

AN EXECUTION ABSTRACTION FOR
COMPACT COMPUTATIONAL KERNELS ON
UNSTRUCTURED MESHES

CONNOR J. WARD

Department of Mathematics
Imperial College London

September 2024

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Abstract

The numerical solution of partial differential equations (PDEs) is critical to a wide range of scientific and engineering disciplines. Developing software to solve such systems is difficult, requiring expertise in many disparate areas and months or years of developer time. To ease this burden, a number of simulation frameworks make use of high-level domain-specific languages (DSLs) that mirror the underlying mathematics. The use of DSLs introduces a separation of concerns between the top-level mathematics and the low-level implementation and lowers the barrier to entry to high fidelity simulation codes, leading to more and better science.

The expressive power, and hence to some extent the value, of a DSL as a tool for computation hinges upon the existence of the right abstractions. If the abstractions that underly the DSL are a poor fit to the top-level mathematics then many problems that should naturally be expressible are not, either barring the user from implementing the desired method, or requiring sui-generis solutions that compose poorly with the rest of the software stack.

This thesis focusses on one particular type of abstraction that is relevant to the solution of PDEs: the application of compact computational kernels (a.k.a. stencils) to unstructured meshes. Classically, finding abstractions for this operation is difficult because data stored on unstructured meshes have complex layouts, with additional considerations such as parallelism making things even more challenging. This work presents a suite of novel abstractions that encapsulate the processes of storing and accessing complex data layouts on unstructured meshes, as well as an implementation of said abstractions in the new Python library `pyop3`. These abstractions better capture the semantics of what it means to apply a kernel over a mesh, leading to a framework where a greater range of algorithms may be expressed, transformed, and optimised.

Declaration

I hereby certify that the work constituting this thesis is my own except when properly acknowledged.

Connor J. Ward

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC). Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose. When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes. Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Acknowledgements

I am very lucky to be able to say that I have honestly enjoyed almost every second (thesis writing notwithstanding!) of my PhD. The fact that I can say this is down entirely to the wonderful people I have had the pleasure of spending time with over the last four years.

My thanks go first to my supervisors, David Ham and Jack Betteridge. Their support and expertise offered throughout the PhD has been invaluable, and without it there is no way I would have achieved the same amount that I have as cheerfully as I have.

I would also like to thank my colleagues in the core Firedrake team, both past and present: Koki, Daiane, Nacime, Reuben, and Sophia, for their helpfulness and companionship. As is often said, it is Firedrake's community that makes it so special and that was certainly my experience. I really appreciate getting the opportunity to work with you all. A special mention is also needed for former core team member Lawrence Mitchell for suggesting that I apply to do this PhD. Without you this would never have happened.

Further thanks are due to the wonderful group of friends that I have shared an office with through my PhD: Thivan, Sara, Jamie, Karina, Henry, and Maria. I cannot believe I was somehow put in the friendliest office in all of Huxley and I count myself very lucky to have gotten to know you all.

Lastly my biggest thanks of all go to my family for their unwavering support throughout this journey. In particular, Rhiannon has been nothing short of amazing. By now she has heard more about my 'trees and triangles' than anyone else and yet she still puts up with me!

For Opa,

Contents

1	Introduction	8
1.1	Contributions	10
1.2	Dissemination	11
1.3	Thesis outline	12
2	Background	13
2.1	The finite element method	13
2.1.1	Deriving a weak formulation	14
2.1.2	Discretising the system of equations	15
2.1.3	Choosing a basis	16
2.1.4	Finite element assembly	18
2.1.5	Finite element data structures	20
2.2	Abstractions for mesh stencils	22
2.2.1	Desirable characteristics for a mesh stencil package	25
2.2.2	Existing mesh stencil packages	28
2.2.3	The missing abstraction	30
3	Existing abstractions for arrays and stencils	32
3.1	NumPy	32
3.1.1	N-dimensional arrays	32
3.1.2	Indexing arrays	33
3.2	PETSc DMPlex	34
3.2.1	Topological queries	35
3.2.2	Representing data layouts	35
3.2.3	Parallel	38
3.3	PyOP2	41

3.3.1	Data structures	41
3.3.2	Parallel loops	41
3.3.3	Code generation	45
3.3.4	Limitations	48
3.4	Outlook	48
4	Axis trees	49
4.1	Data structures	52
4.2	Alternative data layouts	52
4.3	Ragged axis trees	53
4.4	Computing offsets	55
4.4.1	Intermediate algorithm 1: just linear axis trees	55
4.4.2	Intermediate algorithm 2: with multi-component axis trees	58
4.4.3	Complete algorithm: with ragged axis trees	60
4.5	Outlook	63
5	Indexing	65
5.1	Index trees	65
5.1.1	Index composition	70
5.2	Outer loops	72
5.3	Maps	73
5.3.1	Ragged maps	75
5.3.2	Map composition	77
5.4	Indexing with alternative data layouts	77
5.5	Outlook	80
6	pyop3	81
6.1	The execution model	81
6.1.1	Example 1: Constant assignment	82
6.1.2	Example 2: Outer loops and maps	84
6.2	Matrices	88
6.2.1	Indexing	89
6.2.2	Precomputing sparsity patterns	89
6.2.3	Code generation	90
6.3	Optimising index expressions	92
6.4	Outlook	93

7	Parallel	94
7.1	Parallel data layouts	94
7.1.1	Parallel layout functions	95
7.1.2	DoF star forests	97
7.2	Communication optimisations	98
7.2.1	Minimising communication	98
7.2.2	Overlapping communication with computation	99
7.3	Outlook	102
8	Firedrake integration	103
8.1	Firedrake	103
8.1.1	PyOP2's contribution	106
8.2	Representing global data with axis trees	107
8.2.1	Renumbering the mesh	108
8.3	Finite element assembly	113
8.3.1	More integral types	113
8.3.2	Handling orientations	115
8.4	Outlook	118
9	Demonstrator applications	119
9.1	Solving the Helmholtz equation	121
9.2	One form assembly	123
9.3	Vertex-based slope limiter	127
9.3.1	Performance results	130
9.4	Outlook	131
10	Conclusions	132
10.1	Summary of benefits	133
10.2	Future research directions	134

Chapter 1

Introduction

The numerical solution of partial differential equations (PDEs) is ubiquitous across a wide range of applications, from climate modelling to structural mechanics and manufacturing. Naturally, it is therefore critical that scientists and engineers have access to high-productivity, high-performance libraries with which to develop simulations.

For such libraries, performance portability - being able to run simulations on disparate hardware without losing efficiency - is obviously a significant concern for optimising both programmer productivity and energy efficiency. Achieving this portability when solving PDEs is challenging because, to best take advantage of the properties of the underlying hardware, it can be necessary to change high-level aspects of the computation such as the choice of discretisation and/or solver strategy.

For hand-written codes, making these sorts of high-level changes is time-consuming and difficult, and so many libraries implement high-level, mathematical *abstractions* to streamline development. Abstractions hide low-level details of a program behind intuitive interfaces, thus making it easier to write applications, and, in particular, to apply the sorts of transformations necessary for performance portability. We say that abstractions induce a *separation of concerns* between the user, who reasons about and writes their program in terms of the abstractions, and the library developer, who implements them.

This thesis focusses on the development of software abstractions that encapsulate the process of applying compact computational kernels to (unstructured) meshes. Also known as *mesh stencil calculations*, applying compact kernels to a mesh is a widespread technique underpinning many common numerical methods for solving

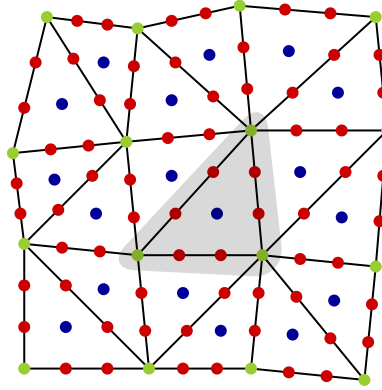


Figure 1.1: An example of a mesh stencil operation where cells are looped over and adjacent DoFs are accessed by the stencil (grey). DoFs stored on the cells, edges, and vertices are represented by blue, red, and green dots respectively.

PDEs, including the finite difference method, finite element method, and finite volume method. To apply a mesh stencil calculation, one loops over a particular collection of mesh entities and, for each entity, computes a contribution to some global data structure using the degrees-of-freedom (DoFs) that lie in the neighbourhood of the current iterate. An example of a mesh stencil operation is shown in fig. 1.1.

Developing abstractions for mesh stencil calculations is challenging because mesh data structures are complex and map poorly to traditional representations of structured data (e.g. N-dimensional arrays). As a result, existing software either make limiting assumptions about the types of permissible data layouts, or sacrifice important structural information about them. Consequently, they all have limited applicability, working *either* for a range of different algorithms *or* a range of complex data structures, but not both.

This thesis aims to bridge this divide by developing a novel suite of abstractions for describing unstructured mesh data. Unlike existing approaches, the abstractions are fully capable of describing complex data layouts without losing essential structural information. This allows for an expressive, composable interface suitable for representing a wider range of algorithms.

1.1 Contributions

The core contributions presented in this thesis are:

- The development of new abstractions for describing unstructured mesh data, called *axis trees* and *index trees*.
- The implementation of these abstractions as part of the new mesh stencil library `pyop3`, which has been integrated into the finite element library Firedrake [31].

Through these abstractions, `pyop3` improves upon its predecessor, PyOP2, in the following ways:

- Unlike PyOP2, `pyop3` data structure can store hierarchical data with variable-sized inner dimension (sometimes referred to as *ragged* data structures) (section 4.3)
- Through *indexing*, `pyop3` is able to describe much more complex data access patterns inside loops that compose arbitrarily without causing any data to be copied (chapter 5).
- By using axis trees and index trees together `pyop3` can describe data layout transformations that generalise the classical array-of-structs/struct-of-arrays optimisation (section 2.2.1) through only modifying the specification of the data layout (sections 4.2 and 5.4). This is a single line change and no other modifications are required.
- By better preserving the complex structure of mesh-based data structures, `pyop3`'s domain-specific language for expressing computations (chapter 6) has far greater composability than PyOP2, allowing it to express algorithms more complex than just those with a single-loop and single-kernel.
- `pyop3` can inspect the access patterns of the data structures in a computation and optimise the indirection expressions to improve data locality (section 6.3).
- `pyop3` further addresses a number of limitations of PyOP2 when run in a distributed-memory parallel environment (chapter 7):
 - The iteration set is no longer conflated with the data layout (section 3.3.2), enabling `pyop3` to work correctly when using different stencil sizes in a

single simulation (section 7.2.2) and hence suitable for applications beyond the finite element method (section 10.2).

- Arrays storing data for multiple fields always partition ghost DoFs at the end of the array, enabling no-copy transformations between global and local views of the data (section 7.1).
- To reduce parallel communication, additional state tracking is employed to minimise ghost DoF exchange (section 7.2.1).

1.2 Dissemination

The work described in this thesis has been presented at the following workshops and conferences:

- *pyop3: A new domain-specific language for automating high-performance mesh-based simulation codes.* Firedrake22 workshop, January 2023, Totnes, UK.
- *pyop3: A new domain-specific language for automating high-performance mesh-based simulation (stencil) codes.* SIAM CSE23 conference, March 2023, Amsterdam, NL.
- *Latest developments in pyop3.* Firedrake23 workshop, September 2023, Missenden, UK.
- *pyop3: An Execution Abstraction for Compact Computational Kernels on Unstructured Meshes.* PDESoft conference, July 2024, Cambridge, UK.
- *pyop3 is coming.* Firedrake24 workshop, September 2024, Oxford, UK.

Seeking publication for this work in a journal is a high-priority future task. The article would focus on how, compared with existing approaches, the new abstractions are better able to represent the complexity inherent in unstructured mesh data, and as such are suitable for a wider range of applications, as well as facilitate a number of non-trivial performance optimisations.

1.3 Thesis outline

The remainder of this thesis proceeds as follows:

- Chapter 2 describes some of the relevant background, motivating the need for mesh stencil computations and identifying shortcomings with existing approaches.
- Chapter 3 introduces in more detail the abstractions and software packages that contributed to the design of the new abstractions.
- Chapters 4 and 5 introduce the new abstractions developed for describing mesh data layouts: the *axis tree* and *index tree*.
- Chapter 6 presents **pyop3**, a new mesh stencil package built upon these abstractions. Additional considerations to enable distributed memory parallelism in **pyop3** are then discussed in chapter 7.
- Chapter 8 describes how **pyop3** has been integrated into the finite element framework Firedrake, and chapter 9 demonstrates this integration with a number of examples.
- Lastly, we conclude in chapter 10 and consider a number of potential directions for future research.

Chapter 2

Background

In this chapter we present the relevant background material necessary for the rest of the thesis. In particular, we motivate the need for mesh stencil software using the finite element method as an example, before making a case for high-level, composable abstractions and their place in mesh stencil computations.

2.1 The finite element method

As a motivating example demonstrating mesh stencils in practice we consider solving the Stokes equations, a linearisation of the Navier-Stokes equations in common use in the fluid dynamics community, using the finite element method (FEM). Our exposition will focus on the aspects of the computation that are relevant for mesh stencils, for a more complete review of FEM we refer the reader to [15] and [54].

The Stokes equations are given as follows: for domain Ω with boundary Γ , find the fluid velocity u and pressure p such that

$$\begin{aligned} -\nu\Delta u + \nabla p &= 0 && \text{in } \Omega, \\ \nabla \cdot u &= 0 && \text{in } \Omega, \\ u &= g && \text{on } \Gamma, \end{aligned} \tag{2.1}$$

where ν is the viscosity and g is some prescribed Dirichlet boundary condition fixing the velocity across the entire boundary. Forcing terms are omitted for simplicity. Since this is a coupled system of two variables (u and p), we refer to it as being *mixed*.

2.1.1 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. (2.1), using v and q as the test functions for the velocity and pressure spaces respectively, and integrating by parts this gives

$$\begin{aligned} \int \nu \nabla u : \nabla v \, d\Omega - \int p \nabla \cdot v \, d\Omega - \int \nu (\nabla u \cdot n) \cdot v \, d\Gamma - \int p n \cdot v \, d\Gamma &= 0 \quad \forall v \\ \int q \nabla \cdot u \, d\Omega &= 0 \quad \forall q \end{aligned} \quad (2.2)$$

Choosing function spaces

From this weak form it is now possible to classify the various function spaces for u , p , v and q . For u , we already know that the space must be vector-valued and constrained to g on the boundary. Equation (2.2) further shows us that u must have at least one weak derivative. We can therefore say that $u \in V$ where, for dimension d ,

$$V = \{ v \in [H^1(\Omega)]^d : v|_{\Gamma} = g \}. \quad (2.3)$$

The function space of the test function v is similar, but, since the values of u at the boundary are already prescribed, it is defined to be zero at those nodes

$$V_0 = \{ v \in [H^1(\Omega)]^d : v|_{\Gamma} = 0 \}. \quad (2.4)$$

Unlike u and v , p and q are scalar-valued and need no derivatives. Since no boundary conditions are applied to the space of p , we can draw p and q from the same space

$$Q = \{ q \in L^2(\Omega) : \int q \, d\Omega = 0 \}, \quad (2.5)$$

where the additional constraint $\int q \, d\Omega = 0$ has been added to ensure well-posedness as, since only the gradient of p appears in eq. (2.1), p is otherwise only determined up to an arbitrary constant.

Given the definition of these spaces and in particular the fact that v is defined to vanish on the boundary, which allows us to drop the surface integrals, we may restate

the problem posed in eq. (2.2) as follows: find $(u, p) \in V \times Q$ such that

$$\begin{aligned} \int \nu \nabla u : \nabla v \, d\Omega - \int p \nabla \cdot v \, d\Omega &= 0 \quad \forall v \in V_0 \\ \int q \nabla \cdot u \, d\Omega &= 0 \quad \forall q \in Q. \end{aligned} \quad (2.6)$$

Boundary conditions

To represent the inhomogeneous boundary conditions on space V , we split the space in two:

$$V = V_0 \oplus V_\Gamma, \quad (2.7)$$

where V_0 has the usual definition from eq. (2.4), and V_Γ is defined as the space spanned by the functions of V that are non-zero on the boundary. With this split we may then rewrite the function u as the sum of the functions $u_0 \in V_0$ and $u_\Gamma \in V_\Gamma$

$$u = u_0 + u_\Gamma. \quad (2.8)$$

u_Γ is prescribed by the boundary condition and hence known. This allows us to move it to the right hand side of eq. (2.6) giving the new formulation: find $(u_0, p) \in V_0 \times Q$ such that

$$\begin{aligned} \int \nu \nabla u_0 : \nabla v \, d\Omega - \int p \nabla \cdot v \, d\Omega &= - \int \nu \nabla u_\Gamma : \nabla v \, d\Omega \quad \forall v \in V_0 \\ \int q \nabla \cdot u \, d\Omega &= 0 \quad \forall q \in Q. \end{aligned} \quad (2.9)$$

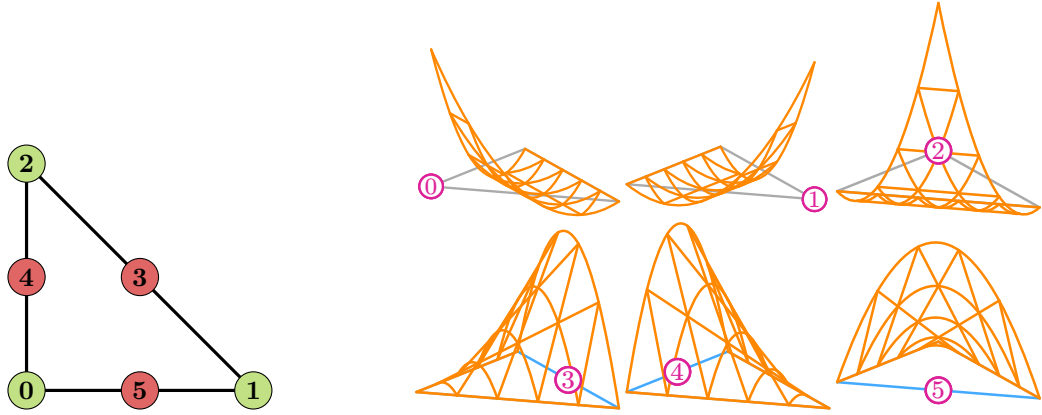
2.1.2 Discretising the system of equations

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

$$V_0 \rightarrow V_{0,h} \subset V_0, \quad Q \rightarrow Q_h \subset Q. \quad (2.10)$$

Each of these discrete spaces is spanned by a set of basis functions, allowing functions to be expressed as a linear combination of basis functions and coefficients. For example, we can write the function $u_{0,h} \in V_{0,h}$ as

$$u_{0,h} = \sum_{i=1}^N \hat{u}_i \psi_i^{V_{0,h}}, \quad (2.11)$$



(a) The dual basis for a P_2 element (b) The basis functions. The images were sourced from consists of point evaluation at the defelement.com [76]. vertices (green) and edges (red).

Figure 2.1: The P_2 (Lagrange, polynomial degree 2) finite element.

where $\psi_i^{V_{0,h}}$ are the basis functions that span $V_{0,h}$ and $\hat{u}_i$ the coefficients.

If we substitute these discrete spaces into eq. (2.9), and discard the basis coefficients for the arbitrary test functions v_h and q_h , we obtain the discrete problem: find the vector $(\hat{u}, \hat{p})$ such that

$$\begin{aligned} \int \nu \hat{u} \nabla \psi^{V_{0,h}} : \nabla \psi^{V_{0,h}} d\Omega - \int \hat{p} \psi^{Q_h} \nabla \cdot \psi^{V_{0,h}} d\Omega &= - \int \nu \nabla u_\Gamma : \nabla \psi^{V_{0,h}} d\Omega & \forall \psi^{V_{0,h}}, \\ \int \psi^{Q_h} \hat{u} \nabla \cdot \psi^{V_{0,h}} d\Omega &= 0 & \forall \psi^{Q_h}. \end{aligned} \quad (2.12)$$

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

$$\left(\begin{array}{c|c} \int \nu \nabla \psi^{V_{0,h}} : \nabla \psi^{V_{0,h}} d\Omega & - \int \psi^{Q_h} \nabla \cdot \psi^{V_{0,h}} d\Omega \\ \hline \int \psi^{Q_h} \nabla \cdot \psi^{V_{0,h}} d\Omega & 0 \end{array} \right) \begin{pmatrix} \hat{u} \\ \hat{p} \end{pmatrix} = \begin{pmatrix} - \int \nu \nabla u_\Gamma : \nabla \psi^{V_{0,h}} d\Omega \\ 0 \end{pmatrix}. \quad (2.13)$$

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

2.1.3 Choosing a basis

In order to numerically evaluate the integrals of eq. (2.13), the basis functions $\psi^{V_{0,h}}$ and ψ^{Q_h} must be known. In the finite element method these are generated from the choice of *finite element*.

First formalised by Ciarlet [19], a finite element is the triple (K, P, N) , where:

- K is a bounded closed set, or *cell*, 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 Lagrange element with polynomial degree 2, written P_2 - is shown in fig. 2.1. For this element K is a triangle, P is the space of order 2 polynomials, and the linear functionals of N are defined as the point evaluations

$$l_i(v) = v(x_i) \quad i = 0, 1, \dots, 5 \quad (2.14)$$

for each of the 6 nodes shown in fig. 2.1a, with x_i being the coordinates of the i -th node.

Given this definition, one can determine a basis, ψ_j , for P by imposing that

$$l_i(\psi_j) = \delta_{ij} \quad i, j = 0, 1, \dots, 5 \quad (2.15)$$

which for the P_2 element yields the basis functions on the reference triangle:

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

that are shown in fig. 2.1b.

Note that the basis functions determined from the finite element have *local support*. That is, they are only non-zero in a small region around their associated entities.

Choosing a basis for the Stokes equations

The choice of finite element has significant implications for the convergence and stability of the method. For the two-dimensional Stokes equations, a common choice of element pair, or *mixed element*, that is inf-sup stable and has properties matching the constraints given in eq. (2.4) and eq. (2.5) is the Scott-Vogelius element [74]. The element consists of a continuous vector-valued degree k Lagrange element for the

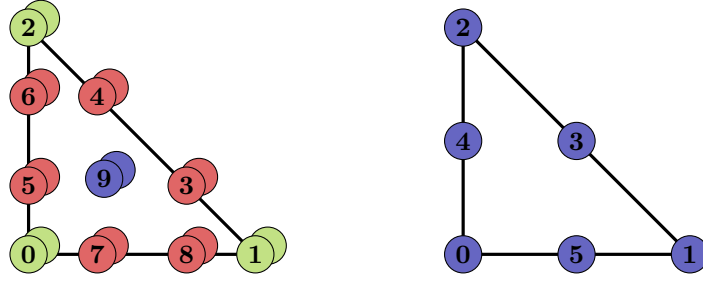


Figure 2.2: The degree 3 Scott-Vogelius element pair $([P_3]^2 \oplus P_2^{\text{disc}})$. The lower-order space (right) is discontinuous and hence all of the DoFs are marked as belonging to the cell (blue).

velocity space, written $[P_k]^d$, and a discontinuous Lagrange element of degree $k - 1$ for the pressure space, written P_{k-1}^{disc} . These¹ are shown in fig. 2.2 for $k = 3$.

2.1.4 Finite element assembly

The finite element basis functions from the Ciarlet definition above are given in *reference space*. For them to be usable for assembling the Stokes system of eq. (2.13) we must first draw a connection between the domain of the problem, Ω , with the domain of the finite element, K . To do this, we partition Ω into a set of disjoint cells, $\mathcal{K}$, called a *mesh*, where

$$\bigcup \mathcal{K} = \Omega. \quad (2.16)$$

Local assembly

Owing to the local support property of finite element basis functions, one can now rewrite the integrals in eq. (2.13) as a sum of cell-wise contributions. Using the top left block of the matrix as an example, the integral can be rewritten as

$$\sum_{\mathcal{K}} \int \nu \nabla \psi_i^{\mathcal{K}} : \nabla \psi_j^{\mathcal{K}} \, d\Omega_{\mathcal{K}}, \quad (2.17)$$

with $\psi^{\mathcal{K}}$ the basis functions over the cell and i and j indicating the different element nodes. Since the basis functions have only local support the resulting matrix is *sparse*.

Subject to the addition of some Jacobian-like terms, called *pullbacks*, to map between the geometry of the physical and reference cells, these cell-wise integrals can

¹In two dimensions the Scott-Vogelius element is technically known to be stable only for $k \geq 4$ [29], but we show $k = 3$ for simplicity.

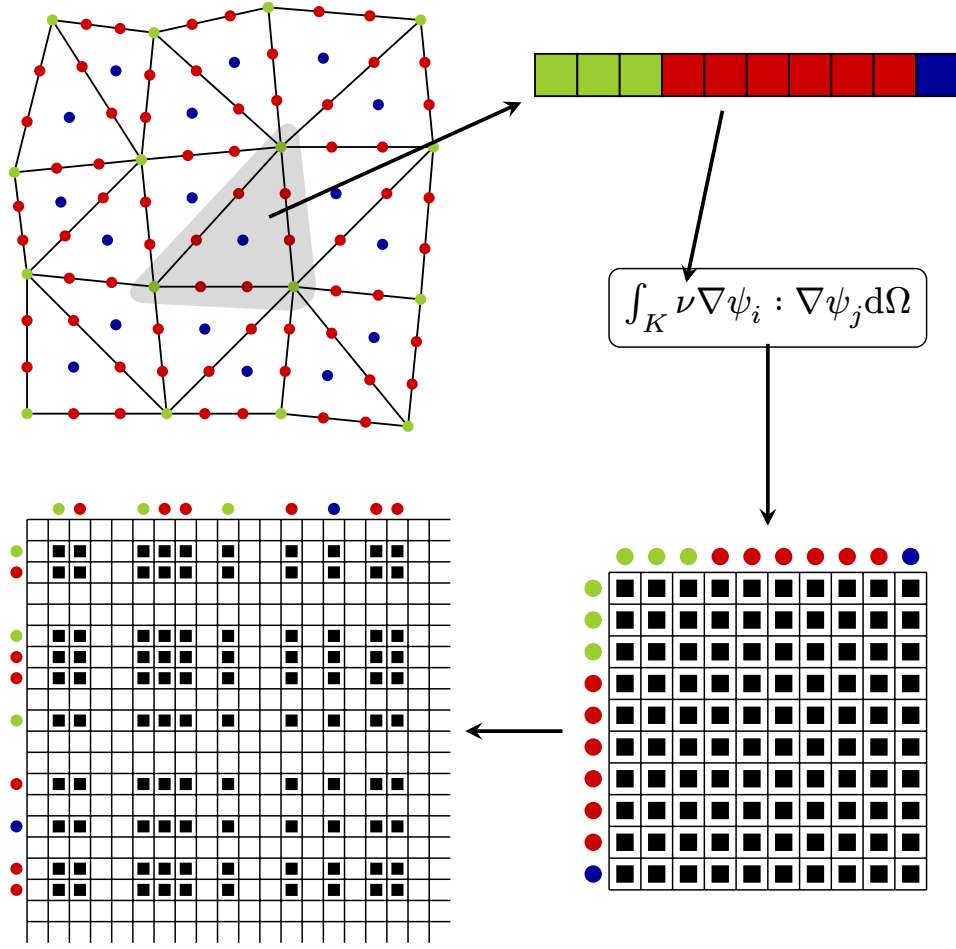


Figure 2.3: Diagram of the finite element global assembly process. For each cell in the mesh a coefficient is packed into a temporary array before the local kernel is evaluated. The result, a dense matrix, is then scattered to the global sparse matrix.

now be evaluated numerically using a quadrature scheme. This evaluation is termed *local assembly*, and code that performs it is called the *local kernel*.

Global assembly

Given such a local kernel, the final piece required for finite element assembly is an outer loop over the cells, $\mathcal{K}$, that adds cell-wise contributions into the global matrix or vector. This process is called *global assembly*.

Since the local kernel operates at the level of a single cell, it expects data to be provided to it as a small dense array, containing only those DoFs necessary for the calculation. However, the global data structures store DoFs differently, and so

Algorithm 1 Algorithm for assembling a finite element data structure with a single coefficient in the expression.

Require: <i>coefficient, output</i> Require: <i>temp0, temp1</i> 1: for <i>cell</i> in <i>mesh.cells</i> do 2: READ (<i>temp0, coefficient, cell</i>) 3: ZERO (<i>temp1</i>) 4: LOCALKERNEL (<i>temp0, temp1</i>) 5: INCREMENT (<i>output, temp1, cell</i>) 6: end for	▷ Input and output variables ▷ Work arrays ▷ Pack the coefficient ▷ Reset <i>temp1</i> ▷ Evaluate the integral for each cell ▷ Scatter (unpack) the result
---	---

preparing the data for the local kernel requires an explicit transformation between the representations. This transformation usually involves copying data from the global structure to a local temporary array, though this is not necessary in cases where the required data is contiguous in memory (e.g. with discontinuous function spaces). Transforming from global to local representations is called *packing*, and from local to global *unpacking*.

A diagram showing the global assembly loop process, including explicit pack/unpack steps, is shown in fig. 2.3 and an algorithm for its execution in algorithm 1. In the algorithm there is a single pack instruction, **READ** (line 2), and a single unpack instruction, **INCREMENT** (line 5). Both instructions take the current iterate, *cell*, as an argument because the data that is packed/unpacked is dependent upon it.

Finite element assembly is a classic example of a mesh stencil calculation: one loops over entities in the mesh (the cells), accesses data that are ‘close’ to the current iterate, and performs a local computation. Further examples of mesh stencil calculations are discussed in section 10.2.

2.1.5 Finite element data structures

Determining the right pack/unpack operations for a finite element global assembly loop is challenging because: (a) unstructured mesh data has a non-trivial structure that is difficult to reason about, and (b) DoFs corresponding to different types of mesh entity (e.g. edges and vertices) must be packed together contiguously. These issues are jointly demonstrated in fig. 2.4, which shows a packing operation for a Scott-Vogelius function space suitable for solving the Stokes equations. Both the local and global representations of the data have some hierarchical structure. For instance:

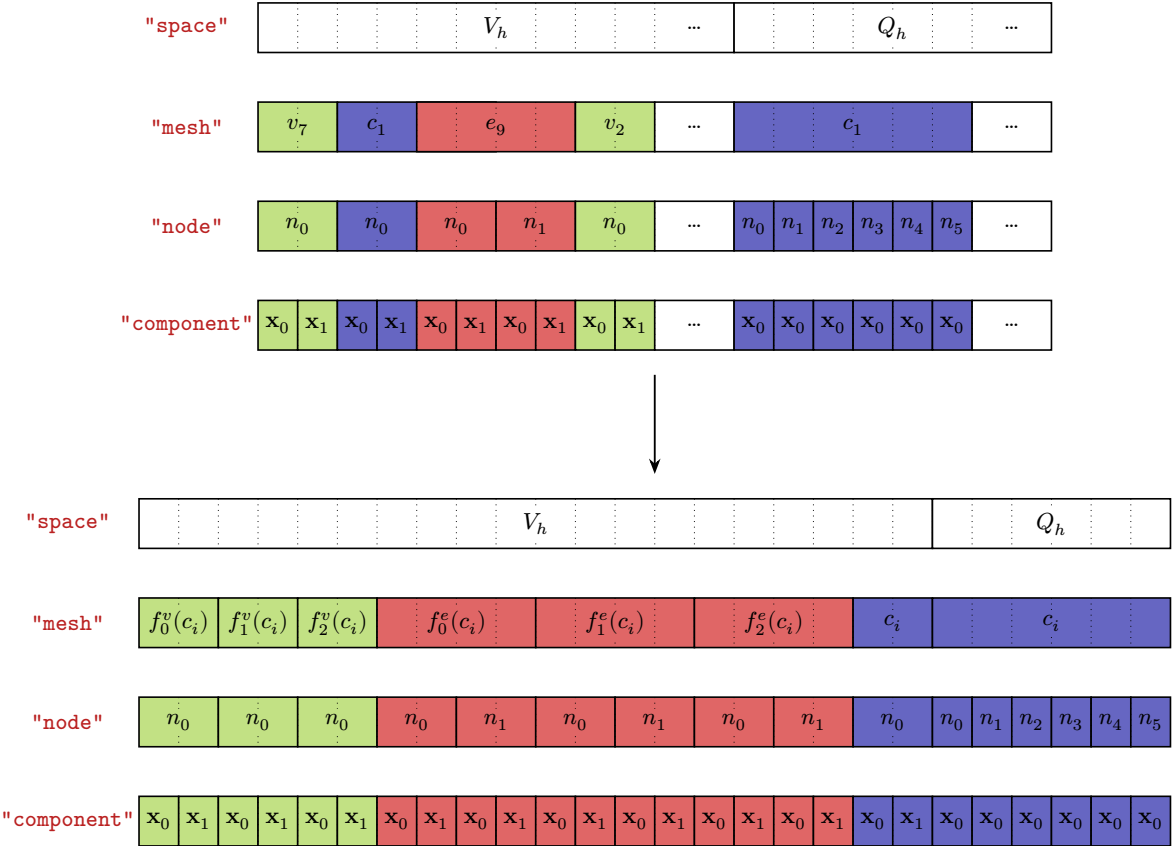


Figure 2.4: The packing transformation taking global data (top) to packed local data (bottom) for a two-dimensional Scott-Vogelius function space. The global data stores all DoFs for the entire function space whilst the packed data only contains the DoFs present in the current stencil iterate. DoFs associated with vertices, edges, and cells are coloured green, red, and blue respectively.

- All of the DoFs in space V_h precede those of Q_h .
- The $\mathbf{x}_0$ and $\mathbf{x}_1$ vector components of V_h are stored adjacently in memory.
- For V_h , vertices and cells have one node, whilst edges have two.

Since the packed local data (bottom) only exists as a temporary its contents are parametrised by the current cell iterate, c_i . For example, the first vertex that is packed for a particular cell is represented by the function $f_0^v(c_i)$ where f_i^v is the map relating cells to incident vertices.

Developing systems to represent this hierarchical structure is challenging, and forms one of the main contributions of this thesis.

2.2 Abstractions for mesh stencils

When writing continuum mechanics simulations there are myriad considerations that must be taken into account. These include both high-level mathematical choices such as the PDE to be solved, the discretisation, and the choice of solver; as well as low-level software design choices including the target backend (CPU or GPU), the sparse matrix format, and the layout of vector data.

Even assuming a specific PDE, the unfortunate fact that unites the remaining variables is that they are all *hardware specific*. Variations in characteristics between different high-performance computing (HPC) systems mean that, to get optimal performance, any of these may need to be changed. This limitation is obvious in some cases, for example CPU-only applications cannot be run on a GPU, but less obvious in others where, for instance, one may need to modify the numerics of the simulation (e.g. [12]).

Many continuum mechanics software packages commit eagerly to particular implementation choices and ‘bake them in’ to the software stack, making backing out such changes at a later date very difficult. Nowhere is this more obvious than with the recent ongoing transition from CPUs to GPUs as the dominant architecture on HPC systems. Many popular simulation packages still do not support GPUs and those that do are only able to because of an immense community effort involving large numbers of developers (e.g. [61, 62, 18, 1]).

In order to avoid this sort of huge re-engineering effort, developers can use libraries with interfaces that exist at a high level of *abstraction*. Dealing with abstract

interfaces allow low-level details to be hidden, resulting in a unified codebase where support for different implementation choices is delegated to the library. Examples of such abstractions include SYCL [85] and OpenCL [80], which provide unified interfaces for programming heterogeneous hardware, and, at a higher level, the Basic Linear Algebra Subprograms (BLAS) linear algebra interface [55], which allows users to reason about their program using vectors and matrices as primitives, instead of lower-level operations like loops and scalar operations.

The use of high-level abstractions leads to a separation of concerns between users of a library, who reason at the level of the abstraction, and the library developers, who implement it. The result is a programming environment where users are able to produce high performance code in a productive way.

Composable abstractions

The value of high-level abstractions can be further increased by making them *composable*. Instead of presenting a fixed, finite set of function calls to the user, a library may instead present a collection of building blocks from which a user may build a bespoke program.

Having composable abstractions is beneficial for two reasons. Firstly, it enables the library to describe, and hence automate, the solution of a much wider selection of programs. Secondly, it makes it easy to experiment with different approaches for solving a problem, which is sometimes necessary to get optimal performance on a particular platform.

One example of a successful library built upon composable abstractions is PETSc [7, 8, 6], a library for massively parallel linear algebra. Provided with the requisite vectors and matrices, PETSc exposes a powerful solver configuration language, with which different types of solvers and preconditioners may be specified and composed arbitrarily [17, 78]. This allows one to achieve excellent performance on different systems using only high-level configuration changes (e.g. [12]).

Composable abstractions are sometimes referred to as *domain-specific languages* (DSLs).

Code generation

It is especially challenging to develop high-level abstractions for mesh stencil calculations because of the level at which problem-specific information is exposed.

Compared with BLAS-like linear algebra routines, where operations are expressed over entire matrices and vectors, the specifics of the PDE and discretisation manifest in the performance-critical local kernel and pack/unpack operations.

As a result, implementing high-level abstractions with function calls is not suitable because the additional overhead associated with dispatching to a function call incurs a substantial performance penalty. Code generation can be used to solve this problem. Instead of hiding implementation details behind abstract operations at run-time, a *compiler* can be used to take a symbolic representation of the problem and convert it into fast code. Since the details of the abstraction are only present at compile-time, code generation can be thought of as a ‘zero-cost’ abstraction.

Examples of continuum mechanics simulation packages where code generation is used for the mesh stencil step include the finite element libraries DUNE-FEM [23], FEniCSx [9], and Firedrake [31], all using the Unified Form Language (UFL) [3] as a DSL for expressing weak forms, as well as the finite difference library Devito [56, 57] which uses SymPy [60] as a DSL.

In addition to providing a zero-cost abstraction, the use of domain-specific compilers can also facilitate the application of specialised optimisations that are challenging to implement by hand. One example of this is sum factorisation, also known as strength-reduction, a FLOP-reduction technique applicable to finite element simulations with tensor-product structure [51, 36]. The compiler can even sometimes perform auto-tuning, where a space of possible solutions is explored to find the fastest one. Examples include ATLAS (dense linear algebra) [89], SPIRAL (signal processing) [66], FFTW (discrete Fourier transforms) [27], and Halide (image processing) [67]. If performed at runtime with additional problem-specific information, auto-tuning is also sometimes referred to as an *inspector-executor strategy* (e.g. [10, 48, 81]).

Mesh stencil abstraction design

In order for any abstraction to be effective it must adequately capture the semantics of the problem domain so that the desired program can be automated. The choice of appropriate types and interfaces is therefore critical to its utility. For the design of a mesh stencil abstraction there are a number of clear requirements:

- There must be a mechanism for declaring loops over mesh entities.

- It should either accept, or be able to express, arbitrary local kernels that accept any number of inputs and outputs.
- There must be a mechanism for specifying the pack/unpack operations that relate global data to the current iterate.
- Scalar, vector and (sparse) matrix types should all be supported.

Further, it is not always the case that stencil algorithms consist of a single loop and single local kernel. Whilst this model holds for finite element assembly, other algorithms important for solving continuum mechanics problems are not expressible in this form. Examples include hybridisation [4, 14], where multiple local kernels need to be evaluated at each stencil iterate [28], and additive-Schwartz methods [5], which operate on small patches of cells and hence two nested loops: one over the patches and another over the cells in each patch [25]. For the abstraction to be able to express these algorithms it is thus important that it has a flexible and composable interface, in effect, a DSL.

2.2.1 Desirable characteristics for a mesh stencil package

In addition to the requirements of a mesh stencil abstraction listed above, there a number of characteristics that it is desirable for a mesh stencil package to have support for. These are:

Distributed memory parallelism

In massively parallel simulations the data structures are often too large to be stored in the memory of any individual node. Instead, they are *distributed* between all of the processes, with each process owning, and seeing, only a small piece of the entire structure.

For vectors, where DoFs are associated with topological entities, it is usual to store the DoFs according to some partitioning of the mesh. Depending on the stencil calculation, this requires that processes store additional DoFs, called *ghost* DoFs, at their boundaries that belong to different processes. An example of a vector partitioned according to a mesh partition is shown in fig. 2.5. Ghost DoFs are shown as existing on both processes which ensures that all of the values accessed by the stencil are present locally.

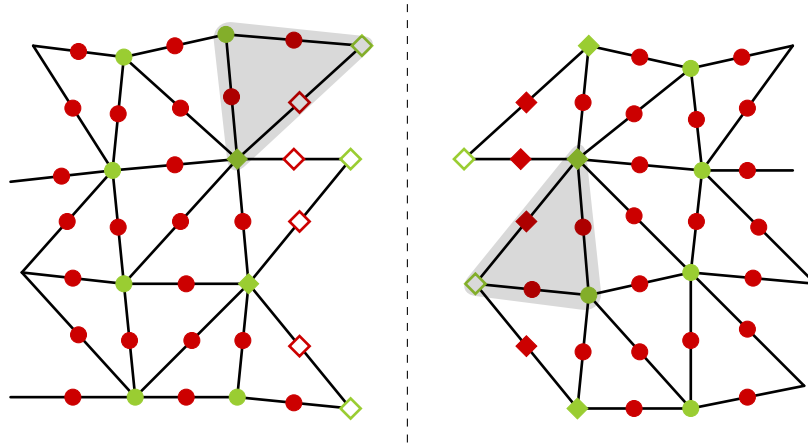


Figure 2.5: Diagram showing the DoFs for a function distributed between 2 processes. DoFs existing on multiple processes are represented by diamonds, with filled diamonds being those belonging to that process and hollow ones being ghost DoFs. DoFs that are not duplicated across processes are shown as circles. Sample stencils are shown in grey to demonstrate that the ghost DoFs are required for the stencils to be computable.

For matrices, the DoF partitioning process is less straightforward. Rather than implementing a parallel matrix type by-hand it is common to use existing library implementations instead. PETSc, for example, provides a parallel matrix type that has routines for accessing values that hide the complexities of reading off-process DoFs.

It is important to note that the focus here is solely on *distributed* memory parallelism. *Shared* memory approaches to parallelism using tools such as OpenMP [65] are not considered because they cannot scale past a single node of a supercomputer and hence have limited applicability. It is possible to employ *hybrid* memory models that combine both paradigms (e.g. [52]), but that is considered out of scope for this work.

Mesh renumbering

During the execution of a mesh stencil, the input and output global data structures are accessed with indirection maps. Accessing data indirectly incurs a performance penalty because it defeats hardware-implemented memory optimisations such as prefetching, where adjacent blocks of memory are loaded into a cache line together. To improve data locality, a common optimisation is to *renumber* the mesh entities such that, say, edge DoFs are ‘close’ to the DoFs of their incident cell, as well as ensuring

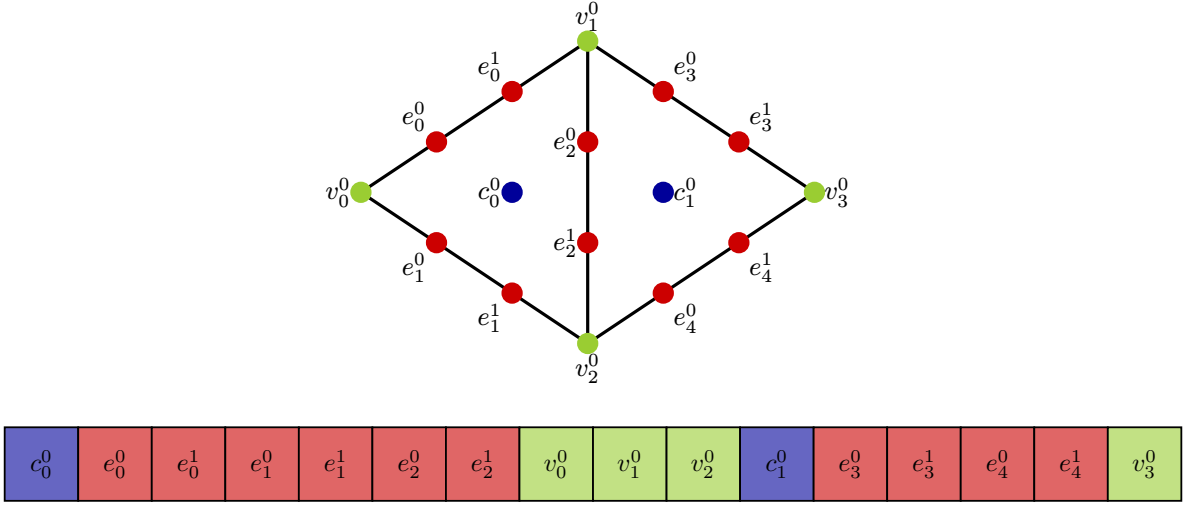


Figure 2.6: Data layout for a P_3 function space on a two cell mesh. The mesh data has been renumbered such that edge and vertex DoFs are ‘close’ to their incident cell.

that adjacent cells are also close in memory (e.g. [53], algorithm 3). This increases both the chance that adjacent required DoFs will be loaded onto the same cache line (spatial locality), and also the chance that DoFs shared between successive iterates will already be in cache (temporal locality). An example of such a renumbering is shown in fig. 2.6; the data for the vector is arranged such that incident edge and vertex DoFs are adjacent in memory to their cell.

Data layout flexibility

In order to get optimal performance on different platforms, especially when transitioning from CPU to GPU architectures, the way in which data are laid out must be altered (e.g. [59, 83]). This poses a challenge to many simulation packages as they often commit to a particular representation of the data, typically one optimised for CPU execution, making performance portability difficult to achieve without a substantial rewrite of much of the software. One possible way in which this manifests is whether to use a ‘struct-of-array’ (SoA) or ‘array-of-structs’ (AoS) data layout. Applied to a finite element function space these data layouts can manifest as, say, having vector components DoFs stored adjacent to one another (AoS, fig. 2.4) or as separate contiguous blocks (SoA, fig. 2.7).

For mesh stencil codes it is therefore desirable that a variety of different data layouts are supported. One way in which this can be achieved is to decouple the

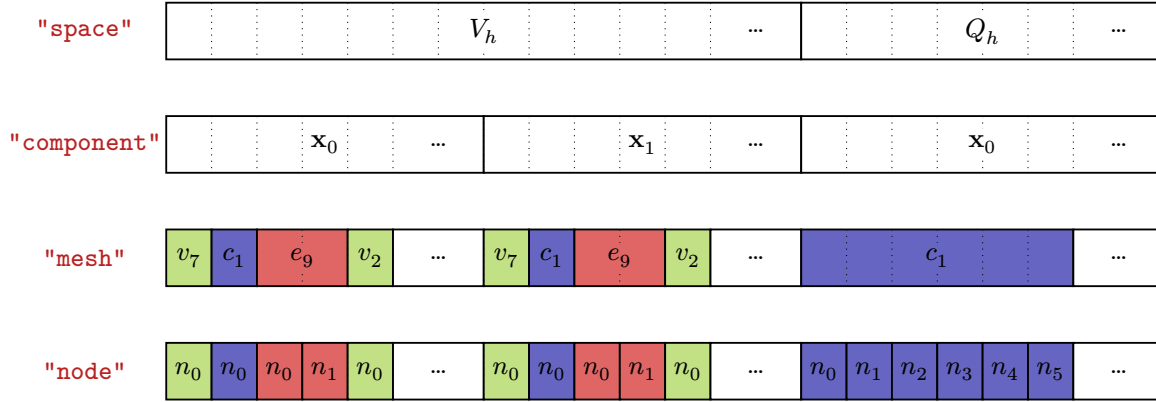


Figure 2.7: An alternative, ‘struct-of-array’ data layout for the Scott-Vogelius function space of fig. 2.4. Notice how the components of the vector space V_h are now stored apart from one another.

semantics of the data from its implementation. Taichi [39], a DSL for graphics simulations, and TACO [42], a compiler for tensor algebra, both provide high-level interfaces for constructing data layouts. These specifications exist independently of the algorithm, and code generation is used to bridge the gap. This lets one write a generic algorithm once and experiment with a range of different storage formats to find the one that performs best.

2.2.2 Existing mesh stencil packages

A number of mesh stencil packages already exist, each meeting a subset of the requirements laid out above (see table 2.1). **Liszt** [24], **Simit** [41], **Ebb** [11], and **MeshTaichi** [91] are DSLs for mesh stencil computations embedded in Scala, C++, Lua, and Python respectively. Each provides a rich API for expressing a variety of loop constructs and stencil operations that may be compiled and executed on a selection of different backends. Liszt has support for shared-memory, distributed-parallel, and GPU backends whereas Simit, Ebb, and MeshTaichi support shared-memory parallelism and GPUs.

In contrast with these expressive DSLs, **OP2** [64] and **PyOP2** [68] are mesh stencil packages with much more restrictive interfaces. Rather than providing a full problem-solving environment, OP2 and PyOP2 programs have a single entrypoint, termed a *parallel loop*, that takes an externally provided local kernel along with an *iteration set* and a list of *arguments*, representing the pack instructions for the kernel inputs and outputs. This approach maps well to FEM-style assembly loops such as

	Liszt	Simit	Ebb	MeshTaichi	OP2	PyOP2	pyop3
Composable DSL	✓	✓	✓	✓	✗	✗	✓
Distributed parallelism	✓	✗	✗	✗	✓	✓	✓
Mesh renumbering	✗	✓	✗	✗	✓	✓	✓
Data layout flexibility	✗	✗	✗	~	✓	✗	✓
Actively maintained	✗	✗	✗	✓	✓	✓	✓

Table 2.1: Comparison of the features of `pyop3` with pre-existing mesh stencil packages. ‘~’ indicate partial implementation of a feature.

that shown in algorithm 1, but more complex loop or kernel structures cannot be represented without sui-generis, non-composable changes.

OP2 is an *active library* that provides source-to-source translation from C, C++ or Fortran to a range of different backends. PyOP2 is a reimplementation of the OP2 API in Python, using run-time code generation to compile the parallel loops instead of source-to-source translation. By using code generation, PyOP2 is easier to integrate with other code generation packages and in particular it is a core component of the Firedrake finite element framework [31].

Of the packages that support distributed memory parallelism, OP2 and PyOP2 differ from Liszt in that their abstractions have no native mesh type, they instead deal in more abstract *sets* and *mappings* between sets. Taking a set-centric approach is beneficial as it allows one to interface with arbitrary mesh packages (e.g. PETSc DMPlex section 3.2) instead of a custom implementation, and thus delegate difficult to implement and unrelated operations like parallel partitioning and refinement. Further, by dealing with sets instead of topological entities, OP2 and PyOP2 have the freedom to store data for function spaces where the DoFs are tied to multiple types of topological entity. This makes mesh renumbering optimisations possible.

Regarding data layout flexibility, Liszt, Simit, Ebb, and PyOP2 all eagerly commit to a particular data layout which cannot be changed. MeshTaichi inherits some of the data layout flexibility from Taichi but this only applies to fields per topological entity, which are always considered separately. OP2 alternates between AoS and SoA layouts depending on the target backend and whether or not the data are addressed directly or indirectly.

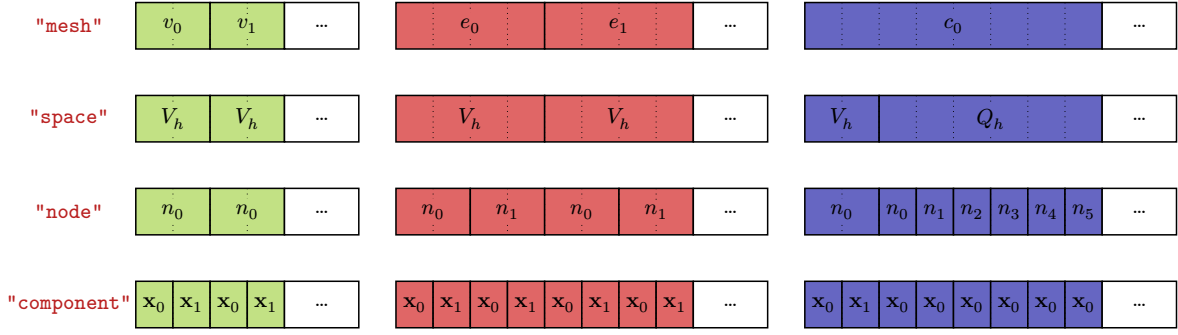


Figure 2.8: The closest possible data layout for fig. 2.4 for a library that associates unknowns with topological entities. Data for each topological entity are stored in separate arrays.

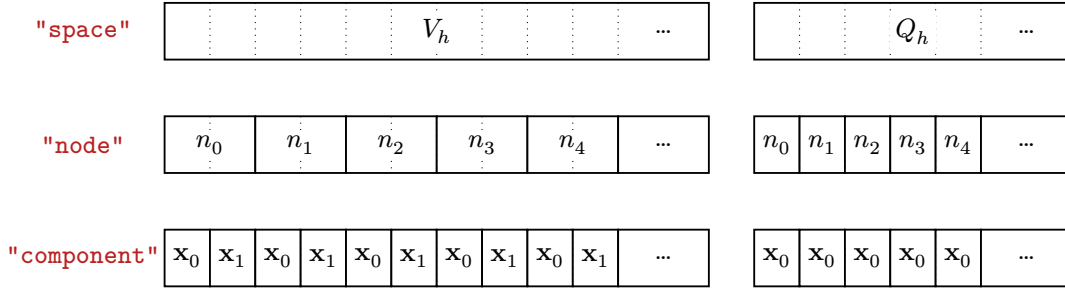


Figure 2.9: The data layout matching fig. 2.4 as it would be stored by PyOP2. Data for each function space (V_h and Q_h) are stored in separate arrays.

2.2.3 The missing abstraction

Of the packages listed above, none of them are able to meet all of the requirements in table 2.1 because of fundamental limitations in their abstractions for representing mesh data.

Liszt, Ebb, and MeshTaichi all store mesh data by associating fields with mesh entities. This facilitates the development of an expressive DSL because mesh entities are first-class citizens of both the DSL and the data layout specification, making mapping between them straightforward. However, this approach to storing data means that high-order function spaces, that store DoFs on multiple entity types (e.g. fig. 2.2), are difficult to express and have to be stored with the DoFs on each entity type stored separately. An example of this for the Scott-Vogelius element pair is shown in fig. 2.8. Since the entity arrays are stored separately, mesh renumbering data locality optimisations are impossible to express.

As an alternative approach, Simit, OP2, and PyOP2 are able to represent function space DoFs more generically as *sets of nodes*. Being able to treat all the nodes of a function space equivalently means that mesh renumbering is possible, but it comes at the expense of topological information. This is demonstrated in fig. 2.9, which shows the data layout for a Scott-Vogelius function space as it would be stored in PyOP2. Compared with fig. 2.4, one can immediately see that the **"mesh"** layer of the structure is no longer present, illustrating that the relation between nodes and their owning mesh entities is lost. Since topological information is lost, it is difficult to build a composable DSL, explaining why OP2 and PyOP2 both only have restricted interfaces. Simit can define data structures using either mesh entities or node sets and so it can either be a composable DSL or support high-order function spaces, but not both at the same time.

To meet the requirements of table 2.1 it is therefore necessary that a new abstraction for mesh data be developed that is capable of representing complex, interleaved data layouts whilst retaining sufficient topological information to enable an expressive DSL. Developing such an abstraction forms the central contribution of this thesis.

Chapter 3

Existing abstractions for arrays and stencils

In this chapter we review three pieces of existing software: NumPy, PETSc DMplex, and PyOP2, whose designs have served as inspiration for the contributions of this thesis. In particular, NumPy provides an intuitive language for describing hierarchically structured data, PETSc DMplex provides an abstraction for reasoning about data stored on meshes, and PyOP2 is an existing mesh stencil library that is the precursor to the new stencil library introduced by this thesis, `pyop3`.

3.1 NumPy

As established in chapter 2, data structures for mesh-based computations have a non-trivial hierarchical structure that existing mesh stencil packages are incapable of representing in full. Prior to introducing any new abstractions it is therefore valuable to examine existing techniques for handling hierarchically structured data. To that end we will review NumPy [32], the fundamental package for array-based computation in Python. In addition to considering implementation details, elements of NumPy’s interface will be presented since, given NumPy’s ubiquity, our DSL will aim to reuse much of its terminology and syntax.

3.1.1 N-dimensional arrays

In NumPy, hierarchical data structures are represented by *N-dimensional arrays*. These have a multi-level *shape*, with each level called an *axis* (e.g. fig. 3.1, left).

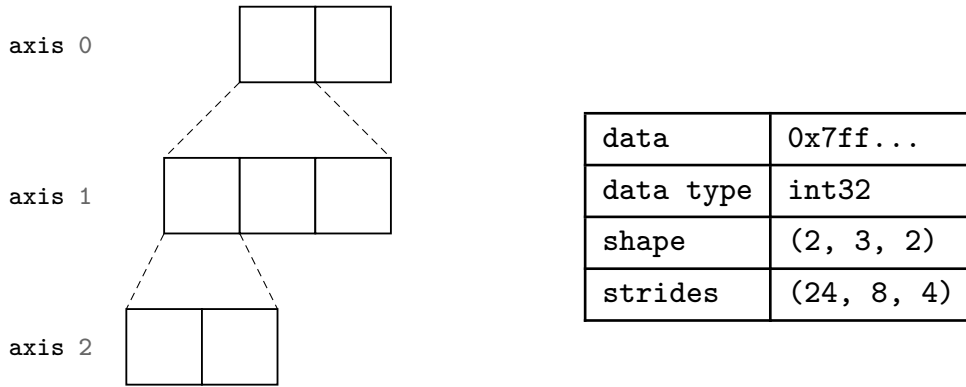


Figure 3.1: The data layout (left) and internal representation (right) of an `int32` NumPy array with shape `(2, 3, 2)`. Strides are given in bytes for each axis.

To represent arrays, NumPy internally combines a memory buffer, containing the actual values, with additional *metadata* that is used to interpret the buffer. In particular, the metadata includes the shape of the array, how to stride through it, and its data type (e.g. fig. 3.1, right).

In order to access a value from the array a *multi-index* - a list of integers, one per axis - must be provided. Given a multi-index, the correct value may be read from the array by first computing an offset, and then reading a value at that offset. The function used to compute this offset is a function of each multi-index. As an example, for the array shown in fig. 3.1, the offset is given by

$$\text{offset}(i_0, i_1, i_2) = 6i_0 + 2i_1 + i_2 \quad (3.1)$$

where i_0 , i_1 , and i_2 are the values of the multi-index for axes 0, 1, and 2 respectively. The Python syntax for performing this access would be `array[i0, i1, i2]`.

Representing data in this way also enables array transformations to be expressed without needing to copy data. Arrays can be reshaped or transposed just by constructing a new metadata struct with alternative shape and/or stride information. These alternative representations for the same array are termed *views*.

3.1.2 Indexing arrays

As well as being reshapable, arrays may also be *indexed*, where portions of the full array are extracted to yield a new array. For this thesis the most relevant indexing operations (demonstrated in table 3.1) are:

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

Table 3.1: Important indexing operations for NumPy arrays. The examples shown apply the index to the string array ["A", "B", "C", "D", "E", "F"] (called `array` above).

Slicing For affine indexing operations one can pass a Python `slice` object to index an axis. The syntax to construct a slice is `[start:stop:step]`, where omitting `start`, `stop`, or `step` will default to 0, the end of the array, and 1 respectively. Due to the regular access pattern of a slice, indexing NumPy arrays with them will return a view.

Integer array indexing For irregular access patterns, an array of indices may be used to select entries from a given axis. In contrast with slices, this method is inherently irregular and so a copy of the array is returned, instead of a view.

3.2 PETSc DMplex

As discussed in section 2.2.3, existing mesh stencil packages take one of two approaches to describe mesh data, either associating data directly with specific entity types, or by using the more flexible concept of node sets. Both representations are imperfect: the former cannot represent complex data layouts and the latter loses topological information.

One piece of software that offers a unifying solution to these conflicting approaches is DMplex [46, 47, 53], the unstructured mesh component of PETSc [6, 7, 8]. DMplex represents meshes as directed acyclic graphs (DAGs) where mesh entities, termed *points*, are the vertices of the graph and entity covering relations are its edges (e.g. cells connect to facets). Since the DAG forms a partially-ordered set, it is common to visualise DMplex instances as Hasse diagrams (fig. 3.2). All entities with the same topological dimension are shown as different layers, or *strata*, of the diagram.

By treating mesh entities equivalently, DMplex is able to express mesh algorithms generically in a dimension-independent fashion (e.g. parallel mesh distribution [47],

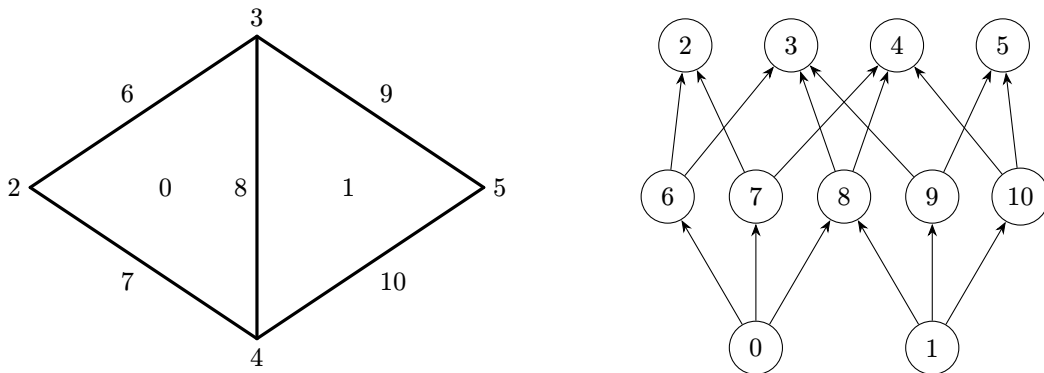


Figure 3.2: The Hasse diagram (right) for a two-cell triangular mesh (left).

checkpointing [30]). It also becomes trivial to represent data stored on different mesh entities (section 3.2.2).

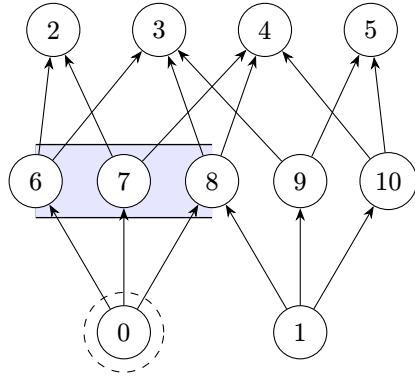
3.2.1 Topological queries

To traverse the DAG and build appropriate stencils, DMPlex provides two core topological query operations: *cone* and *support*. The cone of a point, written $\text{cone}(p)$, is the set of points that are pointed to *from* p . The support of a point, written $\text{supp}(p)$, is the dual of cone and yields the set of points that point *to* p . Transitive closures for both of these operations, where the operation is applied repeatedly and all points are accumulated, are also supplied. The transitive closure of cone is termed the *closure*, written $\text{cl}(p)$, and the transitive closure of support is called the *star*, written $\text{st}(p)$. These operations are shown in fig. 3.3.

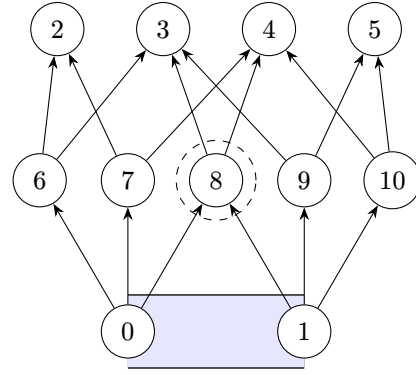
It is possible to compose these operations to yield larger stencils. For instance $\text{supp}(\text{cone}(p))$ with p a cell produces the set of cells sharing an edge with p (and p itself). Similarly, $\text{cl}(\text{st}(p))$ is the *adjacency* relation for finite element calculations: basis functions on points within this stencil have non-zero support with the basis functions on p .

3.2.2 Representing data layouts

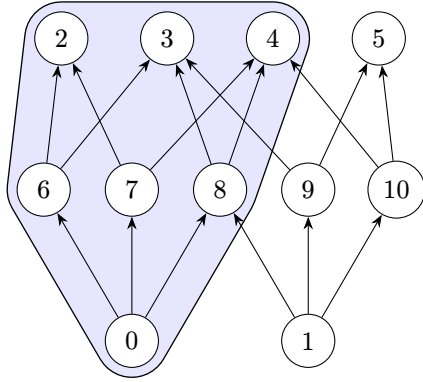
In order to store data on a DMPlex, users must construct a `PetscSection`, a simple data structure that associates DoFs with mesh points. Code for constructing a `PetscSection` is shown in listing 1.



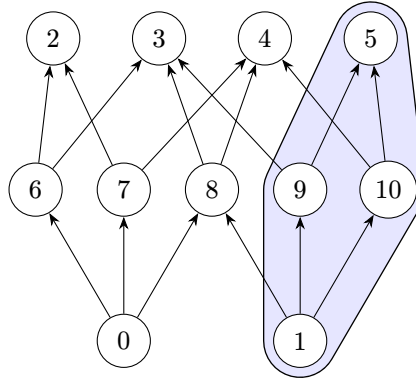
(a) cone: $\text{cone}(0) = \{6, 7, 8\}$



(b) support: $\text{supp}(8) = \{0, 1\}$



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



(d) star: $\text{st}(5) = \{5, 9, 10, 1\}$

Figure 3.3: The possible DMplex covering queries applied to the mesh from fig. 3.2.

Listing 1 C code constructing an appropriate `PetscSection` for a $[P_3]^2$ finite element (section 2.1.3). Some boilerplate code is omitted.

```

1  // set cell DoFs (2 per cell)
2  DMPlexGetDepthStratum(dmplex, 2, &start, &end);
3  for (int cell=start; cell<end; cell++)
4      PetscSectionSetDof(section, cell, 2);
5
6  // set edge DoFs (4 per edge)
7  DMPlexGetDepthStratum(dmplex, 1, &start, &end);
8  for (int edge=start; edge<end; edge++)
9      PetscSectionSetDof(section, edge, 4);
10
11 // set vertex DoFs (2 per vertex)
12 DMPlexGetDepthStratum(dmplex, 0, &start, &end);
13 for (int vertex=start; vertex<end; vertex++)
14     PetscSectionSetDof(section, vertex, 2);
15
16 PetscSectionSetUp(section);

```

Algorithm 2 The tabulation algorithm that determines the right offsets from a `PetscSection`. This code is executed during `PetscSectionSetUp()`.

Require: *permutation* ▷ Integer array storing a point renumbering
Require: *offsets* ▷ Output offset array

```

1: counter ← 0
2: offset ← 0
3: for point in mesh.points do
4:     offsets[counter] = offset ▷ Store the current offset
5:     counter ← counter + 1
6:     renumbered ← permutation[point]
7:     offset ← offset + PETSCSECTIONGETDOF(renumbered)
8: end for

```

PetscSections work by converting DoF counts for points into offsets that, like NumPy, can be used to read from an array. As a simple example, given the DoF count `[1, 0, 3, 2, 0, 1]` the **PetscSection** would tabulate the offsets to be `[0, 1, 1, 4, 6, 6]`. These offsets are computed when **PetscSectionSetUp()** is called using the tabulation algorithm shown in algorithm 2.

Also, **PetscSections** are able to express the renumbering locality optimisations described in section 2.2.1. To do so one simply provides the **PetscSection** with an optional *permutation* that is accounted for during set up. This is demonstrated in algorithm 2.

Limitations

Whilst a powerful and flexible tool for describing data layouts, **PetscSections** have a number of shortcomings:

They are fully ragged **PetscSections** do not distinguish between different types of topological entity and so important information about the structure of the computation cannot be represented. DoF information is stored per point in a completely unstructured way so **PetscSections** are incapable of knowing, say, that every cell in the mesh stores exactly one DoF. This can prevent the compiler from making certain optimisations (e.g. loop unrolling) which can significantly affect performance. Further, not knowing the loop sizes a priori means that an additional array needs to be passed about during the calculation, increasing the memory footprint and thus directly harming the performance of memory-bound codes.

Shape information is lost Though **PetscSections** 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 **PetscSection**, a point with a single vector-valued node is indistinguishable from a point with multiple scalar nodes.

3.2.3 Parallel

DMplex works in distributed memory environments by partitioning the entire global mesh into local pieces that are kept on each process. To enable stencil calculations

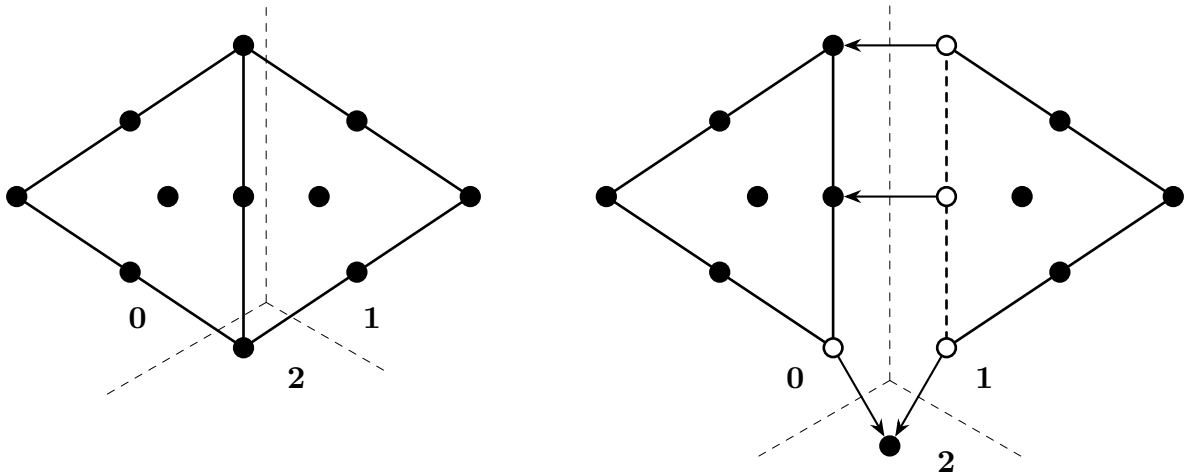


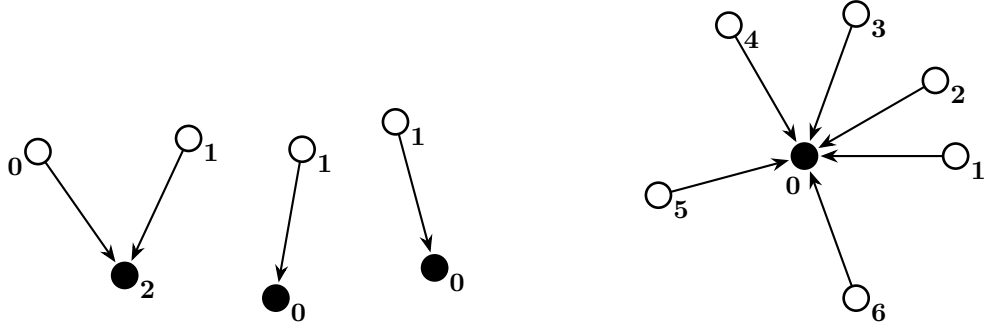
Figure 3.4: Global (left) and process-local (right) views of a mesh partitioned between three processes. The black dots represent the different mesh entities (the *points*). Ghost points are shown as hollow circles.

ghost points are duplicated across processes. An example partitioned mesh is shown in fig. 3.4.

Parallel communication in DMPlex is handled by *star forests* [92]. Star forests relate equivalent points across processes as a collection (forest) of depth-1 trees (stars) consisting of a single *root* and any number of *leaves*. They support two main operations: *broadcasts* from roots to leaves and *reductions* from leaves to roots.

Star forests are a powerful abstraction for describing a range of different parallel communication patterns. For example, a value shared globally across n processes can be represented as a star forest containing a single star, with the root node on process 0 and $n - 1$ leaves, 1 for each other process (fig. 3.5b). For distributed meshes, star forests encode point ownership by making each shared point into a separate star, with the owning process the root and ghost processes the leaves. A sketch of the star forest encoding the point ownership for the partitioned mesh in fig. 3.4 is shown in fig. 3.5a. DMPlex refers to this object as the *point star forest*.

To share parallel data, instead of points, the point star forest is not suitable because the number of DoFs per point is variable. A new star forest, called the *DoF star forest*, is therefore created. To produce it, DMPlex composes the point star forest with the `PetscSection` encoding the data layout. Figure 3.6 shows an example of how mesh data might be shared between processes.



(a) Star forest suitable for describing the point ownership in fig. 3.4.

(b) Star forest encoding the communication pattern of a globally consistent value shared between 7 processes.

Figure 3.5: Common star forest patterns. Each point is labelled with its owning process and root and leaf points are shown in black and white respectively.

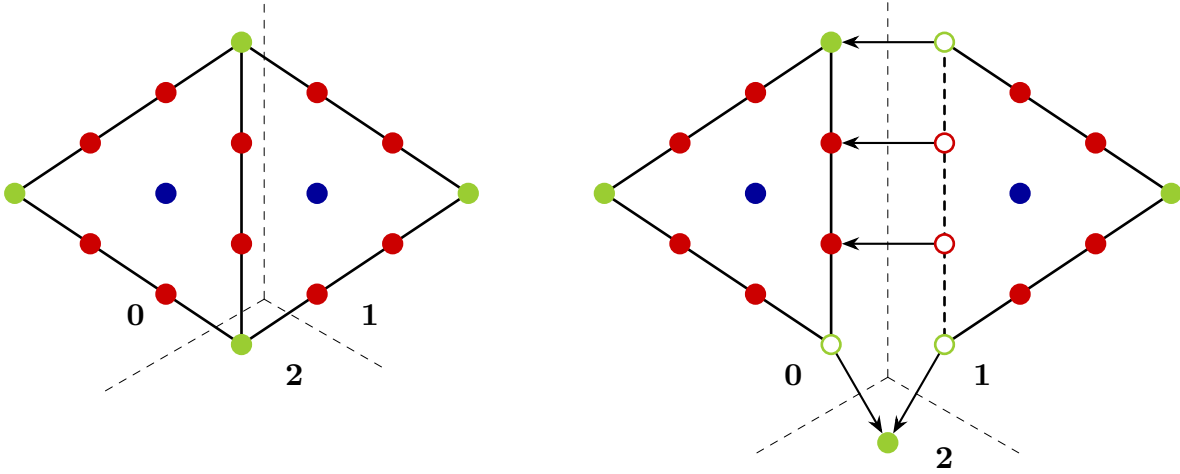


Figure 3.6: Global (left) and process-local (right) views of a P_3 function space partitioned between three processes. Ghost DoFs are shown as hollow circles.

	Global	Dat	Mat
Usage	Globally constant value	Distributed vector	Distributed matrix
Constructor	GlobalDataSet	DataSet	2 DataSets
Storage type	NumPy array	NumPy array	PETSc Mat

Table 3.2: PyOP2 global data structures.

3.3 PyOP2

As first introduced in section 2.2.2, PyOP2 is a library for doing mesh stencil calculations [68]. It was originally introduced to provide the same abstractions as OP2 [64], but use runtime code generation instead of source-to-source translation. It is a core component of the Firedrake finite element framework [31].

3.3.1 Data structures

Summarised in table 3.2, PyOP2 has three supported data structures: **Globals** (constants), **Dats** (vectors), and **Mats** (matrices). Like NumPy, each of these are implemented by coupling ‘simple’ data structure to metadata that provides additional semantic information. In PyOP2 this metadata is called a *data set*. Internally, **Globals** and **Dats** both use NumPy arrays to store data, whereas **Mats** use PETSc matrices (that are also called **Mats**).

For multi-field problems such as the Stokes example in chapter 2 PyOP2 has *mixed data sets*, which are simply a collection of regular data sets. Mixed data sets produce **MixedDats** which are, again, simply a collection of regular **Dats**. An example of a PyOP2 data layout for a multi-field problem is shown in fig. 2.9. Observe that, since the DoFs are associated with multiple mesh entities, data are stored on a *node set*, losing topological information.

3.3.2 Parallel loops

In order to apply kernels to these data structures, a *parallel loop* is constructed and executed. The loop takes as arguments a *local kernel*, *iteration set* and any number of *arguments* that specify the pack/unpack instructions needed to prepare data for the local kernel. Constructing a parallel loop requires an invocation like that shown in listing 2. **kernel** is the local kernel representing the stencil operation to be performed

Listing 2 Code to construct and execute a PyOP2 parallel loop.

```

1  par_loop(kernel,                # local kernel
2      src_set,                    # iteration set
3      dat0(READ, src_to_target_map), # arguments
4      dat1(WRITE, src_to_target_map)) # "
```

per iteration, `src_set` is the set that is iterated over (termed the *iteration set*), and `dat0` and `dat1` are `Dat` objects that are passed through to the local kernel. `kernel` may be either a string of C code or a *loopy kernel* (see section 3.3.3). Both `Dats` associate data with some `target_set` and hence are addressed with a map relating these sets (`src_to_target_map`). `dat0` is accessed using the `READ` access descriptor, meaning that it only needs to be packed and passed into the local kernel, whereas `dat1` is accessed with `WRITE`, so it only requires unpacking.

Interleaving computation and communication

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, by the user, into 3 parts:

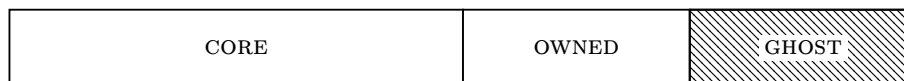
CORE Set elements that do not require any data from other processes during a parallel loop.

OWNED Set elements that are owned by the current process but require data from other processes.

GHOST Set elements present on a process that belong to another process.

From this partitioning it is possible to interleave computation and communication. Computations over `CORE` points do not read any ghost data, and hence can be executed without waiting for any communication to complete, whereas `OWNED` points need valid ghost data and so must wait for all communication to be completed before beginning. This is shown in algorithm 3. `GHOST` points are not iterated over.

To facilitate this algorithm, PyOP2 arranges sets, and thus stores data, in a partitioned manner, where `CORE` points precede all `OWNED` points and all `OWNED` precede all `GHOST` points:



Algorithm 3 The PyOP2 parallel loop execution algorithm to interleave computation and communication.

```

1: BEGINHALOEXCHANGES ▷ Begin sending halo data
2: for item in iter.set.core do ▷ Do the computations that do not need halo data
3:   COMPUTE(item)
4: end for
5: AWAITHALOEXCHANGES ▷ Wait for halo exchanges to complete
6: for item in iter.set.owned do ▷ Do the computations that do need halo data
7:   COMPUTE(item)
8: end for

```

Limitations

Arranging the data like this is problematic for one key reason: it conflates the partitioning of the data layout (GHOST/non-GHOST) with that of the iteration set (CORE/OWNED).

The partitioning of GHOST/non-GHOST points comes from the star forest description of the parallel overlap, and hence is known *when the array is initialised*. It therefore makes sense to embed this information in the data layout. By contrast, OWNED points are defined as being points that need ghost data in order to be executable. *This is dependent upon the stencil*. Differently sized stencils touch different numbers of adjacent points, changing the CORE/OWNED partitioning. PyOP2 will therefore not work correctly in parallel for programs that use several different stencil sizes acting on the same data.

Further, the partitioning of CORE/OWNED points is associated with the *iteration set*, instead of the data structures. This means that PyOP2 cannot do the right thing for cases where data structures do not share the same parallel decomposition (e.g. when transferring data between meshes). Stencils may touch only non-GHOST points on one mesh, leading to the iterate being classified as CORE, but they may be GHOST on the other mesh, and hence the iterate ought to be labelled OWNED.

This issue is demonstrated in fig. 3.7. Given two meshes, one a quadrilateral mesh (black) and the other a mesh of disconnected vertices (blue), with differing parallel distributions, and assuming a loop over the cells of the quadrilateral mesh, it can be seen that it is possible for the CORE/OWNED labels to be erroneously applied. Specifically, the cells labelled with a red **C** would be considered CORE by PyOP2 despite needing to access GHOST DoFs from the vertex-only mesh, and hence should really be marked as OWNED. Similarly, the cells labelled with a red **O** are considered

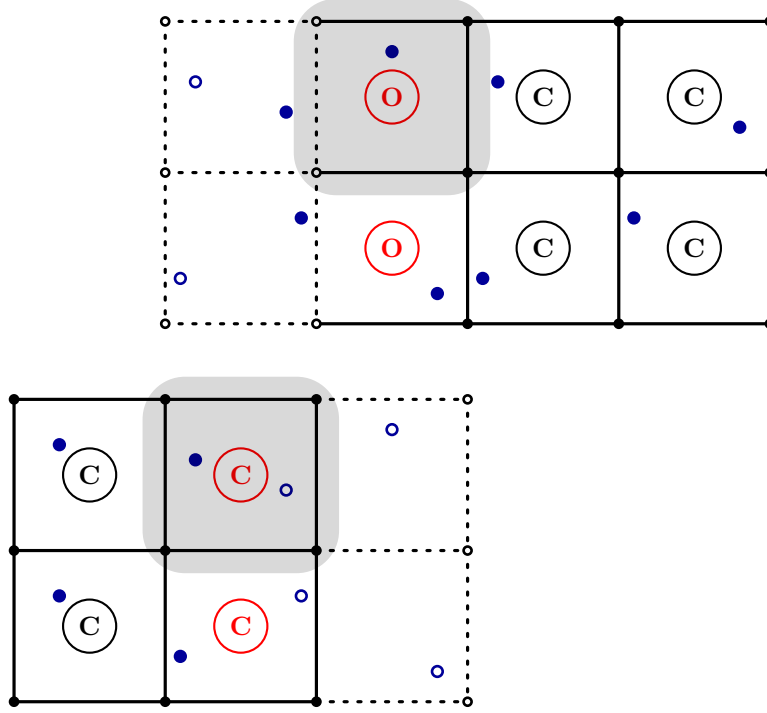


Figure 3.7: Example illustrating the issues that occur when assigning CORE/OWNED labels based solely upon the iteration set. In this example one loops over the cells of a partitioned quadrilateral mesh (black) and accesses data defined on a mesh of disconnected vertices (blue), which has a different parallel distribution. Ghost edges and vertices are marked with dashes and hollow circles respectively. Cells with the label CORE are marked with a **C** and those with OWNED are marked with a **O**. The problematic cells, erroneously labelled as either CORE or OWNED, are shown in red.

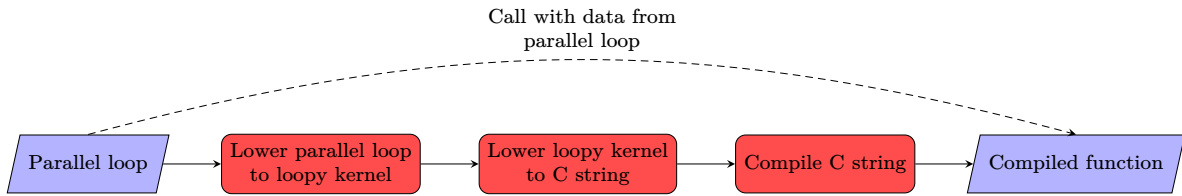


Figure 3.8: Simplified code generation pathway for a PyOP2 parallel loop.

OWNED even though they do not touch any GHOST DoFs, and thus could safely be labelled CORE.

3.3.3 Code generation

To turn abstract parallel loop definitions into executable, fast code, PyOP2 utilises run-time code generation. First, the parallel loop is *lowered* to loopy, which is then lowered to C code and compiled to a binary. This binary may then be called passing in the arguments from the original parallel loop. This process is summarised in fig. 3.8.

Loopy

Loopy [44], a polyhedral-model-inspired code generation library written in Python, is the primary intermediate representation of PyOP2. Given a parallel loop, PyOP2 generates a loopy `LoopKernel` which is then lowered to C code and compiled.

In order to construct a `LoopKernel` the user needs to provide *domains* (loops), *instructions*, and *arguments* (data passed into the kernel). For the parallel loop in listing 2, the `LoopKernel` and generated C code are shown in listing 3 and listing 4 respectively. The local kernel, here omitted, is also (usually) a `LoopKernel` and would be included in the generated code as a separate function.

Given its high level of abstraction, loopy is a convenient target for PyOP2 for a number of reasons:

- `LoopKernels` may be transformed to apply optimisations such as loop tiling, common subexpression elimination and vectorisation [43, 45, 84].
- Code may be generated for targets other than CPUs including the GPU languages OpenCL [80] and CUDA [20]. This means that PyOP2 is capable of executing parallel loops on GPUs without making intrusive changes to the code - a fact demonstrated (as a proof-of-concept) by [49].

Listing 3 Abbreviated textual representation of the loopy kernel generated for the example parallel loop in listing 2. The pack and unpack instructions are shown on lines 29 and 33 respectively.

```

1  -----
2  KERNEL: pyop2_kernel
3  -----
4  ARGUMENTS:
5  start: ValueArg, type: np.dtype('int32')
6  end: ValueArg, type: np.dtype('int32')
7  dat0: type: np.dtype('float64'), shape: (None, 2), in
8  dat1: type: np.dtype('float64'), shape: (None, 2), out
9  map0: type: np.dtype('int32'), shape: (None, 3), in
10 -----
11 DOMAINS:
12 [end, start] -> { [n] : start <= n < end }
13 { [i6] : 0 <= i6 <= 2 }
14 { [i7] : 0 <= i7 <= 1 }
15 { [i10] : 0 <= i10 <= 2 }
16 { [i11] : 0 <= i11 <= 1 }
17 { [i13] : 0 <= i13 <= 2 }
18 { [i14] : 0 <= i14 <= 1 }
19 -----
20 TEMPORARIES:
21 t0: type: np.dtype('float64'), shape: (1, 3, 2)
22 t1: type: np.dtype('float64'), shape: (1, 3, 2)
23 -----
24 INSTRUCTIONS:
25 for n, i14, i13
26     t1[i13, i14] = 0
27 end i14, i13
28 for i11, i10
29     t0[i10, i11] = dat0[map0[n, i10], i11]
30 end i11, i10
31 t1 = local_kernel(t0)
32 for i7, i6
33     dat1[map0[n, i6], i7] = t1[i6, i7]
34 end n, i7, i6
35 -----

```

Listing 4 The C code generated from the loopy kernel in listing 3. The pack and unpack instructions are shown on lines 13 and 20 respectively.

```
1  void pyop2_kernel(int32_t const start, int32_t const end,
2                      double const *__restrict__ dat0,
3                      double *__restrict__ dat1,
4                      int32_t const *__restrict__ map0)
5  {
6      double t0[3 * 2];
7      double t1[3 * 2];
8
9      for (int32_t n = start; n <= -1 + end; ++n)
10     {
11         for (int32_t i10 = 0; i10 <= 2; ++i10)
12             for (int32_t i11 = 0; i11 <= 1; ++i11)
13                 t0[2 * i10 + i11] = dat0[2 * map0[3 * n + i10] + i11];
14         for (int32_t i13 = 0; i13 <= 2; ++i13)
15             for (int32_t i14 = 0; i14 <= 1; ++i14)
16                 t1[2 * i13 + i14] = 0.0;
17         local_kernel(&(t0[0]), &(t1[0]));
18         for (int32_t i7 = 0; i7 <= 1; ++i7)
19             for (int32_t i6 = 0; i6 <= 2; ++i6)
20                 dat1[2 * map0[3 * n + i6] + i7] = t1[2 * i6 + i7];
21     }
22 }
```

3.3.4 Limitations

As an abstraction for mesh stencil computations, PyOP2 is limited because of its inflexible interface. Not all mesh operations are expressible as a single kernel executed within a single loop. For instance, algorithms for physics-based preconditioners such as hybridisation [28] and additive Schwartz methods [25] involve multiple kernels and nested loops. Though each has been implemented with PyOP2 and Firedrake, *sui-generis* code has been needed making the resulting software harder to compose.

One of the main reasons for this limiting interface is the fact that set-based approaches for describing mesh data *discard important topological information*. This makes it harder to build a composable interface because it creates a divide between the natural language for expressing mesh stencil algorithms - the mesh entities - and the description of how data are stored - on the nodes.

3.4 Outlook

In this chapter we introduced three existing pieces of software: PyOP2, NumPy, and PETSc DMPlex. PyOP2 is a powerful library for automating mesh stencil calculations but weaknesses in its ability to describe mesh data limit its expressivity and hence certain mesh stencil algorithms cannot be described. NumPy and DMPlex both provide abstractions for describing different types of data: NumPy describing fully structured, hierarchical N-dimensional arrays and DMPlex describing fully unstructured mesh data.

In this thesis we seek a unifying abstraction for mesh data, combining elements of NumPy and DMPlex, which may then be implemented in a mesh stencil package reusing concepts from PyOP2. The new abstractions, called *axis trees* and *index trees* are discussed in chapters 4 and 5, and the new mesh stencil package, called **pyop3**, is introduced in chapter 6.

Chapter 4

Axis trees

As we have seen thus far, existing software abstractions for mesh-like data layouts are limited by their ability to flexibly describe complex layouts without discarding important topological information. We address this by introducing a new abstraction for describing data layouts: *axis trees*.

Axis trees are analogous to the metadata of a NumPy array, or a PyOP2 `DataSet`: they exist alongside a block of memory and provide the information necessary to interpret it. Axis trees are especially similar to NumPy arrays in that they both describe multi-dimensional data and accept multi-indices that are transformed into offsets. Indeed, single-component axes are equivalent to the axes of a NumPy array, hence the choice of name. The core difference to these existing abstractions that axis trees bring is the ability to describe more heterogeneous data layouts since axes can *branch*.

The fundamental objects making up an axis tree are as follows:

- An *axis tree* is a directed tree whose vertices are *axis* objects.
- An *axis* is an object carrying two attributes: an identifier called a *label*, and an ordered collection of *axis components*.
- An *axis component* is an object that carries a *label* as well as a *size* (usually an integer).

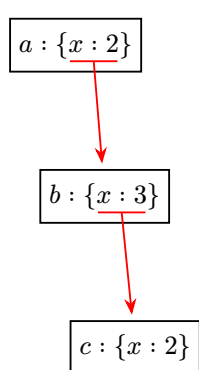
When constructing an axis tree there are a number of important constraints to consider:

```

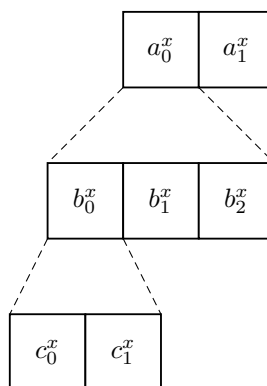
1 axis_a = Axis({"x": 2}, "a")
2 axis_b = Axis({"x": 3}, "b")
3 axis_c = Axis({"x": 2}, "c")
4 axes = AxisTree.from_iterable([axis_a, axis_b, axis_c])

```

(a) pyop3 code to build the axis tree.



(b) The axis tree.



(c) The data layout.

Figure 4.1: A linear axis tree with shape $(2, 3, 2)$. The axes are labelled a , b , and c and all have a single component, labelled x .

- Child axes are associated with a particular component of their parent axis. That is, each (non-root) axis has a *parent* that is the 2-tuple (*parent axis*, *parent component*).
- Along each possible *path* from root to leaf, axes in a tree are not allowed to have the same label.
- All of the labels of the axis components for an axis must be distinct.

Example 1: Linear axis trees

As a first example to demonstrate axis trees we consider the simplest case: *linear* axis trees. Linear axis trees are defined as being axis trees where all of the axes have a single component. They are equivalent to N-dimensional arrays in NumPy.

An example of a linear axis tree is shown in fig. 4.1. Consisting of 3 axes labelled a , b , and c , it is equivalent to a NumPy array with shape $(2, 3, 2)$ with axis a of the axis tree matching axis 0 of the NumPy array and so on. Each axis has a single component, all labelled x , with sizes 2, 3, and 2.

Some noteworthy features of this axis tree include:

- The axes have unique labels (a , b , and c). Labelling two of the axes a , say, would not be allowed.
- The axis components all have the same label (x). This is permitted because component labels need only be unique *per axis*.
- The parent-child relationships of the different axes are as follows:
 - Axis a is the root of the tree and so has no parent.
 - Axis b is the child of component x of axis a , hence its parent is the 2-tuple (a, x) , sometimes written a^x .
 - Axis c is the child of (b, x) .
- As the axis tree is linear, there is only one path from root to leaf: $\{a^x, b^x, c^x\}$.

Axis components only need labels to be distinguishable within a multi-component axis. For single-component axes this ambiguity is not present and so in future component labels will be omitted in this case for brevity.

Example 2: Multi-component axis trees

The natural extension to linear axis trees is to consider the case where axes are permitted to have more than one component. An example of such a multi-component axis tree is shown in fig. 4.2. Compared with the linear case, this axis tree has a few key differences:

- Axes b and c attach to *different components* of the shared axis a . The parent of b is (a, x) whilst the parent of c is (a, y) .
- Since axes b and c have a single component, the component label has been omitted.
- There are now two possible paths from root to leaf: $\{a^x, b\}$ and $\{a^y, c\}$.

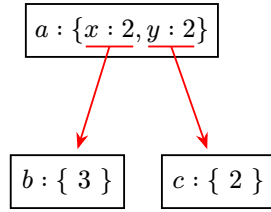
When materialised as a data layout (fig. 4.2c), the multiple components of axis a are stored apart from each other, with all entries of component x preceding those of component y .

```

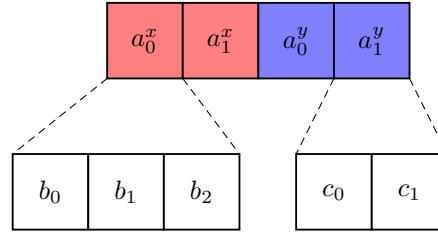
1 axis_a = Axis({"x": 2, "y": 2}, "a")
2 axis_b = Axis(3, "b")
3 axis_c = Axis(2, "c")
4 axes = AxisTree.from_nest({axis_a: [axis_b, axis_c]})

```

(a) pyop3 code to build the axis tree.



(b) The axis tree.



(c) The data layout. The distinct components of axis a are coloured red and blue for emphasis.

Figure 4.2: A simple multi-component axis tree.

4.1 Data structures

Axis trees are *data layout descriptors* providing a means to interpret blocks of memory. In the mesh stencil library **pyop3**, where axis trees have been implemented, actually storing data requires new objects coupling axis trees to data. Reusing terminology from **PyOP2**, **pyop3** has **Globals** (scalars), **Dats** (vectors), and **Mats** (matrices). **Globals**, being scalar-valued, need no axis trees, **Dats** require one, and **Mats** require two (one each for the rows and columns).

Being the most widely used, we will mostly focus on **Dats** during this thesis. **Mats** are more complicated and a discussion of their implementation is deferred to section 6.2.

4.2 Alternative data layouts

By relying on labels to identify axes in an axis tree, it is straightforward to express alternative layouts for the same data - a desirable characteristic for mesh stencil models (section 2.2.1) - simply by reordering the axes in the tree. For example, fig. 4.3 shows two equivalent axis trees with differently ordered axes. By leaving

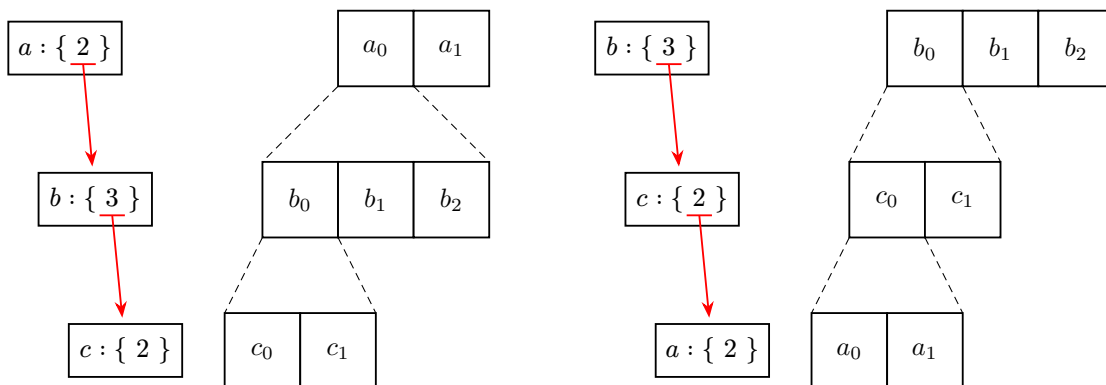


Figure 4.3: Two equivalent data layouts storing the same information. With the left axis tree, axis c is innermost, with unit stride, whilst for the right tree this is axis a .

the axes themselves unchanged, and only switching their order, the semantics of the layout are preserved even though the access patterns are changed.

In order to further facilitate this behaviour, `pyop3` is carefully designed to avoid making assumptions about the ordering of axes. For instance, paths, an essential component to many algorithms, are represented as unordered *sets*, rather than lists. For fig. 4.3, this means that the paths from root to leaf - $\{a, b, c\}$ and $\{b, c, a\}$ - are exactly the same. By avoiding assumptions about axis ordering, a user can create a data layout using any ordering they choose, and the rest of the code will continue to work without modification.

4.3 Ragged axis trees

Thus far we have only considered data layouts where the axis components have constant sizes. However, there are circumstances where the size is variable; for instance when representing a map from vertices in a mesh to their incident edges ($\text{supp}(v)$, section 3.2.1), as the number of incident edges is not constant for all vertices. Axis trees describing this sort of structure are described as *ragged*.

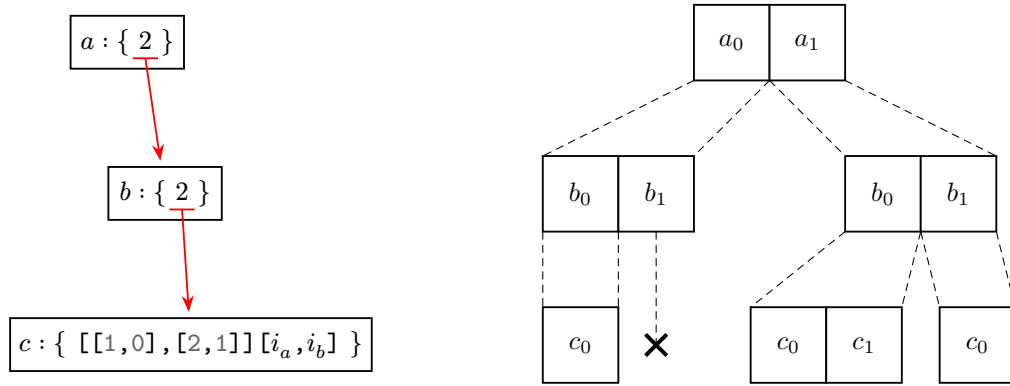
Ragged axis trees are created by passing a `Dat` as the size of the axis component, instead of an integer. This is shown in fig. 4.4. Here, the axis tree has 3 axes labelled a , b , and c . Axes a and b both have a constant size of 2 but c , the innermost axis, has a variable size given by a linear `Dat` with shape $(2, 2)$ and axis labels a and b (line 8). The values of this `Dat` can be written as $[[1, 0], [2, 1]] [i_a, i_b]$, meaning that the size of c is determined from the indices i_a and i_b as shown in table 4.1.

```

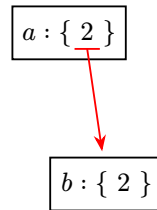
1  # make the size array
2  axis_a = Axis(2, "a")
3  axis_b = Axis(2, "b")
4  size_axes = AxisTree.from_iterable([axis_a, axis_b])
5  size_dat = Dat(size_axes, data=[1, 0, 2, 1])
6
7  # make the full axis tree
8  axis_c = Axis(size_dat, "c")
9  axes = AxisTree.from_iterable([axis_a, axis_b, axis_c])

```

(a) pyop3 code to build the axis tree.



(b) The axis tree (left) and resulting data layout (right). The cross in the data layout indicates that no values exist for the multi-index $\{a_0, b_1\}$.



(c) The axis tree for the size array $[[1,0], [2,1]] [i_a, i_b]$.

Figure 4.4: A ragged axis tree.

i_a	i_b	$\text{size}(i_a, i_b)$
0	0	1
0	1	0
1	0	2
1	1	1

Table 4.1: Lookup table illustrating how to interpret the size array $[[1, 0], [2, 1]] [i_a, i_b]$ from fig. 4.4.

4.4 Computing offsets

In order for axis trees to provide a complete specification of a data layout, they need to provide a mechanism that transforms multi-indices into offsets. To that end, axis trees carry additional *layout functions*, that take multi-indices and return offsets.

Layout functions are computed automatically from the specification of the axis tree. They are computed in two stages:

1. First, *partial layout functions* are determined for each axis component.
2. Then, the full layout functions, one per path through the axis tree, are determined by summing the partial layout functions.

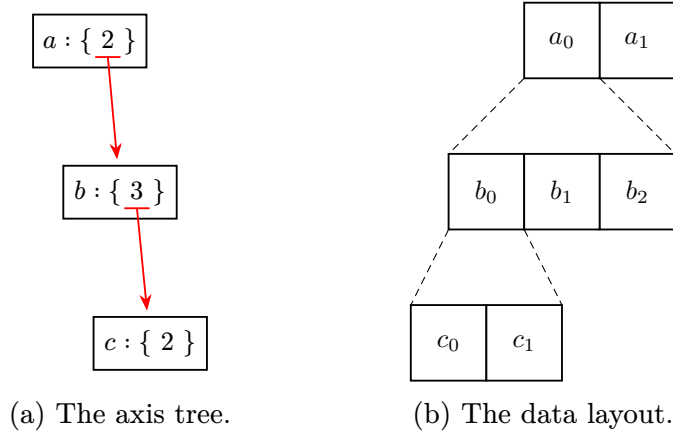
To explain how the layout functions are actually computed we present a series of algorithms, starting from the simplest case and iteratively adding features until the complete¹ algorithm is reached.

4.4.1 Intermediate algorithm 1: just linear axis trees

As an introduction to computing layout functions we return to the simplest possible case: linear (and non-ragged) axis trees. Figure 4.5 shows the axis tree, data layout and layout functions for a linear axis tree with shape $(2, 3, 2)$. Inspecting the layout functions we make the following observations:

- Partial layouts, expressed per axis, are always just the index multiplied by the step size. For example, the partial layout for axis b is $2i_b$ because the strides are over axis c , which has size 2.

¹There are additional considerations in parallel that are not considered here, see section 7.1.



Path	Partial layout	Full layout
$\{a\}$	$\text{offset}(i_a) = 6i_a$	$\text{offset}(i_a) = 6i_a$
$\{a, b\}$	$\text{offset}(i_b) = 2i_b$	$\text{offset}(i_a, i_b) = 6i_a + 2i_b$
$\{a, b, c\}$	$\text{offset}(i_c) = i_c$	$\text{offset}(i_a, i_b, i_c) = 6i_a + 2i_b + i_c$

(c) The layout functions.

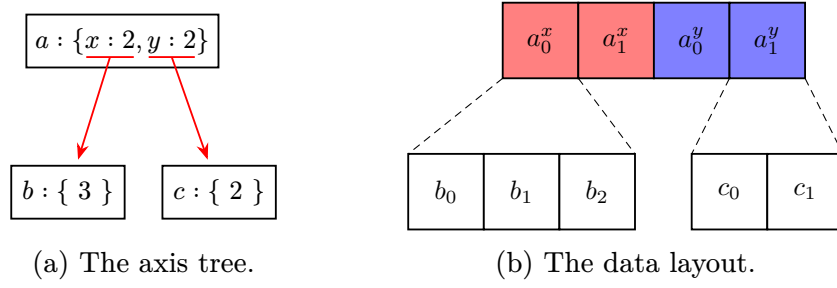
Figure 4.5: The layout functions for a linear axis tree.

- The innermost axis (here c) always has a trivial partial layout of just the index (here i_c) because it always has a step size of 1.
- Full layout functions are the sum of the partial layout functions as the tree is descended. For instance, the full layout function at path $\{a, b\}$ is the sum of the partial layouts $6i_a$ and $2i_b$.
- Multiple full layout functions exist, one for each partial descent of the axis tree. This means that it is still possible to compute an offset if only a partial multi-index is provided. For example, if provided the indices i_a and i_b , corresponding to the path $\{a, b\}$, it is legitimate to compute the offset as $6i_a + 2i_b$. However, if instead the indices i_b and i_c were provided, corresponding to the path $\{b, c\}$, then it is not possible to compute an offset because it is ambiguous whether (a_0, i_b, i_c) or (a_1, i_b, i_c) is meant.

Pseudocode for determining the right partial layout functions for such an axis tree is shown in algorithm 4. The function `collect_layouts_linear` is invoked recursively on each axis of the tree, starting with the root. For each axis the partial layout functions are collected into the `layouts` dictionary (line 2) using the current

Algorithm 4 Algorithm for computing the partial layout functions of a linear, non-ragged axis tree such as that shown in fig. 4.5. The function is initially invoked by passing the root axis of the tree.

```
1  def collect_layouts_linear(axis: Axis):
2      layouts = {}
3
4      # partial layout for the current axis
5      if has_subaxis(axis):
6          subaxis = get_subaxis(axis)
7          step = get_axis_size(subaxis)
8      else:
9          step = 1
10     layouts[axis] = AxisVar(axis.label) * step
11
12     # traverse subtree
13     if has_subaxis(axis):
14         subaxis = get_subaxis(axis)
15         sublayouts = collect_layouts_linear(subaxis)
16         layouts.update(sublayouts)
17
18     return layouts
```



Path	Partial layout	Full layout
$\{a^x\}$	$\text{offset}(i_a) = 3i_a$	$\text{offset}(i_a) = 3i_a$
$\{a^x, b\}$	$\text{offset}(i_b) = i_b$	$\text{offset}(i_a, i_b) = 3i_a + i_b$
$\{a^y\}$	$\text{offset}(i_a) = 2i_a + 6$	$\text{offset}(i_a) = 2i_a + 6$
$\{a^y, c\}$	$\text{offset}(i_c) = i_c$	$\text{offset}(i_a, i_c) = 2i_a + 6 + i_c$

(c) The layout functions.

Figure 4.6: The layout functions for a multi-component axis tree.

axis (`axis`) as the key. The partial layouts are symbolic expressions of the form `AxisVar(axis.label) * step` (line 10), where `step` is an integer and `AxisVar()` is a symbolic index object representing indices such as i_a , i_b , and i_c above.

4.4.2 Intermediate algorithm 2: with multi-component axis trees

When considering multi-component axis trees, a number of complications to the simple linear case are introduced. First, there is no longer a single partial layout function per axis, instead there is a partial layout function *per axis component*. Also, as well as weighted sums of indices, layout functions can now include *integer offsets*.

These modifications are demonstrated in fig. 4.6. Unlike before, axis a now has two components, x and y , with each component having a separate child axis. If we again inspect the computed layout functions we may make the following observations:

- The paths are now partitioned by component: a^x and a^y are treated separately.
- The fact that all the y component entries of axis a come after x is accounted for by the addition of an integer offset of 6, which is the size of a^x .

- As before, the full layout functions are constructed by adding together the partial layout functions. However, the fact that there are two distinct paths through the axis tree, $\{a^x, b\}$ and $\{a^y, c\}$, means that not all partial layouts are added together.

Algorithm 5 Algorithm for computing the partial layout functions of an axis tree where any of the axes may have multiple components. Some lines are highlighted in red to emphasise differences with algorithm 4.

```

1  def collect_layouts_multi_component(axis: Axis):
2      layouts = {}
3
4      # partial layouts for the current axis
5      start = 0
6      for component in axis.components:
7          if has_subaxis(axis, component):
8              subaxis = get_subaxis(axis, component)
9              step = get_axis_size(subaxis)
10         else:
11             step = 1
12
13         layouts[(axis, component)] = AxisVar(axis.label) * step + start
14         start += step
15
16     # traverse subtree
17     for component in axis.components:
18         if has_subaxis(axis, component):
19             subaxis = get_subaxis(axis, component)
20             sublayouts = collect_layouts_multi_component(subaxis)
21             layouts.update(sublayouts)
22
23     return layouts

```

To account for these changes some modifications must be made to algorithm 4. Shown in algorithm 5, the key differences are highlighted in red and include:

- The axis tree traversal now visits each component of each axis. This means that additional loops over `axis.components` are required (lines 6, 17).
- Since separate partial layouts are required for each axis component, the `layouts` dictionary now stores values per `(axis, component)` pair (line 13).

- The integer offset between partial layouts is tracked by the `start` variable (line 5), and it is added to the partial layout function (line 13).

4.4.3 Complete algorithm: with ragged axis trees

For ragged axis trees a new approach is needed. Since the strides between entries are no longer uniform it is no longer sufficient for layout functions to be expressed as a simple affine function, and instead the offsets must be tabulated into some lookup table in an analogous process to the approach taken by `PetscSections` (algorithm 2).

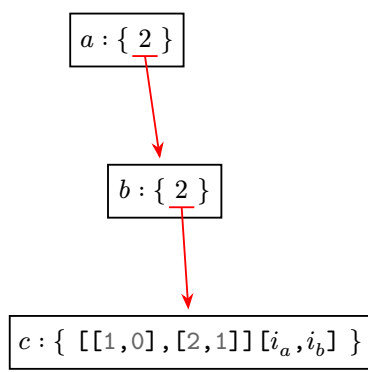
As an example, consider the axis tree shown in fig. 4.7. Though linear, the innermost axis (c) of the tree has a non-uniform size. Compared with previous, non-ragged axis trees, this has a number of implications when computing the layouts:

- Due to the non-uniform size of axis c , both axes a and b have non-uniform strides. The offsets must therefore be tabulated into *offset arrays*, which are implemented as `Dats` with linear axis trees.
- The axis trees for the offset arrays must, at a minimum, include the current axis. For example, axis a has partial layout function $[0, 1][i_a]$, meaning that it is a 1-dimensional array with an axis tree consisting of only axis a .
- Since the stride of axis b depends on the size of axis c , which in turn depends on the index i_a , the offset array for axis b depends on *both* i_a and i_b . By depending on axis a as well as itself, we say that axis a is a *dependent* axis for axis b .

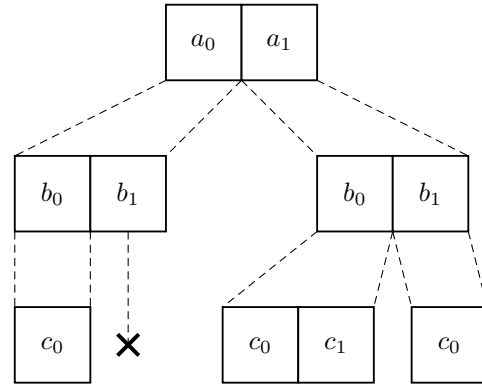
Note that at present it is not possible to have multi-component axes if one of the components is ragged.

To determine the partial layouts for ragged axis trees we must modify algorithm 5 to account for the case where axis component step size is non-uniform. The new version is shown in algorithm 6. In it, none of the code from the earlier algorithms is changed, but an additional branch is added to handle this new case (line 7). If the axis component's steps are found to be non-uniform then, instead of making an affine layout function involving `AxisVars`, the function `tabulate_offsets()` is called (line 12), returning an appropriate offset array to be used as the partial layout instead.

The behaviour of `tabulate_offsets()` is shown in algorithm 7. First, an empty `Dat` (`offset_dat`) is constructed to hold the tabulated offsets (line 6). Its axis tree (line 5) is linear and built from the current axis as well as any dependent



(a) The axis tree.



(b) The data layout.

Path	Partial layout	Full layout
$\{a\}$	$\text{offset}(i_a) = [0, 1] [i_a]$	$\text{offset}(i_a) = [0, 1] [i_a]$
$\{a, b\}$	$\text{offset}(i_a, i_b) =$ $[[0, 1], [0, 2]] [i_a, i_b]$	$\text{offset}(i_a, i_b) =$ $[0, 1] [i_a] + [[0, 1], [0, 2]] [i_a, i_b]$
$\{a, b, c\}$	$\text{offset}(i_c) = i_c$	$\text{offset}(i_a, i_b, i_c) =$ $[0, 1] [i_a] + [[0, 1], [0, 2]] [i_a, i_b] + i_c$

(c) The layout functions.

Figure 4.7: The layout functions for a ragged axis tree.

Algorithm 6 Algorithm for computing the layout functions of an axis tree where any of the contained axes may be ragged. Key differences with algorithm 5 are highlighted in red.

```
1  def collect_layouts_ragged(axis: Axis):
2      layouts = {}
3
4      # partial layouts for the current axis
5      start = 0
6      for component in axis.components:
7          if has_constant_step(axis, component):
8              # affine case, see algorithm 5
9              ...
10         else:
11             # axis component has a ragged subaxis
12             layouts[(axis, component)] = tabulate_offsets(...)
13
14     # traverse subtree
15     for component in axis.components:
16         if has_subaxis(axis, component):
17             subaxis = get_subaxis(axis, component)
18             sublayouts = collect_layouts_ragged(subaxis)
19             layouts.update(sublayouts)
20
21     return layouts
```

Algorithm 7 Algorithm tabulating the offsets for an axis with non-uniform strides.

```
1  def tabulate_offsets(axis: Axis, component: AxisComponent):
2      dependent_axes = collect_dependent_axes(axis, component)
3
4      # make the offset array
5      offset_axes = AxisTree.from_iterable([dependent_axes, axis])
6      offset_dat = Dat(offset_axes)
7
8      # loop over dependent axes
9      for dep_indices in dependent_axes.iter():
10         # reset the offset counter
11         offset = 0
12         # loop over the current axis
13         for current_index in axis.iter():
14             # store the current offset in offset_dat
15             offset_dat[*dep_indices, current_index] = offset
16             offset += step_size(axis, component)
17
18     return offset_dat
```

axes (line 2). Then, in a process very similar to what a `PetscSection` does during `PetscSectionSetUp()` (algorithm 2), both the dependent axes and current axis are looped over, incrementing and storing the offset in `offset_dat`.

The instructions `dependent_axes.iter()` (line 9) and `axis.iter()` (line 13) return iterators²that, when looped over, yield a sequence of multi-indices that are used to index into `offset_dat`.

4.5 Outlook

Axis trees are a novel abstraction for representing hierarchically structured, tree-like data layouts. They generalise NumPy N-dimensional arrays to ragged and multi-component settings and are capable of representing more structure than PETSc DMplex can using `PetscSections`.

²At present in `pyop3` these iterators are implemented in Python, making them very slow to execute for large axis trees. Accelerating this algorithm through either rewriting it in a faster language like C or using code generation is a high priority.

As we will show in later chapters, axis trees provide a natural means to describe the sorts of complex data layouts that occur in continuum mechanics calculations in a way that retains important structural information and facilitates the development of expressive interfaces.

Chapter 5

Indexing

Just like NumPy (section 3.1.2), data structures in `pyop3` can be indexed, extracting portions of the full structure and returning them as a new object. However, unlike NumPy where indexing is accomplished with a list of index-like objects, one for each axis, `pyop3` needs a more complicated solution to accommodate for the fact that data structures are now tree-like. Instead of lists of indices, `pyop3` instead uses *index trees*.

5.1 Index trees

In the same way that axis trees are built from labelled axis objects and axis components, index trees are built from labelled *index* objects and *index components*. Whereas axes and axis components encode levels in data layouts, indices in index trees encode *axis transformations* of the form

$$\text{axis} \circ \text{index} \rightarrow (\text{axis tree}, \text{index information}). \quad (5.1)$$

That is, each index takes an input axis and transforms it, returning a new axis tree alongside additional information encoding the relation between the old and new axes.

When dealing with index trees, as opposed to individual indices, this process expands to index entire axis trees, giving the transformation

$$\text{axis tree} \circ \text{index tree} \rightarrow (\text{axis tree}, \text{index information}). \quad (5.2)$$

In other words, to index an axis tree, an index tree must be applied to it, with the result being an *indexed axis tree* consisting of a new axis tree plus additional indexing information mapping the new axis tree back to the original one.

Algorithm 8 Algorithm that indexes an axis tree using an index tree. The function is initially called passing the root of the index tree.

```

1  def index_axes(index: Index, input_axis_tree: AxisTree):
2      # process the current index
3      axis_tree, index_info = index_handler(index, input_axis_tree)
4
5      # recursively visit child indices
6      for component in index.components:
7          if has_subindex(index, component):
8              subindex = get_subindex(index, component)
9              subaxis_tree, subindex_info = index_axes(subindex,
10                                                         input_axis_tree)
11              axis_tree.add_subtree(subaxis_tree)
12              index_info.update(subindex_info)
13
14  return axis_tree, index_info

```

Pseudocode for the indexing transformation that occurs when indexing an axis tree with an index tree (eq. (5.2)) is shown in algorithm 8. The index tree is traversed and the function `index_handler()`, representing the transformation in eq. (5.1), is called on each index (line 3), with the resulting axis tree and index information collected together and returned. Together these attributes are called an *indexed axis tree*.

Mapping between axis trees

The index information returned by the indexing transformation allows one to relate the old and new axis trees. In particular, it enables one to take multi-indices given for the new axis tree (the *source*) and map them to multi-indices for the old tree (the *target*). The index information consists of two mappings:

Target path A map connecting labels of the source tree with labels of the target tree.

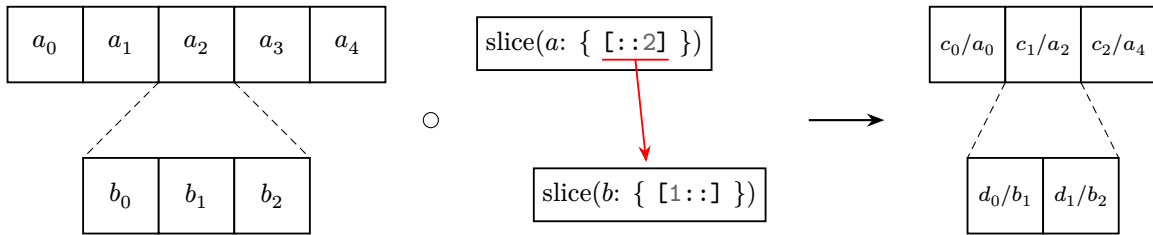
Target expressions A map connecting index expressions of the source tree with those of the target tree.

```

1  # set up the axis tree and array
2  axis_a = Axis(5, "a")
3  axis_b = Axis(3, "b")
4  axes = AxisTree.from_iterable([axis_a, axis_b])
5  dat = Dat(axes)
6
7  # index the array
8  indexed_dat = dat[:,2, 1::]

```

(a) pyop3 indexing code. The syntax used to index `dat` (`[:,2, 1::]`) is syntactic sugar that is expanded to a complete index tree. The syntax is identical to the equivalent NumPy code for an array with shape (5, 3).



(b) The axis tree transformation. The relation between old and new axis indices are indicated by the labels on the indexed axis tree (right). For example, the notation c_2/a_4 means that axis entry c_2 maps to a_4 in the original axis tree.

Figure 5.1: 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 .

With these mapping, one has sufficient information to: (a) select the right layout function from the target axis tree, and (b) substitute index expressions into the layout function, resulting in a *substituted layout function* that accepts multi-indices of the source axis tree instead of the target. By taking this symbolic approach to indexing, `pyop3` can represent all supported indexing operations as views, regardless of the indexing operation applied. This contrasts with NumPy where only ‘simple’ indexing operations produce views, with more complex operations like integer array indexing incurring a copy (section 3.1.2).

Example 1: Linear index trees

To illustrate axis tree indexing with a simple example, we index a linear axis tree, consisting of two axes labelled a and b , using two slices, where slices have the same semantics as they do in NumPy (section 3.1.2). The resulting axis tree is smaller

than the original and has new labels: c and d . Code for the transformation is shown in fig. 5.1a, and a diagrammatic representation is shown in fig. 5.1b.

As well as creating a new axis tree, indexing the array produces the following index information:

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

The target path encodes the fact that the path through the source tree $\{c, d\}$ is equivalent to the path through the target tree $\{a, b\}$ and the target expressions capture the semantics of the slices. For example, indexing axis a with the slice $[: : 2]$ means that entries in the new axis c map to even entries in a - which is exactly the index expression $i_a = 2i_c$.

Given this information, we may now construct a *substituted layout function* for addressing the original array using multi-indices of the indexed axis tree. First, the target path tells us which layout function to use from the original array, which in this case is

$$\text{offset}(i_a, i_b) = 3i_a + i_b. \quad (5.3)$$

Then, applying the substitutions from the target expressions, this produces the substituted layout function

$$\text{offset}(i_c, i_d) = 6i_c + i_d + 1, \quad (5.4)$$

which maps multi-indices in the source axis tree to offsets in the original array as desired.

Example 2: Multi-component index trees

As a further example of indexing transformations we consider a multi-component index tree applied to a multi-component axis tree. Shown in fig. 5.2, there are a number of changes compared with the linear case:

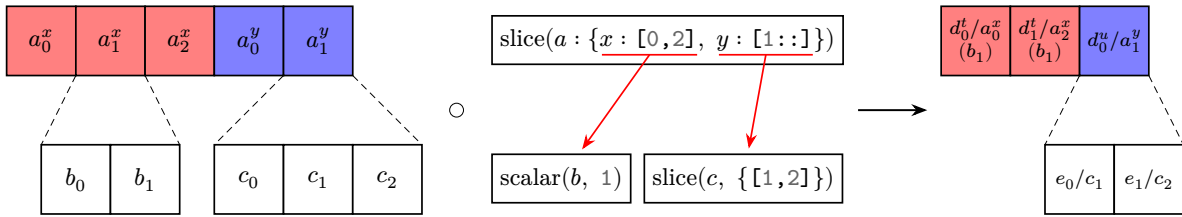
- The slice over axis a now has two components, one slicing the x axis component and one the y component. This means that the indexed axis tree (fig. 5.2b, right) has two components also.

```

1  # set up the axis tree and array
2  axis_a = Axis({"x": 3, "y": 2}, "a")
3  axis_b = Axis(2, "b")
4  axis_c = Axis(3, "c")
5  axes = AxisTree.from_nest({axis_a: [axis_b, axis_c]})
6  dat = Dat(axes)
7
8  # create the index tree
9  index_a = Slice("a", [Subset("x", [0, 2]),
10                      AffineSliceComponent("y", start=1)])
11  index_b = ScalarIndex("b", 1)
12  index_c = Slice("c", [Subset([1, 2])])
13  index_tree = IndexTree.from_nest({index_a: [index_b, index_c]})
14
15  # index the array
16  indexed_dat = dat[index_tree]

```

(a) pyop3 indexing code.



(b) The axis tree transformation.

Figure 5.2: The axis tree transformation resulting from indexing a multi-component axis tree with a multi-component index tree.

- Some of the slices now include *subsets*. Instead of providing **start**, **stop**, and **step** arguments an integer array is provided. This is equivalent to ‘integer array indexing’ for NumPy (section 3.1.2)¹.
- Instead of a slice, axis b is indexed with a *scalar index*. This accesses just one of the entries in b and so the index transformation *consumes* the axis and it does not appear in the indexed axis tree.
- Unlike the linear case, the multi-component nature of the index tree means that it is no longer possible to hide its instantiation behind syntactic sugar like in section 5.1 and it must instead be explicitly constructed (fig. 5.2a, line 13).

Indexing the array produces the following target path and target expressions:

Target path	Target expressions
$\{d^t\} \rightarrow \{a^x, b\}$	$\{i_a = [0, 2][i_d], i_b = 1\}$
$\{d^u, e\} \rightarrow \{a^y, c\}$	$\{i_a = i_d + 1, i_c = [1, 2][i_e]\}$

Unlike the linear case above, having subsets introduces index expressions like $[0, 2][i_d]$, and the scalar index of axis b consumes an axis, meaning that i_b is simply a constant instead of an expression of some other axis index.

With these target paths and expressions it is now straightforward to again determine appropriate substituted layouts for the indexed axis tree using the same process as before:

Path	Original layout	Substituted layout
$\{a^x, b\}$	$\text{offset}(i_a, i_b) = 2i_a + i_b$	$\text{offset}(i_d) = 2([0, 2][i_d]) + 1$
$\{a^y, c\}$	$\text{offset}(i_a, i_c) = 3i_a + i_c + 6$	$\text{offset}(i_d, i_e) = 3(i_d + 1) + [1, 2][i_e] + 6$

5.1.1 Index composition

Since indexing an axis tree just gives back another axis tree, it is possible to repeat the process and index the axis tree again. This is called *index composition*. An example composition operation is shown in fig. 5.3. Using the indexed array from fig. 5.1 we index it again to produce another indexed axis tree that remains capable of mapping multi-indices back to the original target axis tree.

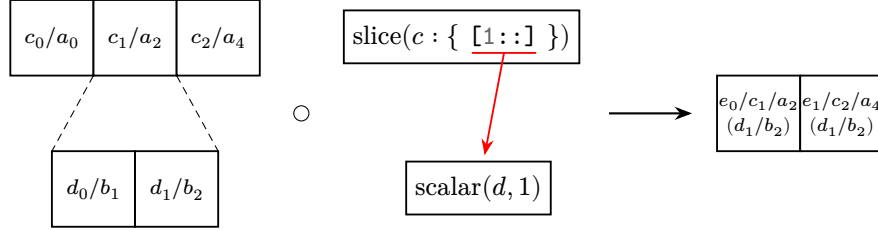
¹A possible source of confusion between `pyop3` and NumPy slices is that in `pyop3` slices refer generically to the operation of indexing a subset of an axis, either affinely (equivalent to a regular NumPy slice) or using a lookup table (equivalent to NumPy’s integer array indexing).

```

1  # use indexed_dat from fig. 5.1
2  indexed_dat = ...
3
4  # index the array again
5  indexed_dat2 = indexed_dat[1::, 1]

```

(a) pyop3 indexing code.



(b) The axis tree transformation.

Figure 5.3: The composition of an already indexed axis tree (from fig. 5.1) with another index tree.

If we disregard any prior indexing information and simply treat the transformation as we would a ‘normal’ indexing operation, treating the input axis tree as an unindexed axis tree with labels c and d , we get the following index information:

Target path	Target expressions
$\{e\} \rightarrow \{c, d\}$	$\{i_c = i_e + 1, i_d = 1\}$

From section 5.1 we know that the index information for the input (indexed) axis tree is given by:

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

Which may now be *composed* with the new indexing information to produce the new set of index information:

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

We can now perform the same sort of layout substitution as before:

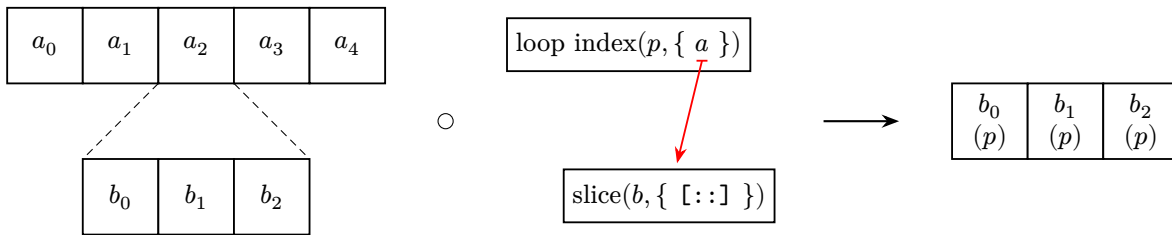
Path	Original layout	Substituted layout
$\{a, b\}$	$\text{offset}(i_a, i_b) = 3i_a + i_b$	$\text{offset}(i_e) = 6(i_e + 1) + 2$

```

1  # set up the axis tree and array
2  axis_a = Axis(5, "a")
3  axis_b = Axis(3, "b")
4  axes = AxisTree.from_iterable([axis_a, axis_b])
5  dat = Dat(axes)
6
7  # create the "outer" loop index
8  p = axis_a.index()
9
10 # index the array
11 indexed_dat = dat[p, :]

```

(a) pyop3 indexing code.



(b) The axis tree transformation.

```

1  # create an empty array with shape (5, 3)
2  array = numpy.empty((5, 3))
3  for p in range(5):           # the "outer" loop
4      array[p, :]              # index one axis of the array with p

```

(c) NumPy code performing an equivalent operation.

Figure 5.4: The axis tree transformation resulting from indexing with a loop index.

Giving us a way to map from the doubly-indexed axis tree back to the original.

5.2 Outer loops

For mesh stencil calculations it is often the case that one has ‘outer loops’ over some set of mesh entities, accessing global data relative to the current iterate. `pyop3` has full support for expressing this sort of iteration.

In `pyop3`, outer loops may be created over any axis tree via the instantiation of a *loop index*, constructed by calling the `index()` method. Indexing axis trees with

loop indices is similar to using scalar indices because the loop index consumes an axis. However, they differ in the fact that a scalar index sets the index expression to a scalar value, whereas loop indices pin the index to the value of the outer loop iterate.

An example demonstrating loop index usage is shown in fig. 5.4. In it one indexes a **Dat** (**dat**) with shape (5, 3) and labels a and b using a loop index over the axis a (line 8), and a full slice over axis b . NumPy code for an equivalent operation is shown in fig. 5.4c. Note that this example only demonstrates the way in which data are indexed using loop indices. A complete example demonstrating how to perform computations with them is deferred until section 6.1.2.

In order to map between source and target axes we follow the same process as before. This time, the index information is:

Target path	Target expressions
$\{b\} \rightarrow \{a, b\}$	$\{i_a = L_a^p, i_b = i_b\}$

Where the index expression L_a^p has been introduced to represent loop index p 's loop over axis a . Note that axis b has not been relabelled in the indexed array as a full slice was taken. Determining a substituted layout for the indexed array is then straightforward:

Path	Original layout	Substituted layout
$\{a, b\}$	$\text{offset}(i_a, i_b) = 3i_a + i_b$	$\text{offset}(L_a^p, i_b) = 3L_a^p + i_b$

It is important to note that, unlike with scalar indices and slices, arrays that have been indexed with loop indices have loop index terms (e.g. L_a^p) in their substituted layouts. Practically this means that it is not possible to evaluate offsets - and thus interpret - the array without knowing the value of these terms. Since the terms are provided externally, we say that arrays indexed with loop indices are *context-sensitive*, as they only make sense within the context of an outer loop.

5.3 Maps

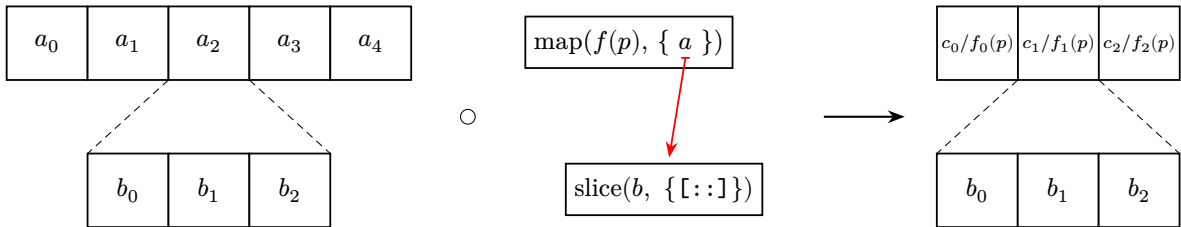
Indexing with loop indices is an important step towards generating expressions for stencil codes. However, they are not sufficient to produce ‘packed’ temporaries of the sort shown in fig. 2.4 - for that we need *maps*.

```

1  # set up the axis tree and array
2  axis_a = Axis(5, "a")
3  axis_b = Axis(3, "b")
4  axes = AxisTree.from_iterable([axis_a, axis_b])
5  dat = Dat(axes)
6
7  # create the "outer" loop
8  axis_x = Axis(10, "x")
9  p = axis_x.index()
10
11 # prepare a mapping from axis "x" to "a"
12 f = Map({"x": [MapComponent("a", arity=3, ...)]})
13
14 # index axis "a" with the map and take a full slice of axis "b"
15 indexed_dat = dat[f(p), :]

```

(a) pyop3 indexing code.



(b) The axis tree transformation.

Figure 5.5: Index transformation representing the packing of an axis tree using the map f , which has arity 3.

In `pyop3`, maps are functions that accept a single loop index as an argument (or another map, section 5.3.2) and yield values that are parametrised by the loop index. The number of values returned by the map for each iterate is called the *arity*; for example, the map from cells to incident edges (`cone(p)`, section 3.2.1) for a mesh with triangular cells has an arity of 3.

An example of indexing an array using a map is shown in fig. 5.5. We declare an outer loop over an axis labelled x (5.5a, line 8) as well as an arity 3 map, f , mapping from x to a (line 12). `dat` is then indexed with f , which has been *called* by passing it the loop index (i.e. $f(p)$). The resulting axis tree (5.5b, right) has lost axis a but gained a new axis in its place, labelled c , containing the 3 entries indicated in the map for a given p .

The treatment of the indexing information to retain view-like semantics remain exactly the same as before. For this example the returned index information is:

Target path	Target expressions
$\{c, b\} \rightarrow \{a, b\}$	$\{i_a = f(L_x^p, i_c), i_b = i_b\}$

Where $f(L_x^p, i_c)$ is used to represent the map. L_x^p is the outer loop index over axis x and i_c is an index for the new axis c . The final, substituted layout function can then be generated:

Path	Original layout	Substituted layout
$\{a, b\}$	$\text{offset}(i_a, i_b) = 3i_a + i_b$	$\text{offset}(L_x^p, i_b, i_c) = 3f(L_x^p, i_c) + i_b$

Note that we have not specified the form of the map function f . This is intentional because the function could either be a lookup with an indirection map, something like $f(L_x^p, i_c) := [\dots][L_x^p, i_c]$, or it could instead be some affine function of the form $f(L_x^p, i_c) := C \cdot L_x^p + i_c + D$ (for constants C and D). By treating maps more generally instead of eagerly committing to a particular representation `pyop3` should be able to work without change for both of these cases. This is useful in cases where both methods would ideally be supported, such as when dealing with both structured (affine function) and unstructured (indirection map) meshes, behind a single interface.

5.3.1 Ragged maps

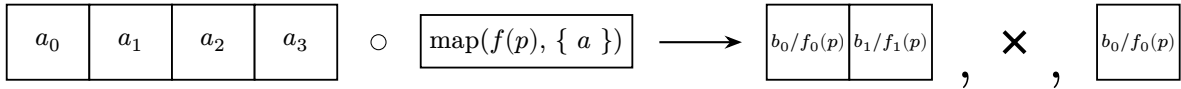
Thus far we have only considered the case where maps have a constant value arity. This is not always the case. To use the same example from section 4.3, the

```

1  # set up the axis tree and array
2  dat = Dat(Axis(4, "a"))
3
4  # create the "outer" loop and loop index
5  axis_x = Axis(3, "x")
6  p = axis_x.index()
7
8  # create a ragged map mapping from "x" to "a"
9  arity_dat = Dat(axis_x, data=[2, 0, 1])
10 f = Map({"x": [MapComponent("a", arity=arity_dat, ...)]})
11
12 # index the array
13 indexed_dat = dat[f(p)]

```

(a) pyop3 indexing code.



(b) The axis tree transformation.

Figure 5.6: Example indexing an array with a ragged map with arity $[2, 0, 1]$. The resulting array has variable size depending on the value of the loop index.

map relating vertices in a mesh to their supported edges has variable arity because the size of this relation is different for each vertex. As with axis trees, maps that have a variable size are also called *ragged*.

To represent ragged maps a simple change in the code is required: one must pass a `Dat` instead of an integer as the `arity` argument to the map. This is demonstrated in fig. 5.6a (line 10). As a result of this non-constant arity, the size of the indexed array is no longer constant and instead depends on the value of the loop index. This is shown in fig. 5.6b (right).

A discussion of layout functions and target paths/expressions is omitted here because it is identical to the example shown in fig. 5.5 above.

5.3.2 Map composition

As discussed in section 3.2.1, it is occasionally desirable to compose maps such that the output of one feeds directly into another. This is natural to express in `pyop3`.

As an example, consider the indexing operation shown in fig. 5.7. Axis a is indexed with the composed map $g(f(p))$ where p is a loop index over axis x , f maps x to y , and g maps y to a . As the arities of f and g are 3 and 2 respectively, two axes with sizes 3 and 2 (labelled c and d) are produced in the resulting indexed array (5.7b, right).

Indexing the array with the composed map gives the following index information:

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

Where the index corresponding to axis c (i_c) is passed as an argument to f , and the one for axis d (i_d) passed to g . With this, the substituted layouts are then:

Path	Original layout	Substituted layout
$\{a, b\}$	$\text{offset}(i_a, i_b) = 3i_a + i_b$	$\text{offset}(L_x^p, i_c, i_d, i_b) = 3g(f(L_x^p, i_c), i_d) + i_b$

5.4 Indexing with alternative data layouts

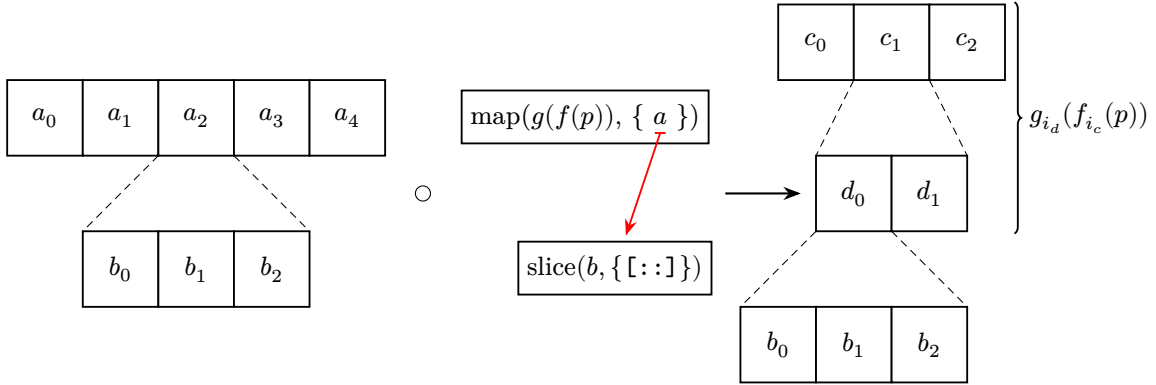
In section 4.2 we introduced the concept of representing alternative data layouts using axis trees and claimed that `pyop3` does not care about the specific ordering of axes. Whilst this is true internally to `pyop3`, more thought is required when interfacing

```

1  # set up the axis tree and array
2  axes = AxisTree.from_iterable([Axis(5, "a"), Axis(3, "b")])
3  dat = Dat(axes)
4
5  # create the "outer" loop and loop index
6  axis_x = Axis(10, "x")
7  p = axis_x.index()
8
9  # create map f, mapping "x" to "y"
10 f = Map({"x": [MapComponent("y", arity=3, ...)]})
11
12 # create map g, mapping "y" to "a"
13 g = Map({"y": [MapComponent("a", arity=2, ...)]})
14
15 # index the array
16 indexed_dat = dat[g(f(p)), :]

```

(a) pyop3 indexing code.



(b) The axis tree transformation. The composed map $g(f(p))$ produces the two-dimensional partial axis tree with labels c and d .

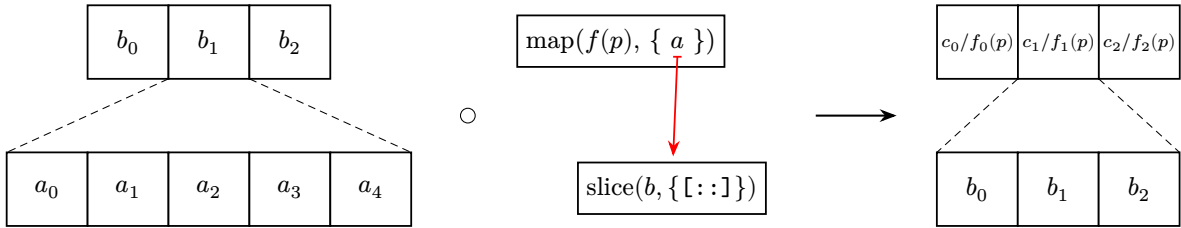
Figure 5.7: Indexing an axis tree using composed maps.

```

1  # set up the (flipped) axis tree and array
2  axes = AxisTree.from_iterable([Axis(3, "b"), Axis(5, "a")])
3  dat = Dat(axes)
4
5  # create the "outer" loop index
6  axis_x = Axis(10, "x")
7  p = axis_x.index()
8
9  # create a map mapping from axis "x" to "a"
10 f = Map({"x": [MapComponent("a", arity=3, ...)]})
11
12 # index the array
13 indexed_dat = dat[f(p), :]

```

(a) pyop3 indexing code.



(b) The axis tree transformation. Note that the ordering of the indexed axis tree (right) is the same as fig. 5.5b even though the input data layouts are different.

Figure 5.8: Indexing an array with a map. The transformation shown is identical to fig. 5.5 except the axes of the input array have been swapped.

with external code. In particular, the local kernels evaluated during mesh stencil calculations assume a specific ordering of the input and output DoFs. In other words, the global data structures do not have a prescribed layout, but the local ones do.

Conveniently, **pyop3**'s approach to indexing entirely obviates this issue. From algorithm 8 it can be seen that the structure of the new, indexed axis tree *comes entirely from the index tree, and not from the input axis tree*. This means that we can prescribe the local DoF ordering through careful construction of the index tree, independent of any global data layout.

To demonstrate this behaviour we consider an identical packing transformation to that shown in fig. 5.5, but with a different axis ordering for the unindexed array: swapping axes a and b . All other code, including the indexing code, is unchanged.

Keeping the indexing code the same produces the same axis tree and index information as fig. 5.5:

Target path	Target expressions
$\{c, b\} \rightarrow \{a, b\}$	$\{i_a = f(L_x^p, i_c), i_b = i_b\}$

But, since the layout functions for the unindexed object are different to before, the substituted layout is also necessarily different:

Path	Original layout	Substituted layout
$\{a, b\}$	$\text{offset}(i_a, i_b) = 5i_b + i_a$	$\text{offset}(L_x^p, i_b, i_c) = 5i_b + f(L_x^p, i_c)$

This demonstrates that even though the packed DoFs are exactly the same and in the same order, *they have been drawn from different places in the global array.*

5.5 Outlook

At this point we now have all of the necessary ingredients for axis trees to be used as part of a mesh stencil package. In particular, this chapter establishes: (a) the ability to symbolically represent views of a larger array, in an analogous manner to views of NumPy arrays, and (b) the ability to express ‘outer loops’ using loop indices and parametrised indirections with maps. These are both essential qualities for mesh data in a stencil computation: loop indices let us declare loops over mesh entities and axis tree views (i.e. indexed axis trees) with maps are appropriate for representing pack/unpack operations.

The abstractions presented are very flexible and enable a wide range of non-trivial transformations. In particular, it is possible to arbitrarily compose indexing operations without ever incurring a copy, compose maps such that the output of one feeds into another, and transparently support alternative layouts for a data structure.

In the following chapters we take the new abstractions of axis trees and index trees and provide an implementation of them in the new library `pyop3`.

Chapter 6

pyop3

The contributions of this thesis may be split into two parts: (a) the development of novel abstractions for mesh data (axis trees and index trees), and (b) the *implementation* of these abstractions as part of a working piece of software, `pyop3`. From here contribution (b) may again be split into the implementation of the *data model* - i.e. the implementation of the new abstractions - and that of the *execution model* - the part of the DSL that allows one to express and execute stencil algorithms. Whilst thus far we have dealt exclusively with (a) and the implementation of the data model, in this chapter we present information pertaining to the execution model of `pyop3`.

6.1 The execution model

In `pyop3`, the execution model refers to the elements of the language used to express actual computations. It has been designed to resemble the pseudocode for mesh stencil algorithms (e.g. algorithm 1) as closely as possible within the constraints of an embedded DSL. Its core components therefore are *loops*, *assignments*, and *function calls*. Using these components, users write symbolic *loop expressions* that describe the algorithm they wish to implement. Then, behind the scenes, runtime code generation is used to lower the symbolic expression down to executable code which is compiled and run.

This code generation pipeline is shown in fig. 6.1. Just like `PyOP2`, `pyop3` loop expressions are lowered first to `loopy` (section 3.3.3) and then to C, before calling a

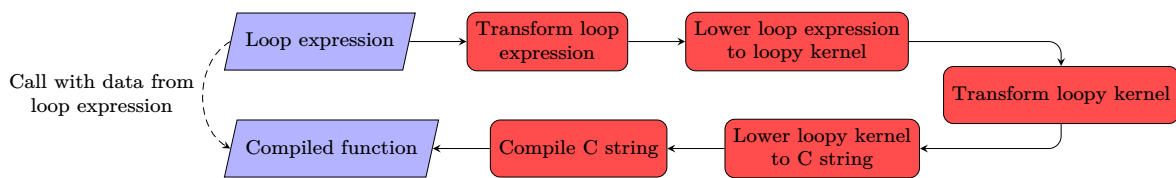


Figure 6.1: The code generation pipeline for the compilation of a loop expression into a callable function. The input (‘Loop expression’) and output (‘Compiled function’) are shown in blue whilst the intermediate processes are red. The dashed line from input expression to output function is included to represent the fact that the compiled function requires additional data from the loop expression in order to be executable.

traditional compiler to build an executable. To execute the computation, the compiled function is then called passing in appropriate data from the input loop expression.

Both `pyop3` loop expressions and loopy kernels are *transformable representations*, so, as the code is compiled, these layers are able to reason about the program and transform it to alternative but equivalent forms. This is typically done for optimisation purposes but `pyop3` also uses the technique more functionally to facilitate compilation. The former of these is discussed in section 6.3 whilst the latter is explained in the forthcoming examples.

6.1.1 Example 1: Constant assignment

As a first example of the code generation process in `pyop3`, we consider the following assignment operation:

```

axes = AxisTree.from_iterable([Axis(5), Axis(3)])
dat = Dat(axes)
dat[:, :2, 1:].assign(666)

```

Here, we assign the scalar value 666 to entries in the indexed linear array `dat[:, :2, 1:]`. Since a newly initialised `Dat` starts with a memory buffer of all zeros, following this operation the entries in `dat` will be

```

[0, 666, 666,
 0,   0,   0,
 0, 666, 666,
 0,   0,   0,
 0, 666, 666]

```

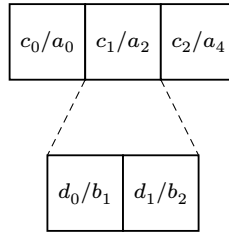
If we assume the axes of the indexed array to be labelled c and d , mapping from a and b respectively, then the indexed array has the following structure:

Listing 5 Abbreviated textual representation of the loopy kernel generated for the example expression in section 6.1.1.

```

1  -----
2  KERNEL: pyop3_kernel
3  -----
4  ARGUMENTS:
5  array_0: ArrayArg, type: np:dtype('float64'), in/out
6  -----
7  DOMAINS:
8  { [i_0] : 0 <= i_0 <= 2 }
9  { [i_1] : 0 <= i_1 <= 1 }
10 -----
11 INSTRUCTIONS:
12 for i_0, i_1
13     array_0[i_0*2*3 + i_1 + 1] = 666
14 end i_0, i_1
15 -----

```



which has

$$\text{offset}(i_c, i_d) = 6i_c + i_d + 1 \quad (6.1)$$

as the substituted layout function (chapter 5).

Code generation

When the `assign()` method is called, the following single-node loop expression is produced:

`write(dat[:, :2, 1:], 666)`

Where `write(x, y)` represents the operation $\mathbf{x} \leftarrow \mathbf{y}$. To generate code for this expression, the expression tree is descended and appropriate loopy constructs - domains, instructions, and arguments - are generated. These are then used to construct a complete loopy `LoopKernel`.

Listing 6 The C code generated from the loopy kernel in listing 5.

```
1 void pyop3_kernel(double *__restrict__ array_0)
2 {
3     for (int32_t i_1 = 0; i_1 <= 1; ++i_1)
4         for (int32_t i_0 = 0; i_0 <= 2; ++i_0)
5             array_0[1 + 6 * i_0 + i_1] = 666.0;
6 }
```

For this example, the loop expression has only a single node and so the traversal is trivial. To generate code from the `write` node we do the following:

1. First, since loopy expects scalar operations, loops are generated for the axes of the *indexed* array. In this case, two loops with extents 3 and 2 and index names, termed *inames*, "`i_0`" and "`i_1`" are created using the syntax

```
"{ [i_0]: 0 <= i_0 <= 2 }"
"{ [i_1]: 0 <= i_1 <= 1 }"
```

As well as emitting these loops, `pyop3` also stores the mapping between axis labels and inames: $\{c \rightarrow \text{"i_0"}, d \rightarrow \text{"i_1"}\}$.

2. Then, within these loops, `pyop3` emits an appropriate assignment instruction using the substituted layout function of `dat` and the axis label, iname mapping. In this case the layout function is given by $6i_c + i_d + 1$, and so the following instruction is generated:

```
dat[6*i_0 + i_1 + 1] = 666
```

The full loopy kernel generated for this expression is shown in listing 5, and the subsequently generated C code in listing 6. Note that `dat` has been passed through as an argument in the generated code but that it has been given a new name: `array_0`. This is because kernel arguments are renamed during code generation to avoid unnecessary code cache misses.

6.1.2 Example 2: Outer loops and maps

We now consider a more complex example that more closely approximates a mesh stencil calculation. Shown in listing 7, we loop over the entries in `axis_a` and apply local kernel `kernel` to indexed arguments `dat0` and `dat1`. `dat0` is a two-dimensional

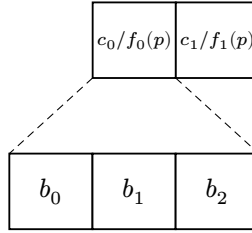
Listing 7 pyop3 code evaluating a local kernel (`kernel`) with intents `READ` and `INC` within a loop over the entries of axis a . The array `dat0` is accessed indirectly using an arity-2 map.

```

1  axis_a = Axis(5, "a")
2  axis_b = Axis(3, "b")
3  dat0 = Dat(AxisTree.from_iterable([axis_a, axis_b]))
4  dat1 = Dat(axis_a)
5  map0 = Map({"a": MapComponent("a", arity=2, ...)})
6
7  loop(
8      ia := axis_a.index(),
9      kernel(dat0[map0(ia), :], dat1[ia]),
10 )

```

array with shape (5, 3) and axis labels a and b , and `dat1` is one-dimensional with shape (5,), having only axis a in its axis tree. When passed to `kernel`, `dat0` is indexed with an arity 2 tabulated map and a slice, giving it an indexed axis tree like:



and substituted layout

$$\text{offset}(L_a^p, i_b, i_c) = 3[\dots][L_a^p, i_c] + i_b \quad (6.2)$$

where $[\dots][L_a^p, i_c]$ is the lookup expression for the tabulated map. `dat1` is fully indexed by the loop index and so contains just a single entry. It simply has the substituted layout

$$\text{offset}(L_a^p) = L_a^p. \quad (6.3)$$

Kernel intents

In listing 7, `kernel` is a `pyop3 Function` that wraps a loopy `LoopKernel`. Upon instantiation, as well as providing the loopy kernel, the user must also specify the *intents* of the various arguments that are expected by the kernel. In this case the kernel is declared as taking two arguments with intents `READ` and `INC`. These intents

Intent	Pack instruction	Unpack instruction
READ	<code>write(temporary, indexed)</code>	-
WRITE	<code>write(temporary, 0)</code>	<code>write(indexed, temporary)</code>
RW	<code>write(temporary, indexed)</code>	<code>write(indexed, temporary)</code>
INC	<code>write(temporary, 0)</code>	<code>inc(indexed, temporary)</code>

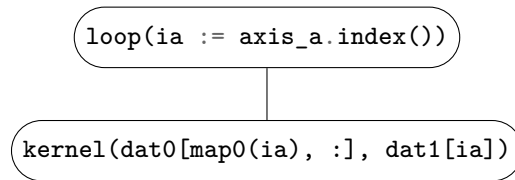
Table 6.1: Intent values supported by `pyop3` kernels and their corresponding pack/unpack instructions. In the instructions, the variable ‘`indexed`’ is used to represent the indexed view of some piece of global data (e.g. `dat0[map0(p)]`) and the variable ‘`temporary`’ is the temporary buffer for storing the materialised data. Table entries marked with a ‘-’ indicate that no pack/unpack instruction is emitted for this intent.

serve as descriptors encapsulating the manner in which the argument is accessed. Possible values are: `READ` (read-only), `WRITE` (write-only), `RW` (read and write), and `INC` (increment).

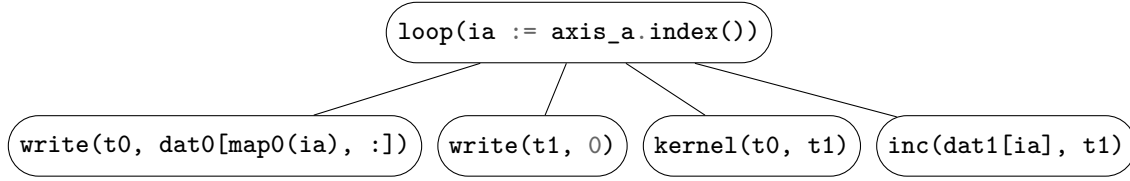
By providing intents, `pyop3` is able to optimise the generated code to reduce the amount of data that is streamed to/from memory by avoiding redundant reads/writes. For example, accessing global data in a read-only fashion (`READ`) means that values only need to be packed before invoking the kernel; unpacking the values back again is unnecessary.

The pack/unpack instructions required for each intent are shown in table 6.1. Only two distinct operations are required: `write` and `inc`. The semantics of `write` have already been explained, and `inc(x, y)` represents the similar operation $x \leftarrow x + y$.

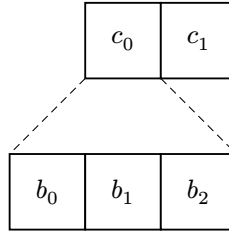
If we inspect the input loop expression from the listing, we see that these pack/unpack operations are implicit:



Since the local kernel expects dense local arrays to be passed to it, rather than symbolically indexed arrays, we *transform* the loop expression to make the pack/unpack operations explicit. This gives:



Where new temporary arrays `t0` and `t1` have been created to carry the packed data. `t0` and `t1` are the same shape as their indexed equivalents but are unindexed; they do not carry any indexing information and do not have substituted layouts. For `t0` this means that it has an axis tree identical to the indexed expression `dat0[map0(ia), :]`



But with the simpler layout function:

$$\text{offset}(i_c, i_b) = 3i_c + i_b. \quad (6.4)$$

Code generation

Having transformed the loop expression we are now able to lower it to a loopy kernel. As before, we traverse the loop expression and process each node in turn to collate the necessary loopy constructs:

- When we visit the `loop` node we generate the single loop over axis *a*:

```
"{ [i_0]: 0 <= i_0 < 5 }"
```

as well as the mapping $\{L_a^p \rightarrow \text{"i_0"}\}$, which is propagated through to other nodes in the loop expression tree.

- The `write` and `inc` nodes are handled in the same way as section 6.1.1. Inames for the outer loop indices (here L_a^p) are substituted into the layout functions in the same way as the inames originating from the axes themselves.
- The `kernel` node is trivial, simply emitting a call to the function and passing in the arguments `t0` and `t1`.

Listing 8 C code generated for the loop expression used in listing 7.

```
1 void pyop3_kernel(double const *__restrict__ array_0,
2                  int64_t const *__restrict__ array_1,
3                  double *__restrict__ array_2)
4 {
5     double t_0[6];
6     double t_1[1];
7
8     for (int32_t i_0 = 0; i_0 <= 4; ++i_0)
9     {
10         for (int32_t i_2 = 0; i_2 <= 2; ++i_2)
11             for (int32_t i_1 = 0; i_1 <= 1; ++i_1)
12                 t_0[i_1*3+i_2] = array_0[array_1[2*i_0+i_1]*3+i_2];
13         t_1[0] = 0.0;
14         kernel(&(t_0[0]), &(t_1[0]));
15         array_2[i_0] = array_2[i_0] + t_1[0];
16     }
17 }
```

For this example, the generated C code can be seen in listing 8. Observe that when reading `dat0` (line 12) the substituted layout from eq. (6.2) has been used for the right hand side, turning into the expression `array_1[2 * i_0 + i_1] * 3 + i_2`, whereas the assignee `t0` uses the unindexed layout from eq. (6.4) which turns into `i_1 * 3 + i_2`. Since a tabulated map is required as part of the loop expression, the generated code expects an integer array (`array_1`) in addition to the input and output arrays for `dat0` and `dat1` (`array_0` and `array_2`).

6.2 Matrices

`pyop3` uses `Mat` objects to represent matrices. Unlike `Globals`, with no axis trees, and `Dats`, with one, `Mats` are constructed with two axis trees, one for the rows of the matrix and one for the columns. As an example, to construct a `Mat` coupling a vector-valued space to a scalar one one might run:

```
vector_axes = AxisTree.from_iterable([Axis(5, "a"),
                                     Axis(3, "b")])

scalar_axes = AxisTree(Axis(5, "a"))
mat = Mat(vector_axes, scalar_axes)
```

Also unlike `Dats` and `Globals`, `Mats` do not use NumPy arrays internally to store data but instead use PETSc matrices (that PETSc also call `Mats`). Unlike vectors, which are almost always represented as some variant of a flat array, matrices can use a wide range of different storage techniques to account for properties like sparsity, symmetry, block structure, distributed in memory, and more. PETSc has wide support for a variety of matrix types, and exposes them via a generic interface, making adding support for them to `pyop3` relatively straightforward. As an additional benefit, using PETSc matrices means that `pyop3` is compatible with PETSc's wide range of solvers and preconditioners.

6.2.1 Indexing

In order to index a `Mat` two index trees are needed, one for each of the axis trees. These compose together in the normal way to give two indexed axis trees, that are stored in a new `Mat` representing the indexed object:

```
row_index_tree = IndexTree(...)
column_index_tree = IndexTree(...)
indexed_mat = mat[row_index_tree, column_index_tree]
```

6.2.2 Precomputing sparsity patterns

As we know from section 2.1, mesh stencil calculations produce *sparse* matrices, where the majority of entries are zeroes. To avoid redundant storage and computation, special sparse matrix formats exist that only store the non-zero values. For such formats to be performant, it is sometimes advantageous for the locations of the non-zeros - its *sparsity pattern* - be known in advance, because inserting new values would otherwise trigger expensive memory allocation.

To avoid this cost, `pyop3` can *preallocate* sparse matrices by precomputing the sparsity pattern and using this when building the `Mat`. The process of doing this is as follows:

1. First, one creates a `Sparsity` object, which is simply a `pyop3 Mat` where the underlying PETSc matrix has type `MATPREALLOCATOR`:

```
sparsity = Sparsity(row_axis_tree, column_axis_tree)
```

Preallocator matrices in PETSc are not capable of storing any values, but they provide an efficient mechanism for tracking where entries are inserted.

Listing 9 Example loop expression involving a `Mat`.

```
1  # build the matrix, passing in 2 axis trees
2  axis_a, axis_b = Axis(5, "a"), Axis(3, "b")
3  axes = AxisTree.from_iterable([axis_a, axis_b])
4  mat = Mat(axes, axes)
5
6  # array to hold the result
7  axis_x = Axis(10, "x")
8  dat = Dat(axis_x)
9
10 # map from "x" to "a" with arity 3
11 f = Map({"x": [MapComponent("a", arity=3, ...)]})
12
13 # make the loop expression, the kernel arguments
14 # have intents READ and INC
15 loop(ix := axis_x.index(),
16       kernel(mat[(f(ix),:), (f(ix),:)], dat[ix]))
```

2. Then, any parallel loops that would insert values in the real matrix should be simulated, passing in `sparsity` instead. For example:

```
loop(p := iterset.index(), dummy_kernel(sparsity[f(p), f(p)]))
```

Note that the local kernel, here `dummy_kernel`, should only set placeholder values in `sparsity`. There is no reason to perform any computation because the inserted values are discarded.

3. Lastly, a ‘regular’ `Mat` can be constructed from the `sparsity`:

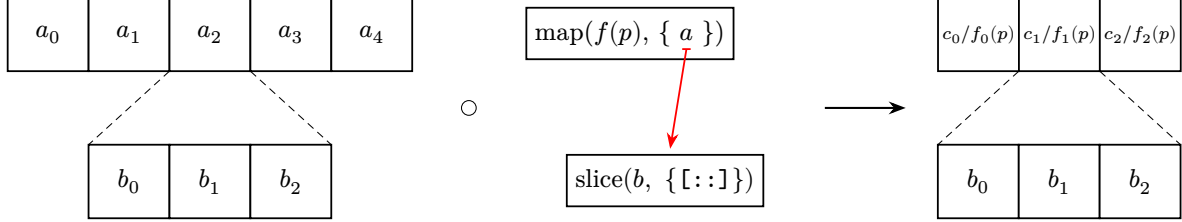
```
mat = Mat.from_sparsity(sparsity)
```

This matrix will use the `sparsity` pattern from `sparsity` so will not trigger any memory allocation when it is later used.

6.2.3 Code generation

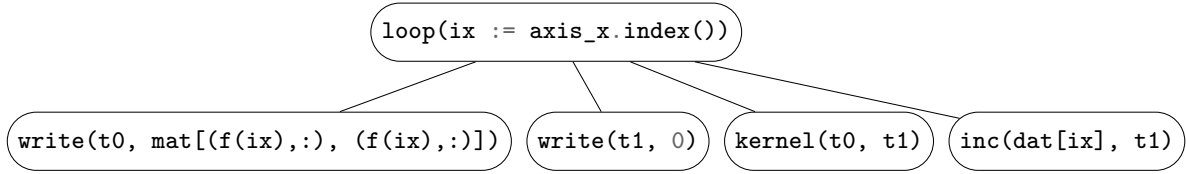
Whilst `Mats` may be treated the same as `Globals` and `Dats` in many circumstances, additional complications arise during code generation. As an illustrative example consider listing 9. In it, we execute a loop expression (line 15) that reads values from a 15×15 matrix (`mat`, line 4) and increments the result into a vector (`dat`, line 8).

The two axis trees of `mat` are identical and are indexed with the same map, slice combination such that the index transformations are:



The indexed matrix therefore has size 9×9 as both the row and column axis trees contain 9 entries.

If we begin to generate code for the loop expression, after expanding implicit pack/unpack instructions we have the loop expression:



Generating code from these nodes follows an identical process to that described in sections 6.1.1 and 6.1.2 with one exception: the matrix packing operation `write(t0, mat[(f(ix),:), (f(ix),:)]`. Since the data are stored in a PETSc `Mat` instead of a NumPy array, its values must be accessed using the PETSc C API, instead of directly addressing offsets in a memory buffer.

In order to read from and write to a PETSc `Mat`, the functions `MatGetValues()` and `MatSetValues()` must be used¹. They have near-identical function signatures, with that for `MatGetValues()` being:

```
MatGetValues(Mat mat,
             PetscInt m, PetscInt idxm[],
             PetscInt n, PetscInt idxn[],
             PetscScalar v[])
```

`mat` is the PETSc matrix object and `v[]` the dense array containing the loaded values². The arrays `idxm[]` and `idxn[]` specify the row and column indices whose values are to be read (with `m` and `n` their respective extents).

¹The scalar operations `MatGetValue()` and `MatSetValue()` do exist but they are much less efficient.

²Note that the data type of `v[]` is `PetscScalar`. Unlike other data structures in `pyop3`, matrices are only able to store values with whichever floating point type PETSc was configured with.

To generate code for the `write` node above we therefore need to emit a function call like:

```
MatGetValues(mat, 9, map0[Lxp*9], 9, map0[Lxp*9], t0)
```

Where `map0` is an array relating outer loop indices (i.e. L_x^p) to the appropriate 9 entries in `mat`. This is not as straightforward as it seems because `map0` *does not exist*. The relation between entries in the outer loop and entries in the `mat` is instead represented by the *symbolic* layout function

$$\text{offset}(L_x^p, i_c, i_b) = 3f(L_x^p, i_c) + i_b, \quad (6.5)$$

which cannot be passed as an argument to `MatGetValues()`.

To get around this, prior to generating code for matrix insertion, `pyop3` has to pre-tabulate the layout function to produce `map0`:

```
for Lp in range(...):
    for ic in range(...):
        for ib in range(...):
            map0[Lp, ic, ib] = 3f(Lxp, ic) + ib
```

which can then be used in the generated code.

6.3 Optimising index expressions

As demonstrated for the specific case of matrices above, by performing code generation at run-time `pyop3` is able to examine and manipulate the access patterns for the provided data structures. This constitutes an inspector-executor optimisation (section 2.2) similar to [22].

To give an example, consider a `Dat` indexed with the composed map $g(f(p))$ with p some outer loop and f and g maps tabulated with arrays `map_f` and `map_g` respectively. With these, the indexed `Dat` would have the substituted layout function

$$\text{offset}(L^p, i_f, i_g) = \text{map_g}[\text{map_f}[L^p, i_f], i_g]. \quad (6.6)$$

Which, if we further stipulate g to have arity 1, simplifies to

$$\text{offset}(L^p, i_f) = \text{map_g}[\text{map_f}[L^p, i_f]]. \quad (6.7)$$

Since these maps are available at compile-time, `pyop3` can pack the results into a new array, `map_fg`:

```
for  $L^p$  in range(...):
    for  $i_f$  in range(...):
        map_fg[ $L^p$ ,  $i_f$ ] = map_g[map_f[ $L^p$ ,  $i_f$ ]]
```

Giving us the new substituted layout

$$\text{offset}(L^p, i_f) = \text{map_fg}[L^p, i_f]. \quad (6.8)$$

Compared with the uncompressed version, using this new layout is advantageous for two reasons:

Better streaming memory accesses In the original expression, `map_g` is accessed indirectly, incurring performance slowdowns due to an increased rate of hardware cache misses. By combining the maps together we ensure that entries are read in a contiguous fashion, avoiding this problem.

Less memory is used Over the course of a calculation, all entries in `map_fg` will be streamed through memory. If the outer loop is said to have extent n and `map_fg` to have arity a then $n \cdot a$ entries will be loaded. Comparing this with the uncompressed expression we can immediately see that less memory is required. `map_f` has the same arity as `map_fg` and so the memory needed for the calculation is $n \cdot a$ *plus* however much memory is read from `map_g`.

Note that this optimisation does not need to be implemented in PyOP2 because the ‘compressed’ maps are provided at parallel loop construction. It therefore provides a way to reproduce existing PyOP2 performance.

6.4 Outlook

In this chapter we have presented details of how axis trees and index trees are used to generate code and perform computations as part of the mesh stencil library `pyop3`. `pyop3` has a similar approach to code generation to PyOP2 but it has a much more composable, and hence flexible, interface as a result of using these new abstractions.

The next chapters will discuss the changes necessary for `pyop3` to support distributed-memory parallelism, before applying `pyop3` to some real-world examples of mesh stencil calculations.

Chapter 7

Parallel

By now we have presented enough information about `pyop3` to be able to deploy it to compute mesh stencil calculations in a serial setting. To extend its functionality to support distributed-memory environments, a number of changes to the described infrastructure are required. Specifically, `pyop3` now needs to consider both how parallel data are arranged (section 7.1), and how ghost data are exchanged (sections 7.1.2 and 7.2).

Note that most of the following exposition is specific to parallel arrays (i.e. `Dats` and `Globals`). `Mats`, by using PETSc matrices as their underlying storage mechanism, are parallel by default and so require little intervention by `pyop3`.

7.1 Parallel data layouts

As described in sections 2.2.1 and 3.2.3, the usual way to distribute a stencil computation between multiple processors is to *partition* the mesh and distribute the DoFs of the different vectors such that DoFs associated with ‘owned’ mesh entities are stored on that process. Since mesh stencils typically require DoFs from adjacent entities in order to be computable, a small number of *ghost* DoFs must also be kept in the local array.

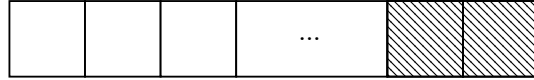
Whilst essential for the stencil computation, ghost points can be inconvenient for other operations occurring during a simulation. Using the finite element method as an example, having assembled the linear system, ghost DoFs need to be discarded so as not to appear in the global vector/matrix during solving. To avoid copying data between ghost-inclusive and ghost-free representations of an array, `pyop3` follows the

example of PyOP2 (section 3.3.2) and reorders the DoFs so that ghost entries are always stored after owned ones. This allows one to transform between the different representations without needing to copy data - the ghost-free version is simply a truncated slice of the full, ghost-inclusive array.

In order to represent distributed parallel data layouts using axis trees, axis components are passed an additional **StarForest** object - simply a thin wrapper around a PETSc star forest (section 3.2.3) - that encodes the point-to-point equivalence between processes:

```
star_forest = StarForest(...)
parallel_component = AxisComponent(size, sf=star_forest)
parallel_axis = Axis(parallel_component)
```

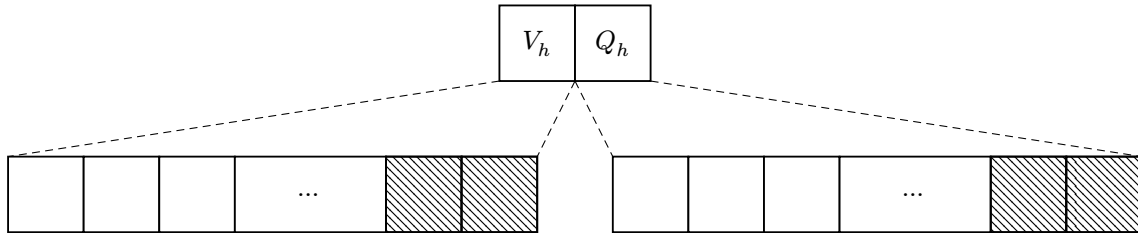
The ghost points are then stored at the end of the resulting data layout, giving a layout like the following:



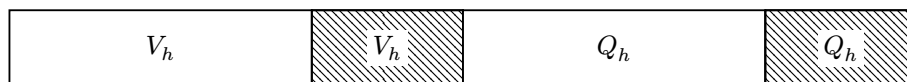
where ghost points are denoted by the hatching pattern.

7.1.1 Parallel layout functions

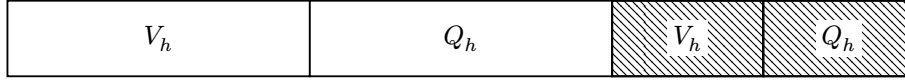
Whilst straightforward for linear axis trees, partitioning ghost points to the end of the array becomes more difficult in a multi-component setting. As an example, consider the ghost/non-ghost partitioning for the nested axis tree:



The natural concatenation of axis components we have seen until now results in an interleaving of owned and ghost points:



Whereas the desired layout should instead look like:



The need for the latter layout suggests some limitations in our existing layout tabulation algorithms (section 4.4). First, the layout functions for V_h and Q_h both now have discontinuous jumps in the location of their points and so can no longer be expressed as affine functions of the form $index \times step + start$, and must instead tabulate offsets into an array in a similar manner to the approach used for ragged axis trees. Further, tabulating layouts for the ghost portions of V_h and Q_h now requires that *the layouts for both their owned portions be tabulated first*. This is because the offset for the ghost section of V_h depends on the size of the owned section of Q_h , and vice versa for Q_h . This differs from the existing approach where different axis components may be visited independently.

To address these new conditions, we introduce the concept of *axis component regions*. Instead of giving axis components a single size attribute, we instead stipulate that they have a collection of $(label, size)$ 2-tuples representing the different classes of point. For the owned/ghost point classification this means that distributed axis components would represent their extents with the regions:

`[("owned", num_owned), ("ghost", num_ghost)]`

Algorithm 9 Algorithm for computing the layout functions of an axis tree with multiple regions.

```

1  def collect_layouts(axis_tree: AxisTree):
2      layouts = {}
3
4      # counter to track the 'shift' between owned and ghost points
5      offset = 0
6      # loop over owned and then ghost points
7      for region in axis_tree.regions:
8          # call the layout collection function from section 4.4, but pass
9          # in an extra offset counter and existing layout functions
10         size = collect_layouts_inner(axis_tree.root, layouts,
11                                     region, offset)
12         offset += size
13     return layouts

```

With the introduction of regions, we now modify the layout tabulation algorithms of section 4.4 to account for them. Shown in algorithm 9, the core differences are:

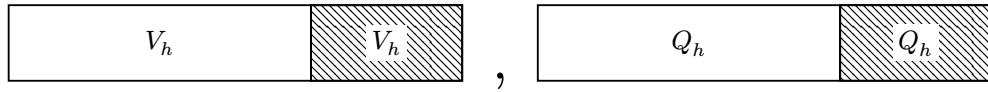
- There is now an additional outer loop over the different regions (line 7).
- To account for offset contributions from other axis components a further **offset** variable is used (line 5). It indicates the initial offset from which offset tabulation should start.

Axis regions are a generic concept and so can be applied to other settings where a DoF partitioning is desirable. One example of where this occurs is with Dirichlet boundary conditions, where values on the boundary are constrained to a particular value. Like ghost DoFs, it can be desirable to store these values in the array but still be able to discard them when solving the assembled system, and hence the constrained DoFs should also be collected towards the end of the array [70]. This can be expressed in `pyop3` by adding additional regions to the axis components:

```
[("owned", num_owned),
 ("constrained_owned", num_constrained_owned),
 ("ghost", num_ghost)
 ("constrained_ghost", num_constrained_ghost)]
```

Comparison with PyOP2

For the multi-field data layout example shown above, PyOP2 would not store the fields in a contiguous array, but instead store each field separately. This results in a data layout that looks like:



Constructing a monolithic global vector from this layout therefore requires copying the owned portions of each array into contiguous storage, which can be expensive on certain hardware. `pyop3`, by using regions to partition the ghost data to the end of the array regardless of the field structure, is able to avoid this copy.

7.1.2 DoF star forests

As described in section 3.2.3, in order to truly represent parallel data layouts the point-to-point star forests of the axis components must be transformed and combined

to form a single DoF-to-DoF star forest that maps the actual unknown values between processes for the distributed arrays. This DoF star forest is generated in two steps:

1. DoF star forests are generated for each parallel axis component from their point star forests and stride information.
2. These per-axis DoF star forests are then combined to form the final DoF star forest for the entire axis tree.

7.2 Communication optimisations

With the DoF star forest, `pyop3` `Dats` can now send data correctly between shared points. Since parallel communication is an expensive operation, we now present optimisations done in `pyop3` to minimise their impact whilst retaining data validity.

7.2.1 Minimising communication

In order to make sure that the data held by each `Dat` are valid, two attributes are stored:

`leaves_valid` A boolean value indicating whether or not ghost DoFs (leaves of the DoF star forest) are up-to-date. Updating them requires a broadcast from root to leaves.

`pending_reduction` A `pyop3` access descriptor indicating the appropriate reduction operation that is needed for owned DoFs (roots of the DoF star forest) to be valid. For instance, if ghost values were just incremented into then `pending_reduction` would be `INC`, meaning that a summation reduction is needed for root values to be valid. If `pending_reduction` is `None` then root values are considered up-to-date.

The reason that these attributes are tracked by `pyop3` is that they enable communication to occur lazily, meaning that under certain circumstances the exchanges may be skipped entirely. Examples include:

- If the array is being read from and all values are already up-to-date, no exchange is necessary.

- If ghost values are not being read then they need not be made valid.
- If the array is being written to, pending communications can be discarded since the data are being overwritten.
- If the array is being repeatedly incremented into, the summation reduction need not be executed between operations because the reductions commute.

This approach is similar to that of PyOP2 with the exception of `pending_reduction`, which is a novel contribution. Instead PyOP2 performs leaf-to-root reductions eagerly, which has been found to be a bottleneck in certain finite element simulations with multiple assembly loops executed in sequence.

7.2.2 Overlapping communication with computation

Algorithm 10 The `pyop3` parallel loop execution algorithm to interleave computation and communication.

1: REDUCELEAVESTOROOTSBEGIN	▷ Begin updating ROOT points
2: for <i>item</i> in <i>iter.set.core</i> do	▷ Compute CORE points
3: COMPUTE(<i>item</i>)	
4: end for	
5: REDUCELEAVESTOROOTSEND	▷ Finish updating ROOT points
6: BROADCASTROOTSTOLEAVESBEGIN	▷ Start updating LEAF points
7: for <i>item</i> in <i>iter.set.root</i> do	▷ Compute ROOT points
8: COMPUTE(<i>item</i>)	
9: end for	
10: BROADCASTROOTSTOLEAVESEND	▷ Finish updating LEAF points
11: for <i>item</i> in <i>iter.set.leaf</i> do	▷ Compute LEAF points
12: COMPUTE(<i>item</i>)	
13: end for	

Similarly to PyOP2, `pyop3` parallel loops overlap communication and computation to minimise the impact of network latency on performance. Since reductions are not done eagerly in `pyop3` the transfer algorithm (algorithm 10) includes an additional step: before one can update ghost values via a broadcast from roots to leaves, the owned values themselves must be updated with a reduction.

To perform this overlap we first partition the iteration set into 3 disjoint subsets:

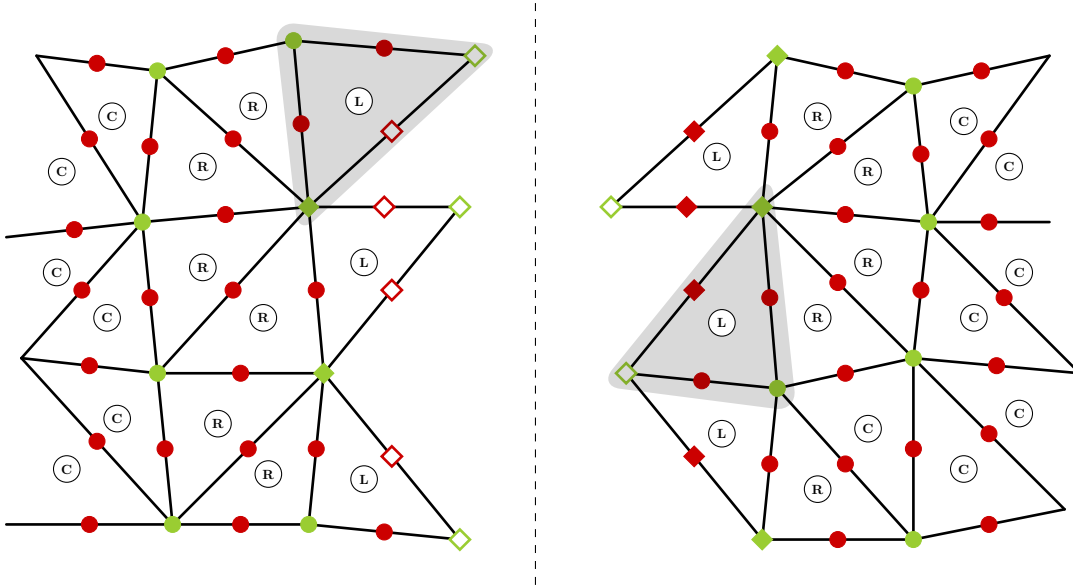


Figure 7.1: CORE (**C**), ROOT (**R**), and LEAF (**L**) labels for a cell-wise stencil computation with the stencil shown by the shaded region. Owned DoFs are filled in and ghost DoFs are hollow. Entries that exist in the DoF star forest (those that are duplicated across processes) are shown as diamonds.

CORE Entities that can be evaluated without waiting for any parallel exchanges to complete because all of the data necessary for the computation is already correctly stored locally.

ROOT Entities that use data from roots of the DoF star forest. Computing at these points therefore requires that the root data are (potentially) updated via a reduction.

LEAF Entities that access data from the leaves of the DoF star forest. Executing computations at these points requires that both reductions and broadcasts have been completed beforehand.

An example iteration set partitioning is shown in fig. 7.1, with CORE, ROOT, and LEAF points marked as **C**, **R**, and **L** respectively. It can be seen that points in the iteration set (the cells) are labelled as LEAF if their stencil includes a ghost point (hollow diamond), and ROOT if it includes a shared but owned point (filled diamond). All other points do not require any shared data and hence are marked CORE.

From the figure it is clear to see that the partitioning of the iteration set is a *stencil-dependent process*. Larger stencils would result in more points being classified as ROOT

and LEAF. `pyop3` therefore addresses the stated limitation of `PyOP2` (section 3.3.2) where the iteration set partition is decided at array initialisation, leading to incorrect partitioning for different stencil sizes and parallel decompositions.

Determining CORE, ROOT, and LEAF

Algorithm 11 Algorithm to partition an iteration set into CORE, ROOT, and LEAF.

Require: *iteration set*

Require: *arrays*

▷ Arrays used in the computation

Require: *labels*

▷ Label for each iterate, defaults to CORE

```

1: for point in iteration set do
2:   for array in arrays do                                ▷ Check the stencil of each array
3:     if labels[point] = LEAF then                             ▷ Skip if already LEAF
4:       continue
5:     end if
6:     for stencil point in COLLECTSTENCIL(array, point) do
7:       if stencil point in array.star forest then
8:         star forest label ← GETSTARFORESTLABEL(stencil point)
9:         if star forest label = LEAF then
10:          labels[point] ← LEAF
11:          break                                              ▷ Most restrictive case, stop further checks
12:        else
13:          labels[point] ← ROOT
14:        end if
15:      end if
16:    end for
17:  end for
18: end for

```

The algorithm to partition the iteration set into CORE, ROOT, and LEAF subsets is straightforward. Shown in algorithm 11, one simulates the stencil operation and, for each array, inspects the points touched at each iterate (line 6) to see whether they are a shared point (i.e. whether they are present in the DoF star forest). If so then the iterate is marked as ROOT or LEAF as appropriate. LEAF, as the most restrictive condition, takes precedence over ROOT. The algorithm is very similar to the approach taken by `PyOP2` except the loop over arrays (line 2) is omitted, and the function `COLLECTSTENCIL` is hard-coded as the point's closure (section 3.2.1), meaning that other stencil sizes are not supported.

7.3 Outlook

In this chapter we have presented the modifications necessary to extend the heretofore serial-only version of `pyop3` to work in a distributed memory parallel environment. For this to work, additional care must be taken to ensure that data are stored correctly on a single process, and that they are correctly mapped between processes. Further consideration is also given to minimise the impact of any communication that may take place.

Matrices were not discussed because their parallel implementation is entirely provided by PETSc.

Now having a parallel implementation, all of the fundamental features of `pyop3` have been established and we may now consider integrating it into existing PDE-solving frameworks. This is the subject of the next chapter.

Chapter 8

Firedrake integration

One of the main advantages to a framework like `pyop3` is that, by automating a process typically done by hand, it allows for higher level abstractions to be built on top of it. In this chapter we will demonstrate this by integrating `pyop3` into the finite element framework Firedrake [31].

8.1 Firedrake

Firedrake is a Python framework for automating the solution of PDEs using the finite element method. It uses code generation to convert near-mathematical expressions provided by the user into fast C code, with the result being a programming environment that is both productive to work in and performant. Firedrake is parallel by default; a script that works on someone’s laptop can run without modification on a cluster¹.

The framework consists of a number of interacting software components. Shown in fig. 8.1 these include, in addition to Firedrake: PyOP2, UFL (Unified Form Language) [3], TSFC (Two-Stage Form Compiler) [37], loopy, and PETSc. The contributions of these packages to the overall framework will be made clear below. A number of packages not relevant for `pyop3` have been omitted for simplicity.

The way in which these packages interact is best shown by way of an example. Listing 10 shows a typical Firedrake script for solving the Stokes equations, using the formulation from section 2.1. It consists of the following steps:

¹For performance reasons the user may wish to modify their choice of solver and/or discretisation. See, for example, [12].

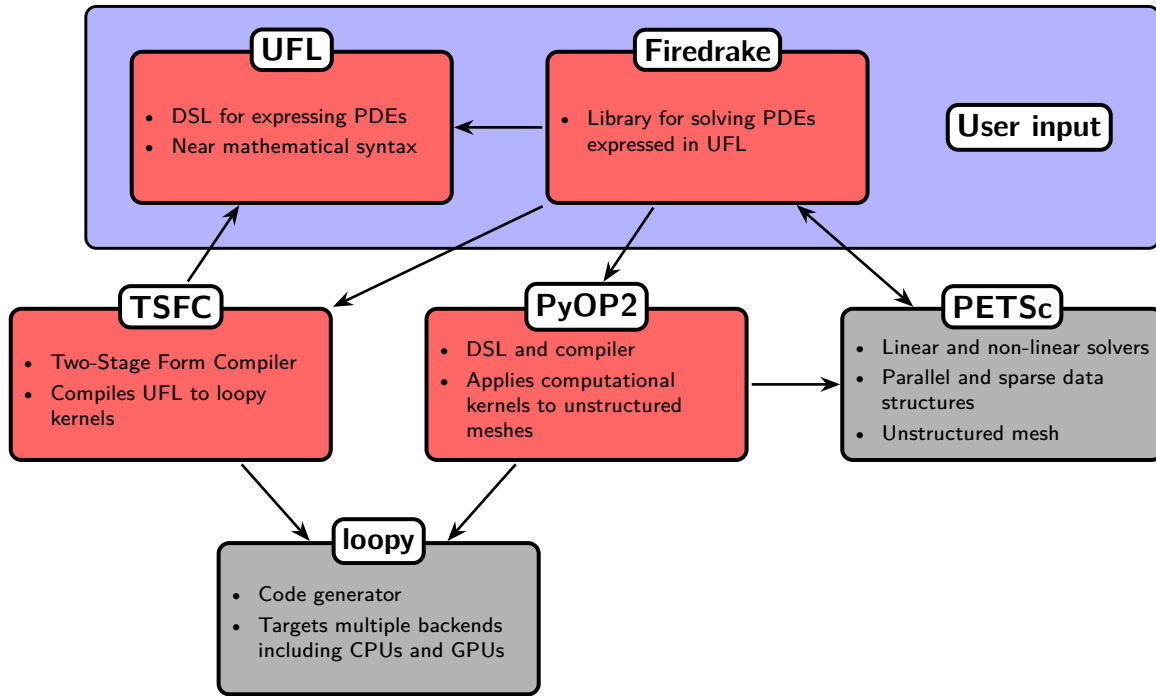


Figure 8.1: The core software components of the Firedrake finite element framework. Packages shown in red are managed by Firedrake whereas the grey packages are key external dependencies. Users typically express computations as Python scripts using both UFL and Firedrake; this is represented by the blue ‘User input’ box. The direction of the arrows indicate a caller-callee relationship. For instance Firedrake calls PyOP2 but not vice versa. The bidirectional arrow between Firedrake and PETSc indicates that they each call the other, which for PETSc is via callback functions.

Listing 10 Firedrake code for setting up and solving the Stokes problem from section 2.1.

```

1  from firedrake import *
2
3  # initialise the mesh and function spaces
4  mesh = UnitSquareMesh(10, 10)
5  V = VectorFunctionSpace(mesh, "P", 3)
6  Q = FunctionSpace(mesh, "DP", 2)
7  W = MixedFunctionSpace([V, Q])
8
9  u, p = TrialFunctions(W)
10 v, q = TestFunctions(W)
11
12 # viscosity
13 nu = Constant(666)
14
15 # define lhs and rhs
16 a = (nu * inner(grad(u), grad(v)) * dx - p * div(v) * dx
17      + q * div(u) * dx)
18 L = Cofunction(W.dual()) # zero rhs
19
20 # construct the boundary condition
21 g = Constant([666, 666])
22 bc = DirichletBC(W.sub(0), g, "on_boundary")
23
24 # assemble and solve the problem
25 solution = Function(W)
26 solve(a == L, solution, bcs=[bc])

```

1. The mesh (line 4) and function spaces (lines 5-7) are created. Internally the mesh is stored as a PETSc DMPlex (section 3.2), which is distributed in parallel between processes. The function spaces associate DoFs with topological entities of the mesh according to the provided finite element definition. In this case the finite elements are specified by the arguments **"P"** and 3 to **VectorFunctionSpace**, and **"DP"** and 2 to **FunctionSpace**. These correspond to the $[P_3]^2$ and P_2^{disc} elements that together form the Scott-Vogelius finite element pair.
2. Having done this, the linear (L, line 18) and bilinear (a, line 16) forms are then declared symbolically using the UFL DSL. The code used to construct both forms resemble the mathematics of the original weak form (eq. (2.6)), in particular the bilinear form

$$\begin{aligned} \int \nu \nabla u : \nabla v \, d\Omega - \int p \nabla \cdot v \, d\Omega &= 0 \quad \forall v \in V_0 \\ \int q \nabla \cdot u \, d\Omega &= 0 \quad \forall q \in Q. \end{aligned} \tag{8.1}$$

The linear form is trivially zero and so a pre-assembled cofunction is used.

3. Lastly, the PDE is solved with appropriate boundary conditions (line 26). To do so, the bilinear and linear forms are first assembled into a matrix and vector respectively. Then, having assembled the linear system, PETSc solves it with its wealth of different solver types, putting the solution for both the velocity and pressure together into the mixed function **solution**.

8.1.1 PyOP2's contribution

PyOP2 is a crucial component to this process. To start with, *all of Firedrake's global data structures wrap PyOP2 ones*. For instance **Functions** and **Cofunctions** wrap PyOP2 **Dats**, and assembled bilinear forms, represented by Firedrake **Matrix** objects, wrap PyOP2 **Mats**. The **DataSets** that act as data layout descriptors for the PyOP2 objects are determined from the function spaces.

Further, *PyOP2 is responsible for global assembly*. During assembly the UFL forms are lowered by TSFC into local kernels (i.e. stencil payloads) expressed as loopy **LoopKernel** objects. PyOP2 then handles the outer loop over mesh entities, evaluating the local kernel for each.

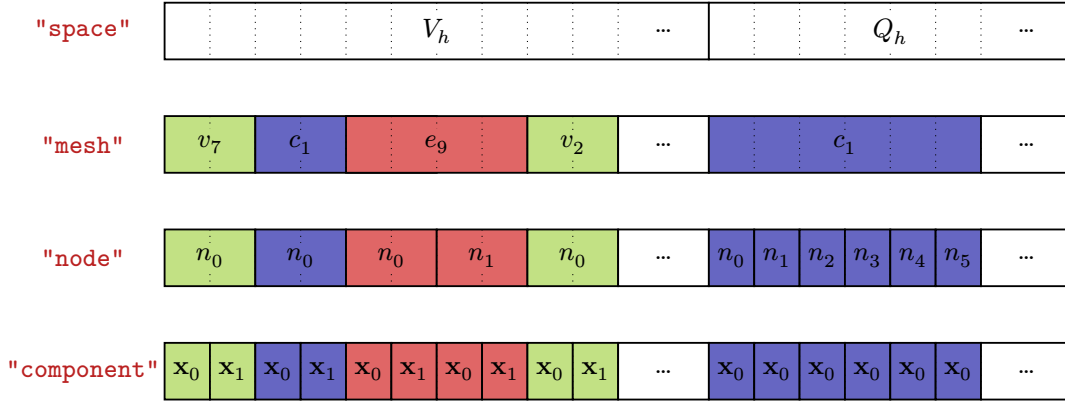


Figure 8.2: A possible global data layout for a degree 3 Scott-Vogelius function space.

To do so, Firedrake produces a PyOP2 parallel loop resembling:

```

par_loop(local_kernel,                                # payload
         mesh.cell_set,                               # iteration set
         dat0(READ, cell2nodes)                       # arguments
         mat0(INC, (cell2nodes, cell2nodes)))         # "
```

Notice that since **Dats** and **Mats** store data relative to nodes of the function spaces, as opposed to topological entities of the mesh, the indirection maps (**cell2nodes**) map from the iteration set to nodes of the function space.

In order to integrate **pyop3** with Firedrake and replace PyOP2, we must clearly replicate these two pieces of functionality, global data structures and executing assembly loops, with their **pyop3** equivalents. More precisely, PyOP2 data sets must be replaced with axis trees and PyOP2 parallel loops must be replaced by **pyop3** loop expressions.

8.2 Representing global data with axis trees

In many respects the fundamental behaviour of global data structures is unchanged between PyOP2 and **pyop3**. Both use NumPy arrays and PETSc matrices to store vectors and matrices as **Dats** and **Mats**, and **Dats** still use star forests to exchange ghost data. The core difference between the two libraries is in the addition of axis trees, which replace PyOP2 data sets.

Therefore, to store function space data in **pyop3** **Dats** and **Mats**, appropriate axis trees must first be constructed to prescribe the data layouts. For a Scott-Vogelius function space we would expect, having applied a mesh renumbering, to have a data

layout like that shown in fig. 8.2. The layout naturally decomposes into multiple separate axes: the function spaces ("**space**"), the points of the mesh ("**mesh**"), the per-entity nodes ("**node**"), and any vector nature of the nodes ("**component**").

8.2.1 Renumbering the mesh

Given this decomposition it would appear simple to construct an axis tree appropriate for describing this layout using the following for the "**mesh**" axis:

```
mesh_axis = Axis({"cell": ncells,
                 "edge": nedges,
                 "vertex": nvertices}, "mesh")
```

However, this does not work because axis components are stored contiguously. Constructing a "**mesh**" axis naively as above leads to a *non-interleaved* layout with all cell DoFs preceding all edge DoFs preceding all vertex DoFs. This clearly does not work as an approach for constructing *interleaved* layouts.

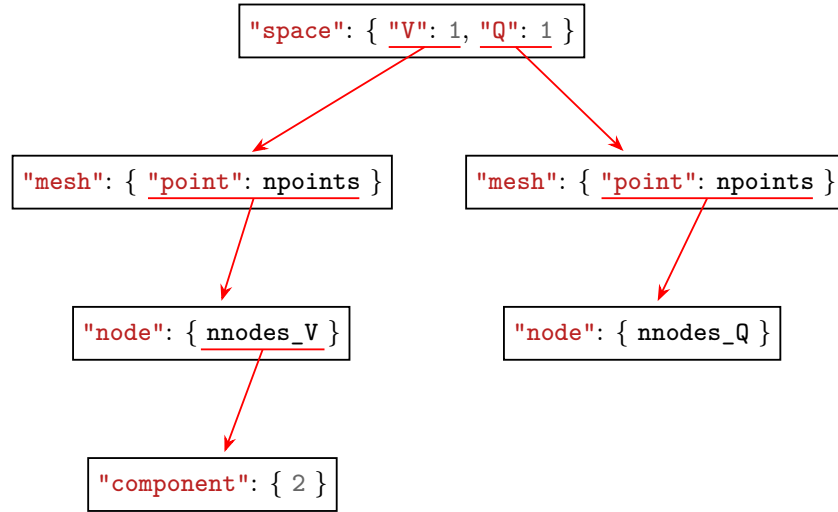
To resolve this we take a two step approach. First, instead of constructing a multi-component "**mesh**" axis from the start, we instead construct a single component axis with the component representing all topological entities (points) in the mesh:

```
mesh_axis = Axis({"point": npoints}, "mesh")
```

We can then construct an axis tree with the correct DoF ordering, albeit without being able to distinguish the different mesh entities. This is demonstrated in fig. 8.3. Observe that, since mesh entities are indistinguishable, the number of nodes per mesh point is no longer fixed and so we have a ragged axis tree.

"space"	V_h ...										Q_h ...									
"mesh"	p_0	p_1	p_2	p_3	...	p_0														...
"node"	n_0	n_0	n_0	n_1	n_0	...	n_0	n_1	n_2	n_3	n_4	n_5								
"component"	x_0	x_1	x_0	x_1	x_0	x_1	x_0	x_1	x_0	x_1	...	x_0	x_0	x_0	x_0	x_0	x_0			

(a) The data layout. Notice that the "mesh" axis, unlike in fig. 8.2, is unable to distinguish between different types of mesh entity, so the points are labelled $p_0, p_1, \dots$ instead of $v_7, c_1, \dots$. The layout is thus ragged since the number of nodes per point now must vary.



(b) The axis tree. The variables `nnodes_V` and `nnodes_Q` are `Dats`, indicating that the data layout is ragged.

Figure 8.3: The initial ragged data layout for a degree 3 Scott-Vogelius function space.

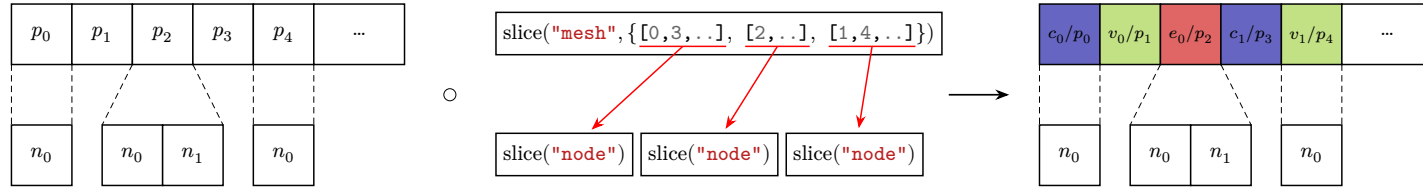


Figure 8.4: The indexing transformation taking a ragged axis tree for a function space to one where the different topological entities are distinguishable. The arrays $[0,3,\dots]$, $[2,\dots]$, and $[1,4,\dots]$ represent the indices of the cells, edges, and vertices respectively.

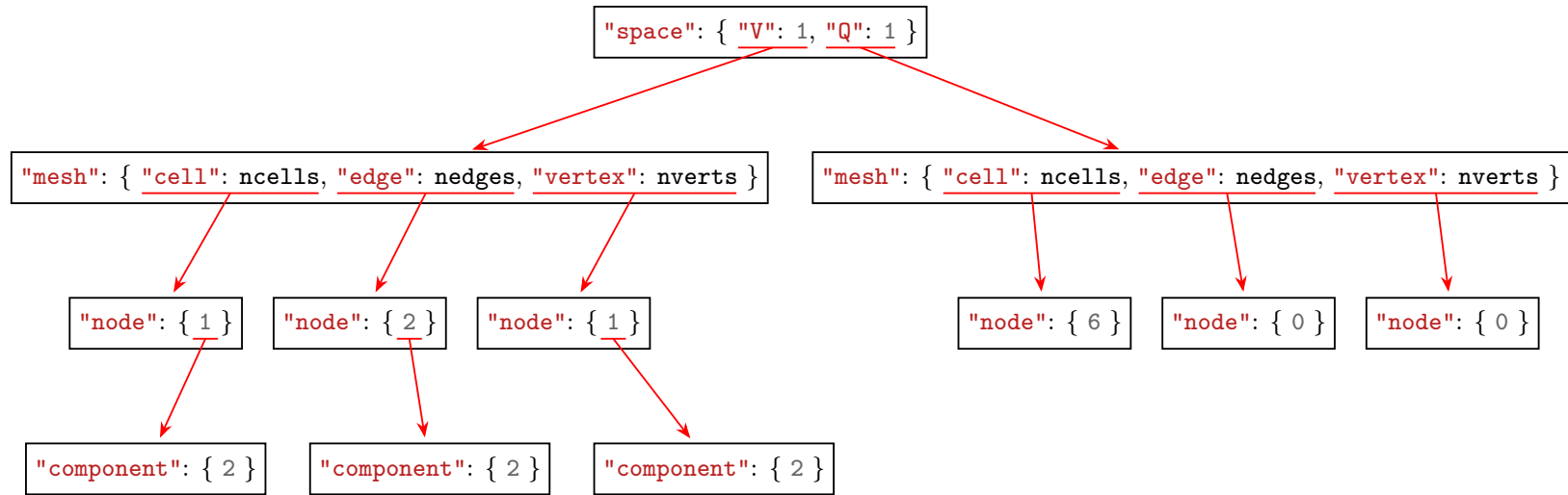


Figure 8.5: The indexed axis tree generated after indexing the ragged axis tree in fig. 8.3b. The mesh entities are distinguishable whilst remaining interleaved.

Given this axis tree, the correct entity information may be recovered via *indexing*. To do so we index the **"mesh"** axis with the slice:

```
Slice("mesh", [Subset("point", icell, label="cell"),
               Subset("point", iedge, label="edge"),
               Subset("point", ivert, label="vertex")])
```

where `icell`, `iedge` and `ivert` are integer arrays containing the indices of the cells, edges, and vertices respectively. Indexing the **"mesh"** axis with this slice yields a new multi-component axis where the different entity types may be distinguished and separately addressed. This transformation is shown in fig. 8.4 and the result of applying this index to a single-component, ragged Scott-Vogelius axis tree (fig. 8.3b) is shown in fig. 8.5.

By indexing the **"mesh"** axis in this way we make it such that the resulting axis tree has two distinct representations, as flat points or as a collection of distinct mesh entities, which exactly emulates DMPlex's data model.

Computing the renumbering

Our explanation so far has omitted how the full point numbering, used to interleave DoFs from different entity types, is produced. Shown in algorithm 12, one loops over the cells of the mesh and packs entities in their closure together. This means that DoFs associated with edges and vertices will be close in memory to their incident cell, improving performance for finite element assembly loops where one accesses all DoFs in a cell's closure together. To further improve locality, the outer loop over cells is ordered with an reverse Cuthill-McKee numbering [21] (`cell_order` in algorithm 12). This ensures that nearby cell closures are kept close in memory, improving data reuse for shared entities between adjacent cells.

The algorithm is very similar to that already used by Firedrake and PyOP2 (algorithm 3 in [53]) except, for reasons explained in section 7.2, points are partitioned only into OWNED and GHOST groups, instead of CORE, OWNED, and GHOST.

The integer arrays `icell`, `iedge`, and `ivert` are then all derived from this flat numbering.

Algorithm 12 Algorithm that computes the point renumbering to improve data locality in finite element assembly loops. Owned points are always stored before ghost points.

```
1  def plex_renumbering(plex: PETSc.DMPlex, cell_order: list[int]):
2      # bookkeeping to track result
3      renumbering = np.empty(num_points(plex))
4      owned_ptr = 0
5      ghost_ptr = num_owned_points(plex)
6      seen_points = set()
7
8      # loop over cells
9      c_start, c_end = plex.getHeightStratum(0)
10     for cell in range(c_start, c_end):
11         # renumber the cell
12         cell_renum = cell_order[cell]
13
14         # pack points in the cell closure together, skipping already
15         # seen points and respecting the owned/ghost partition
16         closure = get_closure(plex, cell_renum)
17         for pt in closure:
18             if pt in seen_points:
19                 continue
20             else:
21                 if is_ghost(pt):
22                     renumbering[ghost_ptr] = pt
23                     ghost_ptr += 1
24                 else:
25                     renumbering[owned_ptr] = pt
26                     owned_ptr += 1
27             seen_points.add(pt)
28     return renumbering
```

8.3 Finite element assembly

Now that we have the required global data structures we may now consider the finite element assembly process. For the Stokes problem above, to assemble the LHS matrix Firedrake constructs the following pyop3 loop expression:

```
loop(cell := mesh.cells.owned.index(),
      kernel(viscosity_global,
              output_mat[closure(cell), closure(cell)]))
```

which declares a loop over the locally-owned cells in the mesh (`mesh.cells.owned`), executing the local kernel (`kernel`) for each cell. `kernel` is a `pyop3 Function` that wraps the TSFC-generated loop kernel for the cell-wise evaluation. Its arguments have intents `READ` and `INC`.

The viscosity, being a `Global`, requires no indexing, whereas the matrix needs two indices to restrict it to a view of the DoFs supported on the cell. For a finite element assembly loop this means that each axis tree must be restricted to the closure of the current cell (section 3.2.1), which is represented by the map `closure(cell)`. Starting with an axis tree representing a Scott-Vogelius function space (fig. 8.5), the layout of the indexed tree that one gets from indexing with the cell closure is shown in fig. 8.6b. Note that, since the axis tree is nested, simply indexing with `closure(cell)` is syntactic sugar. Internally it is expanded into a complete index tree that can index all of the axes. This is shown in fig. 8.6a.

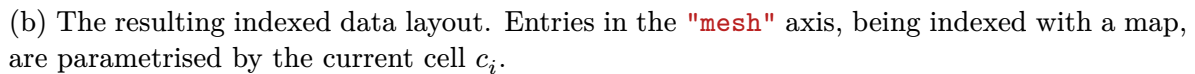
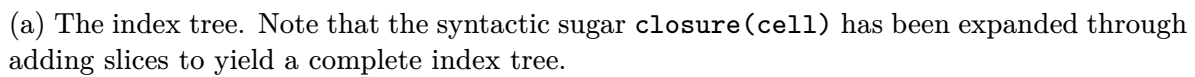
From this loop expression, `pyop3` is able to generate and execute distributed-memory parallel code for finite element assembly using the processes established in chapters 6 and 7. Nothing more is required.

8.3.1 More integral types

As well as cells, Firedrake also supports integrating over the interior and exterior facets of the mesh. In these cases the data that is packed and passed to the local kernel consists of the ‘macro’ cell formed from the cells in the support of the facet (e.g. fig. 8.7). Written in terms of DMPlex queries, this macro cell is constructed via the composition of closure and support, specifically `cl(supp(p))` (with p a facet).

With `pyop3` it is straightforward to express assembly loops for facet integrals using map composition (section 5.3.2). For example:

```
loop(facet := mesh.interior_facets.owned.index(),
      kernel(output_dat[closure(support(facet))]))
```



114

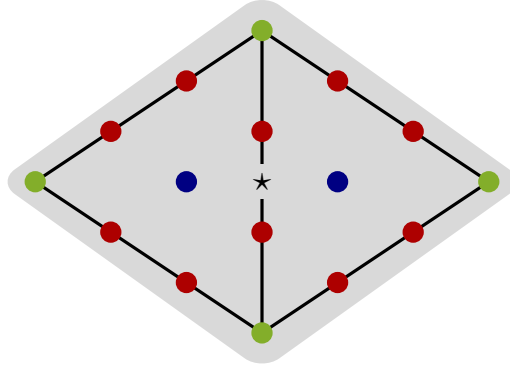


Figure 8.7: Stencil for an interior facet integral. The facet currently being iterated over is marked with the star.

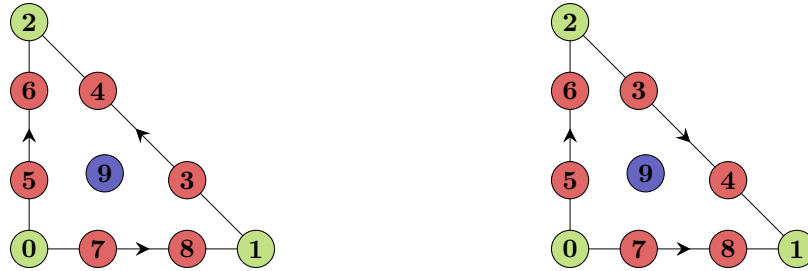


Figure 8.8: Reference P_3 element (left), and with an edge flipped (right).

8.3.2 Handling orientations

For finite element assembly, it is important that the data being packed into the local temporary is provided in ‘canonical’ form. In order to do this, adjacent cells must agree on the orientation of shared entities (i.e. facets, edges, and vertices). If these orientations do not agree then the local assembly kernel is passed mangled data, leading to an erroneous calculation.

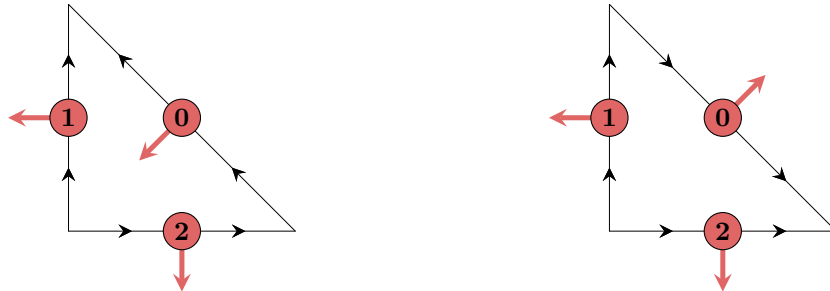


Figure 8.9: Reference Raviart-Thomas element (left), and with an edge flipped (right).

Two simple examples of where orientations have an impact are shown in figs. 8.8 and 8.9. Both show the impact of packing data for a finite element calculation for a triangle where one of the edges has been flipped. In both cases the packed data would be mangled. In fig. 8.8 the DoFs **3** and **4** are packed in the wrong order and the vector DoF in fig. 8.9 is pointing the wrong direction.

A number of approaches exist to rectify this issue. For meshes with simplex cells one can globally number the vertices of the mesh such that adjacent cells always agree on the orientation of the shared entities [69]. This numbering approach is also possible, with some limitations, for meshes with quadrilateral [35] and hexahedral [2] cells. However, the approach does not work in general across the full gamut of cell types, including mixed cell meshes, that can occur in finite element simulations. Instead, one needs to encode the orientation of the shared entities relative to the cell and *transform* the DoFs appropriately.

Some DoF transformations, such as the one necessary for fig. 8.8, are expressible as permutations. Permutations can be handled in advance by permuting the cell-node map used to pack the temporary such that DoFs are put into the right places in the temporary array. This is the approach taken by Firedrake that allows it to support Lagrange elements on hexahedral meshes. However, some transformations require more general operations that cannot be computed in advance [77, 75]. For example, to un-mangle the DoFs for fig. 8.9 the offending DoF must be flipped by multiplying it by -1 . This operation cannot be done in advance and must be done as part of the packing transformation. PyOP2 has no support for runtime transformations and so non-Lagrange elements (e.g. Raviart-Thomas) are not supported on hexahedral meshes in Firedrake.

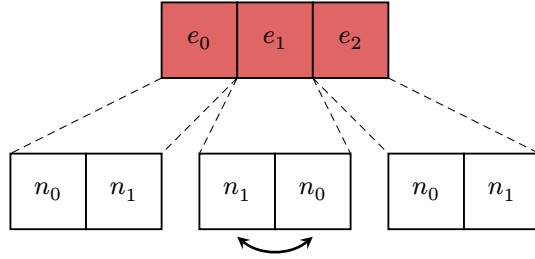
Expressing transformations with pyop3

Unlike PyOP2, expressing DoF transformations of this sort is relatively straightforward to do with pyop3. Since DoF packing is expressed simply as an indexing operation taking, say, a `Dat` and returning another `Dat`, further transformations can be applied to the packed object. For example, one can write repeated permutations of a packed `Dat` as follows:

```
packed0 = dat[closure(cell)]
packed1 = packed0[transform1]
packed2 = packed1[transform2]
```

This is not expressible in PyOP2 because packed temporaries are represented by a different type, called an `Arg`, which prevents repeated indexing.

To give an example of this working in practice we consider packing the edge DoFs for the case shown in fig. 8.8, omitting cell and vertex DoFs for simplicity. We want to permute the DoFs for a single edge, so the transformation that we want to express can be visualised as:



Which corresponds the following `pyop3` code:

```
cell = 0 # there is only one cell
packed = dat[closure(cell)][: , [[0,1],[1,0],[0,1]]]
```

The slice `(:)` included in the transformation indicates that the edge axis $(\{e_0, e_1, e_2\})$ should not be changed.

In order to extend this transformation to the more general case with multiple cells two additional pieces of information are required:

1. **The orientation of the edges, stored as an array of integers per cell closure.** For this simple case the orientation array of the flipped triangle is $\{c_0 \rightarrow [0,1,0]\}$, or, equivalently, `orientations = [[0,1,0]]`. An orientation of 0 is considered 'default', and 1 means that it is flipped.
2. **The permutation transformation for each possible orientation.** In this case, since there are only two ways that an edge can be oriented this is stored as $\{0 \rightarrow [0,1], 1 \rightarrow [1,0]\}$, or in code as `perms = [[0,1],[1,0]]`.

With these we can begin to transform the packing expression above to something more general. If we first substitute the permutations `[[0,1],[1,0],[0,1]]` with the appropriate entries from `perms` then we get:

```
cell = 0
packed = dat[closure(cell)][: , perms[[0,1,0]]]
```

Then, we can observe that the entries `[0,1,0]` are precisely the orientations of the edges, allowing us to make another substitution:

```

cell = 0
packed = dat[closure(cell)][:, perms[orientations[cell]]]

```

This gives us a general expression for permuting DoFs that applies to meshes with any number of cells.

By being able to encode the transformation symbolically, **pyop3** is able to generate code for performing them as part of the assembly loop, a necessary step towards enabling run-time transformations like that required for fig. 8.9. At present only permutation transformations have been implemented, to allow the **pyop3** version of Firedrake to assemble Lagrange elements on hexahedra, but extending the approach to support more general transformations should be straightforward. For instance, to implement the sign flip transformation shown in fig. 8.9 one could write:

```

packed0 = dat[closure(cell)]
flip(packed0, orientations[cell])

```

with `flip()` an automatically generated local kernel that conditionally applies the sign flip depending upon the provided orientation.

8.4 Outlook

In this chapter we have demonstrated the changes necessary to integrate **pyop3** into Firedrake and replicate existing PyOP2 functionality. In particular, these changes consist of replacing PyOP2 global data structures with **pyop3** ones and using **pyop3** loop expressions instead of PyOP2 parallel loops. The changes are all internal and so existing Firedrake scripts will run without modification.

In chapter 9 we take this integration further and explore the functionality and performance of a number of simple examples.

Chapter 9

Demonstrator applications

In order to demonstrate `pyop3`'s capability to replace `PyOP2` inside the Firedrake framework, we now present a number of examples comparing the old and new software stacks in terms of both functionality and performance. In particular, we consider two prototypical finite element problems, that run under `pyop3` without modification, and a more complex slope limiter computation for which `pyop3` significantly facilitates the development of.

Performance metrics

Analysing the performance of OP2-like mesh stencil codes is challenging because, since the specification of the local kernel is left to the user, the framework itself does no computation. For such a package to have ‘good’ performance, therefore, it must simply be able to *efficiently transfer data in and out of the local kernel*. Since the majority of mesh stencil computations are memory-bound, doing this efficiently can be critical to achieving good performance.

A suitable technique for quantifying memory performance is to produce a *roofline plot*. Shown in fig. 9.1, a roofline plot compares *arithmetic intensity*, defined as

$$\text{arithmetic intensity (FLOP/byte)} = \frac{\text{operation count (FLOP)}}{\text{memory volume (byte)}}, \quad (9.1)$$

against arithmetic throughput, measured in FLOP/s. Roofline plots are useful because they provide theoretical bounds for the maximum attainable performance for a computation. Given a computation with a high arithmetic intensity, performing many floating-point operations per byte of data loaded, performance is limited by the

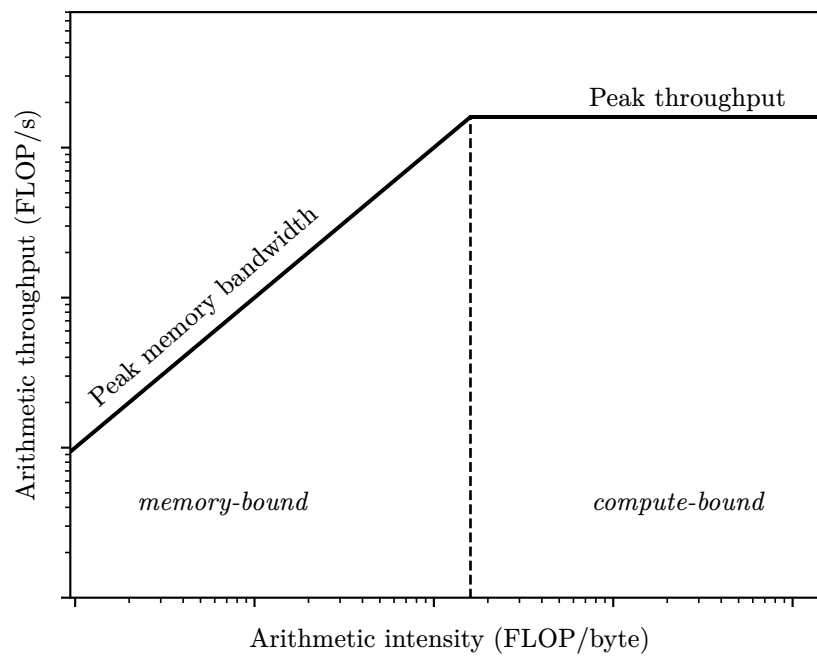


Figure 9.1: An example roofline plot. The solid lines represent a performance ‘ceiling’ that all computations must sit somewhere below. Note that additional lines for the bandwidths of the different cache levels, as well as one for vectorised instructions, have been omitted for simplicity.

Physical cores	32
Base frequency	3.6 GHz
L1 cache	32×32 kB
L2 cache	32×512 kB
L3 cache	4×32 MB
System memory	4×16 GB DDR4 3200 MT/s
Memory bandwidth (single core)	31.2 GB/s
Peak scalar throughput (single core)	11.9×10^9 FLOP/s
Peak AVX2 throughput (single core)	51.4×10^9 FLOP/s

Table 9.1: Details of the hardware used to run the experiments. Memory bandwidth and arithmetic throughput were measured with LIKWID [87]. Note that, since the experiments in this chapter were all run on a single core, only single core metrics are included.

rate at which FLOPs may be executed - we say that it is *compute-bound*. By contrast, computations with low arithmetic intensities are limited by the rate at which data may be streamed from main memory - these we call *memory-bound*.

Hardware

Experiments were performed on a workstation with a single AMD Ryzen Threadripper PRO 5975WX processor. Its details are summarised in table 9.1.

9.1 Solving the Helmholtz equation

Before any investigation of performance, we first demonstrate that Firedrake remains capable of solving PDEs under `pyop3`. To do so, we solve the sign-positive Helmholtz equation on a unit square using the formulation from [31] (pp. 91-93). The formulation reads: find $u \in V$ where $V \subset H^1(\Omega)$ such that

$$\int \nabla u \cdot \nabla v + uv \, d\Omega = \int f v \, d\Omega, \quad (9.2)$$

Listing 11 Firedrake code for solving the Helmholtz equation using degree 3 Lagrange elements.

```

1  from firedrake import *
2
3  mesh = UnitSquareMesh(...)
4  V = FunctionSpace(mesh, "P", 3)
5
6  u = TrialFunction(V)
7  v = TestFunction(V)
8
9  f = Function(V)
10 x, y = SpatialCoordinate(mesh)
11 f.interpolate((1+8*pi*pi)*cos(x*pi*2)*cos(y*pi*2))
12
13 # bilinear and linear forms
14 a = (inner(grad(u), grad(v)) + inner(u, v)) * dx
15 L = inner(f, v) * dx
16
17 u = Function(V) # function to hold the solution
18 solve(a == L, u)

```

where u is a trial function, v a test function, and f a known function. For this example we do not prescribe any boundary conditions and set

$$f = (1 + 8\pi^2) \cos(2\pi x) \cos(2\pi y), \quad (9.3)$$

which results in a known analytic solution of

$$u = \cos(2\pi x) \cos(2\pi y). \quad (9.4)$$

Code to solve this system using Firedrake is also taken from [31] and is shown in listing 11.

To demonstrate correctness the L_2 error between the numerical and analytical solutions, given by

$$\left(\int (u_{\text{numeric}} - u_{\text{analytic}}) \cdot (u_{\text{numeric}} - u_{\text{analytic}}) \, d\Omega \right)^{1/2}, \quad (9.5)$$

was measured against mesh refinement level for polynomial degree 1, 2, and 3. The resulting convergence behaviour is plotted in fig. 9.2. The results are bitwise identical to those produced by a pre-pyop3 Firedrake and show the expected convergence of at

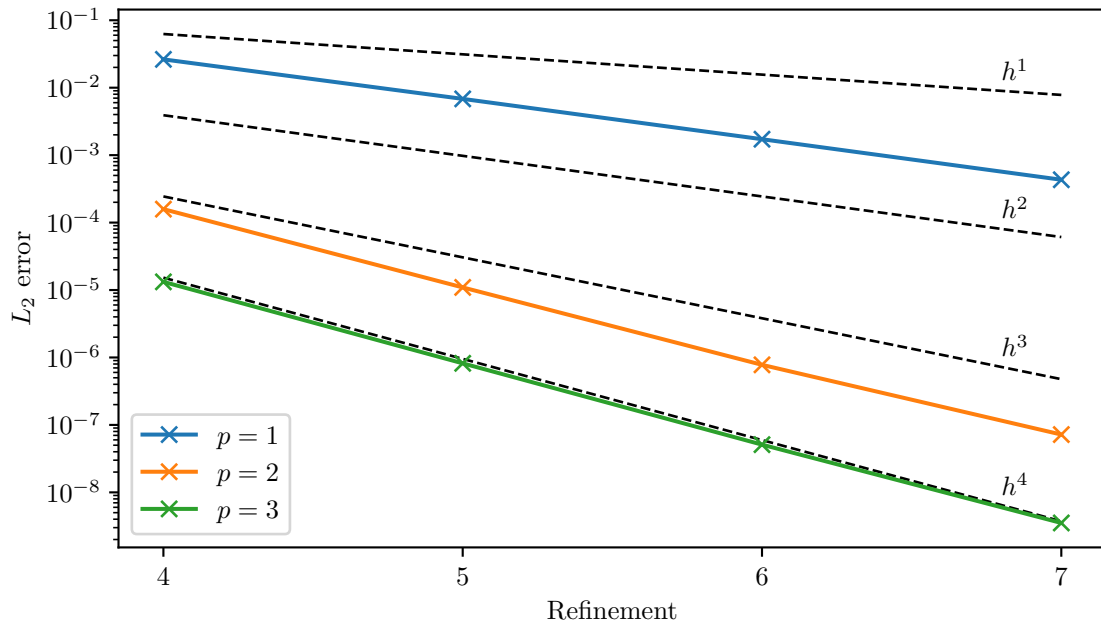


Figure 9.2: Convergence of the L_2 error against refinement level for polynomial degrees (p) 1, 2, and 3 when solving the Helmholtz equation with Firedrake and pyop3.

least $\sim O(h^{p+1})$ [15]. Thus we may safely conclude that Firedrake works as expected under pyop3.

9.2 One form assembly

For a simpler example where the contribution of `pyop3` may be better analysed we assembled the one form

$$\int \sum_i^N f_i v \, d\Omega, \quad (9.6)$$

with f_i known functions and v a test function. This form was chosen because it is ‘simple’, so the local kernel evaluation will not dominate the computation time, and to avoid invoking any sparse matrix insertion, which is often a bottleneck. The majority of the computation time is therefore spent performing pack/unpack operations, which is where the differences between PyOP2 and `pyop3` manifest. Firedrake code for assembling such a form is shown in listing 12.

Listing 12 Firedrake code to assemble the one form of eq. (9.6) in 2D with P_3 elements and 2 coefficients.

```

1  from firedrake import *
2
3  mesh = UnitSquareMesh(200, 200)
4  V = FunctionSpace(mesh, "P", 3)
5
6  coeff1 = Function(V)
7  coeff2 = Function(V)
8  v = TestFunction(V)
9  form = (coeff1 + coeff2) * v * dx
10
11 cofunction = assemble(form)

```

For this case, ‘good’ performance can be quantified as the generated code hitting a reasonable fraction of peak performance, according to a roofline plot, and also being close in performance to PyOP2.

In order to enable roofline analysis, two methods were used to approximate the total amount of data streamed from main memory. The first, termed *optimal caching*, assumes that values are only loaded from main memory once, remaining in cache for all subsequent uses. The other, called *pessimal caching*, does the opposite; values are never in cache so must be loaded from memory at each local kernel invocation.

For optimal caching the amount of data streamed from main memory (in bytes) is given by

$$\dim(V) \times (N + 1) \times 8 + n_{\text{map}}, \quad (9.7)$$

where $\dim(V)$ represents the number of DoFs associated with the function space and the $N + 1$ term accounts for the fact that there are N input arrays (one per coefficient) and a single output array. The factor of 8 is included as the values being streamed are in double precision (8 bytes). The additional term n_{map} accounts for the cell-node map that the packing code needs because the mesh is unstructured. It is given by

$$n_{\text{map}} = n_{\text{cells}} \times n_{\text{nodes}} \times 4 \quad (9.8)$$

where n_{cells} is the number of cells in the mesh, n_{nodes} is the number of nodes addressed by the map per iteration, equivalently the size of the packed temporary, and the factor of 4 is included because the map uses 4 byte integers. For pessimal caching, where

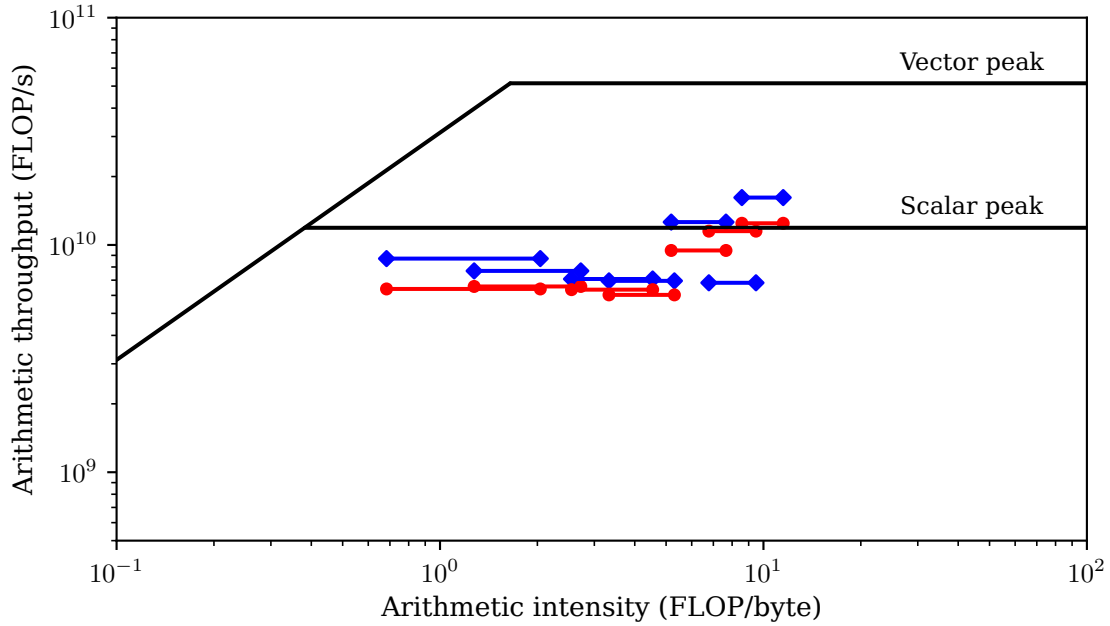


Figure 9.3: Roofline plot comparing the performance of PyOP2 (blue, diamonds) and `pyop3` (red, circles) for one form assembly (eq. (9.6)) where the polynomial degree is varied between 1 and 7. The top and bottom flat lines represent peak vector and scalar throughput respectively. As polynomial degree is increased so does arithmetic intensity, and so the points are ordered with lowest degree to the left. Entries are plotted as pairs of points to show both pessimal (eq. (9.9), left) and optimal (eq. (9.7), right) caching assumptions.

the cache is always missed, we expect to load

$$n_{\text{cells}} \times (N + 1) \times n_{\text{nodes}} \times 8 + n_{\text{map}} \quad (9.9)$$

bytes.

Due to the various data locality optimisations that PyOP2 and `pyop3` perform we expect that data locality is good, and so optimal caching is likely a better approximation of the true arithmetic intensity.

Performance results

To evaluate the performance of this assembly routine we performed two separate experiments, both run in serial on a 200×200 triangular mesh. First, the polynomial degree was increased between 1 and 7 for the single coefficient version of eq. (9.6) ($N = 1$). The results are shown in fig. 9.3. One can see that with increasing

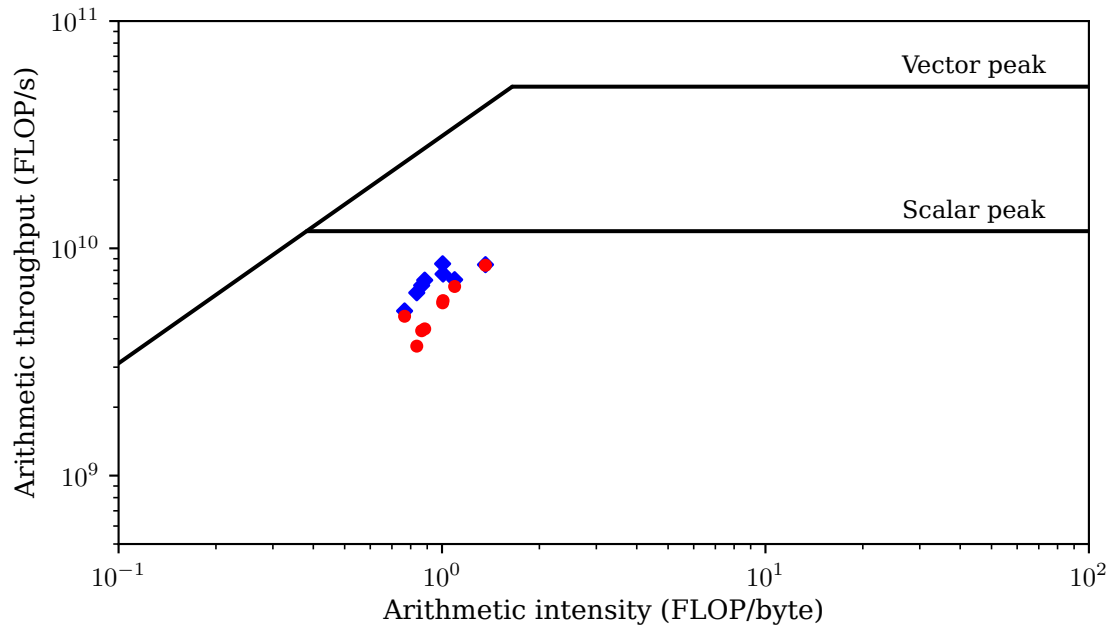


Figure 9.4: Roofline plot comparing PyOP2 (blue, diamonds) and `pyop3` (red, circles) performance for one form assembly (eq. (9.6)) with a varying number of coefficients between 0 and 7. Increasing the number of coefficients reduces the arithmetic intensity and so the left-most values are for the 7 coefficient case. For the arithmetic intensity the average from the pessimal (eq. (9.9)) and optimal (eq. (9.7)) caching assumptions was used.

degree (increasing arithmetic intensity) the throughput generally increases, eventually become compute-bound. This is because the computational payload of the local kernel is increasing and so the overhead from pack/unpack operations is amortized. At higher degree, some of the recorded throughput results surpass the measured scalar peak. This is because a sizeable number of vector instructions were generated by the compiler for these kernels.

In fig. 9.4 the polynomial degree was fixed to 1, and the number of coefficients varied between 0 and 7. As the number of coefficients increases a larger volume of data are loaded with relatively few floating point operations and thus the arithmetic intensity decreases. Both PyOP2 and `pyop3` show the same trend where increasing the number of functions reduces the arithmetic throughput. We anticipate that this is caused by the larger working set needed for the computation, though more investigation would be required to validate this.

Both sets of results show that the performance of `pyop3` is competitive, achieving in the majority of cases over 70% of the arithmetic throughput recorded for PyOP2 as well as operating at close to peak theoretical performance as defined by the roofline. Since the code generated by `pyop3` is very similar to that of PyOP2 we are confident that, with additional tuning, performance parity is achievable.

One possible source of confusion with both figures is the misleading fact that the kernels are shown to be compute-bound even at low degree. This only happens because we are running in serial. At full-node utilisation the total memory bandwidth increases by only a small amount, whilst the total throughput should increase by (approximately) a factor of 32. Performing these experiments in parallel would therefore lead to a significantly different roofline plot: the peak throughput would be much higher whilst the slope of the memory bandwidth would stay approximately the same. Shifting the lines like this moves the inflection point between memory-bound and compute-bound further to the right, which would move most of the points into the memory-bound regime.

9.3 Vertex-based slope limiter

As a final demonstrator application we implement part of a slope limiter algorithm. Slope limiters are postprocessing filters that bound solutions to prevent spurious oscillations near discontinuities [13]. Following the explanation in [50], one applies a

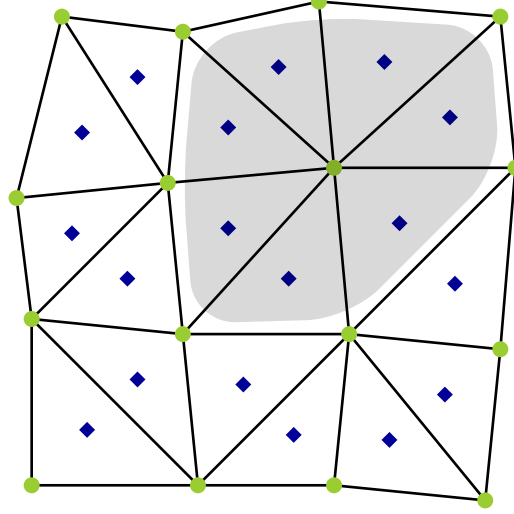


Figure 9.5: Stencil pattern for a slope limiter. The vertex value is set to the maximum value of the surrounding cells. Diamond-shaped and circular DoFs represent the P_0^{disc} (input) and P_1 (output) functions respectively.

Algorithm 13 Algorithm that writes the maximum values from a P_0^{disc} function (dg) to a P_1 function (cg) around each vertex. Values in cg are assumed to be initialised to a sufficiently large negative number that MAX always overwrites the initial value.

Require: cg, dg ▷ Input and output functions

```

1: for  $vertex$  in  $mesh.vertices$  do
2:   for  $cell$  in  $vertex2cells(vertex)$  do ▷ Loop over incident cells
3:      $cg[vertex] \leftarrow \text{MAX}(cg[vertex], dg[cell])$ 
4:   end for
5: end for

```

Listing 13 Firedrake and pyop3 code for a simple slope limiter (algorithm 13).

```
1  from firedrake import *
2  import pyop3 as op3
3
4  mesh = UnitSquareMesh(200, 200)
5  V_cg = FunctionSpace(mesh, "CG", 1)
6  V_dg = FunctionSpace(mesh, "DG", 0)
7  cg = Function(V_cg)
8  dg = Function(V_dg)
9
10 # this kernel sets the second argument to the maximum of
11 # both it and the first argument
12 max_kernel = op3.Function(..., [op3.READ, op3.RW])
13
14 op3.loop(v := mesh.vertices.index(),
15         op3.loop(c := mesh.star(v, k=2).index(),
16                 max_kernel(dg.dat[c], cg.dat[v])))
```

slope limiter by correcting some function u with a scalar correction factor α_e to

$$u(\mathbf{x}) = u_c + \alpha_e (\nabla u)_c \cdot (\mathbf{x} - \mathbf{x}_c), \quad 0 \leq \alpha_e \leq 1, \quad (9.10)$$

where $(\cdot)_c$ denote cell-averaged quantities and $\mathbf{x}$ the coordinates. For our demonstrator application we are interested in determining the value of the correction factor α_e , which depends on both the minimum ($u^{\min}$) and maximum ($u^{\max}$) values of u around each vertex.

If we restrict our attention to only the lowest-order discontinuous space, P_0^{disc} , the algorithm for determining both $u^{\min}$ and $u^{\max}$ is simple. Shown in algorithm 13 and fig. 9.5, one needs to visit each vertex, loop over adjacent cells, and set the vertex value, kept in a P_1 function space, to the minimum/maximum value from the surrounding cells.

Implementing the algorithm is straightforward in pyop3 and can be done in only a few lines (listing 13, line 14). To identify the cells surrounding the current vertex a star (st, section 3.2.1) map is used. The extra keyword argument $k=2$ is passed to ensure that the map only includes the cells in the star, skipping the vertex and edges.

The code that is generated from the loop expression is shown in listing 14. Notice that the inner loop is ragged (line 10) with extent set by the variable `p_0`, drawn

Listing 14 (Abbreviated) C code generated from the loop expression in listing 13. To better show the code’s behaviour the nested indirections have not been optimised away.

```

1  void pyop3_loop(...)
2  {
3      int32_t p_0;
4      double t_0[1];
5      double t_1[1];
6
7      for (int32_t i_0 = 0; i_0 <= end; ++i_0)
8      {
9          p_0 = array_0[i_0];
10         for (int32_t i_1 = 0; i_1 <= -1 + p_0; ++i_1)
11         {
12             t_0[0] = array_3[array_4[array_1[array_2[i_0] + i_1]]];
13             t_1[0] = array_5[array_6[i_0]];
14             max_kernel(&(t_0[0]), &(t_1[0]));
15             array_5[array_6[i_0]] = t_1[0];
16         }
17     }
18 }
```

from an externally provided array `array_0` (line 9). We also remark that the index expression for packing the P_0^{disc} cell data (line 12)

$$\text{array_3}[\text{array_4}[\text{array_1}[\text{array_2}[i_0] + i_1]]]$$

is relatively complex, illustrating `pyop3`’s ability to compose advanced index expressions.

As written, `PyOP2` is not capable of expressing this type of operation because the map from vertices to cells has variable arity. One can reframe the algorithm as a loop over cells, writing the maximum values to the vertices in the cell’s closure, but that is specific to this example; more complex algorithms like vertex-centered patch smoothers [25], where one solves a small finite element problem on each patch of cells, cannot be expressed in this way. `pyop3` therefore provides a more general solution.

9.3.1 Performance results

Lacking independent variables to change, the code was profiled for a single configuration, again in serial. The roofline results are shown in table 9.2. The

Arithmetic throughput	1.9×10^9 FLOP/s
Arithmetic intensity	0.072 FLOP/byte
Peak throughput	2.2×10^9 FLOP/s
Percentage of peak throughput	85%

Table 9.2: Roofline results from running the slope limiter code (listing 13). The arithmetic intensity was measured by LIKWID. Peak throughput was computed as the product *arithmetic intensity* $\times$ *memory bandwidth*.

arithmetic intensity is extremely low because only one floating point operation (a comparison) is performed, with many arrays being streamed from memory. The calculation is therefore strongly memory-bound.

Irrespective of this, the code achieves a good fraction of theoretical peak performance, meaning that `pyop3` is clearly generating performant code that makes effective use of the available hardware.

9.4 Outlook

This chapter demonstrates that `pyop3` has the capability to be used to solve real-world mesh stencil problems. Integration with Firedrake is sufficiently complete that PDEs may be solved, and one can even express new types of stencil computations not previously expressible with PyOP2.

In terms of performance, we consider the results to be encouraging. The performance of the code that `pyop3` generates is both comparable to PyOP2 and sufficiently close to theoretical peak to suggest that the approach is sound. The slope limiter results additionally show that reasonable performance is attainable even for circumstances not expressible with PyOP2.

Chapter 10

Conclusions

This thesis introduces a novel suite of abstractions that together are suitable for representing general-purpose mesh stencil calculations. Compared with existing frameworks, the new abstractions are better able to represent the semantics of the problem domain, enabling a greater range of algorithms, transformations, and optimisations to be expressed and applied.

The core abstractions introduced by this work are the axis tree (chapter 4) and index tree (chapter 5). Axis trees provide a means to flexibly describe a wide range of data layouts common to continuum mechanics simulations. They were inspired by the observation that data layouts are better described as trees, instead of traditional N-dimensional arrays or unstructured sets.

Index trees take axis trees and transform them into other, indexed, axis trees that carry information about the structure of the data access. They are valuable for creating *views* of arrays: new versions of arrays that impose additional structure symbolically, without requiring any copying of data. Views are essential for stencil calculations as they enable large, global arrays to be *packed* and *unpacked* into small local temporaries at each iteration.

By better capturing the structure that underlies mesh data, these abstractions allow for far more expressive ways of describing how the data are accessed during mesh stencil algorithms, thus enabling one to program computations intuitively from a high level of abstraction with a DSL (chapter 6). Axis trees and index trees therefore together form a powerful abstraction that ‘bridges the gap’ between the different limitations of existing stencil codes (section 2.2.2), allowing one to solve problems with *both* complex data structures *and* complex algorithms.

This thesis is also accompanied by a new Python library, `pyop3`, that provides an implementation of these new abstractions (chapter 6). From a high-level specification of the algorithm `pyop3` is able to generate high-performance, low-level code that efficiently evaluates the computation, even in parallel (chapter 7).

In chapter 8 we demonstrated `pyop3`'s functionality by integrating it into the finite element library Firedrake, replacing its precursor PyOP2. The code generated by `pyop3` has performance that is both comparable with PyOP2 and close to theoretical peak (chapter 9).

10.1 Summary of benefits

Compared with existing frameworks, in particular its predecessor PyOP2, `pyop3` can boast a number of significant improvements:

- By constructing data layouts programmatically with axis trees, `pyop3` is able to handle a much wider range of different possible data layouts that may appear in mesh stencil calculations. In particular, it has support for *ragged* data layouts (section 4.3), which PyOP2 has no capability to handle.
- Indexing operations may be composed arbitrarily without incurring any copying of data. This facilitates the writing of complex *symbolic* transformations that can be evaluated at run-time (e.g. section 8.3.2).
- By decoupling the data *layout* specification (axis trees) from the data *access* specification (index trees), `pyop3` has transparent support for different semantically-equivalent data layouts (sections 4.2 and 5.4).
- Unlike PyOP2, which only has support for single-loop, single-kernel algorithms, `pyop3` has a DSL capable of expressing a much wider range of algorithms involving multiple (possibly nested) loops and multiple kernels (e.g. section 9.3).
- `pyop3` can perform inspector-executor optimisations on the indirection expressions used to address global data structures to improve data locality and reduce data volume (section 6.3).
- Various improvements have been made to the abstractions and algorithms related to distributed memory parallelism (chapter 7). In particular:

- The CORE/OWNED/GHOST set partitioning of PyOP2 has been broken up into OWNED/GHOST for parallel data layouts and CORE/ROOT/LEAF for iteration sets (sections 7.1 and 7.2.2). This decouples the specification of the data layout from the specification of the parallel loop, enabling support for stencils of different sizes and for arrays with mismatching parallel distributions.
- Ghost DoFs for multi-field data structures are always stored at the end of the array, allowing for no-copy transformations between local and global representations (section 7.1).
- The statuses of both ROOT and LEAF points are tracked to reduce the amount of communication that is needed between loop invocations (section 7.2.1).

10.2 Future research directions

The core motivation for creating `pyop3` was to enable the development of computational methods not possible with other mesh stencil frameworks. The number of future research directions is therefore very high; we summarise a selection of them below:

Firedrake on GPUs

One of the core benefits of having high-level abstractions in scientific software is that implementation-specific detail may be hidden, allowing multiple computational backends to be supported with little to no effort. Notably this means that GPUs, fast becoming the dominant hardware on the largest supercomputers, may be supported with very few changes to the code. This contrasts hugely with codes written in low-level languages, where porting functionality to GPUs requires rewriting much of the software stack and can take multiple years to achieve.

In order for Firedrake to be able make effective use of GPUs it is important that finite element assembly is supported on them. The execution of the assembly loops is the responsibility of `pyop3` (section 8.3) and so therefore extending `pyop3` to support evaluating mesh stencils on GPUs is necessary.

The work needed to enable this functionality is not too onerous. It consists of two parts: (1) generating GPU code (e.g. CUDA) instead of C from the loop expressions,

and (2) having GPU-aware data structures that can transparently copy data between host and device. Of these, (2) is straightforward to do, a very similar lazy-transfer process already exists to minimise ghost exchanges (section 7.2.1); and functionality for (1) has already been implemented in loopy and PyOP2 [49], it must simply be ported over to `pyop3`.

Once finite element assembly on a GPU is supported, Firedrake should be able to offload entire solves. PETSc efforts to support solving linear and non-linear problems on a GPU are becoming increasingly mature [62, 61] and the rest of the Firedrake software stack (e.g. TSFC) exist at an abstraction level where the CPU/GPU consideration is not relevant.

Beyond finite elements

One of the original inspirations for `pyop3`'s development was the fact that a number of algorithms, specifically additive-Schwartz methods [25] and hybridisation [28], did not fit into the single-loop, single-kernel paradigm enforced by PyOP2 and so sui generis changes were needed to implement them. `pyop3`, with its much more flexible interface, will enable these algorithms to be rewritten in a more general way.

Furthermore, it should also become possible to implement other mesh stencil algorithms outside of a finite element setting. Examples of such applications include:

Graph neural networks Graph neural networks (GNNs) are a type of neural network that operates on graphs [73]. They are valuable for modelling objects with complex structures across a wide range of disciplines including molecular chemistry [79] and social networks [63] where graphs are a natural way of representing the problem input.

One of the core operations needed by a GNN is *message passing*, where nodes exchange information with their neighbours. Mathematically, this operation can be represented as the node-wise update

$$\mathbf{h}_u = \phi \left(\mathbf{x}_u, \bigoplus_{v \in \mathcal{N}_u} \psi(\mathbf{x}_u, \mathbf{x}_v) \right), \quad (10.1)$$

where $\mathbf{x}_{(\cdot)}$ are the current node states and $\mathcal{N}_u$ is the set of nodes sharing an edge with u [16]. The update is computed by applying a non-parametrised aggregation function $\bigoplus$ (e.g. addition) to the result of a *message function* ψ inside an *update function* ϕ . Both ψ and ϕ can be learned by the network. This

operation, where local updates are computed using local unknowns, is a stencil operation and thus could be computed using `pyop3`.

Further, there is ongoing research in the machine learning community regarding whether more higher-dimensional data structures such as simplices [40] and hypergraphs [26] have advantages over graphs due to their ability to express relations between more than two vertices at once. These data structures are very similar in nature to unstructured meshes, again suggesting that `pyop3` would be effective.

Molecular dynamics Molecular dynamics (MD) codes simulate physical phenomena by using a very large number of particles and evolving them in time according to computed forces between them. Such codes are applicable to a wide range of domains including computational chemistry [88], material modelling [86], astrophysics [38], nuclear physics [90], and molecular biology [34].

One of the core operations performed by an MD code is the evaluation of the interactions between each possible pair of particles. To improve on the naive $O(n^2)$ complexity of this operation it is common to separate the interactions of nearby particles, which must be handled individually, with those of more distant particles, which may be handled using agglomeration strategies such as the Ewald summation method or Fast Multipole Method ([71], ch. 4), or disregarded altogether.

To classify particles as nearby or distant, a common approach is to partition the simulation domain into a mesh. With appropriately sized cells one can then identify nearby particles as those that lie within the stencil of cells adjacent to that containing the current particle. Since particles are free to migrate, the number of them within each cell is not fixed and so the resulting stencil operation is *ragged*.

`pyop3` is fully capable of describing stencil loops over ragged data structures and thus should be able to express this type of computation. In fact, the value of such an abstraction for MD codes is demonstrated by the fact that a very similar piece of software already exists. PPMD [72] is an MD code that performs computations by applying local kernels to either individual particles or pairs of particles using a PyOP2-like interface. Like PyOP2 and `pyop3`, performance portability is achieved with code generation and the code is transparently

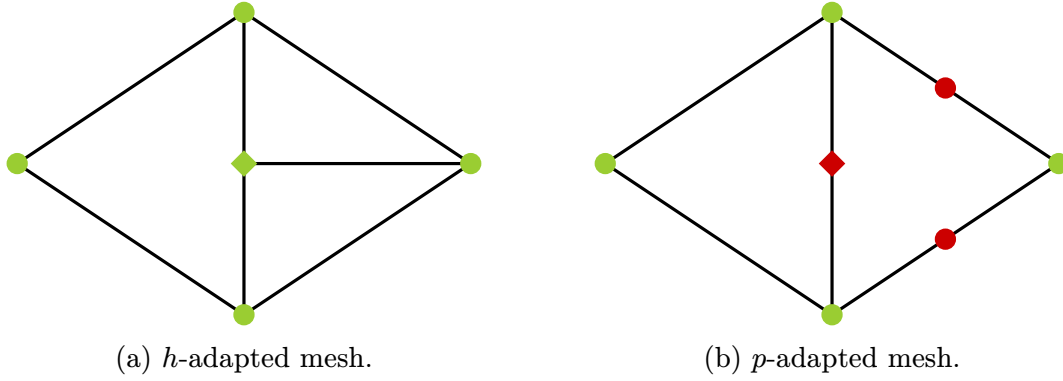


Figure 10.1: A two-cell mesh where h -refinement (left) and p -refinement (right) have been applied to the right-most cell. Diamonds indicated constrained DoFs.

parallel. Despite obvious similarities between PPMD and PyOP2, the codes exist independently because limitations in PyOP2's design - specifically the lack of support for ragged data structures - made it unsuitable for MD simulations. The abstraction provided by `pyop3` is sufficiently general that it could enable a unified approach.

Runtime transformations

As described in section 8.3.2, cells in unorientable meshes may disagree on the relative orientation of shared entities. Performing local assembly therefore requires that the global DoFs be transformed into some canonical setting. Since the shared DoFs may need to be transformed differently for each cell, this transformation must occur at runtime, during the pack/unpack operation. For 'simple' function spaces this transformation is merely a permutation, but for others (e.g. $H(\text{div})$ and $H(\text{curl})$) the transformation involves the application of a matrix. With some small modifications, `pyop3` should be capable of representing such transformations symbolically, enabling fast code to be generated that applies them.

A similar situation can occur with adaptive methods. Shown in fig. 10.1 for the h -adaptive (left, mesh refinement) and p -adaptive (right, polynomial degree refinement) cases, adjacent cells can have 'hanging nodes' - DoFs that are visible to one cell but not the other. These DoFs are not true degrees of freedom and they do not appear in the global matrix/vector. Instead, they are computed using the 'true' unconstrained DoFs. For example, the extra vertex in fig. 10.1a is usually constrained to be the average value of the surrounding vertices. The application of these constraints is

another cell-wise transformation of the global data, similar in nature to the orientation transformations above, and so `pyop3` should facilitate the implementation of adaptive methods. Furthermore, p -adaptive methods store a variable number of DoFs per element and so require ragged/non-trivially structured arrays. These should be expressible in `pyop3` using axis trees. Note h -adaptivity and p -adaptivity can be combined to hp -adaptivity and any techniques developed should apply equally to this case.

Runtime data layout transformations

As described in section 2.2.1, it is desirable for mesh stencil codes to have support for expressing the same data in a variety of layouts, so as to take advantage of the performance characteristics of different hardware. In sections 4.2 and 5.4 we established that this is entirely possible within `pyop3`. However, at present data layouts are considered immutable for the duration of a program. This is not always correct since there are circumstances where one data layout may be optimal for one loop and another layout be optimal for another, for example if one is executed on a GPU and another on the CPU. By committing to one layout over another `pyop3` is sacrificing performance.

To remedy this, `pyop3` data structures could internally store multiple equivalent arrays, each with a different data layout. The optimal layout for different loops could then be used, transparently to the user. To ensure data consistency between the arrays, `pyop3` would need to employ state-tracking to lazily copy between them, similarly to how packages ensure consistency between data on a CPU and GPU (e.g. [62]).

Despite the differing layouts, axis trees make performing this copy trivial. For example, given two arrays, `dat_soa` and `dat_aos`, with axis trees representing different-but-equivalent data layouts, copying one to the other is as simple as executing:

```
dat_soa.assign(dat_aos)
```

Different data structures

The abstractions introduced in this thesis are sufficiently high-level that the specific choices of underlying data structure, currently NumPy arrays and PETSc matrices,

are not fixed. For cases like additive-Schwartz methods, using PETSc matrices may have poor performance because they are designed for massively parallel applications, whereas patch problems instead require the assembly of many small matrices. Instead, using dense arrays or matrix types from other linear algebra libraries like LIBXSMM [33] may be more suitable. Similarly, for particle and adaptive methods the data structures may need to be frequently updated to account for moving particles or extra unknowns. For these cases one may want to use dynamically resizable data structures like the `vector` type from the C++ standard library. By providing a higher level of abstraction for data structures - `Globals`, `Dats`, and `Mats` - `pyop3` abstracts away the underlying data structures such that alternatives could be used transparently.

Communication optimisations

In some circumstances the DoF star forests encapsulating the overlap between adjacent processes (section 7.1.2) are suboptimal. Depending on the stencil being applied, some or all of the ghost data may not need to be transferred.

One example where this occurs is with cell-wise assembly loops for discontinuous Galerkin (DG) finite element methods. Since DG methods involve integration over facets (section 8.3.1), the mesh overlap between processes must be large enough for the correct macro cells to exist. This is suboptimal for cell-wise assembly loops because the overlap is now larger than necessary: the extra DoFs in the overlap will never be modified and so they do not need to be communicated. An example demonstrating this is shown in fig. 10.2; the ghost DoFs (hollow circles) are never touched by the stencil and so their values do not need to be exchanged. This is especially a problem at high order, where the number of redundantly transferred DoFs can be very large.

In order to fix this, one needs to have *per-loop* parallel exchange behaviour. This is possible to represent in `pyop3` because it is aware of the parallel communication patterns. Using an algorithm similar to algorithm 11, `pyop3` could inspect the mesh stencils to construct per-loop star forests that only exchange the needed data.

Temporal locality optimisations

As well as the optimisations listed above, it may be valuable to pursue implementing more traditional compiler optimisations within `pyop3`. As an example, one could consider performing a *sparse tiling* transformation to DG finite element assembly

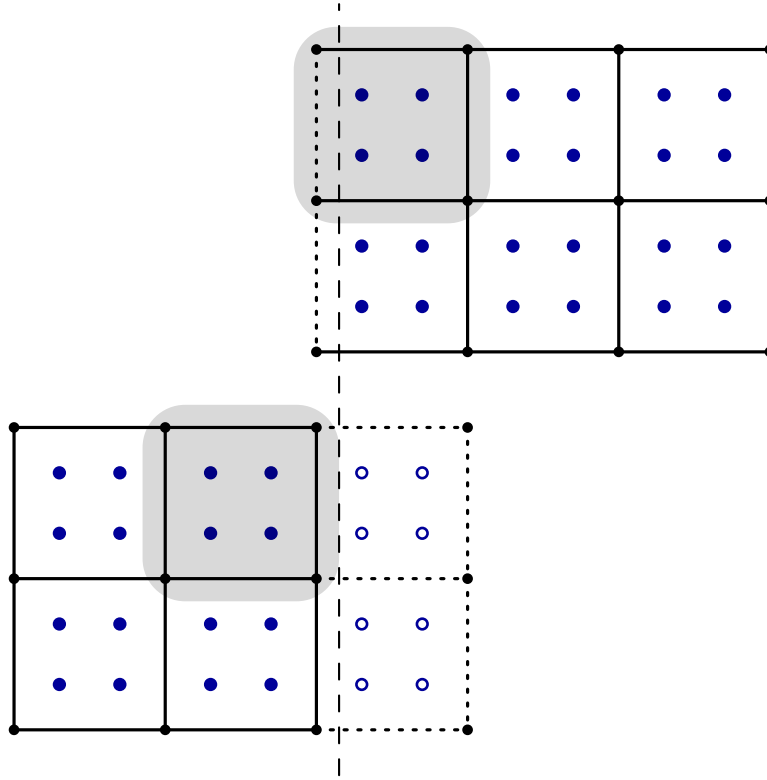


Figure 10.2: Cell-wise parallel stencil for a finite element calculation involving a Q_1^{disc} function space. DoFs are represented as blue circles, with hollow circles representing ghost DoFs owned by the other process. The mesh overlap is larger than necessary since interior facet integrals over ‘macro’ cells must be accommodated.

loops. It is usual for DG methods to have assembly loops over both the cells and interior facets of the mesh:

```
loop(c := mesh.cells.index(), cell_kernel(...))
loop(f := mesh.interior_facets.index(), facet_kernel(...))
```

However, this approach exhibits poor data locality because each loop will load much of the same data from memory. To improve performance we could therefore partition the mesh into small *tiles*, and then perform both assembly loops on each tile in turn:

```
loop(tile := mesh.tiles().index(), [
    loop(c := tile.cells.index(), cell_kernel(...))
    loop(f := tile.interior_facets.index(), facet_kernel(...))
])
```

This will improve data locality since data for the second loop will not yet have been evicted from cache. `pyop3` should have enough information to represent and perform this transformation.

Note that this is a simplified approach not accounting for complications such as different access descriptors. More general sparse tiling approaches (e.g. [48, 82, 58]) could also be pursued and may benefit from the improved access to topological information exposed by `pyop3`'s abstractions.

Bibliography

- [1] Ahmad Abdelfattah et al. “GPU Algorithms for Efficient Exascale Discretizations”. In: *Parallel Computing* 108 (Dec. 2021), p. 102841. ISSN: 01678191. DOI: 10.1016/j.parco.2021.102841. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0167819121000879> (visited on 03/04/2024).
- [2] Rainer Agelek et al. “On Orienting Edges of Unstructured Two- and Three-Dimensional Meshes”. In: *ACM Transactions on Mathematical Software* 44.1 (July 24, 2017), pp. 1–22. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3061708. URL: <https://dl.acm.org/doi/10.1145/3061708> (visited on 07/01/2021).
- [3] Martin S. Alnæs et al. “Unified Form Language: A Domain-Specific Language for Weak Formulations of Partial Differential Equations”. In: *ACM Transactions on Mathematical Software* 40.2 (2014). DOI: 10.1145/2566630. arXiv: 1211.4047.
- [4] Douglas N. Arnold and Franco Brezzi. “Mixed and Nonconforming Finite Element Methods : Implementation, Postprocessing and Error Estimates”. In: *ESAIM: Mathematical Modelling and Numerical Analysis* 19.1 (1985), pp. 7–32. ISSN: 0764-583X, 1290-3841. DOI: 10.1051/m2an/1985190100071. URL: <http://www.esaim-m2an.org/10.1051/m2an/1985190100071> (visited on 09/23/2024).
- [5] Douglas N. Arnold, Richard S. Falk, and Ragnar Winther. “Preconditioning in $H(\text{Div})$ and Applications”. In: *Mathematics of Computation* 66 (1997).
- [6] 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.

- [7] Satish Balay et al. *PETSc Users Manual*. ANL-95/11 - Revision 3.15. Argonne National Laboratory, 2021. URL: <https://www.mcs.anl.gov/petsc>.
- [8] Satish Balay et al. *PETSc Web Page*. 2021. URL: <https://www.mcs.anl.gov/petsc>.
- [9] Igor A. Baratta et al. *DOLFINx: The next Generation FEniCS Problem Solving Environment*. 2023. DOI: 10.5281/zenodo.10447666. Pre-published.
- [10] Doug Baxter, Ravi Mirchandaney, and Joel H Saltz. “Run-Time Parallelization and Scheduling of Loops”. In: *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*. 1989, pp. 303–312.
- [11] 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).
- [12] Jack D. Betteridge, Patrick E. Farrell, and David A. Ham. “Code Generation for Productive, Portable, and Scalable Finite Element Simulation in Firedrake”. In: *Computing in Science & Engineering* 23.4 (July 1, 2021), pp. 8–17. ISSN: 1521-9615, 1558-366X. DOI: 10.1109/MCSE.2021.3085102. URL: <https://ieeexplore.ieee.org/document/9447889/> (visited on 09/27/2024).
- [13] Rupak Biswas, Karen D. Devine, and Joseph E. Flaherty. “Parallel, Adaptive Finite Element Methods for Conservation Laws”. In: *Applied Numerical Mathematics* 14.1–3 (Apr. 1994), pp. 255–283. ISSN: 01689274. DOI: 10.1016/0168-9274(94)90029-9. URL: <https://linkinghub.elsevier.com/retrieve/pii/0168927494900299> (visited on 09/02/2024).
- [14] Daniele Boffi, Franco Brezzi, and Michel Fortin. *Mixed Finite Element Methods and Applications*. Vol. 44. Springer, 2013.
- [15] Susanne C. Brenner and L. 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.
- [16] Michael M. Bronstein et al. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. May 2, 2021. DOI: 10.48550/arXiv.2104.13478. arXiv: 2104.13478 [cs]. URL: <http://arxiv.org/abs/2104.13478> (visited on 01/06/2025). Pre-published.

- [17] Jed Brown et al. “Composable Linear Solvers for Multiphysics”. In: *2012 11th International Symposium on Parallel and Distributed Computing*. 2012 11th International Symposium on Parallel and Distributed Computing (ISPD). Munich, Germany: IEEE, June 2012, pp. 55–62. ISBN: 978-1-4673-2599-8. DOI: 10.1109/ISPD.2012.16. URL: <http://ieeexplore.ieee.org/document/6341494/> (visited on 09/22/2024).
- [18] Jed Brown et al. “libCEED: Fast Algebra for High-Order Element-Based Discretizations”. In: *Journal of Open Source Software* 6.63 (July 9, 2021), p. 2945. ISSN: 2475-9066. DOI: 10.21105/joss.02945. URL: <https://joss.theoj.org/papers/10.21105/joss.02945> (visited on 02/04/2022).
- [19] 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>.
- [20] *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
- [21] E. Cuthill and J. McKee. “Reducing the Bandwidth of Sparse Symmetric Matrices”. In: *Proceedings of the 1969 24th National Conference*. Acm ’69. New York, NY, USA: Association for Computing Machinery, 1969, pp. 157–172. ISBN: 978-1-4503-7493-4. DOI: 10.1145/800195.805928. URL: <https://doi.org/10.1145/800195.805928>.
- [22] Raja Das, Joel Saltz, and Reinhard Hanxleden. “Slicing Analysis and Indirect Accesses to Distributed Arrays”. In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee et al. Red. by Gerhard Goos and Juris Hartmanis. Vol. 768. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 152–168. ISBN: 978-3-540-57659-4 978-3-540-48308-3. DOI: 10.1007/3-540-57659-2_9. URL: http://link.springer.com/10.1007/3-540-57659-2_9 (visited on 11/27/2023).
- [23] Andreas Dedner et al. “A Generic Interface for Parallel and Adaptive Discretization Schemes: Abstraction Principles and the Dune-Fem Module”. In: *Computing* 90.3–4 (Nov. 2010), pp. 165–196. ISSN: 0010-485X, 1436-5057. DOI:

10.1007/s00607-010-0110-3. URL: <http://link.springer.com/10.1007/s00607-010-0110-3> (visited on 08/16/2024).

- [24] 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).
- [25] 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).
- [26] Yifan Feng et al. “Hypergraph Neural Networks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 17, 2019), pp. 3558–3565. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v33i01.33013558. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/4235> (visited on 01/06/2025).
- [27] Matteo Frigo and Steven G. Johnson. “The Design and Implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 216–231. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840301. URL: <http://ieeexplore.ieee.org/document/1386650/> (visited on 03/26/2022).
- [28] 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).
- [29] 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).

- [30] David A. Ham et al. “Efficient N-to-M Checkpointing Algorithm for Finite Element Simulations”. In: *SIAM Journal on Scientific Computing* 46.6 (2024), B830–B859. DOI: 10.1137/23M1613724. eprint: <https://doi.org/10.1137/23M1613724>. URL: <https://doi.org/10.1137/23M1613724>.
- [31] 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.
- [32] Charles R. Harris et al. “Array Programming with NumPy”. In: *Nature* 585.7825 (Sept. 17, 2020), pp. 357–362. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-2649-2. URL: <https://www.nature.com/articles/s41586-020-2649-2> (visited on 08/19/2024).
- [33] Alexander Heinecke et al. “LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation”. In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Nov. 2016, pp. 981–991. DOI: 10.1109/SC.2016.83.
- [34] Scott A. Hollingsworth and Ron O. Dror. “Molecular Dynamics Simulation for All”. In: *Neuron* 99.6 (Sept. 2018), pp. 1129–1143. ISSN: 08966273. DOI: 10.1016/j.neuron.2018.08.011. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0896627318306846> (visited on 01/06/2025).
- [35] Miklós Homolya and David A. Ham. “A Parallel Edge Orientation Algorithm for Quadrilateral Meshes”. In: *SIAM Journal on Scientific Computing* 38.5 (Jan. 2016), S48–S61. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/15M1021325. URL: <http://epubs.siam.org/doi/10.1137/15M1021325> (visited on 06/29/2021).
- [36] 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).
- [37] Miklós Homolya et al. “TSFC: A Structure-Preserving Form Compiler”. In: *SIAM Journal on Scientific Computing* 40.3 (Jan. 2018), pp. C401–C428. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/17M1130642. URL: <https://epubs.siam.org/doi/10.1137/17M1130642> (visited on 04/14/2021).

- [38] C. J. Horowitz et al. *Neutron Star Crust and Molecular Dynamics Simulation*. Sept. 23, 2011. DOI: 10.48550/arXiv.1109.5095. arXiv: 1109.5095 [astro-ph]. URL: <http://arxiv.org/abs/1109.5095> (visited on 01/06/2025). Pre-published.
- [39] Yuanming Hu et al. “Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures”. In: *ACM Transactions on Graphics* 38.6 (Dec. 31, 2019), pp. 1–16. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3355089.3356506. URL: <https://dl.acm.org/doi/10.1145/3355089.3356506> (visited on 09/15/2022).
- [40] Yechan Hwang et al. “Multi-Order Simplex-Based Graph Neural Network for Brain Network Analysis”. In: *Medical Image Computing and Computer Assisted Intervention – MICCAI 2024*. Ed. by Marius George Linguraru et al. Vol. 15005. Cham: Springer Nature Switzerland, 2024, pp. 532–541. ISBN: 978-3-031-72085-7 978-3-031-72086-4. DOI: 10.1007/978-3-031-72086-4_50. URL: https://link.springer.com/10.1007/978-3-031-72086-4_50 (visited on 01/06/2025).
- [41] 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).
- [42] Fredrik Kjolstad et al. “Taco: A Tool to Generate Tensor Algebra Kernels”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). Urbana, IL: IEEE, Oct. 2017, pp. 943–948. ISBN: 978-1-5386-2684-9. DOI: 10.1109/ASE.2017.8115709. URL: <http://ieeexplore.ieee.org/document/8115709/> (visited on 04/07/2022).
- [43] Andreas Klöckner. “Loo.Py: From Fortran to Performance via Transformation and Substitution Rules”. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY 2015*. The 2nd ACM SIGPLAN International Workshop. Portland, OR, USA: ACM Press, 2015, pp. 1–6. ISBN: 978-1-4503-3584-3. DOI: 10.1145/2774959.2774969. URL: <http://dl.acm.org/citation.cfm?doid=2774959.2774969> (visited on 12/04/2020).

- [44] Andreas Klöckner. “Loo.Py: Transformation-Based Code Generation for GPUs and CPUs”. In: *Proceedings of ARRAY ‘14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming*. Edinburgh, Scotland.: Association for Computing Machinery, 2014. DOI: 10.1145/2627373.2627387.
- [45] Andreas Klöckner, Lucas C. Wilcox, and T. Warburton. “Array Program Transformation with Loo.Py by Example: High-Order Finite Elements”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY 2016*. The 3rd ACM SIGPLAN International Workshop. Santa Barbara, CA, USA: ACM Press, 2016, pp. 9–16. ISBN: 978-1-4503-4384-8. DOI: 10.1145/2935323.2935325. URL: <http://dl.acm.org/citation.cfm?doid=2935323.2935325> (visited on 10/14/2020).
- [46] Matthew G. Knepley and Dmitry A. Karpeev. “Mesh Algorithms for PDE with Sieve I: Mesh Distribution”. In: *Scientific Programming* 17.3 (2009), p. 948613. DOI: 10.3233/SPR-2009-0249. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.3233/SPR-2009-0249>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.3233/SPR-2009-0249>.
- [47] Matthew G. Knepley, Michael Lange, and Gerard J. Gorman. “Unstructured Overlapping Mesh Distribution in Parallel”. June 19, 2015. arXiv: 1506.06194 [cs, math]. URL: <http://arxiv.org/abs/1506.06194> (visited on 02/16/2021).
- [48] Christopher D. Krieger et al. “Loop Chaining: A Programming Abstraction for Balancing Locality and Parallelism”. In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW). Cambridge, MA, USA: IEEE, May 2013, pp. 375–384. ISBN: 978-0-7695-4979-8. DOI: 10.1109/IPDPSW.2013.68. URL: <http://ieeexplore.ieee.org/document/6650909/> (visited on 06/03/2021).
- [49] Kaushik Kulkarni and Andreas Klöckner. “UFL to GPU: Generating near Roofline Actions Kernels”. 2021. DOI: 10.6084/m9.figshare.14495301. URL: <http://mscroggs.github.io/fenics2021/talks/kulkarni.html>.

- [50] Dmitri Kuzmin. “A Vertex-Based Hierarchical Slope Limiter for p -Adaptive Discontinuous Galerkin Methods”. In: *Journal of Computational and Applied Mathematics* 233.12 (Apr. 2010), pp. 3077–3085. ISSN: 03770427. DOI: 10.1016/j.cam.2009.05.028. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0377042709003318> (visited on 09/02/2024).
- [51] Chi-Chung Lam, P. Sadayappan, and Rephael Wenger. “Optimal Reordering and Mapping of a Class of Nested-Loops for Parallel Execution”. In: *Languages and Compilers for Parallel Computing*. Ed. by David Sehr et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 315–329. ISBN: 978-3-540-69128-0.
- [52] Michael Lange et al. “Achieving Efficient Strong Scaling with Petsc Using Hybrid MPI/OpenMP Optimisation”. In: *Supercomputing*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 97–108. ISBN: 978-3-642-38750-0.
- [53] 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).
- [54] 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).
- [55] C. L. Lawson et al. “Basic Linear Algebra Subprograms for Fortran Usage”. In: *ACM Transactions on Mathematical Software* 5.3 (Sept. 1979), pp. 308–323. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/355841.355847. URL: <https://dl.acm.org/doi/10.1145/355841.355847> (visited on 08/16/2024).
- [56] M. Louboutin et al. “Devito (v3.1.0): An Embedded Domain-Specific Language for Finite Differences and Geophysical Exploration”. In: *Geoscientific Model Development* 12.3 (2019), pp. 1165–1187. DOI: 10.5194/gmd-12-1165-2019. URL: <https://www.geosci-model-dev.net/12/1165/2019/>.

- [57] Fabio Luporini et al. “Architecture and Performance of Devito, a System for Automated Stencil Computation”. In: *ACM Trans. Math. Softw.* 46.1 (Apr. 2020). ISSN: 0098-3500. DOI: 10.1145/3374916. URL: <https://doi.org/10.1145/3374916>.
- [58] Fabio Luporini et al. “Automated Tiling of Unstructured Mesh Computations with Application to Seismological Modelling”. In: *ACM Trans. Math. Softw.* 45.2 (May 2019). ISSN: 0098-3500. DOI: 10.1145/3302256. URL: <https://doi.org/10.1145/3302256>.
- [59] Graham R. Markall et al. “Finite Element Assembly Strategies on Multi-Core and Many-Core Architectures”. In: *International Journal for Numerical Methods in Fluids* 71.1 (Jan. 10, 2013), pp. 80–97. ISSN: 02712091. DOI: 10.1002/fluid.3648. URL: <http://doi.wiley.com/10.1002/fluid.3648> (visited on 10/12/2020).
- [60] Aaron Meurer et al. “SymPy: Symbolic Computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: <https://doi.org/10.7717/peerj-cs.103>.
- [61] Richard Tran Mills et al. *PETSc/TAO Developments for Early Exascale Systems*. June 12, 2024. arXiv: 2406.08646 [cs]. URL: <http://arxiv.org/abs/2406.08646> (visited on 07/22/2024). Pre-published.
- [62] Richard Tran Mills et al. “Toward Performance-Portable PETSc for GPU-based Exascale Systems”. In: *Parallel Computing* 108 (2021), p. 102831. ISSN: 0167-8191. DOI: 10.1016/j.parco.2021.102831. URL: <https://www.sciencedirect.com/science/article/pii/S016781912100079X>.
- [63] Federico Monti et al. “Fake News Detection on Social Media Using Geometric Deep Learning”. In: ICLR. New Orleans, USA, 2019.
- [64] 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).

- [65] OpenMP Architecture Review Board. *OpenMP API Specification 6.0*. 2024. URL: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf>.
- [66] M. Puschel et al. “SPIRAL: Code Generation for DSP Transforms”. In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 232–275. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840306. URL: <http://ieeexplore.ieee.org/document/1386651/> (visited on 08/16/2024).
- [67] Jonathan Ragan-Kelley et al. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *ACM SIGPLAN Notices* 48.6 (June 23, 2013), pp. 519–530. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2499370.2462176. URL: <https://dl.acm.org/doi/10.1145/2499370.2462176> (visited on 09/27/2024).
- [68] 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/>.
- [69] Marie E. Rognes, Robert C. Kirby, and Anders Logg. “Efficient Assembly of H(Div) and H(Curl) Conforming Finite Elements”. In: *SIAM Journal on Scientific Computing* 31.6 (Jan. 2010), pp. 4130–4151. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/08073901X. URL: <http://epubs.siam.org/doi/10.1137/08073901X> (visited on 06/29/2021).
- [70] Emma Rothwell. *Implementing a Restricted Function Space Class in Firedrake*. July 24, 2024. arXiv: 2408.05217 [cs, math]. URL: <http://arxiv.org/abs/2408.05217> (visited on 08/14/2024). Pre-published.
- [71] William Robert Saunders. “Development Of A Performance-Portable Framework For Atomistic Simulations”. University of Bath, 2019.
- [72] William Robert Saunders, James Grant, and Eike Hermann Müller. “A Domain Specific Language for Performance Portable Molecular Dynamics Algorithms”. In: *Computer Physics Communications* 224 (Mar. 2018), pp. 119–135. ISSN: 00104655. DOI: 10.1016/j.cpc.2017.11.006. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0010465517303892> (visited on 10/29/2021).

- [73] F. Scarselli et al. “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20.1 (Jan. 2009), pp. 61–80. ISSN: 1045-9227, 1941-0093. DOI: 10.1109/TNN.2008.2005605. URL: <http://ieeexplore.ieee.org/document/4700287/> (visited on 01/06/2025).
- [74] 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).
- [75] Matthew W Scroggs et al. “Construction of Arbitrary Order Finite Element Degree-of-Freedom Maps on Polygonal and Polyhedral Cell Meshes”. In: *ACM Transactions on Mathematical Software (TOMS)* 48.2 (2022), pp. 1–23.
- [76] Matthew W. Scroggs et al. *DefElement: An Encyclopedia of Finite Element Definitions*. 2024. URL: <https://defelement.com>.
- [77] Matthew W. Scroggs et al. “Basix: A Runtime Finite Element Basis Evaluation Library”. In: *Journal of Open Source Software* 7.73 (May 25, 2022), p. 3982. ISSN: 2475-9066. DOI: 10.21105/joss.03982. URL: <https://joss.theoj.org/papers/10.21105/joss.03982> (visited on 10/17/2022).
- [78] Barry Smith et al. “PETSc’s Software Strategy for the Design Space of Composable Extreme-Scale Solvers”. In: *Preprint ANL/MCS-P2059-0312, Argonne National Laboratory* (2012), pp. 16–18.
- [79] Jonathan M. Stokes et al. “A Deep Learning Approach to Antibiotic Discovery”. In: *Cell* 180.4 (Feb. 2020), 688–702.e13. ISSN: 00928674. DOI: 10.1016/j.cell.2020.01.021. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0092867420301021> (visited on 01/06/2025).
- [80] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science & Engineering* 12.3 (May 2010), pp. 66–73. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.69. URL: <http://ieeexplore.ieee.org/document/5457293/> (visited on 09/27/2024).

- [81] 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.
- [82] Michelle Mills Strout et al. “Generalizing Run-Time Tiling with the Loop Chain Abstraction”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014 IEEE International Parallel & Distributed Processing Symposium (IPDPS). Phoenix, AZ, USA: IEEE, May 2014, pp. 1136–1145. ISBN: 978-1-4799-3800-1 978-1-4799-3799-8. DOI: 10.1109/IPDPS.2014.118. URL: <https://ieeexplore.ieee.org/document/6877342> (visited on 06/03/2021).
- [83] András Attila Sulyok et al. “Locality Optimized Unstructured Mesh Algorithms on GPUs”. In: *Journal of Parallel and Distributed Computing* 134 (Dec. 2019), pp. 50–64. ISSN: 07437315. DOI: 10.1016/j.jpdc.2019.07.011. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0743731519301698> (visited on 09/27/2024).
- [84] 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).
- [85] *SYCL™ 2020 Specification (Revision 9)*. 2020. URL: <https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf>.
- [86] A. P. Thompson et al. “LAMMPS - a Flexible Simulation Tool for Particle-Based Materials Modeling at the Atomic, Meso, and Continuum Scales”. In: *Comp. Phys. Comm.* 271 (2022), p. 108171. DOI: 10.1016/j.cpc.2021.108171.
- [87] J. Treibig, G. Hager, and G. Wellein. “LIKWID: A Lightweight Performance-Oriented Tool Suite for X86 Multicore Environments”. In: *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. San Diego CA, 2010.
- [88] Wilfred F. Van Gunsteren and Herman J. C. Berendsen. “Computer Simulation of Molecular Dynamics: Methodology, Applications, and Perspectives in Chemistry”. In: *Angewandte Chemie International Edition in English* 29.9 (Sept.

- 1990), pp. 992–1023. ISSN: 0570-0833. DOI: 10.1002/anie.199009921. URL: <https://onlinelibrary.wiley.com/doi/10.1002/anie.199009921> (visited on 01/21/2025).
- [89] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. “Automated Empirical Optimizations of Software and the ATLAS Project”. In: *Parallel Computing* (2001).
 - [90] Nicholas R. Williams et al. “Atomistic Investigation of the Structure and Transport Properties of Tilt Grain Boundaries of UO₂”. In: *Journal of Nuclear Materials* 458 (Mar. 2015), pp. 45–55. ISSN: 00223115. DOI: 10.1016/j.jnucmat.2014.11.120. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0022311514009271> (visited on 01/06/2025).
 - [91] Chang Yu et al. “MeshTaichi: A Compiler for Efficient Mesh-Based Operations”. In: *ACM Transactions on Graphics* 41.6 (Dec. 2022), pp. 1–17. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3550454.3555430. URL: <https://dl.acm.org/doi/10.1145/3550454.3555430> (visited on 08/15/2024).
 - [92] Junchao Zhang et al. “The PetscSF Scalable Communication Layer”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 842–853. DOI: 10.1109/TPDS.2021.3084070.