FULLY PARALLEL MESH I/O USING PETSC DMPLEX WITH AN APPLICATION TO WAVEFORM MODELING*

VACLAV HAPLA†, MATTHEW G. KNEPLEY‡, MICHAEL AFANASIEV†, CHRISTIAN BOEHM†, MARTIN VAN DRIEL†, LION KRISCHER†, AND ANDREAS FICHTNER†

Abstract. Large-scale PDE simulations using high-order finite-element methods on unstructured meshes are an indispensable tool in science and engineering. The widely used open-source PETSc library offers an efficient representation of generic unstructured meshes within its DMPlex module. This paper details our recent implementation of parallel mesh reading and topological interpolation (computation of intermediate codimensions from a cell-vertex mesh) into DMPlex. We apply these developments to seismic wave propagation scenarios on Mars as a sample application. The principal motivation is to overcome single-node memory limits and reach mesh sizes which were impossible before. Moreover, we demonstrate that scalability of I/O and topological interpolation goes beyond 12'000 cores, and memory imposed limits on maximum mesh size vanish.

Key words. unstructured mesh, directed acyclic graph, partitioning, topological interpolation, parallel I/O, PETSc, DMPlex, seismic waveform modeling, Spectral Element Method

AMS subject classifications. 65-04, 65Y05, 65M50, 05C90, 35L05

1. Introduction.

Finite-element methods (FEM) [37, 24, 51] are widely used in science and engineering to simulate complex physical systems. Many applications require FEM dicretization with high polynomial order on large unstructured meshes requiring distributed memory computer architectures [1, 14, 46, 34, 2]. Here, two complications may arise:

(1) The distributed memory mesh representation should explicitly include mesh entities of all codimensions (vertices, edges, faces, cells). However, mesh datafiles typically store only vertices and cells to avoid redundant storage, disk operations, and to conform to widely used formats. Edges and faces can be computed in runtime using a process which will be called here *topological interpolation*.

(2) Meshing tools typically create a single datafile, but loading the whole mesh onto one processor and distributing onto all remaining processors becomes a bottleneck for a sufficiently large mesh. Instead, we need to load different portions of the mesh file directly onto target processors and maintain a distributed mesh representation right from beginning. Additional load balancing and redistribution can be done using a parallel partitioner.

PETSc DMPlex (section 2) is a flexible mesh implementation which addresses (1) by design, i.e., it can represent any number of codimensions and implements topological interpolation. Moreover, it is completely agnostic to the mesh shape, dimensionality and cell types. However, only with developments presented in this paper, also (2) has been addressed. This in turn brought another challenge to implement the topological interpolation from (1) in parallel.

In this paper, we do not deal with hierarchical mesh representations such as octree data structures [20, 47]. They are advantageous for certain data access patterns, especially in the context of Adaptive Mesh Refinement (AMR) [20, 12], at the expense of generality; DMPlex supports, for instance, arbitrary mesh partitions and extraction of arbitrary subsets of cells (or facets) as submeshes: features which are

*Submitted to the editors on April 18, 2020.

†ETH Zurich, Switzerland (vaclav.hapla@erdw.ethz.ch, andreas.fichtner@erdw.ethz.ch).

‡University at Buffalo, NY (knepley@buffalo.edu).

typically missing from hierarchical meshing frameworks [26]. Note that PETSc offers the DMForest wrapper of p4est [13, 25, 46], and conversion between DMPlex and DMForest [26].

We demonstrate the efficiency and potential of our new DMPlex functionality on a real world application in the context of the full waveform modeling. The spectral-element method on unstructured conforming hexahedral meshes has become the de-facto standard for global-scale simulations of seismic waves [1, 18, 19, 44]. We apply it to simulate full 3D high-frequency wave propagation on Mars, based on data from the NASA InSight mission [6]. This consists in solving a coupled system of the elastic and acoustic wave equations. To accurately model these data in the desired frequency band, large scale simulations are required. In the presented simulation, more than 100 million 4-th order hexahedral mesh elements were used.

Before the developments presented in this manuscript, with mesh reading and topological interpolation being serial, mesh sizes were limited by the single node memory. On our testing platform, Piz Daint supercomputer at the Swiss National Supercomputing Center, the mesh size limit was approximately 16 million elements, rendering such simulations impossible.

The manuscript is organized as follows. First, we describe abstractions for mesh data management of unstructured meshes in high-order finite-element discretizations using PETSc DMPlex. Then we explain our new strategies for the parallel simulation startup (mesh reading, topological interpolation and redistribution) on distributed memory HPC architectures. We continue with a brief introduction of the spectral-element method and its implementation which uses DMPlex. Next, waveform modeling benchmarks demonstrate scalable performance of the parallel startup for up to 256 million hexahedral mesh elements, running on up to 1024 Cray XC50 nodes of Piz Daint. Finally, the mentioned Mars seismic wave propagation simulation is presented as a practical application.

2. DMPlex. PETSc [3, 4, 5, 10] is a well-known library for numerical methods, used by the scientific and engineering computing communities. It provides parallel data management, structured and unstructured meshes, linear and nonlinear algebraic solvers and preconditioners, time integrators, optimization algorithms and others. Many of these methods (such as geometric multigrid and domain decomposition solvers) can take advantage of the geometric/topological setting of a discretized problem, i.e., mesh information.

DMPlex is a PETSc module for generic unstructured mesh storage and operations. It decouples user applications from the implementation details of common mesh-related utility tasks, such as file I/O and mesh partitioning. It represents the mesh topology in a flexible way, providing topological connectivity of mesh entities at all codimensions (vertices, edges, faces and cells), crucial for high-order finite-element method (FEM) simulations, and provides a wide range of common mesh management functionalities.

PETSc's interface for serving mesh data to numerical algorithms is the `DM` object. PETSc has several `DM` implementations. The native implementations of structured grids (`DMDA`), staggered grids `DMStag`, and unstructured meshes (`DMPlex`) have the most complete coverage of the `DM` API, and are developed most actively. Besides these two, PETSc has several `DM` implementations that wrap external libraries, such as `DMMOAB` for MOAB [48] and `DMFOREST` for p4est [13]. Here we will focus on `DMPlex` which proved to be most relevant for the discussed waveform modeling applications.

`DMPlex` encapsulates the topology of unstructured grids and provides a wide range

of common mesh management functionalities to application programmers [45, 36, 35, 7]. It provides a domain topology abstraction that decouples user applications from the implementation details of common mesh-related utility tasks, such as file I/O, mesh partitioning, and parallel load balancing [33]. It aims to increase extensibility and interoperability between scientific applications through librarization [5, 10].

2.1. Mesh representation and basic operations. `DMPlex` uses an abstract representation of the unstructured mesh topology, where the connectivity of topological entities (vertices, edges, faces, cells) is stored as a directed acyclic graph (DAG) [38, 32], also refered to as a Hasse Diagram in topology. Let us call this representation a *plex* and refer to the topological mesh entities as *points*. The plex is constructed of clearly defined layers (*strata*) that enable access to mesh entities by their topological codimension without explicit reference to the overall topological dimension d of the mesh, see Figure 1(b). Note also that d can be reconstructed in linear time from an unlabeled plex using Depth First Search. Let us denote a set of points at the same stratum as $\texttt{Stratum}(h)$, where h is an integer *height*, $h \leq d+1$.

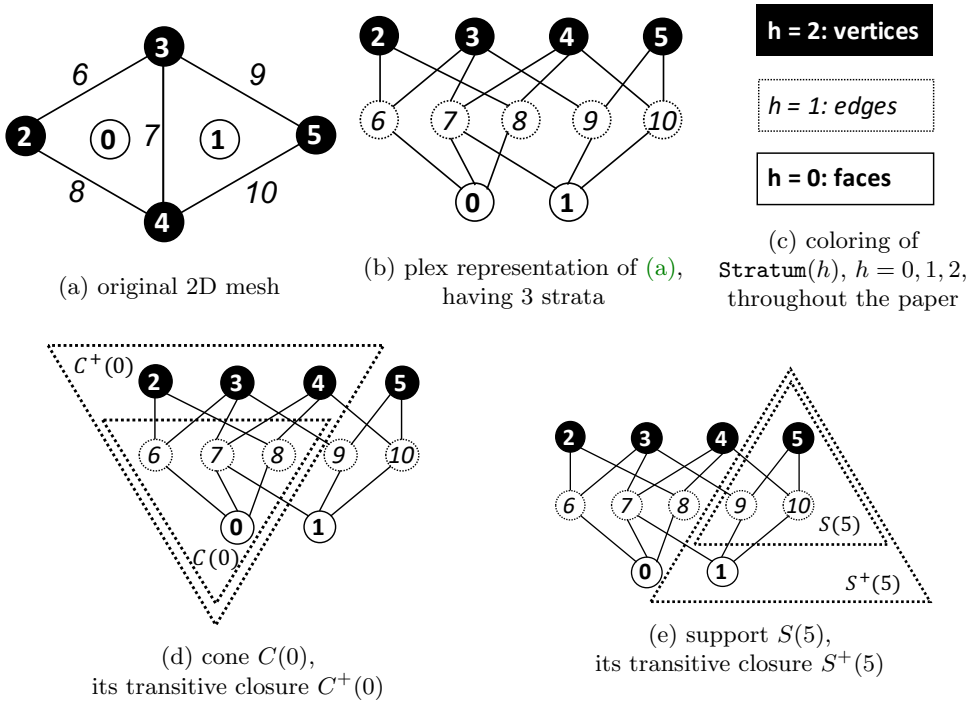
All points in the plex share a single consecutive numbering, emphasizing that each point is treated equally no matter its shape or dimension, and allowing `DMPlex` to store the graph connectivity in a single array where dimensional layers are defined as consecutively numbered subranges. The directional connectivity of the plex is defined by the covering relation called *cone*, denoted here as $C(p)$, yielding a set of all points directly connected to p in the next codimension. The transitive closure of the cone operation shall be denoted by $C^+(p)$. Both $C(p)$ and $C^+(p)$ are illustrated in Figure 1(d). The dual operation called *support* and denoted $S(p)$, and its transitive closure $S^+(p)$ are shown in Figure 1(e).

In addition to the abstract topology data, PETSc provides two utility objects to describe the parallel data layout: a `PetscSection` object maps the graph-based topology information to discretized solution data through an offset mapping, and a star forest (`PetscSF`, see subsection 3.2.1) object holds a one-sided description of shared data in parallel. These data layout mappings allow `DMPlex` to manage distributed solution data by automating the preallocation of distributed vector and matrix data structures and performing halo data exchanges. Moreover, by storing grid topology alongside discretized solution data, `DMPlex` is able to provide the mappings required for sophisticated preconditioning algorithms, such as geometric multigrid methods [11, 17] and multiblock, or "fieldsplit" preconditioning for multiphysics problems [9].

2.2. Topological interpolation. For high order methods we are interested in, we need mesh entities of all codimensions (vertices, edges, faces, cells) be present in the memory mesh representation. However, usual mesh generation algorithms or mesh file readers result in a mesh with cells and vertices only, while edges and faces need to be inferred at runtime. This is accomplished by the so-called topological interpolation.

Topological Interpolation is the process of constructing intermediate levels of the ranked poset describing a mesh, given information at bracketing levels. For example, if we receive triangles and their covering vertices, as in Figure 2, interpolation will construct edges. The first algorithm for interpolation on the Hasse diagram was published in [38], but this version is only appropriate for simplices, ignores orientation of the mesh points, and did not give a complexity bound.

The interpolation procedure selects a given point stratum as cells, for which it will construct facets. It iterates over the cells, whose cones are an oriented set of vertices. The two essential operations are to extract an oriented facet from lower dimensional



(a) original 2D mesh

(b) plex representation of (a), having 3 strata

(c) coloring of `Stratum`(h), $h = 0, 1, 2$, throughout the paper

(d) cone $C(0)$, its transitive closure $C^+(0)$

(e) support $S(5)$, its transitive closure $S^+(5)$

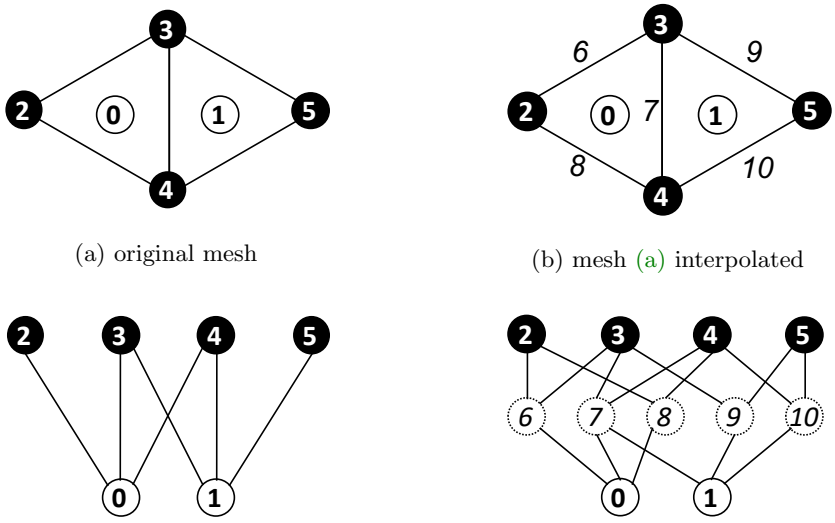
Fig. 1: DMPlex mesh representation and basic relations. A 1D mesh would have 2 strata: $h = 0$ edges, $h = 1$ vertices. A 3D mesh would have 4 strata: $h = 0$ cells, $h = 1$ faces, $h = 2$ edges, $h = 3$ vertices. Note we can also refer to the $h = 1$ entities commonly, in a dimension-independent way, as *facets*, and $h = 0$ as *cells*. The entities at $h = d$, where d is the topological dimension, are always vertices.

points, and then attach it to a cell with the correct orientation. Orientation of mesh points is detailed in subsection 2.3. In order to enumerate the facets for a given cell type, we have `DMPlexGetRawFaces_Internal()` which returns an oriented list of vertices for each facet of the input cell.

An initial iteration over cells constructs all facets, and enters them into a hash table, where the hash key is the sorted list of vertices in each face. We need one pass for each type of face. Once the hash table is constructed, we know the number of new facets to be inserted, and can allocate a new plex. The new plex is identical to the old, except that it has a new face stratum, and the cone sizes of cells need to be calculated anew. For example, hexahderal cells have 8 vertices, but 6 facets, so that cone size would change from 8 to 6.

A second iteration over cells inserts the facets. We clear the hash table and repeat the face extraction above. If a face is missing from the table, we insert it into the table, record its cone, and also insert it into the cell cone with default orientation. If instead it is present in the table, we insert it into the cell cone with orientation computed from comparing the face cone with that returned from `DMPlexGetRawFaces_Internal()`. Again, we need one pass for each face type.

The complexity to interpolate a given stratum is in $\mathcal{O}(N_C N_F N_T)$, where N_C is the



(a) original mesh

(b) mesh (a) interpolated

(c) plex representation of (a)

(d) plex representation of (b)

Fig. 2: Sequential topological interpolation: original mesh and interpolated mesh in classical and plex representation.

number of cells, N_F is the maximum number of faces per cell, and N_T is the number of used face types. However, N_F and N_T are obviously constant, so $\mathcal{O}(N_C N_F N_T) = \mathcal{O}(N_C)$. The $\mathcal{O}$-complexity remains the same even if we sum all strata because the size of (number of points in) stratum $h = k + 1$ is a certain multiple of the size of stratum $h = k$ for any feasible height k. We can conclude that the complexity is linear with respect to the mesh size.

2.3. Orienting edges and faces. Let us focus on two particular points in the interpolated mesh in Figure 2(b): cell 0, and edge $7 \in C(0)$. Their cones are $C(0) = \{6, 7, 8\}$ and $C(7) = \{3, 4\}$, respectively. We have so far spoken about *which* points form $C(p)$ for given p. However, the orientation of p is also important; for instance, to have a well-defined direction of an outer normal, or to assign field values correctly during simulation. The orientation of p is represented as the order of points in $C(p)$. Hence, e.g. $C(0)$ is rather a tuple, $C(0) = (6, 7, 8)$. This is how the array implementing $C(0)$ is stored. Let us from now denote by $C(p, c)$ the c-th point in this tuple, $c = 0, 1, \ldots, n - 1$, where $n = \text{size}(C(p))$.

For consistency, the orientation of points in plex should be in line with the orientation of their supporting points. There is never a problem for the lowermost and uppermost stratum; cells have no supporting points, and vertices have no orientation. However, for the intermediate strata (i.e. edges and faces computed by interpolation), we can get a conflicting situation as depicted in Figure 3. Edge 7 points against the direction of cell 1 but if we flip it, it will point against the direction of cell 0. Thus we need a mechanism to allow for this.

In general, suppose point $p \in \texttt{Stratum}(0)$, its cone point $q = C(p, c) \in C(p) \subset \texttt{Stratum}(1)$, and $C(q) \subset \texttt{Stratum}(2)$. For example, in Figures 3(b) and 3(d), $p = 1$, $c = 0$, $q = 7$, $C(q) = (3, 4)$. To compensate the given (stored) order of $C(q)$, additional

information about the proper interpretation order needs to be defined. This information must be attached to the edge $(p,\ q)$ in the DAG because it varies for different choices of p even for the same q. Naturally, the order is not completely arbitrary; it can be described by (1) the starting point index $S(p,c) \geq 0$ in 0-based local numbering with respect to q, and (2) the direction $D(p,c) \in \{-1,1\}$ (reverse/forward). These two can be represented by a single signed integer $O(p,c)$: $S(p,c)$ by its magnitude, and $D(p,c)$ by its sign. Since the sign is undefined for 0, the negative values are shifted by -1. To summarize,

$$O(p,c) = \begin{cases} S(p,c), & D(p,c) = 1, \\ -S(p,c) - 1, & D(p,c) = -1, \end{cases} \tag{2.1}$$

and the other way around,

$$D(p,c) = \begin{cases} 1, & O(p,c) \geq 0, \\ -1, & O(p,c) < 0, \end{cases} \tag{2.2}$$

$$S(p,c) = \begin{cases} O(p,c), & O(p,c) \geq 0, \\ -O(p,c) - 1, & O(p,c) < 0. \end{cases} \tag{2.3}$$

Note that for $C(p,c)$ being an edge, flipping the orientation of the edge implies changing the starting point, so $O(p,c) \in \{0,-2\}$ only. We can also define the whole tuple of orientations for point p,

$$O(p) = (O(p,0), \ldots, O(p,n-1)), \tag{2.4}$$

where $n = \text{size}(C(p))$. This $O(p)$ is attached to every plex point p in the same way as $C(p)$. We note that $O(p)$ is just a numbering for the elements of the dihedral group for a given face.

3. Parallel simulation startup.

Let us call *startup phase* all steps necessary to load the mesh from disk storage and prepare it for use in the simulation time loop. It consists of the following steps: (1) raw data loading, (2) plex construction, (3) topological interpolation, (4) distribution.

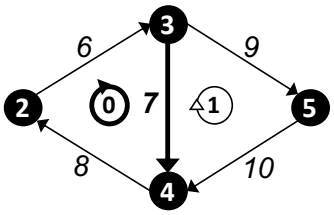
Before the developments of this paper, these steps were serial and only the last step, a one-to-all distribution using a serial partitioner such as METIS [28, 30], resulted in the distributed mesh. This approach inevitably led to the upper limit on the mesh size due to the memory constraints of a single node of a cluster. Therefore we developed a new, completely parallel startup phase where all four steps are done in parallel right from the beginning. This parallel startup phase is schematically depicted in Figure 4. We further show that even for meshes that fit into memory, parallel startup can bring significant time and energy saving. Let us describe these stages more in detail in the following subsections.

3.1. Raw data loading.

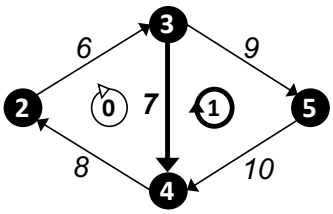
This stage forms the first part of our mesh reader implementation, and consists in reading distributed raw topology and geometry data by generic index set and vector readers, dominated by low level I/O operations.

3.1.1. HDF5.

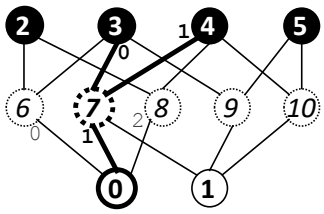
Hierarchical Data Format 5 (HDF5) [22] is a file format and library, designed to store and organize large amounts of N-dimensional array data. It is currently supported by the HDF Group, a not-for-profit corporation whose mission is to ensure continued development of HDF5 technologies and the continued accessibility of data stored in HDF.



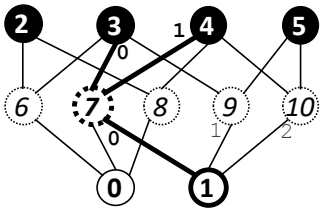
(a) edge 7 in cone of face 0, $C(0,1) = 7$



(b) edge 7 in cone of face 1, $C(1,0) = 7$



(c) the starting point of edge 7 within face 0 is 3; $C(C(0,1), S(0,1)) = C(7,0) = 3$, $O(0,1) = 0$



(d) the starting point of edge 7 within face 1 is 4; $C(C(1,0), S(1,0)) = C(7,1) = 4$, $O(1,0) = -2$

Fig. 3: Mesh from Figures 2(b) and 2(d) with cone points order and orientation. We focus here on the edge 7 within the cones of faces 0 and 1.

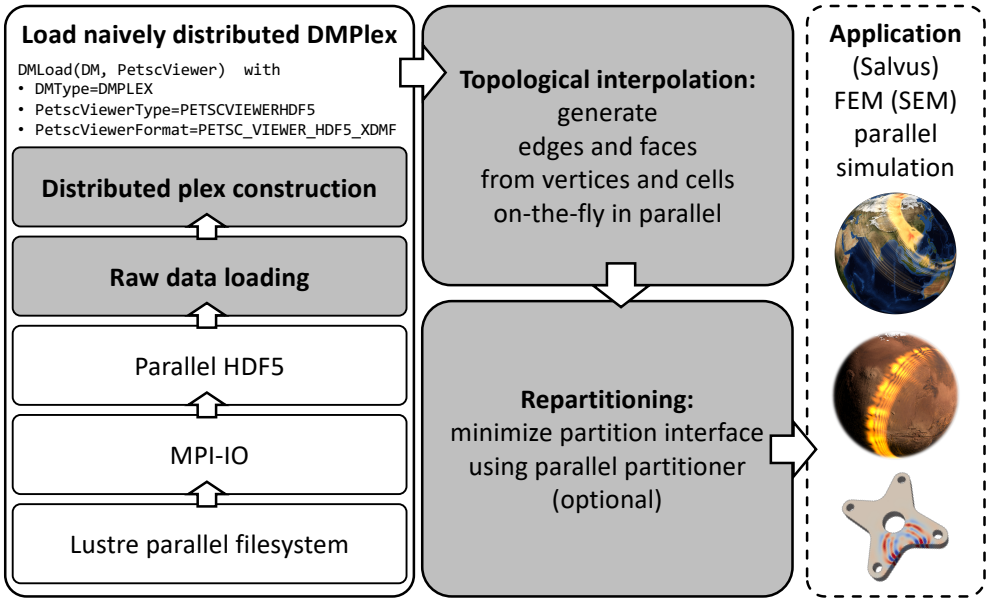


Fig. 4: Schematic diagram of the parallel simulation startup. Grey stages are within PETSc scope.

HDF5's file structure includes two major types of objects: (1) datasets, multidimensional arrays of a homogeneous type; (2) groups, container structures which can hold datasets and other groups. Every HDF5 file has a root group `/`, under which one can add additional groups and datasets. This results in a hierarchical, filesystem-like data format. Resources in an HDF5 file can be accessed using the POSIX-like syntax `/group1/group2/dataset`. Metadata is stored in the form of user-defined, named attributes attached to groups and datasets [22].

HDF5 transparently handles how all the objects map to the actual bytes in the file. HDF5 actually provides an abstracted filesystem-within-a-file that is portable to any system with the HDF5 library installed, regardless of the underlying storage type, filesystem, or endianess. It does automatic conversions between storage datatypes (dictated by the data file) and runtime memory datatypes (dictated by the application) [22].

HDF5 supports parallel shared-file I/O using MPI-IO [40] capabilities which in turn provide scalable access to the underlying parallel filesystem such as Lustre [42]. By default, HDF5 provides uniform access to all parts of the file for all processes of the communicator (passed to HDF5 using `H5Pset_fapl_mpio()`). However, since PETSc uses data parallelism, it would be very inefficient to load all data to all processes and then distribute them again. The most important functionality in this regard are *hyperslabs* which can read or write to a portion of a dataset. A hyperslab can be a logically contiguous collection of points in a dataspace, or a regular pattern of points or blocks in a dataspace. The hyperslab can select a separate chunk of the file for each process individually by means of a rank-dependent offset. The `H5Sselect_hyperslab()` function is used for this purpose [23].

3.1.2. XDMF. XDMF (eXtensible Data Model and Format) [50] is a mesh data file format. It distinguishes the metadata (light data) and the values themselves (heavy data). Light data and heavy data are stored using XML and HDF5, respectively. The data format is stored redundantly in both XML and HDF5. There are two crucial datasets describing the mesh in a minimal sufficient way: (1) `<Geometry>`, a 2D dataset where each row contains coordinates of a vertex (2 or 3 scalars based on dimensionality); (2) `<Topology>`, a 2D dataset where each row represents a cell, listing indices of all incident vertices. Each vertex index in `<Topology>` corresponds to a row index within `<Geometry>`). Both these datasets can be defined within the XML file as plain text, or refer to a dataset path within the standalone HDF5 file (e.g. `MyData.h5:/geometry/vertices`). Both ways can be mixed within the same XDMF file, and are both supported by widely used visualization programs such as ParaView, VisIt and EnSight. Nevertheless, the former way is advisable only for small datasets. We always store all data in the HDF5 file and use the XDMF file only as a descriptor. We will refer to this as HDF5/XDMF format.

3.1.3. PETSc data loading. PETSc contains a class, called `PetscViewer`, designated for all I/O of any `PetscObject` such as a vector, matrix, linear solver. The source/destination is dictated by the type (`PetscViewerType`) such as `ascii`, `binary`, `hdf5`, `socket` and others. A more fine-grained control of how the object is read/viewed is accomplished with the viewer format (`PetscViewerFormat`); e.g. `ascii_info` and `ascii_info_detail` print plain text information about the object with a different level of verbosity.

Most relevant for this work is that `DM` supports both reading and writing with `PetscViewer`, using `DMLoad(DM,PetscViewer)` and `DMView(DM,PetscViewer)`, respectively. Recently, we have implemented a new `PetscViewerFormat hdf5_xdmf`

for the `PetscViewerType hdf5` to enable parallel reading and writing of unstructured meshes represented in memory as plex (`DM` with `DMType plex`) and stored in an HDF5 file with topology and geometry data compatible with XDMF (see subsections 3.1.1 and 3.1.2). HDF5/XDMF has become the first widely used mesh format supported by PETSc which is both readable and writable in parallel.

As we are here interested mainly in the simulation startup phase, we will now focus only on `DMLoad()`, namely its implementation for the combination of `DMType`, `PetscViewerType` and `PetscViewerFormat` described above. This implementation relies on lower level readers for integer vectors (index sets, `IS`) and scalar vectors (`Vec`): `ISLoad()` and `VecLoad()`. They both read datasets from given paths within the HDF5 file (see subsection 3.1.1) using the `H5Dread()` routine but with different HDF5 datatypes (`H5T_NATIVE_INT` and `H5T_NATIVE_DOUBLE`, respectively). They make use of a hyperslab (see subsection 3.1.1) reflecting the given parallel layout (`PetscLayout`) to divide the global dataset into local chunks along the first dimension. The layout is either specified by user, or calculated automatically so that the chunks' lengths differ by 1 at most. The second dimension of the dataset is interpreted as a block size, i.e., the resulting vector is divided into equally sized shorter blocks. Blocks can have various contextual meanings such as DOFs of the same element.

Within `DMLoad()`, `VecLoad()` loads the geometry information (`<Geometry>` in subsection 3.1.2) into a `Vec`, whose blocks and entries represent vertices and their coordinates, respectively. The size of all blocks is the same and corresponds to the spatial dimension of the mesh, and global indices of the blocks form an implicit global vertex numbering.

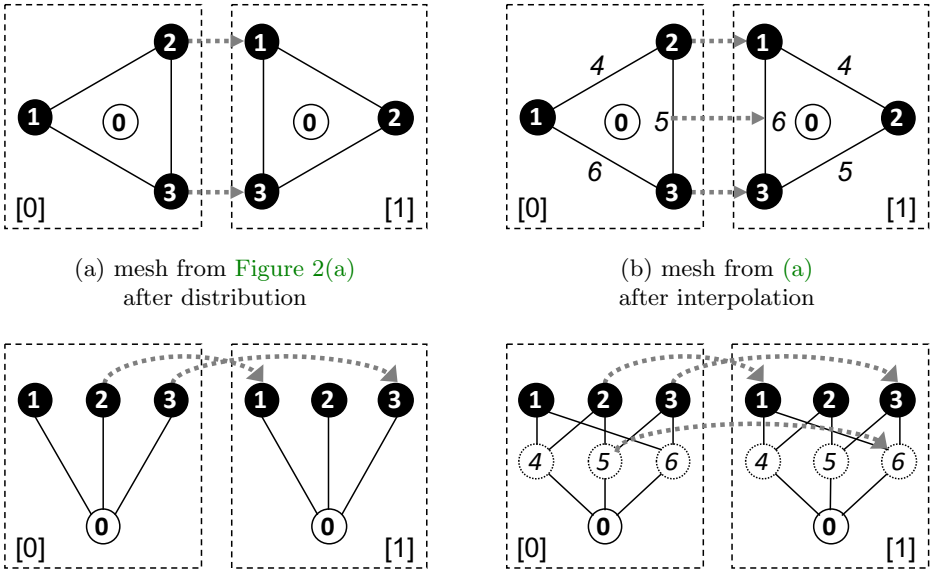
`ISLoad()` then loads the cell-vertex topology information which is represented in XDMF by `<Topology>` (subsection 3.1.2). Each `IS` block corresponds to an element, and each single entry refers to one of this element's vertices using the implicit global vertex numbering. Global indices of the blocks form an implicit global cell numbering.

When we read such representation in parallel, all processes load approximately equally sized, contiguously numbered, disjoint portions of vertices and cells in a single parallel I/O operation. Note that the global vertex and cell numberings are not affected by parallel loading on any number of processes.

3.2. Distributed plex construction. The raw topology and geometry data loaded in subsection 3.1 need to be transformed into a plex representation. This forms the second part of `DMLoad()`, and is realized by a call to a communication-bounded operation described in detail in subsection 3.2.2. The resulting `DMPlex` instance is naively distributed with vertices and cells only. Let us first describe a parallel star forest graph implementation, which allows gluing together serial plexes across different processes.

3.2.1. Star Forest. The Star Forest (SF, in PETSc called `PetscSF` [4, 8]) is a forest of star graphs. The *root* of each star graph corresponds to something owned by a process, such as a mesh point or solution degree-of-freedom (DOF). The *leaves* of each star graph are the shared versions of that point or DOF on other processes, or what is often called ghost points. In particular, the SF is a map from local points p to remote points (r, q), where r is the remote rank and q is the local number for the point on process r. Since the SF only deals with local numberings, not global numberings, parallel mesh modification is much easier. Moreover, SF accepts data buffers with arbitrary MPI types for its communication routines.

SF supports a broadcast operation from roots to leaves, as well as a reduction from leaves to roots. In addition, it supports gather and scatter operations over the



(a) mesh from Figure 2(a) after distribution

(b) mesh from (a) after interpolation

(c) plex representation of (a)

(d) plex representation of (b)

Fig. 5: Parallel DMPlex. Grey dotted arrows denote `sfPoint`.

roots, inversion of the SF to get two-sided information, and a fetch-and-op atomic update. Plex can build a hierarchy of SF objects describing the overlap of partitions, point set, topology, data layout, and Jacobian information [33].

3.2.2. Construction of distributed plex from raw data. Once the raw topology and geometry data is loaded as described in subsection 3.1.3, vertex coordinates are initially distributed independently of the cells, with process p getting some $N_V^{(p)} \times d$ vertex coordinates. For example, if we have 20 vertices in 3D and 4 processes, each process might get 15 doubles, namely the coordinates for 5 contiguously numbered vertices, regardless of the cells it has been assigned.

In order to construct a parallel plex from these vertex and cell portions, we employ `DMPlexCreateFromCellListParallel()`. From the cell-vertex topology information, it determines how many unique local vertices a process owns using a hash table of the vertex numbers. The cone information for a local plex is constructed by translating the global vertex numbers to local vertex numbers, using the index of each local vertex in the hash table. Once we have the local cone information, we symmetrize this to get support information (`DMPlexSymmetrize()`), and construct strata labels (`DMPlexStratify()`). Both these operations have linear complexity.

Given this hash table of local vertices and the global division, we construct an SF mapping the initial vertex division to the local division, `sfVert`. With this SF, we broadcast coordinates from the initial distribution roots to the local distribution leaves, determined by the mesh topology. In addition, we can use the initial SF to construct an SF describing the sharing of local vertices between processes, `sfPoint`. We first construct an array which holds (r, l) for each local vertex. Then we reduce this array to the roots using `sfVert`, taking the maximum over ranks. This information

is then broadcast back to the leaves, giving a unique owner for each vertex, allowing us to construct `sfPoint`.

The resulting distributed plex object has partition-wise complete topological information. Each partition is a serial plex object which includes all incident vertices of its elements. These serial plex objects are combined together by the parallel SF object `sfPoint` as shown in Figure 5.

3.3. Parallel topological interpolation. Since the steps above lead to a distributed plex, we need a parallel version of the topological interpolation (subsection 2.2). This step appeared to be the most challenging one. It consists of the serial interpolation (in-memory computations) and a small communication.

The first step consists in applying the sequential topological interpolation (subsection 2.2) and cone orientation (subsection 2.3) on each rank independently. Then we must alter the `PetscSF` structure which identifies mesh points which are owned by different processes, or *leaf* points. The SF structure is described in subsection 3.2.1. We mark all leaf points which are adjacent to another ghost point as candidates. These candidate points are then gathered to *root* point owners (using `PetscSFBcast()`). For each candidate, the root checks for each point in the cone that either it owns that point in the SF or it is a local point. If so, it claims ownership. These claims are again broadcast, allowing a new SF to be created incorporating the new edges and faces.

The cone orientation has been done on each rank independently, and hence it is only partition-wise correct. However, we have not yet handled the following assumption: If interface edges/faces owned by different ranks represent the same geometrical entity, i.e., they are connected by `pointSF` like edges [0]5 and [1]6 in Figure 6, they must have a conforming order of cone points ($[r]$ means ownership by rank r). This requirement can be written more rigorously as an implication

$$\left.\begin{array}{ll} p_0 & \to p_1 \\ C(p_0) & = (q_{0,0}, \dots, q_{0,n-1}) \\ C(p_1) & = (q_{1,0}, \dots, q_{1,n-1}) \end{array}\right\} \Rightarrow \begin{cases} q_{0,0} & \to q_{1,0} \\ & \cdots \\ q_{0,n-1} & \to q_{1,n-1}, \end{cases} \tag{3.1}$$

using notation from subsection 2.3 and relation $\to$ meaning a connection via `pointSF`.

In Figures 6(a) and 6(c) this assumption is violated for the edges [0]5 and [1]6. They are flipped to each other, more rigorously speaking `pointSF` connects the edge and its incident vertices

$$[0]5 \to [1]6, \qquad [0]2 \to [1]1, \qquad [0]3 \to [1]3,$$

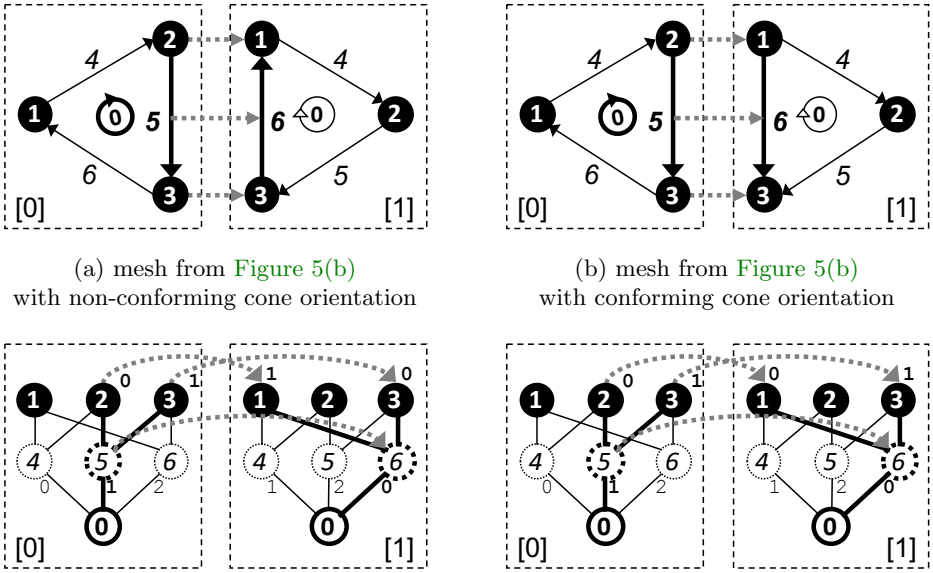
but the order of cone points does not conform,

$$[0]2 = C([0]5, 0) \not\to C([1]6, 0) = [1]3,$$
$$[0]3 = C([0]5, 1) \not\to C([1]6, 1) = [1]1.$$

This would lead to incorrect PDE solution if the used discretization method makes use of the edge.

In order to satisfy this requirement, and additional synchronization of the interface cones must be carried out. We start by synchronization of the interface cone point numbering. Let us remind that `pointSF` is a one-sided structure, so only the origins of the arrows can be found directly.

Let us assume rank r_0, its edge/face $[r_0]p$, and that there is a `pointSF` arrow pointing *from* $[r_0]p$ to some $[r_1]p$ If we detect an arrow directed *from* the cone



(a) mesh from Figure 5(b) with non-conforming cone orientation

(b) mesh from Figure 5(b) with conforming cone orientation

(c) plex representation of (a); $O([1]0,0) = 0$

(d) plex representation of (b); $O([1]0,0) = -2$

Fig. 6: Parallel DMPlex with cone points order and orientation.

point $C([r_0]p, c) \to [r_1]q_c$, we set $\texttt{root}([r_0]p, c) = (r_1, q_c)$, otherwise $\texttt{root}([r_0]p, c) = (r_0, C([r_0]p, c))$. This $\texttt{root}([r_0]p, c)$ is sent to r_1 using `PetscSFBcastBegin/End()`, and stored at the destination rank as $\texttt{leaf}([r_1]p, c)$. This is done for each rank, each point in $\texttt{Stratum}(h)$, $h > 0$, and each $c = 0, 1$.

Now from the rank r_1 view, it has for $c = 0, 1$ point $[r_1]p$, $\texttt{root}([r_1]p, c)$ and the received $\texttt{leaf}([r_1]p, c)$. If $\texttt{root}([r_1]p, c) = \texttt{leaf}([r_1]p, c)$ does not hold for both $c = 0, 1$, we must rotate and/or flip the cone so that this condition gets satisfied. In that case we must also update $O(s)$ for all $s \in S([r_1]p)$ accordingly to compensate the change of cone order.

We can see that the orientation synchronization heavily relies on the correct `pointSF`. This is why it must be processed first.

3.3.1. Redistribution. We now have a correct distributed `DMPlex` instance representing all codimensions. However, the distribution is naive; it is load balanced with respect to the size of partitions but the partition shape is not optimized. It can optionally be further improved using a parallel partitioner to minimize the partition interfaces and hence reduce the halo communication in the subsequent application computation. PETSc offers interfaces to ParMETIS [29, 30, 31] and PT-Scotch [15]. For this paper, we always used ParMETIS. We will not discuss this stage further because it is not critically needed to overcome the single node memory limit (our main motivation), its cost-effectiveness depends on the application, and it was implemented in PETSc beforehand.

4. Seismic wave propagation modeling. As a representative use case and benchmarking tool for the new parallel simulation startup phase described above in

section 3, we use an implementation of the spectral-element method (SEM).

Although originally developed for applications in fluid dynamics [43], continuous-Galerkin SEM on hexahedral elements has emerged as the de-facto standard for global-scale seismic wave simulations [44, 18, 19]. SEM is a high-order finite-element method with very low dispersion and dissipation errors [18]. The choice of the Gauss-Lobatto-Legendre collocation points for the interpolating Lagrange polynomials naturally yields a diagonal mass matrix, which enables the use of explicit time stepping schemes. A second-order Newmark time-stepping additionally allows us to compute coupling terms along any solid-fluid interfaces without the need to solve a linear system [41]. Furthermore, the tensorized structure of the finite element basis on hexahedral elements allows for efficient computations of internal forces. These element-wise operations can be formulated as dense matrix–matrix products, making the method suitable for the current generation of SIMD computing architectures.

Salvus [1] contains a flexible implementation of SEM, separating the wave propagation physics, the spatial discretization and finite-element shape mappings into distinct and functionally orthogonal components. It uses modern C++ features to ensure that and this does not affect runtime performance. It is parallelized using MPI and GPU-accelerated with CUDA. PETSc DMPlex (section 2) is used for mesh management so the developments of this paper can be directly applied.

5. Performance results.

This section presents scalability tests of the new parallel simulation startup phase (section 3) used within seismic wave propagation modeling (section 4).

5.1. Hardware.

All benchmarks were run at Piz Daint, the flagship system of the Swiss National Supercomputing Centre (CSCS). Piz Daint consists of 5704 12-core Cray XC50 nodes with GPU accelerators, and 1813 36-core Cray XC40 nodes without accelerators. All benchmarks presented in this paper ran on the XC50 nodes. Each of them is equipped with one 12-core Intel Xeon E5-2690 v3 (Haswell) processor, one NVIDIA Tesla P100 16GB GPGPU accelerator and 64 GB RAM. Piz Daint has 8.8 PB shared scratch storage with the peak performance of 112 GiB/s. It is implemented using Cray Sonexion 3000 [16] scale-out storage system equipped with the Lustre parallel file system [42], 40 object storage targets (OSTs) and 2 metadata servers (MDSs).

5.2. Middleware (Lustre, MPI-IO, HDF5) settings.

We always used the single shared file approach, i.e., every process reads its disjoint chunk from the common file. There are many good reasons for such choice, such as reduction of metadata accesses, but the main reason is the flexibility in number of processes using the same file, unlike the file-per-process approach. As for Lustre file system settings, we used stripe count of 40 (maximum on Piz Daint) and stripe size of 16 MB. Regarding HDF5/MPI-IO, we always non-collective reading. We tested also collective reading with various numbers of aggregators (MPI-IO hint `cb_nodes`) but never saw any significant benefit. The raw file reading always took less than 2 seconds; we cannot exclude that some scenarios with much bigger files and/or node counts could require more deliberate settings but such scenarios are irrelevant within the context of this paper.

5.3. Cube benchmark.

Our performance benchmark consists in elastic wave propagation in homogeneous isotropic media from a point source in a cubic geometry. The cube is discretized into equally sized hexahedral cells, handled as an unstructured mesh. Each cell hosts a 4-th order spectral element with 125 spatial DOFs. Since

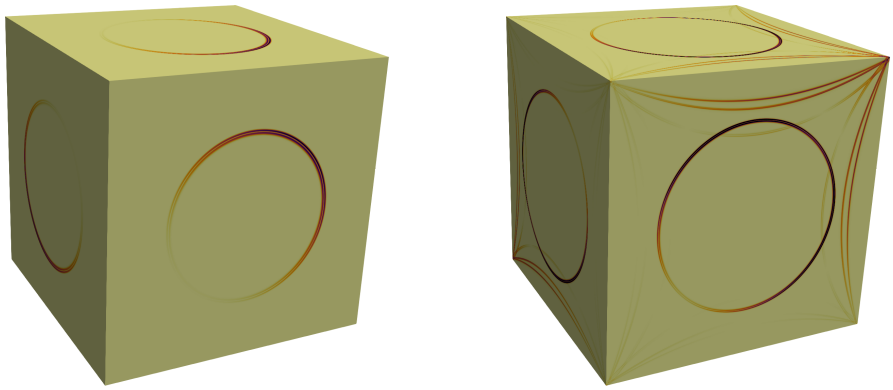
a 3-D vector equation was solved, this resulted in 1125 field variables per element, together representing acceleration, velocity, and displacement. Figure 7 illustrates the solution of the benchmark problem at two different timesteps. Table 1 summarizes dimensions of the stored topology and geometry datasets and resulting file sizes for different numbers of elements in x-direction (NEX). Our sequence of NEX was chosen so that the total number of elements (NE) of each successive mesh is approximately doubled, starting at 8 million.

We present performance of the new parallel startup phase in several graphs. Graphs in Figure 8 present strong scalability for the mesh size of 16 million elements, and serve mainly for comparison of the new parallel startup with the original serial startup. 16 million is an upper bound for the mesh size for the serial startup imposed by the memory of a single Piz Daint Cray XC50 node. Overcoming this limit is for us the most important achievement of the startup phase parallelization. However, obviously the performance improvement is very significant as well. At 1024 nodes, the serial startup takes an amount of wall time equivalent to 202'886 timesteps in the subsequent time loop, whereas the parallel startup takes only 1827 timesteps, which means 111x speedup. The number of timesteps in production simulations varies, but generally 30'000 or more are required.

Graphs in Figure 9 are similar to Figure 8 but show the only the parallel startup scalability for various mesh sizes, so that the displayed time can be limited to 70 seconds. These graph gather all stages in a single graph and different mesh sizes are presented separately. By contrast, graphs in Figure 10 show strong and weak scalability for each stage separately, gathering all mesh sizes in a single graph. We can see that the topological interpolation scales almost perfectly and becomes insignificant for high number of nodes even for very large meshes. The other stages do not scale that well; however, their absolute wall times are rather small for the mesh sizes and node counts of interest. The scalability of the significant redistribution stage breaks at about 256 nodes and the mesh size no more dictates the wall time. ParMetis was used with default settings; there might be some space for slight improvement by tuning its parameters but in general it is well known that current graph partitioners do not scale beyond 10'000 cores. We can hardly do anything about it apart from perhaps testing alternative approaches such as space-filling curves. There might be some space for improvement left for the distributed plex construction; nevertheless, from the stagnation between 256 and 1024 nodes for the largest mesh we conclude that such optimization is probably not worth the effort, at least for now.

6. Application: seismicity on Mars. In late 2018, the NASA InSight mission [6] placed a highly sensitive seismometer [39] on Mars' surface and recorded the first seismic signals ever observed on Mars [21]. The observation of seismic waves is a crucial source of information to investigate the interior structure and composition of Mars. However, as the data shows significant differences to seismic data from both Moon and Earth, numerical simulations of seismic wave propagation on Mars that account for topography as well as 3D scattering due to lateral variations of the material parameters are key to assist the interpretation of the observational data.

Full waveform simulations are essential to constrain the planet's structure using data in the frequency band recorded by the probe. The seismic response to marsquakes or asteroid impacts is governed by a coupled system of the elastic/acoustic wave equation, which models seismic waves propagating through Mars' mantle and the liquid core, respectively. This can be simulated efficiently using the Spectral Element Method (SEM).



(a) $t = 0.1$ s (b) $t = 0.15$ s

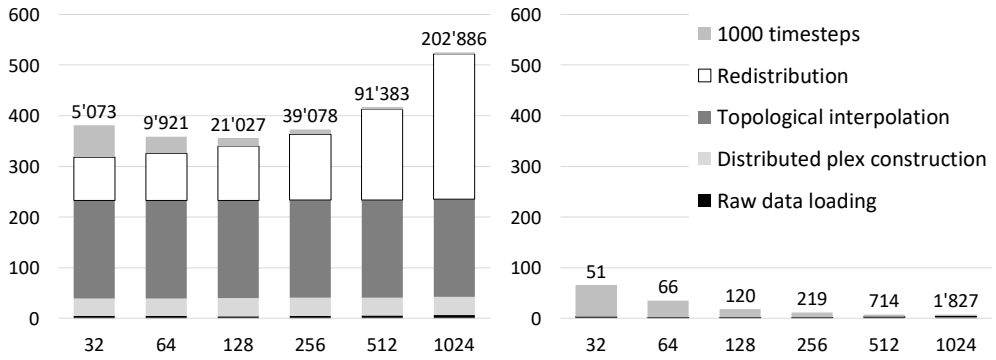
Fig. 7: Cube benchmark with a moment-tensor type source located at the center of the domain. Isotropic elastic material model without attenuation was applied. The mesh size was 256M elements, and 512 Piz Daint nodes (6144 CPUs, 512 GPUs) were used for the computation. The normalized magnitude of the velocity vector at time t is visualized. We can see the P-wave propagating from the source in (a), while (b) shows the S-wave and reflections of the P-wave from the cube boundary.

	topology (int64)		geometry (double)		
NEX	rows = NE = $(\text{NEX})^3$	columns	rows	columns	file size (GB)
200	8'000'000	8	8'120'601	3	0.66
252	16'003'008	8	16'194'277	3	1.32
318	32'157'432	8	32'461'759	3	2.64
400	64'000'000	8	64'481'201	3	5.26
504	128'024'064	8	128'787'625	3	10.51
635	256'047'875	8	257'259'456	3	21.01

Table 1: Cube benchmark datafiles: number of elements in x-direction (NEX); total number of elements (NE); topology (connectivity) and geometry (vertices) dataset sizes; file sizes. Both topology (integer numbers) and geometry (real numbers) are stored with 64-bit precision.

For the computation of the seismic response of Mars, we rely on Salvus' implementation of the SEM (section 4). Salvus' internal mesher uses custom algorithms to generate fully unstructured conforming 3D hexahedral meshes [49], efficiently representing topography and the extreme crustal thickness variations of Mars (∼5–120 km), see Figure 11. The solver then represents these meshes in memory using DMPlex (section 2).

The CFL condition for Salvus' explicit second-order Newmark time-stepping scheme, coupled with a required minimum number of points-per-wavelength in each



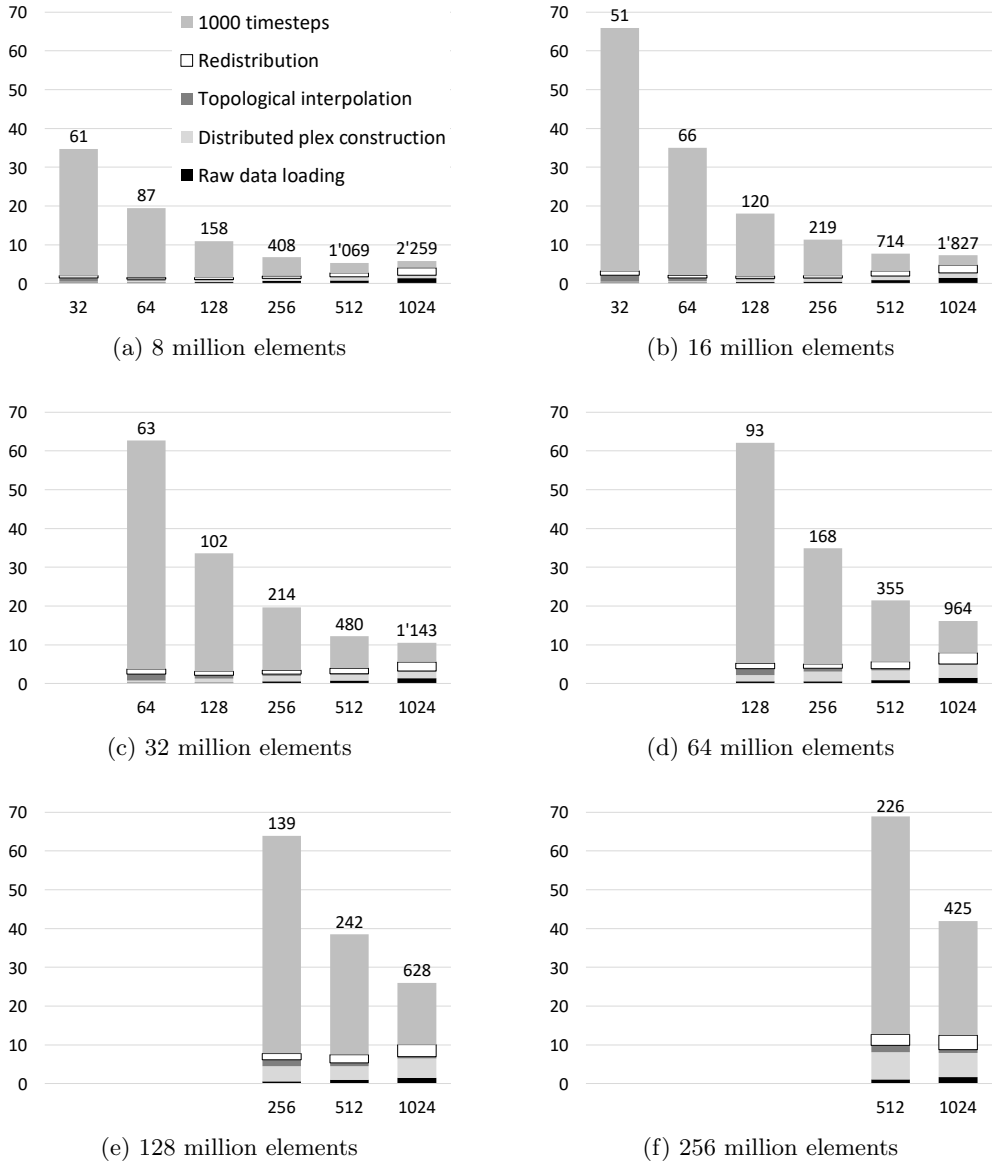
(a) serial startup (b) parallel startup

Fig. 8: **Cube benchmark. Strong scalability, serial/parallel startup, 16 million mesh elements.** The serial and parallel simulation startup phase are compared with each other and 1000 steps of the Salvus time loop in terms of wall time. Approximate wall time for any different number of timesteps can be obtained using simple proportionality. The number of timesteps in production simulations varies, but generally 30'000 or more are required. The particular startup stages are described in section 3. **X-axis**: number of Piz Daint nodes, each equipped with 12 cores and 1 GPU per node. **Y-axis**: wall time in seconds. **Labels above bars**: the number of timesteps that take the same wall time as the startup phase. **Order of colors** in the bars is the same as in the legend.

dimension, results in the computational complexity of a simulation scaling with frequency to the power of 4. When using 4-th order spectral elements, which is common for planetary-scale wave propagation, more than 6 grid points per shortest wavelength are needed to accurately resolve seismic waves [19]. As the quakes are small in magnitude and the noise level increases at low frequencies, large-scale simulations are required to reach the parameter regime of the observations, and the required number of spectral elements can easily reach hundreds of millions.

Such mesh sizes necessitate the parallel simulation startup presented in section 3. Prior to these developments, the largest possible mesh size was limited by the available memory of a single Piz Daint node to approximately 16 million elements. Moreover, loading a mesh of such size took more than four minutes. With the parallel startup in hand, these limitations vanish.

Figure 11(d) shows a snapshot of the surface displacement resulting from a simulation of a hypothetical quake on Mars. Here, the discretized coupled elastic wave equation has approximately 124 million 4-th order spectral elements, and we compute 100'455 timesteps representing a simulated time of 30 minutes. Each element has 125 spatial DOFs, each hosting 9 dynamic field components (vector displacement, velocity, and acceleration). These parameters lead to an unprecedented resolved period of 3.2 s. Using again all 12 cores per Piz Daint node as well as the attached Tesla P100 GPU, this simulation took approximately 2.4 hours of wall time on 256 Piz Daint nodes. From this total wall time, raw data loading took 0.9 s, distributed plex construction 4.5 s, topological interpolation 3.1 s and redistribution 2.3 s, i.e., the whole startup phase took less than 11 s.



(a) 8 million elements

(b) 16 million elements

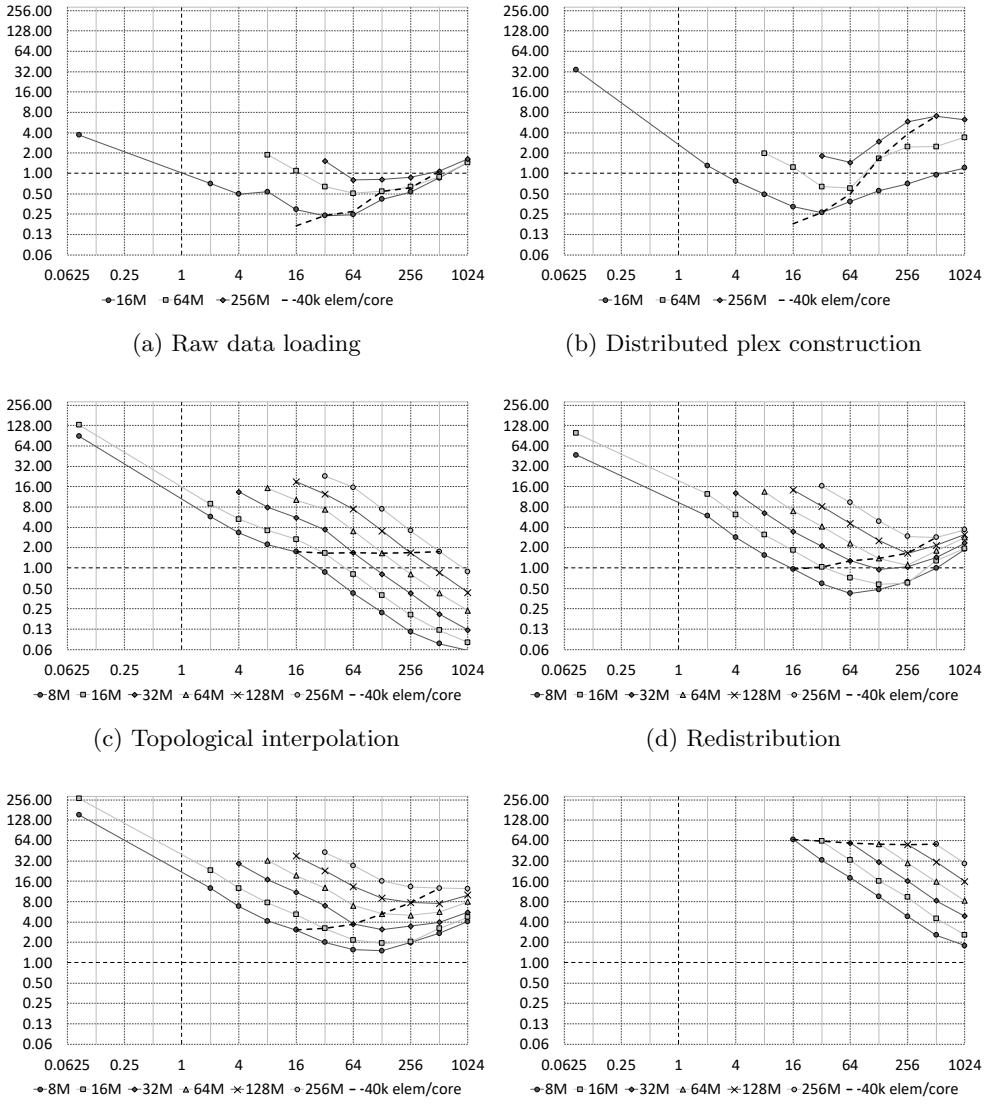
(c) 32 million elements

(d) 64 million elements

(e) 128 million elements

(f) 256 million elements

Fig. 9: **Cube benchmark. Strong scalability, parallel startup, various mesh sizes.** The parallel simulation startup phase is compared with 1000 steps of the Salvus time loop in terms of wall time. Approximate wall time for any number of timesteps can be obtained using simple proportionality. The number of timesteps in production simulations varies, but generally 30'000 or more are required. The particular stages are described in section 3. **X-axis**: number of Piz Daint nodes, each with 12 cores and 1 GPU. **Y-axis:** wall time in seconds. **Labels above bars:** the number of timesteps that take the same wall time as the startup phase. **Missing bars:** out of memory failure during the time loop caused by the memory limit of the GPUs. **Order of colors** in the bars is the same as in the legend. **Note:** (a) is the same as Figure 8(b) but with a re-scaled time axis.



(a) Raw data loading

(b) Distributed plex construction

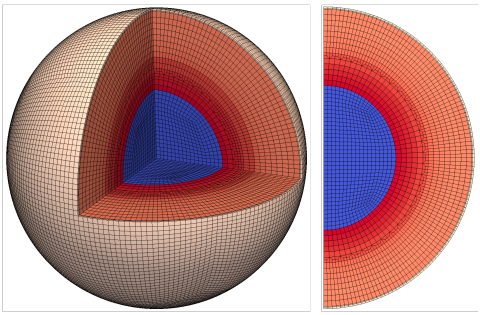
(c) Topological interpolation

(d) Redistribution

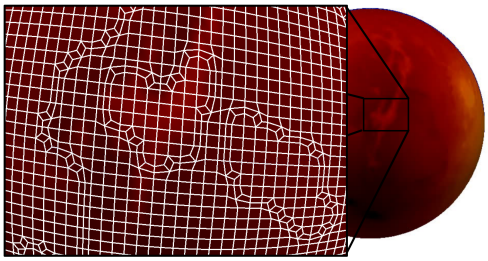
(e) Total startup

(f) 1000 Salvus timesteps

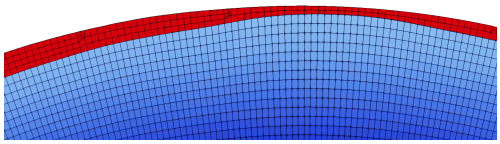
Fig. 10: **Cube benchmark. Strong and weak scalability per stage. X-axis**: number of Piz Daint nodes (log2 scale), each with 12 cores and 1 GPU; $0.0625 = 1/12$ means a single core (serial) run. Note that a single node run was not possible due to the out-of-memory failure during the redistribution phase. **Y-axis:** wall time in seconds (log2 scale). **Solid lines** show the strong scalability for different mesh sizes. **Dashed line** shows the weak scalability for the mesh size of approximately 40'000 elements per core which is the upper bound for the Salvus timeloop imposed by limited GPU memory. **Order of line styles** is the same in the plots and in the legends.



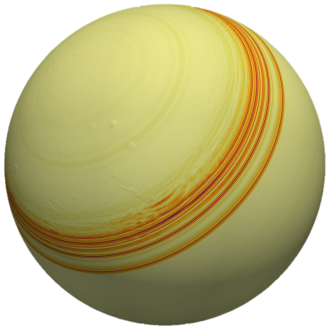
(a) cubed sphere with fluid core (blue) and solid mantle (red)



(b) modeling 3D topography on the surface



(c) crustal thickness variations (red)



(d) snapshot of the seismic wavefield on the surface (result of the simulation)

Fig. 11: Anisotropic mesh refinements to accurately model the structure of Mars on a conforming hexahedral mesh. The Mars texture map is based on NASA elevation and imagery data.

7. Conclusions. We presented algorithmic strategies for handling unstructured meshes for high-order finite-element simulations on complex domains. In particular, we demonstrated new capabilities for parallel mesh reading and topological interpolation in PETSc DMPlex, which enables a fully parallel workflow starting from the initial data file reading. This is beneficial not only for direct users of DMPlex but also users of software libraries and packages employing DMPlex, such as Firedrake [45], Salvus [1], or HPDDM [27].

This work in a sense follows up [35] and addresses the main task stated in their Future Work: *"Most crucially perhaps is the development of a fully parallel mesh input reader in PETSc in order to overcome the remaining sequential bottleneck during model initialisation."* Moreover, that paper mentions the "HDF5-based XDMF output format" but it has become an *input* format as well within this work. Hence, HDF5/XDMF has become the first widely used mesh format supported by PETSc which is both readable and writable in parallel.

The implementation is agnostic to the type of finite elements in the mesh and completely decoupled from the governing equations, and is thus applicable in many scientific disciplines. In particular, our solution overcomes bottlenecks in numerical modeling of seismic wave propagation on Mars and shows excellent parallel scalability in large-scale simulations on more than 12'000 cores.

Acknowledgments. All presented PETSc developments have been made publicly available in PETSc since its release 3.13.

We gratefully acknowledge support from the Swiss National Supercomputing Centre (CSCS) under projects s922 and s961; the Platform for Advanced Scientific Computing (PASC) under the project "Salvus"; the Swiss National Science Foundation (SNF) BRIDGE fellowship program under the grant agreement No. 175322; the European Research Council (ERC) from the EU's Horizon 2020 programme under grant agreement No. 714069; and the ETH Zurich Postdoctoral Fellowship Program which in turn received funding from the EU's Seventh Framework Programme under the grant agreement No. 608881.

REFERENCES

[1] M. Afanasiev, C. Boehm, M. van Driel, L. Krischer, M. Rietmann, D. A. May, M. G. Knepley, and A. Fichtner, *Modular and flexible spectral-element waveform modelling in two and three dimensions*, Geophysical Journal International, 216 (2019), pp. 1675–1692, https://doi.org/10.1093/gji/ggy469.

[2] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, J. Dahm, D. Medina, and S. Zampini, *Mfem: a modular finite element methods library*, 2019, https://arxiv.org/abs/1911.09220.

[3] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc web page*, https://www.mcs.anl.gov/petsc (accessed 2020-03-30).

[4] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc users manual*, Tech. Report ANL-95/11 - Revision 3.13, Argonne National Laboratory, 2020, https://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf (accessed 2020-03-30).

[5] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific

Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhäuser Press, 1997, pp. 163–202, https://doi.org/10.1007/978-1-4612-1986-6_8.

[6] W. B. Banerdt, S. E. Smrekar, D. Banfield, D. Giardini, M. Golombek, C. L. Johnson, P. Lognonné, A. Spiga, T. Spohn, C. Perrin, S. C. Stähler, D. Antonangeli, S. Asmar, C. Beghein, N. Bowles, E. Bozdag, P. Chi, U. Christensen, J. Clinton, G. S. Collins, I. Daubar, V. Dehant, M. Drilleau, M. Fillingim, W. Folkner, R. F. Garcia, J. Garvin, J. Grant, M. Grott, J. Grygorczuk, T. Hudson, J. C. E. Irving, G. Kargl, T. Kawamura, S. Kedar, S. King, B. Knapmeyer-Endrun, M. Knapmeyer, M. Lemmon, R. Lorenz, J. N. Maki, L. Margerin, S. M. McLennan, C. Michaut, D. Mimoun, A. Mittelholz, A. Mocquet, P. Morgan, N. T. Mueller, N. Murdoch, S. Nagihara, C. Newman, F. Nimmo, M. Panning, W. T. Pike, A.-C. Plesa, S. Rodriguez, J. A. Rodriguez-Manfredi, C. T. Russell, N. Schmerr, M. Siegler, S. Stanley, E. Stutzmann, N. Teanby, J. Tromp, M. van Driel, N. Warner, R. Weber, and M. Wieczorek, *Initial results from the InSight mission on Mars*, Nat. Geosci., 13 (2020), pp. 183–189, https://doi.org/10.1038/s41561-020-0544-y.

[7] N. Barral, M. G. Knepley, M. Lange, M. D. Piggott, and G. J. Gorman, *Anisotropic mesh adaptation in firedrake with petsc dmplex*, 2016, https://arxiv.org/abs/1610.09874.

[8] J. Brown, *Star forests as a parallel communication model*, 2011, https://jedbrown.org/files/StarForest.pdf (accessed 2020-03-17).

[9] J. Brown, M. G. Knepley, D. A. May, L. C. McInnes, and B. F. Smith, *Composable linear solvers for multiphysics*, in Proceeedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC 2012), IEEE Computer Society, 2012, pp. 55–62, https://doi.org/10.1109/ISPDC.2012.16.

[10] J. Brown, M. G. Knepley, and B. F. Smith, *Run-time extensibility and librarization of simulation software*, Computing in Science Engineering, 17 (2015), pp. 38–45, https://doi.org/10.1109/MCSE.2014.95.

[11] P. R. Brune, M. G. Knepley, and L. R. Scott, *Unstructured geometric multigrid in two and three dimensions on complex and graded meshes*, SIAM Journal on Scientific Computing, 35 (2013), pp. A173–A191, https://doi.org/10.1137/110827077, https://arxiv.org/abs/1104.0261.

[12] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox, *Extreme-scale AMR*, in SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1–12, https://doi.org/10.1109/SC.2010.25.

[13] C. Burstedde, L. C. Wilcox, and O. Ghattas, *p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM Journal on Scientific Computing, 33 (2011), pp. 1103–1133, https://doi.org/10.1137/100791634.

[14] H. Bériot, A. Prinn, and G. Gabard, *Efficient implementation of high-order finite elements for Helmholtz problems*, International Journal for Numerical Methods in Engineering, 106 (2016), pp. 213–240, https://doi.org/10.1002/nme.5172.

[15] C. Chevalier and F. Pellegrini, *PT-Scotch: A tool for efficient parallel graph ordering*, Parallel Computing, 34 (2008), pp. 318–331, https://doi.org/10.1016/j.parco.2007.12.001.

[16] Cray Inc., *Cray Sonexion 3000 Storage System*, 2016, https://www.cray.com/sites/default/files/resources/WP-Cray-Sonexion-3000-Storage-Systems.pdf (accessed 2020-03-20).

[17] P. E. Farrell, L. Mitchell, and F. Wechsung, *An augmented Lagrangian preconditioner for the 3D stationary incompressible Navier–Stokes equations at high Reynolds number*, SIAM Journal on Scientific Computing, 41 (2019), pp. A3073–A3096, https://doi.org/10.1137/18M1219370.

[18] A. Ferroni, P. Antonietti, I. Mazzieri, and A. Quarteroni, *Dispersion-dissipation analysis of 3-D continuous and discontinuous spectral element methods for the elastodynamics equation*, Geophys. J. Int., 211 (2017), pp. 1554–1574, https://doi.org/10.1093/gji/ggx384.

[19] A. Fichtner, *Full Seismic Waveform Modelling and Inversion*, Advances in Geophysical and Environmental Mechanics and Mathematics, Springer Berlin Heidelberg, 2011, https://doi.org/10.1007/978-3-642-15807-0.

[20] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, *Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws*, Journal of Parallel and Distributed Computing, 47 (1997), pp. 139–152, https://doi.org/10.1006/jpdc.1997.1412.

[21] D. Giardini, P. Lognonné, W. B. Banerdt, W. T. Pike, U. Christensen, S. Ceylan, J. F. Clinton, M. van Driel, S. C. Stähler, M. Böse, R. F. Garcia, A. Khan, M. Panning, C. Perrin, D. Banfield, E. Beucler, C. Charalambous, F. Euchner, A. Horleston, A. Jacob, T. Kawamura, S. Kedar, G. Mainsant, J.-R. Scholz,

S. E. Smrekar, A. Spiga, C. Agard, D. Antonangeli, S. Barkaoui, E. Barrett, P. Combes, V. Conejero, I. Daubar, M. Drilleau, C. Ferrier, T. Gabsi, T. Gudkova, K. Hurst, F. Karakostas, S. King, M. Knapmeyer, B. Knapmeyer-Endrun, R. Llorca-Cejudo, A. Lucas, L. Luno, L. Margerin, J. B. McClean, D. Mimoun, N. Murdoch, F. Nimmo, M. Nonon, C. Pardo, A. Rivoldini, J. A. R. Manfredi, H. Samuel, M. Schimmel, A. E. Stott, E. Stutzmann, N. Teanby, T. Warren, R. C. Weber, M. Wieczorek, and C. Yana, *The seismicity of Mars*, Nat. Geosci., 13 (2020), pp. 205–212, https://doi.org/10.1038/s41561-020-0539-8.

[22] HDF5 Group, *HDF5*, https://portal.hdfgroup.org/display/HDF5/HDF5 (accessed 2020-03-20).

[23] HDF5 Group, *Parallel HDF5*, https://portal.hdfgroup.org/display/HDF5/Parallel+HDF5 (accessed 2020-03-20).

[24] T. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, vol. 78, Dover Publications, 2000.

[25] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas, *Recursive algorithms for distributed forests of octrees*, SIAM Journal on Scientific Computing, 37 (2015), pp. C497–C531, https://doi.org/10.1137/140970963.

[26] T. Isaac and M. G. Knepley, *Support for non-conformal meshes in PETSc's DMPlex interface*, 2015, https://arxiv.org/abs/1508.02470.

[27] P. Jolivet, F. Nataf, et al., *HPDDM – high-performance unified framework for domain decomposition methods*, https://github.com/hpddm/hpddm (accessed 2020-04-18).

[28] G. Karypis et al., *METIS - serial graph partitioning and fill-reducing matrix ordering*, http://glaros.dtc.umn.edu/gkhome/metis/metis/overview (accessed 2020-03-17).

[29] G. Karypis et al., *ParMETIS - parallel graph partitioning and fill-reducing matrix ordering*, http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview (accessed 2020-03-17).

[30] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing, 20 (1998), pp. 359–392, https://doi.org/10.1137/S1064827595287997.

[31] G. Karypis and V. Kumar, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, Journal of Parallel and Distributed Computing, 48 (1998), pp. 71–95, https://doi.org/10.1006/jpdc.1997.1403.

[32] M. G. Knepley and D. A. Karpeev, *Mesh algorithms for PDE with Sieve I: Mesh distribution*, Scientific Programming, 17 (2009), pp. 215–230, https://doi.org/10.3233/SPR-2009-0249.

[33] M. G. Knepley, M. Lange, and G. J. Gorman, *Unstructured overlapping mesh distribution in parallel*, 2017, https://arxiv.org/abs/1506.06194.

[34] T. Kärnä, S. C. Kramer, L. Mitchell, D. A. Ham, M. D. Piggott, and A. M. Baptista, *Thetis coastal ocean model: discontinuous Galerkin discretization for the three-dimensional hydrostatic equations*, Geoscientific Model Development, 11 (2018), pp. 4359–4382, https://doi.org/10.5194/gmd-11-4359-2018.

[35] M. Lange, M. G. Knepley, and G. J. Gorman, *Flexible, scalable mesh and data management using PETSc DMPlex*, in Proceedings of the 3rd International Conference on Exascale Applications and Software, EASC '15, Edinburgh, Scotland, UK, 2015, University of Edinburgh, pp. 71–76, http://dl.acm.org/citation.cfm?id=2820083.2820097 (accessed 2020-03-30).

[36] M. Lange, L. Mitchell, M. G. Knepley, and G. J. Gorman, *Efficient mesh management in Firedrake using PETSc DMPlex*, SIAM Journal on Scientific Computing, 38 (2016), pp. S143–S155, https://doi.org/10.1137/15M1026092.

[37] M. G. Larson and F. Bengzon, *The finite element method: theory, implementation, and applications*, no. 10 in Texts in computational science and engineering, Springer, 2013, https://doi.org/10.1007/978-3-642-33287-6.

[38] A. Logg, *Efficient representation of computational meshes*, International Journal of Computational Science and Engineering, 4 (2009), pp. 283–295, https://doi.org/10.1504/IJCSE.2009.029164.

[39] P. H. Lognonné, W. B. Banerdt, D. Giardini, W. T. Pike, U. Christensen, P. Laudet, S. de Raucourt, P. Zweifel, S. Calcutt, M. Bierwirth, K. J. Hurst, F. Ijpelaan, J. W. Umland, R. Llorca-Cejudo, S. A. Larson, R. F. Garcia, S. Kedar, B. Knapmeyer-Endrun, D. Mimoun, A. Mocquet, M. P. Panning, R. C. Weber, A. Sylvestre-Baron, G. Pont, N. Verdier, L. Kerjean, L. J. Facto, V. Gharakanian, J. E. Feldman, T. L. Hoffman, D. B. Klein, K. Klein, N. P. Onufer, J. Paredes-Garcia, M. P. Petkov, J. R. Willis, S. E. Smrekar, M. Drilleau, T. Gabsi, T. Nebut, O. Robert, S. Tillier, C. Moreau, M. Parise, G. Aveni, S. Ben Charef, Y. Bennour, T. Camus, P. A. Dandonneau, C. Desfoux, B. Lecomte, O. Pot, P. Revuz, D. Mance,

J. TenPierick, N. E. Bowles, C. Charalambous, A. K. Delahunty, J. Hurley, R. Irshad, H. Liu, A. G. Mukherjee, I. M. Standley, A. E. Stott, J. Temple, T. Warren, M. Eberhardt, A. Kramer, W. Kühne, E.-P. Miettinen, M. Monecke, C. Aicardi, M. André, J. Baroukh, A. Borrien, A. Bouisset, P. Boutte, K. Brethomé, C. Brysbaert, T. Carlier, M. Deleuze, J. M. Desmarres, D. Dilhan, C. Doucet, D. Faye, N. Faye-Refalo, R. Gonzalez, C. Imbert, C. Larigauderie, E. Locatelli, L. Luno, J.-R. Meyer, F. Mialhe, J. M. Mouret, M. Nonon, Y. Pahn, A. Paillet, P. Pasquier, G. Perez, R. Perez, L. Perrin, B. Pouilloux, A. Rosak, I. Savin de Larclause, J. Sicre, M. Sodki, N. Toulemont, B. Vella, C. Yana, F. Alibay, O. M. Avalos, M. A. Balzer, P. Bhandari, E. Blanco, B. D. Bone, J. C. Bousman, P. Bruneau, F. J. Calef, R. J. Calvet, S. A. D'Agostino, G. de los Santos, R. G. Deen, R. W. Denise, J. Ervin, N. W. Ferraro, H. E. Gengl, F. Grinblat, D. Hernandez, M. Hetzel, M. E. Johnson, L. Khachikyan, J. Y. Lin, S. M. Madzunkov, S. L. Marshall, I. G. Mikellides, E. A. Miller, W. Raff, J. E. Singer, C. M. Sunday, J. F. Villalvazo, M. C. Wallace, D. Banfield, J. A. Rodriguez-Manfredi, C. T. Russell, A. Trebi-Ollennu, J. N. Maki, É. Beucler, M. Böse, C. Bonjour, J. L. Berenguer, S. Ceylan, J. F. Clinton, V. Conejero, I. J. Daubar, V. Dehant, P. Delage, F. Euchner, I. Estève, L. Fayon, L. Ferraioli, C. L. Johnson, J. Gagnepain-Beyneix, M. Golombek, A. Khan, T. Kawamura, B. Kenda, P. Labrot, N. Murdoch, C. Pardo, C. Perrin, L. Pou, A. Sauron, D. Savoie, S. C. Stähler, É. Stutzmann, N. A. Teanby, J. Tromp, M. van Driel, M. A. Wieczorek, R. Widmer-Schnidrig, and J. Wookey, *SEIS: Insight's seismic experiment for internal structure of Mars*, Space Sci. Rev., 215 (2019), p. 12, https://doi.org/10.1007/s11214-018-0574-6.

[40] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard. Version 3.1*, 2015, https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (accessed 2020-03-20).

[41] T. Nissen-Meyer, A. Fournier, and F. A. Dahlen, *A two-dimensional spectral-element method for computing spherical-earth seismograms - II. Waves in solid-fluid media.*, Geophys. J. Int., 174 (2008), pp. 873–888.

[42] Oracle and Intel Corporation, *Lustre Software Release 2.x - Operations Manual*, 2017, http://doc.lustre.org/lustre_manual.pdf (accessed 2020-03-20).

[43] A. T. Patera, *A spectral element method for fluid dynamics: Laminar flow in a channel expansion*, Journal of Computational Physics, 54 (1984), pp. 468–488.

[44] D. Peter, D. Komatitsch, Y. Luo, R. Martin, N. Le Goff, E. Casarotti, P. Le Loher, F. Magnoni, Q. Liu, C. Blitz, T. Nissen-Meyer, P. Basini, and J. Tromp, *Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes*, Geophys. J. Int., 186 (2011), pp. 721–739.

[45] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, *Firedrake: Automating the finite element method by composing abstractions*, ACM Trans. Math. Softw., 43 (2016), https://doi.org/10.1145/2998441.

[46] J. Rudi, A. C. I. Malossi, T. Isaac, G. Stadler, M. Gurnis, P. W. J. Staar, Y. Ineichen, C. Bekas, A. Curioni, and O. Ghattas, *An extreme-scale implicit solver for complex PDEs: highly heterogeneous flow in earth's mantle*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, Association for Computing Machinery, 2015, pp. 1–12, https://doi.org/10.1145/2807591.2807675.

[47] R. Schneiders, *Octree-based hexahedral mesh generation*, International Journal of Computational Geometry & Applications, 10 (2000), pp. 383–398, https://doi.org/10.1142/S021819590000022X.

[48] T. J. Tautges, C. Ernst, C. Stimpson, R. J. Meyers, and K. Merkley, *MOAB: a mesh-oriented database*, Tech. Report SAND2004-1592, Sandia National Laboratories, 2004, https://doi.org/10.2172/970174.

[49] M. van Driel, C. Boehm, L. Krischer, and M. Afanasiev, *Accelerating numerical wave propagation using wavefield adapted meshes. Part I: forward and adjoint modelling*, Geophysical Journal International, 221 (2020), pp. 1580–1590, https://doi.org/10.1093/gji/ggaa058.

[50] *XDMF – eXtensible Data Model and Format*, http://www.xdmf.org/ (accessed 2020-03-20).

[51] O. Zienkiewicz, R. Taylor, and J. Zhu, eds., *The Finite Element Method: its Basis and Fundamentals (Seventh Edition)*, Butterworth-Heinemann, Oxford, seventh edition ed., 2013, https://doi.org/10.1016/C2009-0-24909-9.