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

Connor J. Ward and David A. Ham

January 31, 2024

Abstract

Chapter 1

Introduction

The remainder of this paper is laid out as follows...

Chapter 2

Background

2.0.1 Inspector-executor model

[?]
[?] [?] [?]

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) larsonFiniteElementMethod2013. 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 + ∇p = f   in Ω,
$\nabla\cdot$u = 0   in Ω,

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

$$u = g \quad \text{on } \Gamma. \tag{2.1}$$

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 2.1.1, with v as the test function and integrating by parts, this gives

$$\int \nu\nabla u : \nabla v \mathrm{d}\Omega - \int p\nabla \cdot v \mathrm{d}\Omega = \int f \cdot v \mathrm{d}\Omega \tag{2.2}$$

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

$$\int q\,\nabla \cdot u \mathrm{d}\Omega = 0. \tag{2.3}$$

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

Chapter 4

`pyop3`

Chapter 5

Parallelism

Just like Firedrake (e.g. [?]) 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 knepleyExascaleComputingThreads2015 and would increase the complexity of the code.

5.1 Message passing with star forests

Almost all message passing in `pyop3` is handled by star forests, specifically by PETSc star forests (PetscSF) zhangPetscSFScalableCommunication2021.

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.

5.2 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. [**?**] 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.

5.2.1 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$_validandpending_reduction.Theformertrackswhetherorn$
$to-datevalues.Thelattertracks, inamannerofspeaking, thevalidityoftherootsofthestarforest.Iftheleave$

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

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

- 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.

5.3 Performance results

5.4 Examples

5.4.1 Residual assembly

5.4.2 Fieldsplit

5.4.3 Ragged maps

5.5 Future work

5.6 Conclusions