`pyop3`: A Domain-Specific Language for Expressing Iterations over Mesh-like Data Structures

Connor J. Ward and David A. Ham

April 29, 2024

Contents

Chapter 1

Introduction

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

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

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

The remainder of this paper is laid out as follows…

Chapter 2

Background

2.0.1 Inspector-executor model

[4]
[7] [6] [1]

2.1 Domain-specific languages

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

For a moderately complex stencil operation that we will refer to throughout this thesis we consider solving the Stokes equations using the finite element method (FEM) [5]. The Stokes equations are a linearisation of the Navier-Stokes equations and are used to describe fluid flow for laminar (slow and calm) media. For domain Ω they are given by

$$-\nu\Delta u + \nabla p = f \quad \text{in } \Omega, \tag{2.1}$$
$$\nabla \cdot u = 0 \quad \text{in } \Omega, \tag{2.2}$$
$$\tag{2.3}$$

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

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

For the finite element method we seek the solution to the *variational*, or *weak*, formulation of these equations. These are obtained by multiplying each

equation by a suitable *test function* and integrating over the domain. For 2.1, with v as the test function and integrating by parts, this gives

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

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

For the second equation we simply get

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

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

2.2 Related work

Chapter 3

Mesh-like data layouts

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

3.1 Data layouts for the finite element method

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

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

To provide a motivating example, consider the mesh shown in section 3.1. Degree 3 Lagrange elements have been used and these have 1 DoF per vertex, 2 per edge and 4 per cell. DoFs are always stored contiguously per mesh point, and so the data layout for this mesh would look something like that shown in section 3.1. It is clear that, due to the variable step size for each mesh point, an N-dimensional array (with $N > 1$) is a poor fit for describing the layout. One could also view the data as just a flat array (figure ZZZ), but this loses the information about the mesh points. We can therefore conclude that mesh data layouts require a new abstraction for comprehensively describing their semantics: *axis trees*.

3.1.1 Axis trees

From section 3.1 it can be observed that the data layout naturally decomposes into a tree-like structure. For every class of topological entity (i.e. vertex, edge

or cell) there is a distinct number of DoFs associated with it.

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

And axis tree is composed of a hierarchy of *axes*, and each axis has one or more *axis components*.

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

Computing offsets

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

3.1.2 Interleaving axis components

3.1.3 Ragged arrays

Chapter 4

Indexing

Why do we want to index arrays? Why do we need a flexible system?

4.1 Indexing axis trees

Begin by giving an example of indexing a linear axis tree (10, 3)[::2, 1:] (and explain this terminology). Explain/show that the axis tree needs to necessarily shrink but the expressions needed to index it go from a*3+b to (2*x)*3+(y+1). Show the (relabelled) axis tree to demonstrate shrinkage.

Describe the process of indexing an array or axis tree: parse index tree, traverse index tree and generate new axes and collect index expressions, produce a new set of axes with the same layout as before but new axes and index expressions. (note that both axis trees and arrays are indexable, since we can loop over slices of axes) Demonstrate this using the same example as above. Use the ASCII view of the trees to demonstrate? Note that when we index we keep the layouts and index expressions separate, see index composition section for the reason.

Explain why we need to use index *trees* here: to build a wider variety of axes. If we want to build a new tree like set of axes (e.g. for closure packing) then we need to be able to pack different axis components together.

4.2 Loops

A loop index is constructed by doing axis[???].index(). This loop index can then be used inside an index tree like a slice or map.

A loop index has no shape. e.g. axes[p] where p is a loop index is scalar.

Perhaps provide a motivating example where we have array[p, :], or something, show that the layout expression contains p and hence we can substitute in an iname or index value.

4.3 Maps

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

How to build a map.

Give closure as an example. Index tree.

4.3.1 Ragged maps

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

4.3.2 Map composition

e.g. g(f(p))

4.4 Index composition

i.e. indexing an indexed thing

cannot directly insert as it breaks composition, need to store index_exprs separately

Chapter 5

pyop3

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

5.0.1 Context-sensitive ...

context-sensitive things: maps and loops

5.1 Code generation

5.1.1 pyop3 transformations

expand loop contexts

pack unpack transforms

5.1.2 Lowering to loopy

5.1.3 Loopy transformations

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

5.2 PETSc integration

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

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

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

Chapter 6

Parallelism

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

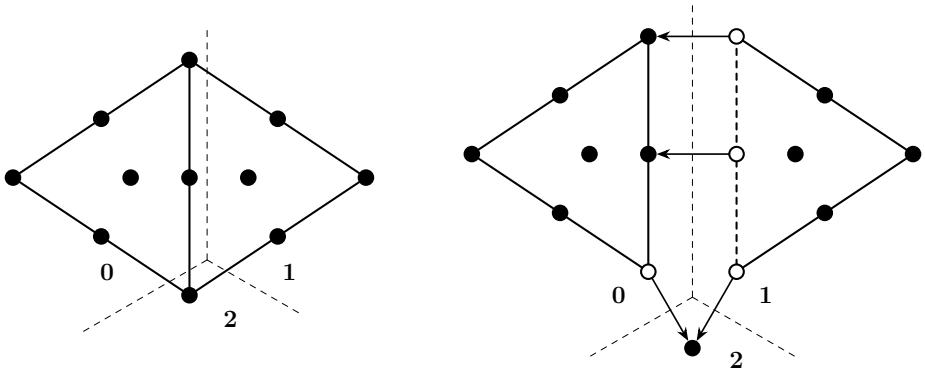
6.1 Message passing with star forests

Almost all message passing in `pyop3` is handled by star forests, specifically by PETSc star forests (`PetscSF`) [8].

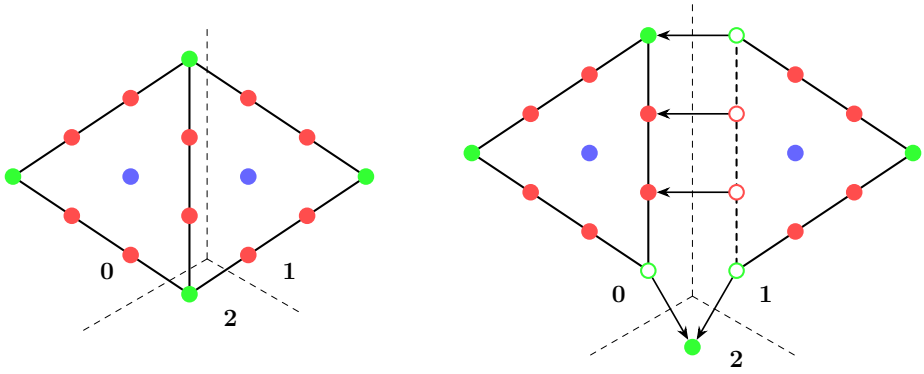
A star forest is defined as a collection of stars, where a star is defined as a tree with a single root and potentially many leaves. Star forests are effective for describing point-to-point MPI operations because they naturally encode the source and destination nodes as roots and leaves of the stars.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

Some terminology

- Owned Points are termed "owned" if they are present on a process and are not a leaf pointing to some other rank.
- Core Points are "core" if they are owned *and* are not part of (i.e. a root of) any star.



(a) TODO



(b) TODO

Figure 6.1: TODO

6.2 Overlapping computation and communication

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

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

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

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

6.2.1 Lazy communication

Coupled with the goal of "don't wait for data you don't need", `pyop3` also obeys the principle of "don't send data if you don't have to". `pyop3` associates with each parallel data structure two attributes: `leaves_valid` and `pending_reduction`. The former tracks whether or not leaves (ghost points) contain up-to-date values. The latter tracks, in a manner of speaking, the validity of the roots of the star forest. If the leaves of the forest were modified, `pending_reduction` stores the reduction operation that needs to be executed for the roots of the star forest to contain correct values. As an example, were values to be incremented into the leaves[1], a `SUM` reduction would be required for owned values to be synchronised. If there is no pending reduction, the roots are considered to be valid.

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

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

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

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

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

6.3 Performance results

6.4 Examples

6.4.1 Residual assembly

6.4.2 Matrix construction

6.4.3 Fieldsplit

6.5 Future work

6.6 Conclusions

Bibliography

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

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

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

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

[5] 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).

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

[7] 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.

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