Imperial College London
Department of Computing
Software Performance Optimisation Group

Productive and Efficient Computational Science Through Domain-specific Abstractions

Florian Rathgeber

October 2014

Supervised by: Dr. David A. Ham, Prof. Paul H. J. Kelly

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing
of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Florian Rathgeber

Abstract

In an ideal world, scientific applications are computationally efficient, maintainable and composable and allow scientists to work very productively. We argue that these goals are achievable for a specific application field by choosing suitable domain-specific abstractions that encapsulate domain knowledge with a high degree of expressiveness.

This thesis demonstrates the design and composition of domain-specific abstractions by abstracting the stages a scientist goes through in formulating a problem of numerically solving a partial differential equation. Domain knowledge is used to transform this problem into a different, lower level representation and decompose it into parts which can be solved using existing tools. A system for the portable solution of partial differential equations using the finite element method on unstructured meshes is formulated, in which contributions from different scientific communities are composed to solve sophisticated problems.

The concrete implementations of these domain-specific abstractions are Firedrake and PyOP2. Firedrake allows scientists to describe variational forms and discretisations for linear and non-linear finite element problems symbolically, in a notation very close to their mathematical models. PyOP2 abstracts the performance-portable parallel execution of local computations over the mesh on a range of hardware architectures, targeting multi-core CPUs, GPUs and accelerators. Thereby, a separation of concerns is achieved, in which Firedrake encapsulates domain knowledge about the finite element method separately from its efficient parallel execution in PyOP2, which in turn is completely agnostic to the higher abstraction layer.

As a consequence of the composability of those abstractions, optimised implementations for different hardware architectures can be automatically generated without any changes to a single high-level source. Performance matches or exceeds what is realistically attainable by hand-written code. Firedrake and PyOP2 are combined to form a tool chain that is demonstrated to be competitive with or faster than available alternatives on a wide range of different finite element problems.

To my family

Acknowledgements

I would like to express my thanks and gratitude towards all the people who supported me throughout my PhD and the writing of this thesis.

Firstly, I would like to thank my supervisors David Ham, who has guided my work throughout, inspired me with his passion and enthusiasm, provided helpful advice, support and vision, and was always ready to share his vast knowledge – and opinions – and Paul Kelly, whose experience, patience and dedication were invaluable in keeping me on track.

Many thanks also to my examiners Marie Rognes and Gerard Gorman for the challenging questions and inspiring discussion during my viva.

I was fortunate to share an office with my William-Penney colleagues Graham Markall, Lawrence Mitchell, Fabio Luporini, Doru Bercea, Carlo Bertolli, Andrew McRae and Francis Russell. Not only were they great collaborators on PyOP2 and Firedrake, enduring my pedantry when it comes to keeping a clean commit history, but also always happy to discuss ideas and help tracking down the odd bug. Graham's work has always been a great inspiration to me, already at the time I was working on my MSc dissertation in Stockholm, which ultimately brought me to Imperial. He is a pleasure to work with and I knew I could rely on him blindly, which he would of course deny in his humble manner. Lawrence is not only a fountain of productivity and knowledge, in particular when it comes to anything related to solvers, but also has a near-psychic ability to track down bugs. Fabio and Doru had the misfortune of starting at a time where my code pedantry had already been establishes, but they coped remarkably well. Their obsession with crocodiles was always a topic that spurred discussions with any visitors to the office. Carlo was always a re-

liable source of support, guidance and honest critique in his own unique self-deprecating manner and was always reachable even after he left us to work at IBM's T.J. Watson research centre. Andrew preferred the company in the William-Penney lab to his colleagues in the maths department and was a steady source of entertainment with his particular kind of humour and choice of (GitHub) user names. Behind the irony however there is a very rigorous mathematical thinker with a critical eye, so don't be fooled. When not vehemently disagreeing with Fabio on the temperature setting of the A/C, Francis mostly kept silently in the background, but was a valuable source of advice for obscure and hard problems with C, C++, systems architecture and Linux in general. He was unfortunately drafted to work on other projects so we could not tap into his expertise for Firedrake and PyOP2 as much as we would have liked.

Michael Lange and Christian Jacobs were regular participants in our group meetings and invaluable contributors to Firedrake. Michael's experience was very helpful to get up Firedrake up and running on various different supercomputers. Colin Cotter was always a reliable source of advice when I was out of my depth on the maths.

Not to forget my former AMCG neighbours Simon Funke, who is not only a good friend, but also an inspiring discussion partner, occasionally until late at night, and Patrick Farrell, who was always happy to share his vast knowledge and provide ideas and advice in his uniquely direct way.

When starting my PhD in the Earth Sciences department I was warmly welcomed by my first pod neighbours Johnny, Frank, Nikos, Eleni and Liwei. Many thanks also go to Tim, Adam, Stephan, Alex, Jon, James, Ben, Simon, Sam, Guiseppe and Dave for many discussions and advice.

I would further like to thank Fabio Luporini, Simon Funke and Carlo Bertolli for helpful suggestions when proofreading parts of this thesis.

Lastly, this work would not have been possible without the constant encouragement and loving support of my family. Although distant spatially, they were always there for me, kept a watchful eye on my well-being and helped me through difficult times and with difficult decisions.

My research was funded by EPSRC Grant EP/I00677X/1: *Multi-layered abstractions for PDEs*. I would further like to acknowledge the use of HPC facilities at Imperial College as well as the UK national supercomputing facility ARCHER.

Contents

Chapter 1

Introduction

1.1 Thesis Statement

The key to computationally efficient, maintainable and composable scientific software in a specific domain is the composition of suitable domain-specific abstractions, which encapsulate domain knowledge with a high degree of expressiveness and enable scientists to conduct very productive research without the need to be experts in their implementation.

1.2 Overview

Many scientific programs and libraries are islands, developed to solve a very specific research problem for a specific kind of user on a particular hardware platform. Performance, robustness, maintainability and extensibility are frequently an afterthought and hard to achieve due to the design of the software. Keeping up with a rapidly changing landscape of hardware platforms, in particular in high-performance computing, is an uphill battle and as a consequence, computational resources are not optimally utilised. Furthermore it is often not feasible to port an application to a different platform due to lack of expertise and resources.

This thesis demonstrates a novel approach of developing scientific software for numerically solving partial differential equations. By abstracting the stages a scientist goes through in formulating the problem, domain-specific abstractions can be built and composed, which encapsulate domain knowledge of each stage with a high degree of expressiveness. This knowledge is used to transform the problem into a different, lower level

representation and decompose it into parts which can be solved using existing tools. A system consisting of the abstraction layers Firedrake and PyOP2 is formulated, in which contributions from different scientific communities are composed to solve sophisticated problems.

Firedrake allows scientists to describe variational forms and discretisations for linear and non-linear finite element problems symbolically in a notation very close to their mathematical models. It is built on top of PyOP2, which abstracts the performance-portable parallel execution of local assembly kernels on a range of hardware architectures, and the FEniCS components UFL and FFC. The presented framework unifies the goals of performance, robustness, maintainability and extensibility.

1.3 Technical Contributions

The primary contribution of this thesis is the design and composition of two abstraction layers for the portable solution of partial differential equations using the finite element method on unstructured meshes.

In Chapter 4, the design and implementation of PyOP2, a domain-specific language (DSL) embedded in Python for performance-portable parallel computations on unstructured meshes across different hardware architectures, is presented. PyOP2 targets multi-core CPUs with OpenMP, GPUs and accelerators with CUDA and OpenCL and distributed parallel computations with MPI. At runtime, optimised, problem and backend specific low-level code is generated, just-in-time (JIT) compiled and scheduled for efficient parallel execution.

PyOP2 is used as the parallel execution layer for Firedrake, a novel portable framework for the automated solution of partial differential equations (PDEs) using the finite element method (FEM), presented in Chapter 5. Firedrake uses the established Unified Form Language UFL [Alnæs et al., 2014] to describe weak forms of PDEs, which are translated into computational kernels by a modified version of the FEniCS Form Compiler FFC [Kirby and Logg, 2006]. Assembly operations are transformed into local computations over mesh entities, and passed to PyOP2 for efficient parallel execution, while PETSc [Balay et al., 1997] is employed to provide the unstructured mesh and solve linear and non-linear systems.

In Chapter 6, the versatility and performance of the approach is demon-

strated on a wide range of different finite element problems.

Some of the work presented has been undertaken in collaboration with other researchers. PyOP2 draws inspiration from the OP2 framework, mainly developed at the University of Oxford by Mike Giles, Gihan Mudalige and Istvan Reguly. I am the primary contributor to PyOP2, responsible for large parts of the overarching design, architecture and API specification, including the backend selection and dispatch mechanism (Section 4.3). Support for mixed types (Section 4.8) is entirely my contribution, as are large parts of the code generation for the sequential and OpenMP backends and the linear algebra interface (Section 4.6), in particular the implementation of sparsity patterns. Early contributions to PyOP2 have been made by Graham Markall [2013]. The CUDA and OpenCL backends (Section 4.4.2) were mostly implemented by Lawrence Mitchell and Nicolas Loriant, who are also the main contributors to the PyOP2 MPI support (Section 4.7) and the parallel execution plan (Section 4.5) respectively. The COFFEE AST optimiser (Section 4.2.2) was contributed by Fabio Luporini.

I am a key contributor to Firedrake's design and architecture and implemented the interface to FFC, the support for splitting forms and assembling mixed systems (Section 5.2), the expression splitter (Section 5.3) and parts of the form assembly (Section 5.4). Major contributions to Firedrake have been made by Lawrence Mitchell, including the solver interface (Section 5.6). The expression assembler (Section 5.3), assembly caching and Dirichlet boundary conditions (Section 5.5) have been mainly implemented by David Ham and the mesh interface to PETSc DMPlex by Michael Lange. Support for extruded meshes and tensor product elements was contributed by Gheorghe-Teodor Bercea and Andrew McRae.

1.4 Dissemination

The work presented in this thesis is based on software released under open source licenses, and the design of this software and results have been disseminated in the scientific community through publications.

PyOP2 is published in the following conference papers:

Rathgeber et al. [2012] "PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes" introduces preliminary work in progress on PyOP2 and the integration with the Fluidity CFD

code base. A performance comparison of an advection-diffusion problem implemented in Fluidity, DOLFIN and the PyOP2 sequential and CUDA backends is presented.

Markall et al. [2013] "Performance-Portable Finite Element Assembly Using PyOP2 and FEniCS", mainly authored by Graham Markall and me, contains updated performance results for advection-diffusion, comparing Fluidity, DOLFIN and the PyOP2 OpenMP, MPI and CUDA backends.

Both publications predate Firedrake and contain code samples that use parts of the PyOP2 public API that have since been revised.

I have presented PyOP2 and Firedrake at the following conferences and workshops: i) Facing the Multicore-Challenge III, Stuttgart, Germany, September 2012; ii) Second International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), Salt Lake City, Utah, November 2012; iii) FEniCS'13 Workshop, Cambridge, UK, March 2013; iv) SciPy 2013 Conference, Austin, Texas, June 2013; v) 17th Workshop on Compilers for Parallel Computing, Lyon, France, July 2013; vi) FEniCS'14 Workshop, Paris, France, June 2014; vii) PDESoft 2014 Workshop, Heidelberg, Germany, July 2014; viii) EuroSciPy 2014 Conference, Cambridge, UK, August 2014.

The PyOP2[1] and Firedrake[2] code repositories are hosted on GitHub, which is also used for issue tracking and code review via pull requests.

1.5 Thesis Outline

Chapter 2 introduces the finite element method, the diverse range of contemporary hardware architectures and programming paradigms used in high performance and scientific computing. In Chapter 3, an overview of related work on high-level abstractions for computational science is given.

Chapters 4 and 5 form the main contribution of this thesis and describe the design and implementation of the PyOP2 domain-specific language for parallel computations on unstructured meshes and the portable finite element framework Firedrake respectively.

An evaluation of the tool chain for a range of different finite element applications is presented in Chapter 6. The thesis concludes with a summary and discussion of the work presented as well as an outlook on planned and potential future work in Chapter 7.

[1] PyOP2 repository: https://github.com/OP2/PyOP2

[2] Firedrake repository: https://github.com/firedrakeproject/firedrake

Chapter 2

Background

This chapter begins with a brief overview of the mathematical theory of the finite element method, followed by an exposition of performance terminology and guidelines, which are used to characterise the different constraints and capabilities of a diverse range of hardware architectures prevalent in high performance and scientific computing. Efficiently using these architectures requires different parallel programming paradigms, which are introduced in the last part of this chapter. Together, these are the foundations built upon in later chapters of this thesis, where the efficient mapping of the finite element method onto different hardware platforms is presented. Related work is discussed in Chapter 3.

2.1 The Finite Element Method

Finite element methods are widely used in science and engineering as a powerful and flexible mechanism for computing approximate solutions of partial differential equations. The finite element method provides a clean mathematical abstraction of a problem that we can reason about and which we can readily express in code. It is particularly suited as a computational method due to the mostly local nature of its operations, which is a very desirable property as will be demonstrated in later chapters.

This section is a brief introduction to the mathematical theory, mainly focussing on the representation of variational forms. Parts of this section are based on Kirby and Logg [2012a], Kirby et al. [2012], Logg et al. [2012b], Ølgaard and Wells [2012], Kirby and Logg [2012b] and adopt the

notation used therein. For a more comprehensive treatment, the reader is referred to mathematical textbooks such as Brenner and Scott [2008].

2.1.1 Variational Problems

Consider a general *linear variational problem* in the canonical form: Find $u \in V$ such that

$$a(v, u) = L(v) \quad \forall v \in \hat{V}, \tag{2.1}$$

where $\hat{V}$ is the *test space* and V is the *trial space*. The variational problem may be expressed in terms of a *bilinear form* a and *linear form* L:

$$\begin{aligned} a &: \hat{V} \times V \to \mathbb{R}, \\ L &: \hat{V} \to \mathbb{R}. \end{aligned}$$

The variational problem is discretised by restricting a to a pair of discrete test and trial spaces: Find $u_h \in V_h \subset V$ such that

$$a(v_h, u_h) = L(v_h) \quad \forall v_h \in \hat{V}_h \subset \hat{V}. \tag{2.2}$$

To solve this *discrete variational problem* (2.2), we make the ansatz

$$u_h = \sum_{j=1}^{N} U_j \phi_j, \tag{2.3}$$

and take $v_{h,i} = \hat{\phi}_i$, $i = 1, 2, \ldots, N$, where $\{\hat{\phi}_i\}_{i=1}^N$ is a basis for the discrete test space $\hat{V}_h$ and $\{\phi_j\}_{j=1}^N$ is a basis for the discrete trial space V_h. It follows that

$$\sum_{j=1}^{N} U_j \, a(\hat{\phi}_i, \phi_j) = L(\hat{\phi}_i), \quad i = 1, 2, \ldots, N.$$

We thus obtain the *degrees of freedom* U of the *finite element solution* u_h by solving a linear system

$$AU = b, \tag{2.4}$$

where

$$\begin{aligned} A_{ij} &= a(\hat{\phi}_i, \phi_j), \quad i, j = 1, 2, \ldots, N, \\ b_i &= L(\hat{\phi}_i). \end{aligned} \tag{2.5}$$

Here, A and b are the discrete operators corresponding to the bilinear

and linear forms a and L for the given bases of the test and trial spaces. The discrete operator A is a – typically sparse – matrix of dimension $N \times N$, whereas b is a dense vector of length N.

The canonical form of a *non-linear variational problem* is as follows: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall v \in \hat{V}, \tag{2.6}$$

where $F : V \times \hat{V} \to \mathbb{R}$ is a *semi-linear* form, known as the *residual form*, with the semicolon splitting the non-linear and linear arguments u and v. Restricting to a pair of discrete trial and test spaces yields a discretised variational problem: find $u_h \in V_h \subset V$ such that

$$F(u_h; v_h) = 0 \quad \forall v_h \in \hat{V}_h \subset \hat{V}. \tag{2.7}$$

The finite element solution $u_h = \sum_{j=1}^{N} U_j \phi_j$ is obtained by solving a non-linear system of equations $b(U) = 0$ with $b : \mathbb{R}^N \to \mathbb{R}^N$ and

$$b_i(U) = F(u_h; \hat{\phi}_i) = 0, \quad i = 1, 2, \ldots, N. \tag{2.8}$$

If the semi-linear form F is differentiable in u, the Jacobian $J = b'$ is given by

$$\begin{aligned} J_{ij}(u_h) &= \frac{\partial b_i(U)}{\partial U_j} = \frac{\partial}{\partial U_j} F(u_h; \hat{\phi}_i) = F'(u_h; \hat{\phi}_i) \frac{\partial u_h}{\partial U_j} = F'(u_h; \hat{\phi}_i) \phi_j \\ J_{ij}(u_h) &\equiv F'(u_h; \phi_j, \hat{\phi}_i). \end{aligned} \tag{2.9}$$

Formally, the Jacobian J is the Gâteaux derivative $dF(u_h; \delta u, v_h)$ in direction δu:

$$dF(u_h; \delta u, v_h) = \lim_{h \to 0} \frac{F(u_h + h\delta u, v_h) - F(u_h, v_h)}{h}. \tag{2.10}$$

Solving the non-linear system with a Newton iteration scheme, the matrix J and vector b are assembled for each iteration to obtain the linear system

$$J(u_h^k)\ \delta U^k = b(u_h^k) \tag{2.11}$$

whose solution δU^k is used to update the solution vector U:

$$U^{k+1} = U^k - \delta U^k, \quad k = 0, 1, \ldots. \tag{2.12}$$

Each iteration can be expressed as a linear variational problem in the canonical form 2.1 since for each fixed u_h, $a = F'(u_h; \cdot, \cdot)$ is a bilinear form and $L = F(u_h; \cdot)$ is a linear form: find $\delta u \in V_{h,0}$ such that

$$F'(u_h; \delta u, v_h) = F(u_h; v_h) \quad \forall v_h \in \hat{V}_h, \tag{2.13}$$

where $V_{h,0} = \{v_h - w_h : v_h, w_h \in V_h\}$. Discretising this form yields the linear system (2.11).

2.1.2 Function Spaces

The term *finite element method* stems from the idea of partitioning the *domain* of interest Ω of spatial dimension d into a *finite set* of disjoint cells $\mathcal{T} = \{K\}, K \subset \mathbb{R}^d$, typically of polygonal shape, forming a *mesh* such that

$$\cup_{K \in \mathcal{T}} K = \Omega.$$

A *finite element* according to Ciarlet [1976] is a *cell* K paired with a *finite dimensional local function space* $\mathcal{P}_K$ of dimension n_K and a *basis* $\mathcal{L}_K = \{\ell_1^K, \ell_2^K, \ldots, \ell_{n_K}^K\}$ for $\mathcal{P}_K'$, the dual space of $\mathcal{P}_K$.

The natural choice of basis for $\mathcal{P}_K$ is the *nodal basis* $\{\phi_i^K\}_{i=1}^{n_K}$, satisfying

$$\ell_i^K(\phi_j^K) = \delta_{ij}, \quad i, j = 1, 2, \ldots, n_K. \tag{2.14}$$

It follows that any $v \in \mathcal{P}_K$ may be expressed by

$$v = \sum_{i=1}^{n_K} \ell_i^K(v) \phi_i^K. \tag{2.15}$$

The degrees of freedom of any function v in terms of the nodal basis $\{\phi_i^K\}_{i=1}^{n_K}$ are obtained by evaluating the linear functionals $\mathcal{L}_K$, which are therefore also known as *degrees of freedom* of the resulting equation system.

Defining a *global function space* $V_h = \text{span}\{\phi_i\}_{i=1}^N$ on Ω from a given set $\{(K, \mathcal{P}_K, \mathcal{L}_K)\}_{K \in \mathcal{T}}$ of finite elements requires a *local-to-global mapping* for each cell $K \in \mathcal{T}$

$$\iota_K : [1, \ldots, n_K] \to [1, \ldots, N]. \tag{2.16}$$

This mapping specifies how the *local degrees of freedom* $\mathcal{L}_K = \{\ell_i^K\}_{i=1}^{n_K}$ are

mapped to *global degrees of freedom* $\mathcal{L} = \{\ell_i\}_{i=1}^N$, given by

$$\ell_{\iota_K(i)}(v) = \ell_i^K(v|_K), \quad i = 1, 2, \ldots, n_K, \tag{2.17}$$

for any $v \in V_h$, where $v|_K$ denotes the restriction of v to the element K.

2.1.3 Mapping from the Reference Element

Finite-element global function spaces as described in Section 2.1.2 are usually defined in terms of a *reference finite element* $\{(\hat{K}, \hat{\mathcal{P}}, \hat{\mathcal{L}})\}$ with $\hat{\mathcal{L}} = \{\hat{\ell}_1, \hat{\ell}_2, \ldots, \hat{\ell}_{\hat{n}}\}$ and a set of invertible mappings $\{\mathcal{G}_K\}_{K \in \mathcal{T}}$ from the reference cell $\hat{K}$ to each cell of the mesh as shown in Figure 2.1:

$$K = \mathcal{G}_K(\hat{K}) \quad \forall K \in \mathcal{T}. \tag{2.18}$$

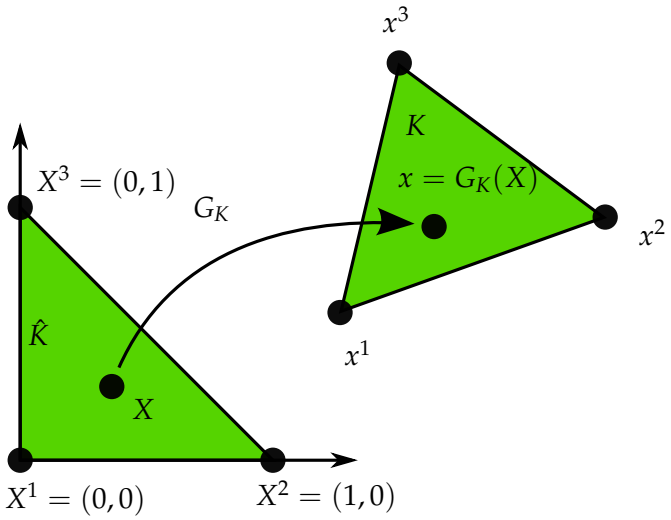

Figure 2.1: Affine map $\mathcal{G}_K$ from a reference cell $\hat{K}$ to a cell $K \in \mathcal{T}$ [adapted from Kirby and Logg, 2012a]

For discretisations in H^1 on simplices, the mapping $\mathcal{G}_K$ is typically *affine* and can be expressed in the form $\mathcal{G}_K(X) = A_T X + b_T$ for some matrix $A_T \in \mathbb{R}^{d \times d}$ and some vector $b_T \in \mathbb{R}^d$. Otherwise the mapping is called *isoparametric* and the components of $\mathcal{G}_K$ are functions in $\hat{\mathcal{P}}$.

2.1.4 The Lagrange Element

The $\mathcal{P}_1$ Lagrange element is in some sense the quintessential finite element. It defines a subspace of the Sobolev space H^1, which requires

piecewise smooth functions on a bounded domain to be C^0 continuous. While the $\mathcal{P}_1$ element uses first order linear polynomial basis functions, the Langrange element can be parametrized for polynomial basis functions of any order q offering higher order approximation properties.

The Lagrange element (P_q) is defined for $q = 1, 2, \ldots$ by

$$\begin{aligned} T &\in \{\text{interval}, \text{triangle}, \text{tetrahedron}\}, \\ \mathcal{V} &= \mathcal{P}_q(T), \\ \ell_i(v) &= v(x^i), \quad i = 1, \ldots, n(q), \end{aligned} \tag{2.19}$$

where $\{x^i\}_{i=1}^{n(q)}$ is an enumeration of points in T defined by

$$x = \begin{cases} i/q & 0 \leq i \leq q, & T \text{ interval}, \\ (i/q, j/q) & 0 \leq i+j \leq q, & T \text{ triangle}, \\ (i/q, j/q, k/q) & 0 \leq i+j+k \leq q, & T \text{ tetrahedron}. \end{cases} \tag{2.20}$$

The number of local degrees of freedom of a Lagrange element corresponds to the dimension of the complete polynomials of degree q on T:

$$n(q) = \begin{cases} q+1, & T \text{ interval}, \\ \frac{1}{2}(q+1)(q+2), & T \text{ triangle}, \\ \frac{1}{6}(q+1)(q+2)(q+3), & T \text{ tetrahedron}. \end{cases} \tag{2.21}$$

Note that the uniform distribution of points $\{x^i\}$ presented above is only one common choice. Different choices are possible as long as boundary points on the exterior of the cell are chosen symmetrically such that those of adjacent cells match to allow C^0 assembly.

Lagrange elements with vector- or tensor-valued basis functions are commonly constructed from a Lagrange element for each component.

2.1.5 The Discontinuous Lagrange Element

Discontinuous Galerkin (DG) finite elements are a typical example of a class of finite element spaces which lie in L^2, however the elements are not C^0 continuous. Such spaces occur e.g. in mixed formulations of the Poisson equation and non-conforming methods, where the desired continuity is imposed weakly. In the case of the DG method, the non-uniqueness

of the solution is modelled by a numerical flux, which is assembled over element facets as interior facet integrals in the weak form.

Weaker coupling between individual elements imposes fewer restrictions on the local basis, allowing different polynomial orders for neighbouring elements. Since all operations are local, discontinuous methods are very amenable to parallelisation and *hp*-adaptivity, where both the characteristic mesh size and the polynomial order of basis functions are varied to achieve a given error tolerance for the least computational cost.

The discontinuous Lagrange element (DG_q) is defined for $q = 0, 1, 2, \ldots$ as given by (2.19), with points in T enumerated by (2.20) and a dimension as specified in (2.21).

2.1.6 $H(\mathrm{div})$ and $H(\mathrm{curl})$ Finite Elements

Spaces occurring in connection with mixed formulations of second-order elliptic problems, porous media flow and elasticity equations often do not fulfil the continuity requirements of $[H^1]^d$ for d-vector fields with $d \geqslant 2$. They do however fall into the Sobolev space $H(\mathrm{div})$, consisting of vector fields for which the components and the weak divergence are square-integrable. $H(\mathrm{div})$-conforming finite element families must have continuous normal components, but each tangential component need not be continuous. Degrees of freedom of $H(\mathrm{div})$-conforming elements usually include normal components on element facets to ensure such continuity. The two most widespread families of $H(\mathrm{div})$-conforming elements are the Raviart–Thomas [Raviart and Thomas, 1977] and Brezzi–Douglas–Marini [Brezzi et al., 1985] elements.

The Sobolev space $H(\mathrm{curl})$ arises frequently in problems associated with electromagnetism. $H(\mathrm{curl})$-conformity requires the tangential component of a piecewise polynomial to be continuous. Therefore, the degrees of freedom for $H(\mathrm{curl})$-conforming finite elements typically include tangential components. Four families of finite element spaces due to Nédélec are widely used and colloquially referred to as *edge elements*.

Nédélec [1980] introduced two families of finite element spaces on tetrahedra, cubes and prisms: one $H(\mathrm{div})$-conforming family and one $H(\mathrm{curl})$-conforming family. These families are known as Nédélec $H(\mathrm{div})$ elements of the *first kind* and Nédélec $H(\mathrm{curl})$ elements of the *first kind*, respectively.

The $H(\mathrm{div})$ elements can be viewed as the three-dimensional extension of the Raviart–Thomas elements.

Nédélec [1986] introduced two more families of finite element spaces: again, one $H(\mathrm{div})$-conforming family and one $H(\mathrm{curl})$-conforming family. These families are known as Nédélec $H(\mathrm{div})$ elements of the *second kind* and Nédélec $H(\mathrm{curl})$ elements of the *second kind*, respectively. The $H(\mathrm{div})$ elements can be viewed as the three-dimensional extension of the Brezzi–Douglas–Marini elements.

A comprehensive overview of these and other common and unusual finite elements is given in Kirby et al. [2012].

2.1.7 Assembly

The discrete operator A from (2.5) is usually computed by iterating over the cells of the mesh and adding the contribution from each local cell to the global matrix A, an algorithm known as *assembly*. If the bilinear form a is expressed as an integral over the domain Ω, we can decompose a into a sum of *element bilinear forms* a_K,

$$a = \sum_{K \in \mathcal{T}} a_K, \tag{2.22}$$

and thus represent the global matrix A as a sum of *element matrices*,

$$A = \sum_{K \in \mathcal{T}} A_i^K, \tag{2.23}$$

with $i \in \mathcal{I}_K$ the index set on the local element matrix

$$\mathcal{I}_K = \prod_{j=1}^{2} [1, \ldots, n_j] = \{(1,1), (1,2), \ldots, (n_1, n_2)\}. \tag{2.24}$$

These *element* or *cell matrices* A^K are obtained from the discretisation of the *element bilinear forms* a_K on a local cell K of the mesh $\mathcal{T} = \{K\}$

$$A_i^K = a_K(\phi_{i_1}^{K,1}, \phi_{i_2}^{K,2}), \tag{2.25}$$

where $\{\phi_i^{K,j}\}_{i=1}^{n_j}$ is the local finite element basis for the discrete function space V_h^j on K. V_h^1 is referred to as $\hat{V}_h$ and V_h^2 as V_h in Section 2.1.1. The

cell matrix A^K is a – typically dense – matrix of dimension $n_1 \times n_2$.

Let $\iota_K^j : [1, n_j] \to [1, N_j]$ denote the local-to-global mapping introduced in (2.17) for each discrete function space V_h^j, $j = 1, 2$, and define for each $K \in \mathcal{T}$ the collective local-to-global mapping $\iota_K : \mathcal{I}_K \to \mathcal{I}$ by

$$\iota_K(i) = (\iota_K^1(i_1), \iota_K^2(i_2)) \quad \forall i \in \mathcal{I}_K. \tag{2.26}$$

That is, ι_K maps a tuple of local degrees of freedom to a tuple of global degrees of freedom. Furthermore, let $\mathcal{T}_i \subset \mathcal{T}$ denote the subset of the mesh on which $\phi_{i_1}^1$ and $\phi_{i_2}^2$ are both non-zero. We note that ι_K is invertible if $K \in \mathcal{T}_i$. We may now compute the matrix A by summing local contributions from the cells of the mesh:

$$\begin{aligned} A_i &= \sum_{K \in \mathcal{T}} a_K(\phi_{i_1}^1, \phi_{i_2}^2) = \sum_{K \in \mathcal{T}_i} a_K(\phi_{i_1}^1, \phi_{i_2}^2) \\ &= \sum_{K \in \mathcal{T}_i} a_K(\phi^{K,1}_{(\iota_K^1)^{-1}(i_1)}, \phi^{K,2}_{(\iota_K^2)^{-1}(i_2)}) = \sum_{K \in \mathcal{T}_i} A^K_{\iota_K^{-1}(i)}. \end{aligned} \tag{2.27}$$

This computation may be carried out efficiently by a single iteration over all cells $K \in \mathcal{T}$. On each cell, the element matrix A_K is computed and then added to the global matrix A as outlined in Listing 1.

Listing 1 Assembly of local element matrices A^K into a global matrix A

```
A = 0
for K ∈ T
    (1) Compute local-to-global mapping ι_K
    (2) Compute element matrix A^K

    (3) Add A^K to A according to ι_K:
    for i ∈ I_K
        A_{ι_K(i)} +≡ A^K_i
    end for
end for
```

2.1.8 Quadrature Representation

A standard approach for evaluating the element matrix A^K on the cell K of spatial dimension d [see Logg, 2007, Chapter 5.1] is known as *quadrature*. It

refers to a summation of the basis functions and their derivatives as given by the variational form evaluated at a set of *quadrature points* and multiplied with suitable *quadrature weights*. Runtime execution of quadrature evaluation can be accelerated by using an affine mapping $\mathcal{G}_K : \hat{K} \to K$ with pre-tabulated basis functions and derivatives at the quadrature points of the reference cell $\hat{K}$ as described in Section 2.1.3.

The element matrix A^K for Poisson's equation on cell K is computed as

$$\begin{aligned} A^K &= \int_K \nabla \phi_i^K \cdot \nabla \phi_i^K \, \mathrm{d}x \\ &\approx \sum_{k=1}^{N_q} w_k \nabla \phi_i^K(x^k) \cdot \nabla \phi_i^K(x^k) |\det \mathcal{G}_K'(x^k)|, \end{aligned} \tag{2.28}$$

with quadrature points $\{x^k\}_{k=1}^{N_q} \in K$ and corresponding quadrature weights $\{w_k\}_{k=1}^{N_q}$ scaled such that $\sum_{k=1}^{N_q} w_k = |\hat{K}|$. For polynomial basis functions, the quadrature points can be chosen such that the approximation (2.28) is exact if $\mathcal{G}_K$ is affine. Note that test and trial function are chosen from the same function space V_h.

The local basis functions $\{\phi_i^K\}_{i=1}^{n_K}$ on K can be generated from the basis $\{\Phi_i\}_{i=1}^{n_0}$ on the reference cell $\hat{K}$ as $\phi_i^K = \Phi_i \circ \mathcal{G}_K^{-1}$ and the coordinates are given as $x^k = \mathcal{G}_K(X^k)$. Hence the evaluation of the gradients of the basis functions is a matrix-vector product

$$\nabla_x \phi_i^K(x_k) = (\mathcal{G}_K')^{-T}(x_k) \nabla_X \Phi_i(X_k) \tag{2.29}$$

for each quadrature point x_k and each basis function ϕ_i^K.

Each gradient is computed in $N_q \cdot n_0 \cdot d^2$ multiply-add pairs and the total cost for the element matrix computation amounts to $N_q \cdot n_0 \cdot d^2 + N_q \cdot n_0^2 \cdot (d+2) \sim N_q \cdot n_0^2 \cdot d$ multiply-add pairs, ignoring the cost of computing the mapping $\mathcal{G}_K$, its determinant, and the inverse of its Jacobian $\mathcal{G}_K'$. Note that this cost can be significantly reduced by applying optimizations such as loop invariant code motion, common subexpression elimination, and precomputation of constants as detailed in Ølgaard and Wells [2010].

2.1.9 Tensor Representation

According to Kirby and Logg [2006, 2007], Logg [2007], the evaluation of the element matrix A^K can in many cases be accelerated by precomputing a constant *reference tensor* A^0 on the *reference element* and contracting with a *geometry tensor* G_K depending on the geometry of the current cell K.

We only consider the case where $\mathcal{G}_K : \hat{K} \to K$ is an affine mapping and take the element matrix A^K for Poisson's equation as an example. Having

$$A^K = \int_K \nabla \phi_i^K \cdot \nabla \phi_i^K \, \mathrm{d}x = \int_K \sum_{\beta=1}^{d} \nabla \frac{\partial \phi_i^K}{\partial x_\beta} \nabla \frac{\partial \phi_i^K}{\partial x_\beta} \, \mathrm{d}x \tag{2.30}$$

with spatial dimension d and local basis functions $\{\phi_i^K\}_{i=1}^{n_K}$, yields, with a change of variables to the basis $\{\Phi_i\}_{i=1}^{n_0}$ on the reference cell $\hat{K}$:

$$A^K = \int_{\hat{K}} \sum_{\beta=1}^{d} \sum_{\alpha_1=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial \Phi_i}{\partial X_{\alpha_1}} \times \sum_{\alpha_2=1}^{d} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \frac{\partial \Phi_i}{\partial X_{\alpha_2}} \, |\det \mathcal{G}_K'| \, \mathrm{d}X. \tag{2.31}$$

Due to the affine mapping $\mathcal{G}_K$, $\det \mathcal{G}_K'$ and the derivatives $\frac{\partial X}{\partial x}$ are constant:

$$\begin{aligned} A^K &= |\det \mathcal{G}_K'| \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_{\hat{K}} \frac{\partial \Phi_i}{\partial X_{\alpha_1}} \frac{\partial \Phi_i}{\partial X_{\alpha_2}} \, \mathrm{d}X \\ &= \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} A^0_{i_\alpha} G_K^\alpha, \end{aligned} \tag{2.32}$$

with

$$A^0_{i_\alpha} = \int_{\hat{K}} \frac{\partial \Phi_i}{\partial X_{\alpha_1}} \frac{\partial \Phi_i}{\partial X_{\alpha_2}} \, \mathrm{d}X \tag{2.33}$$

and

$$G_K^\alpha = |\det \mathcal{G}_K'| \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta}. \tag{2.34}$$

The element matrix can hence be decomposed as

$$A^K = A^0 : G_K, \tag{2.35}$$

with : the Frobenius product of tensors. A^0 is the constant *reference tensor* that does not depend on the cell K and may be precomputed before the

assembly of A, and G_K the *geometry tensor* that needs to be computed for each cell K.

Tabulating A^K involves d^3 multiply-add pairs for computing the rank two geometry tensor G^K and $n_0^2 d^2$ multiply-add pairs for the tensor contraction. The total computational cost can therefore be estimated as $d^3 + n_0^2 d^2 \sim n_0^2 d^2$ and compared to the cost $N_q n_0^2 d$ for quadrature. This results in a speedup of roughly N_q/d, which may be significant particularly for higher order elements that require a large number of quadrature points.

2.1.10 Linear Solvers

The *linear system* (2.4) arising from a variational problem of the form (2.1) contains a generally *sparse matrix* A to be solved for the vector of unknowns U. This kind of linear system also appears in each iteration of a non-linear scheme such as Newton's method in (2.11).

Efficient solvers include the family of *Krylov-type iteration methods*, such as the *conjugate gradient* (CG) iteration for symmetric positive-definite matrices [Hestenes and Stiefel, 1952], and the *generalized minimal residual* method (GMRES) [Saad and Schultz, 1986], which only require matrix-vector products, but not the matrix A in explicit form. *Relaxation* methods have been greatly superseded by *multi-grid* methods [Brandt, 1977], acting on a hierarchy of grids and solving in near linear time. *Direct methods* compute an *LU* factorisation using *Gaussian elimination* [Davis, 2004].

Solvers often have to deal with ill-conditioned matrices and hence the use of *preconditioners* can significantly improve convergence, especially for Krylov methods, whose convergence rate is directly related to the *condition number* of the matrix. A (left-sided) preconditioner transforms the linear system (2.4) into

$$P^{-1}AU = P^{-1}b, \tag{2.36}$$

where P^{-1} is chosen to be a good approximation of A^{-1}, but at the same time cheap to compute. The inverse A^{-1} is the perfect preconditioner, resulting in a condition number of 1 for the preconditioned system, but requires having already solved the problem.

Common choices for preconditioners given in Kirby and Logg [2012a] are classical *relaxation methods*, such as Gauss-Seidel, or *incomplete factorizations*, such as ILU (incomplete LU factorization). Multi-grid methods can

also serve as a powerful preconditioner. For certain problem classes, there exist more advanced *physically-based* preconditioners, which take into account properties of the differential equations being solved.

2.1.11 Action of a Finite Element Operator

Krylov methods do not require the matrix A to be explicitly available, only the *matrix-vector product* AU, which can be treated as a "black box" in many implementations. Thus, they qualify for use with so-called *matrix-free* methods and allow problems to be solved without ever explicitly computing or storing the matrix A. This section gives a brief overview of what is presented in Kirby et al. [2004].

Assembly of the sparse matrix A is replaced by repeated assembly of a vector $v = AU$, the *action* of the *operator* A on the given vector $U \in \mathbb{R}^N$:

$$(AU)_i = \sum_{j=1}^{N} A_{ij} U_j = \sum_{j=1}^{N} a(\phi_i^1, \phi_j^2) U_j = a(\phi_i^1, \sum_{j=1}^{N} U_j \phi_j^2) = a(\phi_i^1, u_h). \quad (2.37)$$

That is, the application of the matrix A on the coefficient vector U is given by the action $\mathcal{A}$ of the bilinear form evaluated at the finite element approximation $u_h = \sum_{j=1}^{N} U_j \phi_j^2$:

$$(AU)_i = \mathcal{A}(a, u_h)(\phi_i^1). \quad (2.38)$$

Initially, all entries of v are set to zero and are then accumulated by looping over all elements $K \in \mathcal{T}$ with $\mathcal{T}$ the decomposition of the domain into elements, and computing

$$v_{\iota_K(i)} \overset{+}{=} \sum_{j=1}^{|\iota_K|} A_{i,j}^K u_{h,\iota_K(j)} \quad i = 1, \ldots, |\iota_K|. \quad (2.39)$$

This can be written as a matrix-vector product for each element K

$$v_{\iota_K} \overset{+}{=} A^K u_{h,\iota_K} \quad (2.40)$$

where ι_K denotes the set of global indices obtained from the local-to-global mapping and A^K is the local element tensor of element K in the quadrature or tensor product representation as described in Sections 2.1.8 and 2.1.9.

The computational cost of (2.40) for Poisson's equation is $d^2|\iota_K|^2$ multiply-add pairs per element, with d the spatial dimension. An additional $3|\iota_K| + d(d+1)/2$ memory reads and writes are required, if the symmetry of G_K is exploited, not counting storing A^K. Note that due to the accumulation v needs to be both read from and written to memory.

The matrix-free approach has the disadvantage that preconditioners commonly used with Krylov methods usually involve manipulations of A and hence cannot be readily applied in this case. However, preconditioners can be adapted for the matrix-free approach if supported by the Krylov solver.

2.2 Contemporary Parallel Hardware Architectures

The last decade has seen processor clock frequencies plateau and the number of cores per chip increase dramatically. GPUs programmable for general purpose computations (GPGPU) have entered and established themselves in the high performance computing market. Unconventional hardware architectures such as the Cell Broadband Engine Architecture and the Intel Xeon Phi have been developed with the intent of finding a sweet spot in terms of power consumption and achievable peak performance.

Solving grand challenge problems at reasonable performance requires taking into account and tuning for characteristics of the hardware. Future proof software design requires abstracted systems, insensitive to the rapidly changing hardware landscape, which is more complex and diverse than it has ever been. This section gives an overview of contemporary multi- and many-core hardware architectures.

2.2.1 Multi-core and Many-core Architectures

Contemporary architectures are commonly classified according to number and complexity of the cores as either multi- or many-core, though the distinction is not very clear cut and there has been a recent trend of convergence. Caches and complex logic take up most transistors and space on a multi-core chip, whereas the arithmetic logic units (ALUs) responsible for integer and floating point computations occupy a rather small portion. Most of the area of a many-core chip is devoted to execution units

rather than caches. In the following, characteristics of both architectures are compared and significant differences highlighted.

Multi-core architectures

Few, complex cores Multi-core CPUs contain a small number, typically two to eight, of complex cores designed to deliver results for general purpose, serial workloads with minimum latency.

Large, hardware managed caches CPUs have a typically deep hierarchy of on-chip caches (L1, L2, L3) which are automatically managed by the hardware and are most efficiently used by computations with good *spatial* and *temporal* locality. Caches may be shared and *cache coherency protocols* commonly ensure consistency between caches visible by different CPU cores.

Instruction Level Parallelism Complex logic such as *prefetching*, *branch prediction* and *out-of-order execution* is used to process instructions in an order that avoids stalls due to main memory accesses and further aids in minimising latency, provided there are enough instruction in the pipeline.

SIMD Modern CPUs achieve their peak arithmetic performance only when using floating point vector registers via single instruction multiple data (SIMD) intrinsics on operands of up to 256-bit length.

Many-core architectures

Many, simple cores Many-core devices have many throughput-optimised, simple cores capable of running hundreds or thousands of concurrent threads. The comparatively small caches can be partly managed by the programmer to collaboratively load data shared between threads.

Limited, shared resources On-chip resources such as registers and caches are commonly shared between a number of resident threads such that there is a trade-off between the number of concurrently executed threads and the amount of resources used by each.

Latency hiding For highly parallel applications, latency incurred by memory accesses is hidden through zero overhead context switches between a large number of concurrent threads in-flight at the same time.

Offloading Many-core devices are often designed as accelerators with dedicated memory, connected via PCIe and controlled from a CPU host pro-

cess offloading computations by launching kernels on the device.

2.2.2 Contemporary GPU Architectures

Kepler [NVIDIA, 2012] is the most recent generation of NVIDIA GPU architectures, integrating up to 15 streaming multiprocessors (SMX) on a single chip, each with 192 single-precision and 64 double precision floating point units capable of one fused multiply-add (FMA) per cycle, 32 Special Function Units (SFUs) and 32 Load/Store units (LD/ST). An SMX has access to a register file with 65,536 32-bit registers, 48kB of read-only data cache and 64kb of on-chip memory that is split between a hardware-managed L1 cache and a software-managed shared memory. All SMX units share a common L2 cache and up to 12GB of global DRAM.

Parallel computations are launched as *kernels* by the controlling CPU process on a given number of *threads*, batched in groups of 32 called *warps* and organised in a *grid* of thread *blocks*. NVIDIA GPUs use the *single-instruction multiple-thread* (SIMT) execution model, where all threads of a warp execute the same instruction. That means a warp must execute each branch of any conditional where at least one of its threads participates. If all threads take the same branch, only that particular branch is executed. Otherwise, the warp is called *divergent* and each relevant branch is executed in sequence with all threads not participating *masked out*, which means no results are written, operands read or addresses evaluated.

Blocks of threads are executed independently of each other and each block has a fixed affinity to an SMX for its lifetime. Warps of a given block have access to the *shared memory* of that SMX, which is explicitly managed by the programmer. It can be used to collaboratively load data from global memory, which can then be accessed with almost the same low latency as the register file. Threads within a block can also synchronise on a barrier. Apart from kernel launches, there are no global barriers and no way of communicating and synchronisation between threads of different blocks, since these need to be able to execute independently.

Access to global memory is cached in L2 in 128 byte cache lines aligned to memory addressed that are multiples of 128. Memory accesses to L2 are served in 128 byte transactions, where transactions targeting global memory are 32, 64 or 128 bytes depending on the size of the word accessed

by each thread in a warp and the access pattern. In the worst case this can mean a 32-byte transaction for a single byte read or written. In the best case, that is when threads read consecutive words from memory, also known as *coalesced access*, this means a four byte word per thread per transaction is transferred.

In contrast, global memory transactions on the older Tesla architecture, which does not have an L2 cache, are scheduled per *half-warp*. To achieve coalesced access, the 16 threads of a half-warp must read 16 consecutive words of four, eight or 16 bytes, which must all lie within the same 64-, 128- or 256-byte aligned segment. If this requirement is not met, 16 separate 32-byte transactions are issued.

2.2.3 Intel Xeon Phi (Knights Corner)

The Intel Xeon Phi coprocessor [Reinders, 2012] code named "Knights Corner" is an x86 SMP-on-a-chip running Linux that connects to the host system via the PCIe bus, much like a GPU. Its 61 in-order dual issue 64-bit processor cores support four concurrent hardware threads and have access to 512 bit wide SIMD registers and 512KB of local L2 cache. Caches are coherent across the entire coprocessor. The cores are interconnected by a bidirectional ring bus which also connects the eight memory controllers in four groups of two each.

Despite being a coprocessor, the Xeon Phi can be programmed using MPI, OpenMP or OpenCL much like a CPU due to its x86 architecture.

2.2.4 Performance Terminology

Common terminology used to characterise the performance of hardware platforms and algorithms is introduced in the list below.

Performance bottlenecks Any optimisation effort should be preceded by profiling to determine the bottleneck of the problem under consideration. The three common bottlenecks are *floating point operations*, *memory bandwidth* and *memory access latency*.

Machine Balance (MB) The machine balance of an architecture is commonly defined as the ratio of peak floating point operations per cycle to peak memory operations per cycle for single or double precision floating point operands [McCalpin, 1995]. In other words it is the number of floating

point operations needed per memory word read or written to be able to saturate the compute units of the machine.

Algorithmic Balance (AB) Similar to the machine balance, the algorithmic balance is the ratio of floating point operations to memory operations of an algorithm, that is the number of floating point operations performed per word of memory read or written.

Compute limited An application is said to be compute limited if it is bound by the available floating point operation throughput of the architecture. An algorithmic balance greater than the machine balance may indicate compute boundedness.

Bandwidth limited An application that saturates the device's memory bandwidth is said to be bandwidth limited. This is the common case and indicated by a machine balance higher than the algorithmic balance.

Latency limited An application is latency limited if it is unable to hide memory access latency with computation. This may be caused by an insufficient degree of parallelism for the chosen architecture, poor cache performance or unsuitable memory access patterns.

2.2.5 Performance Considerations

As outlined in the previous sections and shown in Table 2.1, different contemporary hardware platforms differ quite significantly in their characteristic specifications such as floating point performance, memory bandwidth, cache size and hierarchy and memory access latency. As a consequence they are more or less well suited for certain kinds of algorithms and applications and require different approaches, paradigms and algorithmic considerations when programming them. Some overarching observations and considerations are outlined in this section.

Considerations for multi-core CPUs

- CPUs require a moderate amount of fairly coarse-grained data or task level parallelism with at least one thread per physical core. For simultaneous multithreading (SMP) architectures it can be, but not always is, beneficial to launch more threads, e.g. a thread per virtual core.

Architecture	Cores	Ops/cycle	Clock MHz	GFlop/s	BW GB/s	MB
NVIDIA Kepler K40 (SP)	2880	2 (FMA)	745	4291.2	288	60
NVIDIA Kepler K40 (DP)	960	2 (FMA)	745	1430.4	288	40
AMD Hawaii XT (SP)	2816	2 (FMA)	1000	5632	320	70
AMD Hawaii XT (DP)	352	2 (FMA)	1000	704	320	18
Intel Xeon Phi (SP)	61	16 (AVX)	1238	2416	352	14
Intel Xeon Phi (DP)	61	8 (AVX)	1238	1208	352	14
Intel Xeon E3-1285 (SP)	4	32 (AVX2, FMA)	3600	460.8	25.6	72
Intel Xeon E3-1285 (DP)	4	16 (AVX2, FMA)	3600	230.4	25.6	72

Table 2.1: Characteristic specifications of contemporary hardware architectures: number of processor cores, arithmetic throughput per cycle per core, processor clock frequency, peak arithmetic throughput, peak memory throughput and machine balance (MB) for single (SP) and double precision (DP). Note the performance penalty for using DP of factor 3 and 8 for the NVIDIA and AMD GPU architectures, which also translates to a lower machine balance. For the Intel architectures it is the same since the peak performance for DP is half that of SP but also only half the number of DP words are transferred over the memory bus.

- Large caches allow comparatively large per-thread local working set sizes (in the order of MBs).
- Saturating the multiple ALUs per core requires a sufficient degree of instruction-level parallelism (ILP) in the workload of each thread.
- Achieving peak floating point performance is only possible when making full use of the SIMD vector registers and ALUs. Optimising a compute bound code for vectorisation by a vectorising compiler or using SIMD intrinsics is therefore crucial.
- Memory bound applications may be able to saturate the comparatively low memory bandwidth already using a subset of the cores.
- Caches are automatically managed and often local to a core. Memory accesses need to be optimised for the cache hierarchy, where cache usage is optimal if all data on a given cache line is used by a given thread (*spatial locality*) before the cache line is evicted and it is not necessary to load the same data again at a later time (*temporal locality*).
- Data accessed by a given thread should be stored contiguously in memory, a layout often called array-of-structures (AoS).
- On multi-socket nodes it is crucial to account for non-unified memory access (NUMA), to ensure that memory is allocated local to the socket from where it is accessed and pin threads to CPU cores.

Considerations for many-core accelerators

- GPUs and other many-core platforms require a high degree of fine-grained parallelism, often thousands of threads, to saturate the compute resources and to hide memory access latency.
- On chip local memory and registers are a shared and scarce resource, which requires per-thread local working set sizes to be kept small to allow for a large number of resident threads and high device occupancy. If space runs out, registers may spill to slow global device memory.
- Caches are shared by all threads resident on a given multiprocessor (compute unit) and a cache line is read/written simultaneously by all threads on a warp (wavefront) with each thread accessing a single four or eight byte word.
- Data accessed by a given warp (wavefront) should be stored interleaved in memory to enable coalesced access, where threads with consecutive IDs access consecutive words, a layout called structure-of-arrays (SoA).
- Manually managed shared memory allows threads in a warp (wavefront) to collaboratively stage data that is not suitably laid out in global memory for faster on-chip access.
- Divergent code paths should be kept to a minimum due to the lock-step instruction execution of an entire warp (wavefront) of threads.
- The speed of data transfers across the PCIe bus from host to device is only a fraction of the on-device memory bandwidth and therefore these transfers need to be kept to a minimum. It can be beneficial to (re)compute data on the device and save the transfer cost even though the same computation could be performed more efficiently on the host.

2.3 Programming Paradigms for Many-core Platforms

GPUs and other accelerators have a reputation of being difficult to program and often require very different data structures and algorithms to achieve good performance. This presents a barrier to adoption, which is best overcome by raising the level of abstraction as described in Chapter 4. In this section, the CUDA and OpenCL programming models for accelerators and PGAS languages for parallel computations are introduced.

Listing 2.1: A *saxpy* kernel in *C*

```
void saxpy(int n, float *c, float *a, float *b, float alpha) {
  for (int i=0; i<n; ++i)
    c[i] = a[i] + alpha * b[i];
}
// Call for vectors of length 10000
saxpy(10000, c, a, b, alpha);
```

Listing 2.2: A *saxpy* kernel in *CUDA*

```
__global__ void saxpy(int n, float *c, float *a, float *b,
                      float alpha) {
  if (threadIdx.x < n)
    c[threadIdx.x] = a[threadIdx.x] + alpha * b[threadIdx.x];
}
// Launch 1 thread block with 10000 threads
saxpy<<<1, 10000>>>(10000, c, a, b, alpha);
```

2.3.1 NVIDIA Compute Unified Device Architecture (CUDA)

NVIDIA describes its *Compute Unified Device Architecture (CUDA)* [NVIDIA, 2013] as "a general purpose parallel computing platform and programming model". The term CUDA is often used to refer to *CUDA C*, an extension to the C programming language, allowing programmers to leverage NVIDIA GPUs for general-purpose computations. CUDA C adds keywords to annotate functions as kernels and a notation to specify the launch configuration to invoke a kernel with, that is the number and size of thread blocks. Blocks can be declared as logically 1D, 2D or 3D and arranged in a grid that can itself be 1D or 2D.

Inside the kernel function, a thread has access to this launch configuration via special keywords to query its thread ID within the block, the block's ID within the grid, as well as the block and grid dimensions.

As an illustrative example we consider a *saxpy* kernel performing the computation $\vec{c} = \alpha\vec{a} + \vec{b}$, which frequently occurs in numerical algorithms. It computes the sum of two vector operands of which the first is multiplied by a scalar and stores the result in a third vector. Listing 2.1 shows the C implementation. The CUDA implementation is shown in Listing 2.2, where the keyword `__global__` annotates the function as a kernel. Most noticeably, there is no `for` loop in the CUDA kernel, since the kernel is launched in parallel for `n` threads, each computing a single result given by

their thread ID. It would even be possible to omit passing in the vector length `n` to the kernel and have it only be implicitly given by the launch configuration.

The thread indexing used in Listing 2.2 assumes the thread block shape is 1D and there is only a single thread block launched. To allow launching an arbitrary number of 1D blocks would require the current index to be computed as `blockIdx.x * blockDim.x + threadIdx.x`. Both grid and block shapes can be multi-dimensional, in which case the additional index dimension for the grid is accessed as `blockIdx.y` and those for the block as `threadIdx.y` and `threadIdx.z` respectively.

2.3.2 Open Computing Language (OpenCL)

OpenCL [Stone et al., 2010] is an industry standard for task-parallel and data-parallel heterogeneous computing managed by the Khronos Group [Khronos, 2013] and defines an API with a set of core functionality supported across all different types of devices and optional vendor extensions. An OpenCL application is guaranteed to be portable across all supported devices by different vendors implementing the OpenCL runtime and drivers. Portability only guarantees correctness, not performance, and it is unlikely that an application tuned for a particular architecture will achieve satisfactory performance on a different device.

In the OpenCL platform model a *host* controls one or several *computational devices* which contain one or more *compute units* (CUs) composed of one or more *processing elements* (PEs) executing computations in a SIMD fashion. Within a CU, instructions execute in lock-step. The control flow is called *converged* if all PEs run the same instruction stream and *diverged* otherwise.

Computation is done inside a *kernel* within a *context* bound to a device and managed by the host through a *command queue*, where work occurs through *work items* organised into *work groups*. The environment defined by a context includes *devices*, *kernel objects*, *program objects* and *memory objects*. Kernels are normally passed to the OpenCL runtime as strings and just-it-time (JIT) compiled for the target device at runtime. This allows taking advantage of specific hardware and software features of the target device without having to recompile the application itself.

Listing 2.3: A *saxpy* kernel in *OpenCL*

```
__kernel void saxpy(cl_int n,
                    __global float *c, __global float *a,
                    __global float *b, float alpha) {
  int idx = get_global_id(0);
  if (idx < n)
    c[idx] = a[idx] + alpha * b[idx];
}
```

OpenCL distinguishes four *memory regions*: *global memory* is visible to all devices in a given context and contains a read-only region of *constant memory*, whereas *local memory* is local to a work group and *private memory* private to a work item. These address spaces are logically disjoint, but may share the same physical memory depending on the implementation for a given platform.

CUDA	**OpenCL**
shared memory	local memory
local memory	private memory
thread	work item
thread block	work group
streaming multiprocessor (SMX)	compute unit (CU)
stream processor (SP)	processing element (PE)

Table 2.2: Concepts and terminology in CUDA and OpenCL

Many concepts and terms in CUDA have a one-to-one equivalence in OpenCL as shown in Table 2.2. An OpenCL version of the SAXPY kernel from Listing 2.1 is shown in Listing 2.3. The syntax for annotating a kernel in OpenCL C is very similar to that of CUDA C. OpenCL providing a lower level API than CUDA, the process of launching a kernel from the host code is significantly more complex and therefore omitted from Listing 2.3.

2.3.3 Partitioned Global Address Space (PGAS) Languages

Partitioned Global Address Space (PGAS) is a Single Program Multiple Data (SPMD) programming model where each process or thread owns a partition of a globally addressable memory space. PGAS languages aim to offer shared memory programming abstractions with locality and control

comparable to message passing.

Popular implementations are Unified Parallel C (UPC) [Yelick et al., 2007], an explicitly parallel extension of ISO C, Co-Array Fortran (CAF), a Fortran 95 language extension [Coarfa et al., 2005] and Titanium [Yelick et al., 2007], a scientific computing dialect of Java. In all three cases, source-to-source translation is used to turn a PGAS programme into ISO C (UPC/Titanium) or Fortran 90 (CAF), augmented with communication calls into a runtime (GASnet for UPC/Titanium and ARMCI for CAF).

An important difference to message passing as implemented in MPI is the use of one-sided communication, where a put or get message contains a memory address and payload. Instead of having to match a message tag with a pending receive operation at the target, the communication runtime can directly access the remote processes' memory, typically with a hardware supported RDMA or shared memory operation.

2.4 Conclusions

The finite element method has been introduced as a clean mathematical abstraction for computing approximate solutions of partial differential equations, which is amenable to parallel computations due to the mostly local nature of its operations. A range of contemporary multi- and many-core hardware platforms have been presented, which differ vastly in characteristic specifications such as the number of cores, the peak arithmetic and memory throughput and the degree of concurrency required to achieve good utilisation of the device. As outlined in the performance considerations presented, obtaining good performance commonly requires considerable low-level optimisation and tuning efforts specifically taylored to each individual target architecture using one of the different programming paradigms described. This presents a significant barrier to portability and motivates the design of higher-level frameworks which abstract from architecture-specific characteristics and optimisations.

The material covered in this chapter serves as a foundation for the design of PyOP2 and Firedrake described in Chapters 4 and 5 and the experiments presented in Chapter 6. It also provides background relevant to the discussion of related work in the following chapter.

Chapter 3

High-level Abstractions in Computational Science

In this chapter, an overview of related work on different approaches to abstracting problems in computational science with a focus on finite element frameworks is given, ranging from traditional libraries to domain-specific languages. Some of these have inspired the design of PyOP2 and Firedrake, described in Chapters 4 and 5, or are even used as components.

3.1 Library-based Approaches

Scientific software is traditionally implemented in the form of libraries in Fortran, C or C++. In this section, a number of established frameworks are presented, which take different approaches of abstracting the solution of partial differential equations and the finite element method.

3.1.1 Portable, Extensible Toolkit for Scientific Computation (PETSc)

PETSc [Balay et al., 1997] is a library and tool kit for building scientific applications primarily focused on the scalable parallel solution of partial differential equations and regarded by many as the de facto standard for sparse linear algebra. PETSc is built on top of MPI for distributed parallel computations and provides data structures and routines that can be used at an abstract level without having to write low-level message-passing

code or manage what portion of the data is stored on each process. Communication is automatically managed in an efficient way by overlapping with computation and optimising repeated communication patterns while allowing the user to aggregate data for subsequent communication and dictate when communication can occur.

PETSc provides modules for index sets (`IS`) with support for permutations and renumbering, vector (`Vec`) and matrix (`Mat`) operations, distributed mesh data management (`DM`), Krylov subspace methods (`KSP`) and preconditioners (`PC`), including multigrid and sparse direct solvers, nonlinear solvers (`SNES`) and time stepping (`TS`) for ordinary differential equation (ODE) and differential algebraic equation (DAE) integrators. PETSc interoperates with a number of third-party libraries such as Hypre [Falgout et al., 2006], Trilinos [Heroux et al., 2005], MUMPS [Amestoy et al., 2001] and UMFPACK [Davis, 2004].

While implemented in C for portability, PETSc follows object-oriented design principles. Its data structures are defined by abstract interfaces and objects are opaque handles that can represent different implementations, which may be chosen at runtime. Application code is written against a unified API independent of the concrete instances of data structures.

Most of PETSc's functionality is exposed to Python via the petsc4py [Dalcin et al., 2011] interface implemented in Cython [Behnel et al., 2011]. At very little runtime overhead, petsc4py provides access to data structures and routines through a high-level "pythonic" interface.

Unstructured Meshes (DMPlex)

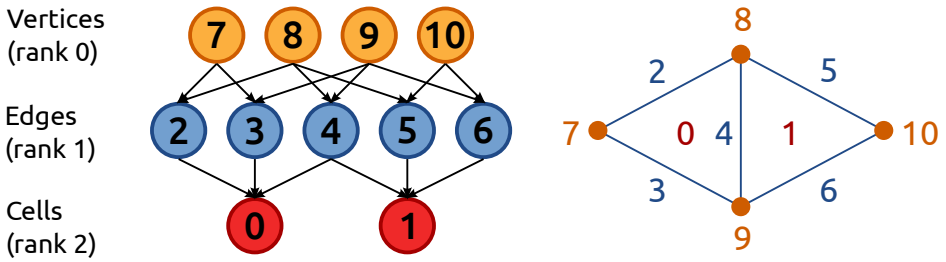

Figure 3.1: Hasse diagram of the partially ordered set representing an unstructured mesh

The PETSc DMPlex module [Knepley, 2013] provides a representation

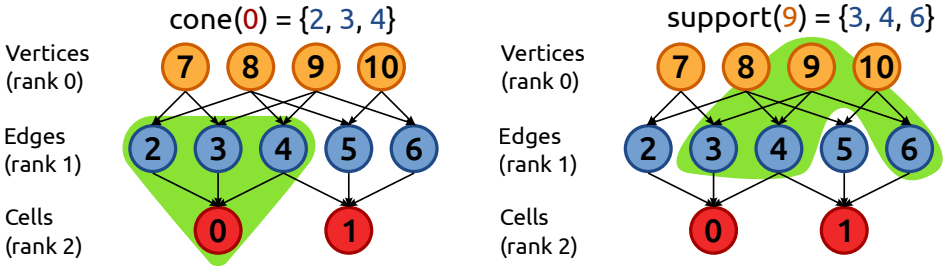

Figure 3.2: Cone (left) and support (right) of an entity in a DMPlex mesh

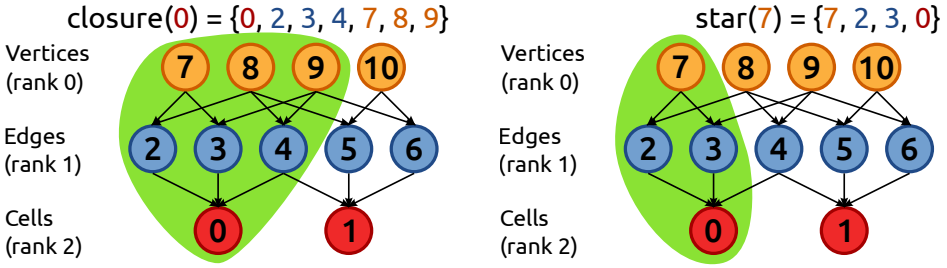

Figure 3.3: Closure (left) and star (right) of an entity in a DMPlex mesh

of a distributed unstructured mesh as a graded partially ordered set, implemented as a directed acyclic graph (DAG). This DAG only stores the mesh topology, whereas the geometry is represented as a mesh function. The visualisation of such a mesh as a Hasse diagram is illustrated in Figure 3.1. The set of all entities of the same rank or grade is called a *stratum*. Entities are numbered by stratum with the highest rank numbered first.

An entity may be any mesh element and DMPlex makes no explicit references to element types. Operations are composed of two basic operations. The *cone* of an entity are the adjacent elements on the rank below and its dual operation, the *support* are the adjacent elements on the rank above, shown in Figure 3.2. The *transitive closure* of an entity is its cone recursively continued across all lower ranks and its dual, the *star*, is the support recursively continued across all higher ranks, given in Figure 3.3.

Meshes can be created either using primitive operations, by setting the cone and support of each mesh element, or by reading from file in common formats such as Gmsh [Geuzaine and Remacle, 2009], CGNS [Poirier et al., 1998] and EXODUS [Mills-Curran et al., 1988]. DMPlex can partition and distribute an existing mesh and supports renumbering by permuting mesh elements.

3.1.2 deal.ii: A General-Purpose Object-Oriented Finite Element Library

Designed as a general-purpose toolkit for finite element applications, deal.ii [Bangerth et al., 2007] is a C++ class library, allowing dimension independent code by using template parameters to define the space dimension. Abstractions are provided for meshes, with support for adaptive refinement and coarsening, degrees of freedom associated with finite element spaces, linear algebra and interfaces for grid generators and visualisation.

Given a `Triangulation` describing the mesh and a `FiniteElement` associating degrees of freedom with vertices, faces and cells, the `DoFHandler` provides a global enumeration of DOFs. Each cell of the mesh may use a different `FiniteElement`, allowing *hp* adaptivity. A range of iterative solvers as well as interfaces to PETSc [Balay et al., 1997], and sparse direct solvers are provided to solve linear systems. Saddle-point problems can be efficiently solved using block or Schur complement preconditioners. Shared memory parallelisation is supported by the library, whereas distributed memory parallelisation needs to be implemented by the user on top of PETSc.

3.1.3 DUNE: Distributed and Unified Numerics Environment

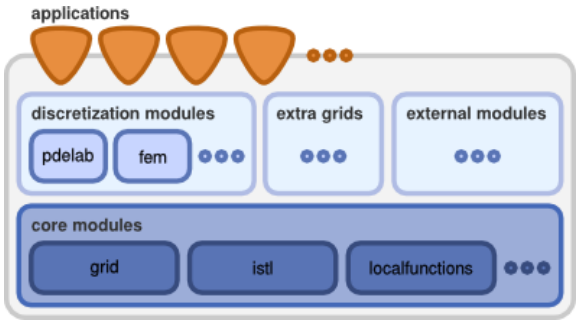

Figure 3.4: Modular design of DUNE [from DUNE Team, 2014]

DUNE [DUNE Team, 2014], the Distributed and Unified Numerics Environment, is a modular C++ template library for solving partial differential equations (PDEs) with grid-based methods. The core module DUNE-Grid [Bastian et al., 2008b,a] is managing the topology of an – optionally – distributed mesh alongside adaptive refinement, coarsening and the rebalancing of the variable work load post adaptation. Templated iterative

solvers are implemented in the DUNE-ISTL core module [Blatt and Bastian, 2007]. Built on top of the core modules is DUNE-FEM [Dedner et al., 2010], an implementation of grid-based discretisation schemes suitable for finite element and finite volume methods.

Template metaprogramming is used to achieve good performance while providing a clean and expressive interface for the programmer. Continuous functions are represented by the parametrised `Function<FunctionSpace>`, their discretised counterparts by the `DiscreteFunction<DiscreteFunctionSpace>` class. Mappings between function spaces are derived from the base class `Operator` or its specialisation `LinearOperator` for linear discrete operators[1]. A singleton class `DofManager` is used to manage degrees of freedom stored on a grid, which does not hold any data itself.

3.1.4 Fluidity

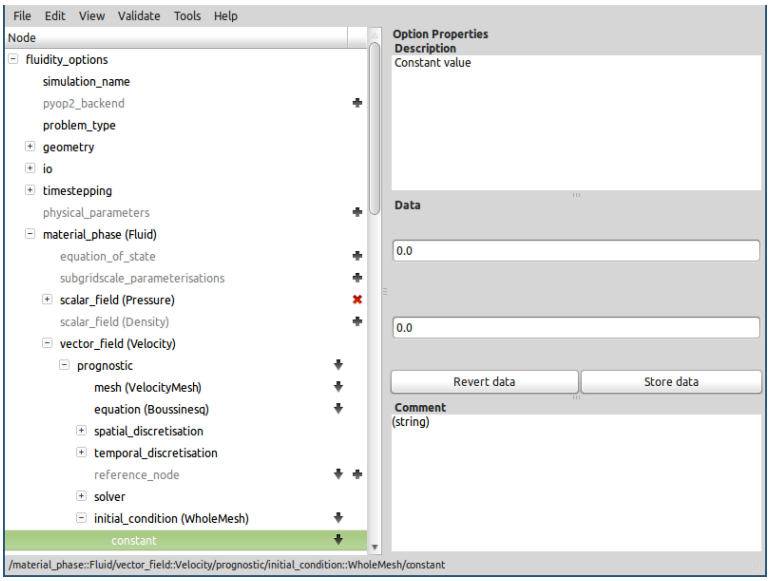

Figure 3.5: Fluidity configuration file for a backward-facing step opened with Diamond

Fluidity [Piggott et al., 2008, Applied Modelling and Computation Group (AMCG), 2013] is a multi-phase computational fluid dynamics code using finite element methods for numerically solving the Navier-Stokes equations on unstructured meshes. Areas of application include geophysical

[1]Refer to `http://dune.mathematik.uni-freiburg.de/doc/dune-fem-howto-1.3.0.pdf` for an example implementation of a linear operator for the Poisson problem.

fluid dynamics, computational fluid dynamics, ocean modelling and mantle convection. Notable features are multi-phase flow, moving meshes with adaptivity over space and time, support for various classes of finite elements including mixed formulations and MPI distributed memory parallelisation. The GUI configuration editor Diamond [Ham et al., 2009] shown in Figure 3.5 allows users to configure simulation runs and parametrise the models implemented by Fluidity in an easy-to-use manner, defining user-defined prescribed fields and boundary conditions in Python without having to write very extensive XML input files by hand.

3.1.5 Nektar++

Nektar++ [Vos et al., 2011] is a C++ template library for the tensor product based finite element method with support for low to high p-order piecewise polynomial basis functions and explicit, implicit and implicit-explicit (IMEX) time-stepping methods. Vos et al. [2010], Cantwell et al. [2011b,a], Bolis et al. [2013] conducted extensive experimental studies investigating the relative performance of h-, p- and h-p-refinement to obtain a specified error tolerance for a given problem and mesh. Their findings highlight the importance of choosing the appropriate data structures for solving the global linear system with an iterative solver on the same CPU architecture.

Nektar++ supports assembling a global sparse matrix, a local matrix approach and a sum-factorisation approach. Assembling a global matrix tends to be favourable for low-order continuous basis functions, whereas sum factorisation is most efficient for high order, in particular for quadrilateral and hexahedral finite elements. For much of the intermediate region and for discontinuous methods, the local matrix approach performs best. These performance differences are expected to be even more pronounced for many-core architectures not presently supported.

3.2 FEniCS

FEniCS [Logg et al., 2012a] is an open source software project founded in 2003 with the goal of automating the efficient solution of differential equations. As outlined in Logg [2007], this involves automation of (i) discretization, (ii) discrete solution, (iii) error control, (iv) modeling,

and (v) optimisation. The three major design goals for FEniCS are *generality*, *efficiency*, and *simplicity*. Dynamic *code generation* is used to combine generality and efficiency, which are generally regarded as opposing goals.

This chapter describes the most important components of FEniCS and how they interact to achieve the set goals. For a more comprehensive introduction, refer to the FEniCS book by Logg et al. [2012a].

3.2.1 DOLFIN

FEniCS is built around the problem solving environment DOLFIN (*Dynamic Object-oriented Library for FINite element computation*) [Logg et al., 2012c, Logg and Wells, 2010], a C++ class library augmented by a SWIG-generated Python interface [Beazley, 2003]. This allows for a seamless integration in a scripting environment, with the remaining FEniCS components implemented in Python, and combines the performance of a C++ library with the versatility of a scripting language. In the following we will always refer to the Python interface of DOLFIN.

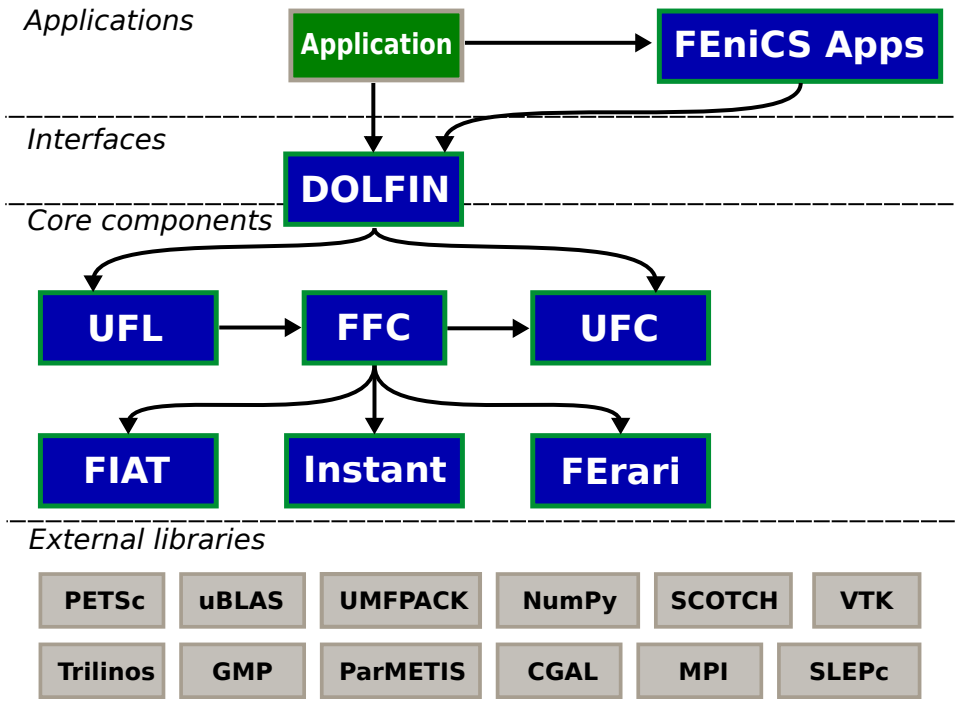

Figure 3.6: FEniCS system architecture with DOLFIN as the main user interface component [adapted from Logg et al., 2012c]

Figure 3.6 illustrates the interaction of the various FEniCS components

centred around DOLFIN as the main user interface for solving differential equations, which will be discussed in the following subsections. The user specifies their problem in form of *function spaces* defined by *finite elements* on a *mesh* and *variational forms* using these function spaces. Variational forms are defined using the domain specific Unified Form Language UFL (Section 3.2.2). The FFC form compiler (Section 3.2.3) translates these forms into C++ code conforming to the UFC interface specification (Section 3.2.5), which are just-in-time compiled and made available as a Python module by the utility Instant (Section 3.2.6).

DOLFIN interfaces to a range of established linear algebra libraries to provide matrix and vector implementations as well as efficient linear solvers. At the time of writing, PETSc [Balay et al., 1997], Trilinos/Epetra [Heroux et al., 2005], and uBLAS [Walter and Koch, 2014] were supported.

Both the C++ and Python interfaces support parallel computations using multiple threads on a single node, using multiple nodes communicating via MPI and a combination thereof. Preprocessing the mesh is required in either case: for multi-threaded computations the mesh needs to be *coloured* to avoid race conditions when updating the same mesh entity from different threads simultaneously and for distributed parallel computations the mesh is partitioned such that each process only owns and reads their respective partition of the mesh.

3.2.2 UFL

The *Unified Form Language* UFL [Alnæs, 2012, Alnæs et al., 2014] is a *domain-specific language* embedded in Python for the description of finite element variational forms and functionals. UFL is only concerned with their representation at the finite element level and is oblivious of meshes and function spaces. It is designed as a front end for form compilers, but also implements analysis and transformation of expressions. Automatic differentiation of forms and expressions is supported as well as common algebraic operators such as transpose, determinant, inverse, trigonometric functions and elementary functions such as `abs`, `pow`, `sqrt`, `exp` and `ln`.

Finite elements are defined by a *family*, *cell* and *polynomial degree*. The family specifies the kind of basis function to be used such as “Lagrange” or “DG”, a shorthand for “Discontinuous Lagrange”. UFL relies on the

Listing 3.1: Examples of UFL finite element declarations. "CG" is a shorthand for "Continuous Galerkin" and "DG" for "Discontinuous Galerkin".

```
P = FiniteElement('Lagrange', triangle, 1)
V = VectorElement('CG', triangle, 2)
T = TensorElement('DG', triangle, 0, symmetry=True)

TH = V * P  # Create a mixed Taylor-Hood element
```

Listing 3.2: UFL representation of the bilinear form a and the linear form L for the Poisson equation.

```
element = FiniteElement('Lagrange', triangle, 1)

u = TrialFunction(element)
v = TestFunction(element)
f = Coefficient(element)

a = dot(grad(v), grad(u))*dx
L = v*f*dx
```

form compiler to provide these basis functions. The cell is the polygonal shape of the reference element and one of *interval*, *triangle*, *quadrilateral*, *tetrahedron*, and *hexahedron*. Other than the scalar `FiniteElement`, UFL supports a vector valued `VectorElement` and a `TensorElement` for rank 2 tensors. The number of components for the vector and tensor valued cases is given by the geometric dimension of the reference cell. Elements can be arbitrarily combined to form *mixed elements*, which can themselves be combined further. Some examples are given in Listing 3.1.

UFL forms are integral expressions whose arguments can be either or both of (unknown) argument functions $\{\Phi^k\}$ and (known) coefficient functions $\{w^k\}$. Forms with a single argument are called *linear*, those with two *bilinear* and those containing more than two arguments *multilinear*. Forms that do not contain any arguments evaluate to a real number and are known as *functionals*. A valid form is any UFL expression that is linear in its arguments $\{\Phi^k\}$, may be non-linear in its coefficients $\{w^k\}$, and is integrated, that is multiplied by a *measure*, exactly once. The measure defines the type of integral, which is either a *cell integral* with measure `dx`, an *exterior facet integral* with measure `ds` or an *interior facet integral* with measure `dS`. Measures can be indexed, in which case the integral is defined only on a subset on the domain. Listing 3.2 shows the UFL representation

of the Poisson equation using $\mathcal{P}_1$ Lagrange elements (Section 2.1.4).

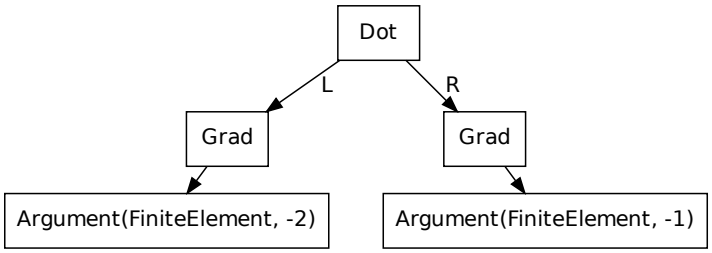

Figure 3.7: Expression tree representation of the bilinear form *a* from the Poisson equation in Listing 3.2

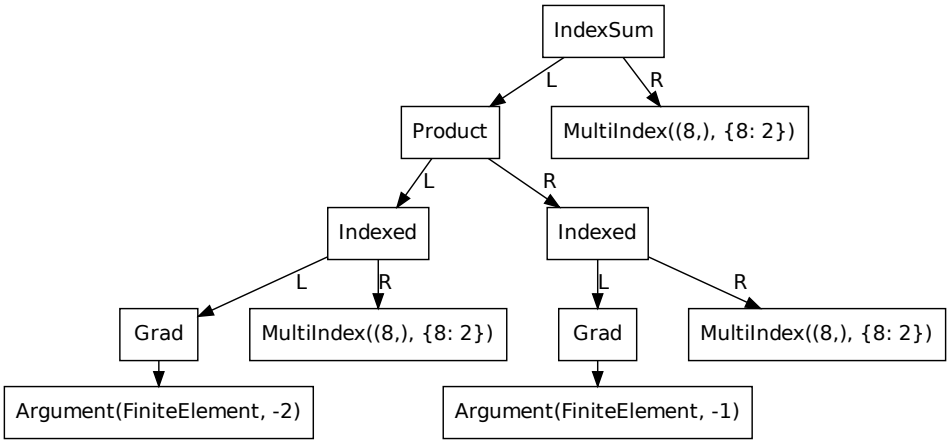

Figure 3.8: Expression tree representation after expanding components

A UFL expression is represented as a direct acyclic graph (DAG) with nodes being either *operators* with their *operands* as children or *terminals*, the leafs of the DAG. UFL provides algorithms for pre- and post-order traversal of these expression trees as well as a number of specialised tree transformers to expand *compound* nodes such as gradients and expand *derivatives* and *indices*. Figure 3.7 shows the expression tree representation[2] of the form `a` from Listing 3.2. The form is the root node with a single cell integral as descendant, which in turn consists of a dot product with the two gradients of the arguments as its children. After expanding compounds, the tree is transformed as shown in Figure 3.8, with the dot

[2]These trees are created using the `ufl2dot` function from `ufl.algorithms`. Note that the representations of `Argument` and `MultiIndex` have been shortened to make the graphs more compact and readable.

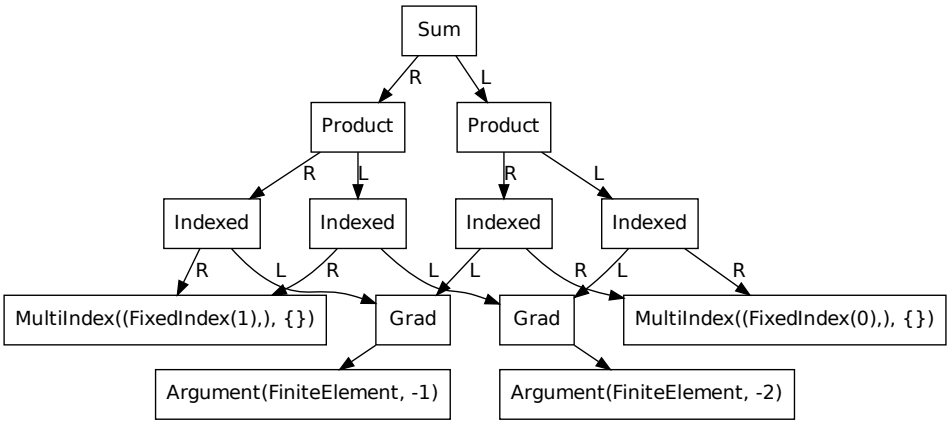

Figure 3.9: Expression tree representation after expanding components and indices

product expanded into an indexed sum over an index space of dimension two, the two spatial dimensions of the gradient. Expanding derivatives does not alter this particular DAG. When subsequently expanding indices as shown in Figure 3.9, the indexed sum is unrolled into a sum of two products whose operands are the first and second component of the gradients respectively.

3.2.3 FFC

The *FEniCS Form Compiler* FFC [Logg et al., 2012d, Kirby and Logg, 2006] automatically generates problem-specific code for the efficient evaluation of element tensors from a given multilinear form specified in UFL. FFC emits optimised low-level code conforming to the UFC interface specification described in Section 3.2.5 to a C++ header file to be used with a UFC compliant finite element implementation, such as DOLFIN. Alternatively, FFC can be called from a Python scripting environment to work as a *just-in-time compiler* (JIT) for the evaluation of multilinear forms.

FFC supports element tensor evaluation by both *quadrature* and *tensor contraction* [Ølgaard and Wells, 2010, Kirby and Logg, 2007, see also Sections 2.1.8 and 2.1.9], and internally calls FIAT (Section 3.2.4) for tabulation of basis functions and their derivatives as well as quadrature points. A heuristic based on the estimated cost of the tensor representation of each integral, given the number of coefficients, is used to determine which representation to use, unless this choice is overridden with a parameter.

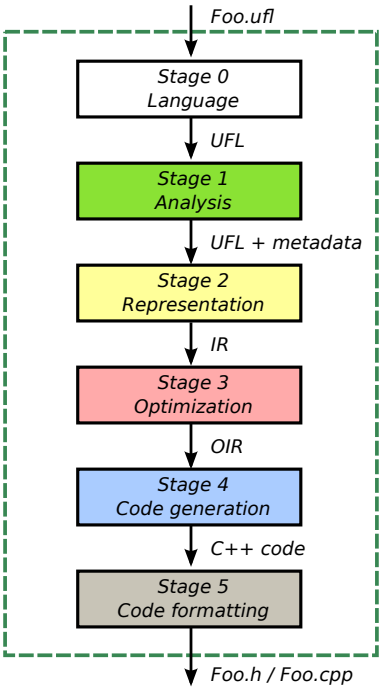

Figure 3.10: Stages of FFC form compilation [adapted from Logg et al., 2012d]

Form compilation in FFC operates in the stages shown in Figure 3.10: A UFL input file is translated into a set of UFL forms by evaluation in the *Language* stage. These forms are preprocessed in the *Analysis* stage to extract metadata on the finite elements, coefficients and integrals used in the forms. In the *Representation* stage, data needed for code generation, such as basis functions and degree of freedom mappings, are prepared in an intermediate representation (IR). If selected, the optional *Optimization* stage uses loop invariant code motion, common subexpression elimination, and precomputation of constants [Ølgaard and Wells, 2010] on the quadrature IR. Optimisation of the tensor contraction IR with respect to the number of arithmetic operations is available with the FErari Python library [Kirby et al., 2005, 2006, Kirby and Scott, 2007], at the expense of a potentially significantly more expensive form compilation. C++ code is generated from the IR in the *Code generation* stage and written to disk conforming to the UFC specification in the final *Code formatting* stage.

3.2.4 FIAT

The *FInite element Automatic Tabulator* FIAT [Kirby, 2004, 2012] is a Python library for the automatic tabulation of finite element basis functions over polynomial function spaces in one, two and three spatial dimensions. FIAT generates quadrature points of arbitrary order on the reference simplex and tabulates the basis functions and their derivatives at any given set of points. The use of FIAT to provide basis functions for FFC enables support for higher-order H^1, $H(\mathrm{div})$ and $H(\mathrm{curl})$ elements.

3.2.5 UFC

The interface between DOLFIN and FFC is specified by the *Unified Form-assembly Code* UFC [Alnæs et al., 2012, Alnaes et al., 2009], a C++ header file defining a standard interface to problem-specific assembly code for a general-purpose finite element library. DOLFIN's assembler implements the UFC interface, while FFC generates the problem-specific inner loop.

UFC defines abstract interfaces for *forms* which contain *finite elements*, *degree-of-freedom mappings* as well as *cell and facet integrals* to be implemented for a concrete variational form of interest. Furthermore, it specifies concrete representations of a *mesh*, a *cell* of that mesh and a *function*, which are used to transfer data between the library and the problem specific implementation. Lastly, UFC establishes numbering conventions for reference cells in 1D (interval), 2D (triangle, quadrilateral) and 3D (tetrahedron, hexahedron).

For cells, interior and exterior facet integrals, UFC assumes a five-step assembly process which proceeds sequentially over the mesh cell by cell:

1. Fetch a cell from the mesh and populate a UFC `cell` with coordinates.
2. Restrict coefficients to the cell, possibly by interpolation of the coefficient evaluated at the set of nodal points, calling `evaluate_dofs`.
3. Tabulate a local-to-global map of degrees of freedom for each function space, using `tabulate_dofs`.
4. Compute the contribution of the local element or interior/exterior facet tensor, calling `tabulate_tensor`.
5. Add the local element tensor contribution to the global tensor using the local-to-global map computed in step 3.

3.2.6 Instant

Instant [Wilbers et al., 2012] is a tool for the just-in-time (JIT) compilation of C/C++ code into a module callable from Python. It is used for inlining generated code with FFC and DOLFIN. Code to be inlined is passed to Instant as a string and compiled together with wrapper code generated by SWIG [Beazley, 2003] into a Python C extension module using Distutils or CMake. To not incur the compilation overhead more than once, compiled extension modules are cached in memory and on disk using the SHA1 checksum of the compiled code as the cache key.

SWIG type maps are provided to automatically convert back and forth between NumPy arrays on the Python side and pairs of plain pointers and array lengths on the C side. To be able to use OpenMP or external libraries, Instant allows the specification of extra headers to include, libraries to link against and the customisation of include directories, library search directories and compiler flags.

3.3 OP2

OP2 [Giles et al., 2012, 2013] is a domain-specific abstraction for the parallel execution of loop kernels over data defined on unstructured grids. The key feature of OP2 is the transparent control of an optimised parallel schedule for a variety of target architectures.

3.3.1 Key Concepts

The basic ingredient for the OP2 abstraction is the notion of *sets* (`op_set`), *mappings* (`op_map`) between pairs of those sets, and *data* (`op_dat`) associated with a particular set. Data is manipulated by *parallel loops* (`op_par_loop`) executing a user supplied *kernel* over a given iteration set. As further parameters, the parallel loop takes *access descriptors*, containing a data set, a mapping with an index, and an access mode. The mapping associates a fixed number of items from the set the data is declared over with each item of the set the loop iterates over. As a consequence data access can be *direct*, in the case where both sets coincide and the mapping is the identity, or *indirect*, via the given mapping. There is no need for this mapping to be injective or surjective. It is furthermore possible to associate each

element of the source with multiple elements of the target set, as long as the *arity*, the number of target elements associated with each source element, of this mapping is constant. The access mode specifies how data is accessed by the kernel: read only, write only, read/write, or read/write with contention. The kernel is invoked for each element of the given set and passed data associated with that element, possibly via an indirection.

OP2 has exclusive control over scheduling the parallel execution of the loop and thus mandates that the result has to be independent of the order of processing the elements of the iteration set. Loops are always executed for the entire set they are called with and hence are expected to touch all the data they are passed.

In writing an OP2 programme, the programmer needs to distinguish two layers. The *user level* is the layer where OP2 data structures are initialised and manipulated via parallel loop calls. The *kernel level* is the layer on which user kernels are implemented. A user kernel's signature must match the data passed to it in the parallel loop call, which is the data the kernel is working on. The kernel has no information on which element of the iteration set is being processed and there is no possibility for synchronisation as this would break the arbitrary schedulability.

3.3.2 Design

OP2 is implemented as an *active library* [Czarnecki et al., 2000], using domain-specific code generation to produce an optimised implementation for a specific target architecture. Order of traversal of the set in the parallel loop, granularity of parallelisation, partitioning and data layout can be adapted to characteristics of the target architecture via source-to-source translation of the user programme and kernels. Multi-core CPUs with OpenMP, NVIDIA GPUs with CUDA and inter-node parallelisation with MPI were supported at the time of writing.

Design concepts and the components OP2 consists of are shown in Figure 3.11, which illustrates the control flow in an OP2 programme, the role of source-to-source translation and how user code interacts with OP2 library components. The user supplies a host programme implemented using the OP2 host API and a number of kernel subroutines that conform to the kernel API. This host programme declares OP2 data structures and

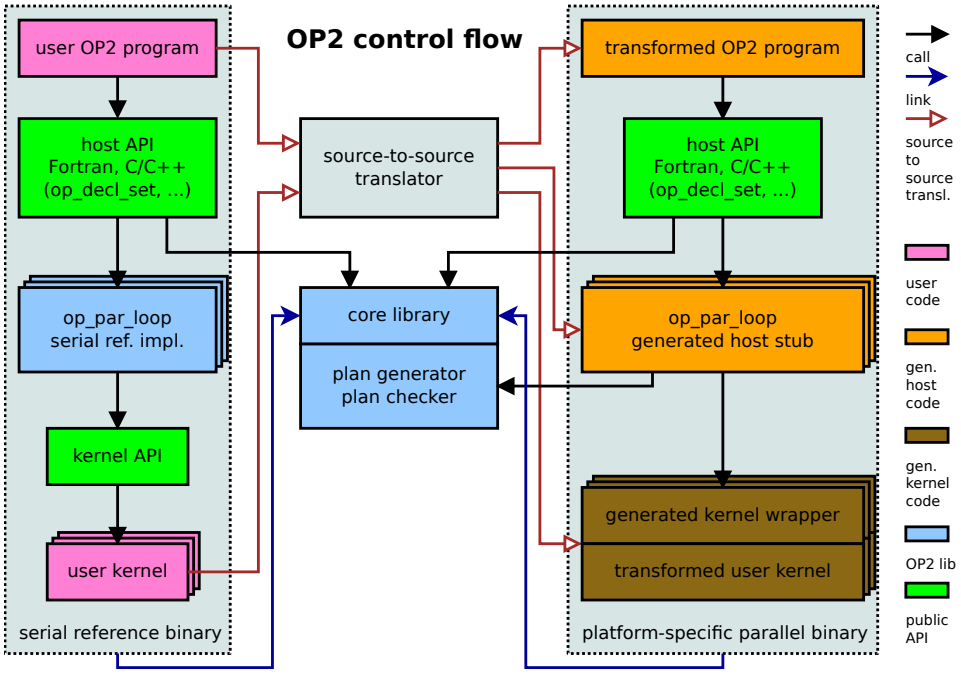

Figure 3.11: Control flow in an OP2 programme for the serial reference implementation (left) and a platform-specific parallel implementation using source-to-source translation (right)

initialises those with data that can be generated, read from disk or from another external source, and invokes parallel loops that manipulate this data using the kernels the user supplied. Since both host and kernel API are available in C/C++ and Fortran, the user can exploit the full power of these languages in OP2 programmes as long as the restrictions mentioned in section 3.3.1 are fulfilled.

While developing a programme or for verification purposes, the user can run a serial reference implementation where no source-to-source translation is used, as shown on left hand side of Figure 3.11. Instead, a static implementation of the parallel loop interface is called, which in turn directly calls the unmodified user kernels. Performance is expected to be poor in this case and for a production run a generated implementation optimised for the target platform is preferable as shown on the right hand side of Figure 3.11. The source-to-source translator analyses each parallel loop invocation and transforms the host programme to call a generated stub routine specific to a given kernel. Based on the characteristics of the loop, a parallel execution plan is requested which defines how the

datasets are partitioned for parallel execution, coloured to avoid data races and write contention, and mapped to hardware execution units. Subsequently, the platform-specific kernel is called, which is a wrapper of the kernel routine provided by the user and transformed as required.

3.4 Stencil Languages

3.4.1 Stencil Computations on Structured Meshes

A large class of computations on regular two, three or higher dimensional grids in space and time can be described by stencils, which define the rule for updating a grid point as a function of itself and its nearby neighbours. Stencil computations have been extensively studied and cache-oblivious algorithms pioneered by Frigo and Strumpen [2005, 2007].

There are various implementations of DSLs and compilers for generating and tuning stencil codes on multi-core CPUs and GPUs. These include Mint [Unat et al., 2011], a pragma-based programming model for stencil computations targeting CUDA, which has been applied to accelerate a 3D earthquake simulation with minimal changes to the existing application [Unat et al., 2012]. The SBLOCK framework [Brandvik and Pullan, 2010] provides a DSL embedded in Python and accompanying runtime library for defining stencil computations that are translated into low-level CPU and GPU kernels which can automatically exchange halo data via MPI in distributed parallel runs. Zhang and Mueller [2012] delivered a stencil computation framework that performs auto-tuning of the alignment of data structures and kernel launch parameters on GPUs. Pochoir [Tang et al., 2011] is a compiler for a domain-specific stencil language embedded in C++, compiling down to a parallel cache-oblivious algorithm in Cilk, targeting multi-core CPUs. Performance portability of automatically tuned parallel stencil computations on many-core architectures is demonstrated by Kamil et al. [2010], generating code from a sequential stencil expressed in Fortran 95, and the code generation and auto-tuning framework PATUS [Christen et al., 2011] with stencils specified in a DSL embedded in C. Although all these DSLs successfully generate optimized code, they are limited to structured meshes, which are not discussed further.

Listing 3.3: Algorithm and schedule for a 3x3 normalised box filter in two passes: The first pass produces the image `blur_x` from `input`, which is transformed into `blur_y` in the second pass.

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x,y-1) + blur_x(x,y) + blur_x(x,y+1))/3;

  blur_y.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

3.4.2 Halide

Halide [Ragan-Kelley et al., 2012, 2013] is a domain-specific language embedded in C++ and optimising compiler for multi-stage image processing pipelines. The compiler synthesises high performance implementations from a Halide *algorithm* and *schedule*, targeting various hardware architectures including x86/SSE, ARM v7/NEON, CUDA, Native Client, and OpenCL. Optimisation takes into account parallelism, locality and the reuse versus recomputation trade-off in a stochastic search over the space of possible schedules. The authors report speedups of up to 9x over expertly hand tuned code for a complex local Laplacian pipeline.

Stages in the image processing pipeline are described using the Halide DSL in a pure functional style, mapping coordinates on an infinite integer domain to colour values representing an image. Pipelines are formed by chaining functions. The function in Listing 3.3 defines the algorithm and schedule for a 3x3 normalised box filter in two passes.

3.4.3 Liszt

Liszt [DeVito et al., 2011] is a domain-specific language embedded in Scala for the construction of unstructured mesh based PDE solvers. It targets distributed parallel clusters, SMP systems and GPUs via MPI, pthreads and CUDA backends. Liszt enhances Scala with first-class abstract data types for mesh elements such as vertices, edges, faces and cells, which

are grouped into sets. Topological relationships of the mesh are implicitly given through the use of built-in functions to access neighbouring elements. These allow the Liszt compiler to automatically infer the stencil for a given computation, assuming a fixed topology during execution.

All data-parallel computations on the mesh are expressed via for-comprehensions over a particular entity type of the mesh, where arbitrary nesting of for-comprehensions is allowed. Calculations in the body of a comprehension are independent and have no dependencies, allowing Liszt freedom in the choice of different parallel schedules for different backends. Liszt implements two different parallel execution strategies: partitioning with automatic discovery of ghost elements that need to be communicated for the MPI backend and colouring to avoid data races for the pthreads and CUDA backends.

Data in a Liszt application is stored in fields, associated with all mesh elements of a particular type. For the duration of any for-comprehension, any field can be in exactly one of three states, enforced by the compiler: read-only, write-only or reduction using an operator. The compiler performs phase-change analysis on each field to determine synchronisation points and insert backend-specific communication calls.

At first glance, Liszt appears very similar to OP2 described in Section 3.3. The level of abstraction is an element of the mesh in both cases, however the description the mesh differs significantly. While Liszt uses local topological functions, OP2 uses explicit user-defined relations between mesh entities, called maps, forming a graph-like data structure. Local computations in OP2 are expressed as kernels with explicitly provided dependencies, whereas in Liszt computations are encapsulated in for-comprehensions within the regular program flow and dependencies are automatically inferred by program analysis.

Since Liszt is restricted to storing field data only on mesh elements, it is considerably less flexible in what a field can represent. Consider a field representing the degrees of freedom (DOFs) of a higher-order numerical scheme. In the case of quadratic or second-order basis functions, DOFs are associated with the vertices and the midpoints of edges. While this can be naturally expressed in OP2 by associating six elements of the set of DOFs with each triangle element, Liszt requires the user to explicitly manage two fields, one on the vertices and one on the edges.

Liszt's high-level semantic operators provide elegant means to build solvers operating on unstructured meshes. However, its embedding in Scala makes it considerably harder to interface with third party code and established scientific libraries such as linear algebra backends.

3.5 Conclusions

In this chapter, a landscape of successful approaches to abstracting scientific computations has been laid out, spanning a range of applications and implementation choices. Combining the goals of efficiency, portability, maintainability and composability however remains a challenge.

Libraries, presented in Section 3.1, have been successfully employed to abstract coarse-grained operations, such as linear algebra. Portability between different hardware architectures however is challenging and adding support for a new platform usually requires a substantial rewrite and deep familiarity with the library's implementation. Due to their frequently monolithic nature, the composability of libraries is furthermore restricted to use cases foreseen by the developers and exposed in the API.

Domain-specific approaches for stencil computations, exposed in Section 3.4, have successfully demonstrated the generation of efficient code on different hardware architectures for structured grid applications. While Liszt is explicitly aimed at portable PDE solvers on unstructured meshes, some design decisions have been identified as questionable regarding its applicability to complex finite element computations.

OP2, introduced in Section 3.3, is a suitable abstraction for executing kernels over unstructured meshes. However, the limitations imposed by static source-to-source translation and the lack of support for matrices are a barrier to adoption for finite element local assembly. Adding those missing features to the OP2 framework proved infeasible, which led to the design of PyOP2, described in Chapter 4, drawing inspiration from OP2.

The FEniCS project provides a comprehensive tool box for finite element computations, centred around the C++ library DOLFIN. Its unified form language UFL and the FEniCS form compiler FFC are essential building blocks in the design of the finite element framework Firedrake described in Chapter 5.

Chapter 4

PyOP2 - A DSL for Parallel Computations on Unstructured Meshes

Many numerical algorithms and scientific computations on unstructured meshes can be viewed as the *independent application* of a *local operation*, expressible as a *computational kernel*, everywhere on a mesh, which lends itself naturally to parallel computation. In some cases contributions from local operations are aggregated in a *reduction* to produce the final result.

PyOP2 is a domain-specific language (DSL) embedded in Python for the parallel executions of computational kernels on unstructured meshes or fixed-degree graphs, which implements this abstraction. The topology of an unstructured mesh is described by sets of entities such as vertices, edges and cells, and the connectivity between them. Data defined on the mesh is managed by abstracted data structures for vectors and matrices.

PyOP2 targets both multi- and many-core architectures via a unified API which requires no changes to user code to be able to run on different backends. Computations over the mesh are efficiently executed as low-level, platform-specific code generated and just-in-time (JIT) compiled at runtime. This runtime architecture allows PyOP2 to reason about and interact with concrete objects representing the actual data and problem being solved instead of having to parse, analyse and transform code.

At the same time PyOP2 is carefully optimised to avoid unncessary recomputation through sophisticated caching mechanisms. Computation-

ally heavy parts are either executed in native generated code or implemented as Python extension modules, combining the efficiency of optimised low-level code with the flexibility and interactivity of Python.

This chapter describes the design of PyOP2, starting with an overview of the concepts and architecture, and continuing with detailed expositions of the support for multiple backends, distributed parallelism, mixed and coupled problems and the linear algebra interface. PyOP2 is an essential building block in the design of Firedrake described in Chapter 5.

4.1 Concepts

PyOP2 conceptually distinguishes the topological connectivity between sets of user defined classes of entities (Section 4.1.1), data defined on these sets (Section 4.1.2) and computations as the uniform execution of kernels over sets of entities (Section 4.1.3). There is no predefined meaning associated with the classes of entities described by sets and no inherent concept of a mesh. This makes PyOP2 applicable as a building block for a wide range of unstructured applications.

While the following discussion is limited to mesh based applications, it is worth noting that unstructured meshes are isomorphic to graphs, with vertices of a mesh corresponding to nodes of a graph and mappings between mesh entities corresponding to links in a graph. PyOP2 is therefore equally suitable for computations on fixed-degree graphs.

4.1.1 Sets and Mappings

The topology of an unstructured mesh is defined by sets of entities and mappings between these sets. While maps define the connectivity between entities or degrees of freedom (DOFs), for example associating an edge with its incident vertices, sets are abstract representations of mesh entities or DOFs on which data is defined, and are characterised only by their cardinality. Sets representing DOFs may coincide with sets of mesh entities, however unlike Liszt presented in Section 3.4.3, PyOP2 is not limited to this case. For example, DOFs for a field defined on the vertices of the mesh and the midpoints of edges connecting the vertices are defined by a single set with a cardinality equal to the sum of vertices and

edges. The data values associated with these DOFs can therefore be stored contiguously as described in Section 4.1.2.

Maps are defined by a *source* and *target* set and a *constant arity*, that is, each element in the source set is associated with the same number of elements in the target set. Associations are stored in a logically two-dimensional array, allowing the lookup of elements associated with each source set element by index. This restriction to constant arity is due to PyOP2's computation model of uniformly executing the same kernel for each iteration set element and excludes certain kinds of mappings. A map from vertices to incident edges or cells is only possible on a very regular mesh in which the multiplicity of every vertex is constant. Similarly, meshes containing different types of cells, for example mixing triangles and quads, cannot be straightforwardly represented. Sets and maps are immutable and considered equal only if they are object-identical.

A set `vertices`, a set `edges` and a map `edges2vertices` associating the two incident vertices with each edge are declared as follows:

```
vertices = op2.Set(4)
edges = op2.Set(3)
edges2vertices = op2.Map(edges, vertices, 2, [[0,1],[1,2],[2,3]])
```

4.1.2 Data

Three kinds of user provided data are distinguished by PyOP2: data defined on a `Set`, often referred to as a field, is represented by a `Dat`, data that has no association with a `Set` by a `Global` and read-only data that is visible globally and referred to by a unique identifier is declared as `Const`. Examples of the use of these data types are given in Section 4.1.3.

Dat

A PyOP2 `Dat` provides a completely abstracted representation of a vector holding data defined on a given `Set`, where the actual values might be stored in CPU or GPU memory, depending on the chosen backend (Section 4.4). When running distributed parallel computations (Section 4.7), the `Dat` is partitioned among all participating processors and manages its *halo*, the overlap region containing data from neighbouring processors required to perform computations over the partition boundary. Storage,

layout, halo exchange and host to device transfer of this data are automatically managed transparently to the user or application using PyOP2. Users need to explicitly request access to modify this data outside of a PyOP2 context, which allows PyOP2 to keep track of these modifications.

Unlike the immutable sets and maps, `Dats` are dynamic data types which manage their own state and respond appropriately to queries from PyOP2 or the user. This state includes whether or not data is allocated in CPU or, if applicable, accelerator memory and which copies of the data are up-to-date. Similarly, the `Dat` is aware of user modifications and whether a halo exchange is needed. PyOP2 can therefore call operations such as a data transfer or halo exchange on the `Dat` unconditionally where they might be needed and the `Dat` decides on its own authority whether an action is required or the operation returns immediately. Unnecessary operations are thereby avoided and PyOP2's design considerably simplified.

In most contexts, a `Dat` can be used as a vector in a mathematical or linear algebra sense. Common arithmetic operations such as pointwise addition, subtraction, multiplication and division are supported, provided the shape matches. If one of the operands is a scalar, the operation is applied to all components of the `Dat`. All operations are implemented in a backend-independent manner using parallel loops (Section 4.1.3).

Since a `Set` only defines a cardinality, data declared as a `Dat` on a `Set` needs additional metadata to allow PyOP2 to interpret the data and to specify how much memory is required to store it. This metadata is the *data type* and the *shape* of the data associated with any given set element. The shape can be a scalar or a one- or higher-dimensional vector of arbitrary extent and is associated with a `DataSet` on which the `Dat` is defined. The number of data values stored by a `Dat` is fully defined by its `DataSet`. Similar to the restriction on maps, the shape and therefore the size of the data associated which each `Set` element must be uniform. PyOP2 supports all common primitive data types provided by NumPy.

Declaring coordinate data on the `Set` of `vertices` defined above, where two float coordinates are associated with each vertex, is done like this:

```
dvertices = op2.DataSet(vertices, dim=2)
coordinates = op2.Dat(dvertices,
                      [[0.0,0.0],[0.0,1.0],[1.0,1.0],[1.0,0.0]],
                      dtype=float)
```

Global

Data with no association to a set is represented by a `Global`, characterised by a shape and data type, which have the same interpretation as for a `Dat`. A 2x2 elasticity tensor would be defined as follows:

```
elasticity = op2.Global((2,2), [[1.0,0.0],[0.0,1.0]], dtype=float)
```

Const

Data that is globally visible and read-only to kernels is declared with a `Const` and needs to have a globally unique identifier. `Const` data does not need to be passed as an argument to a parallel loop, but is accessible in a kernel by name. A globally visible parameter `eps` is declared as follows:

```
eps = op2.Const(1, 1e-14, name="eps", dtype=float)
```

Mat

In a PyOP2 context, a – generally sparse – matrix is a linear operator from one `Set` to another. In other words, it is a linear function which takes a `Dat` on one set A and returns the value of a `Dat` on another set B. Of course, in particular, A may be the same set as B. This makes the operation of some matrices equivalent to the operation of a particular PyOP2 kernel.

PyOP2 parallel loops can be used to assemble matrices, represented by the `Mat` class, which are defined on a sparsity pattern. The row and column spaces the sparsity maps between are given by the `DataSets` of A and B. A sparsity pattern is built from one or more pairs of `Maps`, supporting matrices that are assembled from more thane one kernel. Each pair contains a `Map` for the row and column space of the matrix respectively. The sparsity uniquely defines the non-zero structure of the sparse matrix and can be constructed from those mappings alone, using an algorithm detailed in Section 4.6.2. A `Mat` is therefore defined by a `Sparsity` and a data type.

Since the construction of large sparsity patterns is a very expensive operation, the decoupling of `Mat` and `Sparsity` allows the reuse of the same sparsity pattern for a number of matrices. PyOP2 caches sparsity patterns as described in Section 4.3.2. Declaring a sparsity on the same maps as a previously declared sparsity yields the cached object instead of building

another one. A matrix of floats on a sparsity which spans from the space of vertices to the space of vertices via the edges is declared as follows:

```
sparsity = op2.Sparsity((dvertices, dvertices),
                        [(edges2vertices, edges2vertices)])
matrix = op2.Mat(sparsity, float)
```

4.1.3 Parallel Loops

Computations in PyOP2 are executed by mapping the application of a kernel over an *iteration set*. Parallel loops are the dynamic core construct of PyOP2 and hide most of its complexity such as parallel scheduling, code generation, data transfer from and to device memory, if needed, and staging of data into on-chip memory. Kernels must be independent of the order in which they are executed over the iteration set to allow PyOP2 maximum flexibility to schedule the computation in the most efficient way.

The kernel is executed over the entire iteration set, which is usually a `Set` of mesh entities or degrees of freedom. Iteration over certain regions of the mesh, for example the boundary, is supported through `Subsets`, which are restrictions of a `Set` to a given list of entities identified by their index, which can be empty. When running in parallel, an empty iteration set is a way for a process to not participate in a parallel loop, which is a collective operation (Section 4.7). In Firedrake, subsets are used to hold boundary entities of a given marker region, which may only be defined on some of the partitions of a distributed mesh. Subsets may only be used as the iteration set in a parallel loop and can not hold data or define `Maps`.

A parallel loop invocation takes the iteration set and the kernel to operate on as its first two arguments, followed by a number of *access descriptors* defining how data is accessed by the kernel. Access descriptors are constructed from a data carrier, a `Dat`, `Mat` or `Global`, by passing the access mode and the map, in the case of a `Dat`, or pair of maps, in the case of a `Mat`, to be used to indirectly access the data[1]. The mapping is required for an *indirectly accessed* `Dat`, which is declared on a `Set` different from the iteration set of the parallel loop. For *directly accessed* data, defined on the iteration set,

[1] `Dat`, `Mat` and `Global` implement the `__call__` method, creating and returning an `Arg` type representing an access descriptor, which is not part of the public API. It serves as a transient container for the data carrier, access mode and mapping in a parallel loop call.

the map is omitted and only the access mode is specified. A `Mat` is always accessed indirectly through a pair of maps used to build the `Sparsity`.

Access descriptors define how data is accessed by the kernel and tell PyOP2 whether to stage in data before and stage it out after kernel execution and whether write contention needs to be accounted for. Valid access modes are `READ` (read-only), `WRITE` (write-only), `RW` (read-write), `INC` (increment), `MIN` (minimum reduction) or `MAX` (maximum reduction). Not all descriptors apply to all PyOP2 data types. A `Dat` can have modes `READ`, `WRITE`, `RW` and `INC`. For a `Global`, the valid modes are `READ`, `INC`, `MIN` and `MAX`, where the three latter imply a reduction. `Mats` only support `WRITE` and `INC`[2].

Parallel loops are a generic interface, which can be used for very different purposes as illustrated by the examples given in the following. Kernel signatures and the way data is accessed are described in Section 4.2.

Direct loop example

Consider a parallel loop that translates the `coordinate` field by an `offset` defined as a `Const`. The kernel therefore has access to the local variable `offset` even though it has not been passed as an argument to the parallel loop. This loop is direct and the argument `coordinates` is read and written:

```
op2.Const(2, [1.0, 1.0], dtype=float, name="offset");

translate = op2.Kernel("""
void translate(double * coords) {
  coords[0] += offset[0];
  coords[1] += offset[1];
}""", "translate")

op2.par_loop(translate, vertices, coordinates(op2.RW))
```

Matrix example

A parallel loop assembling the `matrix` via a `kernel`, which is omitted for brevity, iterating over the `edges` and taking `coordinates` as input data is given below. The `matrix` is the output argument of this parallel loop with access descriptor `INC`, since contributions from different vertices are accumulated via the `edges2vertices` mapping. Note that the mappings are indexed

[2]Reading from a `Mat` is conceptually possible, however not presently implemented.

with the *iteration indices* `op2.i[0]` and `op2.i[1]` respectively. This means that PyOP2 generates a two-dimensional *local iteration space* (Section 4.2.4) with an extent in each dimension equal to the arity of the `Map` `edges2vertices` for any given element of the iteration set. The `coordinates` are accessed via the same mapping as read-only input data using the access descriptor `READ`:

```
op2.par_loop(kernel, edges,
             matrix(op2.INC, (edges2vertices[op2.i[0]],
                              edges2vertices[op2.i[1]])),
             coordinates(op2.READ, edges2vertices))
```

Global reduction example

Globals are used primarily for reductions where a given quantity on a field is reduced to a single number by summation or finding the minimum or maximum. Consider a kernel computing the L2 norm of the `pressure` field defined on the set of `vertices` as `l2norm`. Note that the `Dat` constructor automatically creates an anonymous `DataSet` of dimension 1 if a `Set` is passed as the first argument. We assume `pressure` is the result of some prior computation and only give the declaration for context.

```
pressure = op2.Dat(vertices, [...], dtype=float)
l2norm = op2.Global(dim=1, data=[0.0])

norm = op2.Kernel("""
void norm(double * out, double * field) {
  *out += field[0] * field[0];
}""", "norm")

op2.par_loop(norm, vertices,
             l2norm(op2.INC),
             pressure(op2.READ))
```

4.2 Kernels

Local computations to be performed for each element of an iteration set are defined by a *kernel*, which has a *local view* of the data and can only read and write data associated with the current element directly or via one level of indirection. Any data read by the kernel, that is accessed as `READ`, `RW` or

`INC`, is automatically gathered via the mapping relationship in the *staging in* phase and the kernel is passed pointers to local data. Similarly, after the kernel has been invoked, any modified data i.e. accessed as `WRITE`, `RW` or `INC` is scattered back out via the `Map` in the *staging out* phase. It is only safe for a kernel to manipulate data in the way declared via the access descriptor in the parallel loop call. Any modifications to an argument accessed read-only would not be written back since the staging out phase is skipped for this argument. Similarly, the result of reading an argument declared as write-only is undefined since the data has not been staged in and the memory is uninitialised. The access mode `WRITE` is only safe to use for directly accessed arguments where each set element is written to exactly once. When accumulating data via a map where multiple iteration set elements are associated with the same element of the target set, the access descriptor `INC` must be used to notify PyOP2 of the write contention.

4.2.1 Kernel API

Kernels are declared as a *C code string* or an *abstract syntax tree* (AST), which efficiently supports programmatic kernel generation as well as hand-written kernels. The AST representation allows loop nest optimisations, in particular for finite element kernels, by the COFFEE AST optimiser described in Section 4.2.2. Kernel code is implemented in a restricted subset of C99, which is supported by all PyOP2 backends, detailed in Section 4.4, without requiring a full source-to-source translation of the kernel code. The *kernel function name* passed to the constructor must match the function name in the C kernel signature. Consider a kernel computing the midpoint of a triangle given the three vertex coordinates:

```
midpoint = op2.Kernel("""
void midpoint(double p[2], double *coords[2]) {
  p[0] = (coords[0][0] + coords[1][0] + coords[2][0]) / 3.0;
  p[1] = (coords[0][1] + coords[1][1] + coords[2][1]) / 3.0;
}""", "midpoint")
```

Below is the parallel loop invocation for the `midpoint` kernel above. A convenience shorthand allows the declaration of an anonymous `DataSet` of a dimension greater than one by using the `**` operator. The actual data in the declaration of the `Map cell2vertex` and `Dat coordinates` is omitted.

```
vertices = op2.Set(num_vertices)
cells = op2.Set(num_cells)

cell2vertex = op2.Map(cells, vertices, 3, [...])

coordinates = op2.Dat(vertices ** 2, [...], dtype=float)
midpoints = op2.Dat(cells ** 2, dtype=float)

op2.par_loop(midpoint, cells,
             midpoints(op2.WRITE),
             coordinates(op2.READ, cell2vertex))
```

Kernel arguments and access descriptors are matched by position. The kernel argument `p` corresponds to the access descriptor for `midpoints` and `coords` to the access descriptor for `coordinates` respectively. Direct arguments such as `midpoints` are passed to the kernel as a `double *`, indirect arguments such as `coordinates` as a `double **` with the first indirection index due to the map and the second index due the data dimension.

The optional flag `flatten` is used to create access descriptors for kernels which expect data to be laid out by component of the `Dat` (Section 4.2.3):

```
midpoint = op2.Kernel("""
void midpoint(double p[2], double *coords[1]) {
  p[0] = (coords[0][0] + coords[1][0] + coords[2][0]) / 3.0;
  p[1] = (coords[3][0] + coords[4][0] + coords[5][0]) / 3.0;
}""", "midpoint")

op2.par_loop(midpoint, cells,
             midpoints(op2.WRITE),
             coordinates(op2.READ, cell2vertex, flatten=True))
```

4.2.2 COFFEE Abstract Syntax Tree Optimiser

Kernels are initialised with either a C code string or an abstract syntax tree (AST), from which C code is generated. The AST representation provides the opportunity for optimisation through the COFFEE (COmpiler For Finite Element local assEmbly) AST optimiser [Luporini et al., 2014], which specialises on finite element local assembly kernels.

COFFEE performs platform-specific optimisations on the AST with the goals of minimising the number of floating-point operations and improv-

ing instruction level parallelism through the use of SIMD (Single Instruction Multiple Data) vectorisation. The optimiser can detect invariant sub expressions and hoist them out of the loop nest, permute and unroll loop nests and vectorise expressions. The last step may require padding of the data and enforcing alignment constraints to match the target SIMD architecture. COFFEE supports both SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) instruction sets.

4.2.3 Data Layout

Data for a `Dat` declared on a `Set` is stored contiguously for all elements of the `Set`. For each element, this is a contiguous chunk of data of a shape given by the `DataSet dim` and the data type of the `Dat`, laid out in row-major order. Its size is the product of the `dim` tuple extents and the data type size.

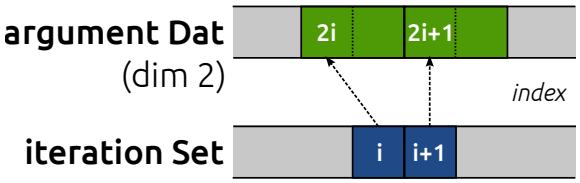

Figure 4.1: Data layout for a directly accessed `Dat` argument with `dim` 2

During execution of the `par_loop`, the kernel is called for each element of the iteration set and passed data for each of its arguments corresponding to the current `Set` element `i` only. For a directly accessed argument such as `midpoints` above, the kernel is passed a pointer to the beginning of the chunk of data for the element `i` the kernel is currently called for as illustrated in Figure 4.1. In CUDA and OpenCL `i` is the global thread id since the kernel is launched in parallel for all elements.

For an indirectly accessed argument such as `coordinates` above, PyOP2 gathers pointers to the data via the indirection `Map`. The kernel is passed a list of pointers of length corresponding to the `Map` *arity*, in the example above 3. Each of these points to the data chunk for the target `Set` element given by `Map` entries `(i, 0)`, `(i, 1)` and `(i, 2)` as shown in Figure 4.2.

If the argument is created with the keyword argument `flatten` set to `True`, a flattened vector of pointers is passed to the kernel as illustrated in Figure 4.3. The length of this vector is the product of the extents of the `dim` tuple and the arity of the `Map`, which is 6 in the example above. Each entry

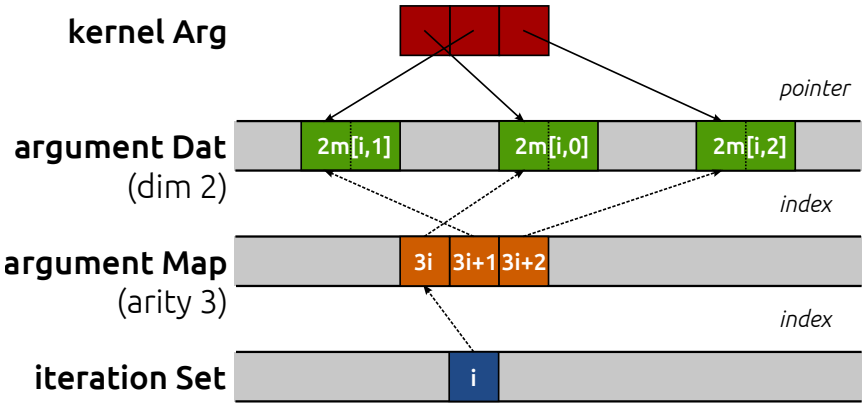

Figure 4.2: Data layout for a `Dat` argument with `dim` 2 indirectly accessed through a `Map` of `arity` 3

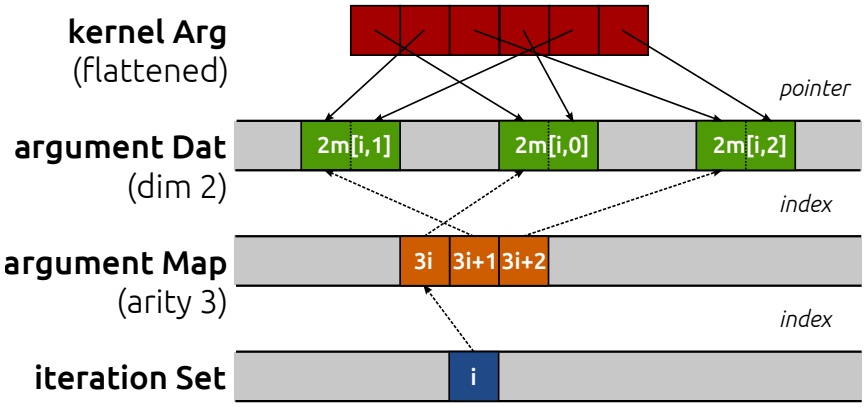

Figure 4.3: Data layout for a flattened `Dat` argument with `dim` 2 indirectly accessed through a `Map` of `arity` 3

points to a single data value of the `Dat`. The ordering is by component of `dim`, that is the first component of each data item for each element in the target set pointed to by the map followed by the second component etc.

4.2.4 Local Iteration Spaces

PyOP2 does not impose any limitations on the complexity of kernels and the amount of data they may read or write. In general, a kernel is executed by a single thread per iteration set element and the resource usage is proportional to the size of its output data, termed *local tensor*.

Executing a complex kernel by a single thread may therefore not be the most efficient strategy on many-core platforms with a high degree of concurrency but a limited amount of registers and on-chip resources per thread. To improve the efficiency for kernels with large working set sizes, their execution over the *local iteration space* for each iteration set element can be distributed among several threads. Each thread computes only a subset of this local iteration space, thereby increasing the level of parallelism and lowering the amount of resources required per thread.

To illustrate the concept, consider a finite element local assembly kernel for vector-valued basis functions of second order on triangles. This is merely an example and there are more complex kernels computing considerably larger local tensors commonly found in finite element computations, in particular for higher-order basis functions. Invoked for each element in the iteration set, this kernel computes a 12×12 local tensor:

```
void kernel(double A[12][12], ...) {
  ...
  // loops over the local iteration space
  for (int j = 0; j < 12; j++) {
    for (int k = 0; k < 12; k++) {
      A[j][k] += ...
    }
  }
}
```

Using iteration space, the kernel above changes as follows:

```
void kernel(double A[1][1], ..., int j, int k) {
  ...
  // compute result for position (j, k) in local iteration space
  A[0][0] += ...
}
```

Note how the doubly nested loop over basis functions is hoisted out of the kernel, which receives the position in the local iteration space for which to compute the result as additional arguments `j` and `k`.

PyOP2 is then free to schedule the execution over the local iteration space for each set element and choose the number of threads to use. On a CPU with large caches and a small number of concurrent threads, a single thread for the entire local iteration space is more efficient for most cases. On many-core platforms, where the kernel is executed in parallel over the iteration set, a larger number of threads can be launched to compute a subset of the local iteration space each, as shown in Figure 4.4 for a kernel computing a 6×6 local tensor with a single and 36 threads respectively.

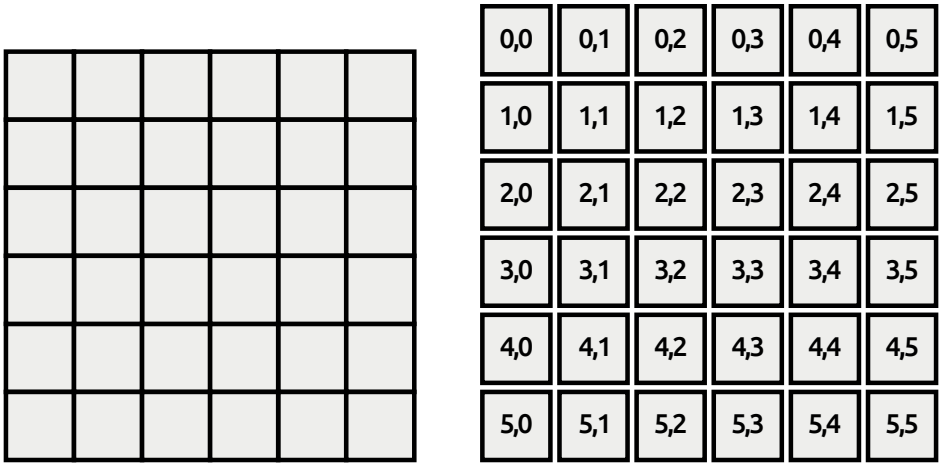

Figure 4.4: Unified iteration space (left) and local iteration space (right) for a kernel computing a 6×6 local tensor by a single thread and by 36 threads $(0,0) \ldots (5,5)$

When using a kernel with a local iteration space, the corresponding maps need to be indexed with an `IterationIndex i` in the access descriptor.

4.3 Architecture

As described in 4.1, PyOP2 exposes an API that allows users to declare the topology of unstructured meshes in the form of `Sets` and `Maps` and data in the form of `Dats`, `Mats`, `Globals` and `Consts`. Computations on this data are defined in `Kernels` described in 4.2 and executed by parallel loops.

The API is the frontend to the PyOP2 runtime code generation and compilation architecture, which supports the generation and just-in-time

(JIT) compilation of low-level code for a range of backends described in 4.4 and the efficient scheduling of parallel computations. A schematic overview of the PyOP2 architecture is given in Figure 4.5.

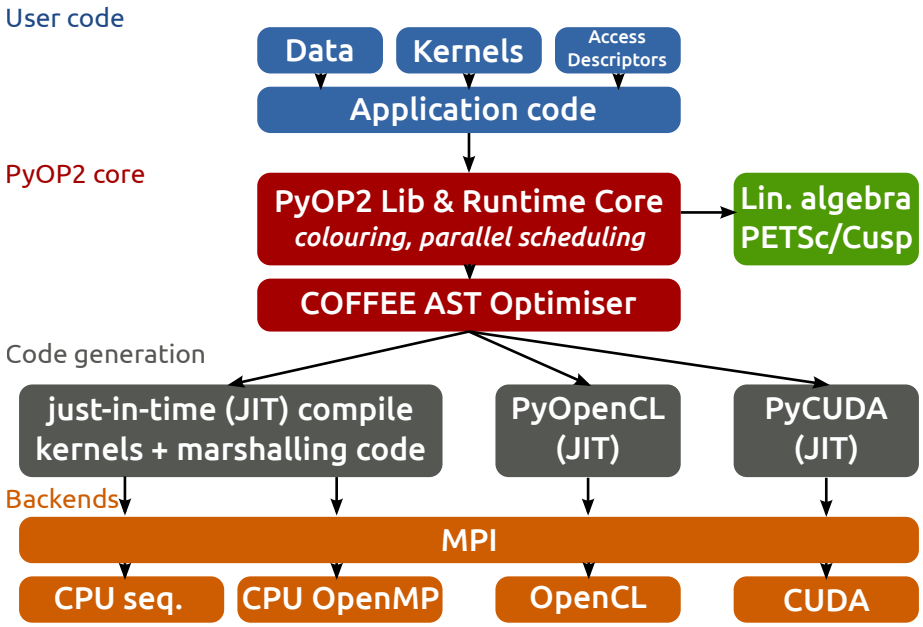

Figure 4.5: Schematic overview of the PyOP2 runtime computation architecture

4.3.1 Parallel Loops

From an outside perspective, PyOP2 is a Python library, with performance critical library functions implemented in Cython [Behnel et al., 2011]. A user's application code makes calls to the PyOP2 API, most of which are conventional library calls. The exception are `par_loop` calls, which encapsulate PyOP2's runtime core functionality performing backend-specific code generation. Executing a parallel loop comprises the following steps:

1. Compute a parallel execution plan, including information for efficient staging of data and partitioning and colouring of the iteration set for conflict-free parallel execution. This process is described in Section 4.5 and does not apply to the sequential backend.
2. Generate backend-specific code for executing the computation for a given set of `par_loop` arguments as detailed in Section 4.4 according to the execution plan computed in the previous step.

3. Pass the generated code to a backend-specific tool chain for just-in-time compilation, producing a shared library callable as a Python module which is dynamically loaded. This module is cached on disk to save recompilation when the same computation is launched again for the same backend.
4. Build the backend-specific list of arguments to be passed to the generated code, which may initiate host to device data transfer for the CUDA and OpenCL backends.
5. Call into the generated module to perform the actual computation. To efficiently overlap distributed parallel computations with communication, this involves separate calls for the regions owned by the current processor and the halo as described in Section 4.7.
6. Perform any necessary reductions for `Globals`.

When the `par_loop` function is called, PyOP2 instantiates a backend-specific `ParLoop` object behind the scenes, which manages the process laid out above. Similar to other types, this is a runtime data structure, which manages state and can be queried and interacted with. The `ParLoop` keeps a list of arguments and classifies those as direct, indirect, needing global reduction etc. If all the arguments are direct, a `ParLoop` is identified as direct, otherwise as indirect. Code generated for direct loops is considerably simpler and the `ParLoop` guides code generation accordingly.

The manifestation of parallel loops as objects enables the design of a lazy evaluation scheme where computations are postponed until results are requested. Upon creation, a `ParLoop` is not immediately executed. Instead, its read and write dependencies are identified and it is appended to an execution trace. While not presently implemented, this architecture enables transformations of the execution trace, including fusion of kernels and loops where allowed by the dependencies. When the actual data values of a `Dat`, `Mat` or `Global` are accessed, the evaluation of the dependency chain of this result is scheduled and the corresponding `ParLoops` executed. The correct execution of deferred computation is performed transparently to the users by enforcing read and write dependencies of Kernels.

4.3.2 Caching

Runtime code generation and compilation used when executing parallel loops carries a substantial cost that is not immediately apparent to a user

writing a PyOP2 programme. To mitigate this overhead as much as possible and ensure compilation of code for a particular loop is only done once, PyOP2 uses caching at several levels.

Global caches

Shared objects built from generated code are cached on disk, using an md5 hash as a fingerprint of the backend-specific code as the cache key. Any subsequent run using the same parallel loop, not necessarily from the same user programme, will not have to pay any compilation cost. In addition to that, the function pointer to the compiled parallel loop is cached in memory, such that further invocations of the same loop can jump straight into the compiled code without even having to go through the code generation stage again.

This cache is keyed on the kernel and the metadata of the arguments, such as the arity of maps and datasets, but not the data itself. In particular the size of the iteration set and any maps derived from it do not factor into the cache key, since the generated code does not depend on it. Parallel loops are therefore independent of the data they are executed over and for this reason, the cache is required to be global.

Kernels are also cached globally since they undergo a preprocessing stage, which is saved when a kernel is instantiated again with the same code or an equivalent abstract syntax tree (AST).

Object caches

Furthermore, PyOP2 builds a hierarchy of transient objects on immutable `Sets` and `Maps`. Other objects built on top of these are cached on their parent object. A `DataSet` is cached on the `Set` it is built from and a `Sparsity` is cached on the `Set` of its row `DataSet`. Mixed types described in Section 4.8 are cached on the `Set` underlying their first component. Thereby, the axiom that equality for a `Set` and `Map` means identity is extended to all these types, which makes equality checks very cheap, since comparing the id of the Python object, its memory address, is sufficient.

The motivation for caching sparsities is that they are expensive to construct but only depend on the unique set of arguments they are built from. Caching avoids rebuilding an identical sparsity from identical arguments.

Caching on the underlying `Set` rather than building a global cache means cached objects do not need to be kept around indefinitely, but their lifetime is tied to the lifetime of the `Set` they are cached on. Once all references to the `Set` and the cached objects are lost they become eligible for garbage collection. This strategy can however be defeated by users holding on to references to either of these objects for longer than needed.

The data carriers `Dat`, `Mat` and `Global` are not cached. `Const` objects are globally unique, but for semantic rather than efficiency reasons.

4.3.3 Multiple Backend Support via Unified API

While PyOP2 supports multiple backends, a unified API is provided to the user. This means no changes to user code are required, regardless of which backend the computations are running on. The backend is selected when initialising PyOP2 or by exporting an environment variable and defaults to `sequential` if not set through either mechanism. Once chosen, the backend cannot be changed in the running Python interpreter session.

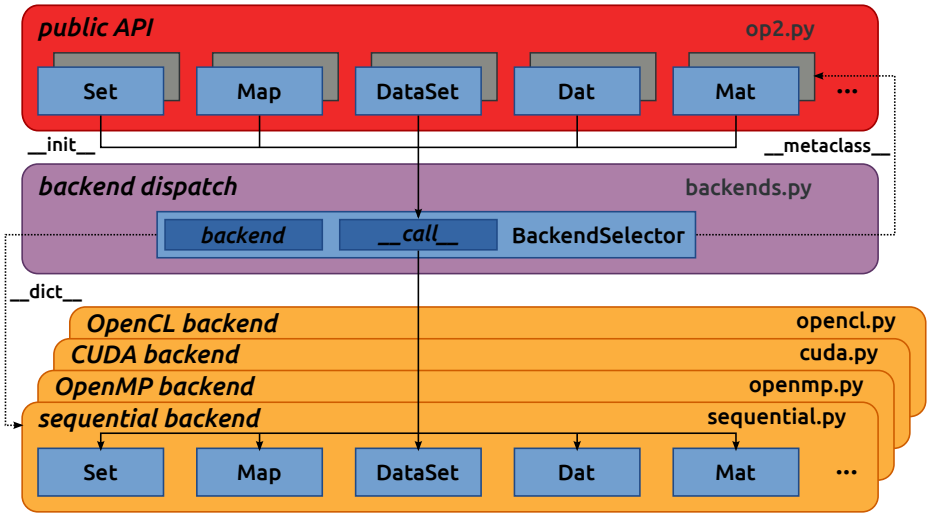

Figure 4.6: PyOP2 backend selection and dispatch mechanism

The implementation of this unified API is achieved with a dispatch mechanism, where all classes and functions that form the public API defined in the `op2` module are proxies for the chosen backend-specific implementations. As illustrated in Figure 4.6, a metaclass, the `BackendSelector`, takes care of instantiating a backend-specific version of the requested class

when an object of such a proxy class is constructed. During PyOP2 initialisation, the `BackendSelector` imports the Python module implementing the backend chosen via the configuration mechanism described above and keeps a handle to it. When an object of a proxy class is constructed, the `BackendSelector`'s `__call__` method looks up the requested class in the backend module namespace, calls its constructor and returns the object. To make this process entirely transparent to the user, the metaclass furthermore takes care of setting docstrings on the proxy class and forwarding any instance and subclass checks to the backend-specific class. As a consequence, the unified API gives the user transparent access to the chosen backend-specific implementation.

This design of the backend selection and dispatch process contained in a single module and completely orthogonal to the implementation of the individual backends significantly simplifies the overall PyOP2 architecture and could be suitably adapted for other projects. Each backend is implemented in its own module, oblivious to the described dispatch process, without any restriction on the use of established object-oriented design practices such as inheritance and imports from other modules.

4.4 Backends

PyOP2 allows writing portable applications, which run on different multi- and many-core architectures without changes to the code, as detailed in Section 4.3.3. Problem- and platform-specific code is generated and compiled at runtime with a tool chain specialised for each backend described in this section. At the time of writing, the supported backends included:

Sequential: Runs sequentially on a single CPU core.

OpenMP: Runs multiple threads on an SMP CPU using OpenMP. The number of threads is set with the environment variable `OMP_NUM_THREADS`.

CUDA: Offloads computation to an NVIDIA GPU.

OpenCL: Offloads computation to an OpenCL device (CPU, accelerator)

For computations running on an accelerator with a dedicated memory space not shared with the host, PyOP2 manages the data in both host and device memory. Data is transferred automatically as needed while minimizing the number of data tranfers as described in Section 4.4.2.

All backends support distributed parallel computations using MPI as detailed in Section 4.7. The CUDA and OpenCL device backends support parallel loops only on `Dats`, while the sequential and OpenMP host backends have full MPI support. Hybrid parallel computations with OpenMP are possible, where `OMP_NUM_THREADS` threads are launched per MPI rank.

4.4.1 Host Backends

Any computation in PyOP2 involves the generation of code at runtime specific to each individual `par_loop`. The host backends generate code which is just-in-time (JIT) compiled using a vendor compiler into a shared library callable as a Python module via *ctypes*, Python's foreign function interface from the standard library. Compiled shared objects are cached on disk, keyed on a hash of the generated code, to save recompilation.

Sequential backend

The code generated for orchestrating a sequential `par_loop` is a C wrapper function with a `for` loop, calling the kernel for each element of the respective iteration set. This wrapper also takes care of staging in and out the data as prescribed by the access descriptors of the parallel loop and provides the kernel with the local view of the data for the current element.

Both kernel and wrapper function are just-in-time compiled in a single compilation unit. The kernel call is inlined and does not incur any function call overhead. It is important to note that this is only possible because the loop marshalling code is also generated. A library calling into code that is just-in-time compiled at runtime cannot benefit from inlining.

Recall the parallel loop calling the `midpoint` kernel from Section 4.2:

```
op2.par_loop(midpoint, cells,
             midpoints(op2.WRITE),
             coordinates(op2.READ, cell2vertex))
```

PyOP2 compiles the following kernel and wrapper code for this loop:

```
inline void midpoint(double p[2], double *coords[2]) {
  p[0] = (coords[0][0] + coords[1][0] + coords[2][0]) / 3.0;
  p[1] = (coords[0][1] + coords[1][1] + coords[2][1]) / 3.0;
}
```

```
void wrap_midpoint(int start, int end,
                   double *arg0_0,
                   double *arg1_0, int *arg1_0_map0_0) {
  double *arg1_0_vec[3];
  for ( int n = start; n < end; n++ ) {
    // Stage in data for the indirect argument
    arg1_0_vec[0] = arg1_0 + (arg1_0_map0_0[n * 3 + 0])* 2;
    arg1_0_vec[1] = arg1_0 + (arg1_0_map0_0[n * 3 + 1])* 2;
    arg1_0_vec[2] = arg1_0 + (arg1_0_map0_0[n * 3 + 2])* 2;
    midpoint(arg0_0 + n * 2, arg1_0_vec);
  }
}
```

Since iteration over subsets is possible, the arguments `start` and `end` define the iteration set indices to iterate over. All remaining arguments are data pointers from NumPy arrays extracted from the `par_loop` access descriptors by ctypes. Variable names are generated to avoid name clashes.

The first argument, `midpoints`, is direct and therefore its data pointer is passed straight to the kernel. Since two double values are associated with each element, the offset is twice the current iteration set element. The second argument `coordinates` is indirect and hence a `Dat`-`Map` pair is passed to the wrapper. Pointers to the data are gathered via the `Map` of arity 3 and staged in the array `arg1_0_vec`, which is passed to the kernel. Each pointer is to two consecutive double values, since there are two `coords` per vertex, which also requires scaling the indirection indices obtained via the map. The indirection is completely hidden from the kernel's point of view and coordinate data is accessed using the local vertex indices 0 to 2.

OpenMP backend

In the OpenMP backend, the loop over the iteration set is annotated with pragmas to execute in parallel with multiple threads, each responsible for a section of iteration set elements. For indirect arguments this may lead to multiple threads trying to update the same value concurrently. A thread safe execution schedule is therefore computed as described in Section 4.5.3, where the iteration set is partitioned and partitions are coloured such that those of the same colour can be safely executed concurrently.

The code generated for the parallel loop from above is as follows:

```
void wrap_midpoint(int boffset, int nblocks,
                   int *blkmap, int *offset, int *nelems,
                   double *arg0_0,
                   double *arg1_0, int *arg1_0_map0_0) {
  double *arg1_0_vec[32][3];
  #pragma omp parallel shared(boffset, nblocks, nelems, blkmap)
  {
    int tid = omp_get_thread_num();
    // Loop over blocks of each colour in parallel
    #pragma omp for schedule(static)
    for ( int __b = boffset; __b < boffset + nblocks; __b++ ) {
      int bid = blkmap[__b];     // Block id
      int nelem = nelems[bid];   // # elements in the block
      int efirst = offset[bid];  // Offset of first element
      for (int n = efirst; n < efirst+ nelem; n++ ) {
        // Stage indirect data into thread private memory
        arg1_0_vec[tid][0] = arg1_0 + (arg1_0_map0_0[n*3 + 0])*2;
        arg1_0_vec[tid][1] = arg1_0 + (arg1_0_map0_0[n*3 + 1])*2;
        arg1_0_vec[tid][2] = arg1_0 + (arg1_0_map0_0[n*3 + 2])*2;
        midpoint(arg0_0 + n * 2, arg1_0_vec[tid]);
      }
    }
  }
}
```

This wrapper is called for each colour with the appropriate number of blocks `nblocks` starting at an initial offset `boffset`. The loop over blocks of each colour can be executed conflict free in parallel and is therefore enclosed in an OpenMP parallel region and annotated with an `omp for` pragma. For each block, the block id `bid` given by the block map `blkmap` is used to look up the number of elements in a given block and its starting index in the arrays `nelems` and `offset` provided by the execution plan. Each thread needs its own staging array `arg1_0_vec`, which is therefore scoped by the thread id. Note that the loop above is direct and hence there is no potential for conflicting writes and no need for colouring.

4.4.2 Device Backends

Device backends target accelerators with dedicated memory spaces. The data carriers `Dat`, `Global` and `Const` therefore have a data array in host memory and a separate array in device memory, which are automatically managed by PyOP2. The `state` flag indicates the present state of a data carrier:

DEVICE_UNALLOCATED No data is allocated on the device.

HOST_UNALLOCATED No data is allocated on the host.

DEVICE Data is up-to-date (valid) on the device, but invalid on the host.

HOST Data is up-to-date (valid) on the host, but invalid on the device.

BOTH Data is up-to-date (valid) on both the host and device.

When a `par_loop` is called, PyOP2 uses the access descriptors to determine which data needs to be allocated or transferred from host to device prior to launching the kernel. Data is only transferred if it is out of date at the target location. All data transfer is triggered lazily, which means the actual copy only occurs once the data is requested. In particular there is no eager transfer back of data from device to host. A transfer is only triggered once data is accessed on the host, avoiding unnecessary transfers. On the other hand this can lead to longer latencies compared to eagerly transferring data, which could potentially overlap with computation.

A newly created device `Dat` has no associated device data and starts out in the state `DEVICE_UNALLOCATED`. Figure 4.7 shows all actions that involve a state transition, which can be divided into three groups: calling explicit data transfer functions (red), access data on the host (black) and using the `Dat` in a `par_loop` (blue). There is no need for users to explicitly initiate data transfers and the transfer functions are only given for completeness.

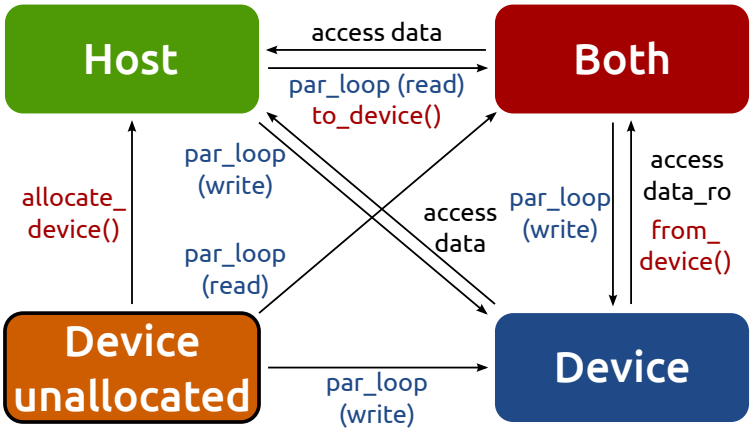

Figure 4.7: State transitions of a data carrier on PyOP2 device backends

When a device `Dat` is used in a `par_loop` for the first time, data is allocated on the device. If the `Dat` is only read, the host array is transferred to device if it was in state `HOST` or `DEVICE_UNALLOCATED` before the `par_loop` and the `Dat` is in the state `BOTH` afterwards, unless it was in state `DEVICE` in which case

it remains in that state. If the `Dat` is written to, data transfer before the `par_loop` is necessary unless the access descriptor is `WRITE`. The host data is out of date afterwards and the `Dat` is in the state `DEVICE`. An overview of the state transitions and necessary memory allocations and data transfers for the two cases is given in Table 4.1.

Initial state	**par_loop read**	**par_loop written to**
`DEVICE_UNALLOCATED`	`BOTH` (alloc, transfer)	`DEVICE` (alloc, transfer unless `WRITE`)
`DEVICE`	`DEVICE`	`DEVICE`
`HOST`	`BOTH` (transfer)	`DEVICE` (transfer unless `WRITE`)
`BOTH`	`BOTH`	`DEVICE`

Table 4.1: Overview of the state transitions and necessary memory allocations (alloc) and host-to-device data transfers (transfer) for data carriers read and written to as parallel loop arguments

Accessing data on the host initiates a device to host data transfer if the `Dat` is in state `DEVICE` and leaves it in state `HOST` when accessing data for reading and writing and `BOTH` when accessing it read-only.

The state transitions described above apply in the same way to a `Global`. A `Const` is read-only, never modified on device and therefore never out of date on the host. Hence there is no state `DEVICE` and it is not necessary to copy back `Const` data from device to host.

CUDA backend

When executing a parallel loop with the CUDA backend, a CUDA kernel is launched on the host, computing over each element of the iteration set simultaneously, replacing the `for` loop used in the host backends as described in 2.3.1. This generated `__global__` stub routine takes care of data marshalling, staging data in fast shared memory and calling the inlined user kernel, which is therefore automatically annotated with a `__device__` qualifier. Kernels require no CUDA-specific modifications by the user.

As with OpenMP, the iteration set is partitioned and the partitions are coloured such that all partitions of the same colour can be executed simultaneously with a single kernel launch. Colouring, kernel launch configuration and resource requirements as well as placement of data in shared memory is computed as part of the parallel schedule described in Section 4.5. Each partition is computed by a block of threads in parallel and requires a second level of colouring for the threads within a block.

The CUDA backend uses PyCUDA's [Klöckner et al., 2012] infrastructure for just-in-time compilation of CUDA kernels and interfacing them to Python. Linear solvers and sparse matrix data structures implemented on top of the CUSP library [Bell et al., 2014] are described in Section 4.6.

PyCUDA automatically generates a host stub for the kernel wrapper generated by PyOP2, given a list of parameter types and unpacks C data pointers from Python objects and NumPy arrays, which allows PyOP2 to launch a CUDA kernel straight from Python. Consider the `midpoint` kernel from previous examples. The generated CUDA code is as follows:

```
__device__ void midpoint(double p[2], double *coords[2])
{
  p[0] = ((coords[0][0] + coords[1][0]) + coords[2][0]) / 3.0;
  p[1] = ((coords[0][1] + coords[1][1]) + coords[2][1]) / 3.0;
}

__global__ void __midpoint_stub(int size, int set_offset,
    double *arg0,
    double *ind_arg1, int *ind_map,
    short *loc_map,   // Offsets of staged data in shared memory
    int *ind_sizes,   // Number of indirectly accessed elements
    int *ind_offs,    // Offsets into indirection maps
    int block_offset, // Offset into the blkmap for current colour
    int *blkmap,      // Block ids
    int *offset,      // Offsets of blocks in the iteration set
    int *nelems,      // Number of elements per block
    int *nthrcol,     // Number of thread colours per block
    int *thrcol,      // Thread colours for each thread and block
    int nblocks) {    // Number of blocks
  extern __shared__ char shared[];
  __shared__ int *ind_arg1_map;
  __shared__ int ind_arg1_size;
  __shared__ double * ind_arg1_s;
  __shared__ int nelem, offset_b, offset_b_abs;
  double *ind_arg1_vec[3];

  if (blockIdx.x + blockIdx.y * gridDim.x >= nblocks) return;
  if (threadIdx.x == 0) {
    int blockId = blkmap[blockIdx.x + blockIdx.y * gridDim.x +
        block_offset];
    nelem = nelems[blockId];
    offset_b_abs = offset[blockId];
    offset_b = offset_b_abs - set_offset;
    ind_arg1_size = ind_sizes[0 + blockId * 1];
    ind_arg1_map = &ind_map[0*size] + ind_offs[0 + blockId*1];
    int nbytes = 0;
    ind_arg1_s = (double *) &shared[nbytes];
  }
```

```
38    __syncthreads();
39
40    // Copy into shared memory
41    for (int idx=threadIdx.x; idx<ind_arg1_size*2; idx+=blockDim.x)
42      ind_arg1_s[idx] = ind_arg1[idx%2+ind_arg1_map[idx/2]*2];
43    __syncthreads();
44
45    // process set elements
46    for ( int idx = threadIdx.x; idx < nelem; idx += blockDim.x ) {
47      ind_arg1_vec[0] = ind_arg1_s + loc_map[0*size+idx+offset_b]*2;
48      ind_arg1_vec[1] = ind_arg1_s + loc_map[1*size+idx+offset_b]*2;
49      ind_arg1_vec[2] = ind_arg1_s + loc_map[2*size+idx+offset_b]*2;
50
51      midpoint(arg0 + 2 * (idx + offset_b_abs), ind_arg1_vec);
52    }
53  }
```

The CUDA kernel `__midpoint_stub` is launched for each colour with a block per partition and 128 threads per block. Inside the kernel each thread is identified by its thread id `threadIdx` within a block of threads identified by a two dimensional block id `blockIdx` within a grid of blocks.

All threads of a thread block have access to a region of fast, on-chip shared memory, which is used as a staging area initialised by thread 0 of each block, (lines 28-37 above). A call to `__syncthreads()` ensures these initial values are visible to all threads of the block. After this barrier, all threads cooperatively gather data from the indirectly accessed `Dat` via the `Map`, followed by another synchronisation. Following that, each thread loops over the elements in the partition with an increment of the block size. In each iteration a thread-private array of pointers to coordinate data in shared memory is built, which is then passed to the `midpoint` kernel. The first argument is directly accessed and passed as a pointer to global device memory with a suitable offset.

OpenCL backend

The OpenCL backend is structurally very similar to the CUDA backend. It uses PyOpenCL [Klöckner et al., 2012] to interface to the OpenCL drivers and runtime. Due to the unavailability of a suitable OpenCL linear algebra backend at the time of writing, linear algebra operations are executed by PETSc [Balay et al., 1997] on the host, as described in Section 4.6.

Consider the `midpoint` kernel from previous examples, which requires no user modification. Parameters in the kernel signature are automatically

annotated with OpenCL storage qualifiers. PyOpenCL provides Python wrappers for OpenCL runtime functions to build a kernel from a code string, set its arguments and enqueue the kernel for execution. It also takes care of extracting C data pointers from Python objects and NumPy arrays. PyOP2 generates the following code for the `midpoint` example:

```
#define ROUND_UP(bytes) (((bytes) + 15) & ~15)

void midpoint(__global double p[2], __local double *coords[2]);
void midpoint(__global double p[2], __local double *coords[2])
{
  p[0] = ((coords[0][0] + coords[1][0]) + coords[2][0]) / 3.0;
  p[1] = ((coords[0][1] + coords[1][1]) + coords[2][1]) / 3.0;
}

__kernel __attribute__((reqd_work_group_size(668, 1, 1)))
void __midpoint_stub(
    __global double* arg0,
    __global double* ind_arg1,
    int size,
    int set_offset,
    __global int* p_ind_map,
    __global short *p_loc_map,
    __global int* p_ind_sizes,
    __global int* p_ind_offsets,
    __global int* p_blk_map,
    __global int* p_offset,
    __global int* p_nelems,
    __global int* p_nthrcol,
    __global int* p_thrcol,
    __private int block_offset) {
  __local char shared [64] __attribute__((aligned(sizeof(long))));
  __local int offset_b;
  __local int offset_b_abs;
  __local int active_threads_count;

  int nbytes;
  int bid;

  int i_1;
  // shared indirection mappings
  __global int* __local ind_arg1_map;
  __local int ind_arg1_size;
  __local double* __local ind_arg1_s;
  __local double* ind_arg1_vec[3];

  if (get_local_id(0) == 0) {
    bid = p_blk_map[get_group_id(0) + block_offset];
    active_threads_count = p_nelems[bid];
    offset_b_abs = p_offset[bid];
```

```
45      offset_b = offset_b_abs - set_offset;
46      ind_arg1_size = p_ind_sizes[0 + bid * 1];
47      ind_arg1_map = &p_ind_map[0 * size] + p_ind_offsets[0+bid*1];
48
49      nbytes = 0;
50      ind_arg1_s = (__local double*) (&shared[nbytes]);
51      nbytes += ROUND_UP(ind_arg1_size * 2 * sizeof(double));
52    }
53    barrier(CLK_LOCAL_MEM_FENCE);
54
55    // staging in of indirect dats
56    for (i_1 = get_local_id(0); i_1 < ind_arg1_size * 2; i_1 +=
          get_local_size(0)) {
57      ind_arg1_s[i_1] = ind_arg1[i_1 % 2 + ind_arg1_map[i_1/2]*2];
58    }
59    barrier(CLK_LOCAL_MEM_FENCE);
60
61    for (i_1 = get_local_id(0); i_1 < active_threads_count; i_1 +=
          get_local_size(0)) {
62      ind_arg1_vec[0] = ind_arg1_s+p_loc_map[i_1+0*size+offset_b]*2;
63      ind_arg1_vec[1] = ind_arg1_s+p_loc_map[i_1+1*size+offset_b]*2;
64      ind_arg1_vec[2] = ind_arg1_s+p_loc_map[i_1+2*size+offset_b]*2;
65
66      midpoint((__global double* __private)(arg0 + (i_1 + offset_b_abs) *
            2), ind_arg1_vec);
67    }
68  }
```

Parallel computations in OpenCL are executed by *work items* organised into *work groups*. OpenCL requires the annotation of all pointer arguments with the memory region they point to: `__global` memory is visible to any work item, `__local` memory to any work item within the same work group and `__private` memory is private to a work item. Local memory therefore corresponds to CUDA's shared memory and private memory is called local memory in CUDA (Table 2.2). The work item id within the work group is accessed via the OpenCL runtime call `get_local_id(0)`, the work group id via `get_group_id(0)`. A barrier synchronisation across all work items of a work group is enforced with a call to `barrier(CLK_LOCAL_MEM_FENCE)`. Bearing these differences in mind, the OpenCL kernel stub is structurally equivalent to the corresponding CUDA version above.

The required local memory size per work group `reqd_work_group_size` is computed as part of the execution schedule and hard coded as a kernel attribute. In CUDA this value is a launch parameter to the kernel.

4.5 Parallel Execution Plan

All PyOP2 backends with the exception of sequential use shared memory parallelism and require an execution schedule to be computed at runtime for each parallel loop. This schedule contains information on the partitioning, staging and colouring of the data for efficient parallel processing and guides both the code generation and execution of parallel loops.

4.5.1 Partitioning

The iteration set is split into a number of equally sized and contiguous mini-partitions such that the working set of each mini-partition fits into shared memory or last level cache. This is orthogonal to the partitioning required for distributed parallelism with MPI described in Section 4.7.

4.5.2 Local Renumbering and Staging

While a mini-partition is a contiguous chunk of the iteration set, the indirectly accessed data it references is not necessarily contiguous. For each mini-partition and unique `Dat-Map` pair, a mapping from local indices within the partition to global indices is constructed as the sorted array of unique `Map` indices accessed by this partition. At the same time, a global-to-local mapping is constructed as its inverse.

Data for indirectly accessed `Dat` arguments on device backends is staged in shared device memory as described in Section 4.4. For each partition, the local-to-global mapping indicates where data to be staged in is read from and the global-to-local mapping gives the location in shared memory where data has been staged at. The amount of shared memory required is computed from the size of the local-to-global mapping.

4.5.3 Colouring

A two-level colouring is used to avoid conflicting writes. Partitions are coloured such that those of the same colour can safely be executed concurrently. On device backends, threads executing on a partition in parallel are coloured such that no two threads indirectly reference the same data. Only `par_loop` arguments performing an indirect reduction (mode `INC`) or assembling a matrix require colouring. Matrices are coloured per row.

For each element of a `Set` indirectly accessed in a `par_loop`, a bit vector is used to record which colours indirectly reference it. To colour each thread within a partition, the algorithm proceeds as follows:

1. Loop over all indirectly accessed arguments and collect the colours of all `Set` elements referenced by the current thread in a bit mask.
2. Choose the next available colour as the colour of the current thread.
3. Loop over all `Set` elements indirectly accessed by the current thread again and set the new colour in their colour mask.

Since the bit mask is a 32-bit integer, up to 32 colours can be processed in a single pass, which is sufficient for most applications. If not all threads can be coloured with 32 distinct colours, the mask is reset and another pass is made, where each newly allocated colour is offset by 32. Should another pass be required, the offset is increased to 64 and so on until all threads are coloured. Thread colouring is shown in Figure 4.8.

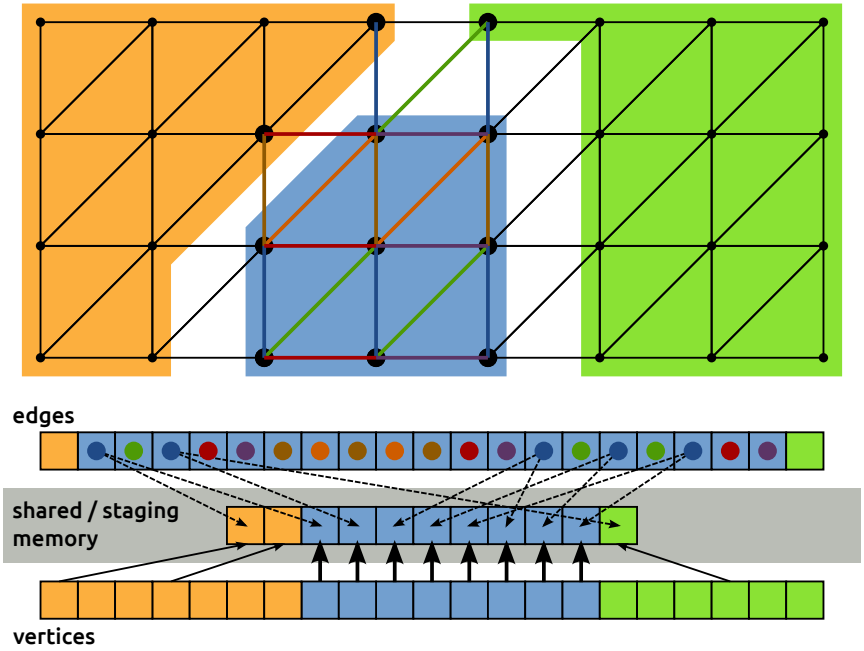

Figure 4.8: Thread colouring within a mini-partition for a `Dat` on vertices indirectly accessed in a computation over the edges. The edges are coloured such that no two edges touch the same vertex within the partition.

The colouring of mini-partitions is done in the same way, except that all `Set` elements indirectly accessed by the entire partition are referenced, not only those accessed by a single thread.

4.6 Linear Algebra interface

Parallel loops can be used to assemble a sparse matrix, represented by a `Mat`, which is declared on a `Sparsity`, representing its non-zero structure. As described in Section 4.1, a sparse matrix is a linear operator that maps a `DataSet` representing its row space to a `DataSet` representing its column space and vice versa. These two spaces are commonly the same, in which case the resulting matrix is square.

The kernel in such a loop describes the *local contribution* and PyOP2 takes care of the necessary *global reduction*, in this case the assembly of a global matrix, using the pair of maps provided with the access descriptor.

PyOP2 interfaces to backend-specific third-party libraries to provide sparse matrix formats, linear solvers and preconditioners. The CUDA backend uses a custom wrapper around the Cusp library [Bell et al., 2014], described in Sections 4.6.4 and 4.6.6. PETSc interfaces to Cusp and ViennaCL to provide matrices and vectors on the GPU, however insertion via device kernels is not supported. Other backends harness the PETSc [Balay et al., 1997] library via its petsc4py [Dalcin et al., 2011] interface.

4.6.1 Sparse Matrix Storage Formats

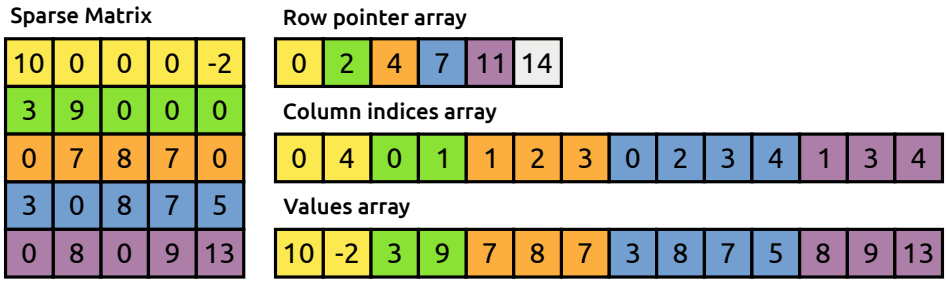

Figure 4.9: A sparse matrix and its corresponding CSR row pointer, column indices and values arrays

PETSc uses the popular Compressed Sparse Row (CSR) format to only store the non-zero entries of a sparse matrix. In CSR, a matrix is stored as three one-dimensional arrays of *row pointers*, *column indices* and *values* as shown in Figure 4.9. Values are stored as floats, usually double precision, and the indices as integer. As the name suggests, non-zero entries are stored per row, where each non-zero is defined by a pair of column index

and corresponding value. The column indices and values arrays therefore have a length equal to the total number of non-zero entries. Row indices are given implicitly by the row pointer array, which contains the starting index in the column index and values arrays for the non-zero entries of each row. In other words, the non-zeros for row `i` are at positions `row_ptr[i]` up to but not including `row_ptr[i+1]` in the column index and values arrays. For each row, entries are sorted by column index to allow for faster lookups using a binary search.

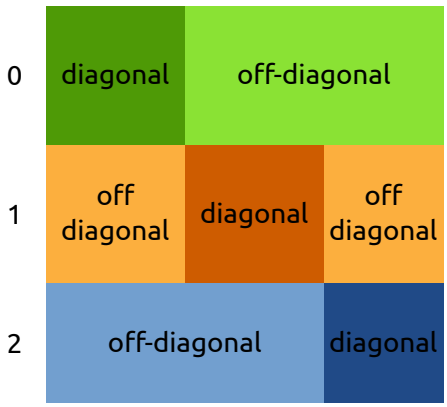

Figure 4.10: Distribution of a sparse matrix among 3 MPI processes

For distributed parallel storage with MPI, the rows of the matrix are distributed evenly among the processors. Each row is then again divided into a *diagonal* and an *off-diagonal* part as illustrated in Figure 4.10. The diagonal part comprises columns `i` to `j` if `i` and `j` are the first and last row owned by a given processor, and the off-diagonal part all other columns.

4.6.2 Building a Sparsity Pattern

The sparsity pattern of a matrix is uniquely defined by the dimensions of its row and column space, and the local-to-global mappings defining its non-zero structure. In PyOP2, row and column space of a `Sparsity` are defined with a pair of `DataSets` and the non-zero entries with one or more pairs of `Maps`. For a valid sparsity, each row and column map must target the `Set` of the row and column `DataSet` respectively, and each pair of maps must have matching origin sets. Since sparsity patterns can be expensive to compute and store, they are cached using these unique attributes as the

cache key. Whenever a `Sparsity` is initialised, an already computed pattern with the same unique signature is returned if it exists.

A frequent occurrence in finite element methods is the assembly of a matrix from a form containing integrals over different entity classes, for example cells and facets. This is naturally supported with multiple parallel loops over different iteration sets assembling into the same matrix, which is declared over a sparsity built from multiple pairs of maps.

Sparsity construction proceeds by iterating each pair of maps and building a set of indices of the non-zero columns for each row. Each pair of entries in the row and column maps gives the row and column index of a non-zero entry in the matrix and therefore the column index is added to the set of non-zero entries for that particular row. The array of non-zero entries per row is then determined as the size of the set for each row and its exclusive scan yields the row pointer array. The column index array is the concatenation of all the sets. The sequential algorithm is given below:

```
for rowmap, colmap in maps:  # Iterate over pairs of maps
  for e in range(rowmap.from_size):  # Iterate over elements
    for r in range(rowmap.arity):
      # Look up row in local-to-global row map
      row = rowmap.values[r + e*rowmap.arity]
      for c in range(colmap.arity):
        # Look up column in local-to-global column map
        diag[row].insert(colmap.values[c + e * colmap.arity])
```

In the MPI parallel case, a set of diagonal and off-diagonal column indices needs to be built for each row as described in 4.6.1:

```
for rowmap, colmap in maps:  # Iterate over pairs of maps
  for e in range(rowmap.from_size):  # Iterate over elements
    for r in range(rowmap.arity):
      # Look up row in local-to-global row map
      row = rowmap.values[r + e*rowmap.arity]
      if row < nrows:  # Drop off-process entries
        for c in range(colmap.arity):
          # Look up column in local-to-global column map
          col = colmap.values[c + e*colmap.arity]
          if col < ncols:  # Insert into diagonal block
            diag[row].insert(col)
          else:  # Insert into off-diagonal block
            odiag[row].insert(col)
```

4.6.3 Matrix Assembly

As described in Section 2.1.7, matrices are assembled by adding up local contributions that are mapped to global matrix entries via a local-to-global mapping, which in PyOP2 is represented by a pair of maps for the row and column space. PyOP2 infers from the access descriptors of a parallel loop whether a matrix is assembled and automatically accumulates the local contributions into a sparse matrix as illustrated in Figure 4.11.

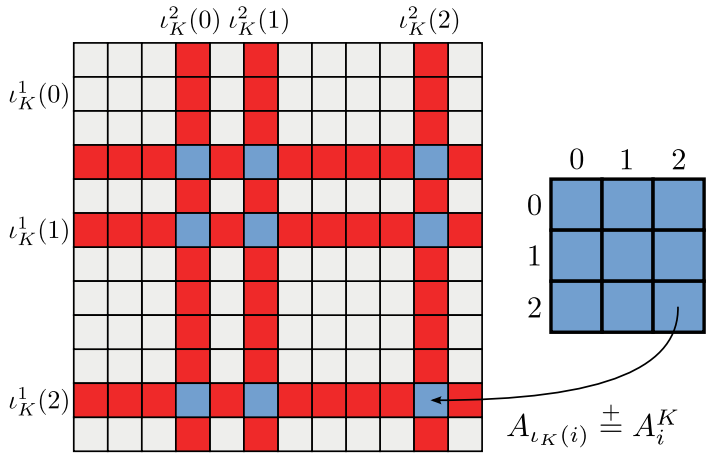

Figure 4.11: Assembly of a local tensor A^K into a global matrix A using the local-to-global mapping ι^1_K for rows and ι^2_K for columns

Consider assembling a matrix `A` on a sparsity defined by a map from triangular `elements` to `nodes`. The assembly `par_loop` iterates over the set of `elements`, where the `elem_node` map defines the local-to-global mapping:

```
nodes = op2.Set(NUM_NODES)
elements = op2.Set(NUM_ELE)
elem_node = op2.Map(elements, nodes, 3, ...)

# Sparsity mapping from nodes to nodes using the elem_node map
sparsity = op2.Sparsity((nodes, nodes), (elem_node, elem_node))
A = op2.Mat(sparsity, np.float64)

# Assemble the matrix A using the local assembly kernel
op2.par_loop(assembly_kernel, elements,
             A(op2.INC,(elem_node[op2.i[0]],elem_node[op2.i[1]])),
             ...)
```

The generated wrapper code for the above `par_loop` with the sequential backend is similar to the following, where initialisation and staging code described in 4.4.1 have been omitted for brevity. For each element of the iteration set a buffer for the local tensor is initialised to zero and passed to the local assembly kernel. The `addto_vector` call is a wrapper around PETSc's `MatSetValues`, adding the local contributions computed by the user kernel to the global matrix using the maps given in the access descriptor. After the loop over the iteration set has finished PyOP2 automatically calls `MatAssemblyBegin` and `MatAssemblyEnd` to finalise matrix assembly.

```
void wrap_mat_kernel__(...) {
  ... // Initialisation code (omitted)
  for ( int n = start; n < end; n++ ) {
    ... // Staging code (omitted)
    // local tensor initialised to 0
    double buffer_arg0_0[3][3] = {{0}};
    // local assembly kernel
    mat_kernel(buffer_arg0_0, ...);
    addto_vector(arg0_0_0, buffer_arg0_0, // Mat, local tensor
                 3, arg0_0_map0_0 + n*3,  // #rows, global row idx
                 3, arg0_0_map1_0 + n*3,  // #cols, global col idx
                 0);                      // mode: 0 add, 1 insert
  }
}
```

4.6.4 GPU Matrix Assembly

When assembling a matrix on the GPU using the CUDA backend, a CSR structure is built in two steps, launching separate kernels. The local contributions are first computed for all elements of the iteration set and stored in global memory in a structure-of-arrays (SoA) data layout such that all threads can write the data in a coalesced manner. For the example above, the generated CUDA wrapper kernel is given below, again omitting initialisation and staging code described in 4.4.2. The user kernel only computes a single element in the local iteration space as detailed in 4.2.3.

```
__global__ void __assembly_kernel_stub(...,
  /* local matrix data array */   double *arg0,
  /* offset into the array   */   int arg0_offset,
                                  ... ) {
  ... // omitted initialisation and shared memory staging code
```

```
  for ( int idx = threadIdx.x; idx < nelem; idx += blockDim.x ) {
    ... // omitted staging code
    for ( int i0 = 0; i0 < 3; ++i0 ) {
      for ( int i1 = 0; i1 < 3; ++i1 ) {
        assembly_kernel(
          (double (*)[1])(arg0 + arg0_offset + idx*9 + i0*3 + i1),
          ..., i0, i1);
      }
    }
  }
}
```

A separate CUDA kernel given below is launched afterwards to compress the data into a sparse matrix in CSR storage format. Only the values array needs to be computed, since the row pointer and column indices have already been computed when building the sparsity on the host and subsequently transferred to GPU memory. Memory for the local contributions and the values array is only allocated on the GPU.

```
__global__ void __lma_to_csr(double *lmadata, // local matrix data
                             double *csrdata, // CSR values array
                             int *rowptr,     // CSR row pointer
                             int *colidx,     // CSR column idx
                             int *rowmap,     // row map array
                             int rowmapdim,   // row map arity
                             int *colmap,     // column map array
                             int colmapdim,   // column map arity
                             int nelems) {
  int nentries_per_ele = rowmapdim * colmapdim;
  int n = threadIdx.x + blockIdx.x * blockDim.x;
  if ( n >= nelems * nentries_per_ele ) return;

  int e = n / nentries_per_ele;                    // set element
  int i = (n - e*nentries_per_ele) / rowmapdim;    // local row
  int j = (n - e*nentries_per_ele - i*colmapdim);  // local column

  // Compute position in values array
  int offset = pos(rowmap[e*rowmapdim+i], colmap[e*colmapdim+j],
                   rowptr, colidx);
  __atomic_add(csrdata + offset, lmadata[n]);
}
```

This structure is naturally extensible to matrix-free methods such as the Local Matrix Approach, which has been demonstrated to be beneficial

for many problems on many-core architectures by Markall et al. [2012]. Instead of building a CSR structure using the kernel above, a custom implementation of the sparse matrix-vector product is provided to be called as a black-box routine by an iterative solver.

4.6.5 Solving a Linear System

PyOP2 provides a `Solver` which wraps the PETSc KSP Krylov solvers [Balay et al., 2013, Chapter 4] which support various iterative methods such as Conjugate Gradients (CG), Generalized Minimal Residual (GMRES), a stabilized version of BiConjugate Gradient Squared (BiCGStab) among others. The solvers are complemented with a range of preconditioners from PETSc's PC collection, which includes Jacobi, incomplete Cholesky and LU decompositions as well as multigrid and fieldsplit preconditioners.

Solving a linear system of the matrix `A` assembled above and the right-hand side vector `b` for a solution vector `x` is done with a call to the `solve` method, where solver and preconditioner are chosen as `gmres` and `ilu`:

```
x = op2.Dat(nodes, dtype=np.float64)

solver = op2.Solver(ksp_type='gmres', pc_type='ilu')
solver.solve(A, x, b)
```

4.6.6 GPU Linear Algebra

Linear algebra on the GPU with the CUDA backend uses the Cusp library [Bell et al., 2014], which supports CG, GMRES and BiCGStab solvers and Jacobi, Bridson approximate inverse and algebraic multigrid preconditioners. The interface to the user is the same as for the sequential and OpenMP backends. An exception is raised if an unsupported solver or preconditioner type is requested. A Cusp solver with the chosen parameters is automatically generated when `solve` is called and subsequently cached.

4.6.7 Vector Operations

A `Dat` represents an opaque vector and as such supports the common vector operations addition, subtraction, multiplication and division both pointwise by another `Dat` or by a scalar, in which case the operation is

broadcast over all its values. In addition, the computation of inner products and reductions is supported. All these operations are implemented in a backend-independent manner using parallel loops.

4.7 Distributed Parallel Computations with MPI

As illustrated in Figure 4.5, all PyOP2 backends support distributed parallel computations with MPI, where the parallelism is abstracted and communication is automatically managed. Sets and maps must be distributed among the processors with partly overlapping partitions. These overlap regions, called *halos*, are required to be able to compute over entities on the partition boundaries and are kept up to date by automatically managing data exchange between neighbouring processors when needed. This section introduces work partly presented by Mitchell [2013].

4.7.1 Local Numbering

The partition of each `Set` local to each process consists of entities *owned* by the process and the *halo*, which are entities owned by other processes but required to compute on the boundary of the owned entities. To efficiently overlap communication and computation and avoid communication during matrix assembly as described below, PyOP2 enforces a constraint on the numbering of the local set entities of each partition, which are therefore partitioned into four contiguous sections. Figure 4.12 illustrates the four sections for a mesh distributed among two processors. Each locally stored `Set` entity belongs to one of these four sections:

Core Entities owned which can be processed without accessing halo data.

Owned Entities owned which need access to halo data when processed.

Exec halo Off-processor entities which are redundantly executed over because they touch owned entities.

Non-exec halo Off-processor entities which are not processed, but read when computing the exec halo.

Data defined on the `Set` is stored contiguously per section, where local `Set` entities must be numbered in order of section, with core entities first, followed by owned, exec halo and non-exec halo. A good partitioning

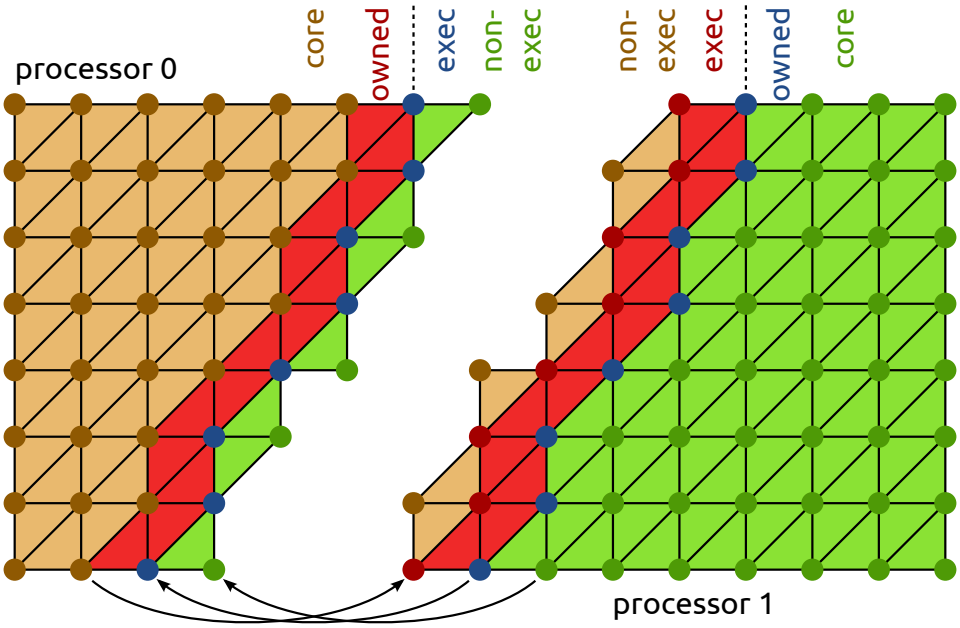

Figure 4.12: A mesh distributed among two processors with the entities of each mesh partition divided into *core*, *owned*, *exec halo* and *non-exec halo*. Matching halo sections are highlighted in matching colours.

maximises the size of the core section and minimises halo regions, such that the vast majority of local entities can be assumed in the core section.

4.7.2 Computation-communication Overlap

The ordering of `Set` entities into four sections allows for a very efficient overlap of computation and communication. While the halo exchange is in progress, core entities, which by construction do not access halo data, can be processed entirely. Execution over the owned and exec halo regions requires up to date halo data and can only start once the halo exchange is completed. Depending on communication latency, bandwidth and the size of the core section relative to the halo, the halo exchange is, in the best case, complete before the computation on the core section has finished.

The entire process for all backends is given in the listing below:

```
halo_exchange_begin()          # Initiate halo exchange
maybe_set_dat_dirty()          # Mark Dats as modified
compute(itset.core_part)       # Compute core region
halo_exchange_end()            # Wait for halo exchange
compute(itset.owned_part)      # Compute owned region
```

```
reduction_begin()                # Initiate reductions
if needs_exec_halo:              # Any indirect Dat not READ?
    compute(itset.exec_part)     # Compute exec halo region
reduction_end()                  # Wait for reductions
maybe_set_halo_update_needed()   # Mark halos as out of date
```

Any reductions depend on data from the core and owned sections and are initiated as soon as the owned section has been processed and execute concurrently with computation on the exec halo. If no action is required for any of the operations above, it returns immediately.

By dividing entities into sections according to their relation to the halo, there is no need to check whether or not a given entity touches the halo or not during computations on each section. This avoids branching in kernels or wrapper code and allows launching separate GPU kernels for execution of each section with the CUDA and OpenCL backends.

4.7.3 Halo exchange

Exchanging halo data is only required if the halo data is actually read, which is the case for `Dats` used as arguments to parallel loops in `READ` or `RW` mode. PyOP2 keeps track whether or not the halo region may have been modified and marks them as out of date. This is the case for `Dats` used in `INC`, `WRITE` or `RW` mode or when a `Solver` or a user requests access to the data. A halo exchange is performed only for halos marked as out of date.

4.7.4 Distributed Assembly

For an MPI distributed matrix or vector, assembling owned entities at the boundary can contribute to off-process degrees of freedom and vice versa.

There are different ways of accounting for these off-process contributions. PETSc supports insertion with local stashing and subsequent communication of off-process matrix and vector entries, however its implementation is not thread safe. Concurrent insertion into PETSc MPI matrices *is* thread safe if off-process insertions are not cached and concurrent writes to rows are avoided, which is done for the OpenMP backend through colouring as described in Section 4.5.3.

PyOP2 therefore disables PETSc's off-process insertion feature, which saves the additional communication step to exchange off-processor ma-

trix entries when finalising the global assembly process. Instead, all off-process entities which are part of the *exec halo* section described above are redundantly computed over. Maintaining a larger halo, the *non-exec halo* section, is required to perform the redundant computation. Halos grow by about a factor two, however in practice this is still small compared to the interior region of a partition. The main cost of halo exchange is the latency, which is independent of the exchanged data volume.

4.8 Mixed Types

When solving linear systems of equations as they arise for instance in the finite element method (FEM), one is often interested in *coupled* solutions of more than one quantity. In fluid dynamics, a common example is solving a coupled system of velocity and pressure as it occurs in some formulations of the Navier-Stokes equations. PyOP2 naturally supports such use cases by providing generalised block-structured data types, which mirror the structure of coupled systems. These are a crucial prerequisite for designing mixed function spaces in Firedrake, described in Section 5.2.

4.8.1 Mixed Set, DataSet, Map and Dat

PyOP2 provides a range of mixed types, lightweight containers which do not own any data and are instantiated by combining the element elementary data types `Set`, `DataSet`, `Map` and `Dat` into a `MixedSet`, `MixedDataSet`, `MixedMap` and `MixedDat` respectively. Mixed types provide the same attributes and methods as their base types allow iteration over their constituent parts[3]. This design allows mixed and non-mixed types to be used interchangeably and simplifies implementation by not having to special case code.

4.8.2 Block Sparsity and Mat

Sparsity patterns for coupled linear systems exhibit a characteristic block structure, which PyOP2 exploits when declaring a `Sparsity` from pairs of mixed maps. Such a sparsity is composed of elementary sparsities arranged in a square block structure with as many block rows and columns

[3]For consistency and convenience, base types yield themselves when iterated.

as there are components in the `MixedDataSet` forming its row and column space. In the most general case a `Sparsity` is constructed as follows:

```
it = op2.Set(...)  # Iteration set, not mixed
sr0, sr1 = op2.Set(...), op2.Set(...)  # Sets for row spaces
sc0, sc1 = op2.Set(...), op2.Set(...)  # Sets for column spaces
# MixedMaps for the row and column spaces
mr = op2.MixedMap([op2.Map(it, sr0, ...), op2.Map(it, sr1, ...)])
mc = op2.MixedMap([op2.Map(it, sc0, ...), op2.Map(it, sc1, ...)])
# MixedDataSets of dim 1 for the row and column spaces
dsr = op2.MixedDataSet([sr0**1, sr1**1])
dsc = op2.MixedDataSet([sc0**1, sc1**1])
# Blocked sparsity
sparsity = op2.Sparsity((dsr, dsc), [(mr, mc), ...])
```

The relationships of each component of the mixed maps and datasets to the blocks of the `Sparsity` is shown in Figure 4.13.

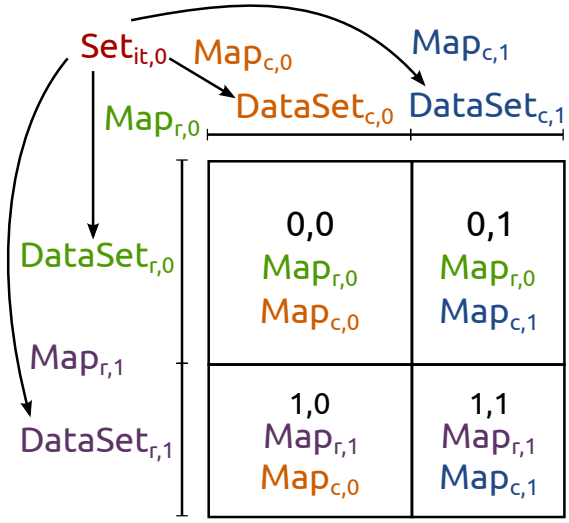

Figure 4.13: The contribution of sets, maps and datasets to the blocked sparsity

Although not a separate type, a block sparsity is a container for the sparsity objects forming each block, similar to the other mixed types described above. Sparsity patterns for each block are computed separately using the same code path described in Section 4.6.2 and the same validity rules apply. A `Mat` defined on a block `Sparsity` inherits the block structure and is implemented using a PETSc MATNEST [Balay et al., 2013, Section 3.1.3], where the nested submatrices are stored separately.

4.8.3 Mixed Assembly

Assembling a coupled system into a mixed vector or matrix is usually done with a single parallel loop and kernel. The local iteration space as seen by this kernel is a combination of local iteration spaces of all the sub-blocks of the vector or matrix. PyOP2 ensures that indirectly accessed data is gathered and scattered via the maps corresponding to each sub-block and packed together into a contiguous vector to be passed to the kernel. This combined local iteration space is, however, logically block structured and PyOP2 takes care of assembling contributions from the local tensor into the corresponding blocks of the `MixedDat` or `Mat`.

To orchestrate this computation, an unrolled loop over the two dimensional block structure of the iteration space is generated, accumulating contributions of each block into the corresponding submatrix as described in Section 4.6.3. The same code path is used for assembling regular elementary vectors and matrices, where the iteration space only consists of the $(0,0)$ block such that no special casing is necessary.

Consider the following example loop assembling a block matrix:

```
it, cells, nodes = op2.Set(...), op2.Set(...), op2.Set(...)
mds = op2.MixedDataSet([nodes, cells])
mmap = op2.MixedMap([op2.Map(it, nodes, 2, ...),
                     op2.Map(it, cells, 1, ...)])
mat = op2.Mat(op2.Sparsity(mds, mmap))
d = op2.MixedDat(mds)
op2.par_loop(kernel, it,
             mat(op2.INC, (mmap[op2.i[0]], mmap[op2.i[1]])),
             d(op2.READ, mmap))
```

The `kernel` for this `par_loop` assembles a 3×3 local tensor and is passed an input vector of length 3 for each iteration set element:

```
void kernel(double v[3][3] , double **d ) {
  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      v[i][j] += d[i][0] * d[j][0];
}
```

The top-left 2×2 block of the local tensor is assembled into the $(0,0)$ block of the matrix, the top-right 2×1 block into $(0,1)$, the bottom-left $1\times$ block into $(1,0)$ and finally the bottom-right 1×1 block into $(1,1)$. Figure 4.14 illustrates the assembly of the block `Mat`.

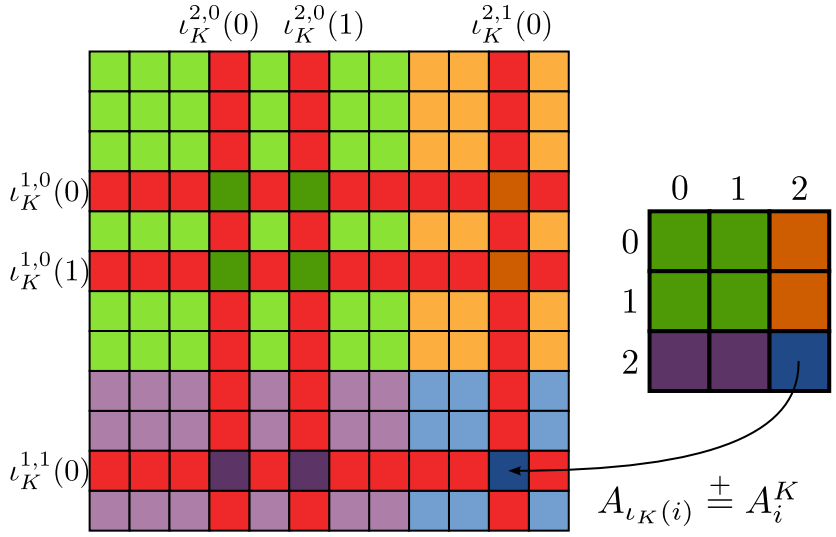

Figure 4.14: Assembling into the blocks of a global matrix A: block $A^{0,0}$ uses maps $\iota^{1,0}$ and $\iota^{2,0}$, $A^{0,1}$ uses $\iota^{1,0}$ and $\iota^{2,1}$, $A^{1,0}$ uses $\iota^{1,1}$ and $\iota^{2,0}$ and finally $A^{1,1}$ uses $\iota^{1,1}$ and $\iota^{2,1}$ for the row and column spaces respectively.

4.9 Comparison with OP2

PyOP2 shares fundamental concepts with OP2, described in Section 3.3, however differs in several important design decisions. These lead to very different implementations which do not share code. Among the shared concepts are the description of the topology of an unstructured mesh with sets and maps and the uniform execution of a kernel over an iteration set. Both OP2 and PyOP2 support performance-portable computations on different backends from a single source through code generation. While PyOP2 allows the backend to be selected at runtime, an OP2 application is compiled and linked for a particular backend. Switching the backend requires invoking the appropriate code generator, compiling the application and linking against backend-specific runtime support libraries.

Similarly, PyOP2 dynamically generates code at runtime by inspecting objects and data structures, while OP2 relies on static analysis of an input programme, which is transformed into a backend-specific implementation through source-to-source translation at compile time. All information needed for code generation must be gleaned by parsing the user programme. In practice, this analysis is limited to the parallel loop call itself,

since OP2's custom translator, relying on regular expression matching and string substitution, cannot reliably backtrack to variable declarations. Access to declarations would either depend upon a more sophisticated analysis, capable of tracing variables from the point of declaration to the point of use, or require to limit the control flow allowed in a user programme. Even that is not sufficient to support the general case, where an argument can be a variable only known at runtime.

A particular consequence of this limitation is the requirement to repeat access descriptors for indirectly accessed arguments, explicitly specifying the index into the map, since the map's arity cannot be determined.

In the following, a parallel loop call for the `adt_calc` kernel in the Airfoil example application, which is part of both the OP2 and PyOP2 distributions, is compared to highlight the differences. Airfoil is a finite volume code operating on a quadrilateral mesh. The `adt_calc` kernel, executed over cells, accesses `p_x` indirectly via the map `pcell` from cells to vertices of arity four, and `p_q` and `p_adt` directly. Consider the OP2 parallel loop call:

```
op_par_loop(adt_calc,"adt_calc",cells,
            op_arg_dat(p_x,   0,pcell, 2,"double",OP_READ ),
            op_arg_dat(p_x,   1,pcell, 2,"double",OP_READ ),
            op_arg_dat(p_x,   2,pcell, 2,"double",OP_READ ),
            op_arg_dat(p_x,   3,pcell, 2,"double",OP_READ ),
            op_arg_dat(p_q,  -1,OP_ID, 4,"double",OP_READ ),
            op_arg_dat(p_adt,-1,OP_ID, 1,"double",OP_WRITE));
```

Access descriptors for `op_dat` arguments are explicitly instantiated as an `op_arg_dat`, which takes the `op_dat`, the index into the map, the map itself, the size and data type of the `op_dat` and the access mode, where index and map are -1 and `OP_ID` for direct arguments. The equivalent PyOP2 parallel loop call shown below is much more compact. Access descriptors are created directly from `Dats`, only passing the access mode and the map for indirect access. Since PyOP2 can determine the arity from the map, there is no need to explicitly specify an index, even though this is supported. For most applications, the kernel accesses all associated entities, which is the default PyOP2 behaviour for non-indexed maps. Similarly, the shape and type of data are queried from the `Dat` and need not be repeated:

```
op2.par_loop(adt_calc, cells,
             p_x(op2.READ, pcell),
             p_q(op2.READ),
             p_adt(op2.WRITE))
```

Being embedded in Python, PyOP2 provides a significantly more compact, clean, readable and expressive DSL syntax. More importantly, the choice of host language and the runtime nature have enabled several design decisions and features that would not have been possible with OP2, such as the backend dispatch, dynamic data structures, support for matrices, linear algebra operations and mixed types. In particular the last three are crucial prerequisites for applications such as finite element computations and extensively used by Firedrake, described in Chapter 5.

4.10 Conclusions

In this chapter, PyOP2 has been demonstrated to be a high-level versatile abstraction for parallel computations on unstructured meshes, supporting a wide range of hardware architectures through a unified API. At runtime, domain knowledge is exploited to generate efficient, problem-specific, low-level code taylored to each platform. Applications built on top of PyOP2 are therefore immediately portable and can execute on any supported backend without requiring code changes.

Data storage, layout and transfer as well as parallel computations and MPI communication of vector and matrix data are managed for the user by PyOP2, whose data structures form suitable building blocks for higher level constructs, encapsulating the topology of unstructured meshes and the data defined on them. A `Dat` is a completely abstracted representation of a vector, where the actual values may be stored in CPU or GPU memory, depending on which backend computations are running on.

PyOP2's conceptual abstraction is applicable to a variety of different kinds of computations on unstructured meshes or fixed-degree graphs. The support for sparse matrix assembly and linear algebra as well as mixed problems make PyOP2 a suitable execution layer for a broad class of scientific applications. In particular, these features are crucial to effectively supporting finite element computations as demonstrated by Firedrake described in the next chapter.

Chapter 5

Firedrake - A Portable Finite Element Framework

Firedrake is a high-level framework for solving linear and non-linear finite element problems described as variational forms on discrete function spaces. Solving such a problem numerically involves a number of mathematical operations such as formulating a variational problem, assembling forms, manipulating functions, and solving the variational problem. Firedrake abstracts this process and provides high-level representations of these mathematical operations in Python code, which are themselves efficiently implemented as compositions of different, lower level abstractions employed by Firedrake, as explained in this chapter.

The central abstraction is PyOP2, described in the previous chapter, used as the parallel execution layer. Firedrake does not directly manipulate any field data. Instead, all computation and manipulation of data is done exclusively via parallel loops (Section 4.1.3) and is therefore inherently backend-independent and performance portable. PyOP2 also manages storage, layout, transfer of data as well as any communication and exchange of halo data between processes when running in parallel.

Management, distribution and renumbering of the unstructured mesh topology and the solution of linear and non-linear systems is handled by the PETSc abstraction, introduced in Section 3.1.1, which is partly leveraged via PyOP2's linear algebra interface described in Section 4.6. The mesh and non-linear solver interfaces are implemented in Firedrake.

The highly sucessful UFL abstraction (Section 3.2.2) is used for the de-

scription of variational forms. Firedrake internally employs a customised version of the FEniCS form compiler FFC (Section 3.2.3) for compiling forms and FIAT (Section 3.2.4) for tabulating local basis functions.

In its design, Firedrake follows a clear separation of concerns. Most operations are closed over their abstractions, which means they return Firedrake objects, unless they are supposed to lower the abstraction, in which case they return a first class object of the layer below.

The API exposed by Firedrake is intentionally compatible to DOLFIN introduced in Section 3.2.1, with a few exceptions, such as strong boundary conditions. If they wish however, users may extract the underlying PyOP2, PETSc or UFL objects and inspect or manipulate those directly.

As a consequence of these design principles, Firedrake is purley a system for reasoning about variational forms with a very compact and maintainable code base, which contains no parallel code, since all parallelism and communication is handled by either PyOP2 or PETSc.

This chapter begins by introducing concepts and constructs fundamental to Firedrake and the definition of variational problems in Section 5.1. Details on the treatment of mixed function spaces are given in Section 5.2. Assembling expressions and variational forms is described in Sections 5.3 and 5.4 and the application of strong boundary conditions in Section 5.5. Solving linear and non-linear systems of equations is detailed in Section 5.6 and a comparison to the DOLFIN/FEniCS tool chain is presented in Section 5.7, before the chapter concludes with Section 5.8.

5.1 Concepts and Core Constructs

To solve a variational problem, starting from the strong form of a partial differential equation, a weak variational form is derived by choosing suitable discrete function spaces for test and trial functions and any coefficients present in the form as described in Section 2.1.

These discrete function spaces are defined on a discretised domain represented by a mesh and characterised by finite element basis functions of a certain family and degree. Firedrake represents coefficients and unknown solutions as functions defined on these function spaces. A diagram of the Firedrake core classes `Mesh`, `FunctionSpace` and `Function` and their associated PyOP2, PETSc and UFL objects is given in Figure 5.1.

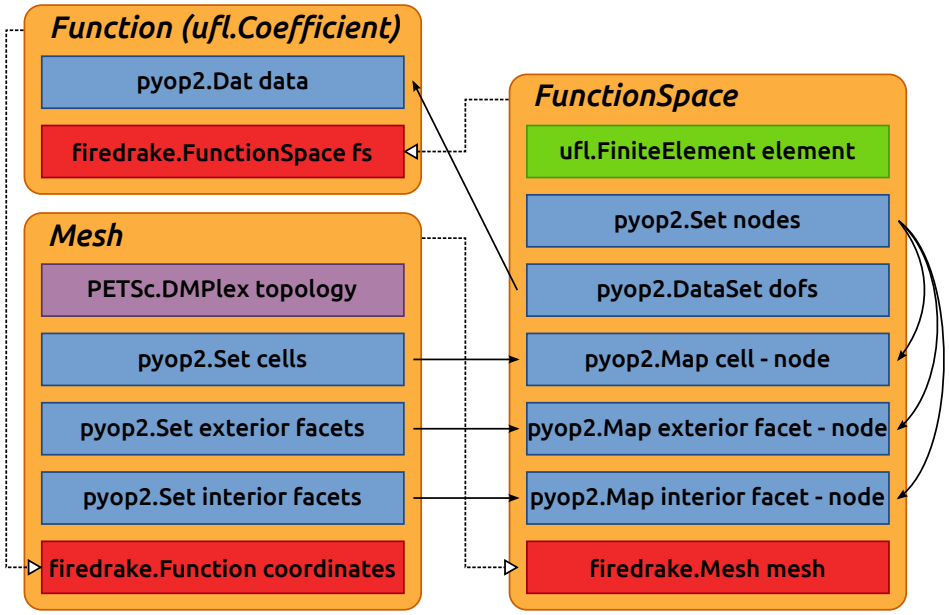

Figure 5.1: Firedrake core classes and their associated PyOP2, PETSc and UFL objects

5.1.1 Functions

In finite element terminology, a field is data defined on a set of degrees of freedom (DOFs), which is exactly the representation in a PyOP2 `Dat`, described in Section 4.1.2. Firedrake therefore uses `Dats` to store the values of fields, represented as a `Function`. When a `Function` is created, Firedrake instructs PyOP2 to allocate a new `Dat` on the function's DOF `DataSet`.

This choice exemplifies two of Firedrake's design principles. The first is the principle of single responsibility: functions are the only objects in Firedrake that carry field data and all fields are functions. The other is the clear separation of abstractions: PyOP2 is used as the portable computation layer and responsible for all data storage. As far as Firedrake is concerned, a `Dat` is a fully abstracted, distributed vector, whose data might live in CPU or GPU memory, depending on the chosen PyOP2 backend.

5.1.2 Function Spaces

A function is defined on a function space, which defines the DOFs and their relationship to the mesh topology. On a given mesh, a function space is characterised by a family and degree of finite element basis functions.

Upon creation of a function space, Firedrake obtains the number and distribution of local DOFs on the finite element of given family and degree from FIAT, described in Section 3.2.4. This information is used to define the `Set` of global DOFs and a global numbering of entities conforming to the PyOP2 numbering requirements for distributed parallel computations, described in Section 4.7. When running in parallel, this includes the definition of halo regions. Global numberings are also computed for DOFs on exterior and interior facets if required. Firedrake uses the renumbering support of PETSc's DMPlex module for distributed unstructured meshes [Balay et al., 2013, Chapter 17] to compute the global numbering.

PyOP2 maps, defining the connectivity to the mesh topology as shown in Figure 5.1, are created from these numberings as required by assembly computations detailed in Section 5.4. Since a function space of a given family and degree is unique and the computation of the numbering expensive, function spaces are only created once and cached on the mesh.

Firedrake provides a `FunctionSpace` for function spaces with scalar degrees of freedom, `VectorFunctionSpace` for vector- and `TensorFunctionSpace` for tensor-valued degrees of freedom, such as velocity or diffusivity.

5.1.3 Meshes

A mesh representing a discretised domain defines the abstract topology, that is how mesh entities such as cells, edges, vertices and facets are connected. The concrete geometry is given by the coordinates of vertices in 2D or 3D space. In Firedrake, the topology is described by sets of entities and maps between them using PyOP2 as described in Section 4.1.1.

Geometric information is stored as a `Function` for the coordinate field defined on a Lagrange vector function space of degree one, computed as described above. The coordinate field uses the same conforming global numbering, which implicitly defines the connectivity between cells and vertices of the mesh, so that there is no need to store this information separately. The only data associated with the mesh is its DMPlex, the coordinate `Function`, a `Set` of cells and data on exterior and interior facets.

Unlike many finite element frameworks, the coordinates are not treated specially in any way. Instead, coordinates are a `Function` like any other and can be reasoned about and manipulated in the same way. For instance, it

is straightforward to scale, rotate or otherwise transform the mesh simply by performing a computation on the coordinate `Function`.

When running in parallel with MPI, Firedrake takes care of decomposing the mesh among processors transparently, using PETSc's DMPlex module for distributed unstructured meshes [Balay et al., 2013, Chapter 17]. Firedrake delegates reading meshes in Gmsh [Geuzaine and Remacle, 2009], CGNS [Poirier et al., 1998] and EXODUS [Mills-Curran et al., 1988] format to DMPlex and adds support for the Triangle [Shewchuk, 1996] format. A number of utility mesh classes are provided for uniform discretisations of standard domains such as intervals in 1D, rectangles and circles in 2D and cubes and spheres in 3D.

Immersed manifolds

Firedrake also supports solving problems on orientable immersed manifolds. These are meshes in which the entities are *immersed* in a higher dimensional space such as the surface of a sphere in 3D [Rognes et al., 2013]. In this case, the geometric dimension of the coordinate field is not the same as the topological dimension of the mesh entities.

Semi-structured extruded meshes

In order to support the solution of PDEs on high-aspect ratio domains, such as in the ocean or atmosphere, the numerics dictate that the "short" dimension should be structured. Firedrake supports solving such problems on *extruded meshes*, which are built by extruding an unstructured base mesh for a given number of layers to form this structured "short" dimension. Non-uniform layer heights can be computed using a PyOP2 kernel and radial extrusion is also supported.

5.1.4 Expressing Variational Problems

Firedrake uses the high-level language UFL (Section 3.2.2) to describe variational problems. The Firedrake classes `Mesh`, `FunctionSpace` and `Function` provide the same interface to the user as their DOLFIN equivalents (Section 3.2.1), albeit with an entirely different implementation.

A first variational form

As an example, consider the identity equation on a unit square Ω:

$$u = f \text{ on } \Omega \tag{5.1}$$

Test and trial functions on a function space using piecewise linear polynomials on a unit square mesh are obtained as follows:

```
mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)

u = TrialFunction(V)
v = TestFunction(V)
```

It is worth noting that, despite their name, test and trial functions do not represent a Firedrake `Function`, but are purely symbolic objects only used in the context of a UFL variational form.

A function to hold the right hand side f is populated with the x component of the coordinate field:

```
f = Function(V).interpolate(Expression('x[0]'))
```

The variational formulation of (5.1) is: find $u \in V$ such that

$$\int_\Omega uv \, \mathrm{d}x = \int_\Omega fv \, \mathrm{d}x \quad \forall v \in V \tag{5.2}$$

and is defined in UFL as

```
a = u * v * dx
L = f * v * dx
```

where the measure `dx` indicates that the integration should be carried out over the cells of the mesh. UFL can also express integrals over the boundary of the domain (`ds`) and the interior facets of the domain (`dS`).

As described in more detail in Section 5.6, the resulting variational problem is solved for a function `x` as follows:

```
x = Function(V)
solve(a == L, x)
```

Incorporating boundary conditions

Boundary conditions enter the variational problem in one of two ways. *Natural* (often termed *Neumann* or *weak*) boundary conditions, which pre-

scribe values of the derivative of the solution, are incorporated into the variational form. *Essential* (often termed *Dirichlet* or *strong*) boundary conditions, which prescribe values of the solution, become prescriptions on the function space. In Firedrake, the former are naturally expressed as part of the formulation of the variational problem, the latter are represented by the class `DirichletBC` and are applied when solving the variational problem, as described in Section 5.5. A strong boundary condition is imposed in a function space, setting degrees of freedom on a given subdomain, defined by the mesh generator, to a given value:

```
bc = DirichletBC(V, value, subdomain_id)
```

Strong boundary conditions are prescribed by passing a list of boundary condition objects to the `solve` call:

```
solve(a == L, bcs=[bc])
```

Alternatively, they can be given when assembling a form into a tensor:

```
A = assemble(a, bcs=[bc])
b = assemble(L, bcs=[bc])
```

Finally, boundary conditions can also be explicitly applied to a tensor:

```
bc.apply(A)
bc.apply(b)
```

It is important to note that the method Firedrake utilises internally for applying strong boundary conditions described in Section 5.5 does not destroy the symmetry of the linear operator. If the system without boundary conditions is symmetric, it will continue to be so after the application of any boundary conditions.

5.2 Mixed Function Spaces

Many finite element problems involve some form of coupling between different fields, such as between the pressure and velocity in a fluid flow. Such problems are therefore commonly modeled with mixed function spaces, which are combinations of the function spaces of each of the fields and treated as if they were stacked on top of each other. The resulting coupled systems exhibit a block structure, which is readily expressible using PyOP2 mixed types introduced in Section 4.8.

5.2.1 Mixed Formulation for the Poisson Equation

Consider the Poisson equation $\nabla^2 u = -f$ using a mixed formulation on two coupled fields [Rognes, 2012]. Introducing the negative flux $\sigma = \nabla u$ as an auxiliary vector-valued variable results in the following PDE on a domain Ω with boundary $\Gamma = \Gamma_D \cup \Gamma_N$

$$\sigma - \nabla u = 0 \text{ on } \Omega \qquad \nabla \cdot \sigma = -f \text{ on } \Omega \tag{5.3}$$

$$u = u_0 \text{ on } \Gamma_D \qquad \sigma \cdot n = g \text{ on } \Gamma_N \tag{5.4}$$

for some known functions f and g. The solution to this equation will be some functions $u \in V$ and $\sigma \in \Sigma$ for some suitable function spaces V and Σ. Multiply by arbitrary test functions $\tau \in V$ and $\nu \in \Sigma$, integrate over the domain and then integrate by parts to obtain a weak formulation of the variational problem: find $\sigma \in \Sigma$ and $\nu \in V$ such that:

$$\int_\Omega (\sigma \cdot \tau + \nabla \cdot \tau \, u) \, \mathrm{d}x = \int_\Gamma \tau \cdot n \, u \, \mathrm{d}s \quad \forall \, \tau \in \Sigma, \tag{5.5}$$

$$\int_\Omega \nabla \cdot \sigma v \, \mathrm{d}x = -\int_\Omega f \, v \, \mathrm{d}x \quad \forall \, v \in V. \tag{5.6}$$

The flux boundary condition $\sigma \cdot n = g$ becomes an *essential* boundary condition to be enforced on the function space, while the boundary condition $u = u_0$ turns into a *natural* boundary condition which enters into the variational form, such that the variational problem can be written as: find $(\sigma, u) \in \Sigma_g \times V$ such that

$$a((\sigma, u), (\tau, v)) = L((\tau, v)) \quad \forall \, (\tau, v) \in \Sigma_0 \times V \tag{5.7}$$

with the variational forms a and L defined as

$$a((\sigma, u), (\tau, v)) = \int_\Omega \sigma \cdot \tau + \nabla \cdot \tau \, u + \nabla \cdot \sigma \, v \, \mathrm{d}x \tag{5.8}$$

$$L((\tau, v)) = -\int_\Omega f v \, \mathrm{d}x + \int_{\Gamma_D} u_0 \tau \cdot n \, \mathrm{d}s \tag{5.9}$$

The essential boundary condition is reflected in the function spaces $\Sigma_g = \{\tau \in H(\mathrm{d}iv) \text{ such that } \tau \cdot n|_{\Gamma_N} = g\}$ and $V = L^2(\Omega)$.

A stable combination of discrete function spaces $\Sigma_h \subset \Sigma$ and $V_h \subset V$ to

Listing 5.1: Mixed Poisson problem formulated in Firedrake.

```
V = FunctionSpace(mesh, "BDM", 1)
Q = FunctionSpace(mesh, "DG", 0)
W = V * Q

sigma, u = TrialFunctions(W)
tau, v = TestFunctions(W)
f = Function(Q)

a = (dot(sigma, tau) + div(tau)*u + div(sigma)*v)*dx
L = - f*v*dx
```

form a mixed function space $\Sigma_h \times V_h$ is Brezzi-Douglas-Marini elements [Brezzi et al., 1985] of polynomial order k for Σ_h and discontinuous elements of polynomial order $k-1$ for V_h. Listing 5.1 shows this problem formulated in Firedrake[1] for $k = 1$ and $u_0 = 0$.

5.2.2 Mixed Elements, Test and Trial Functions in UFL

The mixed function space `W` is obtained by combining the function spaces `V` and `Q` using the `*` operator, where `V` is the first and `Q` the second subspace in `W`. Test and trial functions for these subspaces are extracted via `TrialFunctions` and `TestFunctions`, which return an ordered tuple of indices into the mixed test and trial functions. Note that this is fundamentally different from creating separate test and trial functions on the spaces `V` and `Q`, which would have no connection to the mixed space `W`.

The test and trial spaces on `W` have dimension 3, where index 0 and 1 refer to the Brezzi-Douglas-Marini space, which is a vector valued function space of dimension two, and index 2 refers to the scalar valued discontinuous Galerkin space. The subspaces `sigma` and `u` are represented as shown in Figure 5.2 and 5.3 and `tau` and `v` analogous but for the test function.

Maintaining the relationship to the parent mixed space allows UFL to infer the full shape of the form, which is given by the mixed test and trial spaces, even though only the subspace arguments are used in defining the form. The index is used keep track of the position these subspace arguments belong to in the form in whichever expression they are used.

Consider the linear form `L` in Listing 5.1, which contains the coefficient

[1] This problem is implemented as a demo in DOLFIN and Firedrake.

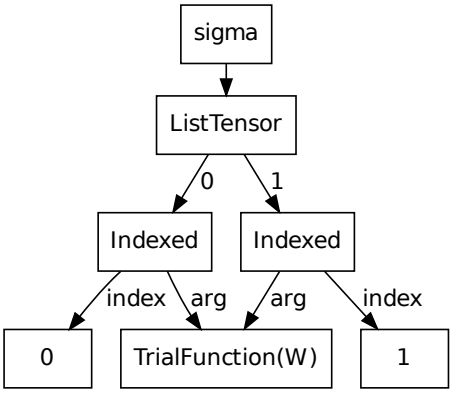

Figure 5.2: UFL expression tree for σ

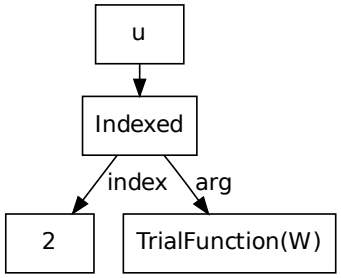

Figure 5.3: UFL expression tree for u

`f` defined on `Q` and the test function `v` defined on the second subspace of `W`. This information allows UFL to infer that this form is defined on a mixed space, but there is only a contribution to its second subspace. Similar information can be inferred for the bilinear form a shown in Figure 5.4. Substituting the representations for σ, τ, u and v as shown in Figures 5.2 and 5.3 gives the final expression tree for a shown in Figure 5.5.

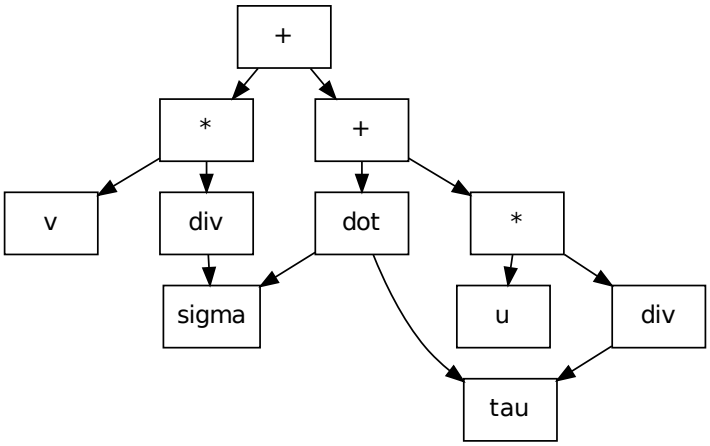

Figure 5.4: Simplified UFL expression tree for the mixed Poisson formulation

5.2.3 Mixed Systems

The bilinear form a corresponds to a linear operator with a block structure given by the contributions from each component of the mixed trial space

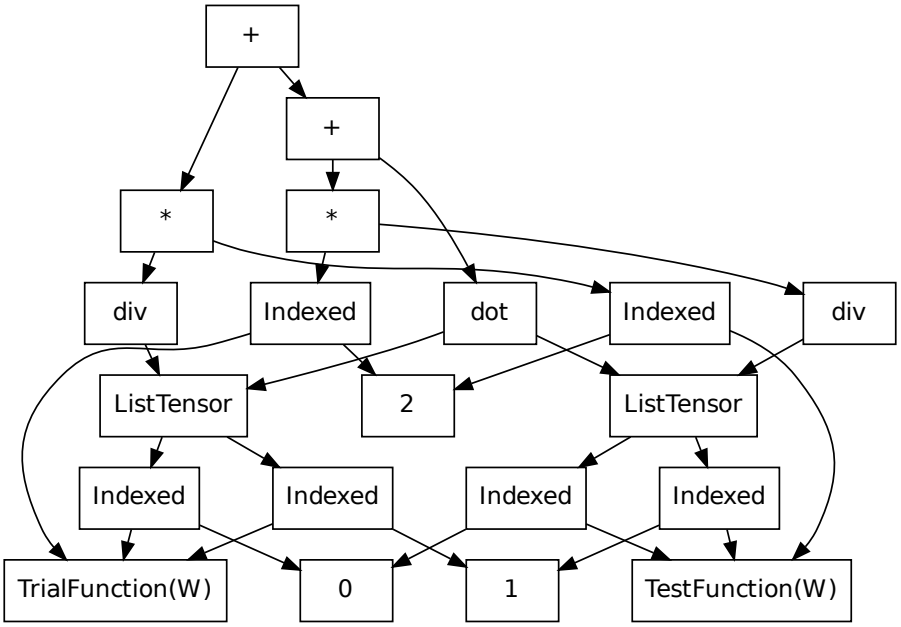

Figure 5.5: UFL expression tree for the mixed Poisson formulation

(σ, u) and test space (τ, v):

$$\begin{pmatrix} a(\sigma,\tau) & a(u,\tau) \\ a(\sigma,v) & a(u,v) \end{pmatrix}. \tag{5.10}$$

Identifying each term in a results in the system

$$\begin{pmatrix} \langle\sigma,\tau\rangle & \langle\mathrm{div}(\tau),u\rangle \\ \langle\mathrm{div}(\sigma),v\rangle & 0 \end{pmatrix} \tag{5.11}$$

where there is no contribution to the lower right block.

As described in Section 4.8.3, assembling such a coupled monolithic system involves picking apart the contributions of the local assembly kernel to each of the blocks of the matrix and using the appropriate maps to determine the rows and columns where contributions need to be added to the matrix block. This has a number of drawbacks with respect to performance. The local assembly kernels require tabulated basis functions for multiple function spaces with large blocks of zeros for the regions where there is no contribution from that function space. Keeping these in on-chip resources such as registers leads to large local working set sizes,

which limit the number of concurrently active threads in particular on many-core architectures, as described in Section 2.2.5.

Ølgaard and Wells [2010] have developed optimisations for the quadrature representation of FFC, one of which is the elimination of zero columns from the basis function tables by introducing an indirection map for non-zero columns. This strategy is not employed by the COFFEE abstract syntax tree optimiser [Luporini et al., 2014] since it destroys the structure of the quadrature loop nest and thereby prevents vectorisation. Such transformations are furthermore incompatible with the concept of PyOP2 iteration spaces detailed in Section 4.2.4.

5.2.4 Splitting Mixed Forms

To mitigate the mentioned performance issues for assembling mixed forms, Firedrake pre-splits forms before passing them to FFC for compilation, obtaining a separate kernel for each of the blocks in (5.11), which is assembled in a separate parallel loop. This keeps working set sizes smaller since kernels only contain basis functions for the pair of subspaces used and the need for padding tabulated basis functions with zeros is avoided.

Obtaining contributions to the individual blocks from (5.11) involves iterating the outer product spanned by the vectors of test and trial subspaces and extracting the contribution to the mixed form a for each combination of subspaces (σ, τ), (u, τ), (σ, v) and (u, v). Splitting a is implemented as a transformation of its UFL expression tree shown in Figure 5.5. A separate pass is made for each combination of test and trial subspaces, where all other subspaces are disabled by setting them to zero.

As described above, the mixed trial and test functions are represented as the three component vectors (σ_0, σ_1, u) and (τ_0, τ_1, v) respectively. Transforming the expression tree requires the retention of this shape such that all indices into the mixed trial and test functions shown in Figure 5.5 remain valid and setting the non-participating components to zero:

$$\begin{pmatrix} (\sigma_0, \sigma_1, 0), (\tau_0, \tau_1, 0) & (0, 0, u), (\tau_0, \tau_1, 0) \\ (\sigma_0, \sigma_1, 0), (0, 0, v) & (0, 0, u), (0, 0, v) \end{pmatrix}. \tag{5.12}$$

When visiting the expression tree, those `argument` nodes representing test and trial functions on a mixed function space are replaced with a vector of

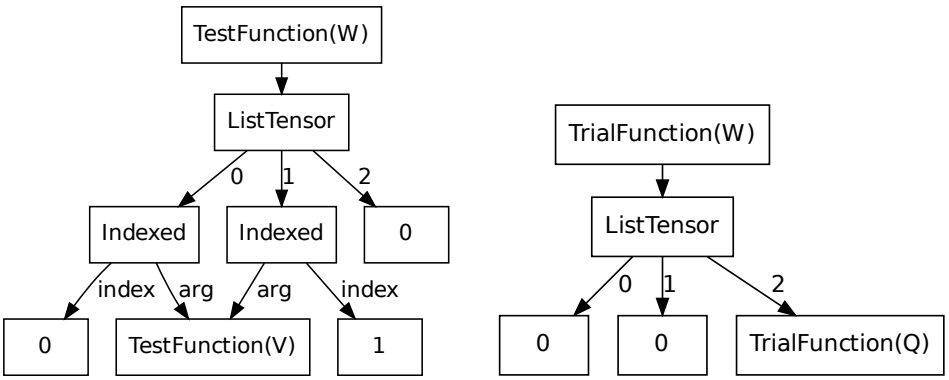

Figure 5.6: UFL expressions replacing the mixed test (left) and trial (right) function

the same value shape as the mixed function space. The argument selected for test and trial subspace in the current pass is inserted in the corresponding positions in the vector and all other components are set to UFL's zero value as shown in (5.12). Note that it is crucial to build the arguments on the individual subspaces and not extract them from the mixed arguments, such that the resulting form is not mixed and has the shape given by the subspaces and does not retain the shape given by the mixed space.

In the following, consider the pass for block $(0, 1)$, where only τ is selected for the test and u for the trial space. The mixed test and trial functions are replaced by the expressions in Figure 5.6. Since the BDM space is vector valued, two components in the vector replacing the mixed test and trial function are non-zero and zero respectively in this pass.

5.2.5 Simplifying Forms

The UFL expression tree for the bilinear form a with mixed arguments replaced by the expressions from Figure 5.6 is displayed in Figure 5.7. This representation seems rather more complicated than the original expression tree from Figure 5.5. However, observe that most nodes in the tree are indexed expressions of the test and trial functions, many of which now point to subspaces that have been replaced by zero. UFL has been equipped with additional simplification rules to eliminate expressions that evaluate to zero when building the modified expression tree.

When selecting a component of a vector valued expression (a `ListTensor` node in the tree) with a fixed index, the indexing operation is eliminated

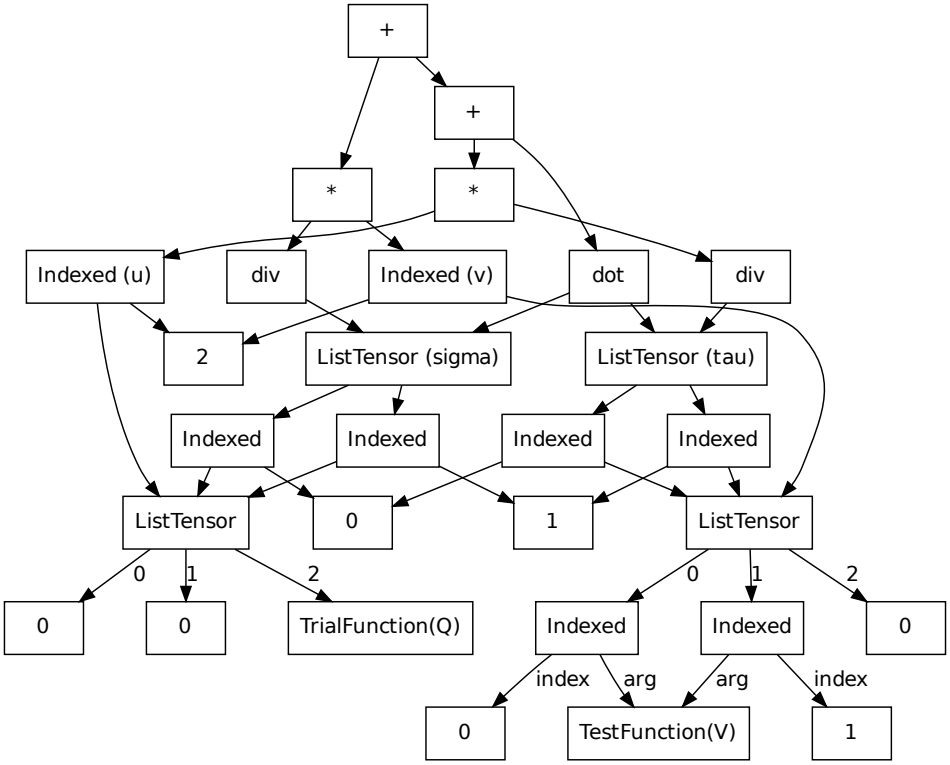

Figure 5.7: UFL expression tree for block $(0, 1)$ of a with mixed test/trial spaces replaced

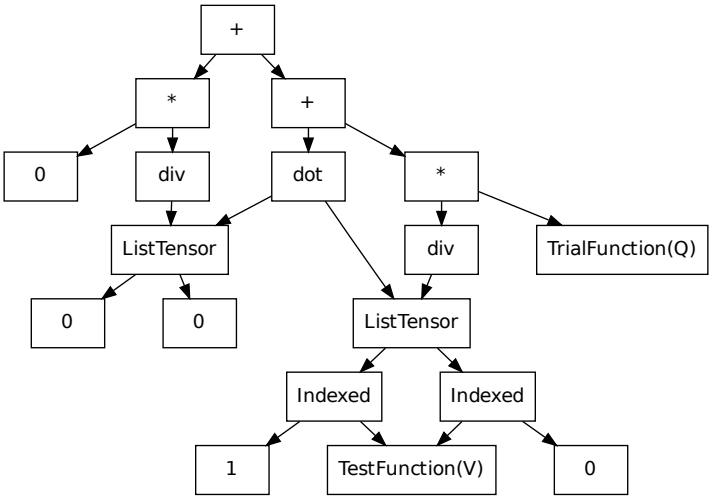

Figure 5.8: UFL expression tree for block $(0, 1)$ of a with indexing simplification applied

and replaced by the selected scalar expression. This means that selecting a subspace which has been set to zero yields zero instead of an indexing operation, which enables further UFL simplifications. Applying this rule transforms the tree from Figure 5.7 into the tree shown in Figure 5.8.

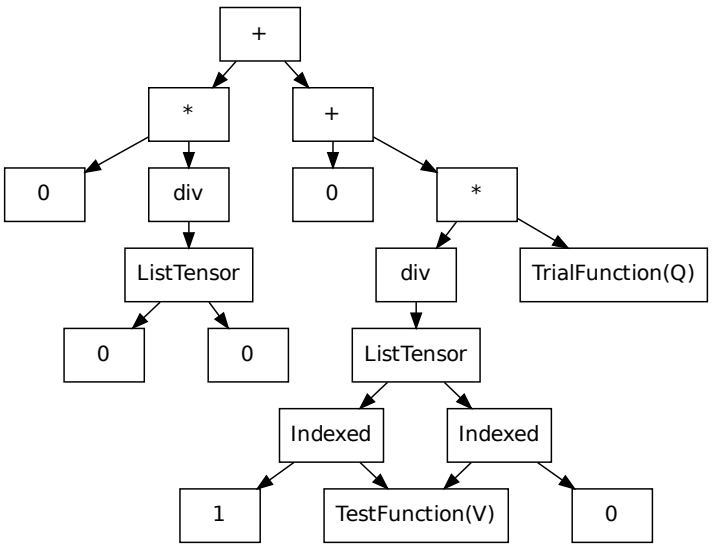

Figure 5.9: UFL expression tree for block $(0,1)$ of a with inner product simplification applied

The other simplification rule that has been added eliminates an inner product of two vector valued expressions (`ListTensor` nodes) and replaces it by zero if for each matching pair of components either component is zero. This rule is applicable to the `dot` product node in Figure 5.8, simplifying the tree further to what is shown in Figure 5.9.

Existing UFL simplification rules for reducing products to zero where one factor is zero and replacing sums with a zero summand by the subtree of the other summand allow simplifying the expression tree from Figure 5.9 further to the final representation for the (u, τ) block in Figure 5.11.

In a similar way, these new simplification rules enable UFL to simplify the forms for the other blocks from (5.12) as shown in Figure 5.10 for the (σ, τ) and Figure 5.12 for the (σ, v) block. Most importantly, (u, v) is reduced to zero such that Firedrake need not pass this form to FFC.

However, there are cases where these rules are not sufficient to detect that a form reduces to zero at UFL level and this only becomes apparent after further preprocessing and analysis by the form compiler. Even

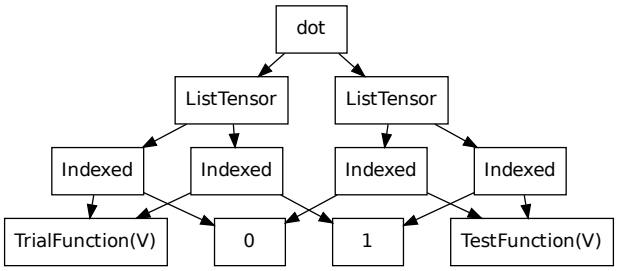

Figure 5.10: UFL expression tree for the (σ, τ) block of the mixed Poisson formulation

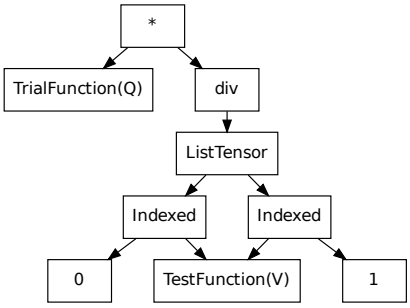

Figure 5.11: UFL expression tree for the (u, τ) block of the mixed Poisson formulation

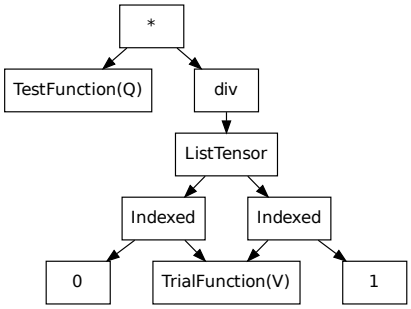

Figure 5.12: UFL expression tree for the (σ, v) block of the mixed Poisson formulation

though these forms do incur the FFC compilation cost, Firedrake detects empty forms and does not launch a parallel loop to assemble them.

5.3 Assembling Expressions

In a finite element context, fields are functions on the mesh, and Firedrake allows them to be manipulated like mathematical functions. From a computational perspective, manipulations of field data are operations on potentially large and, when running in parallel, distributed vectors. In object oriented libraries, complex vector expressions can be formulated succinctly using overloaded operators on the vector types. However, the short circuit evaluation of these operators leads to the creation of potentially large numbers of temporaries for complex expressions. Not only does this consume extra memory but more importantly also requires reading and writing full length vectors many times, losing temporal locality of computations and thereby limiting the achievable performance. A C++ metaprogramming technique known as expression templates [Pflaum, 2001] has been developed to efficiently evaluate such expressions at compile time.

5.3.1 Expression Compiler

Firedrake provides high level operations on functions with overloaded operators, which form an expression tree similar to that of a form, described in Section 5.2, instead of being immediately evaluated. Such an expression is evaluated only when assigned to a result function, at which point an expression compiler is invoked to translate the expression into a kernel suitable for efficient execution by a single PyOP2 parallel loop.

The expression compiler is first of all responsible for verifying that an expression is valid. For an expression to be valid, the left and right hand sides of the final assignment need to be "compatible", which means that one of two conditions is fulfilled. Either all functions in the right hand side expression are defined on the same function space as the left hand side function, or, if the left hand side function is defined on a mixed function space, the right hand side expression only contains functions defined on indexed subspaces of the mixed function spaces. The latter condition is required such that the expression compiler can determine which part of the mixed function the expression needs to be assigned to.

Note that the parallel loop to evaluate an expression is always a direct loop. In the case of the left hand side being defined on a mixed function space, the expression needs to be split into its individual subspaces, since a parallel loop cannot execute over an iteration set that is mixed. Splitting the expression is also required as the right hand side expression may be defined only on a subspace of the left hand side function.

Consider functions `f` and `g` defined on a mixed function space `W` and functions `h0` and `h1` defined on the two subspaces of `W`:

```
V = VectorFunctionSpace(mesh, 'CG', 2)
Q = FunctionSpace(mesh, 'CG', 1)
W = V * Q
f = Function(W)
g = Function(W)
h0 = Function(W[0])
h1 = Function(W[1])
```

For the simplest and most common case, where the right hand side expression is simply a function defined on the same function space as the left hand side function, the expression compiler is bypassed and the operation is directly expressed as a PyOP2 operation on the `Dat` underlying

the function. This is the case for assignment, addition and subtraction of a function or scalar and multiplication or division by a scalar as shown in the following with the PyOP2 operation given in the right column:

```
f.assign(g)   # Assignment                 -> g.dat.copy(f.dat)
f += g        # Addition of a function     -> f.dat += g.dat
f += 1.0      # Addition of a scalar       -> f.dat += 1.0
f -= g        # Subtraction of a function  -> f.dat -= g.dat
f -= 1.0      # Subtraction of a scalar    -> f.dat -= 1.0
f *= 2.0      # Multiplication by a scalar -> f.dat *= 2.0
f /= 2.0      # Division by a scalar       -> f.dat /= 2.0
```

Since `f` is defined on a mixed function space, the operation is transparently applied to both components of the `MixedDat` underlying the function `f`.

5.3.2 Expression Splitting

The expression splitter is conceptually similar to the form splitter described in Section 5.2.4 and also implemented as a UFL tree visitor. Unlike the form splitter however, which splits arguments by making multiple passes over the form and setting components of the mixed function space to zero, expressions are split at the level of coefficients and only a single pass is needed.

Before visiting the tree, the function space of the left hand side function is recorded. When a function node is visited, it is split if defined on the same function space and replaced by the vector of sub functions. If defined on an indexed function space, a vector is returned which contains the sub function for the indexed component and zero for all other components. All the operators are reconstructed by distributing them over the components of this vector. In the case of a zero operand this means the operator is replaced by zero in the case of a product and by the other summand in the case of a sum by applying standard UFL simplification rules as described in 5.2.5.

Consider the following assignment of an expression of `h1`, defined on a subspace of `W`, to `f`, defined on the entire mixed space `W`:

```
f.assign(2*h1)
```

which is represented by the expression tree shown in the left of Figure 5.13. Splitting this expression results in the trees shown in the centre

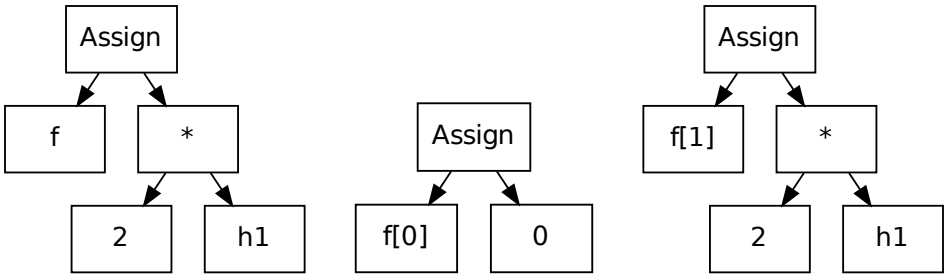

Figure 5.13: Expression (left) split into a first (centre) and second (right) component

and right of Figure 5.13 respectively. The contribution from `h1` is only assigned to the second component of `f`, whereas the first component is set to zero since there is no contribution from the right hand side. The same expression with addition instead of assignment

```
f += 2*h1
```

results in the same expression tree with an `IAdd` instead of an `Assign` as the root note. As a consequence, the split expression for the first component is discarded since it amounts to the addition of zero.

5.3.3 Expression Code Generation and Evaluation

Each of the split expressions, which is a single one if the function space of the left hand side of the original expression was not a mixed function space, is subsequently visited a second time to build the list of arguments for the PyOP2 parallel loop call to evaluate the expression. In this second visit, the expression tree is transformed into a COFFEE abstract syntax tree (AST), from which the expression kernel is initialised. The kernel for the expression tree given at the right of Figure 5.13 is given below:

```
void expression (double* fn_0 , double* fn_1 ) {
  for (int dim = 0; dim < 1; ++dim)
    fn_0[0] = 2 * fn_1[0];
}
```

This kernel is then executed in a direct parallel loop over the set of DOFs of function space `Q`, with the first component of the `MixedDat` underlying `f` used in `WRITE` and the `Dat` underlying `h1` in `READ` access mode.

5.4 Assembling Forms

Solving a linear or non-linear variational problem as described in Sections 5.6.2 and 5.6.1 requires the assembly of either a linear system of equations which can be solved using a linear solver, or the Jacobian and residual form for evaluation by a non-linear solver. In Firedrake, the `assemble` function is the unified interface used to assemble a form into a global tensor. This function is called in a number of different contexts, either explicitly by the user to obtain a pre-assembled system which is solved as described in Section 5.6.4, or implicitly, when solving a variational problem.

Conceptually, the assembly operation maps out the local evaluation of integrals defined by the form over all mesh entities given by the integral's measure and gathers the results into a global tensor. Firedrake formulates this operation in terms of a PyOP2 parallel loop over those mesh entities as described in Section 4.1.3 with a kernel produced by the FEniCS form compiler FFC introduced in Section 3.2.3.

The result of the assembly operation depends on the rank of the form being assembled, that is the number of unknown argument functions. For a bilinear form of rank two with both a test and trial function, a rank two tensor, that is a sparse matrix, is assembled. Similarly, a rank one form with only a test function yields a vector, that is a tensor of rank one, whereas a form of rank zero that contains neither test nor trial function and is commonly called a functional, produces a scalar result.

Assembling a form in Firedrake involves the following steps:

1. Split the form as described in Section 5.2.4 and compile each block with FFC to obtain a list of kernels, one for each integral in each block of the form (Section 5.4.1).

2. Use the form rank to determine the kind of tensor to assemble and initialise the output tensor (Section 5.4.2).

3. Build the form-specific list of arguments for the PyOP2 parallel loop call performing the local assembly computation from the arguments and coefficients of the form (Section 5.4.3).

5.4.1 Assembly Kernels

A UFL form may contain one or more integrals over the cells, the interior or the exterior facets of the mesh. Each of these integrals corresponds to a local assembly kernel performing the numerical quadrature. After splitting the form as described in Section 5.2.4, each resulting non-zero block is compiled by FFC to obtain an abstract syntax tree used to initialise a PyOP2 kernel. As a side effect FFC also preprocesses the form, which gives Firedrake access to a `form_data` object containing various metadata about the form, such as the rank, the arguments, the coefficients and the integrals used in the form.

Since calling FFC is a potentially very costly operation, Firedrake avoids repeatedly compiling the same form by caching the produced kernels in memory and on disk, keyed on the unique signature of the form.

5.4.2 Assembling Matrices, Vectors and Functionals

On a finite element level, a matrix is a linear operator between two function spaces, which are used in a combination of cell and facet integrals and define the sparsity pattern of the matrix. The finite element abstraction can be directly expressed in terms of PyOP2 constructs: As described in Section 4.6, a matrix in PyOP2 terms is a linear operator mapping between two datasets, whose sparsity pattern is built from one or several pairs of maps for the row and column spaces of the matrix respectively.

For a rank two form, Firedrake builds a PyOP2 `Mat` to assemble into, which is defined on a `Sparsity` as described in Section 4.1.2. The row and column spaces of the matrix are defined by test and trial function spaces of the form, which are available as part of the form data. The datasets defining the row and column space of the matrix as well as the map pairs for each integral are obtained from the test and trial function spaces.

A pair of maps according to the integral's domain type is built from the test and trial spaces for each integral in the form and added to the sparsity pattern. For an integral over cells, the map from cells to degrees of freedom is extracted from the function spaces and similarly for integrals over exterior or interior facets. The sparsity is used to initialise a new matrix, whose row and column space are given by the degree of freedom datasets of the test and trial space. The result tensor is a `Matrix` object.

For a linear form, Firedrake creates a new `Function` defined on the test space of the form and returns the underlying `Dat` as the result tensor.

For a functional, Firedrake assembles into a PyOP2 `Global` and returns its scalar data value as the result.

5.4.3 Parallel Loops for Local Assembly Computations

Having initialised the global tensor to assemble into, Firedrake hands off the local assembly computation to PyOP2, launching a parallel loop for each of the kernels compiled from the form. These parallel loop calls, as described in 4.1.3, require the kernel to execute, the iteration set to execute over and access descriptors matching the kernel arguments, defining how data is to be passed to the kernel. The first two arguments to an FFC kernel, which are always present, are the local tensor that is being computed and the coordinates of the vertices of the current cell or facet. Any further arguments are coefficients used in the form.

The iteration set for a local assembly operation is, depending on the domain type, the cells, exterior or interior facets of the mesh respectively, which is extracted from the function space of the test function for bilinear and linear forms. For constant functionals, which do not contain a function, Firedrake attaches the coordinate field as domain data to the integral measure, where the mesh can be extracted from, assuming a single mesh.

An access descriptor for the output tensor contains the access mode and the map, in the case of a linear form, or pair of maps, in the case of a bilinear form, used for the indirect access. The access mode for the output tensor is `INC`, such that PyOP2 deals with write contention from accumulating contributions from any cell sharing a degree of freedom. As explained previously, the map to be used is determined from the domain type of the integral corresponding to the kernel being assembled. The appropriate pair of maps for a bilinear form is obtained from the test and trial function spaces, the map for a linear form from the test function space. Since functionals reduce into a scalar, no indirection and therefore no map is required.

The remaining access descriptors are built from the coordinate function of the mesh and any coefficients present in the form obtained from its form data. All these arguments are read-only and indirectly accessed via

a map from cells or facets to DOFs, obtained from the function space of the coefficient. The algorithm is outlined in Listing 2. As described in Section 5.5.1, matrix assembly is delayed until the point where the final set of boundary conditions is known. Assembly is therefore implemented as a callback function taking the boundary conditions as an argument.

Listing 2 thunk (bcs): Callback function to assemble a matrix

for kernel **in** kernels
    (1) Determine iteration set and maps based on domain type:
    Maps are modified according to the boundary conditions (Section 5.5.1)
    if domain type **is** "cell"
        iteration set = cell set
        maps = cell → node map (bcs) for test / trial function
    elif domain type **is** "exterior facets"
        iteration set = exterior facet set
        maps = exterior facet → node map (bcs) for test / trial function
    elif domain type **is** "interior facets"
        iteration set = interior facet set
        maps = interior facet → node map (bcs) for test / trial function
    (2) Build list of arguments that are always present
    arguments = [kernel, iteration set, tensor (`INC`, maps)]
    arguments $\overset{+}{=}$ coordinates Dat (`READ`, cell → node map)
    (3) Add arguments for form coefficients
    for coefficient **in** form coefficients
        arguments $\overset{+}{=}$ coefficient Dat (`READ`, cell → DOF map)
    call PyOP2 parallel loop (arguments)

5.5 Imposing Dirichlet Boundary Conditions

Essential boundary conditions, also referred to as *Dirichlet* or *strong*, prescribe values of the solution for a certain region or certain points in the domain and become constraints on the function space, which lead to modifications of the system being solved.

5.5.1 Assembling Matrices with Boundary Conditions

Firedrake always imposes strong boundary conditions in a way that preserves the symmetry of the operators, which is efficiently implemented

using the PyOP2 and PETSc abstractions as described in the following.

Symmetry is preserved by zeroing rows and columns of the matrix corresponding to boundary nodes. This operation would be very costly to apply to an already assembled matrix in CSR format, as it would require searching each row of the non-zero structure for the column positions to zero. Entries to be set to zero are therefore already dropped during assembly and never added to the global matrix in the first place. For this to happen, it is sufficient to modify the maps used in the PyOP2 access descriptor of the matrix, replacing entries corresponding to boundary nodes by -1, which causes any contribution in the same matrix row and column to be ignored by PETSc. Subsequently, diagonal entries for rows corresponding to boundary nodes are set to 1.

This implementation illustrates the power of the composition of abstractions: a Firedrake operation efficiently expressed as a combination of PyOP2 and PETSc operations, where neither of the lower layers has, or needs to have, a concept of boundary conditions. To PyOP2 it is "just" a different map and to PETSc it "just" a value to be ignored.

Firedrake however supports a number of ways in which strong boundary conditions can be prescribed: when specifying a linear or non-linear variational problem, in the call to `assemble` or `solve`, or by explicitly applying the boundary condition to a function. When pre-assembling a system and only specifying the boundary conditions in the call to `solve`, the boundary conditions are not available at the time `assemble` is called:

```
A = assemble(a)
b = assemble(L)
solve(A, x, b, bcs=bcs)
```

In the case where `assemble` is called with boundary conditions, different boundary conditions explicitly applied at a later point or specified in the `solve` call take precedence. A naive implementation of the strategy described above may therefore lead to assembling a matrix with boundary conditions which is never used to solve a system and require unnecessary and costly reassembly. Firedrake therefore delays the actual assembly until the point where the final set of boundary conditions is known.

A call to `assemble` returns an *unassembled* `Matrix` object and no actual assembly takes place. The sparsity pattern is built as described in Section 5.4.2 and the underlying PyOP2 `Mat` object is created. A callback function

Figure 5.14: Stages in assembling a Firedrake `Matrix`

termed the assembly *thunk*[2] is set on the returned `Matrix` object, which is called during the solve with the final list of boundary conditions to perform the actual assembly and obtain an assembled matrix. The stages of assembling a Firedrake `Matrix` are illustrated in Figure 5.14.

The operation of the callback is detailed in Section 5.4.3 and outlined in Listing 2: for each kernel, a PyOP2 parallel loop is called to assembly into the matrix, after its list of arguments has been created. However, the pair of maps used to initialise the parallel loop argument for the output matrix is modified according to the prescribed boundary conditions by setting map entries corresponding to boundary nodes to -1. When all parallel loops have been processed, the entries on the diagonal of rows corresponding to boundary nodes are set to 1.

5.5.2 Boundary Conditions for Variational Problems

When solving a variational problem with strong boundary conditions, the first step is modifying the provided initial guess `u` to satisfy the boundary conditions at the boundary nodes before invoking the non-linear solve. As described in Section 5.6.3, the PETSc SNES solver requires callbacks for residual and Jacobian evaluation, which are implemented in Firedrake in terms of assembling the residual form `F` and Jacobian form `J` respectively.

The Jacobian form `J` is assembled by calling the matrix assembly thunk with the boundary conditions, causing boundary condition node indices to be replaced by negative values in the indirection maps, which instructs PETSc to drop the corresponding entries. After assembly has completed, diagonal entries of `J` corresponding to boundary nodes are set to 1.

The residual form `F` is assembled without taking any boundary conditions into account. Boundary condition nodes in the assembled residual therefore contain incorrect values, which are set to zero after assembly has completed, whereas the residual is correct on all other nodes. Note that

[2] A subroutine generated to aid the execution of another routine is often called thunk.

the same strategy of dropping boundary contributions used above could be applied, but is not necessary for an efficient implementation, since setting vector entries is comparatively inexpensive.

5.5.3 Boundary Conditions for Linear Systems

Linear systems, in which the matrix is pre-assembled, are solved with boundary conditions using the implementation described in Section 5.5.1.

When `assemble` is called on the bilinear form `a`, an unassembled `Matrix` is returned and no actual assembly takes place. The `Matrix` object defines a callback method, the *assembly thunk*, which is called with the final set of boundary conditions. At the point where `solve` is called, Firedrake applies boundary conditions supplied to the `solve` call with highest priority. If none are given, any boundary conditions applied when `assemble` was called on A or subsequently added with `apply` are used.

The assembled matrix is then stored in the `Matrix` object so that reassembly is avoided if the matrix is used in another `solve` call with the same boundary conditions.

The right-hand side vector is computed by subtracting the assembled action $\mathcal{A}$ of the bilinear form a on a vector u_{bc} which has the boundary conditions applied at the boundary nodes and is zero everywhere else

$$r = b - \mathrm{assemble}(\mathcal{A}(a, u_{\mathrm{bc}})) \tag{5.13}$$

and subsequently applying the boundary conditions to r.

5.6 Solving PDEs

Variational problems are commonly expressed in the canonical linear and bilinear forms presented in Section 2.1.1. Firedrake's `solve` function provides a unified interface for solving both linear and non-linear variational problems as well as linear systems where the arguments are already assembled matrices and vectors, rather than UFL forms.

This unified interface continues into the implementation, where linear and non-linear variational forms are solved using the same code path. Linear problems are transformed into residual form and solved using a non-linear solver, which always converges in a single non-linear iteration.

5.6.1 Solving Non-linear Variational Problems

Recall from Section 2.1.1 the general non-linear variational problem expressed in the semilinear residual form

$$F(u;v) = 0 \quad \forall v \in V \tag{5.14}$$

with the unknown function u as a possibly non-linear and the test function v as a linear argument.

A method commonly used in Firedrake to solve non-linear systems is Newton's method, where the solution is successively approximated by

$$u_{k+1} = u_k - J(u_k)^{-1}F(u_k)\ k = 0, 1, \ldots \tag{5.15}$$

starting with an initial guess u_0 of the solution. The *Jacobian* of the residual $J(u_k) = \frac{\partial F(u_k)}{\partial u_k}$ is is required to be non-singular at each iteration.

The Newton iteration (5.15) is implemented in two steps:

1. approximately solve $J(u_k)\Delta u_k = -F(u_k)$, and
2. update $u_{k+1} = u_k + \Delta u_k$.

A Jacobian can be supplied explicitly by the user if know, although this is not required. If not supplied, Firedrake invokes UFL to compute the Jacobian by automatic differentiation of the residual form F with respect to the solution variable u.

5.6.2 Transforming Linear Variational Problems

A weak variational problem is expressed in the canonical linear form as

$$a(u, v) = L(v) \quad \forall v \in V \tag{5.16}$$

with a bilinear part a, which is linear in both the test and trial functions v and u, and a linear part L, which is linear in the test function v.

This problem is transformed into residual form by taking the action $\mathcal{A}(a, u)$ of the bilinear form a onto the unknown function u and subtracting the linear form L:

$$F(u, v) = \mathcal{A}(a, u)(v) - L(v) = 0 \quad \forall v \in V. \tag{5.17}$$

In this case, the Jacobian is known to be the bilinear form a and hence

there is no need to compute it using automatic differentiation. When solving a linear variational problem, Firedrake therefore computes the residual form according to (5.17) and passes it to the non-linear solver along with the solution function u and the bilinear form a as the Jacobian. A single non-linear iteration is required to solve this system and PETSc is instructed to skip the unnecessary convergence check afterwards.

Observe how for the residual form (5.17), the first Newton step

$$u_1 = u_0 - J(u_0)^{-1}F(u_0) \tag{5.18}$$

with an initial guess $u_0 = 0$, solution $u = u_1$ and identifying J as the linear operator A assembled from the bilinear form a, is equivalent to solving the linear system

$$Au = b, \tag{5.19}$$

where b is the assembled residual form equivalent to the right-hand side of (5.16), since the action of a on a zero vector vanishes:

$$-F(u_0) = -(\mathcal{A}(a, u_0)(v) - L(v)) = L(v). \tag{5.20}$$

5.6.3 Non-linear Solvers

To solve non-linear systems, Firedrake uses PETSc SNES [Balay et al., 2013, Chapter 5], a uniform interface to Newton-like and quasi-Newton solution schemes. All these schemes are implemented using evaluations of the residual and its derivative, the Jacobian, at given points. SNES therefore requires two callbacks to be provided, one for evaluation of the residual and one for evaluation of the Jacobian.

Firedrake implements the residual callback by assembling the residual form of the non-linear variational problem. Similarly, the Jacobian callback is implemented in terms of assembling the Jacobian form, which was either supplied or computed by automatic differentiation of the residual form as detailed above.

The linear system for each non-linear iteration is solved using the PETSc KSP family of Krylov subspace method solvers [Balay et al., 2013, Chapter 4]. Firedrake uses PETSc's options database directly to give users full control when specifying solver options. By default, the solve call will use GMRES with an incomplete LU factorisation as the preconditioner.

5.6.4 Solving Pre-assembled Linear Systems

When solving a time-dependent linear system, often the bilinear form a does not change between time steps, whereas the linear form L does. It is therefore desirable to pre-assemble the bilinear forms in such systems as described in Section 5.4 to reuse the assembled operator in successive linear solves and save the potentially costly and unnecessary reassembly in the time stepping loop. The linear pre-assembled system has the form

$$A\vec{x} = \vec{b} \tag{5.21}$$

where A and $\vec{b}$ are the assembled bilinear and linear forms and $\vec{x}$ is the unknown vector to solve for. In Firedrake, the `Matrix A` and `Function b` are obtained by calling `assemble` on the UFL bilinear and linear forms a and L defining the variational problem:

```
A = assemble(a)
b = assemble(L)
```

The same unified `solve` interface as for variational problems is used with different arguments, passing in the assembled `Matrix A`, the solution `Function x` and the assembled right-hand side `Function b`:

```
solve(A, x, b)
```

When called in this form, `solve` directly calls the PyOP2 linear solver interface detailed in Section 4.6.5 instead of using the non-linear solver.

It is worth highlighting that Firedrake implements a caching mechanism for assembled operators, avoiding the reassembly of bilinear forms when using the variational solver interface in a time stepping loop. To avoid cached operators filling up the entire memory, Firedrake monitors available system memory and evicts cached operators base on their "value" until memory usage is below a set threshold. Presently, this value is the assembly time, corresponding to a first-in first-out (FIFO) eviction strategy. Explicit pre-assembly is therefore unnecessary for most practical cases, since it provides no performance advantage over the variational interface.

5.6.5 Preconditioning Mixed Finite Element Systems

To solve mixed problems with multiple coupled variables as described in Section 5.2 efficiently, it is important to exploit the block structure of the system in the preconditioner. PETSc provides an interface to composing "physics-based" preconditioners for mixed systems using its fieldsplit technology [Balay et al., 2013, Chapter 4.5]. As described in Section 4.8.2, PyOP2 stores the block matrices arising in such problems in nested form using the PETSc MATNEST format, which provides efficient access to individual sub-matrices without having to make expensive copies. Firedrake can therefore efficiently employ the fieldsplit method to build preconditioners from Schur complements when assembling linear systems as described in this section.

Recall the mixed formulation of the Poisson equation from Section 5.2.1:

$$\langle \sigma, \tau \rangle - \langle \mathrm{div}\tau, u \rangle + \langle \mathrm{div}\sigma, v \rangle = \langle f, v \rangle \quad \forall (\tau, v) \in \Sigma \times V \tag{5.22}$$

As described in Section 5.2.3, the monolithic left-hand side is conceptually a 2×2 block matrix

$$\begin{pmatrix} \langle \sigma, \tau \rangle & \langle \mathrm{div}\tau, u \rangle \\ \langle \mathrm{div}\sigma, v \rangle & 0 \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \tag{5.23}$$

which can be factored into lower triangular, diagonal and upper triangular parts:

$$LDU = \begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} I & A^{-1}B \\ 0 & I \end{pmatrix}. \tag{5.24}$$

This is the *Schur complement factorisation* of the block system with inverse

$$P = \begin{pmatrix} I & -A^{-1}B \\ 0 & I \end{pmatrix} \begin{pmatrix} A^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -CA^{-1} & I \end{pmatrix}. \tag{5.25}$$

where S is the *Schur complement*

$$S = D - CA^{-1}B. \tag{5.26}$$

Firedrake takes care of setting up the fieldsplit blocks in the case where

a mixed system is solved. Using such a factorisation therefore requires no change to the user code other than configuring the `solve` call to use it via the solver parameters:

```
solve(a == L, u,
      solver_parameters={'ksp_type': 'gmres'
                         'pc_type': 'fieldsplit',
                         'pc_fieldsplit_type': 'schur',
                         'pc_fieldsplit_schur_fact_type': 'FULL',
                         'fieldsplit_0_ksp_type': 'cg',
                         'fieldsplit_1_ksp_type': 'cg'})
```

As configured above with Schur complement factorisation type `'FULL'`, PETSc uses an approximation to P to precondition the system, which is applied via block triangular solves with the grouping $L(DU)$. Other available options are `'diag'`, `'lower'` and `'upper'`, which use only the D block, with the sign of S flipped to make the preconditioner positive definite, the L and D blocks and the D and U blocks of (5.24) respectively.

Inverses of A and S are never computed explicitly. Instead, the actions of A^{-1} and S^{-1} are approximated using a Krylov method, which is selected using the `'fieldsplit_0_ksp_type'` and `'fieldsplit_1_ksp_type'` options shown above respectively.

5.7 Comparison with the FEniCS/DOLFIN Tool Chain

Firedrake is deliberately compatible to DOLFIN in its public API as far as possible, but rather different in its implementation. Figure 5.15 shows the Firedrake and FEniCS/DOLFIN tool chains side by side. This section discusses a number of differences worth highlighting.

A key design decision in Firedrake and PyOP2 is the use of Python as the primary language of implementation. Performance critical library code such as processing the mesh or building sparsity patterns is implemented in Cython [Behnel et al., 2011], which is also used to interface to third party libraries, in particular PETSc [Balay et al., 1997] via its petsc4py [Dalcin et al., 2011] interface. For executing kernels over the mesh, PyOP2 generates native code for the target platform. DOLFIN takes the opposite approach, where the core library is implemented in C++ and an interface is exposed to Python via the SWIG [Beazley, 2003] interface generator.

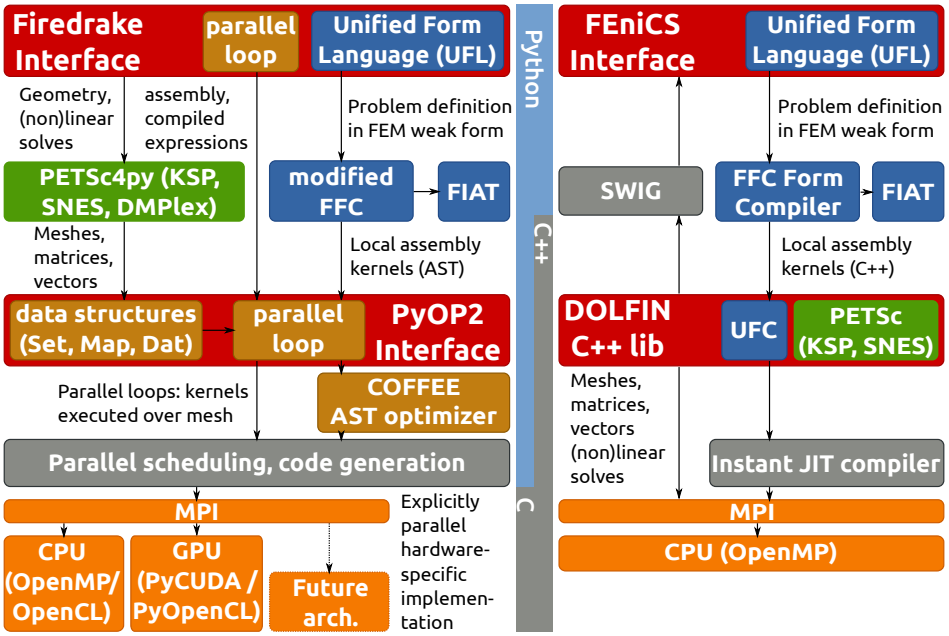

Figure 5.15: Overview of the Firedrake (left) and FEniCS/DOLFIN (right) tool chains

Even though the interface is automatically generated, the interface definitions still need to be maintained by hand. Since the Python API mimics the corresponding C++ API in most cases, some Python features that could make the API more intuitive to use, such as properties, are not used. Firedrake inherits this API design for compatibility reasons. Furthermore, the SWIG layer presents an impenetrable barrier to the PyDOLFIN user, whereas the Firedrake user can inspect or access Firedrake and PyOP2 constructs as Python objects all the way down to the parallel loop level.

Both DOLFIN and Firedrake use UFL as part of their interface and FFC to translate forms into local assembly kernels, however the role of FFC is different. For DOLFIN, FFC generates C++ code strings conforming to the UFC interface specification, which are used unaltered. FFC is therefore responsible for producing optimised code. The modified FFC version used by Firedrake on the other hand produces an unscheduled kernel loop nest in form of an abstract syntax tree (AST). This AST is passed on to the COFFEE AST optimiser described in Section 4.2.2, which can take into account particular characteristics of the PyOP2 backend in its optimisations. Firedrake presently requires custom versions of UFL, FFC and FIAT, however all effort is made to retain compability with the FEniCS mainline such that

any modifications can easily be integrated back into the mainline.

The abstract syntax tree optimised by COFFEE is used to produce a kernel suitable for execution in a PyOP2 parallel loop, which is a generic interface, parametrised by access descriptors and capable of executing any kernel currently used in Firedrake. Adding further types of kernels requires no modification of this interface and it presents a natural way of running computations which are not expressible in UFL, providing an "escape hatch" to break out of the Firedrake abstraction. A common use case is computing the maximum of a continuous and a discontinuous field for every degree of freedom of the continuous field as part of a slope limiter.

DOLFIN instead implements the UFC interface specification, a fixed set of kernels with prescribed interfaces, defined in a C++ header file. Even though UFC purports to be a "black box" interface for assembly, it is designed for an assembler that operates sequentially on a cell-by-cell basis. Every task needs its own interface and adding support for a new type of computation or modifying the signature of an existing operation requires a modification of UFC. Parallel assembly on a many-core architecture would even require a fundamental redesign.

Using PyOP2 as the parallel execution layer for assembly kernels takes a significant amount of complexity out of Firedrake's responsibility, keeping its code base very compact and maintainable. Storage, transfer and communication of data as well as support for multiple backends are thereby abstracted away and handled by PyOP2. Furthermore, Firedrake contains no parallel code since all parallelism is handled by PyOP2 or PETSc.

With this design, a separation of concerns is achieved, where Firedrake is purely a system for reasoning about variational forms, whereas PyOP2 is an execution layer for parallel computations over the mesh, which is agnostic to the higher abstraction layer driving it. A contributor to PyOP2 needs no specific knowledge of the finite element method and how to implement it, while a Firedrake contributor does not need to be expert on parallel computations or programming accelerators. Similarly, a Firedrake user can break out of the abstraction by extracting the underlying PyOP2 data structures from Firedrake objects and making direct calls to PyOP2.

5.8 Conclusions

In this chapter, it has been demonstrated how Firedrake abstracts the mathematical operations and concepts involved in solving partial differential equations with the finite element method to form a modular, extensible and maintainble framework for scientific computations capable of solving a diverse range of problems. Firedrake composes a variety of building blocks from different scientic communities and, where appropriate, established solutions are used in favour of custom implementations. These components and their responsibilities are listed below:

The Unified Form Language (UFL) is used to describe variational forms and their discretisations.

The FEniCS form compiler (FFC) translates variational forms into numerical kernels describing the local assembly operations.

The FInite element Automatic Tabulator (FIAT) is called by FFC to tabulate finite element basis functions and their derivatives.

PETSc provides linear and non-linear solvers, preconditioners and distributed data structures for matrices, vectors and unstructured meshes.

evtk is used for writing out fields to files in the popular VTK format.

PyOP2 is the parallel execution layer for finite element assembly kernels on different hardware platforms, and abstracts both the mesh topology and the data storage, layout and communication for fields and matrices.

This design keeps responsibilities clearly separated and the Firedrake code base very compact and maintainable. Firedrake keeps operations closed over their abstractions wherever possible, and when not, the number of code paths using a lower level abstraction are minimised. An example of this practice is the solver interface, which always uses a non-linear solver, automatically transforming linear problems into residual form.

Chapter 6

Experimental Evaluation

Firedrake is a tool chain capable of solving a wide range of finite element problems, which is demonstrated in this chapter through experiments chosen to cover different characteristics of the Firedrake implementation. These include assembling and solving a stationary Poisson problem, the non-linear time-dependent Cahn-Hilliard equation and the linear wave equation using an explicit time stepping scheme. Implementation aspects investigated are the assembly of left- and right-hand sides for regular and mixed forms, solving linear and non-linear systems as well as evaluating expressions. All benchmarks represent real-world applications used in fluid dynamics to model diffusion, phase separation of binary fluids and wave propagation.

Source code for all benchmarks and the scripts used to drive them are available as part of the firedrake-bench repository hosted on GitHub[1].

6.1 Experimental Setup

Computational experiments were conducted on the UK national supercomputer ARCHER, a Cray XC30 architecture [Andersson, 2014] with an Aries interconnect in Dragonfly topology. Compute nodes contain two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) series processors linked via a Quick Path Interconnect (QPI) and 64GB of 1833MHz DDR3 memory accessed via 8 channels and shared between the processors in two 32GB NUMA regions. Each node is connected to the Aries router via a PCI-e 3.0 link.

[1] https://github.com/firedrakeproject/firedrake-bench

Firedrake and PETSc were compiled with version 4.8.2 of the GNU Compilers and Cray MPICH2 6.3.1 with the asynchronous progress feature enabled was used for parallel runs. Generated code was compiled with the `-O3 -mavx` flags. The software revisions used were Firedrake revision c8ed154 from September 25 2014, PyOP2 revision f67fd39 from September 24 2014 with PETSc revision 42857b6 from August 21 2014 and DOLFIN revision 30bbd31 from August 22 2014 with PETSc revision d7ebadd from August 13 2014.

Generated code is compiled with `-O3 -fno-tree-vectorize` in the Firedrake and `-O3 -ffast-math -march=native` in the DOLFIN case.

Unless otherwise noted, DOLFIN is configured to use quadrature representation with full FFC optimisations and compiler optimisations enabled and Firedrake makes use of COFFEE's loop-invariant code motion, alignment and padding optimisations described in Luporini et al. [2014] using quadrature representation. Meshes are reordered using PETSc's implementation of reverse Cuthill-McKee in the Firedrake case and DOLFIN's implementation respectively.

Benchmark runs were executed with exclusive access to compute nodes and process pinning was used. All measurements were taken preceded by a dry run of the same problem to pre-populate the caches for kernels and generated code to ensure compilation times do not distort measurements. Reported timings are the minimum of three consecutive runs.

6.2 Poisson

Poisson's equation is a simple elliptic partial differential equation. A primal Poisson problem for a domain $\Omega \in \mathbb{R}^n$ with boundary $\partial\Omega = \Gamma_D \cup \Gamma_N$ is defined as:

$$-\nabla^2 u = f \quad \text{in } \Omega, \tag{6.1}$$

$$u = 0 \quad \text{on } \Gamma_D, \tag{6.2}$$

$$\nabla u \cdot n = 0 \quad \text{on } \Gamma_N. \tag{6.3}$$

The weak formulation reads: find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x \quad \forall v \in V \tag{6.4}$$

where V is a suitable function space satisfying the Dirichlet boundary condition $u = 0$ on Γ_D.

This benchmark demonstrates assembly of a bilinear and linear form into a sparse matrix and vector, and solving a linear system with a preconditioned Krylov method.

6.2.1 Problem Setup

The source term f is defined as

$$f = 48\pi^2 \cos(4\pi x) \sin(4\pi y) \cos(4\pi z) \tag{6.5}$$

so that the analytical solution is know to be

$$u = \cos(4\pi x) \sin(4\pi y) \cos(4\pi z). \tag{6.6}$$

Since the operator is symmetric positive definite, the problem is solved using a CG solver with the HYPRE Boomeramg algebraic multigrid preconditioner [Falgout et al., 2006] on a unit cube mesh of varying resolution and for varying polynomial degrees. Listing 6.1 shows the Firedrake code for this problem.

6.2.2 Results

Strong scaling runtimes for matrix and right-hand side assembly and linear solve comparing DOLFIN and Firedrake on up to 1536 cores are shown in Figure 6.1 for problems of approximately 0.5M DOFs for first order, 4M DOFs for second order and 14M DOFs for third order. Solve time clearly dominates in all cases, in particular for higher order and in the strong scaling limit, where the scaling flattens out at around 5k DOFs per core. The differences in solving times between Firedrake and DOLFIN are caused by different global DOF numberings due to different mesh reordering implementations, which affect the effectiveness of the AMG preconditioner.

Firedrake is faster at assembling left- and right-hand sides in all cases, demonstrating the efficiency of low overhead assembly kernel execution through PyOP2. Matrix assembly is notably faster for the P3 case and scales considerably further in the strong scaling limit, flattening out only at about 1k DOFs per core, compared to approximately 5k for DOLFIN.

Listing 6.1: Firedrake code for the Poisson equation. `mesh` and `degree` are assumed to have been defined previously. UFL functions and operations are defined in orange, while other FEniCS language constructs are given in blue.

```
V = FunctionSpace(mesh, "Lagrange", degree)

bc = DirichletBC(V, 0.0, [3, 4])  # Boundary condition for y=0,y=1

u = TrialFunction(V)
v = TestFunction(V)
f = Function(V).interpolate(Expression(
    "48*pi*pi*cos(4*pi*x[0])*sin(4*pi*x[1])*cos(4*pi*x[2])"))
a = inner(grad(u), grad(v))*dx
L = f*v*dx

u = Function(V)
A = assemble(a, bcs=bc)
b = assemble(L)
bc.apply(b)
params = {'ksp_type': 'cg',
          'pc_type': 'hypre',
          'pc_hypre_type': 'boomeramg',
          'pc_hypre_boomeramg_strong_threshold': 0.75,
          'pc_hypre_boomeramg_agg_nl': 2,
          'ksp_rtol': 1e-6,
          'ksp_atol': 1e-15}
solve(A, u, b, solver_parameters=params)
```

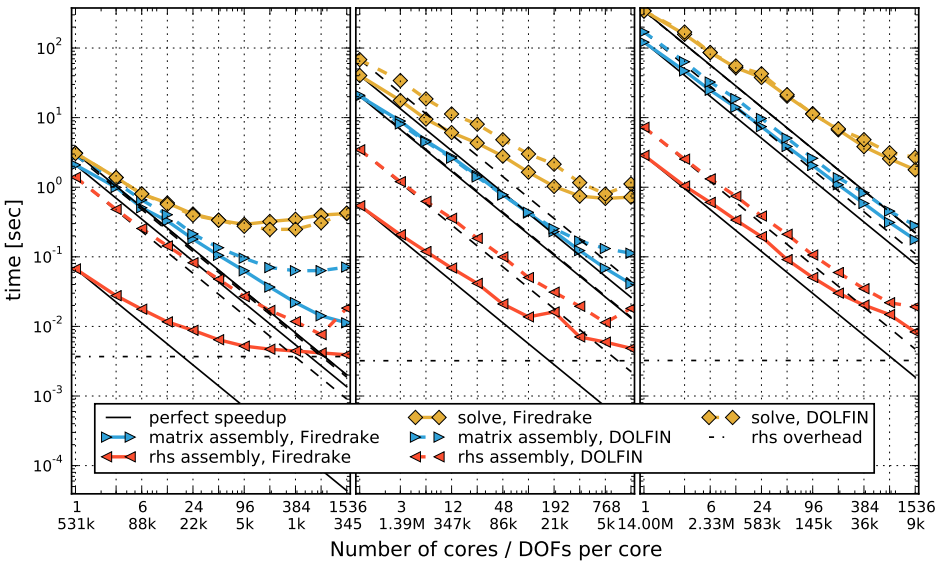

Figure 6.1: Poisson strong scaling on 1-1536 cores for degree one (left), two (center) and three (right) basis functions. Perfect speedup is indicated with respect to a single core.

Right-hand side assembly is considerably faster for Firedrake in all cases, with more than an order of magnitude difference for the P1 sequential base line case. Due to this faster sequential base line, the Firedrake right-hand side assembly is affected by non-parallelisable overheads in the strong scaling limit sooner than DOLFIN. The Firedrake overhead is indicated in Figure 6.1 and in particular the scaling curve for P1 shows that this overhead causes the scaling to flatten out from about 10k DOFs per core. The time spent on right-hand side assembly however is negligible such that the overall run time for Firedrake is not greatly affected.

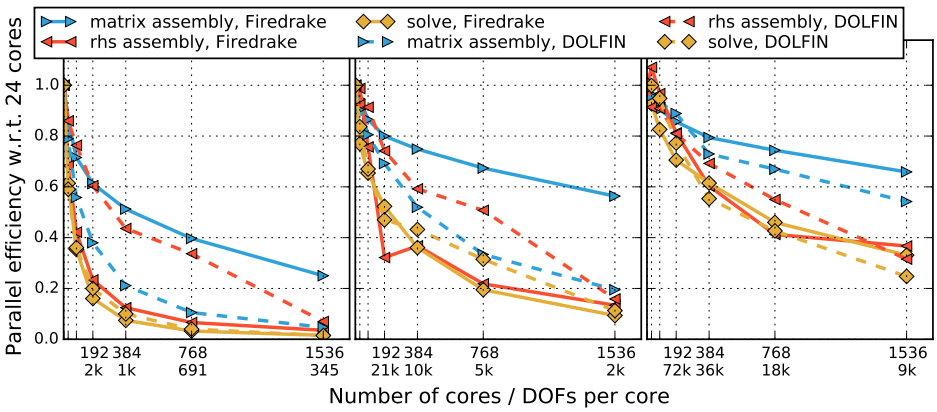

Figure 6.2: Poisson strong scaling efficiency with respect to a full node (24 cores) on up to 1536 cores for degree one (left), two (center) and three (right) basis functions. Firedrake matrix assembly shows the highest efficiency across the board, whereas the solver drops off very quickly. Firedrake right-hand side assembly achieves considerably lower efficiencies compared to DOLFIN due to the faster baseline performance.

Parallel efficiency for the strong scaling results with respect to a full node (24 cores) is shown in Figure 6.2. Solver efficiency is similar for both Firedrake and DOLFIN, dropping to below 40% on 10k, 20% for 2k and 10% for 1k DOFs per core. Left-hand side assembly is significantly more efficient in Firedrake, in particular for P1 and P2, where efficiencies of over 25%, 55% and 65% are maintained for P1, P2 and P3 with 345, 2k and 9k DOFs per core respectively. Efficiency of right-hand side assembly drops quicker for Firedrake due to the better baseline performance, reaching a similar level as DOLFIN of approximately 5%, 15% and 35% for P1, P2 and P3 with 345, 2k and 9k DOFs per core at the highest core count.

Weak scaling run times and efficiencies for P1 basis functions are shown

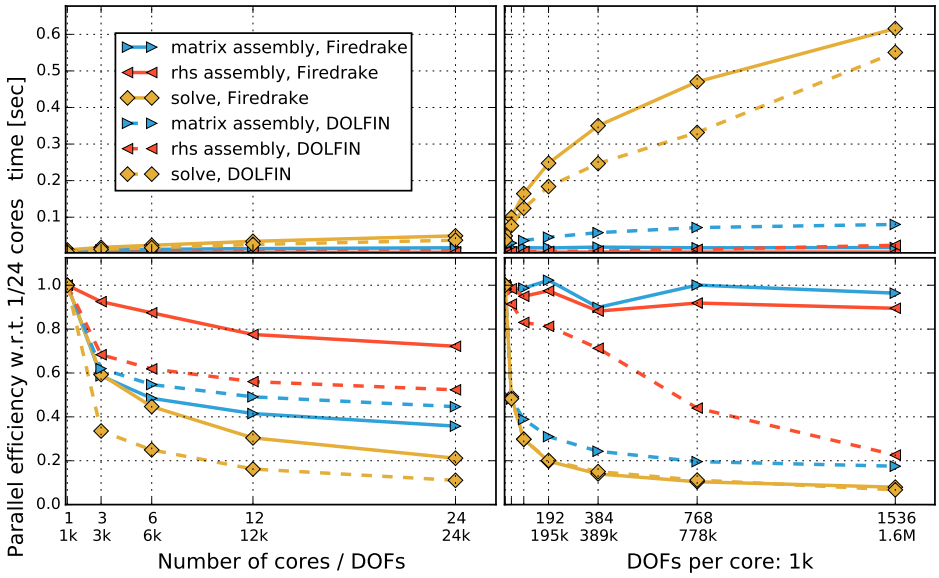

Figure 6.3: Poisson for P1 basis functions weak scaling intra node on 1-24 cores (left) and inter node on 24-1536 cores with a base size of 1k DOFs per core. The solver scales poorly as expected given the low number of DOFs per core. Firedrake achieves almost perfect weak scaling for assembly beyond one node, whereas DOLFIN drops off significantly.

in Figure 6.3 separately for the intra node case for up to 24 cores and the inter node case for 24 to 1536 cores. Within a node, processes share resources, in particular memory bandwidth, which limits achievable performance for these bandwidth bound computations. Scaling beyond a node, resources per core remain constant, and the limiting factor for scalability is network communication latency. The base size was chosen deliberately small with only 1k DOFs per core. Within a node, efficiency drops off significantly from one to three and three to six cores due to resource contention. The solver drops most significantly, to 20% in the Firedrake and 10% in the DOLFIN case, whereas right-hand side assembly achieves above 70% and 50% respectively. DOLFIN maintains a better efficiency for left-hand side assembly of above 45%, whereas Firedrake drops to about 35%. Beyond one node, the observed performance is significantly different with Firedrake weak scaling almost perfectly for assembly with efficiencies above 90%, whereas DOLFIN drops to around 20% on 1536 cores. The solver scales poorly, with an efficiency dropping to 50% already on two nodes and dropping further to below 10% efficiency on 1536 cores,

which is expected given the low number of DOFs per core. The number of Krylov iterations increases from 7 on 1 core to 11 on 24 and 16 on 1536 cores. Similarly, the AMG preconditioner uses 4 levels of coarsening on 1, 10 on 24 and 16 on 1536 cores.

6.3 Linear Wave Equation

The strong form of the wave equation, a linear second-order PDE, on a domain $\Omega \in \mathbb{R}^n$ with boundary $\partial\Omega = \Gamma_N \cup \Gamma_D$ is defined as:

$$\frac{\partial^2 \phi}{\partial t^2} - \nabla^2 \phi = 0, \tag{6.7}$$

$$\nabla \phi \cdot n = 0 \text{ on } \Gamma_N, \tag{6.8}$$

$$\phi = \frac{1}{10\pi} \cos(10\pi t) \text{ on } \Gamma_D. \tag{6.9}$$

To facilitate an explicit time stepping scheme, an auxiliary quantity p is introduced:

$$\frac{\partial \phi}{\partial t} = -p \tag{6.10}$$

$$\frac{\partial p}{\partial t} + \nabla^2 \phi = 0 \tag{6.11}$$

$$\nabla \phi \cdot n = 0 \text{ on } \Gamma_N \tag{6.12}$$

$$p = \sin(10\pi t) \text{ on } \Gamma_D \tag{6.13}$$

The weak form of (6.11) is formed as: find $p \in V$ such that

$$\int_\Omega \frac{\partial p}{\partial t} v \, \mathrm{d}x = \int_\Omega \nabla \phi \cdot \nabla v \, \mathrm{d}x \quad \forall v \in V \tag{6.14}$$

for a suitable function space V. The absence of spatial derivatives in (6.10) makes the weak form of this equation equivalent to the strong form so it can be solved pointwise.

An explicit symplectic method is used in time, where p and ϕ are offset by a half time step. Time stepping ϕ in (6.10) is a pointwise operation, whereas stepping forward p in (6.14) involves inverting a mass matrix. However, by lumping the mass, this operation can be turned into a pointwise one, in which the inversion of the mass matrix is replaced by a point-

Listing 6.2: Firedrake implementation of the linear wave equation.

```
from firedrake import *
mesh = Mesh("wave_tank.msh")

V = FunctionSpace(mesh, 'Lagrange', 1)
p = Function(V, name="p")
phi = Function(V, name="phi")

u = TrialFunction(V)
v = TestFunction(V)

p_in = Constant(0.0)
bc = DirichletBC(V, p_in, 1)  # Boundary condition for y=0

T = 10.
dt = 0.001
t = 0

while t <= T:
    p_in.assign(sin(2*pi*5*t))
    phi -= dt / 2 * p
    p += assemble(dt*inner(grad(v),grad(phi))*dx) / assemble(v*dx)
    bc.apply(p)
    phi -= dt / 2 * p
    t += dt
```

wise multiplication by the inverse of the lumped mass.

This benchmark demonstrates an explicit method, in which no linear system is solved and therefore no PETSc solver is invoked. The expression compiler is used for the p and ϕ updates and all aspects of the computation are under the control of Firedrake. The implementation of this problem in Firedrake is given in Listing 6.2.

6.3.1 Results

Strong scaling runtimes are shown in Figure 6.4 for up to 384 cores and are limited by the measured non-parallelisable overhead indicated by the horizontal lines in the graph. The ϕ update is a very simple expression executed as a direct loop and follows the projected scaling curve (dashed) based on the sequential run time and the overhead almost perfectly. The p update involves assembling a vector, which is executed as an indirect loop and requires exchanging halo data. Therefore, the measured scaling trails behind the projected scaling due to communication overhead

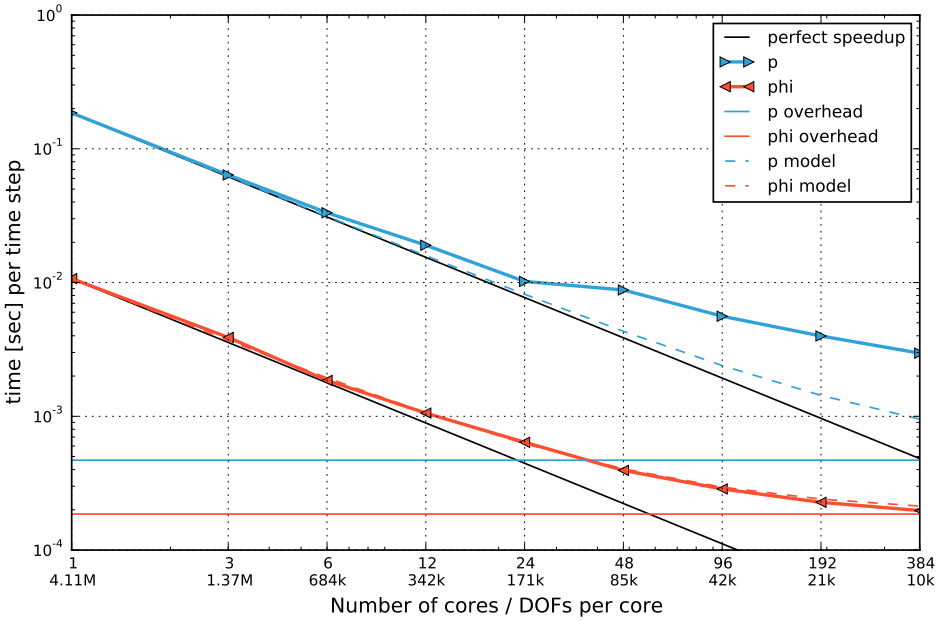

Figure 6.4: Explicit wave strong scaling on up to 384 cores. Perfect speedup is indicated with respect to a single core. Strong scaling is limited by non-parallelisable overheads.

notably starting from 48 cores, which amounts to two full nodes. Communication between the nodes has to pass over the Aries interconnect. Caching of the assembled expressions in the expression compiler keeps the sequential overheads low.

Parallel efficiency for the strong scaling results with respect to a single core is given in Figure 6.5. An efficiency of about 40% and above is maintained down to 85k and 42k DOFs per core for the p and ϕ updates respectively, dropping to about 15% for 10k DOFs per core.

Weak scaling runtimes and efficiencies are shown in Figure 6.6 separately for the intra node case for up to 24 cores and the inter node case for 24 to 384 cores. The ϕ and p update show a significant drop in efficiency to about 50% and 10% respectively from one to three cores due to contention for memory bandwidth and subsequently maintain this level within the node. Across nodes, scaling is almost perfect, with the ϕ update showing superlinear speedups and the p update dropping to 80% efficiency due to increased communication overhead only for 384 cores.

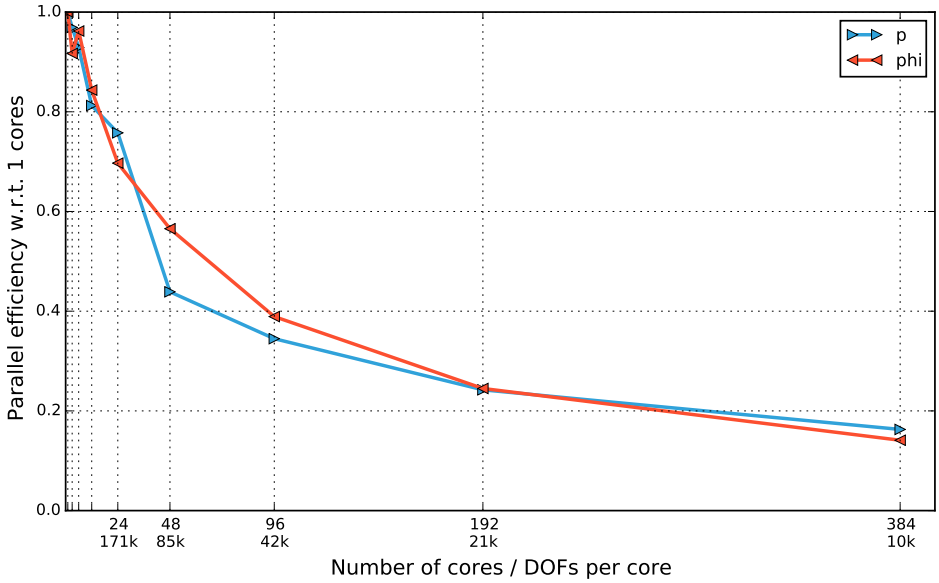

Figure 6.5: Explicit wave strong scaling efficiency with respect to a single node on up to 384 cores.

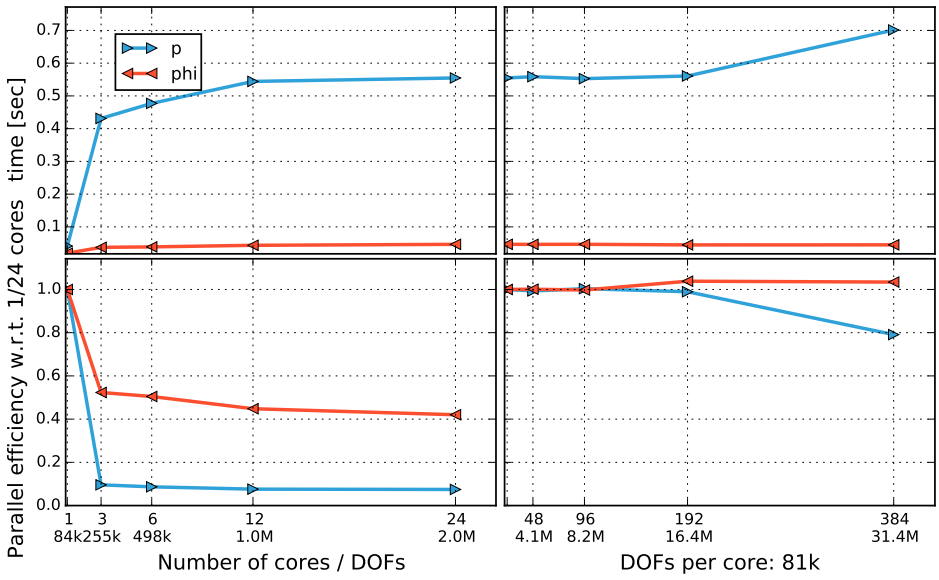

Figure 6.6: Explicit wave weak scaling intra node on 1-24 cores and inter node on 24-384 cores.

6.4 Cahn-Hilliard

The final experiment presented in this section, is the fourth-order parabolic time-dependent non-linear Cahn-Hilliard equation, based on a DOLFIN demo[2], which involves first-order time derivatives, and second- and fourth-order spatial derivatives. It describes the process of phase separation of the two components of a binary fluid and is written as

$$\frac{\partial c}{\partial t} - \nabla \cdot M \left(\nabla \left(\frac{df}{dc} - \lambda \nabla^2 c \right) \right) = 0 \quad in\ \Omega, \tag{6.15}$$

$$M \left(\nabla \left(\frac{df}{dc} - \lambda \nabla^2 c \right) \right) = 0 \quad on\ \partial\Omega, \tag{6.16}$$

$$M\lambda \nabla c \cdot n = 0 \quad on\ \partial\Omega \tag{6.17}$$

with c the unknown fluid concentration, f a non-convex function in c, M the diffusion coefficient and n the outward pointing boundary normal.

Introducing an auxiliary quantity μ, the chemical potential, allows the equation to be rephrased as two coupled second-order equations:

$$\frac{\partial c}{\partial t} - \nabla \cdot M \nabla \mu = 0 \quad in\ \Omega, \tag{6.18}$$

$$\mu - \frac{df}{dc} + \lambda \nabla^2 c = 0 \quad in\ \Omega. \tag{6.19}$$

The time-dependent variational form of the problem with unknown fields c and μ is given as: find $(c, \mu) \in V \times V$ such that

$$\int_\Omega \frac{\partial c}{\partial t} q \, \mathrm{d}x + \int_\Omega M \nabla \mu \cdot \nabla q \, \mathrm{d}x = 0 \quad \forall\ q \in V, \tag{6.20}$$

$$\int_\Omega \mu v \, \mathrm{d}x - \int_\Omega \frac{df}{dc} v \, \mathrm{d}x - \int_\Omega \lambda \nabla c \cdot \nabla v \, \mathrm{d}x = 0 \quad \forall\ v \in V \tag{6.21}$$

for a suitable function space V.

Applying the Crank-Nicolson scheme for time discretisation yields:

$$\int_\Omega \frac{c_{n+1} - c_n}{dt} q \, \mathrm{d}x + \int_\Omega M \nabla \frac{1}{2}(\mu_{n+1} + \mu_n) \cdot \nabla q \, \mathrm{d}x = 0 \quad \forall\ q \in V \tag{6.22}$$

$$\int_\Omega \mu_{n+1} v \, \mathrm{d}x - \int_\Omega \frac{df_{n+1}}{dc} v \, \mathrm{d}x - \int_\Omega \lambda \nabla c_{n+1} \cdot \nabla v \, \mathrm{d}x = 0 \quad \forall\ v \in V \tag{6.23}$$

[2] http://fenicsproject.org/documentation/dolfin/1.4.0/python/demo/documented/cahn-hilliard/python/documentation.html

Listing 6.3: A custom Kernel setting the initial condition for the Cahn-Hilliard example.

```
# Expression setting the initial condition
init_code = "A[0] = 0.63 + 0.02*(0.5 -
    (double)random()/RAND_MAX);"
# Setup code setting the random seed (executed once)
user_code = """int __rank;
MPI_Comm_rank(MPI_COMM_WORLD, &__rank);
srandom(2 + __rank);"""
par_loop(init_code, direct, {'A': (u[0], WRITE)},
         headers=["#include <stdlib.h>"], user_code=user_code)
```

6.4.1 Problem Setup

The problem is solved on the unit square with $f = 100c^2(1-c^2)$, $\lambda = 0.01$, $M = 1$ and $dt = 5 \cdot 10^{-6}$. The function space V is the space of first order Lagrange basis functions.

Firedrake allows the initial condition to be set by defining a custom `Kernel` and executing a parallel loop, in which the expression may be written as a C string. The custom `Kernel` used to set the initial condition in this case is shown as Listing 6.3.

To solve the mixed system, a GMRES solver with a fieldsplit preconditioner using a lower Schur complement factorisation as described in Section 5.6.5 is employed. When solving a mixed system with a 2×2 block matrix with blocks A, B, C, D the Schur complement S is given by

$$S = D - CA^{-1}B. \tag{6.24}$$

and the lower factorisation is an approximation to

$$\begin{pmatrix} A & 0 \\ C & S \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -CA^{-1} & I \end{pmatrix}. \tag{6.25}$$

where A^{-1} and S^{-1} are never explicitly formed.

An approximation to A^{-1} is computed using a single V-cycle of the HYPRE Boomeramg algebraic multigrid preconditioner. The inverse Schur complement, S^{-1}, is approximated by

$$S^{-1} \approx \hat{S}^{-1} = H^{-1}MH^{-1}, \tag{6.26}$$

using a custom PETSc `mat` preconditioner[3], where H and M are defined as

$$H = \sqrt{a}\langle u, v\rangle + \sqrt{c}\langle \nabla u, \nabla v\rangle \quad \forall v \in V \times V \tag{6.27}$$

$$M = \langle u, v\rangle \quad \forall v \in V \times V \tag{6.28}$$

with $a = 1$ and $b = \frac{\mathrm{d}t * \lambda}{1+100\mathrm{d}t}$ [Bosch et al., 2014].

6.4.2 Results

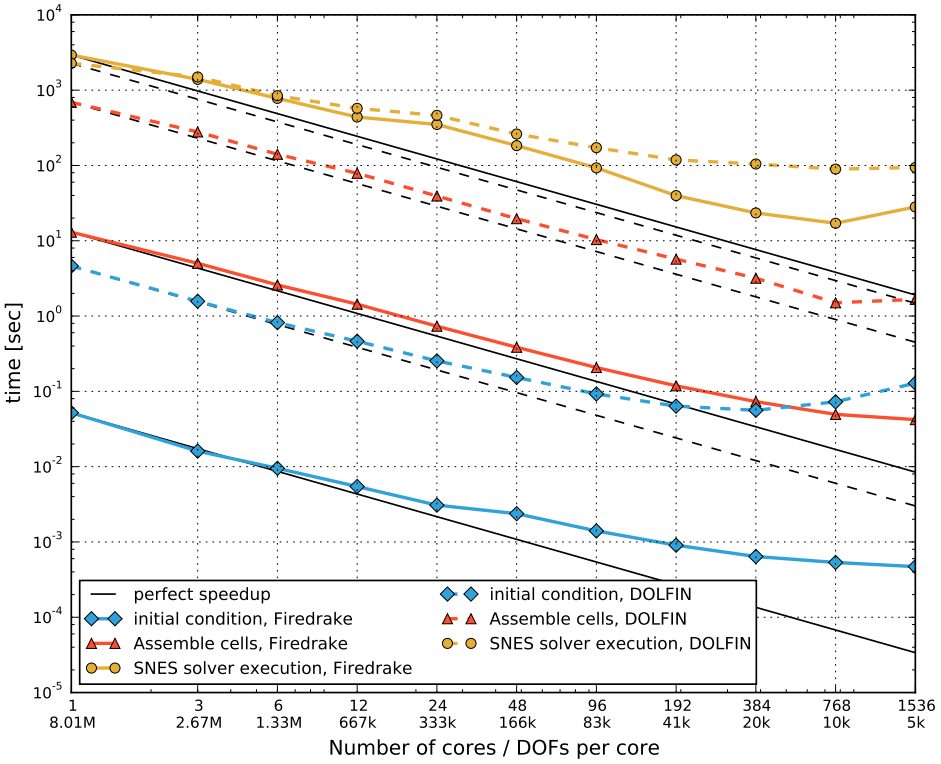

Figure 6.7: Cahn-Hilliard strong scaling for a problem with 8M DOFs for ten time steps on up to 1536 cores. Perfect speedup is indicated with respect to a single core.

Strong scaling runtimes for up to 1536 cores comparing Firedrake and DOLFIN for solving the nonlinear system, assembling the residual and Jacobian forms as well as evaluating the initial condition on an 8M DOF mesh for ten time steps are shown in Figure 6.7. Both Firedrake and DOLFIN achieve close to linear scaling for assembly down to 10k DOFs

[3]The preconditioner implementation is based on https://bitbucket.org/dolfin-adjoint/da-applications/src/520230b/ohta_kawasaki/

per core. Firedrake however is consistently faster between one and two orders of magnitude, demonstrating the efficiency of assembling mixed spaces using the form splitting approach described in Section 5.2.4. Furthermore, the parallel loop objects for residual and Jacobian evaluation are cached on their respective forms, allowing subsequent loops to be called immediately. The loops themselves execute in efficient native code through PyOP2, where the kernels are inlined.

Similar scaling behavior is observed for evaluating the initial condition with Firedrake again faster by about two orders of magnitude, demonstrating the efficiency of expression evaluation using a PyOP2 kernel for the initial condition as opposed to a C++ virtual function call which is required for DOLFIN. Scaling however flattens out in both cases from about 40k DOFs per core due to non-parallelisable overheads. Solver scaling is initially equivalent, with Firedrake gaining a significant advantage starting from about 80k DOFs per core. This is due to the use of a PETSc MATNEST [Balay et al., 2013, Section 4.5] as described in Section 4.8.2, which is more efficient when using a fieldsplit preconditioner since it does not require expensive copies for extracting the sub blocks of the matrix.

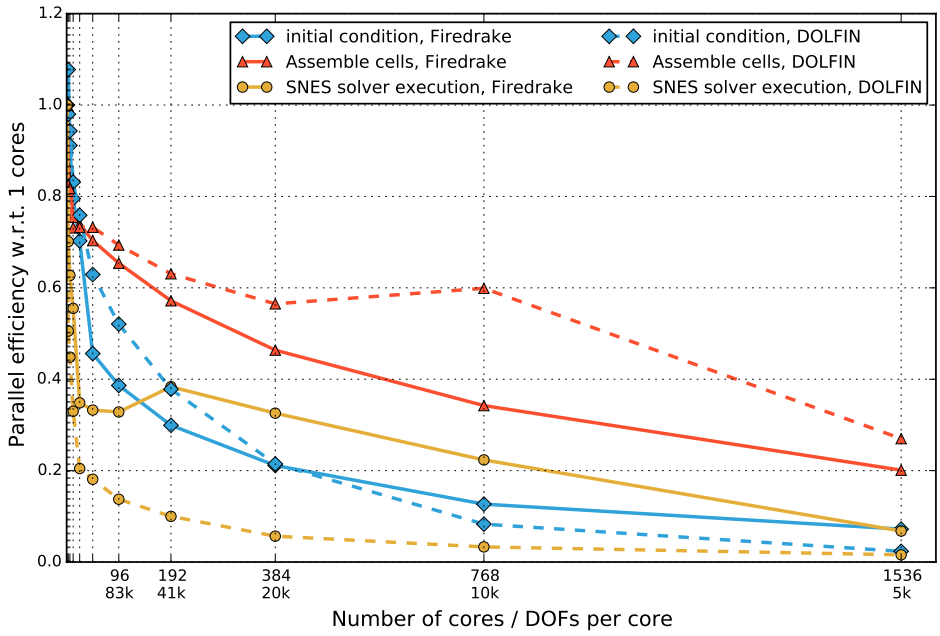

Figure 6.8: Cahn-Hilliard strong scaling efficiency with respect to a single core for a problem with 8M DOFs on up to 1536 cores run for ten time steps.

The parallel efficiency for strong scaling shown in Figure 6.8 shows advantages for DOLFIN for assembly due to the faster sequential baseline of Firedrake. DOLFIN maintains an efficiency of 60% down to 10k DOFs per core, whereas Firedrake drops to 35% at the same number of DOFs. Efficiency for evaluating the initial condition is comparable for both and considerably lower compared to assembly due to non-parallelisable overheads. Solver efficiency is considerably higher for Firedrake, maintaining 20% or above down to 10k DOFs per core, whereas DOLFIN drops to below 5% at the same number of DOFs.

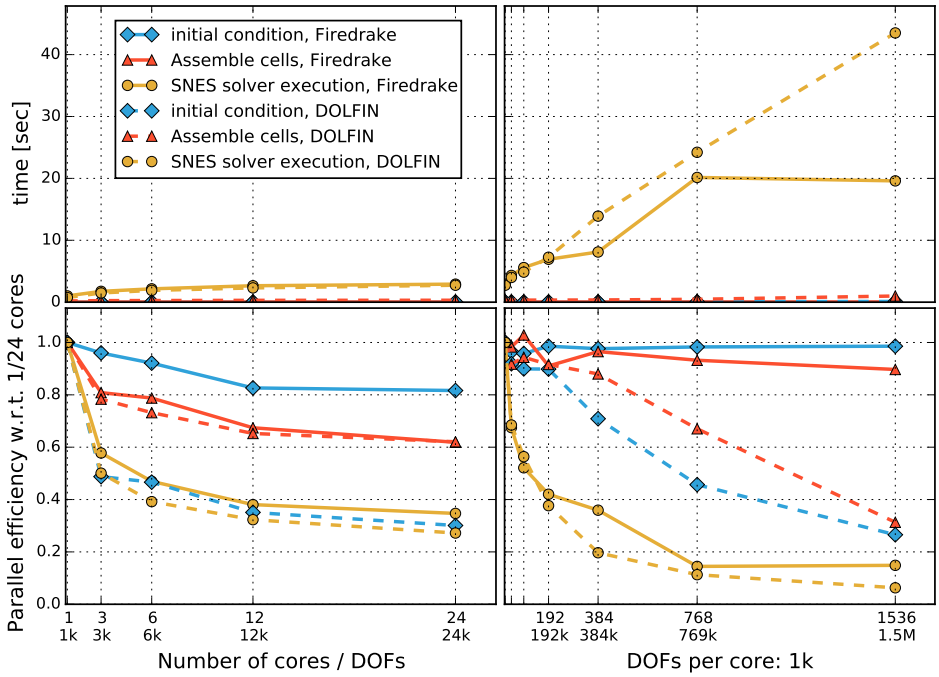

Figure 6.9: Cahn-Hilliard weak scaling intra node on 1-24 cores and inter node on 24-1536 cores.

Weak scaling run times and parallel efficiencies are shown separately for 1-24 cores intra and 24-1536 cores inter node in Figure 6.9, comparing Firedrake and DOLFIN for assembly, nonlinear solve and evaluation of the initial condition. For all of these, efficiency drops intra node from one to 12 cores and then remains at the same level for 24 cores. Evaluation of the initial condition in Firedrake achieves the best efficiency of consistently above 80%, whereas DOLFIN drops to about 30% on 24 cores. Assembly efficiency is comparable for Firedrake and DOLFIN, achieving 60% effi-

ciency or above. Solver efficiency drops to below 40% for Firedrake and below 30% for DOLFIN at 24 cores respectively.

Inter node, the behavior is considerably different, with Firedrake showing close to perfect weak scaling for evaluating the initial condition and maintaining an efficiency of 90% or greater for assembly. DOLFIN drops to below 30% for both at 1536 cores. The solver is considerably less efficient, as expected due to the low number of DOFs per core, with efficiencies dropping to about 15% for Firedrake and below 10% for DOLFIN respectively at 1536 cores.

6.5 Conclusions

The results presented in this chapter demonstrate that Firedrake delivers competitive or superior performance and scalability compared to the FEniCS tool chain for a range of finite element problems running sequentially or parallel. In particular, assembly of matrices achieves moderate and right-hand sides considerable speedups for all problems investigated.

This result is evidence of the efficiency of the PyOP2 parallel loop interface used by Firedrake to execute assembly computations. The assembly function implemented in DOLFIN's C++ library requires populating a C++ data structure to be passed in a function call into an external shared object, to evaluate the generated local assembly kernel for every cell of the mesh. PyOP2 on the other hand generates an assembly loop specifically tailored to the form in the same compilation unit as the local assembly kernel, which can therefore be inlined and leads to parallel loops running with significantly less overhead.

Choosing Python as the main language of implementation is shown to not negatively affect performance, until the strong scaling limit is hit and sequential overheads have a measureable effect. This is to be expected since Firedrake and PyOP2 implement performance critical library code in Cython, which has minimal overhead compared to native C or C++. Computations over mesh entities are always executed as PyOP2 parallel loops and therefore run in natively compiled code, generated specifically for the platform and problem.

Chapter 7

Conclusions

In this chapter, a summary of the main contributions of this thesis is given and the way these contributions support the thesis statement from Section 1.1 is examined. Controversial implementation choices are discussed and an outlook of planned and potential future work are given.

7.1 Summary

In this thesis the design and composition of two abstraction layers, PyOP2 and Firedrake, for the portable solution of partial differential equations using the finite element method on unstructured meshes has been presented. It has been argued that this composition of domain-specific abstractions is the key to computationally efficient, maintainable and composable scientific applications.

Computational efficiency for a range of different finite element problems ranging from stationary over time dependent to non-linear problems on mixed function spaces has been demonstrated in Chapter 6. Firedrake has been proven to be competitive and in many instances faster compared to the best available alternative, the DOLFIN/FEniCS tool chain, for both single core and parallel runs. This efficiency is achieved primarily by using PyOP2 as the execution layer for all computations over the mesh, in particular local assembly. The PyOP2 parallel loop construct, described in Section 4.1.3, executes a computational kernel over the mesh in natively compiled and optimised code with very low overhead.

The PyOP2/Firedrake tool chain has a very maintainable code base

which is relatively small compared to most fully featured finite element frameworks implemented as C++ or Fortran libraries such as DOLFIN or Fluidity, described in Sections 3.2.1 and 3.1.4 respectively. PyOP2 consists of only about 9,000 source lines of Python and Cython, Firedrake of about 5,000. This compact code base is achieved to a great extent by reusing tools and solutions established in the scientific community, such as the FEniCS components UFL, FFC and FIAT, the PETSc toolkit as well as PyCUDA and PyOpenCL. Python as a very high-level, accessible and interpreted language chosen for the implementation plays another important role for maintainability. There is no need to maintain a build system or an interface layer to expose a C++ or Fortran library to Python.

Composability is a main driver in the design of Firedrake. In Chapter 5 it has been shown how high-level constructs used in Firedrake are transformed and expressed in terms of the PyOP2 and PETSc abstractions. With PyOP2 as the flexible execution layer, Firedrake users have the opportunity to formulate computations not expressible in UFL as custom kernels to be executed over the mesh, thereby escaping the abstraction. Firedrake and applications built on top of it are therefore readily extendable with non finite element features. Python as the language of implementation all the way down to the level of generated code allows an application built on top of Firedrake to easily access and manipulate data structures, which are implemented in terms of PyOP2 and PETSc abstractions, As described in Section 5.1. The underlying lower level data structures can be extracted and manipulated if needed.

PyOP2 itself is highly composable, accepting and exposing data as NumPy arrays, which can be manipulated outside of a PyOP2 context using other tools from the vast scientific Python ecosystem interoperating with NumPy data types. Furthermore, the parallel loop interface is entirely agnostic to the kinds of kernels and computations to be executed as long as the constraints laid out in Chapter 4 are fulfilled. This opens up the space for extending Firedrake and applications built on top of it with non finite element features while retaining the portability afforded by PyOP2 and shows that PyOP2 is suitable as a building block in a wide range of applications operating on unstructured data.

Extensibility of the tool chain has been demonstrated by the contributions of the support for extruded meshes and the COFFEE AST optimiser.

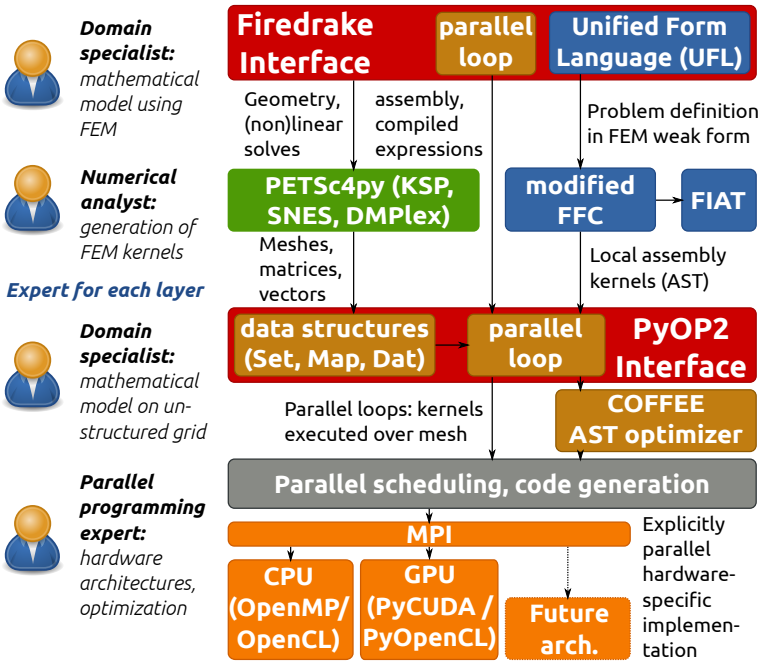

Figure 7.1: Users with different background and expertise can work on different aspects of the Firedrake/PyOP2 tool chain.

The decoupling of the Firedrake finite element layer from the PyOP2 parallel computation layer allows scientists of varying backgrounds and expertise to collaborate effectively, use and contribute to different aspects of the tool chain without having to be experts in all aspects of the implementation. As shown in Figure 7.1, the two main entry points for users of the Firedrake/PyOP2 tool chain are the Firedrake interface, for domain scientists who want to solve partial differential equations with the finite element method, and the PyOP2 interface, for those who want to execute custom kernels over the mesh. Contributors to Firedrake do not need to be concerned with the efficient execution of assembly kernels or with parallelism, since both of these are handled by PyOP2 and PETSc. A contributor to PyOP2 on the other hand does not need to be familiar with the finite element method, since from PyOP2's perspective, a local assembly kernel is "just another" kernel that can be treated as a black box. Clearly separating and encapsulating those responsibilities with two abstraction layers is not only beneficial for computational efficiency as demonstrated but also for the productivity of scientists working with them.

7.2 Discussion

In this section, a number of controversial implementation choices as well as current limitations of the Firedrake/PyOP2 tool chain are discussed.

Firedrake currently fully supports the PyOP2 sequential and OpenMP CPU backends in combination with MPI. Accelerators are not yet fully supported, which is partly due to missing or incomplete support in linear algebra libraries. As described in Section 5.6, Firedrake uses the PETSc SNES interfaces for solving linear and non-linear systems. While PETSc supports solving linear systems on CUDA and OpenCL devices via its interfaces to Cusp [Bell et al., 2014] and ViennaCL [Rupp et al., 2010], there was no interface for global matrix assembly from a GPU kernel at the time of writing. Knepley and Terrel [2013] do however acknowledge the potential of finite element integration on GPUs. To be able to support GPU assembly, PyOP2 implements its own linear algebra backend for CUDA using Cusp as described in Section 4.6.4, which does not presently support distributed parallel matrices and solvers and mixed types. Furthermore, the range of linear solvers and preconditioners is limited to those supported by Cusp, which is only a subset of PETSc's functionality.

Since PyOP2 is a runtime code generation framework, a full Python environment as well as a compiler suitable for the target platform need to be available on each compute node. This can be a limitation in certain high performance computing environments, where no suitable Python environment is available or compilation on backend compute nodes is not possible. The requirement for loading a large number of dynamic shared objects on Python interpreter startup has been identified by Frings et al. [2013] as a potential bottleneck for scalability on shared file systems. However, these limitations are not specific to PyOP2 or Firedrake and the scientific community has strong incentives to improve scalability of Python applications, from which both would immediately benefit.

As described in Section 4.8, PyOP2 represents the block structure of matrices arising from mixed systems with a PETSc MATNEST [Balay et al., 2013, Section 3.1.3], which consists of separately stored nested submatrices. In combination with the approach of splitting mixed forms, detailed in Section 5.2.4, assembling into such a nested structure greatly simplifies the design of the code generation infrastructure. Each submatrix can

be independently targeted, which allows using the same code path as for assembling a regular non-mixed problem. Furthermore, the implementation of fieldsplit preconditioners presented in Section 5.6.5 is simplified and the large memory cost of extracting submatrices is avoided, which can be a significant performance advantage when solving large systems.

7.3 Future Work

In the following, a number of areas of planned and potential future work building on and extending Firedrake and PyOP2 are identified.

7.3.1 Implementation of Fluidity Models on Top of Firedrake

Firedrake has reached a level of maturity and feature parity with DOLFIN which make it suitable for third parties to use and build upon. Jacobs and Piggott [2014] in the Applied Modelling and Computation Group (AMCG) at Imperial College have started working on porting models implemented in the Fluidity CFD code described in Section 3.1.4 to Firedrake[1]. Previously, adding a new model required contributors to be familiar with and modify different parts of the Fluidity Fortran library, while making sure none of the existing functionality is negatively affected. Building upon the high-level interface provided by Firedrake is an enormous productivity gain for Fluidity developers, who are able to implement and test models as independent modules in a matter of days. Features of the model not expressible in UFL can be added as custom kernels and directly executed on the PyOP2 level. At the same time, the Fluidity community can take advantage of the portability offered by Firedrake and immediately benefit from new features or performance improvements in the tool chain. Fluidity users not familiar with Python can continue to configure these models using the familiar graphical user interface Diamond, interfaced through the Python bindings to the SPUD library [Ham et al., 2009].

7.3.2 Automated Derivation of Adjoints

The automated derivation of the adjoint of a forward model implemented in the DOLFIN Python interface is enabled by dolfin-adjoint [Farrell et al.,

[1]Project repository: https://github.com/firedrakeproject/firedrake-fluids

2013]. This automation is achieved by making use of the high-level symbolic representation of the problem and annotates the temporal structure of the model at runtime. The adjoint variational forms are derived symbolically and the FEniCS form compiler is used to generate assembly kernels for the derived adjoint model. This approach naturally translates when applying dolfin-adjoint to Firedrake, which exposes a DOLFIN-compatible API such that hooks can put in place in the same way. A proof-of-concept implementation of firedrake-adjoint, dolfin-adjoint applied to Firedrake, only required minimal changes on either side. This implementation has only been lightly tested and cannot yet be considered robust, however future work on this integration is planned.

To be able to make use of the full Firedrake functionality including user-defined parallel loops, it is necessary to also put hooks in place to differentiate custom kernels not provided by FFC. While it may be feasible to apply automatic differentiation to the C kernel code, a more promising approach is a linearisation and derivation of the adjoint on the level of the COFFEE abstract syntax tree.

7.3.3 Geometric Multigrid Methods

A feature currently in development is support for geometric multigrid and, as a prerequisite, parallel uniform mesh refinement and coarsening to be able to implement restriction and prolongation operators. This work is being used to construct a matrix-free multigrid preconditioner for the shallow water pressure correction equation. The equations were discretised using both $DG_0 + RT_0$ and $DG_1 + BDFM_1$ mixed function spaces. Preliminary benchmarks obtained up to 23% of the achievable STREAM bandwidth on an ARCHER compute node [Mueller et al., 2014].

7.3.4 Scalability of Firedrake and LLVM Code Generation

An evaluation of the scalability of Firedrake to very large core counts is planned on the UK national supercomputing facility Archer. A potential issue is the need to call a vendor compiler for the just-in-time compilation of code on all the backend nodes and compiling a shared object file that needs to be written to and re-read from disk on a shared file system.

To overcome this issue, an investigation of using in-memory compila-

tion with LLVM [Lattner and Adve, 2004] is planned, replacing the current CPU code generation and runtime compilation infrastructure of PyOP2 with a backend that builds an LLVM intermediate representation (IR) directly, which is compiled to machine code in memory and entirely avoids having to touch the file system. This increases both the scalability on current systems and reduces porting efforts to future HPC systems.

7.3.5 Firedrake on Accelerators

Fully supporting Firedrake on accelerators and demonstrating performance portability over a broader range of architectures is another priority on the roadmap. A promising strategy in particular on GPU architectures are matrix-free methods, where no sparse matrix is explicitly assembled. Instead, a callback function for evaluating the sparse matrix vector product (SpMV), used as a black box routine by iterative solvers, is provided. One possible implementation that has been been shown by Markall et al. [2012] to outperform matrix assembly on GPU architectures is the Local Matrix Approach (LMA), where the local element matrices are used directly in the SpMV instead of first assembling a global sparse matrix.

PETSc has extensive support for matrix-free methods both for its linear and non-linear solvers [Balay et al., 2013, Section 3.3, 5.5]. However, most preconditioners are implemented to work on matrices and can therefore not be used with matrix-free methods, where custom preconditioners have to be implemented.

7.3.6 Adaptive Mesh Refinement

Adaptive mesh refinement and coarsening, where the mesh topology and geometry are dynamically changed by splitting or combining cells based on some error indicator computed over the mesh, is essential for achieving good performance in certain types of applications. PyOP2 assumes a fixed, immutable mesh topology and dynamically changing the sizes of data structures is currently not supported. Adapting the mesh requires creating new versions of PyOP2 data structures with different sizes and efficiently transferring data from the old to the new mesh. A promising implementation choice would be the integration of PRAgMaTIc, the Parallel anisotRopic Adaptive Mesh ToolkIt [Rokos et al., 2011], with the

PyOP2/Firedrake tool chain. PRAgMaTIc has already been interfaced to DOLFIN and the integration with PETSc DMPlex is on the roadmap.

Bibliography

Martin S. Alnaes, Anders Logg, Kent-Andre Mardal, Ola Skavhaug, and Hans Petter Langtangen. Unified framework for finite element assembly. *International Journal of Computational Science and Engineering*, 4(4):231–244, 2009.

Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2): 9:1–9:37, March 2014. doi: 10.1145/2566630.

Martin Sandve Alnæs. UFL: a finite element form language. In Anders Logg, Kent-Andre Mardal, Garth Wells, Timothy J. Barth, Michael Griebel, David E. Keyes, Risto M. Nieminen, Dirk Roose, and Tamar Schlick, editors, *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, pages 303–338. Springer Berlin Heidelberg, 2012.

Martin Sandve Alnæs, Anders Logg, and Kent-Andre Mardal. UFC: a finite element code generation interface. In Anders Logg, Kent-Andre Mardal, and Garth Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, number 84 in Lecture Notes in Computational Science and Engineering, pages 283–302. Springer Berlin Heidelberg, January 2012.

Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. MUMPS: A general purpose distributed memory sparse solver. In Tor Sørevik, Fredrik Manne, Assefaw Hadish Gebremedhin, and Randi Moe, editors, *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*, number 1947 in Lecture Notes in Computer Science, pages 121–130. Springer Berlin Heidelberg, January 2001. doi: 10.1007/3-540-70734-4_16.

Stefan Andersson. Cray XC30 architecture overview, January 2014.

URL http://www.archer.ac.uk/training/courses/craytools/pdf/architecture-overview.pdf.

Applied Modelling and Computation Group (AMCG). Fluidity manual 4.1.11, November 2013. URL http://launchpad.net/fluidity/4.1/4.1.11/+download/fluidity-manual-4.1.11.pdf.

S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. Curfman McInnes, B. Smith, and H. Zhang. PETSc users manual revision 3.4, May 2013. URL http://www-unix.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf.

Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 163–202. Birkhäuser Boston, January 1997.

W. Bangerth, R. Hartmann, and G. Kanschat. deal.II - a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.*, 33(4), August 2007. doi: 10.1145/1268776.1268779.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. part II: implementation and tests in DUNE. *Computing*, 82(2-3):121–138, July 2008a. doi: 10.1007/s00607-008-0004-9.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. part i: abstract framework. *Computing*, 82(2-3):103–119, July 2008b. doi: 10.1007/s00607-008-0003-x.

D. M. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599–609, July 2003. doi: 10.1016/S0167-739X(02)00171-1.

S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, 2011. doi: 10.1109/MCSE.2010.118.

Nathan Bell, Steven Dalton, Filipe Maia, and Michael Garland. CUSP : A c++ templated sparse matrix library, 2014. URL http://cusplibrary.github.io/.

Markus Blatt and Peter Bastian. The iterative solver template library. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors,

Applied Parallel Computing. State of the Art in Scientific Computing, number 4699 in Lecture Notes in Computer Science, pages 666–675. Springer Berlin Heidelberg, January 2007.

A. Bolis, C. D. Cantwell, R. M. Kirby, and S. J. Sherwin. From h to p efficiently: Optimal implementation strategies for explicit time-dependent problems using the spectral/hp element method. 2013. URL http://www2.imperial.ac.uk/ssherw/spectralhp/papers/IJNMF-BoCaKiSh-13.pdf. submitted.

J. Bosch, D. Kay, M. Stoll, and A. Wathen. Fast solvers for Cahn–Hilliard inpainting. *SIAM Journal on Imaging Sciences*, 7(1):67–97, 2014. doi: 10.1137/130921842.

A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, pages 333–390, 1977.

Tobias Brandvik and Graham Pullan. SBLOCK: a framework for efficient stencil-based PDE solvers on multi-core platforms. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, page 1181–1188, Washington, DC, USA, 2010. IEEE Computer Society. doi: 10.1109/CIT.2010.214.

Susanne C. Brenner and L. Ridgway Scott. *The mathematical theory of finite element methods*. Springer, New York NY, 3rd ed. edition, 2008.

Franco Brezzi, Jim Douglas Jr, and L. D. Marini. Two families of mixed finite elements for second order elliptic problems. *Numerische Mathematik*, 47(2):217–235, June 1985. doi: 10.1007/BF01389710.

C. D. Cantwell, S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly. From h to p efficiently: Selecting the optimal spectral/hp discretisation in three dimensions. *Mathematical Modelling of Natural Phenomena*, 6(03):84–96, 2011a. doi: 10.1051/mmnp/20116304.

C.D. Cantwell, S.J. Sherwin, R.M. Kirby, and P.H.J. Kelly. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers & Fluids*, 43(1):23–28, April 2011b. doi: 16/j.compfluid.2010.08.012.

M. Christen, O. Schenk, and H. Burkhart. PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676 –687, May 2011. doi: 10.1109/IPDPS.2011.70.

Philippe G. Ciarlet. *Numerical analysis of the finite element method*. Presses de l'Université de Montréal, 1976.

Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, page 36–47, New York, NY, USA, 2005. ACM. doi: 10.1145/1065944.1065950.

Krzysztof Czarnecki, Ulrich W Eisenecker, Robert Glück, David Vandevoorde, and Todd L Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, page 25–39, London, UK, 2000. Springer-Verlag. ACM ID: 724187.

Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9): 1124–1139, September 2011. doi: 10.1016/j.advwatres.2011.04.013.

Timothy A. Davis. Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, 2004. doi: 10.1145/992200.992206.

Andreas Dedner, Robert Klöfkorn, Martin Nolte, and Mario Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the dune-fem module. *Computing*, 90(3-4):165–196, November 2010. doi: 10.1007/s00607-010-0110-3.

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 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*, SC '11, page 9:1–9:12, New York, NY, USA, 2011. ACM. doi: 10.1145/2063384.2063396.

DUNE Team. DUNE: distributed and unified numerics environment, January 2014. URL http://www.dune-project.org/.

Robert D. Falgout, Jim E. Jones, and Ulrike Meier Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In Are Magnus Bruaset and Aslak Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, number 51 in Lecture Notes in Computational Science and Engineering, pages 267–294. Springer Berlin Heidelberg, January 2006. doi: 10.1007/3-540-31619-1_8.

P. Farrell, D. Ham, S. Funke, and M. Rognes. Automated derivation of the ad-

joint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, 35(4):C369–C393, January 2013. doi: 10.1137/120873558.

Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, page 361–366, New York, NY, USA, 2005. ACM. doi: 10.1145/1088149.1088197.

Matteo Frigo and Volker Strumpen. The memory behavior of cache oblivious stencil computations. *The Journal of Supercomputing*, 39(2):93–112, February 2007. doi: 10.1007/s11227-007-0111-y.

Wolfgang Frings, Dong H. Ahn, Matthew LeGendre, Todd Gamblin, Bronis R. de Supinski, and Felix Wolf. Massively parallel loading. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 389–398, New York, NY, USA, 2013. ACM. doi: 10.1145/2464996.2465020.

Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, September 2009. doi: 10.1002/nme.2579.

M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly. Performance analysis and optimization of the OP2 framework on many-core architectures. *The Computer Journal*, 55(2):168–180, January 2012. doi: 10.1093/comjnl/bxr062.

M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly. Designing OP2 for GPU architectures. *Journal of Parallel and Distributed Computing*, 73(11):1451–1460, November 2013. doi: 10.1016/j.jpdc.2012.07.008.

D. A. Ham, P. E. Farrell, G. J. Gorman, J. R. Maddison, C. R. Wilson, S. C. Kramer, J. Shipton, G. S. Collins, C. J. Cotter, and M. D. Piggott. Spud 1.0: generalising and automating the user interfaces of scientific computer models. *Geosci. Model Dev.*, 2(1):33–42, March 2009. doi: 10.5194/gmd-2-33-2009.

Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005. doi: 10.1145/1089014.1089021.

M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand*, 49(6):409–436, 1952.

C. T. Jacobs and M. D. Piggott. Firedrake-fluids v0.1: numerical modelling of shallow water flows using a performance-portable automated solution framework. *Geosci. Model Dev. Discuss.*, 7(4):5699–5738, August 2014. ISSN 1991-962X. doi: 10.5194/gmdd-7-5699-2014.

S. Kamil, Cy Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1 –12, April 2010. doi: 10.1109/IPDPS.2010.5470421.

OpenCL Working Group Khronos. The OpenCL specification, version 2.0, November 2013. URL http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

R. C. Kirby, M. G. Knepley, and L. R. Scott. Evaluation of the action of finite element operators. Technical Report TR–2004–07, University of Chicago, Department of Computer Science, 2004.

R. C. Kirby, A. Logg, L. R. Scott, and A. R. Terrel. Topological optimization of the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 28(1):224–240, 2006.

Robert C. Kirby. Algorithm 839: FIAT, a new paradigm for computing finite element basis functions. *ACM Trans. Math. Softw.*, 30(4):502–516, 2004. doi: 10.1145/1039813.1039820.

Robert C. Kirby. FIAT: numerical construction of finite element basis functions. In Anders Logg, Kent-Andre Mardal, and Garth Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, number 84 in Lecture Notes in Computational Science and Engineering, pages 247–255. Springer Berlin Heidelberg, January 2012.

Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32(3):417–444, September 2006. doi: 10.1145/1163641.1163644.

Robert C. Kirby and Anders Logg. Efficient compilation of a class of variational forms. *ACM Trans. Math. Softw.*, 33(3), August 2007. doi: 10.1145/1268769.1268771.

Robert C. Kirby and Anders Logg. The finite element method. In Anders Logg, Kent-Andre Mardal, and Garth Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, number 84 in Lecture Notes in Computational Science and Engineering, pages 77–94. Springer Berlin Heidelberg, January 2012a.

Robert C. Kirby and Anders Logg. Tensor representation of finite element variational forms. In Anders Logg, Kent-Andre Mardal, and Garth Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, number 84 in Lecture Notes in Computational Science and Engineering, pages 159–162. Springer Berlin Heidelberg, January 2012b. doi: 10.1007/978-3-642-23099-8_8.

Robert C Kirby and L. Ridgway Scott. Geometric optimization of the evaluation of finite element matrices. *SIAM Journal on Scientific Computing*, 29(2):827–841 (electronic), 2007.

Robert C. Kirby, Matthew Knepley, Anders Logg, and L. Ridgway Scott. Optimizing the evaluation of finite element matrices. *SIAM Journal on Scientific Computing*, 27(3):741–758, January 2005. doi: 10.1137/040607824.

Robert C. Kirby, Anders Logg, Marie E. Rognes, and Andy R. Terrel. Common and unusual finite elements. In Anders Logg, Kent-Andre Mardal, and Garth Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, number 84 in Lecture Notes in Computational Science and Engineering, pages 95–119. Springer Berlin Heidelberg, January 2012. doi: 10.1007/978-3-642-23099-8_3.

Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, March 2012. doi: 10.1016/j.parco.2011.09.001.

Matthew Knepley. The portable extensible toolkit for scientific computing - PETSc tutorial. Orsay, France, June 2013. URL http://calcul.math.cnrs.fr/IMG/pdf/ParisTutorial.pdf.

Matthew G. Knepley and Andy R. Terrel. Finite element integration on GPUs. *ACM Trans. Math. Softw.*, 39(2):10:1–10:13, February 2013. doi: 10.1145/2427023.2427027.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, page 75–, Washington, DC, USA, 2004. IEEE Computer Society.

Anders Logg. Automating the finite element method. *Archives of Computational Methods in Engineering*, 14(2):93–138, June 2007. doi: 10.1007/s11831-007-9003-9.

Anders Logg and Garth N. Wells. DOLFIN: automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):1–28, 2010. doi: 10.1145/1731022.1731030.

Anders Logg, Kent-Andre Mardal, and Garth N. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer Berlin Heidelberg, 2012a.

Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Finite element assembly. In Anders Logg, Kent-Andre Mardal, and Garth Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, number 84 in Lecture Notes in Computational Science and Engineering, pages 141–146. Springer Berlin Heidelberg, January 2012b. doi: 10.1007/978-3-642-23099-8_6.

Anders Logg, Garth N. Wells, and Johan Hake. DOLFIN: a C++/Python finite element library. In Anders Logg, Kent-Andre Mardal, and Garth Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, number 84 in Lecture Notes in Computational Science and Engineering, pages 173–225. Springer Berlin Heidelberg, January 2012c.

Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. FFC: the FEniCS form compiler. In Anders Logg, Kent-Andre Mardal, and Garth Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, number 84 in Lecture Notes in Computational Science and Engineering, pages 227–238. Springer Berlin Heidelberg, January 2012d.

Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. COFFEE: an optimizing compiler for finite element local assembly. 2014. submitted.

Graham R. Markall, Andras Slemmer, David A. Ham, Paul H.J. Kelly, Chris D. Cantwell, and Spencer J. Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids*, 2012. doi: 10.1002/fld.3648.

Graham R. Markall, Florian Rathgeber, Lawrence Mitchell, Nicolas Loriant, Carlo Bertolli, David A. Ham, and Paul H. J. Kelly. Performance-portable finite element assembly using PyOP2 and FEniCS. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings*, number 7905 in Lecture Notes in Computer Science, pages 279–289. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-38750-0_21.

Graham Robert Markall. *Multilayered Abstractions for Partial Differential Equations*. PhD thesis, Imperial College, 2013.

John D. McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, page 19–25, 1995.

William C. Mills-Curran, Amy P. Gilkey, and Dennis P. Flanagan. EXODUS: a finite element file format for pre-and postprocessing. Technical report, Sandia National Labs., Albuquerque, NM (USA), 1988.

Lawrence Mitchell. Partitioning and numbering meshes for efficient MPI-parallel execution in PyOP2. In *FEniCS'13 Workshop*, University of Cambridge, March 2013. URL http://fenicsproject.org/pub/workshops/fenics13/slides/Mitchell.pdf.

Eike H. Mueller, Colin J. Cotter, David A. Ham, Lawrence Mitchell, and Robert Schleichl. Efficient multigrid solvers for mixed finite element discretisations in NWP models, October 2014. URL http://www.ecmwf.int/sites/default/files/HPC-WS-Mueller.pdf.

J.-C. Nédélec. Mixed finite elements in $\mathbf{R}^3$. *Numer. Math.*, 35(3):315–341, 1980. doi: 10.1007/BF01396415.

J.-C. Nédélec. A new family of mixed finite elements in $\mathbf{R}^3$. *Numer. Math.*, 50(1): 57–81, 1986. doi: 10.1007/BF01389668.

NVIDIA. Kepler GK110 compute architecture white paper. Technical Report v1.0, 2012.

NVIDIA. CUDA c programming guide v5.5, July 2013. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

Christoph Pflaum. Expression templates for partial differential equations. *Computing and Visualization in Science*, 4(1):1–8, November 2001. doi: 10.1007/s007910100051.

M. D. Piggott, G. J. Gorman, C. C. Pain, P. A. Allison, A. S. Candy, B. T. Martin, and M. R. Wells. A new computational framework for multi-scale ocean modelling based on adapting unstructured meshes. *International Journal for Numerical Methods in Fluids*, 56(8):1003–1015, 2008. doi: 10.1002/fld.1663.

Diane Poirier, Steven Allmaras, Douglas McCarthy, Matthew Smith, and Francis Enomoto. The CGNS system. In *29th AIAA, Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics, 1998. doi: 10.2514/6.1998-3007.

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4): 32:1–32:12, July 2012. doi: 10.1145/2185520.2185528.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, page 519–530, New York, NY, USA, 2013. ACM. doi: 10.1145/2462156.2462176.

Florian Rathgeber, Graham R. Markall, Lawrence Mitchell, Nicolas Loriant, David A. Ham, Carlo Bertolli, and Paul H.J. Kelly. PyOP2: a high-level framework for performance-portable simulations on unstructured meshes. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1116–1123, 2012. doi: 10.1109/SC.Companion.2012.134.

P.-A. Raviart and J. M. Thomas. A mixed finite element method for 2nd order elliptic problems. In *Mathematical aspects of finite element methods (Proc. Conf., Consiglio Naz. delle Ricerche (C.N.R.), Rome, 1975)*, pages 292–315. Lecture Notes in Math., Vol. 606. Springer, Berlin, 1977.

James Reinders. An overview of programming for intel xeon processors and intel xeon phi coprocessors. Technical report, Intel, 2012.

Marie E. Rognes. Mixed formulation for poisson equation, November 2012. URL http://fenicsproject.org/documentation/dolfin/1.4.0/python/demo/documented/mixed-poisson/python/documentation.html.

Marie E. Rognes, David A. Ham, Colin J. Cotter, and Andrew T. T. McRae. Automating the solution of PDEs on the sphere and other manifolds in FEniCS 1.2. *Geoscientific Model Development Discussions*, 6(3):3557–3614, July 2013. doi: 10.5194/gmdd-6-3557-2013.

Georgios Rokos, Gerard Gorman, and Paul H. J. Kelly. Accelerating anisotropic mesh adaptivity on nVIDIA's CUDA using texture interpolation. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, number 6853 in Lecture Notes in Computer Science, pages 387–398. Springer Berlin Heidelberg, January 2011. doi: 10.1007/978-3-642-23397-5_38.

Karl Rupp, Florian Rudolf, and Josef Weinbub. ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs. *Proc. GPUScA*, page 51–56, 2010.

Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

Jonathan Richard Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry Towards Geometric Engineering*, number 1148 in Lecture Notes in Computer Science, pages 203–222. Springer Berlin Heidelberg, January 1996. doi: 10.1007/BFb0014497.

J.E. Stone, D. Gohara, and Guochun Shi. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, 2010. doi: 10.1109/MCSE.2010.69.

Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, page 117–128, New York, NY, USA, 2011. ACM. doi: 10.1145/1989493.1989508.

D. Unat, Jun Zhou, Yifeng Cui, S.B. Baden, and Xing Cai. Accelerating a 3D finite-difference earthquake simulation with a c-to-CUDA translator. *Computing in Science Engineering*, 14(3):48 –59, June 2012. doi: 10.1109/MCSE.2012.44.

Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, ICS '11, page 214–224, New York, NY, USA, 2011. ACM. doi: 10.1145/1995896.1995932.

Peter E.J. Vos, Spencer J. Sherwin, and Robert M. Kirby. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *Journal of Computational Physics*, 229(13):5161–5181, July 2010. doi: 10.1016/j.jcp.2010.03.031.

Peter E.J. Vos, Claes Eskilsson, Alessandro Bolis, Sehun Chun, Robert M. Kirby, and Spencer J. Sherwin. A generic framework for time-stepping partial differential equations (PDEs): general linear methods, object-oriented implementation and application to fluid problems. *International Journal of Computational Fluid Dynamics*, 25(3):107–125, 2011. doi: 10.1080/10618562.2011.575368.

Joerg Walter and Mathias Koch. Boost basic linear algebra (uBLAS), 2014. URL http://www.boost.org/libs/numeric/ublas/.

Ilmar M. Wilbers, Kent-Andre Mardal, and Martin S. Alnæs. Instant: just-in-time compilation of C/C++ in python. In Anders Logg, Kent-Andre Mardal, and Garth Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, number 84 in Lecture Notes in Computational Science and Engineering, pages 257–272. Springer Berlin Heidelberg, January 2012. doi: 10.1007/978-3-642-23099-8_14.

Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, PASCO '07, page 24–32, New York, NY, USA, 2007. ACM. doi: 10.1145/1278177.1278183.

Yongpeng Zhang and Frank Mueller. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 155–164, New York, NY, USA, 2012. ACM. doi: 10.1145/2259016.2259037.

Kristian B. Ølgaard and Garth N. Wells. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Softw.*, 37(1):8:1–8:23, January 2010. doi: 10.1145/1644001.1644009.

Kristian B. Ølgaard and Garth N. Wells. Quadrature representation of finite element variational forms. In Anders Logg, Kent-Andre Mardal, and Garth Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, number 84 in Lecture Notes in Computational Science and Engineering, pages 147–158. Springer Berlin Heidelberg, January 2012. doi: 10.1007/978-3-642-23099-8_7.