Scalable I/O and Checkpointing for Firedrake

Koki Sagiyama[1], Vaclav Hapla[2], Matthew G. Knepley[3], Lawrence Mitchell[4], and David A. Ham[1]

[1]Department of Mathematics, Imperial College London
[2]Department of Earth Sciences, ETH Zürich
[3]Department of Computer Science and Engineering, University at Buffalo
[4]Department of Computer Science, Durham University

November 11, 2021

Abstract

Firedrake [1] is a system for solving partial differential equations using finite element methods. In this work we enhance the I/O checkpointing capabilities of Firedrake introducing a new interface. The new interface allows for saving and loading functions representing fields in association with meshes of the domain, with domains in the same HDF5 file. The I/O is efficient and scalable, and allows saving and loading on different numbers of MPI processes.

1 Introduction

Firedrake [1] is a high-level, high-productivity system for the specification and solution of partial differential equations (PDEs) using the finite element method. The core data structures used to represent solutions of PDEs are the combination of a mesh of the domain of interest and a function belonging to some discrete function space. Although computation is fully and transparently parallel once a mesh is loaded, Firedrake has long been missing scalable and flexible I/O checkpointing for meshes and functions.

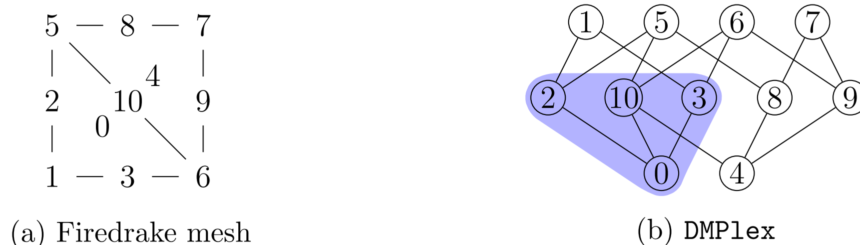
In this work, we implement a scalable and flexible parallel checkpointing infrastructure for meshes and functions using the standard HDF5 parallel file format. We introduce a new class, `CheckpointFile`, which enables saving/loading meshes and functions collectively to/from a single HDF5 file taking advantage of parallel filesystems and allow for restarting and post-processing on a number of processes appropriate to that phase of the simulation. We also provide transparent interfaces for extruded meshes and time-dependent problems.

In Sec. 2 we explain the concept for checkpointing meshes and functions. In Sec. 3 we illustrate example usages of the new interface. In Sec. 4 we evaluate our implementations on Archer2. Sec. 5 summarises this work.

2 Technical details

2.1 Checkpointing meshes

Firedrake uses parallel mesh data structures called `DMPlex` [2, 3, 4] provided by the Portable, Extensible Toolkit for Scientific Computation (PETSc) [5, 6] to describe mesh topologies. A `DMPlex` represents mesh entities such as vertices, edges, faces, and cells, which we will call *points*, and the connectivities between them, such as which faces are adjacent to a given cell. This adjacency relation is embodied in a directed acyclic graph (DAG), referred to in mathematics as a Hasse Diagram [7].



(a) Firedrake mesh

(b) `DMPlex`

Figure 1: (a) An example Firedrake mesh; global point numbers associated with the cells, faces, and vertices are shown. (b) `DMPlex` representation of the mesh topology; an example adjacency relation is highlighted, where the cell with global point number 0 is connected to the faces with global point numbers 2, 10, and 3.

Fig. 1a shows an example Firedrake mesh and Fig. 1b shows the `DMPlex` representing the mesh topology, where each point is shown as a *global point number* that we will introduce shortly. Using this flexible representation, Plex is able to manipulate meshes of any dimension, which any combination of cell shapes, even geometrically non-conforming meshes with hanging nodes [8]. `DMPlex` decorates each edge of the DAG with an orientation number. This identifies the member of the dihedral group for each subcell (face) used to transform it from the canonical ordering before attaching it to a given cell. The orientation concept will turn out to be crucial in Sec. 2.2. Since a `DMPlex` also carries mesh coordinates, checkpointing a Firedrake mesh amounts to checkpointing the associated `DMPlex`, and we use the existing PETSc interfaces for this purpose.

In a local mesh, each mesh point (vertex, edge, face, cell, etc.) carries a distinct *local point number* in `[0, n)`, where `n` is the total number of topological points in that `DMPlex`. A parallel mesh is simply a collection of `DMPlex` objects combined with a map relating points of the mesh on one process to that on another, such as shared vertices along a process boundary. This map is encoded in an instance of `PetscSF` [9], a scalable PETSc implementation of the concept of *star forests* that allows for moving data from one set to another in a specified way. For parallel output, we create a global numbering of mesh points across processes so that each one gets a distinct *global point number* chosen from `X` = `[0, N)`, where `N` is the total number of distinct mesh points in the collection of `DMPlex`es. We save the parallel `DMPlex` in association with those global point numbers.

When we load a `DMPlex`, we use global point numbers to reconstruct the saved topology. Currently, PETSc can load a `DMPlex` in parallel only from the XDMF format [10], but our development hereafter is general and will require no modification when parallel loading becomes available from the HDF5 format in PETSc. Fig. 2 shows the data loading methodology, where two MPI processes, process 0 and process 1, are in use. In Fig. 2 we load the `DMPlex` shown in Fig. 1b as an example and we call the loaded `DMPlex`

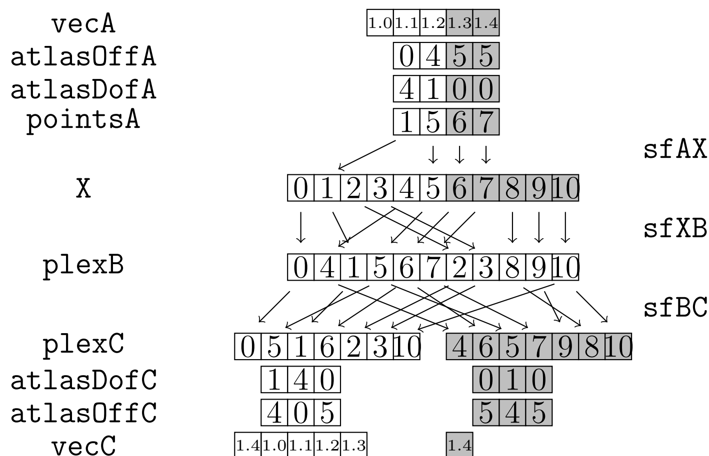


Figure 2: A schematic of an example infrastructure for loading a `PetscSection` and a `Vec` in association with a `DMPlex` in PETSc using two MPI processes, process 0 and process 1; data that process 0 can "see" are shown in uncoloured squares and those that process 1 can see are shown in darker squares.

`plexB`. `plexB` is loaded solely on process 0 representing current PETSc restriction, and the mesh points of `plexB` are shown as an array of associated global point numbers. Fig. 2 also shows `X`, which is partitioned into two; data that process 0 can "see" are shown in uncoloured squares and those that process 1 can see are shown in darker squares. We note that `X` is an abstract sequence of all global point numbers introduced to facilitate loading Firedrake functions as described in Sec. 2.2 and it is not directly stored in the file nor is it a loaded object. Here, we modified the PETSc interface so that a `PetscSF` object, `sfXB`, would be constructed when loading `plexB`; `sfXB` is to move data from the partitioned sequence of global point numbers, `X`, to the serial distribution in `plexB` based on the matching global point numbers, and the data flow is illustrated with arrows in Fig. 2. Note that one can readily compose multiple `PetscSF`s to move data successively. Loaded `DMPlex`es often have poor parallel distributions since chunks loaded from disk rarely correspond to good partitions. We thus redistribute `plexB` across MPI processes to obtain a redistributed `DMPlex`, `plexC`; in Fig. 2 points of `plexC` are again shown as an array of associated global point numbers. Redistribution constructs another `PetscSF`, `sfBC`, which allows for moving data from `plexB` to `plexC`; the corresponding data flow is depicted in Fig. 2.

2.2 Checkpointing functions

Given a Firedrake mesh holding a `DMPlex`, Firedrake defines a finite element function space using a *UFL element*, a symbolic representation of a finite element provided by the Unified Form Language (UFL) package [11]. A Firedrake function is then defined on the finite element function space. In order to describe the data layout of the function in PETSc, Firedrake creates a `PetscSection` object that associates the function space degrees of freedom (DoFs) with the `DMPlex` mesh points; a `PetscSection` is just a map from integers, here the mesh points, to sets of integers, here the global DoFs. For each mesh point, the section stores the number of DoFs associated with that point in its `atlasDof` array. Given a traversal order of the topological points, `PetscSection` then defines the order of the DoFs and stores the offset to the first DoF on each point in its `atlasOff` array. A PETSc `Vec` object associated with the `PetscSection` then stores DoF values of the function, at indices computed from the `atlasDof` and `atlasOff` arrays. A key feature of this new I/O scheme is the ability to save and load `PetscSection`s and `Vec`s in association with a `DMPlex` on which they are defined, using an arbitrary number of MPI processes. Saving a `PetscSection` amounts to saving the `atlasDof` and `atlasOff`

arrays along with an array of associated global point numbers constituting the domain of the mapping. A `Vec` object is then saved in association with the `PetscSection`. For Firedrake, we have in addition enough information to reconstruct the particular UFL element used.

Loading the `PetscSection` amounts to loading the `atlasDof` array, the `atlasOff` array, and the array of global point numbers, which are partitioned into equal chunks across MPI processes. These arrays are denoted `atlasDofA`, `atlasOffA`, and `pointsA`, and they represent the in-memory copy of the on-disk `PetscSection`, here called `sectionA`. The `Vec` is loaded and partitioned independently, and we here denote it `vecA`. Fig. 2 shows a general example, where mesh points with global numbers 1, 5, 6, 7 have 4, 1, 0, 0 DoFs, respectively, and values of DoFs on the first two are [1.0, 1.1, 1.2, 1.3] and [1.4]. We then use `pointsA` to construct a `PetscSF`, `sfAX`, that moves data from `pointsA` to the layout specified by `X`. Composing `sfAX`, `sfXB`, and `sfBC`, one can construct another `PetscSF`, `sfAC`, that directly moves data, `atlasDofA` and `atlasOffA`, from `pointsA` to the layout specified by `plexC`; we denote the arrays thus constructed in reference to `plexC` as `atlasDofC` and `atlasOffC`, respectively. Fig. 2 depicts construction of `atlasDofC` and `atlasOffC`. These arrays contain the number of DoFs that each mesh point in `plexC` has and the offset in `vecA` of the first DoF on that point. This allows us to construct a new `PetscSF` that moves data in `vecA` to `vecC`, a new `Vec` defined on `sectionC`; see Fig. 2 for illustration of the data flow.

In addition to the above we load the UFL element for a full description of the function space. While DoF values on a given mesh point are stored in a contiguous chunk of the `Vec`, how these DoFs are laid out in space, meaning how they map to members of the dual space, cannot be determined without additional knowledge of the element. For functions on CG and DG spaces, we handle this directly by making Firedrake explicitly define the DoF layout on each point based on the canonical orientation of that point determined by PETSc (see Sec. 2.3). This allows one to checkpoint those functions natively. Functions on other function spaces are first *embedded* from the original spaces to the DG spaces of appropriate polynomial degrees, then saved and pushed back from those spaces to the original spaces when loaded.

2.3 Orientations

Defining the DoF layouts of CG and DG finite elements based on the canonical orientations of entities is essential for checkpointing functions as described in Sec. 2.2, but it in turn necessitates Firedrake to explicitly handle how DoFs on a reference element, provided by the FIAT [12] or FInAT [13] packages, map to those on a given mesh entity for these finite element spaces.

Each entity on a FIAT/FInAT reference cell has a canonical orientation based on which DoFs on that entity are laid out. As, for the relevant finite elements, Firedrake lays out DoFs on a given mesh entity based on the `DMPlex` canonical orientation, the mapping of the reference DoFs to the physical DoFs is determined by how the mapped reference cell entity is oriented relative to the target mesh entity, which is simply called *orientation* and represented by an integer. For instance, a triangular cell can have six possible orientations. DoF mappings must then be defined for each entity for each possible orientation. For CG and DG elements, those DoF mappings are represented by permutations and we have implemented those in FIAT and FInAT.

Fig. 2.3 shows an example. Fig. 3a depicts the DoF layout of the DP3 reference finite

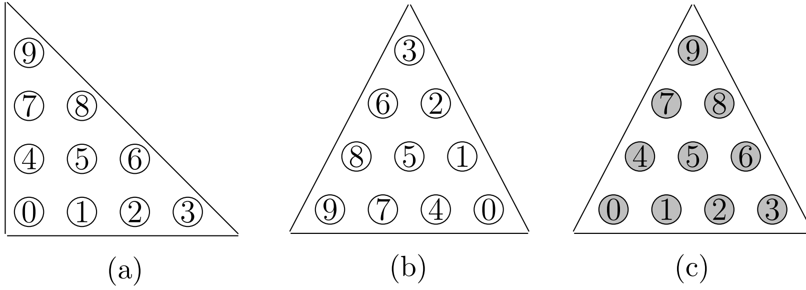


Figure 3: DoF layouts for DP3 element on (a) the FIAT reference cell, (b) the FIAT reference cell rotated once "counterclockwise" to match the mesh cell, and (c) the mesh cell.

element on the reference triangle in FIAT and Fig. 3c depicts the canonical DoF layout on a certain mesh cell for the DP3 function space. If the FIAT reference cell is mapped to the mesh cell under a "counterclockwise" rotation as shown in Fig. 3b, the DoF mapping for the cell entity for this specific orientation is given by [3, 6, 8, 9, 2, 5, 7, 1, 4, 0], since it maps `Vec` values to FIAT values.

2.4 Extrusion

An extruded mesh in Firedrake is a semi-structured mesh that consists of one unstructured Firedrake mesh, which we will call the *base mesh*, and one structured direction in which the base mesh is *extruded*; an extruded mesh thus has one higher topological dimension than the base mesh. Extruded meshes have been used for thin domain problems such as coastal ocean simulations.

Just as with other meshes in Firedrake, checkpointing an extruded mesh amounts to checkpointing its topology and coordinates. An extruded mesh topology is defined by the underlying base mesh topology represented by a `DMPlex` and the `layers` parameter describing number of extruded cells in the extruded direction. The `layers` parameter can be an integer for a uniform extrusion across the base mesh or can represent an integer pair per base mesh cell for a variable layer extrusion; for the latter, the integer pair contains the first layer index and the last layer index and only extruded cells between these layers are included in the extruded mesh. When checkpointing the integer pairs defined on each base mesh cell, we note that the data structure resembles that of a DG0 vector function of dimension two on the base mesh and use the infrastructure for checkpointing functions on the base mesh as described Sec. 2.2.

When checkpointing coordinates and functions, one needs to store the UFL element for the corresponding function space on the extruded mesh. Potentially embedding them in an appropriate DG space, one can then recast them as functions on the base mesh and readily save/load them as described in Sec. 2.2.

We finally note that, as the overall DoF layout of a function is, by construction, unambiguously determined once DoF layout on the base mesh is known, one can readily compute the DoF mappings from the reference element to the physical element from the base mesh entity orientations.

2.5 Timestepping

PETSc provides a convenient interface to store a `Vec` in a time-dependent simulation, where an additional axis representing the timesteps is added to the HDF5 `Dataset` for the `Vec`. Firedrake wraps this PETSc interface and allows for storing functions with an additional integer parameter representing the timestep.

3 Firedrake `CheckpointFile` API

Listing 1 shows a basic usage of `CheckpointFile`, where we save/load a mesh, "`m`", and a function, "`f`", to/from "`a.h5`". The interface for extruded meshes is the same (except that an extruded mesh must be created), as shown in Listing 2 where we checkpoint an extruded mesh, "`extm`", and a function, "`g`", defined on it.

```
mesh = UnitSquareMesh(8, 8, name="m")


V = FunctionSpace(mesh, "CG", 2)
f = Function(V, name="f")
with CheckpointFile("a.h5", "w") as ck:
    ck.save_mesh(mesh)  # optional
    ck.save_function(f)
with CheckpointFile("a.h5", "r") as ck:
    mesh = ck.load_mesh("m")
    f = ck.load_function(mesh, "f")
```

Listing 1: Code example for basic usage.

```
mesh = UnitSquareMesh(8, 8, name="m")
extm = ExtrudedMesh(mesh, layers=4,
                    name="extm")
V = FunctionSpace(extm, "CG", 2)
g = Function(V, name="g")
with CheckpointFile("b.h5", "w") as ck:
    ck.save_mesh(extm)  # optional
    ck.save_function(g)
with CheckpointFile("b.h5", "r") as ck:
    extm = ck.load_mesh("extm")
    g = ck.load_function(extm, "g")
```

Listing 2: Code example for extrusion.

4 Evaluation

We tested our `CheckpointFile` implementation for correctness and performance on Archer2. In the following we fixed the number of processes per node to 128 and the LFS stripe size to the default value, 1,048,576. The `stripe count c` is set to -1 (maximum) unless otherwise noted. We also included `export FI_OFI_RXM_SAR_LIMIT=64K` in all our job scripts. We compiled PETSc for 64 bit integers.

4.1 Correctness

To test correctness, we used a mesh of the unit sphere that contained about 32 million tetrahedral cells. We first used 32 Archer2 nodes. We loaded the mesh, constructed a function with DP4 finite element, let it interpolate `h`$=\sin(16\cdot(2\pi)(x+y+z))$, where x, y, and z are spatial coordinates, and saved it along with the mesh. The L_2 norm error of the interpolation was 0.00378. We then used 64 Archer2 nodes and loaded this function along with the mesh. The loaded function was compared with a new function interpolating `h` with DP4 element on the loaded mesh; the L_2 norm error between the loaded function and the new function was $1.23\cdot 10^{-14}$, which proved that the save-load cycle was virtually lossless.

4.2 Performance

To test weak scalability for saving, we first ran a benchmark test for HDF5 parallel output [benchio]. In the benchmark we saved 2.1 million double-precision numbers per MPI process using 1, 8, and 64 Archer2 nodes and measured the wall-clock times required for three different LFS `stripe counts`, `c` = 1, 4, and -1 (maximum); the result is plotted in Fig. 4a. Observed bandwidth for `c`=-1 were 0.59, 0.98, and 0.86 [GiB/s], respectively, which suggested that saving data of this size was bandwidth limited.

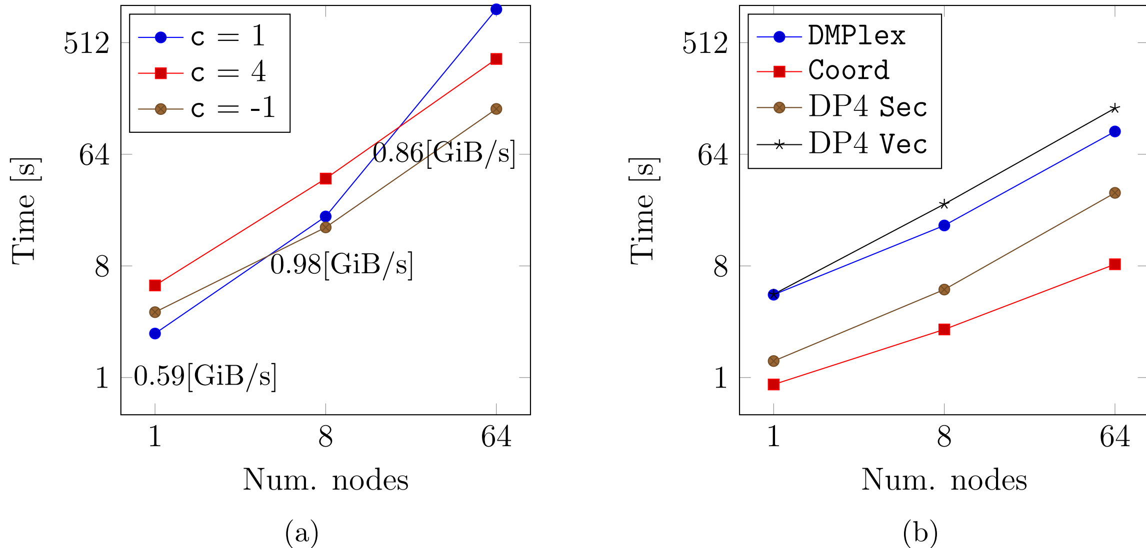


Figure 4: (a) Weak scalability benchmark test for HDF5 parallel saving on Archer2 [benchio] run on 1, 8, and 64 Archer2 nodes for three different `stripe counts`, `c` = 1, 4, and, -1 (maximum): each process saved about 2.1 million double-precision numbers. For `c` = -1, observed bandwidth were 0.59, 0.98, and 0.86 [GiB/s], respectively. (b) Weak scalability test for Firedrake saving run on 1, 8, and 64 Archer2 nodes with `c`=-1. For each case, a tetrahedral mesh and a DP4 function were created so that each process would own about 29 thousand cells and 1.0 million DoFs. The largest problem with 64 nodes thus involved a total of 0.24 billion cells and 8.3 billion DoFs. Wall-clock times in second required for saving `DMPlex`, `Coordinates`, DP4 `PetscSection`, and DP4 `Vec` are plotted.

We then performed a weak scalability test for Firedrake using 1, 8, and 64 nodes with `c` = -1. For each case, a tetrahedral mesh and a DP4 function were created so that each process would own about 29 thousand cells and 1.0 million DoFs. The largest problem with 64 nodes thus involved a total of 0.24 billion cells and 8.3 billion DoFs. Wall-clock times in second required to save the `DMPlex`, `Cordinates`, `PetscSection` for the DP4 function, and `Vec` for the DP4 function were measured; these are plotted in Fig. 4b. Overall our scalability test for Firedrake showed the same trend as the benchmark test. The array saved in the benchmark test was about twice as large as the `Vec` for the DP4 function, while required times for saving these two objects were about the same. We believe that this is because HDF5 chunk size of PETSc `Vec` is automatically set to 33,554,431, which is much larger than 1,048,576, the LFS stripe size that we used. Saving `Coordinates` involves saving the coordiante `Vec` and the associated coordinate `PetscSection`. Saving the `Coordinates` being much faster than saving the DP4 `PetscSection` and `Vec` is consistent with that the coordinate `PetscSection` only stores DoF and offset data for vertex points on the `DMPlex` while DP4 `PetscSection` stores those for all points.

To test performance for loading, we used 8 Archer2 nodes. We refined a three-dimensional tetrahedral mesh three times, each refinement producing eight times more cells, and, for each refinement level, we created a function with DP4 finite element; the largest problem involved about 29 million mesh cells and 1.0 billion DoFs. Wall-clock times in second required for loading the `DMPlex`, redistributing the `DMPlex`, loading the `Coordinates`, loading the DP4 `PetscSection`, loading the DP4 `Vec` were measured and plotted in Fig. 5 for each number of mesh cells. As described in Sec. 2.2, loading `PetscSections` and `Vecs` requires data distribution using `PetscSF` in addition to mere

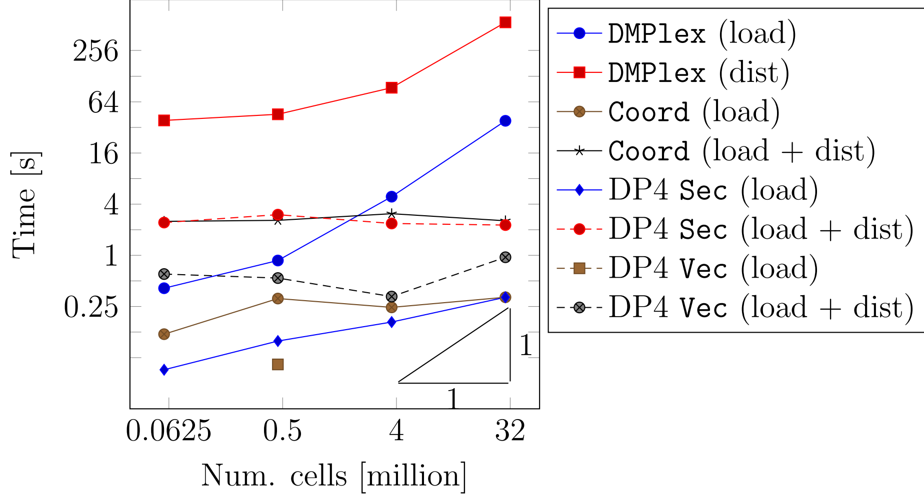


Figure 5: Scalability test for Firedrake loading run on 8 Archer2 nodes with `c = -1`. Tetrahedral meshes of four refinement levels were created and a DP4 function was created on each. The largest problem involved a total of about 29 million mesh cells and 1.0 billion DoFs. Wall-clock times in second required for loading `DMPlex`, distributing `DMPlex`, loading `Coordinates`, loading and distributing `Coordinates`, loading DP4 `PetscSection`, loading and distributing DP4 `PetscSection`, loading DP4 `Vec`, and loading and distributing DP4 `Vec` are shown for each refinement level.

data loading, so wall-clock times required merely for loading data are also separately shown in Fig. 5 for the `Coordinates`, DP4 `PetscSection`, and DP4 `Vec`. Note that `DMPlex`es were loaded in serial due to current PETSc restriction. The wall-clock times required for loading `DMPlex`es and redistributing them increased linearly as expected as the number of cells increased. `Coordinates`, DP4 `PetscSection`, and DP4 `Vec`, were, on the other hand, loaded in parallel and the required time did not increase linearly, indicating that loading performance for these objects are process count limited at this scale due to required data distribution by `PetscSF`s.

5 Conclusions

We have successfully enhanced the I/O capabilities in PETSc and in Firedrake to efficiently save and load functions in association with meshes using arbitrary number of MPI processes. This also required rethinking the way in which Firedrake handles orientations for CG and DG finite elements. We have also added transparent interfaces in Firedrake for extrusion and timestepping problems. We evaluated correctness and scalability of our implementation on Archer2. In upcoming work, the HDF5 mesh loading in PETSc will be parallelized.

Acknowledgment

This work was funded under the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk).

Code availability

The version of Firedrake that can be used to reproduce the results of the experiments in this paper has been archived on Zenodo [14]. The scripts used to run the experiments are also available on Zenodo [15].

References

[1] Rathgeber F, Ham DA, Mitchell L, Lange M, Luporini F, McRae ATT, et al. Firedrake: automating the finite element method by composing abstractions. ACM Trans Math Softw. 2016;43(3):24:1–24:27. Available from: http://arxiv.org/abs/1501.01809.

[2] Knepley MG, Karpeev DA. Mesh Algorithms for PDE with Sieve I: Mesh Distribution. Scientific Programming. 2009;17(3):215–230. Available from: http://arxiv.org/abs/0908.4427.

[3] Lange M, Mitchell L, Knepley MG, Gorman GJ. Efficient mesh management in Firedrake using PETSc-DMPlex. SIAM Journal on Scientific Computing. 2016;38(5):S143–S155. Available from: https://doi.org/10.1137/15M1026092.

[4] Lange M, Knepley MG, Gorman GJ. Flexible, Scalable Mesh and Data Management using PETSc DMPlex. In: Proceedings of the Exascale Applications and Software Conference; 2015. Available from: http://www.easc2015.ed.ac.uk/sites/default/files/attachments/EASC15Proceedings.pdf.

[5] Balay S, Abhyankar S, Adams MF, Benson S, Brown J, Brune P, et al. PETSc/TAO Users Manual. Argonne National Laboratory; 2021. ANL-21/39 - Revision 3.16. Available from: https://petsc.org/release/docs/manual/.

[6] Balay S, Abhyankar S, Adams MF, Benson S, Brown J, Brune P, et al.. PETSc Web page; 2021. Available from: https://petsc.org/.

[7] Wikipedia. Hasse Diagram; 2015. Available from: http://en.wikipedia.org/wiki/Hasse_diagram.

[8] Isaac T, Knepley MG. Support for Non-conformal Meshes in PETSc's DMPlex Interface. ACM Transaction on Mathematical Software. 2017. In review. Available from: https://arxiv.org/abs/1508.02470.

[9] Zhang J, Brown J, Balay S, Faibussowitsch J, Knepley MG, Marin O, et al. The PetscSF Scalable Communication Layer. IEEE Transactions on Parallel and Distributed Systems. 2021. Accepted. Available from: https://arxiv.org/abs/2102.13018.

[10] Hapla V, Knepley MG, Afanasiev M, Boehm C, van Driel M, Krischer L, et al. Fully Parallel Mesh I/O using PETSc DMPlex with an Application to Waveform Modeling. SIAM Journal on Scientific Computing. 2021;43(2):C127–C153. Available from: https://arxiv.org/abs/2004.08729.

[11] Alnæs M, Logg A, Ølgaard K, Rognes M, Wells G. ”Unified Form Language: A domain-specific language for weak formulations of partial differential equations”. ACM Transactions on Mathematical Software. 2014;40(2). Available from: https://doi.org/10.1145/2566630.

[12] Kirby RC. Algorithm 839: FIAT, a New Paradigm for Computing Finite Element Basis Functions. ACM Trans Math Softw. 2004 Dec;30(4):502–516. Available from: https://doi.org/10.1145/1039813.1039820.

[13] Homolya M, Kirby RC, Ham DA. Exposing and exploiting structure: optimal code generation for high-order finite element methods; 2017. Available from: https://arxiv.org/abs/1711.02473.

[14] Software used in ‘Scalable I/O and Checkpointing for Firedrake’; 2021. Available from: https://doi.org/10.5281/zenodo.5648900.

[15] ecse firedrake io: examples and outputs; 2021. Available from: https://doi.org/10.5281/zenodo.5648452.