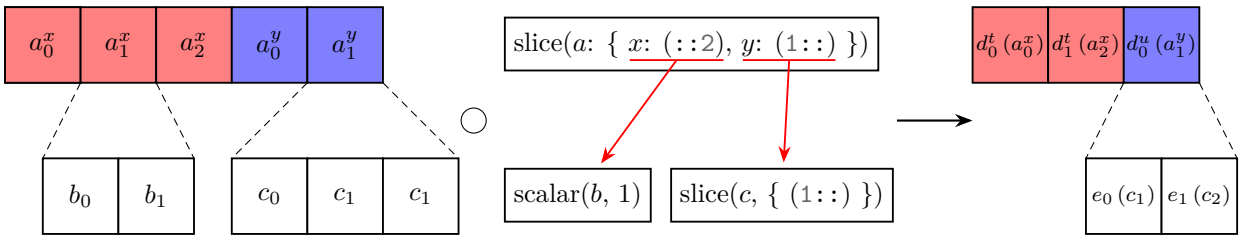
```
1  def index_axes(index, old_axis_tree):
2    # process the current index
3    axis_tree, target_paths, target_exprs = index_handler(index, old_axis_tree)
4
5    for component in index.components:
6      if has_subindex(index, component):
7        # recursively visit child indices
8        subindex = get_subindex(index, component)
9        subaxis_tree, subtarget_paths, subtarget_exprs = index_axes(subindex, old_axis_tree)
10
11       axis_tree.add_subtree(subaxis_tree)
12       target_paths |= subtarget_paths
13       target_exprs |= subtarget_exprs
14
15   return axis_tree, target_paths, target_exprs
```

The construction of an indexed axis tree from an axis tree and index tree is accomplished via a traversal of the index tree (algorithm 7). Each index of the tree is processed by the function `index_handler` (line XXX) to give an axis tree and set of targets paths and expressions specific to that index. These axis trees are then glued together (line YYY) to give the axis tree for the final object while the target paths and target expressions are similarly combined (lines AAA and BBB). Finally, once collected, the three returned variables may be used to construct a finished indexed axis tree.

The original axis tree (`old_axis_tree`) plays only a small role in the indexing algorithm. It is used to determine the sizes of sliced axes and consistency checks.

4.1.2 Index composition

4.2 Outer loops

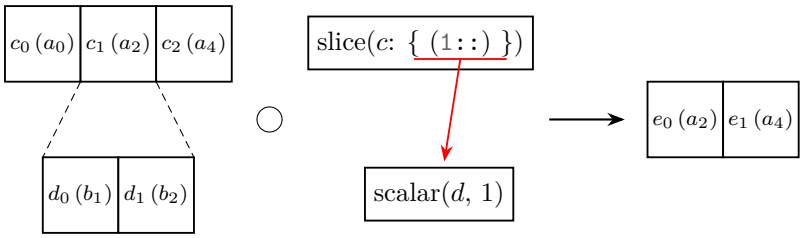


(a) Diagram of the data layout transformation. ...

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.



(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.

```
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.

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).

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).

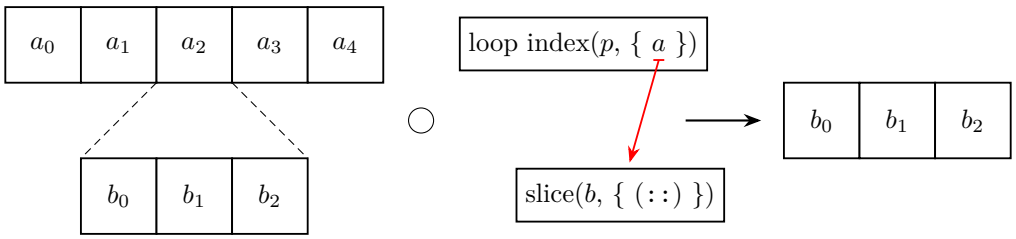
The challenge here is how to represent the indexing operation `dat[i]`...

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*.

4.3 Maps

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

How to build a map.



(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`.

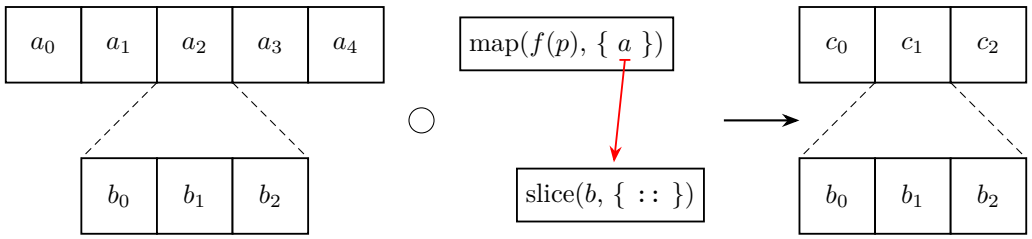
4.3.1 Ragged maps

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

4.3.2 Map composition

4.4 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.

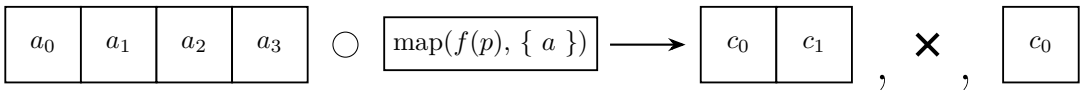


(a) The data layout transformation from the unindexed axis tree (left) to the indexed one (right). Note how the a axis has been replaced by the map axis c.

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)`.

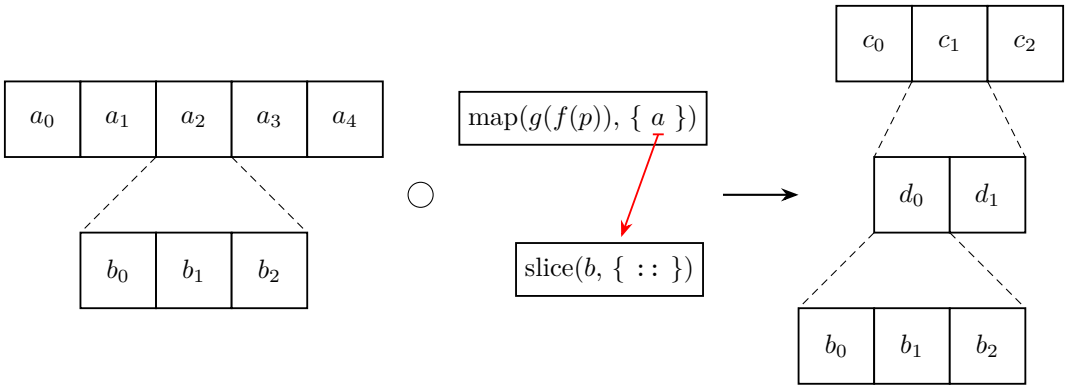


(a) TODO

Source path	Target path	Target expressions

(b) TODO

Figure 4.7: TODO

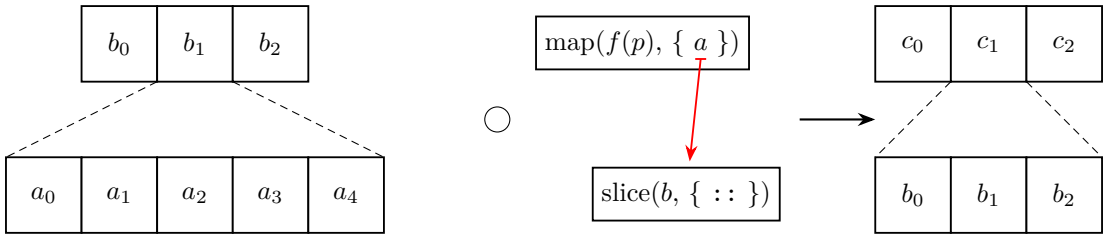


(a) TODO

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
$\{c,\ b\}$	$\{a,\ b\}$	$\{i_a{:}\ f(i_a^p, i_c),\ i_b{:}\ i_b\}$

(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 8.2).

5.1.3 Matrices (`Mats`)

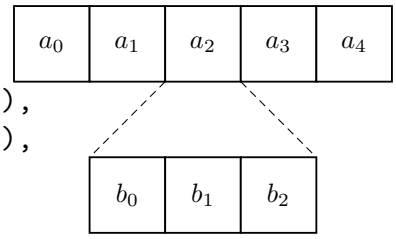
`Mats` require 2 axis trees: one for the rows of the matrix and one for the columns. They rely on PETSc `Mat` objects for the underlying data storage. To improve performance one should preallocate the matrix by constructing a `Sparsity` object and doing a simulated run of all the loop expressions so 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(...),
)
```



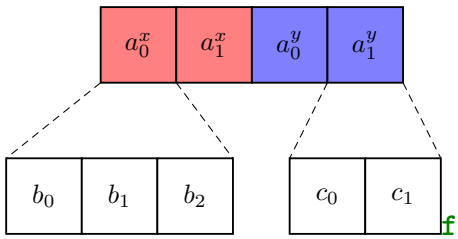

```
for ia in range(5):
  for ib in range(3):
    do_something(...)
```

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.

Context-sensitive loops

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

(a) TODO



(b) 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

5.2.2 Kernels

5.3 Code generation

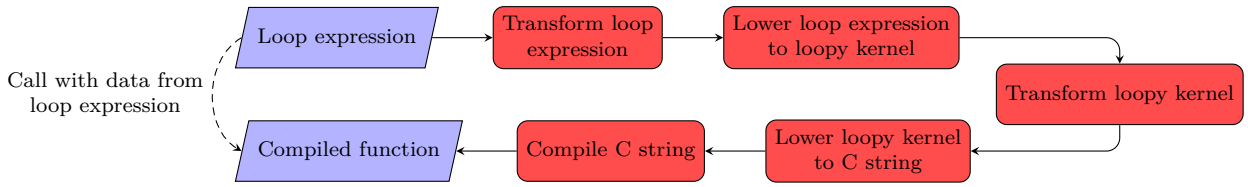


Figure 5.3: The code generation pipeline for the compilation of a loop expression into a callable function. The input ("Loop expression") and output ("Compiled function") are shown in blue whilst the intermediate processes are red. The dashed line from input expression to output function is included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

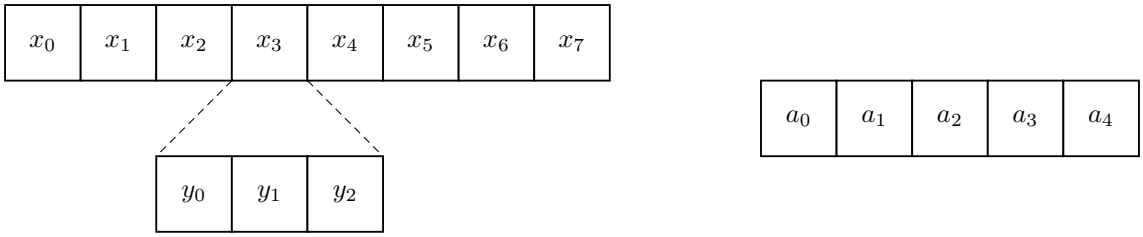
```
loop(
  p := a.index(),
  kernel(dat0[map0(p), :], dat1[p]),
)
```

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

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.

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.

5.3.1 Loop expression transformations

5.3.2 Lowering loop expressions to loopy kernels

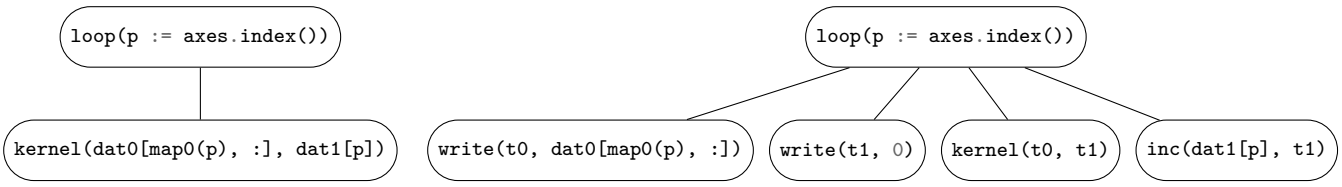


Figure 5.4: The expression tree transformation expanding implicit pack and unpack instructions for the example loop expression (section 5.3). The input loop expression is shown on the left and the output, expanded, loop expression is shown on the right. `kernel` has argument intents of `READ` and `INC`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

```
--------------------------------------------------------------------------------
KERNEL: pyop3_kernel
--------------------------------------------------------------------------------
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
  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 pyop3_kernel(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;
    kernel(&(t_0[0]), &(t_1[0]));
    array_2[i_0] = array_2[i_0] + t_1[0];
  }
}
```

Figure 5.6: TODO

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 [28], or enabling the generated code to run on GPUs [19]. These things are considered to be future work (see section 8.2).

Chapter 6

Parallelism

Just like Firedrake (e.g. [6]) 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 [18] and would increase the complexity of the code.

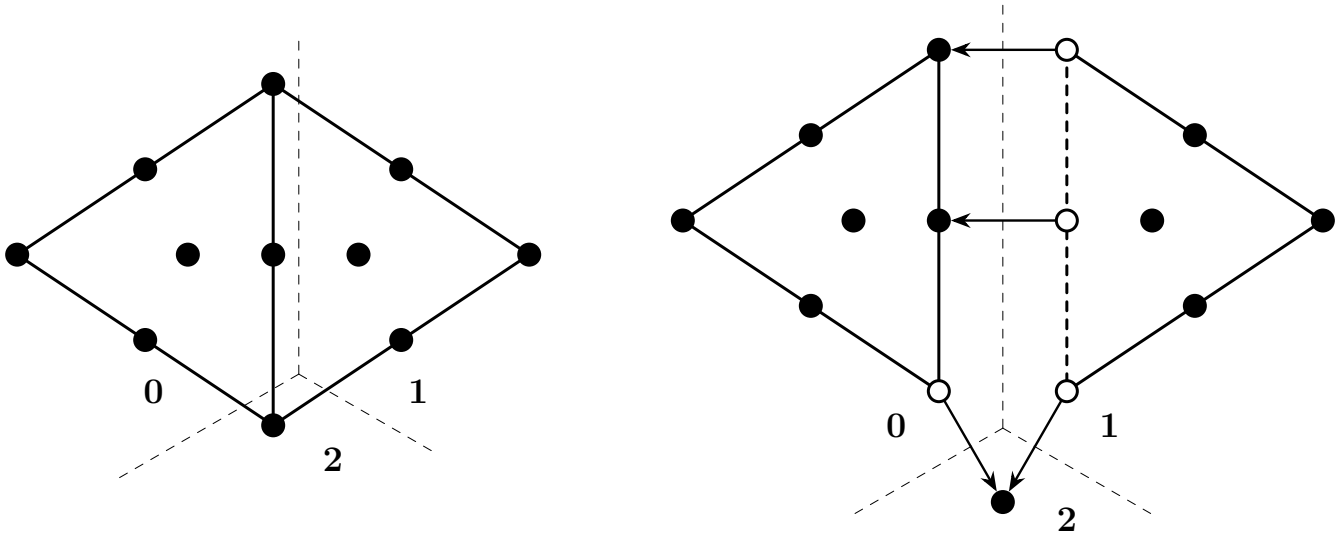
6.1 Message passing with star forests

Almost all message passing in `pyop3` is handled by star forests, specifically by PETSc star forests (`PetscSF`) [29].

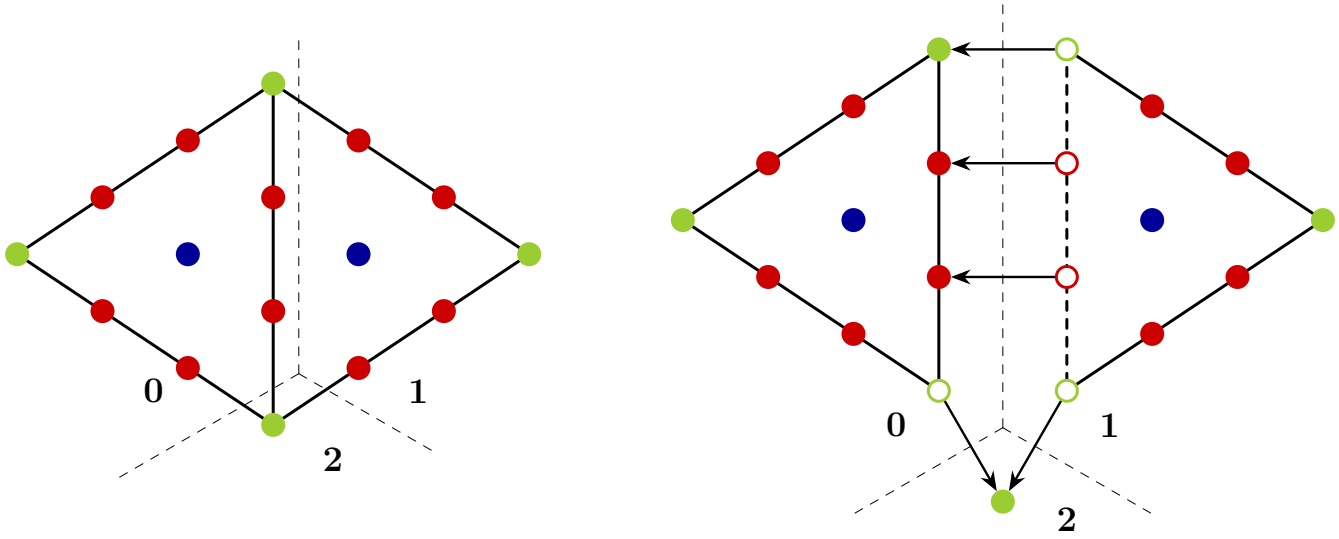
A star forest is defined as a collection of stars, where a star is defined as a tree with a single root and potentially many leaves. Star forests are effective for describing point-to-point MPI operations because they naturally encode the source and destination nodes as roots and leaves of the stars.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

- **The iteration is conflated with the data layout**
  ... For instance, it is not possible to perform parallel loops using larger stencils because the *core-owned* split is different. Embedding the iteration set into the data layout means that the underlying sets of the data structures are invalid.

- **The parallel decomposition is assumed to be the same for all data structures**
  It is not always the case that all data structures will be defined on the same mesh (e.g. mesh transfer operations). In such a case the algorithms used to determine *core* and *owned* points no longer work because the algorithm does not take into account the parallel decomposition of the other mesh. *Core* points on one mesh may be *ghost* in another, and hence the *core* points ought to be labelled *owned*.

These limitations suggest a new approach: the partitioning of data and the partitioning of the iteration set should be *distinct processes*. In `pyop3` we adopt the following new terminology:

- Data structures (axis trees) are partitioned into *owned* and *ghost* points with all *owned* points being stored contiguously before the *ghost* points.

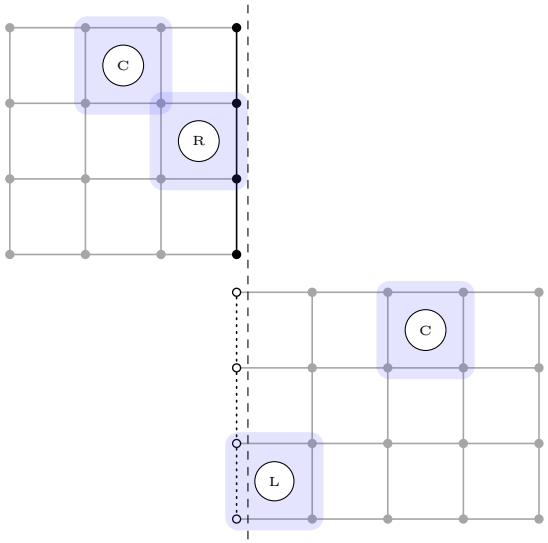
- Loop expression iteration sets are split into *core* and *non-core* sets. This partitioning is established by running over the iteration set and checking the access pattern for all arguments. If any of the arguments require halo data then the iteration point is classified as *non-core*. All remaining points are then classified *core*. As ghost points are not iterated over they are not included in the subsets.

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:

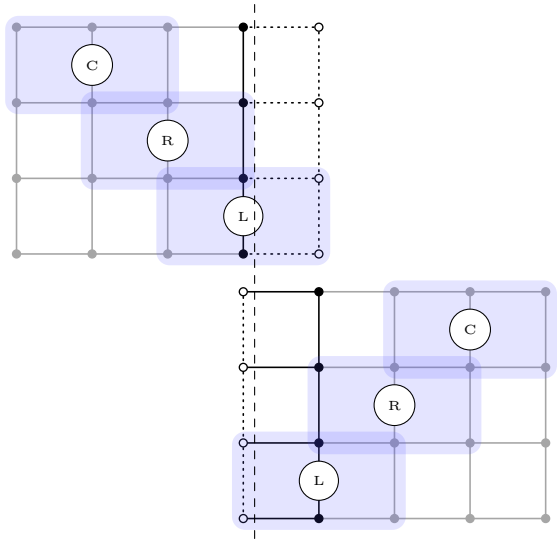
1. Initiate non-blocking halo exchanges.

2. Compute results for data that does not rely on the completion of these halo exchanges.

3. Block until the halo exchanges are complete.



(a) TODO



(b) TODO

Figure 6.2: TODO

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. [7] 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 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,

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

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

Chapter 7

Firedrake integration

7.1 Packing

7.1.1 Tensor product cells

7.1.2 Hexahedral elements

Chapter 8

Summary

8.1 Comparison to related work

8.2 Future work

8.3 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] Satish Balay et al. "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries". In: *Modern Software Tools in Scientific Computing*. Ed. by E. Arge, A. M. Bruaset, and H. P. Langtangen. Birkhäuser Press, 1997, pp. 163–202.

[3] Satish Balay et al. *PETSc Users Manual*. ANL-95/11 - Revision 3.15. Argonne National Laboratory, 2021. URL: https://www.mcs.anl.gov/petsc.

[4] Satish Balay et al. *PETSc Web Page*. 2021. URL: https://www.mcs.anl.gov/petsc.

[5] Gilbert Louis Bernstein et al. "Ebb: A DSL for Physical Simulation on CPUs and GPUs". In: *ACM Transactions on Graphics* 35.2 (May 25, 2016), pp. 1–12. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/2892632. URL: https://dl.acm.org/doi/10.1145/2892632 (visited on 03/28/2022).

[6] 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).

[7] 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.

[8] Susanne C. Brenner and Larkin Ridgway Scott. *The Mathematical Theory of Finite Element Methods*. Third edition. Texts in Applied Mathematics 15. New York, NY: Springer, 2008. 397 pp. ISBN: 978-0-387-75934-0 978-0-387-75933-3.

[9] Philippe G. Ciarlet. *The Finite Element Method for Elliptic Problems*. Society for Industrial and Applied Mathematics, 2002. DOI: 10.1137/1.9780898719208. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719208. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898719208.

[10] Zachary DeVito et al. "Liszt: A Domain Specific Language for Building Portable Mesh-Based PDE Solvers". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. Seattle, Washington: ACM Press, 2011, p. 1. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063396. URL: http://dl.acm.org/citation.cfm?doid=2063384.2063396 (visited on 06/03/2021).

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

[12] Thomas H. Gibson et al. "Slate: Extending Firedrake's Domain-Specific Abstraction to Hybridized Solvers for Geoscience and Beyond". In: *Geoscientific Model Development* 13.2 (Feb. 25, 2020), pp. 735–761. ISSN: 1991-9603. DOI: 10.5194/gmd-13-735-2020. URL: https://gmd.copernicus.org/articles/13/735/2020/ (visited on 11/26/2020).

[13] Johnny Guzmán and L. Ridgway Scott. "The Scott-Vogelius Finite Elements Revisited". In: *Mathematics of Computation* 88.316 (Apr. 16, 2018), pp. 515–529. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/mcom/3346. URL: https://www.ams.org/mcom/2019-88-316/S0025-5718-2018-03346-4/ (visited on 06/17/2024).

[14] David A. Ham et al. *Firedrake User Manual*. First edition. manual. Imperial College London and University of Oxford and Baylor University and University of Washington, May 2023. DOI: 10.25561/104839.

[15] Miklós Homolya, Robert C. Kirby, and David A. Ham. "Exposing and Exploiting Structure: Optimal Code Generation for High-Order Finite Element Methods". Nov. 7, 2017. arXiv: 1711.02473 [cs]. URL: http://arxiv.org/abs/1711.02473 (visited on 08/27/2021).

[16] Fredrik Kjolstad et al. "Simit: A Language for Physical Simulation". In: *ACM Transactions on Graphics* 35.2 (May 25, 2016), pp. 1–21. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/2866569. URL: https://dl.acm.org/doi/10.1145/2866569 (visited on 03/29/2022).

[17] Andreas Klöckner. "Loo.Py: Transformation-Based Code Generation for GPUs and CPUs". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY'14* (2014), pp. 82–87. DOI: 10.1145/2627373.

2627387. arXiv: 1405.7470. URL: http://arxiv.org/abs/1405.7470 (visited on 10/14/2020).

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

[19] 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.

[20] 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).

[21] 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).

[22] 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).

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

[24] Florian Rathgeber et al. "PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. Salt Lake City, UT: IEEE, Nov. 2012, pp. 1116–1123. ISBN: 978-0-7695-4956-9 978-1-4673-6218-4. DOI: 10.1109/SC.Companion.2012.134. URL: http://ieeexplore.ieee.org/document/6495916/.

[25] L. R. Scott and M. Vogelius. "Norm Estimates for a Maximal Right Inverse of the Divergence Operator in Spaces of Piecewise Polynomials". In: *ESAIM: Mathematical Modelling and Numerical Analysis* 19.1 (1985), pp. 111–143. ISSN: 0764-583X, 1290-3841. DOI: 10.1051/

m2an/1985190101111. URL: http://www.esaim-m2an.org/10.1051/m2an/1985190101111 (visited on 06/17/2024).

[26] Matthew W. Scroggs et al. *DefElement: An Encyclopedia of Finite Element Definitions*. 2024. URL: https://defelement.com.

[27] 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.

[28] 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).

[29] 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).