Firedrake: Automating the Finite Element Method by Composing Abstractions

FLORIAN RATHGEBER, DAVID A. HAM, LAWRENCE MITCHELL, MICHAEL LANGE, FABIO LUPORINI, ANDREW T. T. MCRAE, GHEORGHE-TEODOR BERCEA, GRAHAM R. MARKALL, and PAUL H. J. KELLY, Imperial College London

Firedrake is a new tool for automating the numerical solution of partial differential equations. Firedrake adopts the domain-specific language for the finite element method of the FEniCS project, but with a pure Python runtime-only implementation centered on the composition of several existing and new abstractions for particular aspects of scientific computing. The result is a more complete separation of concerns that eases the incorporation of separate contributions from computer scientists, numerical analysts, and application specialists. These contributions may add functionality or improve performance.

Firedrake benefits from automatically applying new optimizations. This includes factorizing mixed function spaces, transforming and vectorizing inner loops, and intrinsically supporting block matrix operations. Importantly, Firedrake presents a simple public API for escaping the UFL abstraction. This allows users to implement common operations that fall outside of pure variational formulations, such as flux limiters.

Categories and Subject Descriptors: G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Finite Element Methods*; G.4 [**Mathematical Software**]: Algorithm Design and Analysis

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Abstraction, code generation, UFL

ACM Reference Format:
Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2016. Firedrake: Automating the finite element method by composing abstractions. ACM Trans. Math. Softw. 43, 3, Article 24 (December 2016), 27 pages.
DOI: http://dx.doi.org/10.1145/2998441

This work was supported by the Engineering and Physical Sciences Research Council (grants EP/I00677X/1, EP/L000407/1, EP/I012036/1), the Natural Environment Research Council (grants NE/G523512/1, NE/I021098/1, NE/K006789/1, NE/K008951/1), and the Grantham Institute, Imperial College London.
Authors' addresses: F. Rathgeber (current address), European Centre for Medium-Range Weather Forecasts, Shinfield Park, Reading, RG2 9AX, United Kingdom; email: florian.rathgeber@ecmwf.int; D. A. Ham, Department of Mathematics, Imperial College London, London, SW7 2AZ, United Kingdom; email: david.ham@imperial.ac.uk; L. Mitchell, Department of Mathematics and Department of Computing, Imperial College London, London, SW7 2AZ, United Kingdom; email: lawrence.mitchell@imperial.ac.uk; M. Lange, Department of Earth Science and Engineering, Imperial College London, London, SW7 2AZ, United Kingdom; email: michael.lange@imperial.ac.uk; F. Luporini, Department of Earth Science and Engineering, Imperial College London, London, SW7 2AZ, United Kingdom; email: f.luporini12@imperial.ac.uk; A. T. T. McRae (current address), Department of Mathematical Sciences, University of Bath, Bath, BA2 7AY, United Kingdom; email: a.t.t.mcrae@bath.ac.uk; G.-T. Bercea (current address), IBM TJ Watson Research Center, 1101 Kitchawan Road, Yorktown Heights, NY, 10598, USA; email: gheorghe-teod.bercea@ibm.com; G. Markall (current address), Embecosm Ltd, Palamos House #208, 66/67 High Street, Lymington, SO41 9AL, United Kingdom; email: graham.markall@embecosm.com; P. H. J. Kelly, Department of Computing, Imperial College London, London, SW7 2AZ, United Kingdom; email: p.kelly@imperial.ac.uk.

ACM 0098-3500/2016/12-ART24 $15.00
DOI: http://dx.doi.org/10.1145/2998441

1. INTRODUCTION

The numerical solution of partial differential equations (PDEs) is an indispensable tool in much of modern science and engineering. However, the successful development and application of advanced PDE solvers on complex problems requires the combination of diverse skills across mathematics, scientific computing, and low-level code optimization, which is rarely at expert level in a single individual. For the finite element method, which will be the focus of this work, this set of skills includes at least knowledge of the system being simulated, analysis of the resulting PDEs, numerical analysis to create appropriate discretizations, mesh generation, graph theory to create data structures on those meshes, the analysis and implementation of linear and nonlinear solvers, parallel algorithms, vectorization, and loop nest optimization under memory constraints.

The development of such software is therefore increasingly a multidisciplinary effort, and its design must enable scientists with different specializations to collaborate effectively without requiring each one of them to understand every aspect of the system in full detail. The key to achieving this is to abstract, automate, and compose the various processes involved in numerically solving PDEs. At some level, this process is a familiar one: few of the people who write C or Fortran code really understand how the compiler works, and they need not do so. Instead, the programmer understands the rules of the language and programs to that model. Similarly, mathematical operations and results are frequently employed without having their derivation or proof immediately at hand. In other words, mathematical and software abstractions such as languages and theorems enable a separation of concerns between *developing* a technique and *employing* it.

This article presents a new contribution to the automation and abstraction of the finite element method. Previous work, most especially the Unified Form Language (UFL) [Alnæs et al. 2014] employed by the FEniCS project [Logg et al. 2012; Logg and Wells 2010], enables scientists to express PDEs in a high-productivity interpreted language close to the mathematics. Implementations of the finite element method have traditionally been tightly coupled to the numerics, requiring contributors to have a deep understanding of both. FEniCS creates a separation of concerns between *employing* the finite element method and *implementing* it. Firedrake goes beyond this by introducing a new abstraction, PyOP2, to create a separation within the implementation layer between the *local discretization* of mathematical operators and their *parallel execution* over the mesh. This separation enables numericists to contribute ever-more sophisticated finite elements, whereas computer scientists, expert in parallel execution but not in numerics, contribute more advanced execution strategies.

In addition to admitting uniformly high-performance mesh iteration, the introduction of the additional abstraction layer results in a very compact code base. The resulting core Firedrake code has only around 5,000 lines of executable code, whereas the PyOP2 parallel execution layer has fewer than 9,000 executable lines. This compactness is evidence of the effectiveness of the abstraction choice made and is of immense benefit to the maintainability and extensibility of the code base.

The rest of the article is organized as follows. Section 2 describes the state of the art in abstractions for scientific computing, particularly the finite element method. Section 3 details the abstractions, and their implementations, which are composed to form the Firedrake toolchain. Sections 4 and 5 describe in more detail the Firedrake and PyOP2 abstraction layers, which are the core contribution of this work. Section 6 describes an extensive computational verification of the performance and capability of the Firedrake system. Finally, Section 7 presents some current limitations and future extensions.

2. MATHEMATICAL AND SOFTWARE ABSTRACTION OF THE FINITE ELEMENT METHOD

A particular advantage of the finite element method as a class of numerical methods for PDEs is that the entire algorithm can frequently be described in highly abstract mathematical terms. In the simplest cases, the mathematics of the method can be completely specified by a PDE in weak form, along with the desired boundary conditions and the discrete function spaces from which the solution and test functions should be drawn. Of course, a complete mathematical specification of the method is not the same as an efficient, parallel, and bug-free software implementation. As a result, countless years of scientists' time have been spent over the decades implementing finite element methods in low-level Fortran and C code.

Although hand coding algorithms at a low level can produce efficient code, that approach suffers from several serious drawbacks. The key among these is a premature loss of mathematical abstraction: the symbolic structure of differential equations, function spaces, and integrals is replaced by loops over arrays of coefficient values and individual floating-point operations. Interspersed among these are parallel communication calls, threading and vectorization directives, and so forth.

The original abstract mathematical expression of the equations embodies a separation of concerns: the equation to be solved is separated from its discretization, from the linear and/or nonlinear solver techniques to be applied, and from the implementation of the assembly and solvers. A low-level implementation loses this separation of concerns. This has many deleterious effects. First, choices are committed to far too early: deciding to change discretization or the equation to be solved requires the implementation to be recoded. Second, the developer must deal with the mixture of equations, discretization, and implementation all at once. Reasoning about the mathematics of the code requires the developer to mentally reinterpret series of primitive instructions as the high-level abstract mathematics that they represent, and any change made to the desired behavior must be implemented by manually working out the correct series of primitive operations. Changing and debugging the code in this way also carries the risks of defeating implementation choices that were made to optimize performance, as well as the risk of introducing bugs.

2.1. Benefits and Limits of Mathematical Library Interfaces

Given the limitations of hand-writing low-level code, it is unsurprising that much effort has been devoted to the development of finite element and other scientific software that maintains something of the mathematical abstraction of the methods. A core feature of these approaches is that they present a programming environment in which the data objects correspond to the higher-level mathematical objects found in the finite element method. For example, there may be data objects corresponding to sparse matrices, distributed vectors, finite elements, and function spaces.

A common and highly successful approach to this is for these mathematical objects to be represented as data objects in object-oriented libraries. High-level mathematical operations are then expressed as operations on these objects, resulting in method calls. The actual implementation of the primitive numerical operations on arrays of floating-point numbers is hidden in the implementation of those methods. Deal.II [Bangerth et al. 2007, 2013] and Dune-FEM [Dedner et al. 2010] are prominent examples of object-oriented finite element packages, and there are many others. The object-oriented library approach has also been very successfully applied by leading sparse linear algebra libraries, notably including PETSc [Balay et al. 2014] and Trilinos' EPetra and TPetra packages [Heroux et al. 2005].

The library approach is most successful where the mathematical operations specified by the application developer have a fairly large granularity: for example, in the case of linear algebra packages, the smallest operations (e.g., scaling vectors or taking a dot product) still involve an iteration over the entire vector, and operations such as a linear solve are much larger. This means that the implementation of tight inner loops and much or all of the parallelism can be hidden from the application developer, thereby achieving the desired separation of algorithm and implementation.

Conversely, a key domain of variability in the field of numerical methods for PDEs lies at the level of the innermost loops: the numerical operations conducted at each mesh entity (cell, face, edge, or vertex) depend on the PDE being solved and the numerical method employed. This means that the level of code at which the algorithm is expressed is at or below the level at which factors such as loop order, data layout, and function calls become performance critical. Finegrain parallelism, such as threading and vectorization, may also need to be expressed at this level. In contrast to the case of linear algebra, in the PDE discretization, a critical part of the user algorithm describes very fine grain operations, which must be woven together to form an efficient, parallel implementation. For this reason, library-based finite element packages such as Dune-FEM and Deal.II require that C++ implementations of integrals expressed as low-level sums over quadrature points be provided by the application developer.

2.2. Domain-Specific Languages for Finite Elements

The desire to express the integrals at the heart of the finite element method in a high-level mathematical language while still producing efficient low-level code implementing these integrals has led some projects to adopt a different approach. Rather than writing directly executable code utilizing library calls to access functionality, the numerics of the finite element method are specified purely symbolically in a special-purpose language. A specialized compiler or interpreter then uses this input to generate low-level, efficient code. Within this category, we can distinguish between stand-alone languages with their own parser and embedded languages implemented in an existing general-purpose compiled or interpreted language. A prominent example of the former class is freefem++ [Hecht 2012], whereas UFL [Alnæs et al. [2014] and Sundance [Long et al. 2010] are examples of finite element domain-specific languages (DSLs) embedded in Python and C++, respectively.

A well-designed DSL not only enables the application programmer to express her problem clearly, mathematically, and concisely, but also provides the compiler writer with a great deal of freedom to make optimal implementation choices, including those that are too verbose, tedious, and error prone to implement by hand. For example, the FEniCS Form Compiler (FFC), which takes UFL as its input language, has been used to develop highly optimized quadrature [Ølgaard and Wells 2010] and tensor reduction [Kirby et al. 2005] implementations of finite element assembly.

A further benefit of the DSL approach is that the symbolic mathematical form of the variational problem is available in the program code. This can be exploited to automate reasoning about the mathematical structure of the problem—for example, to provide high-level differentiation of the algorithm with respect to any of its inputs. This is employed by Sundance and FEniCS [Logg et al. 2012] to compute the linearization of the terms in the equation. It has been further exploited to provide automated adjoint operators, and thereby adaptive error control, functional optimization, and stability analysis [Rognes and Logg 2013; Farrell et al. 2013; Funke and Farrell 2013; Farrell et al. 2014].

3. EXPLOITING COMPOSABLE ABSTRACTIONS IN FIREDRAKE

The novel contribution of Firedrake as a piece of mathematical software is to take the decomposition of the finite element method into automated abstractions further than previous approaches. In particular, we use a uniform abstraction (PyOP2) for the specification of iterations over the mesh, motivated by the observation that the mathematical statement of finite element problems decouples the local computation from its execution over the whole domain.

Firedrake models finite element problems as the composition of several abstract processes, and its software stack is composed of separate packages for each. The core Firedrake package composes these into a largely seamless abstraction for finite element problems. Figure 1 illustrates the Firedrake software stack, showing the relationships

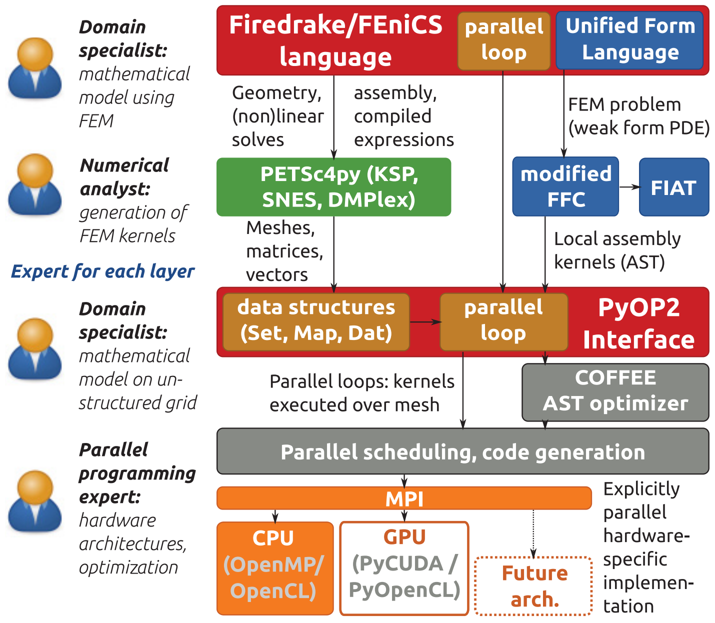
Domain specialist: mathematical model using FEM
Numerical analyst: generation of FEM kernels
Expert for each layer
Domain specialist: mathematical model on un-structured grid
Parallel programming expert: hardware architectures, optimization
Firedrake/FEniCS language
parallel loop
Unified Form Language
Geometry, (non)linear solves
assembly, compiled expressions
FEM problem (weak form PDE)
PETSc4py (KSP, SNES, DMPlex)
modified FFC
FIAT
Meshes, matrices, vectors
Local assembly kernels (AST)
data structures (Set, Map, Dat)
parallel loop
PyOP2 Interface
Parallel loops: kernels executed over mesh
COFFEE AST optimizer
Parallel scheduling, code generation
MPI
CPU (OpenMP/ OpenCL)
GPU (PyCUDA / PyOpenCL)
Future arch.
Explicitly parallel hardware-specific implemen-tation

Fig. 1. Abstractions composed to create the Firedrake toolchain and the separation of concerns that this creates. External tools adopted and/or modified from the FEniCS project are in blue, whereas the tools adopted from the PETSc project are in green. Interface layers are represented in red, and PyOP2 objects are brown. Our code generation and execution layer is represented in grey, and the underlying execution platform is shown in orange. For the reasons given later in Section 7.1, this article presents results for only the CPU backend.

between the various abstractions and software layers. These are described in more detail in the following sections.

An important benefit of well-designed mathematical abstractions is that they facilitate code reuse. Where possible, we have adopted and adapted existing abstractions, as well as existing implementations of those abstractions. This not only saves re-invention of previous work but also means that users and developers of those aspects of Firedrake do not need to learn new interfaces. However, in the case of the tasks of iteration over the mesh graph and the generation of optimal kernel implementations, there was no completely suitable existing solution and so new components were created.

3.1. Specification of Finite Element Problems: The FEniCS Language

The end user of Firedrake wants to specify and solve finite element problems. In some sense, the core part of this is the specification of the weak form of the PDE and the selection of the appropriate finite elements. UFL is a particularly elegant and powerful solution to this problem [Alnæs et al. 2014]. It is a purely symbolic language with well-defined, powerful, and mathematically consistent semantics embedded in Python. This makes interactive use possible and allows Firedrake to use the original implementation of UFL directly, thereby automatically maintaining compatibility with other users of the language. Firedrake adds several extensions to UFL, some of which have already been merged back into the upstream version.

The specification of the PDE and finite elements is necessary but not sufficient to specify a finite element problem. In addition to the weak form of the PDE, it is necessary to specify the mesh to be employed, set field values for initial and/or boundary conditions and forcing functions, and specify the sequence in which solves occur. UFL was developed as part of the FEniCS project, which provides a complete finite element

```
 1  V = FunctionSpace(mesh, "Lagrange", degree)
 2
 3  bc = DirichletBC(V, 0.0, [3, 4])  # Boundary condition for y=0, y=1
 4
 5  u = TrialFunction(V)
 6  v = TestFunction(V)
 7  f = Function(V).interpolate(Expression(
 8      "48*pi*pi*cos(4*pi*x[0])*sin(4*pi*x[1])*cos(4*pi*x[2])"))
 9  a = inner(grad(u), grad(v))*dx
10  L = f*v*dx
11
12  u = Function(V)
13  A = assemble(a, bcs=bc)
14  b = assemble(L)
15  bc.apply(b)
16  solve(A, u, b, solver_parameters={'ksp_type': 'cg',
17                                    'pc_type': 'hypre',
18                                    'pc_hypre_type': 'boomeramg',
19                                    'pc_hypre_boomeramg_strong_threshold':
                                          0.75,
20                                    'pc_hypre_boomeramg_agg_nl': 2,
21                                    'ksp_rtol': 1e-6,
22                                    'ksp_atol': 1e-15})
```

Listing 1. Firedrake code for the Poisson equation. Here, `mesh` and degree are assumed to have been defined previously. UFL functions and operations are defined in orange, whereas other FEniCS language constructs are defined in blue.

problem-solving environment in the form of the Python interface to DOLFIN [Logg et al. 2012b]. We refer to the language for finite element problems defined by DOLFIN and UFL as the FEniCS Language. To ensure compatibility, Firedrake implements (a close variant of) that language and presents a user interface that is identical in most respects to the DOLFIN Python interface. Firedrake implements various extensions to the language, and there are a few features of DOLFIN that are not supported.

A Poisson and linear wave equation finite element problem specified in the FEniCS Language for execution by Firedrake are shown in Listings 1 and 2 (see Section 6). Line 1 defines a finite element function space on a given mesh (whose definition is omitted for brevity) and degree using Lagrange elements. A Dirichlet boundary condition of value 0 on a region of the domain identified by the markers 3 and 4 is defined on line 3. Lines 5 through 10 show the UFL code defining the bilinear and linear forms $a = \nabla u \cdot \nabla v \, \mathrm{d}x$ and $L = f v \, \mathrm{d}x$ with test and trial functions u and v and forcing function f. The resemblance to the mathematical formulation is immediately apparent. In lines 13 through 15, the forms are assembled into a matrix `A` and Function `b` with the boundary conditions applied. The linear system of equations is solved in line 16 for a Function `u` defined on line 12.

3.2. Finite Element Tabulation: FIAT

Firedrake employs the FInite element Automatic Tabulator (FIAT) [Kirby 2004], which implements the classical finite element abstraction of Ciarlet [1978], to support a wide range of finite elements with relatively few element-specific alterations. The process of merging Firedrake's extensions to FIAT back into the original version is under way.

3.3. Iteration over the Mesh Graph: PyOP2

In a typical finite element problem, the operations whose cost in data movement or floating-point operations is proportional to the size of the mesh will be the dominant cost. These operations typically fall into two categories: iterating over data structures associated with the mesh, and sparse linear algebra. Firedrake's solution to the former class of operation is PyOP2 [Rathgeber et al. 2012; Markall et al. 2013].

PyOP2 is a DSL embedded in Python for the parallel execution of computational kernels on unstructured meshes or graphs. Fundamental concepts are shared with OP2 [Giles et al. 2011]; however, the implementation differs in ways that are crucial for the integration with Firedrake and other projects. PyOP2 dynamically generates code at runtime by leveraging Python to inspect objects and data structures. OP2 relies on static analysis of an input program that is transformed through source-to-source translation at compile time, making it very difficult to embed in another application. Furthermore, PyOP2 provides sparse matrices and other data structures required for finite element computations, which are not supported by OP2.

PyOP2 provides an abstract interface for the definition of operations composed of the application of a kernel function for each entry in a fixed arity graph. By representing the computational mesh as such a graph, it becomes possible to represent all of the mesh-visitor operations in the finite element method as instances of this single abstraction. A particularly clean separation of concerns is thereby achieved between the specification of the local kernel functions, in which the numerics of the method are encoded, and their efficient parallel execution. PyOP2 is the key novel abstraction in the Firedrake system. It is documented in much more detail in Section 4.

3.4. Unstructured Meshes: DMPlex

PyOP2 has no concept of the topological construction of a mesh: it works with indirection maps between sets of topological entities and sets of degrees of freedom (DOFs) but has no need to know the origin of these maps. Firedrake derives the required indirection maps for input meshes through an intermediate mesh topology object using PETSc's DMPlex API, a data management abstraction that represents unstructured mesh data as a directed acyclic graph [Knepley and Karpeev 2009; Balay et al. 2014]. This allows Firedrake to leverage the DMPlex partitioning and data migration interfaces to perform domain decomposition at runtime while supporting multiple mesh file formats. Moreover, Firedrake reorders mesh entities to ensure computational efficiency through communication-computation overlap while also employing mesh renumbering techniques provided by DMPlex to improve cache coherency within the resulting datasets [Lange et al. 2016].

3.5. Linear and Nonlinear Solvers: PETSc

As noted previously, the encapsulation of solvers for linear and nonlinear systems of equations is one of the most spectacular success stories for abstraction in scientific computing. The creation of efficient solver algorithms and implementations is also a complex and deep research field, which it is not profitable to attempt to reinvent. We therefore adopt the widespread practice of passing solver problems on to an established high-performance solver library. PETSc is adopted as a particularly well-established and fully featured library that provides access to a large range of its own and third-party implementations of solver algorithms [Balay et al. 2014]. The fully featured Python interface to PETSc [Dalcin et al. 2011] makes its integration with Firedrake particularly straightforward. Employing PETSc for both its solver library and for DMPlex has the additional advantage that the set of library dependencies required by Firedrake is kept small.

4. PYOP2

Many numerical algorithms and scientific computations on unstructured meshes can be viewed as the independent application of a local operation everywhere on the mesh. In the finite element method, this characterization applies most obviously to the assembly of integrals over the domain; however, it also applies to other operations, such as timestep increments and boundary condition implementation. This local operation is often called a *computational kernel*, and its independent application lends itself naturally to parallel computation.

4.1. Sets

A mesh is modeled in PyOP2 as a graph defined by sets of entities (e.g., vertices, edges, and cells) and maps between these sets. Sets are used to represent collections of topological entities: vertices, edges, faces, and cells. Sets are completely abstract entities: they store no bulk data themselves but only record the number of entities that they contain and their distribution among MPI processes. A set may also represent a set of nodes at which data may be stored; this set of nodes need not correspond to a set of topological entities. This facilitates the support of higher-order finite element spaces in which varying numbers of DOFs may be associated with various classes of topological entities. Sets exist only to be the subject of reference of other data objects, most particularly Maps and Dats.

4.2. Maps

A map associates a tuple of entries in a *target* set with each entry of another *source* set. For example, the source set might be the set of cells in a mesh, and the target set might be the set of DOFs of a finite element function space. The map could then record, for each cell, the tuple of DOFs of the target function space that are incident to that cell.

It is important to note that PyOP2 itself has no concept of meshes or function spaces. The semantic meanings of sets and maps are defined and understood only by the Firedrake layer. At the PyOP2 layer, these structures are merely objects over which iteration and indirection can occur.

There is a requirement for the map to be of *constant arity*—in other words, each element in the source set must be associated with a constant number of elements in the target set. The constant arity restriction causes the extent of many tight loop bounds to be fixed, which creates opportunities for vectorization and other optimizations. However, it excludes certain kinds of mappings. A map from vertices to incident edges or cells is only possible on a very regular mesh where the multiplicity of any vertex is constant. Nevertheless, the full set of maps required to implement the finite element method is supported.

4.3. Data

PyOP2 supports three core arrangements of mutable data: Dats, which are abstracted discretized vectors, Mats, which are sparse matrices, and Globals, which represent data not associated with individual set members. In other words, a Mat is equivalent to a bilinear operator over a pair of Sets, a Dat is equivalent to a linear operator over a Set, and a Global is a scalar (a 0-linear operator).

A Dat represents a vector of values, each associated with a particular member of the Set[1] over which that Dat is defined. The Dat presents a completely abstracted interface: the data may actually reside on one or more accelerators (GPUs) and be distributed over multiple MPI processes, but the user will not usually observe this. In particular, Dats are able to reason about the validity and location of their data so that copies to and from the GPU and halo exchanges over MPI happen automatically and only if required.

A Mat object represents a sparse matrix, a linear operator from the data space defined on one Set to that defined on another. The matrix interface is actually a fairly thin layer over PETSc (in the CPU case) or CUSP (in the NVIDIA GPU case), and

[1]There is actually a thin intermediate Dataset between the Set and Dat to parameterize the size of the data at each set element, but this is an implementation detail over which we will not dwell.

linear solves are completely outsourced to those libraries. At this stage, PETSc is the far more complete system and the only one considered production ready. The primary role of the Mat object is to match the sparse linear algebra library abstraction to the PyOP2 abstraction so that a PyOP2 kernel can be employed to assemble matrix entries efficiently and in parallel.

A Global represents a single tuple of values not connected with a Set. The reason for including this type, rather than simply employing a native Python numerical type, is to facilitate reasoning about updating data location. This enables the PyOP2 system to ensure that a Global always has the correct, consistent value even when updated in parallel or located on an accelerator.

4.4. Parloops and Kernels

PyOP2's model of execution is one of parallel loop operations that transform the system state, consisting of a set of Dats, Mats, and Globals. Each parallel loop operation executes a kernel function once for each member of a specified *iteration set*. In finite element computations, this set is usually the set of a particular class of topological entities, thereby allowing a stencil operation to be executed over the whole mesh. The function usually accesses each Dat argument f indirectly through a map m. In other words, when a kernel function k is called for iteration set entry e, the reference to the set of values given by $f(m(e))$ is passed to k. For a computation over cells where k requires data f defined over vertices, m provides the indices into f for each cell e.

For example, if e is the set of cells in a mesh, f is the set of DOF values of a discretized field, and m is the map from cells to the incident DOFs, then $f(m(e))$ will be a reference to the set of DOF values incident to e.

We term the application of a kernel to a particular set of data a *Parloop*. Specification of a Parloop requires a kernel function k, a set of iteration entities E, and data arguments $f_i(a_i, m_i)$, each of which is annotated with an access descriptor a_i and indirection map m_i. A Parloop created with arguments $(k, E, f_0(a_0, m_0), \ldots, f_n(a_n, m_n))$ encodes the mathematical algorithm

```
for all $e \in E$ do
    $k\Big( f_0(m_0(e)) \ldots, f_n(m_n(e))\Big)$,
```

where each element $m_i(e)$ of f_i is accessed according to the descriptor a_i as detailed in the next section.

The kernel only has access to those entries of the Dat arguments that are adjacent to the current iteration set entry under the map provided. It sees the local ordering of the Dat entries to which it has access but has no information about the global indices.

The loop over the iteration set E is explicitly unordered and parallel: PyOP2 is licensed to execute it in any order and using as many threads, vector lanes, or distributed processes as are available. Indirect access to data creates the possibility that this parallel execution may cause write contention—that is, the same piece of data is accessed via more than one entity in the iteration set. PyOP2 must reason to avoid these contentions using coloring, communication, and copies of data as appropriate. This is made possible by the specification of access descriptors for all kernel arguments.

The current coloring implementation in PyOP2 is deterministic, which results in bit-reproducible results when run on the same number of processors. Whether this feature remains sustainable as hardware parallelism becomes more fine grain is yet to be determined.

4.4.1. Access Descriptors. Kernel functions modify their data arguments in place. The critical observation in OP2, which is adopted in PyOP2, is that mesh-based simulation kernels modify their arguments in characteristic ways. By explicitly specifying the character of the operations that the kernel will perform on each Dat, automated reasoning about the parallel execution of the Parloop in the presence of indirectly accessed arguments becomes vastly easier. The available access descriptors are as follows:

READ. The kernel may use previous values of this argument but not set them.
WRITE. The kernel may set the values of this argument, but the kernel's behavior does not depend on the previous values of the argument.
RW. The kernel may set the values of this argument and may use the previous values of the argument. Note that this still does not imply a particular execution order over the iteration set.
INC. The kernel adds increments to the values of the argument using the equivalent of the += operator in C.

The reader will immediately observe that READ, WRITE, and INC are special cases of RW. However, their inclusion enables more sophisticated automated reasoning about data dependencies than would be possible were all arguments labeled RW.

Any data accessed as READ, RW, or INC is automatically gathered via the mapping relationship in a *staging in* phase, and the kernel is passed pointers to local data. After the kernel has been invoked, any data accessed as WRITE, RW, or INC is scattered back out in a *staging out* phase. Only data accessed in INC mode could potentially cause conflicting writes and requires thread coloring to prevent any contention.

4.4.2. Global Arguments. Global reductions are important operations in mesh-based simulations. Users may wish to calculate globally integrated quantities, such as energy, or execute other reductions, such as calculating the maximum Courant number in the simulation domain. Global data does not have an indirection map relationship with the mesh: the same global value is visible from every mesh entity. The kernel is therefore passed an appropriately sized variable into which it can place its contribution to the reduction operation. Globals have their own set of permitted access descriptors that reflect this: READ, SUM, MIN, MAX. PyOP2 is free to create multiple variables in memory corresponding to a single Global to support parallel kernel execution. The access descriptor enables PyOP2 to subsequently reduce these multiple variables to a single value. The addition of further reduction access descriptor operations, or even allowing user-specified reductions, would be straightforward. However, at the time of this writing, there does not appear to be user demand for this feature.

4.4.3. Matrix Arguments. Mat arguments differ from Dat and Global arguments in several important ways. Critically, from PyOP2's perspective, Mats are write-only data structures. Operations that read matrices, such as matrix-vector multiply and solving linear systems, are executed by the sparse matrix library (for CPU execution, this is PETSc). Consequently, the only access descriptors permitted for Mats are WRITE and INC. A Mat represents a linear relationship between two sets, corresponding to the rows and the columns of the matrix, so two maps (which may be identical) are required to map the kernel contribution to the matrix. In terms that may be more familiar to the reader conversant with the finite element method, the kernel is responsible for the local assembly of the integral of a test function against a trial function, and PyOP2 then uses the Maps to execute the global assembly into a sparse matrix.

4.5. Kernel Optimization in COFFEE

Kernels are initialized 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 optimization through the COFFEE AST optimizer [Luporini et al. 2015], a compiler

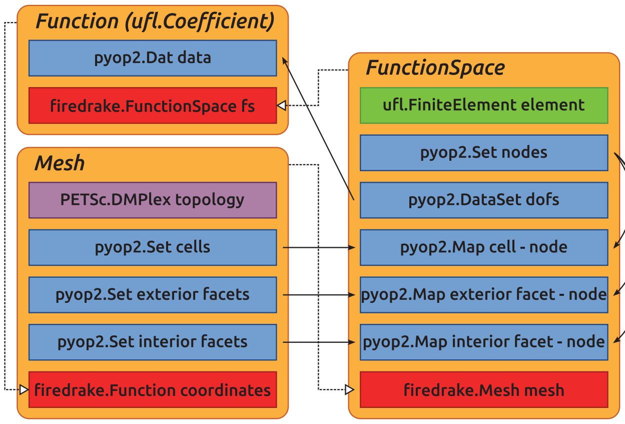
Function (ufl.Coefficient)
pyop2.Dat data
firedrake.FunctionSpace fs
Mesh
PETSc.DMPlex topology
pyop2.Set cells
pyop2.Set exterior facets
pyop2.Set interior facets
firedrake.Function coordinates
FunctionSpace
ufl.FiniteElement element
pyop2.Set nodes
pyop2.DataSet dofs
pyop2.Map cell - node
pyop2.Map exterior facet - node
pyop2.Map interior facet - node
firedrake.Mesh mesh

Fig. 2. PyOP2 and PETSc objects of which key Firedrake objects are composed. PyOP2 objects are shown in blue, references to other Firedrake objects are red, and PETSc objects are purple.

that specializes in advanced optimizations for short loops enclosing nontrivial mathematical expressions of the kind that typify finite element local assembly kernels.

COFFEE performs platform-specific optimizations on the AST with the goals of minimizing the number of floating-point operations and improving instruction-level parallelism through the use of single instruction, multiple data (SIMD) vectorization. The optimizer can detect invariant subexpressions and hoist them out of the loop nest, permute and unroll loop nests, and vectorize expressions. The last step may require padding of the data and enforcing alignment constraints to match the target SIMD architecture. COFFEE supports both Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) instruction sets.

5. THE FIREDRAKE LAYER

The role of the Firedrake layer is to marshal the abstractions provided by UFL, FIAT, FFC, PETSc, and PyOP2 to take finite element problems specified in the FEniCS Language and efficiently produce solutions.

5.1. Mapping Finite Element Constructs to Data Abstractions

The FEniCS Language presents higher-level mathematical objects than PyOP2. Firedrake implements these by composing suitable combinations of PyOP2 and PETSc objects. Figure 2 illustrates this relationship. The Firedrake implementation of operations in the FEniCS Language consists primarily of selecting the relevant PyOP2 objects and composing corresponding parallel loop calls so that the PyOP2 layer can undertake the actual calculation.

5.1.1. Mesh Abstraction. The primary functions of the mesh object are to record adjacency between the topological entities (vertices, edges, faces, facets, and cells) of the mesh and to record the mesh geometry. The former of these is encoded in a PETSc DMPlex that provides arbitrary adjacency relationships [Knepley and Karpeev 2009].

A common approach in PDE toolkits is to treat the coordinates as a special class of data by, for example, storing the coordinates of each vertex in the mesh. Firedrake eschews this approach in favor of treating the coordinates as a first-class vector-valued field represented in a suitable vector-valued function space. An advantage of this approach is that any operation that can be applied to a field may be applied to the coordinates. Solving a finite element problem to determine a new geometry field is therefore straightforward. Representing the coordinates using a fully featured function space also presents a mechanism for supporting curved (isoparametric) elements, support for which is currently available in a branch of Firedrake.

The mesh contains PyOP2 Sets that are proxies for the sets of cells, interior and exterior facets[2] of the mesh. These form the iteration spaces for the PyOP2 parallel loops and correspondingly are the "source" sets of Maps that encode function spaces.

5.1.2. Function Spaces and Functions. A central and distinct feature of the finite element method is its representation of all solution fields as weighted sums of basis functions. The FEniCS Language supports this construction with FunctionSpace and Function objects. The Function objects store the coefficient values and a reference to the corresponding FunctionSpace, whereas the FunctionSpace stores all of the indirections from the Mesh to the DOFs, and the symbolic information required to access the basis functions. As Figure 2 demonstrates, this maps in a rather natural way onto PyOP2 data types.

The Function object holds a PyOP2 Dat. This reflects the separation of concerns in the Firedrake toolchain: the Firedrake layer reasons about the finite element method, and all of the actual data storage and communication is delegated to PyOP2.

FunctionSpace objects contain PyOP2 Map objects that encode the indirections from mesh entities to the collections of DOFs required to implement the finite element method. The cell-node indirection map provides the indices of the nodes incident to each cell. Equivalently, this is the set of nodes whose associated basis functions may be nonzero in that cell. The use of the term *node* here rather than *DOF* reflects the treatment of vector and tensor function spaces: the same indirection map is employed regardless of the number of DOFs at each mesh location, and the indices of the DOFs are calculated from this.

5.1.3. Assembling Forms. Solving a variational problem requires the assembly of a linear system of equations in the linear, and the Jacobian and residual form in the nonlinear case. In the Poisson problem in Listing 1, the bilinear and linear forms `a` and `L` are explicitly assembled into the sparse matrix `A` and vector `b`, respectively. Firedrake hands off assembly computations to PyOP2 Parloops (Section 4.4) with a form-specific list of arguments constructed as follows. The local assembly kernels for the forms are generated by FFC as described in the following section. The iteration set is extracted from the FunctionSpace of the test function `v`, and the first Parloop argument is the output tensor. For the bilinear form, this is a Mat built from a pair of maps extracted from test and trial space, for the linear form a Dat obtained by creating a new Function on the test space. The second Parloop argument is the coordinate field. Each coefficient used in the form, such as `f` in Listing 1, translates into an additional argument.

5.2. A Modified FFC

The FEniCS project provides FFC, which takes variational forms specified in UFL and generates optimized C++ kernel functions conforming to the UFC interface [Logg et al. 2012a]. This approach cuts across the abstraction provided by the PyOP2 interface: in PyOP2, the specification of the kernel is a problem-specific question delegated to the user (in this case, the PyOP2 user is Firedrake). Conversely, the optimization of the kernel body for a given hardware platform is a matter for which PyOP2 (specifically COFFEE) takes responsibility. To reflect this, the version of FFC employed in the Firedrake toolchain is substantially modified. It still accepts UFL input but produces an unoptimized (and indeed unscheduled) AST for the local assembly of the form. Firedrake employs this AST to create a PyOP2 kernel and executes a PyOP2 parallel

[2]A facet is a mesh entity of codimension 1: an edge of a 2D mesh or a face of a 3D mesh.

loop to perform global assembly. The modifications required to FFC are such that the Firedrake version of FFC is effectively a fork and will not be merged back. However, it is hoped that the UFLACS Form Compiler, currently under development by Martin Alnæs, will provide a basis for a unified compiler infrastructure.

5.3. Escaping the Abstraction

It is an inherent feature of software abstractions that they create a division between those algorithms that are expressible in the abstraction and those that are not. In a well-designed abstraction, the former are concise, expressive, and computationally efficient. However, any part of an algorithm not expressible within the abstraction may become impossible to express without completely breaking out of the abstract framework and coding at a much lower level. It will never be possible to represent all algorithms with the same level of elegance in a single abstraction. Instead, the challenge is to ensure that a graceful degradation of abstraction occurs. In other words, operations that lie a little outside the abstraction should require the user to work at only a slightly lower level, and access to aspects of the abstraction that are still applicable should be preserved.

5.3.1. Custom Kernels. The FEniCS Language presents an elegant and powerful abstraction for the expression of the core of the finite element method: weak form PDEs and their solution on piecewise polynomial triangulations of domains. However, it is frequently the case that real simulation challenges also incorporate non–finite element aspects. For example, discontinuous Galerkin discretizations may require shock detectors and slope limiters, parameterizations of unresolved phenomena may require complex pointwise operations, and initial conditions may require access to external data in ways not representable in UFL.

The critical observation is that these operations, and many others, are still characterized by visiting mesh entities and accessing only data local to them: the operations supported by PyOP2. Firedrake therefore presents the user with the option of specifying a custom kernel in either C or as an AST. This kernel can then be explicitly executed over the mesh by invoking a parallel loop. If, as is often the case, the data access patterns are equivalent to those of the finite element method, then the user can invoke the Firedrake wrapper of a parallel loop and let Firedrake extract the correct Maps and Dats from the Firedrake Function. Alternatively, the user may directly invoke a PyOP2 parallel loop and extract the PyOP2 data structures manually. In either case, the automated parallelization provided by PyOP2 remains. In Section 6, Listings 3 and 4 show an example of a randomized initial condition specified with custom Firedrake and PyOP2 kernels, respectively.

5.3.2. Direct Access to Data Structures. At a more direct level, the user may also elect to directly access the data in the Firedrake data structures. Since Firedrake is a pure Python library, the user can then deploy the full armory of Python, NumPy, and compatible libraries. PyOP2 employs the introspection capabilities of Python so that even in this case it remains aware of the data that has been accessed and modified. PyOP2 ensures that copies and halo exchanges occur as necessary to make sure that the user's view of the data is current and correct, and that algorithmic correctness is maintained.

5.3.3. Access to Generated Code. For debugging purposes, it is sometimes useful for the user to access the C code that PyOP2 generates. This is accessible both in the disk cache and in memory attached to the relevant PyOP2 parallel loop object. In the particular case of C code that fails to compile (most commonly due to a syntax error in user-provided custom kernel code), the error message provides the location of the generated source file and the compiler error log.

5.4. Additional Features Facilitated by the Firedrake Abstraction

5.4.1. Factorization of Mixed Function Spaces. When solving PDEs with multiple unknown solution fields, the standard finite element approach is to seek a solution in the mixed function space given by concatenating the function spaces of the solution fields. The test function naturally is drawn from the same space. UFL represents a form defined over a mixed function space as a single form. FFC then constructs a single mixed kernel that iterates over the combined set of basis function of the test space and (in the case of a bilinear form) the trial space. In DOLFIN, this is then assembled into a single monolithic sparse matrix.

In contrast, Firedrake takes the UFL form, represented as an AST, and employs symbolic manipulation to split it into forms for each combination of constituent test and trial space. This results in separate forms for each block of the mixed system, and FFC then creates kernels for those individual blocks. The resulting kernels have simpler loop structures, which aids COFFEE in producing highly optimized implementations. Bilinear forms are then assembled into a hierarchical matrix structure, comprising a matrix for each block combined using PETSc's nested matrix facility (see p. 86 of Balay et al. [2014]). Using PETSc's compressed sparse row storage, the insertion of entries into submatrices is expected to be faster than into a monolithic matrix due to the smaller number of nonzero columns (which have to be searched) in each row. This furthermore enables more efficient exploitation of block solver techniques such as Schur complements. A simulation employing mixed function spaces is presented in Section 6.4. A much more detailed exposition of the mixed form splitting algorithm is presented in Section 5.2.3 of Rathgeber [2014].

5.4.2. Pointwise Operations. Users often need to change the values of fields by means other than solving a variational problem. For example, when employing a Runge-Kutta timestepping scheme, variational problems are solved for the updates to fields, but the actual updates are linear combinations of fields. Similarly, users frequently choose to calculate forcing functions pointwise in terms of other functions or may rescale the coordinate field. All of these are achievable by writing custom kernels; however, they are expressed much more naturally by writing assignments of expressions in which the variables are Function objects. These expressions are then compiled to form a kernel function that is applied pointwise over the mesh. The explicit wave equation code shown in Listing 2 illustrates the simplicity of the user code required. The increments for p and ψ both employ the pointwise expression compiler.

5.4.3. Immersed Manifolds and Extruded Meshes. The support for domains that are manifolds immersed in higher-dimensional spaces introduced in Rognes et al. [2013] extends directly to Firedrake. Furthermore, Firedrake has extended the algebraic representation of finite elements and basis functions in UFL, FFC, and FIAT to enable the algorithmic creation of tensor product finite elements on quadrilateral, triangular prism, and hexahedral cells [McRae et al. 2016a]. A particularly important class of meshes in high aspect ratio domains, such as the atmosphere and ocean, is composed of layers of triangular prism or hexahedral cells aligned in the vertical direction. The PyOP2 abstraction has been extended to exploit the structure induced by this vertical alignment to create very high speed iteration over such "extruded" meshes documented in Bercea et al. [2016].

6. EXPERIMENTS

Firedrake is a tool chain capable of solving a wide range of finite element problems, which is demonstrated in this section through experiments chosen to cover different characteristics of its implementation. These include assembling and solving a stationary Poisson problem, the nonlinear time-dependent Cahn-Hilliard equation and the linear wave equation using an explicit timestepping scheme. Implementation aspects investigated are assembly of left- and right-hand sides for regular and mixed forms, solving linear and nonlinear systems, evaluating expressions, and using field-split preconditioners. All benchmarks represent real-world applications used in fluid dynamics to model diffusion, phase separation of binary fluids, and wave propagation.

The principal contribution of this article is to describe the composition of abstractions and consequent separation of concerns achieved in Firedrake. A comprehensive performance evaluation is beyond its scope, and indeed a comprehensive performance evaluation of a single problem might easily occupy an entire article. Instead, this section is designed to enable the reader to develop an impression of the broad performance characteristics of Firedrake.

We have chosen to compare against DOLFIN for two reasons. The first is that it is the package that provides the closest analogue to Firedrake—many of the same test cases can be run from nearly the same code. The second reason goes to the heart of the difficulty of conducting fair performance comparisons. By using someone else's code, it is difficult to avoid the risk that any performance deficiency is due to inexpert use rather than an inherent flaw. By employing DOLFIN on ARCHER using the compilation flags recommended by the DOLFIN developers and using test cases based on DOLFIN examples, we minimize the chance that any performance deficiencies are due to incorrect use of the software.

Source code for all benchmarks and the scripts used to drive them are available as part of the firedrake-bench repository hosted on GitHub. The particular version used in these experiments has been archived on Zenodo [Rathgeber and Mitchell 2016].

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 2.7GHz, 12-core E5-2697 v2 (Ivy Bridge) series Intel Xeon processors linked via a QuickPath Interconnect (QPI) and 64GB of 1,833MHz DDR3 memory accessed via eight 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. For the reasons given later in Section 7.1, execution is always one core per MPI process: OpenMP is not employed.

Firedrake and PETSc were compiled with version 4.9.2 of the GNU compilers[3] and Cray MPICH2 7.1.1 with the asynchronous progress feature enabled was used for parallel runs. The Firedrake component revisions used are archived on Zenodo and are accessible via the DOIs in the relevant citation: Firedrake [Mitchell et al. 2016], PyOP2 [Rathgeber et al. 2016], FIAT [McRae et al. 2016b], COFFEE [Luporini et al. 2016], ffc [Logg et al. 2016], PETSc [Smith et al. 2016], and PETSc4py [Dalcin et al. 2016]. The DOLFIN used as a comparator is revision 5ec6384 (July 12, 2015) and is linked to the same PETSc version as Firedrake.

Generated code is compiled with `-O3 -fno-tree-vectorize` in the Firedrake and `-O3 -ffast-math -march=native` (as suggested by the FEniCS developers) in the DOLFIN case.

Unless otherwise noted, DOLFIN is configured to use quadrature representation with full FFC optimizations and compiler optimizations enabled, and Firedrake makes use of COFFEE's loop-invariant code motion, alignment, and padding optimizations described in Luporini et al. [2015] using quadrature representation. Meshes are

[3]Due to technical limitations in accessing the license server, Intel and Cray compilers cannot be used on ARCHER compute nodes and are therefore unavailable to PyOP2's just-in-time compilation system.

reordered using PETSc's implementation of reverse Cuthill-McKee in the Firedrake case and DOLFIN's mesh reordering, 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 prepopulate the caches for kernels and generated code to ensure that 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 PDE. A primal Poisson problem for a domain $\Omega \in \mathbb{R}^n$ with boundary $\partial\Omega = \Gamma_D \cup \Gamma_N$ is defined as follows:

$$-\nabla^2 u = f \quad \text{in } \Omega, \tag{1}$$

$$u = 0 \quad \text{on } \Gamma_D, \tag{2}$$

$$\nabla u \cdot n = 0 \quad \text{on } \Gamma_N. \tag{3}$$

The weak formulation reads: find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x \quad \forall v \in V, \tag{4}$$

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 domain Ω is chosen to be the unit cube $[0, 1]^3$, represented as a fully unstructured mesh. The source term f is

$$f(x, y, z) = 48\pi^2 \cos(4\pi x)\sin(4\pi y)\cos(4\pi z) \tag{5}$$

with known analytical solution

$$u(x, y, z) = \cos(4\pi x)\sin(4\pi y)\cos(4\pi z). \tag{6}$$

Since the operator is symmetric positive definite, the problem is solved using a CG solver [Hestenes and Stiefel 1952] 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 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 1,536 cores are shown in Figure 3 for problems of approximately 0.5M to 14M DOFs for first and third order, respectively.

Parallel efficiency for the strong scaling results with respect to a full node (24 cores) is shown in Figure 4.

Weak scaling runtimes and efficiencies for P3 basis functions are shown in Figure 5 separately for the intranode case for up to 24 cores and the internode case for 24 to 1,536 cores. Within a node, processes share resources, particularly 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.

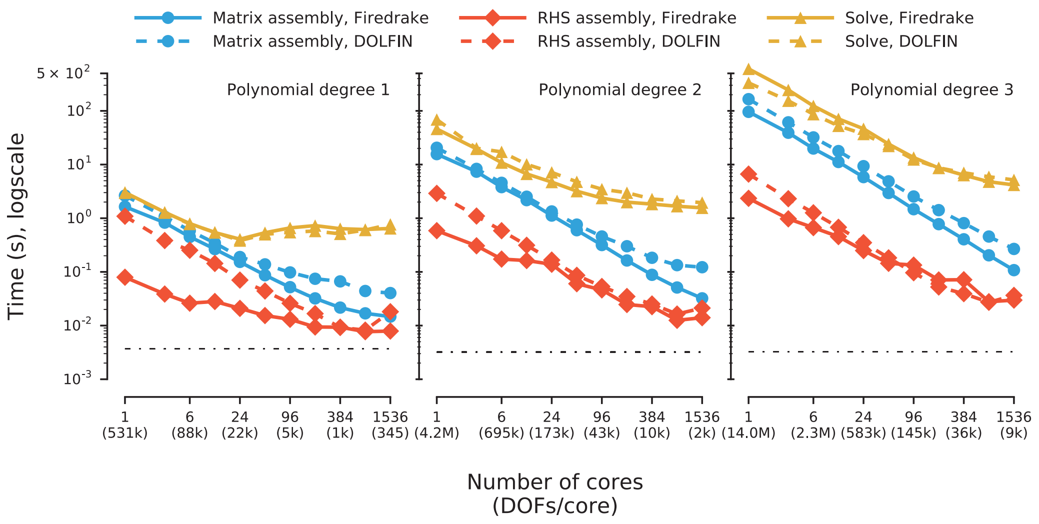

Fig. 3. Poisson strong scaling for degree one (left), two (center), and three (right) basis functions. Overhead for right-hand side assembly is indicated by the horizontal dash-dotted line. Note that times are logscale. Solve time clearly dominates in all cases, particularly for higher order and in the strong scaling limit, where the scaling flattens out at around 10k DOFs per core. Firedrake is faster at assembling left- and right-hand sides in almost all cases, demonstrating the efficiency of low overhead assembly kernel execution through PyOP2. Matrix assembly is considerably faster in the strong scaling limit in particular for low order, which can be attributed to Firedrake's way of enforcing strong boundary conditions described in Section 5.5 of Rathgeber [2014]. Right-hand side assembly has a considerably faster sequential base line for Firedrake such that it is affected by nonparallelizable overheads in the strong scaling limit sooner than DOLFIN. The sequential overhead indicated for Firedrake in this figure causes the scaling to flatten out much earlier than for matrix assembly. The time spent on right-hand side assembly, however, is negligible such that the overall runtime is not greatly affected.

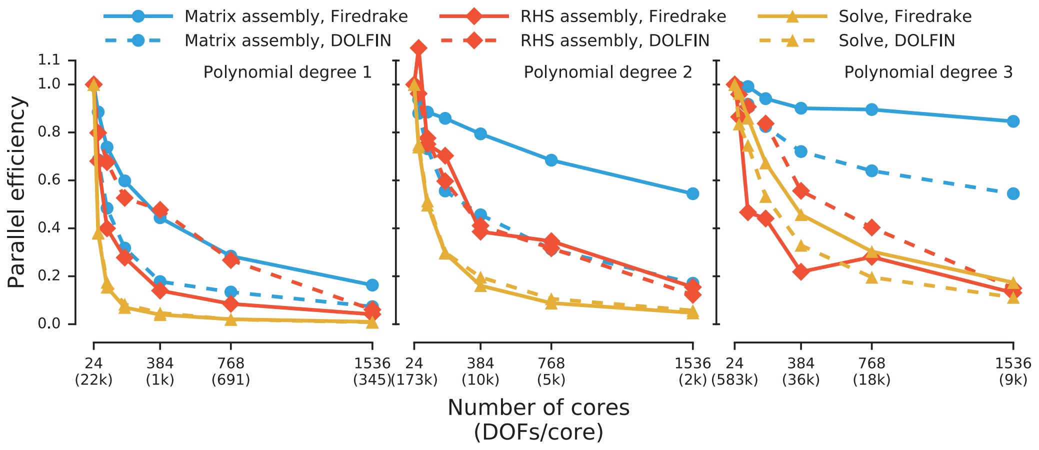

Fig. 4. Poisson strong scaling efficiency with respect to a full node (24 cores) on up to 1,536 cores for degree one, two, and three basis functions (left to right). The Firedrake matrix assembly shows the highest efficiency across the board, whereas the right-hand side assembly tails off compared to DOLFIN due to the faster baseline performance. Solver efficiencies are almost identical, with a slight advantage for Firedrake at third order.

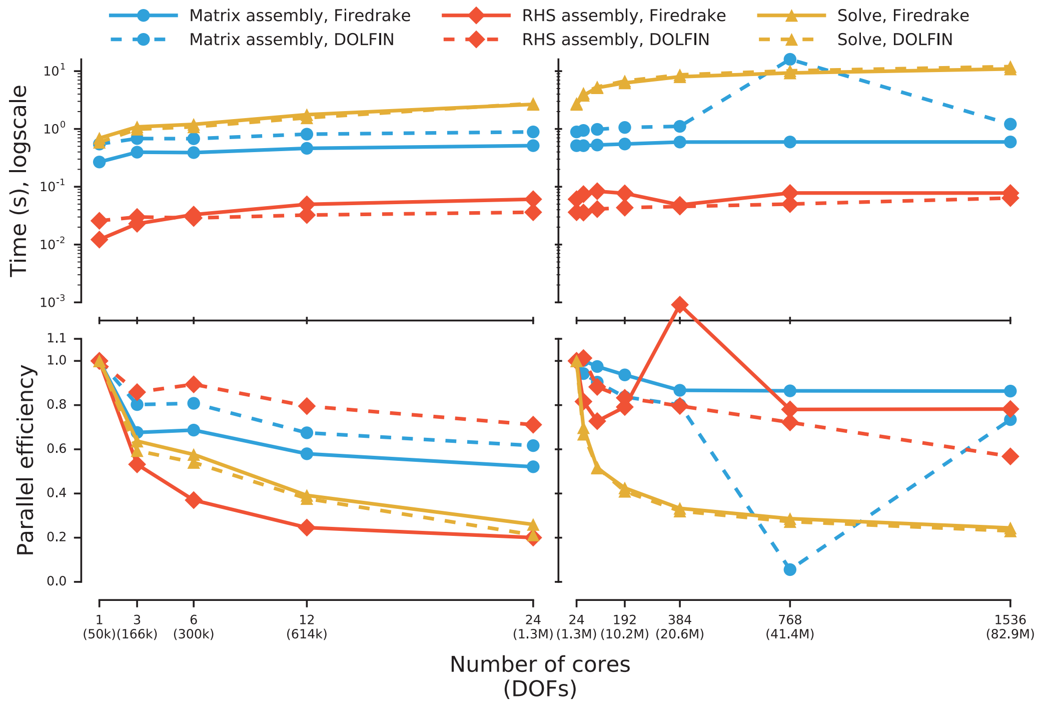

Fig. 5. Weak scaling performance for third-order Poisson basis functions with 50k DOFs per core. Scaling is shown intranode (1 to 24 cores) relative to a single core (left) and internode (24 to 1,536 cores) relative to a single node (right). Within a node, DOLFIN shows better efficiency for assembly due to Firedrake's faster sequential baseline. In particular, Firedrake right-hand side assembly drops off significantly from one to three and three to six cores due to resource contention, leading to DOLFIN overtaking from six cores. Beyond one node, Firedrake shows better assembly efficiency, although DOLFIN remains faster overall for right-hand side assembly. Solver runtimes and efficiencies are almost identical both intra- and internode.

6.3. Linear Wave Equation

The strong form of the wave equation, a linear second-order PDE, is given as

$$\frac{\partial^2\phi}{\partial t^2} - \nabla^2\phi = 0, \tag{7}$$

$$\nabla\phi\cdot n = 0 \text{ on } \Gamma_N, \tag{8}$$

$$\phi = \frac{1}{10\pi}\cos(10\pi t) \text{ on } \Gamma_D. \tag{9}$$

To facilitate an explicit timestepping scheme, an auxiliary quantity p is introduced:

$$\frac{\partial\phi}{\partial t} = -p, \tag{10}$$

$$\frac{\partial p}{\partial t} + \nabla^2\phi = 0, \tag{11}$$

$$\nabla\phi\cdot n = 0 \text{ on } \Gamma_N, \tag{12}$$

$$p = \sin(10\pi t) \text{ on } \Gamma_D. \tag{13}$$

```
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
phi_update = dt / 2 * p
p_constant = dt / assemble(v*dx)
p_form = inner(grad(v), grad(phi))*dx
while t <= T:
    p_in.assign(sin(2*pi*5*t))
    phi -= phi_update
    p += assemble(p_form) * p_constant
    bc.apply(p)
    phi -= phi_update
    t += dt
```

Listing 2. Firedrake code for the linear wave equation. The constant factor for the ϕ update and the form and the constant factor $v\mathrm{d}x$ for the p update are precomputed such that only $\nabla\phi \cdot \nabla v \ \mathrm{d}x$ is assembled at each timestep. The expressions `phi_update` and `p_constant` are purely symbolic and are used by Firedrake to generate and execute pointwise update calculations when the `assign`, $+=$, and $-=$ operations are called. `assemble(v*dx)` is the lumped mass, an integral that is calculated outside the form and then symbolically substituted into the pointwise update of `p`. `p_form` is similarly a symbolic integral that is numerically calculated by the `assemble` call in the `p` update. This means that the `p` update amounts to assembling the right-hand side of (14) and then using this to approximately solve (11) by scaling with the timestep and multiplying (DOF by DOF) with the inverse lumped mass matrix.

The weak form of (11) is formed as follows: find $p \in V$ such that

$$\int_\Omega \frac{\partial p}{\partial t} v \,\mathrm{d}x = \int_\Omega \nabla\phi \cdot \nabla v \,\mathrm{d}x \quad \forall v \in V \tag{14}$$

for a suitable function space V. The absence of spatial derivatives in (10) makes the weak form of this equation equivalent to the strong form so that it can be solved pointwise.

An explicit symplectic method is used in time, where p and ϕ are offset by a half timestep. Timestepping ϕ in (10) is a pointwise operation, whereas stepping forward p in (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 pointwise multiplication by the inverse of the lumped mass.

This benchmark demonstrates a numerical scheme 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 2.

6.3.1. Results. Strong scaling performance is shown in Figure 6 for up to 384 cores and is limited by the measured nonparallelizable overhead indicated by the horizontal lines in the graph. Weak scaling runtimes and efficiencies are shown in Figure 7 separately for the intranode case for up to 24 cores and the internode case for 24 to 384 cores.

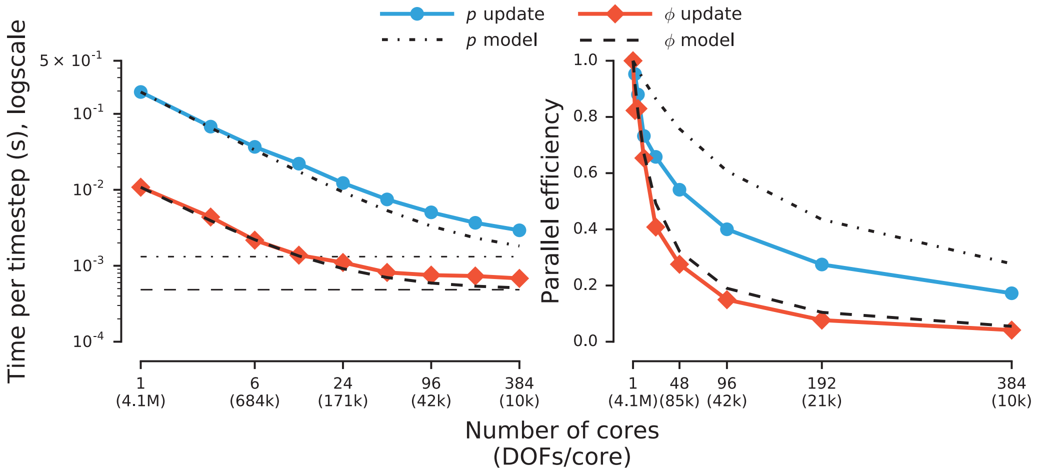

Fig. 6. Strong scaling (left) and parallel efficiency (right) for the p and ϕ updates of the explicit wave equation shown in Listing 2. Horizontal dashed (dotted) lines show the nonparallelizable overhead for the ϕ (p) updates. Given these overheads, models for expected runtime are shown for both updates. The ϕ update is a very simple expression executed as a direct loop and follows the projected scaling curve (dashed) based on the sequential runtime 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 already starting at three cores. Caching of the assembled expressions in the expression compiler keeps the sequential overheads low.

6.4. Cahn-Hilliard Equation

The final experiment presented in this section is the fourth-order parabolic time-dependent nonlinear Cahn-Hilliard equation, based on a DOLFIN demo,[4] 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:

$$\frac{\partial c}{\partial t} - \nabla \cdot \left(M \nabla \left(\frac{df}{dc} - \lambda \nabla^2 c \right) \right) = 0 \quad \text{in } \Omega, \tag{15}$$

$$\nabla \left(\frac{df}{dc} - \lambda \nabla^2 c \right) \cdot n = 0 \quad \text{on } \partial\Omega, \tag{16}$$

$$\nabla c \cdot n = 0 \quad \text{on } \partial\Omega, \tag{17}$$

with c the unknown fluid concentration, f a nonconvex 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 restated as two coupled second-order equations:

$$\frac{\partial c}{\partial t} - \nabla \cdot M \nabla \mu = 0 \quad \text{in } \Omega, \tag{18}$$

$$\mu - \frac{df}{dc} + \lambda \nabla^2 c = 0 \quad \text{in } \Omega. \tag{19}$$

[4]http://fenicsproject.org/documentation/dolfin/1.6.0/python/demo/documented/cahn-hilliard/python/documentation.html.

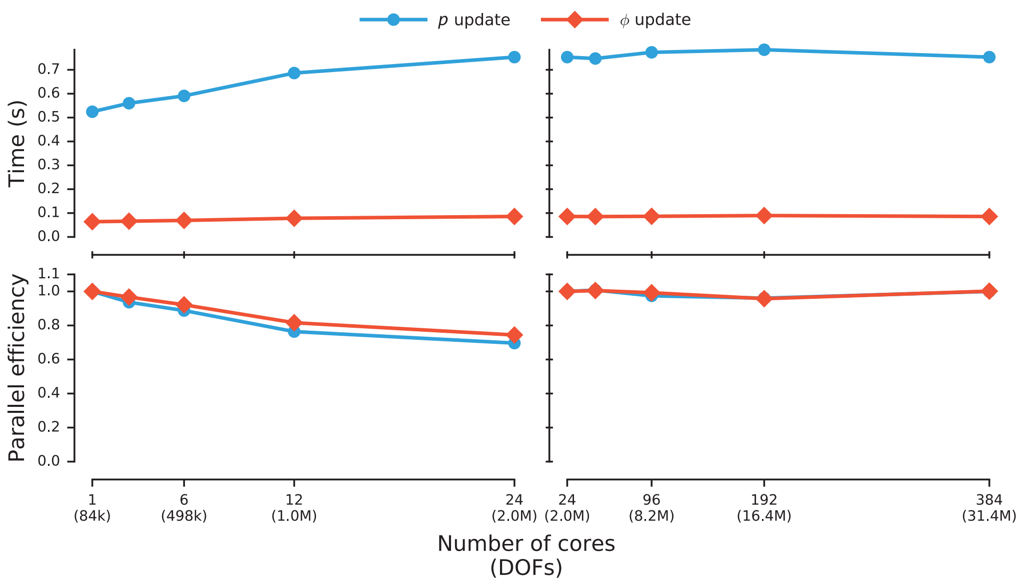

Fig. 7. Weak scaling performance for the explicit wave equation with 84k DOFs per core. Scaling is shown intranode (1 to 24 cores) relative to a single core (left) and internode (24 to 384 cores) relative to a single node (right). Timings are for 100 timesteps. The ϕ and p updates show a similarly high level of efficiency intranode, only dropping to about 80% for 24 cores. Across nodes, scaling is almost perfect for both ϕ and p updates.

The time-dependent variational form of the problem with unknown fields c and μ is given as follows: find $(c, \mu) \in V \times V$ for a suitable function space V such that

$$\int_\Omega \frac{\partial c}{\partial t} q \, \mathrm{d}x + \int_\Omega M \nabla \mu \cdot \nabla q \, \mathrm{d}x = 0 \quad \forall\, q \in V, \tag{20}$$

$$\int_\Omega \mu v \, \mathrm{d}x - \int_\Omega \frac{df}{dc} v \, \mathrm{d}x - \int_\Omega \lambda \nabla c \cdot \nabla v \, \mathrm{d}x = 0 \quad \forall\, v \in V. \tag{21}$$

Applying the Crank-Nicolson scheme for time discretization yields

$$\int_\Omega \frac{c_{n+1} - c_n}{dt} q \, \mathrm{d}x + \int_\Omega M \nabla \frac{1}{2}(\mu_{n+1} + \mu_n) \cdot \nabla q \, \mathrm{d}x = 0 \quad \forall\, q \in V, \tag{22}$$

$$\int_\Omega \mu_{n+1} v \, \mathrm{d}x - \int_\Omega \frac{df_{n+1}}{dc} v \, \mathrm{d}x - \int_\Omega \lambda \nabla c_{n+1} \cdot \nabla v \, \mathrm{d}x = 0 \quad \forall\, v \in V. \tag{23}$$

6.4.1. Problem Setup. The problem is solved on the unit square, represented as a fully unstructured mesh, 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 is shown as Listing 3. For comparison, an equivalent Kernel using the lower-level PyOP2 interface is provided in Listing 4.

To solve the mixed system, a GMRES solver with a fieldsplit preconditioner using a lower Schur complement factorization 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

$$S = D - CA^{-1}B, \tag{24}$$

```
# Setup code setting the random seed depending on MPI rank (executed once)
setup_code = """int __rank;
MPI_Comm_rank(MPI_COMM_WORLD, &__rank);
srandom(2 + __rank);"""
# Expression setting the random initial condition
rand_init = "A[0] = 0.63 + 0.02*(0.5 - (double)random()/RAND_MAX);"
par_loop(kernel=rand_init, measure=direct, args={'A': (u[0], WRITE)},
         headers=["#include <stdlib.h>", "#include <mpi.h>"],
         user_code=setup_code)
```

Listing 3. Code for a custom Firedrake kernel setting the random initial condition for the fluid concentration c in the Cahn-Hilliard example. Inclusion of extra headers to use library functions and an extra piece of setup code to be executed only once are not usually required for custom kernels.

```
# Setup code setting the random seed depending on MPI rank (executed once)
setup_code = """int __rank;
MPI_Comm_rank(MPI_COMM_WORLD, &__rank);
srandom(2 + __rank);"""
# PyOP2 C kernel string setting the random initial condition
rand_init = """void u_init(double A[1]) {
  A[0] = 0.63 + 0.02*(0.5 - (double)random()/RAND_MAX);
}"""
u_init = pyop2.Kernel(code=rand_init, name="u_init",
                      headers=["#include <stdlib.h>", "#include <mpi.h>"],
                      user_code=setup_code)
pyop2.par_loop(kernel=u_init, it_space=u.function_space().node_set[0],
               u.dat[0](op2.WRITE))
```

Listing 4. Code for a custom PyOP2 kernel equivalent to the Firedrake kernel in Listing 3.

and the lower factorization is an approximation to

$$\begin{pmatrix} A & 0 \\ C & S \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -CA^{-1} & I \end{pmatrix}, \tag{25}$$

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 Boomer-amg algebraic multigrid preconditioner [Falgout et al. 2006]. The inverse Schur complement, S^{-1}, is approximated by

$$S^{-1} \approx \hat{S}^{-1} = H^{-1} M H^{-1}, \tag{26}$$

using a custom PETSc `mat` preconditioner, where H and M are defined as

$$H = \sqrt{a}\langle u, v\rangle + \sqrt{c}\langle \nabla u, \nabla v\rangle \quad \forall v \in V \times V \tag{27}$$

$$M = \langle u, v\rangle \quad \forall v \in V \times V \tag{28}$$

with $a = 1$ and $c = \frac{\mathrm{dt}*\lambda}{1+100\mathrm{dt}}$ [Bosch et al. 2014].

6.4.2. Results. Strong scaling runtimes for up to 1,536 cores comparing Firedrake and DOLFIN for solving the nonlinear system, assembling the residual and Jacobian forms, and evaluating the initial condition on an 8M DOF mesh for 10 timesteps are shown in Figure 8. Weak scaling runtimes and parallel efficiencies are shown separately for 1 to 24 cores intranode and 24 to 1,536 cores internode in Figure 9.

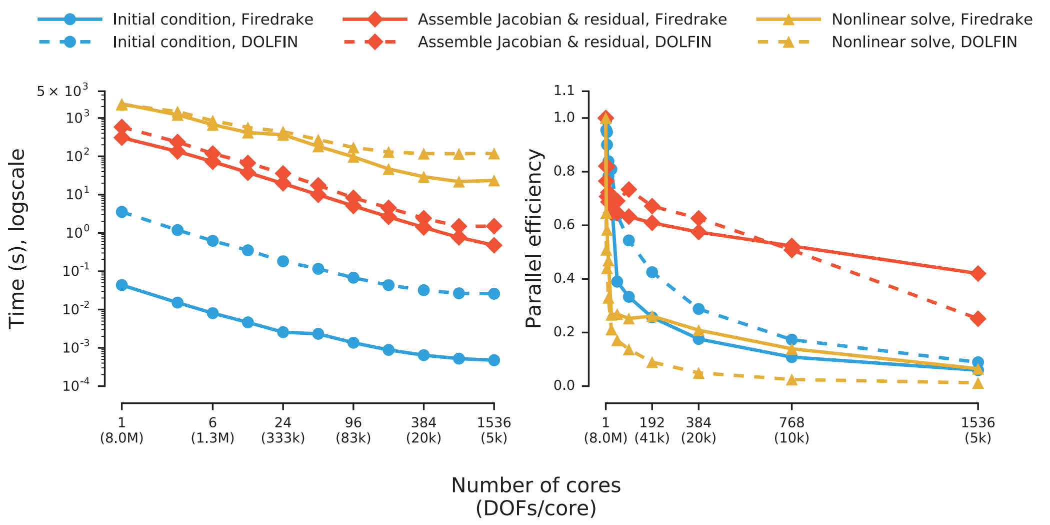

Fig. 8. Strong scaling (left) and parallel efficiency (right) for a Cahn-Hilliard problem with 8M DOFs for 10 timesteps on up to 1,536 cores. Both Firedrake and DOLFIN achieve close to linear scaling for assembly down to 10k DOFs per core. Firedrake is consistently faster by about a factor of 2, demonstrating the efficiency of assembling mixed spaces using the form splitting approach described in Section 5.4.1. Evaluating the initial condition with Firedrake is faster by about two orders of magnitude, demonstrating the efficiency of expression evaluation using a PyOP2 kernel as opposed to a C++ virtual function call required for DOLFIN. Scaling flattens out in both cases from about 40k DOFs per core due to nonparallelizable overheads. Solver scaling is initially equivalent, with Firedrake gaining significantly starting from about 80k DOFs per core. This is due to the use of a PETSc MATNEST, which is more efficient when using a fieldsplit preconditioner by avoiding expensive copies for extracting subblocks of the matrix. The parallel efficiency for strong scaling shows initial advantages for DOLFIN for assembly due to the faster sequential baseline of Firedrake, which catches up at 10k DOFs per core. Efficiency for evaluating the initial condition shows an advantage for DOLFIN again due to a faster Firedrake baseline and is considerably lower than assembly due to nonparallelizable overheads. Solver efficiency is considerably higher for Firedrake.

6.5. Performance Discussion

The experiments presented were selected to demonstrate the performance of Firedrake in several different regimes. By drawing together the results, we can make some observations on the impact of the introduction of the PyOP2 abstraction layer and its implementation.

6.5.1. Assembly in Comparison with DOLFIN. First, assembly of linear and bilinear forms in Firedrake is consistently much faster than in DOLFIN. There are several features of Firedrake that impact on this. Critically, the PyOP2 interface is an abstract basis for code generation, whereas the UFC interface imposed by DOLFIN is a C++ abstract interface [Alnæs et al. 2012]. This means that PyOP2 kernels can be completely inlined, whereas DOLFIN kernels result in multiple virtual function calls per element. The COFFEE optimizations have been found to result in up to a fourfold increase in speed over the quadrature optimizations in FFC [Luporini et al. 2015]. The speedup is most pronounced in the case of the Cahn-Hilliard equation, which employs mixed function spaces. In this case, a performance increase is expected due to the form splitting optimization (see Section 5.4.1).

6.5.2. Scaling Performance. The weak scaling performance of pure Firedrake code (i.e., excluding the PETSc solver) beyond one node is uniformly excellent. Within one node, resource contention results in significantly less than perfect efficiency, but this is expected. In the strong scaling regime, the fixed overhead per field of some hundreds of

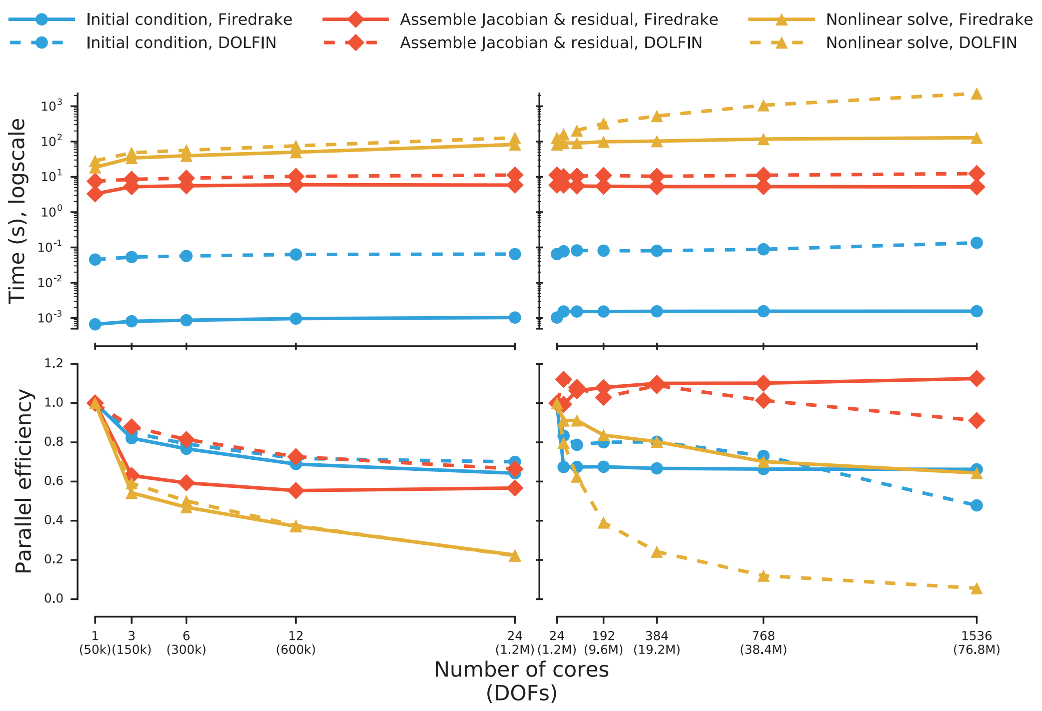

Fig. 9. Weak scaling performance for the Cahn-Hilliard problem with 50k DOFs per core for 10 timesteps. Scaling is shown intranode (1 to 24 cores) relative to a single core (left) and internode (24 to 1,536 cores) relative to a single node (right). Intranode scaling is very similar for Firedrake and DOLFIN with very good efficiencies for all but the solve. Firedrake is faster for assembly and solve by about a factor of 2 and almost two orders of magnitude for the evaluation of the initial condition. Internode, weak scaling for assembly is almost perfect and even superlinear for Firedrake. Efficiency for the initial condition stabilizes at just below 70%. However, the efficiency of the DOLFIN solve slumps, which can be attributed to memory allocations and deallocations required for building the monolithic preconditioner, whereas Firedrake exploits the PETSc MATNEST.

microseconds (Figure 8) results in loss of optimal scaling at a significantly higher DOF count than would be completely optimal. Reduction of the fixed overhead therefore remains an important development objective.

7. CURRENT LIMITATIONS AND FUTURE EXTENSIONS

7.1. Accelerators and Threads

This article presents only performance results for MPI parallel execution on CPUs, with instruction-level vector parallelism facilitated by COFFEE. As Figure 1 shows, PyOP2 also supports execution using OpenMP threads or OpenCL on the CPU, and OpenCL and CUDA on the GPU. Preliminary performance results on these platforms were published in Markall et al. [2013]. However, the available hybrid parallel and GPU linear solver libraries are far more limited than PETSc's MPI-only functionality. The Firedrake developers have therefore given priority to achieving high performance and feature completeness for the CPU backend using MPI and vector parallelism. The other backends are fully functional in the sense that form assembly is supported, and solving is supported to the limits of the relevant solver backends. This demonstrates the utility of the PyOP2 interface in isolating such implementation matters from the specification of the algorithm. However, at this stage, only the MPI CPU backend is considered to be of production quality and suitable for full documentation here. Given the increasingly finegrain parallelism of both CPU and accelerator hardware, hybrid parallel approaches combining message passing with shared memory approaches will be a future direction of development.

7.2. *hp*-Adaptive Finite Element Methods

Support for p-refined finite element methods requires lifting the restriction of PyOP2 maps to fixed arity described in Section 4.2. Permitting variable arity maps and the consequent variable trip count loops in kernels would impede many of the low-level optimizations applied by COFFEE such that both classes of maps should be supported independently. The map storage format would also be required to record the arity of each source element. A more promising option would be to support container maps containing several maps of different arity and corresponding kernels to match. This would enable the support of not only$p$-refinement but also mixed geometry meshes.

7.3. Firedrake Adjoint

Farrell et al. [2013] demonstrated that the mathematical abstraction captured by the FEniCS Language can be exploited to automate the generation and highly efficient execution of the tangent linear and adjoint models corresponding to forward models written in that language. Dolfin-adjoint,[5] the software implementing Farrell et al. [2013], operates on objects at the FEniCS Language level. Using only a short Python wrapper module, dolfin-adjoint has been extended to support Firedrake solvers written using unextended versions of the FEniCS Language. The user-defined extension kernels described in Section 5.3 are not supported by this Firedrake-adjoint, as they cannot be differentiated using UFL's intrinsic symbolic operations. The extension of Firedrake-adjoint to employ traditional algorithmic differentiation methods to custom kernels is planned for the future.

ACKNOWLEDGMENTS

The authors would like to thank Patrick E. Farrell for his help with the Cahn-Hilliard preconditioner implementation and helpful comments on the results section. We would further like to thank the other contributors whose code is in Firedrake: Miklós Homolya, George Boutsioukis, Nicolas Loriant, and Kaho Sato. Finally, we would like to acknowledge the input and ideas that we received from Colin J. Cotter and from the core FEniCS development team, particularly Martin S. Alnæs and Marie E. Rognes.

REFERENCES

Martin S. Alnæs, Anders Logg, and Kent-Andre Mardal. 2012. UFC: A finite element code generation interface. In *Automated Solution of Differential Equations by the Finite Element Method*. Lecture Notes in Computational Science and Engineering, Vol. 84. Springer, 283–302. DOI:http://dx.doi.org/10.1007/978-3-642-23099-8_16

Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software* 40, 2, 9.

Stefan Andersson. 2014. Cray XC30 Architecture Overview. Retrieved November 3, 2016, from http://www.archer.ac.uk/training/courses/craytools/pdf/architecture-overview.pdf.

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Victor Eijkhout, et al. 2014. *PETSc Users Manual*. Technical Report ANL-95/11—Revision 3.5. Argonne National Laboratory, Lemont, IL. http://www.mcs.anl.gov/petsc.

Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. 2007. deal.II—a general purpose object oriented finite element library. *ACM Transactions on Mathematical Software* 33, 4, 24:1–24:7.

Wolfgang Bangerth, Timo Heister, Luca Heltai, Guido Kanschat, Martin Kronbichler, Matthias Maier, Bruno Turcksin, and Toby D. Young. 2013. The `deal.II` library, version 8.1. arXiv:1312.2266v4. http://arxiv.org/abs/1312.2266v4.

[5]http://www.dolfin-adjoint.org/.

Gheorghe-Teodor Bercea, Andrew T. T. McRae, David A. Ham, Lawrence Mitchell, Florian Rathgeber, Luigi Nardi, Fabio Luporini, and Paul H. J. Kelly. 2016. A structure-exploiting numbering algorithm for finite elements on extruded meshes, and its performance evaluation in Firedrake. *Geoscientific Model Development* 9, 10, 3803–3815. DOI:http://dx.doi.org/10.5194/gmd-9-3803-2016

Jessica Bosch, David Kay, Martin Stoll, and Andrew J. Wathen. 2014. Fast solvers for Cahn–Hilliard inpainting. *SIAM Journal on Imaging Sciences* 7, 1, 67–97. DOI:http://dx.doi.org/10.1137/130921842

Philippe G. Ciarlet. 1978. *The Finite Element Method for Elliptic Problems*. Elsevier.

Lisandro Dalcin, Lawrence Mitchell, Jed Brown, Patrick E. Farrell, Michael Lange, Barry Smith, Dmitry Karpeyev, et al. 2016. petsc4py: The Python Interface to PETSc. Retrieved November 3, 2016, from https://zenodo.org/record/56639#.WBvQ0y0rLg8 DOI:http://dx.doi.org/10.5281/zenodo.56639

Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. 2011. Parallel distributed computing using Python. *Advances in Water Resources* 34, 9, 1124–1139.

Andreas Dedner, Robert Klöfkorn, Martin Nolte, and Mario Ohlberger. 2010. A generic interface for parallel and adaptive discretization schemes: Abstraction principles and the Dune-Fem module. *Computing* 90, 3-4, 165–196. DOI:http://dx.doi.org/10.1007/s00607-010-0110-3

Robert D. Falgout, Jim E. Jones, and Ulrike Meier Yang. 2006. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*. Lecture Notes in Computational Science and Engineering, Vol. 51. Springer, 267–294.

Patrick E. Farrell, Colin J. Cotter, and Simon W. Funke. 2014. A framework for the automation of generalized stability theory. *SIAM Journal on Scientific Computing* 36, 1, C25–C48.

Patrick E. Farrell, David A. Ham, Simon W. Funke, and Marie E. Rognes. 2013. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing* 35, C359–393. DOI:http://dx.doi.org/10.1137/120873558

Simon W. Funke and Patrick E. Farrell. 2013. A framework for automated PDE-constrained optimisation. arXiv:1302.3894.

Michael B. Giles, Gihan R. Mudalige, Zohirul Sharif, Graham R. Markall, and Paul H. J. Kelly. 2011. Performance analysis and optimization of the OP2 framework on many-core architectures. *Computer Journal* 55, 2, 168–180.

Frédéric Hecht. 2012. New development in freefem++. *Journal of Numerical Mathematics* 20, 3–4, 251–266.

Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, et al. 2005. An overview of the Trilinos project. *ACM Transactions on Mathematical Software* 31, 3, 397–423.

Magnus R. Hestenes and Eduard Stiefel. 1952. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Institute of Standards and Technology* 49, 6, 409–436.

Robert C. Kirby. 2004. Algorithm 839: FIAT, a new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software* 30, 4, 502–516.

Robert C. Kirby, Matthew Knepley, Anders Logg, and L. Ridgway Scott. 2005. Optimizing the evaluation of finite element matrices. *SIAM Journal on Scientific Computing* 27, 3, 741–758.

Matthew G. Knepley and Dmitry A. Karpeev. 2009. Mesh algorithms for PDE with Sieve I: Mesh distribution. *Scientific Programming* 17, 3, 215–230.

Michael Lange, Lawrence Mitchell, Matthew G. Knepley, and Gerard J. Gorman. 2016. Efficient mesh management in Firedrake using PETSc-DMPlex. *SIAM Journal on Scientific Computing* 38, 5, S143–S155. DOI:http://dx.doi.org/10.1137/15M1026092

Anders Logg, Martin Sandve Alns, Marie E. Rognes, Andrew T. T. McRae, Garth N. Wells, Johannes Ring, Lawrence Mitchell, et al. 2016. ffc: The FEniCS Form Compiler. Retrieved November 3, 2016, from https://zenodo.org/record/56643#.WBvSxi0rLg8 DOI:http://dx.doi.org/10.5281/zenodo.56643

Anders Logg, Kent-Andre Mardal, and Garth N. Wells (Eds.). 2012. *Automated Solution of Differential Equations by the Finite Element Method*. Springer.

Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2012a. FFC: The FEniCS Form Compiler. In *Automated Solution of Differential Equations by the Finite Element Method*. Lecture Notes in Computational Science and Engineering, Vol. 84. Springer, 227–238.

Anders Logg and Garth N. Wells. 2010. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software* 37, 2, Article No. 20. DOI:http://dx.doi.org/10.1145/1731022.1731030

Anders Logg, Garth N. Wells, and Johan Hake. 2012b. DOLFIN: A C++/Python finite element library. In *Automated Solution of Differential Equations by the Finite Element Method*. Lecture Notes in Computational Science and Engineering, Vol. 84. Springer, 173–225. DOI:http://dx.doi.org/10.1007/978-3-642-23099-8_10

Kevin Long, Robert Kirby, and Bart van Bloemen Waanders. 2010. Unified embedded parallel finite element computations via software-based Fréchet differentiation. *SIAM Journal on Scientific Computing* 32, 6, 3323–3351.

Fabio Luporini, Lawrence Mitchell, Mikls Homolya, Florian Rathgeber, Andrew T. T. McRae, David A. Ham, Michael Lange, Graham Markall, and Francis Russell. 2016. COFFEE: A Compiler for Fast Expression Evaluation. Retrieved November 3, 2016, from https://www.zenodo.org/record/58659#.WBvwCC0rLg8 DOI:http://dx.doi.org/10.5281/zenodo.56636

Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. 2015. Cross-loop optimization of arithmetic intensity for finite element local assembly. *ACM Transactions on Architecture and Code Optimization* 11, 4, 57.

Graham R. Markall, Florian Rathgeber, Lawrence Mitchell, Nicolas Loriant, Carlo Bertolli, David A. Ham, and Paul H. J. Kelly. 2013. Performance-portable finite element assembly using PyOP2 and FEniCS. In *Supercomputing*. Lecture Notes in Computer Science, Vol. 7905. Springer, 279–289. DOI:http://dx.doi.org/10.1007/978-3-642-38750-0_21

Andrew T. T. McRae, Gheorghe-Teodor Bercea, Lawrence Mitchell, David A. Ham, and Colin J. Cotter. 2016a. Automated generation and symbolic manipulation of tensor product finite elements. *SIAM Journal on Scientific Computing* 38, 5, S25–S47. DOI:http://dx.doi.org/10.1137/15M1021167

Andrew T. T. McRae, Marie E. Rognes, Anders Logg, David A. Ham, Mikls Homolya, Jan Blechta, Nico Schlmer, et al. 2016b. fiat: The Finite Element Automated Tabulator. Retrieved November 3, 2016, from https://zenodo.org/record/56637#.WBvxjC0rLg8 DOI:http://dx.doi.org/10.5281/zenodo.56637

Lawrence Mitchell, David A. Ham, Florian Rathgeber, Mikls Homolya, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Michael Lange, et al. 2016. firedrake: An automated finite element system. Retrieved November 3, 2016, from https://zenodo.org/record/56640#.WBvx5y0rLg8 DOI:http://dx.doi.org/10.5281/zenodo.56640

Kristian B. Ølgaard and Garth N. Wells. 2010. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software* 37, 1, Article No. 8. DOI:http://dx.doi.org/10.1145/1644001.1644009

Florian Rathgeber. 2014. *Productive and Efficient Computational Science Through Domain-Specific Abstractions*. Ph.D. Dissertation. Imperial College London.

Florian Rathgeber, Graham R. Markall, Lawrence Mitchell, Nicolas Loriant, David A. Ham, Carlo Bertolli, and Paul H. J. Kelly. 2012. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *Proceedings of 2012 SC Companion: High Performance Computing, Networking, Storage, and Analysis*. IEEE, Los Alamitos, CA. 1116–1123.

Florian Rathgeber and Lawrence Mitchell. 2016. firedrake-bench: Benchmarks for Firedrake. Retrieved November 3, 2016, from https://zenodo.org/record/56646#.WBvzji0rLg8 DOI:http://dx.doi.org/10.5281/zenodo.56646

Florian Rathgeber, Lawrence Mitchell, Fabio Luporini, Graham Markall, David A. Ham, Gheorghe-Teodor Bercea, Mikls Homolya, et al. 2016. PyOP2: Framework for performance-portable parallel computations on unstructured meshes. Retrieved November 3, 2016, from https://zenodo.org/record/56635#.WBv0Ey0rLg8 DOI:http://dx.doi.org/10.5281/zenodo.56635

Marie E. Rognes, David A. Ham, Colin J. Cotter, and Andrew T. T. McRae. 2013. Automating the solution of PDEs on the sphere and other manifolds in FEniCS 1.2. *Geoscientific Model Development* 6, 6, 2099–2119.

Marie E. Rognes and Anders Logg. 2013. Automated goal-oriented error control I: Stationary variational problems. *SIAM Journal on Scientific Computing* 35, 3, C173–C193.

Barry Smith, Satish Balay, Matthew Knepley, Jed Brown, Lois Curfman McInnes, Hong Zhang, Peter Brune, et al. 2016. petsc: Portable, Extensible Toolkit for Scientific Computation. Retrieved November 3, 2016, from https://zenodo.org/record/56641#.WBv0hC0rLg8 DOI:http://dx.doi.org/10.5281/zenodo.56641

Received January 2015; revised July 2016; accepted September 2016