SIAM J. SCI. COMPUT.
Vol. 33, No. 3, pp. 1103–1133
© 2011 Society for Industrial and Applied Mathematics

# p4est: SCALABLE ALGORITHMS FOR PARALLEL ADAPTIVE MESH REFINEMENT ON FORESTS OF OCTREES*

CARSTEN BURSTEDDE†, LUCAS C. WILCOX‡, AND OMAR GHATTAS§

**Abstract.** We present scalable algorithms for parallel adaptive mesh refinement and coarsening (AMR), partitioning, and 2:1 balancing on computational domains composed of multiple connected two-dimensional quadtrees or three-dimensional octrees, referred to as a forest of octrees. By distributing the union of octants from all octrees in parallel, we combine the high scalability proven previously for adaptive single-octree algorithms with the geometric flexibility that can be achieved by arbitrarily connected hexahedral macromeshes, in which each macroelement is the root of an adapted octree. A key concept of our approach is an encoding scheme of the interoctree connectivity that permits arbitrary relative orientations between octrees. Based on this encoding we develop interoctree transformations of octants. These form the basis for high-level parallel octree algorithms, which are designed to interact with an application code such as a numerical solver for partial differential equations. We have implemented and tested these algorithms in the `p4est` software library. We demonstrate the parallel scalability of `p4est` on its own and in combination with two geophysics codes. Using `p4est` we generate and adapt multioctree meshes with up to $5.13 \times 10^{11}$ octants on as many as 220,320 CPU cores and execute the 2:1 balance algorithm in less than 10 seconds per million octants per process.

**Key words.** forest of octrees, parallel adaptive mesh refinement, Morton code, scalable algorithms, large-scale scientific computing

**AMS subject classifications.** 65M50, 65Y05, 68W10, 65D18

**DOI.** 10.1137/100791634

## 1. Introduction.

Physical phenomena are often characterized by multiple scales in both space and time. While in some examples the multiscale nature is prevalent throughout the system (such as in turbulence [16]), others give rise to highly localized small-scale processes within a large-scale system (such as convection in earth's mantle [32]). We are concerned with numerical methods for solution of the partial differential equations (PDEs) describing the latter class of phenomena. Resolving the multiscale features with a uniformly fine computational mesh is impractical in many cases even on petascale supercomputers, while coarser uniform meshes under-resolve the fine-scale features, leading to erroneous results. These considerations motivate the use of adaptive mesh refinement and coarsening (AMR) methods, which allow for locally varying mesh size and thus can potentially save many orders of magnitude in the number of unknowns required for a given accuracy.

*Submitted to the journal's Methods and Algorithms for Scientific Computing section April 9, 2010; accepted for publication (in revised form) February 2, 2011; published electronically May 17, 2011. This work was partially supported by NSF grants OCI-0749334, CCF-0427985, CNS-0540372, CNS-0619838, DMS-0724746, and OPP-0941678 and DOE grants 06ER25782, 08ER25860, and SC0002710.

http://www.siam.org/journals/sisc/33-3/79163.html

†Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712 (carsten@ices.utexas.edu).

‡Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712. Current address: HyPerComp, Inc., Westlake Village, CA 91361 (lucasw@ices.utexas.edu).

§Department of Mechanical Engineering, Jackson School of Geosciences and Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712 (omar@ices.utexas.edu).

Different approaches exist for managing nonuniform meshes. Hierarchical hybrid grids [4] split the domain into unstructured macroelements which are then uniformly refined; here adaptivity is confined to the coarse scale. Completely unstructured AMR provides greater geometric flexibility and fine-scale adaptivity at the cost of explicitly storing all neighborhood relations between mesh elements; see, e.g., [15, 22, 27]. Block-structured AMR methods, e.g., [5, 6, 11, 12, 18, 23], utilize unions of possibly mapped regular grids, which can be encoded cheaply and permit reuse of uniform-mesh code. They also incur some numerical complexity due to multiple grids or patches of differing resolutions, especially for high-order discretizations. Tree-based methods make use of recursive encoding schemes while allowing nonoverlapping refinement; in this sense they combine efficiency and simplicity. Since the spatial domain encoded by a quadtree [14] or an octree [24] is cube-shaped, various extensions have been developed for representing more general geometries (see, e.g., [15, 21, 28, 29, 33]). One possible strategy is to connect multiple adaptive octrees into a *forest of octrees* that can represent a wide variety of geometric shapes. The goal of this article is to present efficient parallel algorithms for the construction and management of forests of octrees that can scale to the $\mathcal{O}(10^5)$ CPU cores characterizing today's petascale systems. To our knowledge, these are the first AMR algorithms to achieve this goal.

Parallel AMR poses significant challenges both in encoding and storing a distributed mesh and in equipartitioning the workload among processes. The flexible-geometry and general-purpose finite element libraries libMesh [20] and `deal.II` [3] circumvent the mesh-encoding problem by replicating the global mesh on all processes, which limits scalability to a few hundred processes at most. Parallel load-balancing for unstructured and hybrid meshes is often delegated to special-purpose software such as parMETIS [19] and Zoltan [13]. Depending on the computational approach chosen (such as parallel graph partitioning), this can incur significant overhead. Tree-based schemes naturally lead to the concept of space-filling curves, which can be exploited for both parallel encoding and partitioning.

Much progress has been made in recent years developing scalable parallel algorithms on single octrees. The octor code [37] stores the single octree using pointers between parent and child nodes and has been scaled to 2,000 processes [1] and later to over 32,000 [9] and 62,000 processes [8]. Octor has been used for earthquake modeling [1] and online parallel visualization [38]. The Dendro code stores the leaves of the single octree in a linear array and has been scaled to over 4,000 processes [35]. Adaptive finite-element implicit solvers for PDEs have been developed based on both Dendro (up to 12,288 processes [30, 31]) and octor (up to 16,384 processes [7, 10]).

The forest-of-octree approach to AMR is not new in itself. It has, for example, been used in `deal.II` for many years [3]. There, however, parallelism is restricted due to replicated mesh storage on each process. The contribution of this article is to present a set of scalable parallel algorithms that strictly adhere to distributed encoding and storage of the forest and provide scalable mesh handling capabilities for a general (not necessarily finite-element–based) numerical application. These high-level capabilities include the creation, refinement, and coarsening of a forest, the operation to ensure at most 2:1 size relations between neighboring octants (2:1 balance), a parallel (weighted) partition, and creating a ghost layer and a globally unique node numbering. Furthermore, we present a partition-independent parallel forest checksum. To realize these capabilities we develop the following key concepts:

1. A flexible encoding scheme of the interoctree connectivity that allows arbitrary relative orientations of two-dimensional (2D) quadtrees and three-dimensional (3D) octrees, and mixed periodic and nonperiodic identification of octree boundaries.

2. Exterior octants that exist in the coordinate system of an octree but outside of its root domain, and transformations of interior and exterior octants between octrees that connect through a face, edge, or corner, are used to pass neighborhood information across interoctree boundaries.

3. Continuation of the space-filling $z$-(Morton-)curve [26, 36] across all octrees in order; this is used to efficiently determine the owner process of octree-octant pairs.

We adopt several important ideas introduced for single octrees [34], in particular the storage of leaf octants in an array (linear octree), the notion of the insulation layer of an octant, and the two-stage approach to 2:1 balance, in which a local balance is executed before communicating the insulation layer and merging and rebalancing the received octants, avoiding iterative communication. However, we do not use the partitioning of local octrees into coarse blocks; in our work we draw the two-stage division between the balance of complete local octrees, and balancing across both interprocess and interoctree boundaries.

We have implemented the algorithms discussed in this article in the `p4est` software library, which will be released soon as free and open source software. `p4est` is built on C and MPI and has been used to model large-scale mantle convection [8, 32] and seismic wave propagation [7, 39], and is the basis of a new generation ice sheet modeling code. Work is underway enabling `deal.II` to use `p4est` as its parallel mesh engine [2]. We have scaled `p4est` to the full size of both the NSF Ranger SUN/AMD supercomputer (over 62,000 CPU cores) at the Texas Advanced Computing Center and the NCCS Jaguar Cray XT5 supercomputer (over 220,000 CPU cores). We are able to construct meshes containing up to $5.13 \times 10^{11}$ octants and execute the 2:1 balance algorithm in less than 10 seconds per million octants per process; total meshing times amount to less than one minute in all of our largest examples.

This article is organized as follows. In section 2 we introduce notation and describe the encoding of the interoctree connectivity, the distributed storage and encoding of the forest, and elementary octant operations and transformations. We conclude section 2 with an algorithm that computes all possible process owners of an exterior octant, demonstrating the general structure of our forest-of-octree algorithms. In section 3 we describe high-level algorithms that are designed to interface with third-party applications such as PDE solvers. These parallel algorithms make use of the elementary local forest and octant operations described in section 2. In section 4 we discuss performance and scalability results obtained in both synthetic and simulation-driven environments. We draw conclusions in section 5.

**2. A parallel forest of octrees.** The term "octree" denotes a recursive tree structure where each node is either a leaf or has eight children. The analogous construction in two dimensions is named "quadtree," where nodes have four children instead of eight. Octrees and quadtrees can be associated with 3D and 2D cubic domains, where tree nodes are called octants and quadrants, respectively, and the root node corresponds to a cubic domain that is recursively subdivided according to the tree structure. We use the term "forest" to describe a collection of such logical cubes that are connected conformingly through faces, edges, or corners, each cube corresponding to an independent tree. We use the word "parallel" to imply distributed storage; i.e., each octant or quadrant belongs to precisely one process and is stored only there. For simplicity, we refer below to both 2D and 3D setups as octrees and octants, while sometimes providing pictures of quadtrees and quadrants for illustration.

**2.1. Overview.** A forest of octrees can be understood as a two-tier decomposition of a geometric domain. On the first (macro-)tier we split the domain $\Omega_d \subset \mathbb{R}^3$ into $K$ conforming subvolumes, each mapped from a reference cube by a smooth function $\phi_k$,

$$\bar{\Omega}_d = \bigcup_k \phi_k([0, 2^b]^d), \qquad \phi_k : [0, 2^b]^d \to \mathbb{R}^3, \qquad 0 \le k < K, \qquad d \in \{2, 3\}. \tag{2.1}$$

Specifically, we allow any manifold that can be covered by smooth images of a finite collection of $K$ reference octrees (chosen here with a maximum refinement level $b$ and length $2^b$) subject to the restriction that any macroface or macroedge shared between two octrees is shared in full or not at all (edges exist only for $d = 3$). This approach is more general than domains mappable by a single octree since any macroedge can be shared by a number of octrees different from 4, and any macrocorner can be shared by a number of octrees different from $2^d$; we also permit arbitrary periodic connectivities. Examples for $d = 2$ include the 2D circle and embeddings of 2D surfaces into 3D space such as the Möbius strip or Klein's bottle. With $d = 3$ we can represent 3D volumes such as the all-periodic unit cube, the torus, and solid and hollow spheres. In general, output from 2D or 3D hexahedral mesh generators is suitable as macromesh.

It is an important feature of `p4est` that connectivity and neighborhood relations are computed discretely (integer-based). No floating-point arithmetic is used, avoiding topological errors due to roundoff. The transformations $\phi_k$ are used for visualization only, and to encode the geometry for numerical applications external to `p4est`.

The second (micro-)tier denotes the recursive subdivision of each individual octree where the number of octants per octree is essentially unlimited. On the microtier we allow nonconforming (hanging) faces and edges. `p4est` allows unconstrained size relations between neighboring octants, and additionally provides a method to guarantee exclusively 1:1 and 2:1 size relations (called 2:1 balance) which is required for many numerical applications. Neighborhood size relations are equally respected both for octants within the same octree and for octants that are members of different octrees and connect through an octree macroface, macroedge, or macrocorner.

Octants within an octree can be assigned a natural ordering by a traversal across all leaves. By the equivalence of tree nodes and octants this one-dimensional sequence corresponds to a space-filling $z$-shaped curve in the geometric domain. We extend this concept to a forest of octrees by connecting the space-filling curve between octrees, thus generating a total ordering of all octants in the domain. A parallel partition is created by dividing the curve into $P$ segments with $P$ being the number of parallel processes. A 2:1 balanced forest and its space-filling curve and parallel partition are shown in Figure 1.

The macrostructure of the forest is static and shared between all processes. The number of octrees is thus limited by local memory (which allows up to a few million octrees in our experiments). The microstructure, i.e., the division of each octree into octants and the partition of these octants between processes, is dynamic and distributed in parallel. The main objective of this article is to outline fast and scalable algorithms that operate on this two-tier structure. With regard to common mesh-management capabilities required by numerical applications, the following high-level algorithms are presented.

`New` Create an equipartitioned, uniformly refined forest.

`Refine` Adaptively subdivide octants based on a refinement marker or callback function, once or recursively.

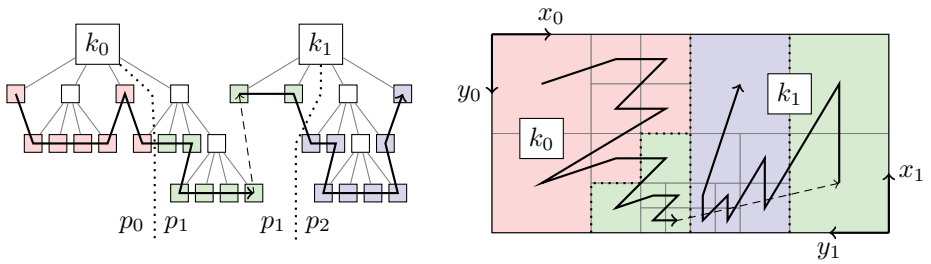



FIG. 1. *One-to-one correspondence between a forest of octrees (left) and a geometric domain partitioned into elements (right), shown for a 2D example with two octrees $k_0$ and $k_1$. The leaves of the octrees bijectively correspond to elements that cover the domain with neither holes nor overlaps. A left-to-right traversal of the leaves through all octrees creates a space-filling z-curve (black "zig-zag" line) that imposes a total ordering of all octants in the domain. For each octree the z-curve follows the orientation of its coordinate axes. In this example the forest is partitioned among three processes $p_0$, $p_1$, and $p_2$ by using the uniform partitioning rule* (2.5)*. This partition divides the space-filling curve and thus the geometric domain into three process segments of equal ($\pm 1$) octant count.*

`Coarsen` Replace families of eight child octants by their common parent octant, once or recursively.

`Partition` Redistribute the octants in parallel, according to a given target number of octants for each process, or weights prescribed for all octants.

`Balance` Ensure at most 2:1 size relations between neighboring octants by local refinement where necessary.

`Ghost` Collect one layer of off-process octants touching the process boundaries from the outside.

`Nodes` Create a globally unique numbering of the mesh nodes (i.e., the vertices at the corners of octants, not to be confused with octree nodes), taking into account the classification into "independent" and "hanging" nodes.

`Checksum` Compute a partition-independent integer "fingerprint" of a forest.

The `New`, `Refine`, and `Coarsen` algorithms are process-local and do not require communication. The `Partition` algorithm consists of an `Allgather` operation on a small number of bytes per process and point-to-point communications whose global volume is proportional to the global number of octants. The `Balance` algorithm requires collective and point-to-point communication between processes whose octants are geometrically close to each other. The communication volume for `Balance`, `Ghost`, and `Nodes` roughly scales with the number of octants on the boundaries between processes. `Checksum` requires only a `Gather` call of two unsigned integers. These high-level algorithms are discussed in detail in section 3. In order to describe their internal functioning we must first discuss a number of low-level algorithms that control the encoding and transformation of octants both within and between octrees. These encoding schemes and elementary algorithms are described in the remainder of section 2.

**2.2. Interoctree connectivity.** The connectivity between octrees is defined by topological relations within the macromesh of octrees. This connectivity can be constructed by hand, or by translating output from hexagonal mesh generators such as CUBIT [25]. In `p4est`, octrees can share faces, edges, and corners, where neighboring octrees can be arbitrarily rotated against each other. Connections can also be nonlocal, which allows for implementing arbitrary periodicities. In fact, self-penetrating connectivities are permitted. This generality is introduced for the purpose of accommodating a large class of possible computational domains.

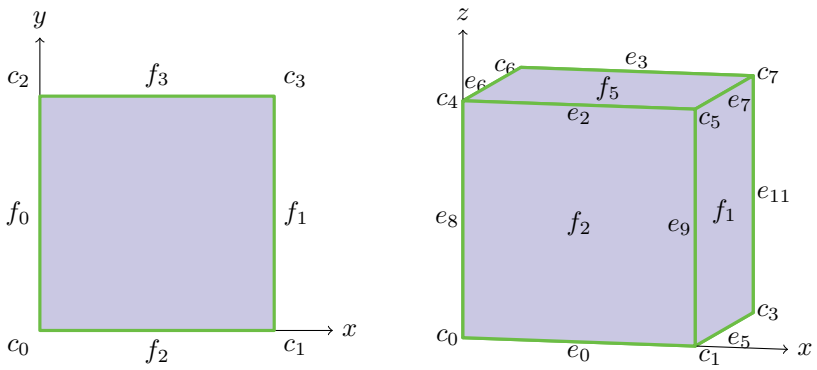



FIG. 2. *z-order numbering convention for 2D quadtrees (left) and 3D octrees (right). The symbols indicate corners $c_i$, edges $e_i$ (3D only), and faces $f_i$. Quadtrees and octrees can be arbitrarily embedded into 3D space. Octrees always have right-handed coordinate systems. Quadrants/octants inherit the coordinate system from the quadtree/octree they belong to. Indices are 0-based.*

TABLE 1

*Edge lookup tables for three dimensions. The entries $\mathcal{S}(e,i)$ store the two faces $f_i$ adjacent to edge $e$. The pairs $\mathcal{T}(e,f)$ store the two face corners $\xi_i$ of the face $f$ along edge $e$. Nonexisting combinations of faces and edges are denoted by $(*,*)$. The entries $\mathcal{U}(e,i)$ store the two corners $c_i$ touching edge $e$.*

$$\mathcal{S} = \begin{pmatrix} 2 & 4 \\ 3 & 4 \\ 2 & 5 \\ 3 & 5 \\ 0 & 4 \\ 1 & 4 \\ 0 & 5 \\ 1 & 5 \\ 0 & 2 \\ 1 & 2 \\ 0 & 3 \\ 1 & 3 \end{pmatrix} \qquad \mathcal{T} = \begin{pmatrix} (*,*) & (*,*) & (0,1) & (*,*) & (0,1) & (*,*) \\ (*,*) & (*,*) & (*,*) & (0,1) & (2,3) & (*,*) \\ (*,*) & (*,*) & (2,3) & (*,*) & (*,*) & (0,1) \\ (*,*) & (*,*) & (*,*) & (2,3) & (*,*) & (2,3) \\ (0,1) & (*,*) & (*,*) & (*,*) & (0,2) & (*,*) \\ (*,*) & (0,1) & (*,*) & (*,*) & (1,3) & (*,*) \\ (2,3) & (*,*) & (*,*) & (*,*) & (*,*) & (0,2) \\ (*,*) & (2,3) & (*,*) & (*,*) & (*,*) & (1,3) \\ (0,2) & (*,*) & (0,2) & (*,*) & (*,*) & (*,*) \\ (*,*) & (0,2) & (1,3) & (*,*) & (*,*) & (*,*) \\ (1,3) & (*,*) & (*,*) & (0,2) & (*,*) & (*,*) \\ (*,*) & (1,3) & (*,*) & (1,3) & (*,*) & (*,*) \end{pmatrix} \qquad \mathcal{U} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 0 & 2 \\ 1 & 3 \\ 4 & 6 \\ 5 & 7 \\ 0 & 4 \\ 1 & 5 \\ 2 & 6 \\ 3 & 7 \end{pmatrix}$$

The algorithms described below frequently refer to faces $f_i$, edges $e_i$, corners $c_i$, and face corners $\xi_i$. Each octree has $2d$ faces, 12 edges (3D only), and $2^d$ corners; each face has $2^{d-1}$ face corners. Our chosen numbering convention is 0-based and illustrated in Figure 2. We fix the origin in the front lower left corner and use $x$ before $y$ before $z$ for faces, edges, and corners (also called $z$-order). This means that 3D corners have the binary representation $(c_z c_y c_x)_2 \in \{0,\dots,7\}$; cf. [34]. The corners of a face are enumerated in the same sequence as they occur in the corner numbering of the octree. Table 1 shows how adjacent face and face corner numbers depend on an edge, while Table 2 shows how corner numbers are associated with faces for both two dimensions and three dimensions.

We encode 3D face connections by two arrays of dimension $K \times 6$. The array $\mathcal{NO}$ identifies for octree $k$ and face $f$ the number of the neighboring octree $k'$ across that face, $k' = \mathcal{NO}(k,f)$. Similarly, the array $\mathcal{NF}$ stores the number of the connecting face $f'$ from the perspective of octree $k'$. Since face connections can be rotated in 4 different ways, we encode both the neighbor's face number $f' \in \{0,\dots,5\}$ and the relative orientation $r \in \{0,\dots,3\}$ of the faces in the array of face connections as $\mathcal{NF}(k,f) = 6r + f' \in \{0,\dots,23\}$. If there is no neighbor across a face (as is the

TABLE 2

*Corner lookup tables for two dimensions and three dimensions. The entry $\mathcal{V}_d(f,\xi)$ contains the corner $c$ for each face $f$ and face corner $\xi$. Note that the even rows of $\mathcal{V}_3$ yield the first column of $\mathcal{U}$ from Table 1, and the odd rows yield the second column.*

$$\mathcal{V}_2 = \begin{pmatrix} 0 & 2 \\ 1 & 3 \\ 0 & 1 \\ 2 & 3 \end{pmatrix} \qquad \mathcal{V}_3 = \begin{pmatrix} 0 & 2 & 4 & 6 \\ 1 & 3 & 5 & 7 \\ 0 & 1 & 4 & 5 \\ 2 & 3 & 6 & 7 \\ 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix}$$

case on a domain boundary), the originating octree and face numbers $k$ and $f$ are inserted instead (this precludes only the pathological case of an octree that connects periodically to itself through a face rotated against itself). For a periodic connection to the same octree, the originating octree is used together with the connecting face number and orientation. We determine the orientation $r$ by the following procedure.

DEFINITION 2.1. *Suppose that a face is shared between two octrees. The octree who assigns the lower face number to this face is denoted the "primary" octree, and the other the "secondary" octree. From the perspective of the primary octree identify the first corner of the face. The orientation $r$ of the face connection is then defined by the number of this face corner counted from the perspective of the secondary octree.*

In the case of equal face numbers it is verified by enumeration that either perspective yields the same result. It also follows that octrees that are mapped into each other by simple translation always find orientation $r = 0$ for their connecting face. Note that our notion of orientation is meaningful for pairs of touching faces but not for one face on its own.

Based on the encoding of the face connectivity by the orientation $r$, it is possible to construct the permutations between the two face corner enumerations with respect to both neighbors. We propose using lookup tables as follows (see Figure 3 for illustrations).

THEOREM 2.2. *Given two associated faces $f_0$ and $f_1$, their relative orientation $r$ as in Definition 2.1, and a face corner number $\xi_0 \in \{0,\dots,3\}$ as seen from $f_0$, then the face corner number $\xi_1$ as seen from $f_1$ is obtained by*

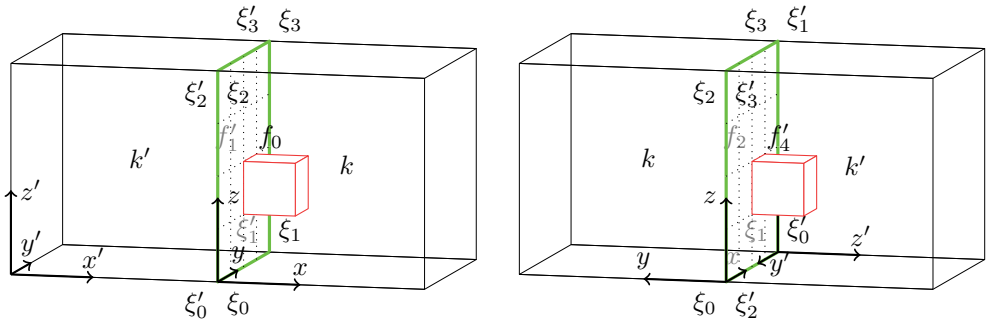
$$\xi_1 = \mathcal{P}(\mathcal{Q}(\mathcal{R}(f_0,f_1),r),\xi_0) = \mathcal{P}'(\mathcal{Q}(\mathcal{R}^T(f_0,f_1),r),\xi_0), \tag{2.2}$$

*where the matrices $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$ are specified in Table 3, and $\mathcal{P}'$ is obtained from $\mathcal{P}$ by inverting the permutation in each row.*

*Proof.* The proof can be accomplished by checking the $6 \times 6$ combinations of faces $f$, $f'$ and the 4 possible orientations $r$, amounting to 144 situations that need to be verified. The second expression follows from the first by a change in perspective. It can also be derived by noticing that transposing $\mathcal{R}$ effectively exchanges the permutations 3 and 4 in matrix $\mathcal{Q}$. From inspecting $\mathcal{P}$ we see that all permutations except for 3 and 4 are their own inverse, and 3 and 4 are inverses of each other, which completes the proof. ∎

The constant matrices $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$ are sufficient to describe all possible face connections between arbitrarily rotated octrees and can be interpreted as follows.

*Remark* 2.3. The matrix $\mathcal{R}$ selects one out of three possible situations, which correspond to rows of $\mathcal{Q}$. Situation $\mathcal{R}(f,f') = 0$ involves 4 possible corner permutations, and situations 1 and 2 involve another 4 permutations. The situations 1 and 2 map into each other when the order of the faces is reversed. Altogether only 8 out of




(a) Aligned octree coordinate systems. (b) Nonaligned octree coordinate systems.

FIG. 3. *Orientation of a macroface connection (green).* (a) *Two octrees with aligned coordinate axes. The number of the connecting face is* 1 *from the perspective of the left octree and* 0 *from the perspective of the right octree. Since the right octree* $k$ *sees the lower face number, we select* $k$ *as primary for the numbering of the face corners* $\xi_i$*. The first face corner* $\xi_0$ *connects to* $\xi'_0$ *in the secondary octree* $k'$*, which leads to the face orientation* $r = 0$*. From Table* 3 *we determine the permutation number* $\mathcal{Q}(\mathcal{R}(0,1),0) = 0$*, selecting the first row of* $\mathcal{P}$*, which is the identity permutation. This is consistent with matching face corners* $\xi_i = \xi'_i$*. For a maximum level* $b = 3$ *the level-2 octant outlined in red has coordinates* $(0,4,2)$ *in octree* $k$ *and* $(8,4,2)$ *in octree* $k'$*.* (b) *Face connection with nonaligned coordinate axes. The face numbers seen from the left and right octrees are* 2 *and* 4*, respectively. Hence the left octree* $k$ *is selected as primary. The first primary face corner* $\xi_0$ *connects to* $\xi'_2$ *in the secondary octree* $k'$*, which yields face orientation* $r = 2$*. The permutation number is* $\mathcal{Q}(\mathcal{R}(2,4),2) = 4$*, and it is verified from the picture that* $\xi_i = \xi'_{\mathcal{P}(4,i)}$*. The coordinates of the red octant are* $(4,-2,2)$ *within octree* $k$ *and* $(2,2,0)$ *within octree* $k'$*.*

TABLE 3

*The matrices* $\mathcal{P}$*,* $\mathcal{Q}$*, and* $\mathcal{R}$ *encode the permutations of face corners between octree face neighbors in three dimensions. For two associated faces* $f, f' \in \{0,\dots,5\}$*, to find the face corner* $\xi'$ *within face* $f'$ *of a given corner, based on its number* $\xi \in \{0,\dots,3\}$ *within face* $f$ *and the orientation* $r \in \{0,\dots,3\}$ *between the two faces, the result is* $\xi' = \mathcal{P}(\mathcal{Q}(\mathcal{R}(f,f'),r),\xi)$*.*

$$\mathcal{R} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 & 1 & 0 \\ 0 & 2 & 2 & 0 & 0 & 1 \\ 0 & 2 & 2 & 0 & 0 & 1 \\ 2 & 0 & 0 & 2 & 2 & 0 \end{pmatrix} \qquad \mathcal{Q} = \begin{pmatrix} 1 & 2 & 5 & 6 \\ 0 & 3 & 4 & 7 \\ 0 & 4 & 3 & 7 \end{pmatrix} \qquad \mathcal{P} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 2 & 1 & 3 \\ 1 & 0 & 3 & 2 \\ 1 & 3 & 0 & 2 \\ 2 & 0 & 3 & 1 \\ 2 & 3 & 0 & 1 \\ 3 & 1 & 2 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix}$$

the 24 possible permutations occur, namely, $0, 2, 7, 10, 13, 16, 21, 23$, which are listed in ascending order as rows of $\mathcal{P}$.

For each edge $e$ of an octree $k$ we store all connecting octrees $k'$ and their respective connecting edges $e' \in \{0,\dots,11\}$ in the set $\mathcal{ET}_{k,e}$. Different octrees connecting through this edge can be oriented in two ways against each other. If from both of their perspectives the two corners of the edge have the same sequence within the octree corners (and thus faces), the edge orientation is set to $s = 0$; for the opposite sequence it is set to $s = 1$. While any number of octrees can connect through an edge, only the relative edge orientation between any two octrees matters for our purposes; thus we can choose an arbitrary octree on this edge as a reference. Octrees mapped by simple translation always find edge orientation 0 between them. Edge connections are stored as $(k', 12s + e') \in \mathcal{ET}_{k,e}$.

Corner connections are identified by an arbitrary number of pairs of an octree $k'$

together with its respective corner $c'$ that is connected, with no need to specify an orientation. Thus, for a corner $c$ we store $(k', c') \in \mathcal{CT}_{k,c}$ for the participating octrees.

The connectivity in two dimensions is described analogously. The two arrays $\mathcal{NO}$ and $\mathcal{NF}$ encoding the face neighborhood are of dimension $K \times 4$. The face orientation $r$ can only be either 0 or 1, with face connectivity values $\mathcal{NF}(k, f) = 4r + f' \in \{0, \ldots, 7\}$. The face orientation $r$ corresponds to one of the two possible permutations of $(0, 1)$. There are no edges; as in three dimensions, corners are not oriented.

**2.3. Encoding of the parallel forest.** An octant is uniquely specified by the number $k$ of the octree it is associated with, the integer coordinates $x, y, z \in \{0, \ldots, 2^b - 1\}$ of its front lower left corner with respect to the octree coordinate system, and its depth of refinement (level $\ell$, $0 \leq \ell \leq b$). We group octants in per-octree arrays; thus it is not necessary to store the octree number $k$ with the octant. In each array we store only the leaves of the octree, which is sometimes called "linear" octree storage. The octant coordinates are stored as integers of a fixed number $b$ of bits, where the highest (leftmost) bit represents the first vertical level of the octree (counting the root as level zero), the second highest bit the second level of the octree, and so on. Bits below the level of the octant are set to 0. The level is stored additionally to distinguish between octants located at identical coordinates, but of different sizes. The edge length of a level-$\ell$ octant in integer coordinates is $h = 2^{b-\ell}$, normalizing the integer length of an octree to $2^b$. The faces, edges, corners, and face corners of an octant are enumerated in the same sequence that is used for the containing octree; see section 2.2.

Based on the information described above, all octants can be ordered linearly. To construct a unique index we prepend the octree number in binary notation to the fixed-length binary Morton index of the octant, thus sorting octants first by octree, and then by their Morton index [26]. The Morton index is composed of the bitwise-interleaved coordinates $m$ and the level $\ell$, with $m$ defined as

$$m_{di+2} = z_i, \qquad m_{di+1} = y_i, \qquad m_{di+0} = x_i, \qquad i = b-1, \ldots, 0 \tag{2.3}$$

(the $z$-coordinate is present only in three dimensions). Thus $m$ has $db$ bits. To better understand the nature of $m$ we regroup it as a base-8 number (base-4 in two dimensions),

$$m = (n_{b-1}, \ldots, n_1, n_0), \qquad n_i \in \{0, \ldots, 2^d - 1\}. \tag{2.4}$$

In three dimensions, each octal digit $n_i$ consists of 3 bits which select the branch of the octree along which to descend from level $b - i - 1$ to level $b - i$. Each of the 8 possibilities corresponds to a specific child of the parent octant (see Algorithm 1 for a routine to obtain the child identification of a given octant). Since by (2.3) these 3 bits are in the order $zyx$, the children are numbered from 0 for the front lower left child, to 1 for the front lower right child, to 2 for the back lower left, and so on, with $4, \ldots, 7$ being the four children on top of the children $0, \ldots, 3$. This concept holds recursively, and thus $(m, \ell)$ encodes the complete path descending from the root of the octree to any given octant at depth $\ell$. To ascend the octree toward the root, a parent octant can be constructed from any of its children by Algorithm 2. The mapping to $z$-order is bijective; thus (2.3) and (2.4) are invertible. The procedure to compute the coordinates $(x, y, z)$ from a Morton index $(m, \ell)$ is given in Algorithm 3. In summary, a total ordering among forest octants is given by sorting all octree-octant pairs $(k_i, o_i)$ first by octree $k$ and second by the Morton index of $o$.

**Algorithm 1** `Child_id` (`octant` $o$)

```
1: i ← 0 ; h ← 2^{b−o.ℓ}                    {the octant o has member variables o.ℓ, o.x, o.y, o.z}
2: i ← i | ((o.x & h) ? 1 : 0)               {bitwise OR and AND operators are denoted by | and &}
3: i ← i | ((o.y & h) ? 2 : 0)
4: i ← i | ((o.z & h) ? 4 : 0)               {remove this line for d = 2}
5: return i
```

**Algorithm 2** `Parent` (`octant` $o$)

```
1: h ← 2^{b−o.ℓ}
2: q.ℓ ← o.ℓ − 1                             {legal for only o.ℓ > 0}
3: q.x ← o.x & ¬h                            {bitwise negation operator is denoted by ¬}
4: q.y ← o.y & ¬h
5: q.z ← o.z & ¬h                            {remove this line for d = 2}
6: return q
```

**Algorithm 3** `Octant` (`int` $\ell \geq 0$, `int` $m \in \{0, \ldots, 2^{d\ell} - 1\}$)

```
1: o.x ← 0 ; o.y ← 0 ; o.z ← 0
2: o.ℓ ← ℓ ; h ← 2^{b−ℓ}
3: for i = 0, ..., ℓ − 1 do
4:     o.x ← o.x | (h(m & 2^{di})/2^{(d−1)i})
5:     o.y ← o.y | (h(m & 2^{di+1})/2^{(d−1)i+1})
6:     o.z ← o.z | (h(m & 2^{di+2})/2^{(d−1)i+2})    {remove this line for d = 2}
7: end for
8: return o
```

Due to the nature of the $z$-order, 2D and 3D forests follow a similar logic. The encoding in two dimensions can be derived by removing the $z$ coordinate and lines in the algorithm that use it (such as line 4 in Algorithm 1, line 5 in Algorithm 2, and line 6 in Algorithm 3). In the 2D case, using $2b$ bits for the interleaved coordinates corresponds to base-4 numbers. This principle enables us to realize both 2D and 3D functionality nonredundantly within the same computer code using compile-time definitions (this would be much harder in right-hand-rule ordering, for example). Even one-dimensional code is a subset obtained by setting $d = 1$ and removing lines that refer to either $y$ or $z$. In principle, there is no limit for dimensions $d \geq 4$ except available memory. Currently one octant is stored using 24 bytes.

For parallel computation the forest of octrees is partitioned; i.e., each octant is assigned to exactly one process (its owner). In general no copies are stored on other processes, so that octant storage is fully distributed. We use the total ordering defined by the global index $(k, m, \ell)$ to assign contiguous ranges of octants to the processes, in ascending order, denoting the number of octants owned by process $p$ by $N_p$. Since the octants of one octree may be divided between multiple processes, and one process may own parts of more than one octree, we also identify the first level-$b$ descendant $o_p$ of the first local octant on the first local octree $k_p$ on process $p$. The parallel partition of the forest is then completely defined by sharing the triples $(N_p, k_p, o_p)$ between all processes. In `p4est` this global information amounts to 32 bytes per process. From $(N_p)$ each process can compute the cumulative octant counts $G_p = \sum_{p'=0}^{p-1} N_{p'}$.

This partitioning scheme allows empty processes $p$ identified by $N_p = 0$ and $(k_p, o_p) = (k_{p+1}, o_{p+1})$. If a uniform partition is desired, the target number of octants $N_p$ can be derived from the total number of octants $N$ and the number of processes

$P$ by the same formula as in [37],

$$N_p^{(\text{uniform})} = \left\lfloor \frac{N(p+1)}{P} \right\rfloor - \left\lfloor \frac{Np}{P} \right\rfloor. \tag{2.5}$$

**2.4. Operations on octants.** The parallel algorithms described in this article are built on various elementary operations on octants. We distinguish two main types: octree-local and interoctree operations. Octree-local operations take place within the same octree coordinate system (see Figure 2) and are generally accomplished by arithmetic operations on the $x$, $y$, and $z$ coordinates of the octree. Examples are Algorithms 1, 2, and 3. We add another octree-local algorithm that, given an octant $o$, computes the two smallest possible octants that fit into the first and last corners of $o$, respectively. These computed octants are called first and last descendants of $o$ since they are connected to $o$ by a path down the octree to the maximum level $\ell = b$; see Algorithm 4. Further examples for octree-local operations are the computations of the neighbor of an octant across a given face, edge, or corner. These operations modify 1, 2, or 3 octant coordinates, respectively, and are displayed as Algorithms 5, 6, and 7.

**Algorithm 4** `Descendants` (`octant` $o$)

1: $q_1.\ell \leftarrow b$ ; $q_2.\ell \leftarrow b$ ; $h \leftarrow 2^{b-o.\ell}$ {assign maximum level $b$ to descendants}
2: $q_1.x \leftarrow o.x$ ; $q_2.x \leftarrow o.x + (h-1)$
3: $q_1.y \leftarrow o.y$ ; $q_2.y \leftarrow o.y + (h-1)$
4: $q_1.z \leftarrow o.z$ ; $q_2.z \leftarrow o.z + (h-1)$ {remove this line for $d = 2$}
5: **return** $(q_1, q_2)$

**Algorithm 5** `Face_neighbor` (`octant` $o$, `int` $f \in \{0, \ldots, 2d-1\}$)

1: $q.\ell \leftarrow o.\ell$ ; $h \leftarrow 2^{b-o.\ell}$
2: $q.x \leftarrow o.x + ((f = 0)\ ?\ -h\ :\ (f = 1)\ ?\ h\ :\ 0)$
3: $q.y \leftarrow o.y + ((f = 2)\ ?\ -h\ :\ (f = 3)\ ?\ h\ :\ 0)$
4: $q.z \leftarrow o.z + ((f = 4)\ ?\ -h\ :\ (f = 5)\ ?\ h\ :\ 0)$ {remove this line for $d = 2$}
5: **return** $q$

**Algorithm 6** `Edge_neighbor` (`octant` $o$, `int` $e \in \{0, \ldots, 11\}$)

1: $a_0 \leftarrow e$ **div** 4 {algorithm defined for $d = 3$ only; **div** denotes integer division}
2: $a_1 \leftarrow (e < 4)\ ?\ 1\ :\ 0$
3: $a_2 \leftarrow (e < 8)\ ?\ 2\ :\ 1$ {identify coordinate directions $a_0$, $a_1$, $a_2$ of edge $e$}
4: $q.\ell \leftarrow o.\ell$ ; $h \leftarrow 2^{b-o.\ell}$
5: $q.x_{a_0} \leftarrow o.x_{a_0}$ {writing $x_0$, $x_1$, $x_2$ for $x$, $y$, $z$}
6: $q.x_{a_1} \leftarrow o.x_{a_1} + (2(e\ \&\ 1) - 1)h$
7: $q.x_{a_2} \leftarrow o.x_{a_2} + ((e\ \&\ 2) - 1)h$
8: **return** $q$

*Exterior octants.* To pass neighborhood information across octree boundaries, it will prove convenient to allow for octants that exist outside of the root octree domain $[0, 2^b]^d$. These "exterior" octants are identified by coordinates $x_i < 0$ or $x_i \geq 2^b$. They are created naturally by seeking a neighbor of an octant that is touching the octree boundary, for example by invoking Algorithm 5 on an octant $o$ with $o.x = 0$ and requesting the neighbor across face $f = 0$. Similarly, exterior octants across an edge or corner can be created by Algorithms 6 or 7, respectively.

**Algorithm 7** `Corner_neighbor` (`octant` $o$, `int` $c \in \{0, \ldots, 2^d - 1\}$)

1: $q.\ell \leftarrow o.\ell$ ; $h \leftarrow 2^{b-o.\ell}$
2: $q.x \leftarrow o.x + (2(c \;\&\; 1) - 1)h$
3: $q.y \leftarrow o.y + ((c \;\&\; 2) - 1)h$
4: $q.z \leftarrow o.z + ((c \;\&\; 4)/2 - 1)h$ {remove this line for $d = 2$}
5: **return** $q$

We accommodate the extended range of coordinates by allocating two additional high bits for the integers $x$, $y$, and $z$. Thus we can fit two level-30 quadrant or three level-19 octant coordinates into a 64-bit integer. Due to the two-complement representation of integers in practically all current hardware, where the highest digit denotes the negated appropriate power of two, bitwise operations as used, for example, in Algorithm 1 yield the correct result even for negative coordinates.

*Interoctree face transformations.* Octants inherit the coordinate system of the octree they belong to. When two octrees that share a common face do not have identically aligned coordinate systems, this also holds for octants on both sides of the shared face. Thus, Algorithm 5 is no longer sufficient to shift an octant through the face boundary from one octree into the other. In this case the shift operation needs to be followed by a coordinate transformation that correctly places the shifted octant within the target octree (see Figure 3 for two illustrated cases). We execute the face transformation in three steps. First, the two coordinate axes tangential to the face and the normal axis are identified for the originating octree as $x_{a_0}$, $x_{a_1}$, and $x_{a_2}$. Second, these axes are matched with a separate ordering of axes in the target octree $x_{b_0}$, $x_{b_1}$, and $x_{b_2}$. Third, both tangential axes can be aligned in parallel or antiparallel coordinate directions; for the normal axis four different transformations are possible. These considerations lead to Algorithm 8. It makes use of the interoctree connectivity (see section 2.2 and Table 3) and is considerably more complex than the octree-local algorithms that we have encountered so far. Due to the $z$-order convention much code can be shared between $d = 2$ and $d = 3$.

**Algorithm 8** `Transform_face` (`int` $k \in \{0, \ldots, K - 1\}$, `int` $f$, `octant` $o$)

1: $k' \leftarrow \mathcal{NO}(k, f)$ {neighbor octree}
2: $f' \leftarrow \mathcal{NF}(k, f)$ **mod** $2d$ {**mod** denotes the remainder of integer division}
3: $r \leftarrow \mathcal{NF}(k, f)$ **div** $2d$ {orientation $r$; **div** denotes integer division}
4: $a_2 \leftarrow f$ **div** $2$ ; $b_2 \leftarrow f'$ **div** $2$ {$a_2$ and $b_2$ are origin and target normal axis}
5: $s' \leftarrow 1 - [(f \;\&\; 1) \oplus (f' \;\&\; 1)]$ {$s'$ is an arithmetic switch; $\oplus$ denotes exclusive OR}
6: **if** $d = 2$ **then**
7: $\quad a_0 \leftarrow 1 - a_2$ ; $b_0 \leftarrow 1 - b_2$ ; $s_0 \leftarrow r$
8: **else**
9: $\quad a_0 \leftarrow (f < 2)\ ?\ 1\ :\ 0$ ; $a_1 \leftarrow (f < 4)\ ?\ 2\ :\ 1$ {$a_0$, $a_1$, $a_2$ are coordinate axes of $f$}
10: $\quad u \leftarrow \mathcal{R}(0, f) \oplus \mathcal{R}(0, f') \oplus (r = 0 \mid r = 3)$ {flag for tangential axis ordering}
11: $\quad b_u \leftarrow (f' < 2)\ ?\ 1\ :\ 0$ ; $b_{1-u} \leftarrow (f' < 4)\ ?\ 2\ :\ 1$ {$b_0$, $b_1$, $b_2$ are coordinate axes of $f'$}
12: $\quad v \leftarrow (\mathcal{R}(f, f') = 1)$
13: $\quad s_v \leftarrow r \;\&\; 1$ ; $s_{1-v} \leftarrow r \;\&\; 2$ {$s_0$, $s_1$ are antiparallel flags for axes 0 and 1}
14: **end if**
15: $q.\ell \leftarrow o.\ell$ ; $g \leftarrow 2^b - 2^{b-o.\ell}$
16: $q.x_{b_0} \leftarrow (s_0 = 0)\ ?\ o.x_{a_0}\ :\ g - o.x_{a_0}$ {writing $x_0$, $x_1$, $x_2$ for $x$, $y$, $z$}
17: $q.x_{b_1} \leftarrow (s_1 = 0)\ ?\ o.x_{a_1}\ :\ g - o.x_{a_1}$ {remove this line for $d = 2$}
18: $q.x_{b_2} \leftarrow (2(f' \;\&\; 1) - 1)2^b + s'g + (1 - 2s')o.x_{a_2}$
19: **return** $q$

*Interoctree edge transformations.* Analogously to interoctree face transformations, interoctree edge transformations are required when an edge neighbor created by Al-

gorithm 6 crosses an octree boundary. In the algorithms `Balance`, `Ghost`, and `Nodes` described in section 3 the edge transformation is executed for all edge connections that are not already covered by a face connection. A priori information on connections of an octree $k$ through an edge $e$ is provided in the set $\mathcal{ET}_{k,e}$ introduced in section 2.2. This set generally contains three redundant connections: the pair $(k, e)$ itself and up to two connections that are already covered by the faces adjacent to this edge. We compute the latter with the function `Lookup_edge_faces`. It queries the matrices $\mathcal{S}$ and $\mathcal{T}$ introduced in Table 1 and returns the edge faces $f_i$, the neighboring octree $k'_i$, face $f'_i$, and orientation $r_i$, and the face corner numbers $\xi_{ij}$ on either side of the edge, $j \in \{0, 1\}$. The nonredundant edge connections $E \in \mathcal{EC}$, augmented with the relative edge orientations and alignment of coordinate axes, are computed in Algorithm 9. Given $E$, we can transform an octant into the coordinate system of the edge-connected octree. This operation is performed in Algorithm 10. It uses an additional flag to decide whether the octant should be placed on the connecting edge either inside or just outside of the connecting octree. Choosing the outside option creates an exterior octant.

**Algorithm 9** `Edge_connections` (`int` $k$, `int` $e$)

1: $(f_i, k'_i, f'_i, r_i, \xi_{ij}) \leftarrow$ `Lookup_edge_faces` $(k, e)$ ; $\mathcal{EC} \leftarrow \emptyset$ {algorithm for $d = 3$}
2: **for** $(k', w') \in \mathcal{ET}_{k,e}$ **do**
3: $e' \leftarrow w'$ **mod** 12 {decode edge number}
4: **if** $e' = e$ **and** $k' = k$ **then**
5: $s \leftarrow w'$ **div** 12 ; break loop {decode reference edge orientation $s$}
6: **end if**
7: **end for**
8: **for** $(k', w') \in \mathcal{ET}_{k,e}$ **do**
9: $e' \leftarrow w'$ **mod** 12 ; skip when $e' = e$ **and** $k' = k$ ; $s' \leftarrow s \oplus (w'$ **div** 12)
10: **for** $i \in \{0, 1\}$ **do**
11: **if** $k' = k'_i$ **then**
12: $\xi'_0 \leftarrow \mathcal{P}(\mathcal{Q}(\mathcal{R}(f_i, f'_i), r_i), \xi_{i0})$ ; $\xi'_1 \leftarrow \mathcal{P}(\mathcal{Q}(\mathcal{R}(f_i, f'_i), r_i), \xi_{i1})$ {access face corners}
13: **if** $\xi'_0 = \mathcal{T}(e', f'_i)_{s'}$ **and** $\xi'_1 = \mathcal{T}(e', f'_i)_{1-s'}$ **then**
14: Skip $(k', e')$ due to face adjacency
15: **end if**
16: **end if**
17: **end for**
18: $b_0 \leftarrow e'$ **div** 4 ; $b_1 \leftarrow (e' < 4)$ ? 1 : 0 ; $b_2 \leftarrow (e' < 8)$ ? 2 : 1 {edge coordinate axes}
19: $E \leftarrow (k', e', s', \{b_i\})$ ; $\mathcal{EC} \leftarrow \mathcal{EC} \cup E$ {add nonredundant edge connection $E$}
20: **end for**
21: **return** $\mathcal{EC}$

**Algorithm 10** `Transform_edge` (`int` $e$, `connection` $E$, `octant` $o$, `bool inside`)

1: $(k', e', s', \{b_i\}) \leftarrow E$ ; $a_0 \leftarrow e$ **div** 4 {algorithm for $d = 3$}
2: $q.\ell \leftarrow o.\ell$ ; $g \leftarrow 2^b - 2^{b-o.\ell}$
3: $h^- \leftarrow$ (`inside`) ? 0 : $-2^{b-o.\ell}$ ; $h^+ \leftarrow$ (`inside`) ? $g$ : $2^b$
4: $q.x_{b_0} \leftarrow s'g + (1 - 2s')o.x_{a_0}$ {transform edge-parallel coordinate direction}
5: $q.x_{b_1} \leftarrow (e' \,\&\, 1 = 0)$ ? $h^-$ : $h^+$
6: $q.x_{b_2} \leftarrow (e' \,\&\, 2 = 0)$ ? $h^-$ : $h^+$ {set remaining two coordinates}
7: **return** $q$

There are two reasons for the separation of `Edge_connection` (Algorithm 9) and `Transform_edge` (Algorithm 10). First, the edge connections $\mathcal{EC}$ for a given octree connectivity may be created at program startup or statically as extended lookup tables, thus providing the option of avoiding the computation of macroedge connections at the cost of increased memory usage. Second, a macroedge connection can be reused

to transform an arbitrary number of octants, which we exploit in several algorithms in section 3.

*Interoctree corner transformations.* In most cases an interoctree corner connection coincides with edge and face connections involving this corner. Algorithm 11 eliminates these redundant connections by calling the functions `Lookup_corner_faces`, which makes use of Table 1, and `Lookup_corner_edges`, which uses Table 2. The latter function determines sets of edge transformations $\mathcal{EC}$ by invoking Algorithm 9 and returns a flag that specifies the position of the corner on the edge. This information is used to eliminate redundancy among the connections $C \in \mathcal{CC}$ computed by Algorithm 11. These corner connections are used in Algorithm 12 to realize the transformation of an octant into a corner-connected octree. Similar to the edge transformation, it accepts a flag to decide on inside or outside placement of the octant in the target octree. The separation of the algorithms `Corner_connections` and `Transform_corner` is due to the same reasons outlined above for edges: to allow for precomputation and to avoid repeated calculations for multiple transformed octants.

**Algorithm 11** `Corner_connections` (`int` $k$, `int` $c$)

1: $(f_i, k'_i, f'_i, r_i, \xi_i) \leftarrow$ `Lookup_corner_faces` $(k, c)$ ; $\mathcal{CC} \leftarrow \emptyset$
2: **for** $(k', c') \in \mathcal{CT}_{k,c}$ **do**
3:  **for** $i \in \{0, \ldots, d-1\}$ **do**
4:  **if** $k' = k'_i$ **then**
5:  $\xi' \leftarrow (d = 2)\ ?\ \xi_i \oplus r_i\ :\ \mathcal{P}(\mathcal{Q}(\mathcal{R}(f_i, f'_i), r_i), \xi_i)$ {access face corner}
6:  **if** $c' = \mathcal{V}_d(f'_i, \xi')$ **then**
7:  Skip $(k', c')$ due to face adjacency
8:  **end if**
9:  **end if**
10:  **end for**
11:  **for** $i \in \{0, 1, 2\}$ **do**
12:  $(\mathcal{EC}, s) \leftarrow$ `Lookup_corner_edges` $(k, c, i)$ {loop applies to $d = 3$ exclusively}
13:  **for** $E \in \mathcal{EC}$ **do**
14:  **if** $k' = E.k'$ **and** $c' = \mathcal{U}(E.e', s \oplus E.s')$ **then**
15:  Skip $(k', c')$ due to edge adjacency
16:  **end if**
17:  **end for**
18:  **end for**
19:  $C \leftarrow (k', c')$ ; $\mathcal{CC} \leftarrow \mathcal{CC} \cup C$ {add nonredundant corner connection $C$}
20: **end for**
21: **return** $\mathcal{CC}$

**Algorithm 12** `Transform_corner` (`int` $c$, `connection` $C$, `octant` $o$, `bool inside`)

1: $(k', c') \leftarrow C$ {decode corner connection}
2: $q.\ell \leftarrow o.\ell$ ; $g \leftarrow 2^b - 2^{b-o.\ell}$
3: $h^- \leftarrow$ (`inside`) ? $0\ :\ -2^{b-o.\ell}$ ; $h^+ \leftarrow$ (`inside`) ? $g\ :\ 2^b$
4: **for** $i = 0, \ldots, d-1$ **do**
5:  $q.x_i \leftarrow (c'\ \&\ 2^i = 0)\ ?\ h^-\ :\ h^+$ {writing $x_0$, $x_1$, $x_2$ for $x$, $y$, $z$}
6: **end for**
7: **return** $q$

The encoding of the connectivity between octrees on the macrolevel and the transformations between neighboring octrees across faces, edges, and corners described above provide the essential tools to transcend from the concept of one octree and a logically cubic domain to a forest of arbitrarily connected octrees and more general computational geometries. Using these interoctree transformations allows us to augment the octants from any given octree by octants from neighboring octrees within

the same coordinate system, greatly facilitating the creation of the higher-level forest-of-octrees parallel algorithms described in section 3.

**2.5. Determining octant process ownership.** A critical building block for parallel mesh algorithms is to determine the owner process of a given mesh primitive. In octree-based meshing the space-filling curve can be used for this purpose. When the global index $g_i$ of an octant is given, its owner can be found in $\log P$ steps by a binary search in the array of cumulative octant sums $G_p$. When the coordinates and associated octree of an interior octant are given, its owner can be found by binary searching the tuples $(k_p, o_p)$ instead. The problem is harder for exterior octants since they can be overlapping with multiple edge- or corner-connected octrees. As an example, an octant that lies diagonally outside of an octree edge can be understood as intersecting any of several octrees that are connected through this edge and can thus have multiple owners. Here we use appropriate interoctree transformation as developed in section 2.4 and loop over the relevant edge- or corner-connecting octrees. We detail these operations in `Find_owners` (Algorithm 13). Assuming that all transformations are executed on $\mathcal{O}(1)$ and the number of edge and corner connections per octree is bounded, it requires runtime $\mathcal{O}(\log P)$.

**Algorithm 13** `Find_owners` (`int` $k$, `octant` $o$)

```
 1: B ← ∅                                                    {initialize return buffer}
 2: if o is outside of octree k across a corner c then
 3:    CC ← Corner_connections (k, c)                        {nonredundant connections}
 4:    for C ∈ CC do
 5:       o' ← Transform_corner (C, o, 1)
 6:       Binary search in (k_p, o_p) for owner process p' of (C.k', o'); add p' to B
 7:    end for
 8: else if d = 3 and o is outside of octree k across an edge e then
 9:    EC ← Edge_connections (k, e)                          {nonredundant connections}
10:    for E ∈ EC do
11:       o' ← Transform_edge (e, E, o, 1)
12:       Binary search in (k_p, o_p) for owner process p' of (E.k', o'); add p' to B
13:    end for
14: else if o is outside of octree k across a face f connected to k' then
15:    o' ← Transform_face (k, f, o)
16:    Binary search in (k_p, o_p) for owner process p' of (k', o'); add p' to B
17: else
18:    Binary search in (k_p, o_p) for owner process p' of (k, o); add p' to B
19: end if
20: return B
```

**3. Scalable parallel algorithms.** In this section we provide details on high-level `p4est` algorithms that are designed to provide complete mesh management capabilities for, e.g., a parallel adaptive finite element simulation. These were briefly motivated in section 2: `New`, `Refine`, `Coarsen`, `Balance`, `Partition`, `Ghost`, and `Nodes`. The first three are process-local, while the latter four require interaction between processes and are significantly more complex. The algorithms can be called any number of times and in any order, with the exception that `Ghost` requires a `Balance`'d forest, and `Nodes` requires a prior execution of `Ghost`. The latter three `Balance`, `Ghost`, and `Nodes` are the only ones that require interactions between neighboring octants. These interactions can take place within the same octree or across an octree boundary and consequently depend on the transformations developed above in section 2. We conclude this section with the description of a parallel forest `Checksum`.

**3.1. Forest creation.** We create a forest from a given interoctree connectivity and a given minimum number of desired octants per process. This number can be as low as 0, which creates only root-level octants and leaves some of the processes empty when there are more processes than octrees. That number can also be chosen higher, leading to some uniform initial refinement. In our applications, most often we use 0 or 1 for this number and increase the number of octants adaptively via calling `Refine`; see below. Algorithm 14 details the creation of the initial forest. It refines all octrees uniformly according to the minimal number of desired octants. Filling processes with coarser octants where permitted by the parallel partition would be equally possible (in fact, we have used this successfully in earlier versions of the code). The bottom-up construction of octrees based on point sets as proposed in [34] can be emulated efficiently by `New` followed by iterations of `Refine` (adaptive subdivision based on the number of points per octant) and `Partition` (load-balance when necessary). `New` creates the forest in $\mathcal{O}(N_p)$ time and allows us to infer the parallel partition $(N_p, k_p, o_p)$ without communication.

**Algorithm 14** `New` (`int min_per_process`)

1: $\ell \leftarrow \lceil \log_2 \lceil \max\{P \,\text{min\_per\_process}/K, 1\} \rceil / d \rceil$ {compute minimum required level}
2: $n \leftarrow 2^{d\ell}$ ; $N \leftarrow nK$ {octants per octree; total octants}
3: $g_{\text{first}} \leftarrow \left\lfloor \frac{Np}{P} \right\rfloor$ ; $g_{\text{last}} \leftarrow \left\lfloor \frac{N(p+1)}{P} \right\rfloor - 1$ {global numbers of first and last local octant}
4: **if** $g_{\text{first}} > g_{\text{last}}$ **then**
5: $\quad \mathcal{T}_{\text{local}} \leftarrow \emptyset$ ; stop. {this process has no octants}
6: **end if**
7: $k_p \leftarrow \lfloor g_{\text{first}}/n \rfloor$ ; $k_{\text{last}} \leftarrow \lfloor g_{\text{last}}/n \rfloor$ ; $\mathcal{T}_{\text{local}} \leftarrow (k_p, \ldots, k_{\text{last}})$ {determine local octrees}
8: **for** $k \in \mathcal{T}_{\text{local}}$ **do**
9: $\quad m_{\text{first}} \leftarrow (k = k_p)\ ?\ g_{\text{first}} - nk\ :\ 0$ {determine octant range within octree}
10: $\quad m_{\text{last}} \leftarrow (k = k_{\text{last}})\ ?\ g_{\text{last}} - nk\ :\ n - 1$
11: $\quad \mathcal{O}_k \leftarrow \emptyset$
12: $\quad$ **for** $m \in \{m_{\text{first}}, \ldots, m_{\text{last}}\}$ **do**
13: $\quad\quad \mathcal{O}_k \leftarrow \mathcal{O}_k \cup$ `Octant` $(\ell, m)$ {construct octant and add it to array}
14: $\quad$ **end for**
15: **end for**

**3.2. Adaptive refinement and coarsening.** A core feature of existing single-octree codes and `p4est` is the ability to dynamically refine and coarsen a mesh. The `p4est` algorithms `Refine` and `Coarsen` described here are designed to be invoked from third-party application codes. They provide two modes of operation: nonrecursive and recursive. Nonrecursive `Refine` replaces an octant with its eight children but does not consider newly created children for refinement. Nonrecursive `Coarsen` replaces eight octant siblings by their parent but does not investigate the role of the newly created parent as a sibling. Many applications prefer this mode in order to interpolate a numerical field to the new mesh more easily. Recursive mode, on the other hand, is capable of radically changing the forest within one call, which is sometimes advantageous for creation of a static or initial mesh according to physical criteria.

Both `Refine` and `Coarsen` (see Figure 4 and Algorithm 15) traverse the leaf octants $\mathcal{O}_k$ for each local octree and modify the octant structure on the fly, thus invalidating $N_p$, which can be recomputed by a call to `Allgather` when needed. `Refine` uses a list moved with the point of traversal to store octants until extra room is created in the array. `Coarsen` uses a sliding window that encloses the growing empty range in the array. During recursive coarsening the beginning of this window may move backward, while the end always moves forward. Their runtime is $\mathcal{O}(N_p)$.

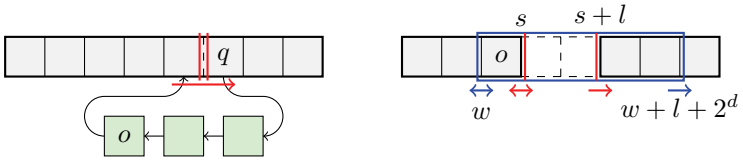



FIG. 4. *Left: schematic operation of* `Refine`. *If the octant $o$ is refined then it is popped from the front of the list and its children are prepended, the first of which is now named $o$. Optionally this is done recursively. If eventually $o$ is not to be refined then the octant $q$ is appended to the list and $o$ is removed and stored where $q$ was stored previously, and the separator (red) and list pointers are moved one step forward. Right: operation of* `Coarsen`. *Coarsening creates an empty range of growing length $l$. The sliding window (blue) starting at $w$ contains $2^d$ octants which are tested for being a valid octant family and being specified for coarsening. If this is true the octant $o$ is replaced by the parent of the family and the empty range is grown by $2^d - 1$. For recursive coarsening $w$ and $s$ may move backward depending on the child number of $o$ (see Algorithm 15).*

**Algorithm 15** `Coarsen` (`bool recursive`, `callback Do_coarsen`)

1: **for** $k \in \mathcal{T}_{\text{local}}$ **do**
2: $n \leftarrow \#\mathcal{O}_k$ {number of octants in local octree}
3: $w \leftarrow 0$ ; $C \leftarrow 2^d$ {start and length of sliding window}
4: $s \leftarrow 1$ ; $l \leftarrow 0$ {start and length of empty range in array}
5: **while** $w + C + l \leq n$ **do**
6: $c \leftarrow n$ {set default $c > w$ for comparison below}
7: **if** `Is_Family` ($w$, $s$, $l$) **and** `Do_coarsen` ($k$, $w$, $s$, $l$) **then**
8: $\mathcal{O}_k[w] \leftarrow$ `Parent` ($\mathcal{O}_k[w]$) ; $c \leftarrow$ `Child_id` ($\mathcal{O}_k[w]$) {replace first child with parent}
9: $s \leftarrow w + 1$ ; $l \leftarrow l + C - 1$ {remove the remaining coarsened octants}
10: **end if**
11: **if** $c \leq w$ **and** `recursive` **then**
12: $w \leftarrow w - c$ {move window backward}
13: **else**
14: $w \leftarrow w + 1$ {advance window}
15: **if** $w = s$ **and** $s + l < n$ **then**
16: $\mathcal{O}_k[s] \leftarrow \mathcal{O}_k[s + l]$ ; $s \leftarrow w + 1$ {move one octant to start of empty range}
17: **end if**
18: **end if**
19: **end while**
20: $\mathcal{O}_k[s, \ldots, n - l - 1] \leftarrow \mathcal{O}_k[s + l, \ldots, n - 1]$ {move octants to start of empty range}
21: $\#\mathcal{O}_k \leftarrow n - l$ {shrink octant array}
22: **end for**

Both `Refine` and `Coarsen` do not change the parallel partition $(k_p, o_p)$, which is an important invariant from an application perspective. This leaves the possibility that a family of octants is prevented from coarsening when it is split between multiple processes. This case can be addressed by an incremental repartitioning [30], which has the same invariants as `Partition` and can in fact be included in that algorithm.

**3.3. Parallel partition and load-balance.** Parallel partitioning generally refers to a redistribution of mesh primitives among processes without changing the global mesh topology (i.e., the global number of primitives or their neighborhood relations). The objective is most often to achieve load-balance (i.e., to equidistribute the computational work uniformly among processes), which is necessary to ensure parallel scalability of an application.

We provide such an operation via the `Partition` algorithm. It relies entirely on the $z$-ordering of the forest, which implies that the relative order of octants with respect to processes stays the same: for two octree-octant pairs $(k_1, o_1) \leq (k_2, o_2)$, this implies that the first pair is assigned to a lower process than the second, or the

same process, both before and after the execution of `Partition`. On the one hand, this is a restriction in comparison to graph-based partitioning schemes, where mesh primitives can be arbitrarily reordered. On the other hand, this additional structure allows for a highly efficient parallel partitioning scheme which works extremely well in practice (see the performance results in section 4).

While an equipartition by local octant counts is often sufficient for satisfactory load-balance, some applications associate varying amounts of computational work with the octants. An example are *hp*-adaptive finite element discretizations, where each octant is interpreted as an element with a variable degree of approximation and thus a variable number of mathematical unknowns to store and compute on. This observation motivates a partition that distributes the work uniformly—not the number of octants—leading to local octant counts that vary among processes. We provide this capability by allowing for a user-specified weight function that returns a nonnegative integer weight for each octant and creating a partition that is evenly distributed by weight.

We realize the weighted partition in a two-step process. First, we compute the target octant counts for each process based on the weights. Second, we determine the new octree boundaries between processes and send the octants to their new owner process. We call the latter algorithm `Partition_given` since it is given a globally shared array $(N_p)$ of target octant counts per process $p$. For the common case of distributing the number of octants uniformly, this array can be computed according to (2.5) without communication. `Partition_given` is more complicated for a forest than a single octree since the nonempty octrees can change on each process; otherwise it is similar to existing single-octree algorithms and thus omitted here.

To create a partition where the octants may have different weights we proceed as follows. Let $W_p$ be the cumulative weight across all processes lower than and not including $p$. This can be computed after communicating the process-local weight sums and makes $W_P$ the global weight which needs to be divided evenly among processes. We split $W_P$ into $P$ ranges that are equal up to the remainder of integer division (this is not a limitation in practice since we use 64-bit integers for the weights). Each boundary between two equiweight ranges identifies one "cut" process that will hold octants of only the higher-weight range. Each process $p$ can determine the cut processors $(p_{\text{low}}, \ldots, p_{\text{high}})$ that intersect its current range of octants as

$$S_p = \left\{ p' \;\middle|\; W_p < \left\lfloor \frac{p' W_P}{P} \right\rfloor \le W_{p+1} \right\}, \qquad p_{\text{low}} = \min_{p'} S_p, \qquad p_{\text{high}} = \max_{p'} S_p, \tag{3.1}$$

which implies either $S_p = \emptyset$ or $0 < p_{\text{low}} \le p_{\text{high}} \le P$. This formula guarantees that the cuts computed on different processes are contiguous and do not overlap. Each process then examines the local cuts $p' \in \{p_{\text{low}}, \ldots, p_{\text{high}}\}$, determines for each cut the octant whose weight matches the end of a range, and sends its global octant index to both $p' - 1$ and $p'$. `Partition` is listed in Algorithm 16.

**3.4. 2:1 balance of neighbor size relations.** Many adaptive-mesh numerical applications are built on the assumption that neighboring mesh elements are either of the same size or at most half or twice as big. `Refine` and `Coarsen` generally destroy this invariant, which necessitates a separate 2:1 `Balance` algorithm to reestablish it. Our implementation combines the insulation layer and noniterative communication proposed in [34] with the concept of exterior octants and interoctree transformations as introduced earlier in this article. We implement full face, edge, and corner balance; face-only and edge-only versions can be derived as a special case.

**Algorithm 16** `Partition` (`callback Do_weight`)

1: $W[0] \leftarrow 0$ ; $W[p+1] \leftarrow 0$ ; $m \leftarrow 0$ {sum local weights; count local octants}
2: **for** $k \in \mathcal{T}_{\text{local}}$ **do**
3: **for** $o \in \mathcal{O}_k$ **do**
4: $w[m] \leftarrow$ `Do_weight` $(k, o)$ ; $W[p+1] \leftarrow W[p+1] + w[m]$ ; $m \leftarrow m+1$ {local sum}
5: **end for**
6: **end for**
7: `Allgather` $(W[1, \ldots, P])$ {communicate weight sums shifted by 1}
8: **for** $p' \in \{1, \ldots, P\}$ **do**
9: $W[p'] \leftarrow W[p'] + W[p'-1]$ {compute global cumulative weight sums}
10: **end for**
11: **for** $i \in \{0, \ldots, m-1\}$ **do**
12: $w[i] \leftarrow w[i] + W[p]$ {compute global cumulative weight for local octants}
13: **end for**
14: Determine cuts $p_{\text{low}}$, $p_{\text{high}}$ as in (3.1)
15: **for** $p' \in \{p_{\text{low}}, \ldots, p_{\text{high}}\}$ **do**
16: **if** $p' < P$ **then**
17: Binary search lowest $i$ in array $w$ such that $w[i] \geq \lfloor p' W_P / P \rfloor$ {guarantees $0 \leq i < l_p$}
18: Send global index $G_p + i + 1$ to process $p'$ {index of first octant that will be on $p'$}
19: **end if**
20: Send global index $G_p + i + 1$ to process $p' - 1$; if $p' = P$ instead send $G_P$
21: **end for**
22: Receive $i_{\text{low}}$ unless $\lfloor p W_P / P \rfloor = 0$ and $i_{\text{low}} \leftarrow 0$ {sender process is inferred from (3.1)}
23: Receive $i_{\text{high}}$ unless $\lfloor (p+1) W_P / P \rfloor = 0$ and $i_{\text{high}} \leftarrow 0$
24: Compute target local octant count $N[p] \leftarrow i_{\text{high}} - i_{\text{low}}$
25: `Allgather` $(N)$ {communicate target octant counts}
26: `Partition_given` $(N)$ {execute parallel partition with given octant counts}

Our `Balance` process (see Algorithm 17) consists of prebalance, identification and communication of insulation octants, and postbalance. The prebalance step referred to as `Balance_octree` executes 2:1 balance independently for each locally stored octree. Arbitrary single-octree algorithms can be used here; we choose a local-octree version of Algorithm 8 in [34]. Having established balanced local octrees it remains to correct imbalance across interprocess and interoctree boundaries. To this end we traverse all octants on these boundaries and construct their extended insulation layers. The insulation layer of an octant is given by its $3^d$ neighborhood; for the forest we need to include exterior octants that reach outside of the root octree. The insulation octants are transformed into their potential neighbor-octree coordinate systems and the range of their process ownership is examined; a boundary octant is sent to all possible owner processes covered by its insulation layer. The latter operation is achieved by `Balance_schedule`.

Once the insulation octants are sent, the receiving processes determine the octants that fall into all foreign octants' insulation layers and send those back when they are small enough to case a subdivision. This is handled by `Balance_response` (Algorithm 18). We use the additional functions `Effective_edge_octant` and `Effective_corner_octant` to replace a response octant with the largest octant closest to the boundary that achieves an equivalent subdivision; this reduces overlap in the postbalance step. The communication pattern in this schedule-response cycle is asymmetric; hence we perform a global encoding of receiver ranges using an `Allgather` operation on a small data set (in our case 50 integers per process). This encoding may be avoided by a coarse-blocked partition at the cost of suboptimal load-balance [34]; we found no need for this additional complication.

All received octants are integrated into the local octree storage which introduces exterior octants and causes overlap due to multiple edge or corner connec-

**Algorithm 17** `Balance`

```
 1: for k ∈ T_local do
 2:    Balance_octree (k)                                   {2:1-balance local octree k}
 3:    for o ∈ O_k do
 4:       for s ∈ I_o do
 5:          if s is outside of octree k across a corner c then
 6:             CC ← Corner_connections (k, c)              {nonredundant connections}
 7:             for C ∈ CC do
 8:                o' ← Transform_corner (C, o, 0)          {original octant lies outside of k'}
 9:                s' ← Transform_corner (C, s, 1)          {insulation octant lies inside of k'}
10:                Balance_schedule (k, C.k', o', s')
11:             end for
12:          else if d = 3 and s is outside of octree k across an edge e then
13:             EC ← Edge_connections (k, e)                {nonredundant connections}
14:             for E ∈ EC do
15:                o' ← Transform_edge (e, E, o, 0)         {original octant lies outside of k'}
16:                s' ← Transform_edge (e, E, s, 1)         {insulation octant lies inside of k'}
17:                Balance_schedule (k, E.k', o', s')
18:             end for
19:          else if s is outside of octree k across a face f connected to k' then
20:             o' ← Transform_face (k, f, o)               {original octant lies outside of k'}
21:             s' ← Transform_face (k, f, s)               {insulation octant lies inside of k'}
22:             Balance_schedule (k, k', o', s')
23:          else
24:             Balance_schedule (k, k, o, s)               {insulation octant within same octree}
25:          end if
26:       end for
27:    end for
28: end for
29: Determine communication pattern; sort and send scheduled insulation octants
30: for processes p' sending insulation layer to p do
31:    Receive insulation octree-octant pairs from p' into array A
32:    Balance_response (p', A) ; send response octree-octant pairs
33: end for
34: for processes p' sending response to p do
35:    Receive response octree-octant pairs from p'
36: end for
37: for k ∈ T_local do
38:    Merge received insulation and response octants into matching octree k
39:    Balance_octree_extended (k)          {received octants may lie outside of the root octree}
40: end for
```

tions. The postbalance needs to take this into account; we denote this operation by `Balance_octree_extended`. As a final step it culls all overlapping coarse octants and those outside of the octree or process boundary to yield a nonoverlapping and globally balanced forest. For the sake of brevity we omit several optimizations implemented in `p4est`; for example, the case when a process does not receive any octants and postbalance is omitted entirely. The cost of schedule-response computation and communication scales with the size of the parallel and intertree boundary surfaces and is dominated by the pre- and postbalance cost on the volume, $\mathcal{O}(N_p \log N_p)$.

**3.5. Creating the ghost layer.** For certain applications using a forest of octrees, such as building a finite element mesh (see section 3.6), it is useful to know which foreign-process octants neighbor the local ones. The implementation in `p4est` is based on `Find_owners` (Algorithm 13, which handles all interoctree transformations) and described in Algorithm 19. We prerequire a 2:1 neighborhood size relation for `Ghost`; hence it is sufficient to communicate each process-boundary octant to the owners of all its potential half-size neighbors. We construct these by variants of Algorithms 5,

**Algorithm 18** `Balance_response` (`int` $p'$, `array` $A$)

```
1: for (k, o) ∈ A do
2:    k_0 ← First_local_descendant (k) ; k_1 ← Last_local_descendant (k)
3:    for s ∈ I_o, s inside of octree k do
4:       (s_0, s_1) ← Descendants (s)          {find first and last smallest octants inside s}
5:       if s_1 < k_0 or k_1 < s then
6:          Skip s since it lies outside of the octants on this octree and process
7:       end if
8:       Binary search for first octant q_0 ∈ O_k with q_0 ≥ s          {do not use s_0 here}
9:       Binary search for last octant q_1 ∈ O_k with q_1 ≤ s_1
10:      if o is outside of octree k across a corner c then
11:         q* ← Effective_corner_octant (s, q_0, q_1, c)
12:         CC ← Corner_connections (k, c)          {nonredundant connections}
13:         for C ∈ CC do
14:            q' ← Transform_corner (C, q*, 0)
15:            Add (C.k', q') to send buffer for process p'          {q' lies outside of octree C.k'}
16:         end for
17:      else
18:         for q ∈ {q_0, ..., q_1} ⊂ O_k do
19:            if d = 3 and o is outside of octree k across an edge e then
20:               q* ← Effective_edge_octant (s, q, e)
21:               EC ← Edge_connections (k, e)          {nonredundant connections}
22:               for E ∈ EC do
23:                  q' ← Transform_edge (e, E, q*, 0)
24:                  Add (E.k', q') to send buffer for process p'          {q' lies outside of octree E.k'}
25:               end for
26:            else if o is outside of octree k across a face f connected to k' then
27:               q' ← Transform_face (k, f, q)
28:               Add (k', q') to send buffer for process p'          {q' lies outside of octree k'}
29:            else
30:               Add (k, q) to send buffer for process p'          {q' lies in the same octree k}
31:            end if
32:         end for
33:      end if
34:   end for
35: end for
36: Remove duplicates from send buffer to process p'
```

6, and 7. The ordered traversal of local boundary octants automatically arranges the received octree-octant pairs in the correct sorted sequence which allows for fast binary searches for ghost octants. `Ghost` is a useful major interface algorithm that provides a third-party application with the complete parallel neighborhood information. This reduces the communication pattern in the application to simple point-to-point transfers of numerical information. Should more than one layer of ghost octants be desired, they can be obtained with relative ease by extending Algorithm 19 for this purpose.

**3.6. Globally unique node numbering.** While `Ghost` is a useful interface for applications that number their numerical unknowns internally, we provide the algorithm `Nodes` for applications that associate unknowns with octant vertices exclusively, such as trilinear finite element codes or visualization packages. For 2:1 balanced adaptive meshes this is complicated by the existence of hanging nodes, which occur on double-size neighbor faces or edges. Only nonhanging nodes are independent and associated with unknowns (see Figure 5). Each independent node is assigned to one owner process. The nodes are numbered globally in sequence of their owner processes. When an independent node is on an octree boundary, it needs to be canonicalized by transforming it into the coordinate system of the lowest participating octree. The node is then subjected to the same $z$-order as the octants in this tree, and the own-

**Algorithm 19** `Ghost`

```
1: for k ∈ T_local do
2:    for o ∈ O_k do
3:       FN ← half-size possible face neighbors of o
4:       EN ← half-size possible edge neighbors of o          {remove this line for d = 2}
5:       CN ← half-size possible corner neighbors of o
6:       for o' ∈ FN ∪ EN ∪ CN do
7:          OP ← Find_owners (k, o')                {transforms between octrees if necessary}
8:          for p' ∈ OP, p' ≠ p do
9:             Add (k, o) to send buffer for process p'
10:         end for
11:      end for
12:   end for
13: end for
14: Send ghost octree-octant pairs
15: Receive ghost octree-octant pairs into sections of array G ordered by sending process
16: return G
```

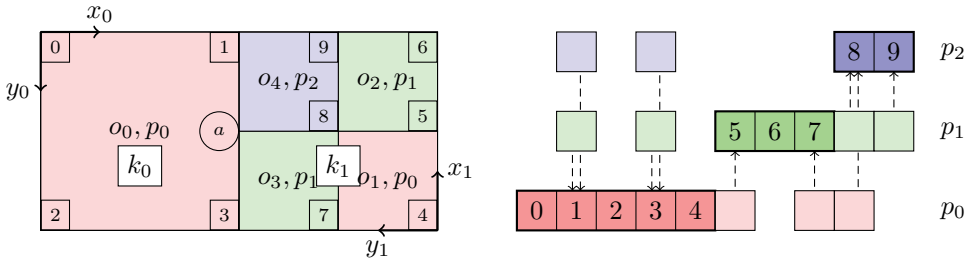



FIG. 5. *Node numbering for a 2D example mesh on two octrees $k_0$ and $k_1$. Left: schematic view of the forest consisting of five octants $o_0, \ldots, o_4$ partitioned across three processes $p_0$, $p_1$, $p_2$ with octant counts $(2, 2, 1)$. The boxed numbers indicate the globally unique numbers for the independent nodes $0, \ldots, 9$. There is one hanging node a. The numbering sequence of octants and nodes follows the z-order with respect to the octree coordinate axes $x_i, y_i$. Nodes are canonicalized; i.e., when shared between multiple octrees they are assigned to the lowest-numbered octree ($k_0$ in this situation). Right: sharing information for independent nodes. One row is drawn for each process containing the owned nodes including their global numbers and the shared nodes. The dependence of shared nodes on their owner processes is indicated by vertical arrows. Due to the hanging node a, node 1 is shared by $p_1$ and node 3 by $p_2$.*

ership of the containing octant determines the owner process of the node.

`Nodes` is described in Algorithm 20. After identifying hanging node status and determining the owner process of all local and boundary nodes, the $L_p$ local independent nodes are numbered. The globally unique numbering is then determined by communicating the array $L_p$ and adding the cumulative sum $\sum_{p'}^{p-1} L_{p'}$ to the local node numbers. In the process of sending owned local node numbers to all sharing processes we also attach the list of all sharers, which is necessary for the scatter/gather operations in a numerical application; see Figure 5. Due to the binary searches in both the local octant and ghost arrays the runtime of `Nodes` is $\mathcal{O}(N_p \log N_p)$.

**3.7. Parallel forest checksum.** It is often helpful to compute a short forest identifier that is unique with high probability, for example, for regression testing. Hash functions or checksums can be used for this purpose. However, these often require a complete serial pass through the data, which precludes efficient parallelization. An algorithm that does not suffer from this limitation is the Adler-32 checksum [17]. It includes the function `adler32_combine`, which computes the checksum of two combined blocks of data from the individual checksums of the two data blocks and the data

**Algorithm 20** `Nodes` (`ghost array` $G$)

**for** $k \in \mathcal{T}_{\text{local}}$ **do**
  **for** $o \in \mathcal{O}_k$, $\tilde{c} \leftarrow$ `Child_id` $(o)$ **do**
    **for** $c \in \{0, \ldots, 2^d - 1\}$ **do**
      **if** $c$ is center of parent face $f$ **and** double-size face neighbor $\in \mathcal{O}_{\text{local}} \cup G$ **then**
        $N(k, o, c) \leftarrow$ `face_hanging`
        **for** $\xi \in \{0, \ldots, 3\}$, $c' \leftarrow \mathcal{V}_3(f, \xi)$, $c' \notin \{c, \tilde{c}\}$ **do**
          $N(k, o, c') \leftarrow$ `edge_hanging` {remove this loop for $d = 2$}
        **end for**
      **end if**
      **if** $c$ is center of parent edge $e$ **and** any double-size edge neighbor $\in \mathcal{O}_{\text{local}} \cup G$ **then**
        $N(k, o, c) \leftarrow$ `edge_hanging` {remove this if-condition for $d = 2$}
      **end if**
    **end for**
    Find independent nodes that hanging nodes depend on
  **end for**
**end for**
Canonicalize and sort all $I_{\text{local}}$ independent nodes; determine their owner processes
Number $L[p]$ owned local independent nodes contiguously {$L[p] \leq I_p$}
`Allgather` $(L)$ {communicate number of owned nodes on all processes}
Determine communication pattern {same encoding as used in `Balance`, see section 3.4}
Send nonowned independent nodes to their owners {$p$ is a sharer of nonowned nodes}
Receive nodes and identify senders as sharers for each owned independent node
For individual queried nodes send local node number and list of sharers to all sharers

length of the second in $\mathcal{O}(1)$ operations. We exploit this capability by first constructing the local checksums in parallel on each process, passing the integer coordinates $x$, $y$, $z$ and the level $\ell$ of all local octants in order to the `adler32` function. Then we execute one `Gather` operation of local checksums and data lengths and combine the local checksums efficiently on process 0. An `Allgather` call could be used instead if so desired. This parallel checksum is entirely integer-based and thus does not suffer from roundoff errors. Moreover, it is independent of the parallel partition.

**4. Numerical results.** In this section we illustrate the variety of meshes that can be created with `p4est` and demonstrate large-scale parallel scalability. We include synthetic examples, where domain and refinement rules are chosen for instructive purposes, and meshes created from geophysical simulations. For the latter, adaptive mesh refinement is essential to resolve the multiscale features of the physical system, while high parallel efficiency is needed due to the large number of unknowns occurring in such simulations. We conduct weak and strong scaling studies and examine the quality of the parallel partition in terms of parallel neighbor faces and processes. Our results indicate that `p4est` performs well in terms of both absolute runtime and overhead incurred due to parallelism.

**4.1. Synthetic examples.** We provide illustrations of both 2D and 3D forests in Figure 6. In both cases, the initial refinement is created by the algorithm `New` followed by a recursive invocation of `Refine`. Then `Balance` and `Partition` are executed to ensure 2:1 neighbor size relations and a load-balanced parallel partition. These examples demonstrate how periodicity is realized and show that octrees can connect to each other in non-Cartesian configurations.

*Weak scalability.* We use the six-octree forest configuration shown on the right of Figure 6 to study "weak" scalability (i.e., the problem size is proportional to the number of processes) for a fractal-type recursive refinement. Using the observation that an increase in the refinement level yields 8 times as many octants, we multiply the process count by 8 for each increment in level. These computations are performed on

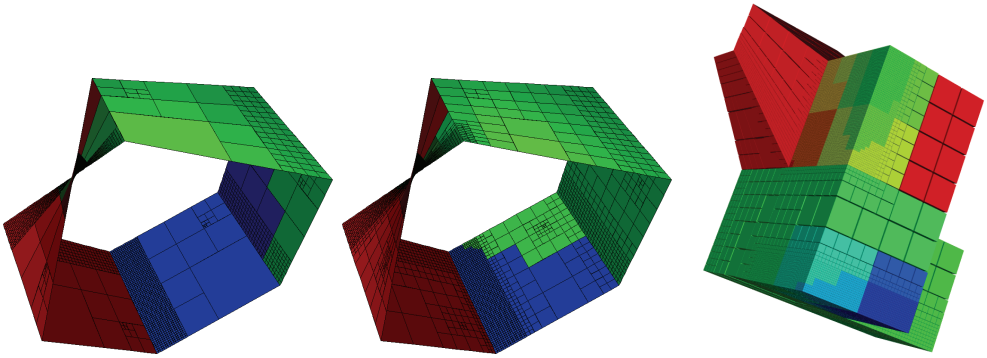


FIG. 6. *Examples of forest-of-octree configurations where color encodes the process number. Left: 2D forest of five octrees that realize the periodic Möbius strip, here shown after initial calls to* `New` *and* `Refine`*. Middle: the same forest after* `Balance` *and* `Partition`*. Right: 3D forest composed of six cubes whose orientations are rotated against each other, with five octrees connecting through the horizontal central axis, after calls to* `New`*,* `Refine`*,* `Balance`*, and* `Partition`*.*

the *Jaguar* Cray XT5 supercomputer, which has 16 GB of memory per 12-core node. We display the measured runtimes of the main `p4est` algorithms in Figure 7. The diagram on the left-hand side shows runtime percentages and gives rise to the following observation: The runtime of `New`, `Refine`, and `Partition` is negligible (`Coarsen` is not used here but is as fast as `Refine`), and `Balance` and `Nodes` consume over 90% of the total runtime. The diagram on the right-hand side of Figure 7 displays the absolute runtimes of the two most expensive algorithms `Balance` and `Nodes` normalized by one million octants per process. The runtimes rise mildly from roughly 6 seconds for 12 processes to between 8 and 9 seconds for 220,320 processes. Thus the parallel efficiency is 65% for `Balance` and 72% for `Nodes` for an 18,360-fold increase in process count.

*Scalability with the number of trees.* We assess the performance impact of large numbers of octrees by assembling a cubic domain from multiple trees that are then `Refine`'d and `Balance`'d in turn. To obtain the results displayed in Figure 8, we compare timings on the *Ranger* Sun/AMD supercomputer (which has 32GB of memory on each 16-core node) for a fixed overall number of octants. We find that the influence of the number of octrees on the runtime is rather weak and that they reach up to several million octants. Moreover, instead of using one octree for a given subdomain, multiple octrees with fewer octants each can even be beneficial due to improved cache performance. In Figure 9 we display 2D and 3D example connectivities with tens of thousands of octrees and demonstrate good scalability of `Balance` up to 4,096 cores and several hundred million octants.

**4.2. Parallel adaptive geophysical simulations.** We discuss two examples motivated by geophysical phenomena to further illustrate the performance of `p4est`.

The first geophysical application that we describe is the simulation of seismic waves propagating through the earth. The numerical portion of the code (called `mangll` [39]) is based on a discontinuous Galerkin discretization which passes information exclusively through octant faces. Therefore, we use a face-only variant of `Ghost` (Algorithm 19) to interface to `mangll`. To cover the solid sphere with a mesh we combine 13 octrees into a forest as described in Figure 10.

*Strong scalability.* We create the globally wavelength-adapted spherical mesh by iterative rounds of nonrecursive `Refine`, `Partition`, and `Balance` (see Figure 11). We start with 13 root octants (one for each octree) and eventually reach a final octant

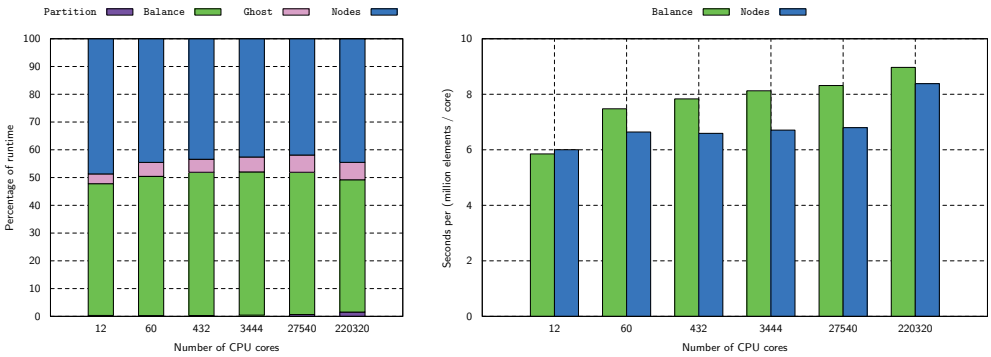
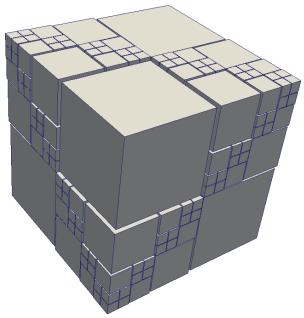



FIG. 7. *"Weak" scaling results up to* 220,320 *processes on* Jaguar. *The refinement is defined by choosing the same six-cube* 3*D connectivity as used on the right-hand side of Figure* 6*, and recursively subdividing octants with child identifiers* 0*,* 3*,* 5*, and* 6 *while not exceeding four levels of size difference in the forest. This leads to a fractal mesh structure. To scale from* 12 *to* 220,320 *processes the maximum refinement level is incremented by one while the number of processes is multiplied by* 8*. Left: runtime is dominated by* `Balance` *and* `Nodes` *while* `Partition` *and* `Ghost` *together take up less than* 10% *(*`New` *and* `Refine` *are negligible and not shown). Right: performance assessed by normalizing the time spent in the* `Balance` *and* `Nodes` *algorithms by the number of octants per process which is held constant at approximately* 2.3 *million (ideal scaling would result in bars of constant height). The largest mesh created contains over* $5.13 \times 10^{11}$ *octants and is* `Balance`*'d in* 21 *seconds.*



| 2D | CPU cores | | |
|---|---|---|---|
| $j$ | 1 | 64 | 4,096 |
| 0 | 0.180 | 0.284 | 1.15 |
| 2 | 0.158 | 0.261 | 0.28 |
| 5 | 0.371 | 0.167 | 0.28 |
| 8 | | 0.616 | 0.19 |
| 11 | | | 1.24 |

| 3D | CPU cores | |
|---|---|---|
| $j$ | 1 | 4,096 |
| 0 | 0.94 | 4.86 |
| 1 | 1.69 | 4.63 |
| 2 | 2.60 | 4.56 |
| 3 | 4.51 | 4.07 |
| 4 | | 3.55 |
| 5 | | 3.69 |
| 6 | | 5.64 |
| 7 | | 10.8 |

FIG. 8. *Scalability with respect to large numbers of trees on* Ranger. *For each column we create a uniform mesh of a certain level* $J$ *(identified by the last filled entry). The composition of this mesh differs for each row* $j$*: it is built of* $2^{jd}$ *trees with* $2^{(J-j)d}$ *octants each. Every octant is then refined up to* 4 *levels in a fractal pattern shown in the left-hand picture. We list the run times in seconds of a subsequent* `Balance` *call. The spread on* 4,096 *cores between* 1 *tree and over* 4 *million trees in* 2*D (*$192 \times 10^6$ *quadrants) is* 6.5*, and that between* 1 *tree and* 2 *million trees in* 3*D (*$1.34 \times 10^9$ *octants) is* 3*.*

number that depends on the earthquake frequency (e.g., $8.6 \times 10^9$ for 1 Hz). Since each refinement increases the octant count by at most a factor of eight, it takes multiple cycles to reach at least one octant per process for the larger parallel runs. Compared to one recursive call of `Balance`, this process is likely more expensive but also more conservative in maintaining a load-balanced partition between cycles. Due to the multiple rounds and many empty processes, this is an aggressive test of the `p4est` algorithms. "Strong" scalability for problems with three different frequencies is good over a wide range of process counts.

*Partition quality.* We assess partition quality by measuring the number of inter-process neighbor faces and process neighbor counts for an earthquake of frequency

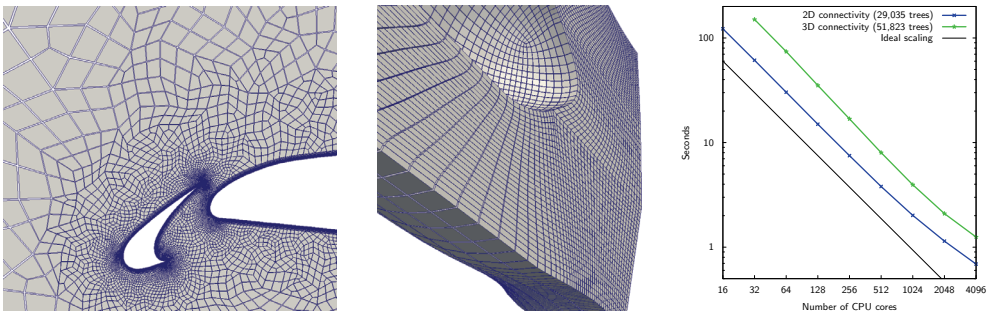



FIG. 9. *Complex 2D and 3D connectivities. The left picture shows a zoom around the front tip of an airfoil with boundary-layer adaptation; this connectivity consists of* 29,035 *quadtrees. The connectivity in the middle consists of* 51,823 *octrees. The diagram on the right shows "strong" scaling of* `Balance` *on* Ranger*, resulting in* 342 *million quadrants for two dimensions and* 264 *million octants for three dimensions, respectively.*

0.14 Hz. The results are shown in Table 4. The small ratio between maximum and average counts indicates a well-balanced parallel partition. The number of neighbor processes is essentially independent of the process counts. All of these metrics become better instead of worse for increasing process counts, which lets us conclude that the forest of octrees is an effective and reliable approach to large-scale partitioning.

A second geophysical application is implemented through the mantle convection code Rhea; see Figure 12. It solves the Stokes equations either instantaneously or on a dynamically evolving mesh. Earth's mantle is a hollow sphere which we divide into six spherical caps of four octrees each, creating a 24-octree forest. Rhea is based on a trilinear finite element discretization which requires face, edge, and corner connectivity and uses parallel algebraic multigrid, which requires a global node numbering. Consequently we interface Rhea to `p4est` via `Nodes` (Algorithm 20). We have successfully scaled Rhea/`p4est` to 122,880 processes of *Jaguar*.

**5. Conclusions.** In this article we present new geometric and parallel algorithms that enable dynamic adaptive mesh refinement and coarsening (AMR) on forest-of-octree meshes in both two dimensions and three dimensions. We have encapsulated these algorithms in the `p4est` software library, which is designed to extend the benefits of scalable single-octree codes to more flexible geometries.

We introduce three key concepts as the basis of our algorithms. The first is an encoding of the connectivity relations between octrees that allows flexible configurations of connected octrees, including 2D manifolds embedded in three dimensions (such as the Möbius strip and manifolds of arbitrary genus) and geometries with mixed periodic and nonperiodic boundary connectivity, such as solid and hollow spheres. The second concept is exterior octants that exist in the coordinate system of an octree but outside of its root domain, and transformations of both interior and exterior octants between octrees that are connected across faces, edges, or corners in arbitrary relative orientations. Third, we create a total ordering of octants by connecting the space-filling $z$-curves that exist separately in all octrees in the sequence of the octree numbering, which enables fast load-balancing and determination of process ownership.

We provide both synthetic and simulation-driven examples and assess weak and strong parallel scalability of our algorithms. All `p4est` routines scale well to the full size of a Cray XT5 supercomputer with over 220,000 CPU cores. Our results indicate that the transformations between octrees affect neither the quality of the parallel par-

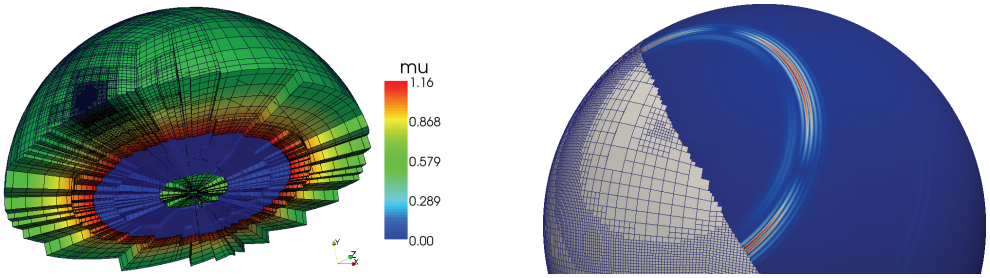



FIG. 10. *Left: Cut view through a spherical mesh that is adaptively refined around the source location of an earthquake. The refinement inside the small region enclosing the source is designed to match the local octant size to the wavelengths that are expected during an earthquake of frequency* 0.05 *Hz. The color encodes the shear modulus in the earth which is zero in the outer core (blue). The forest consists of a center cube and six spherical caps around it to create the inner core (green) and six more spherical caps to create outer core and mantle. The deformation of the mesh is chosen such that the spherical boundary between outer core and mantle is mapped exactly by a surface between level-3 octants. Right: Snapshot from an earthquake simulation where the mesh is dynamically tracking the wave fronts. Both pictures show pronounced nonuniform refinement. (Simulation performed using* `p4est` *via* `mangll` [39].*)*

TABLE 4

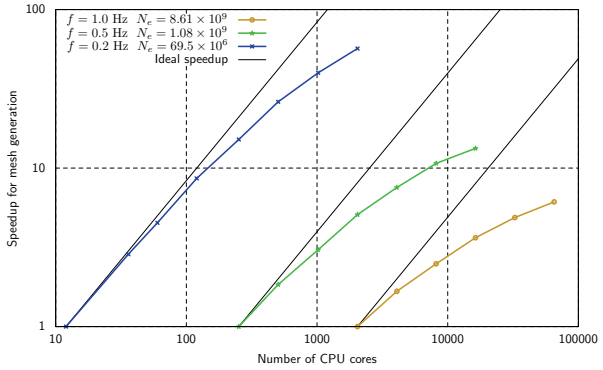
*Parallel neighbor face and process counts for a fixed-size earthquake simulation at* 0.14 *Hz. The mesh contains* 21 *million octants. From low to high process counts the surface-to-volume ratio of process partitions shrinks, which explains the reduction of neighbor faces per process with a factor of* $2^{2/3}$. *The average number of process face neighbors stays remarkably constant at* 11.5 *(the theoretical optimum for a uniform mesh would be* 6*), which indicates good partition quality. For both neighbor faces and processes the uniformity of the partition gets better with increasing process counts, indicated by the shrinking maximum-to-average ratio.*

| #Processes | #Neighbor faces | | | #Neighbor processes | | |
|---|---|---|---|---|---|---|
| | avg | max | ratio | avg | max | ratio |
| 8160 | 1460 | 4858 | 3.32 | 11.3 | 37 | 3.21 |
| 16320 | 936 | 2659 | 2.84 | 11.4 | 29 | 2.52 |
| 32640 | 589 | 1445 | 2.45 | 11.4 | 29 | 2.52 |
| 65280 | 373 | 834 | 2.34 | 11.5 | 24 | 2.09 |
| 130560 | 237 | 490 | 2.07 | 11.5 | 25 | 2.17 |
| 224220 | 165 | 324 | 1.96 | 11.5 | 26 | 2.25 |

tition nor the performance of the code. We maintain small scale-independent numbers of neighbors per process while scaling to 224,220 processes. We execute the 2:1 balance operation on over $5.13 \times 10^{11}$ octants on 220,320 processes in less than 10 seconds per million octants per process, or 21 seconds total. This capability for scalable AMR on general geometries that can scale to contemporary petascale supercomputers is allowing us to address previously intractable global mantle dynamics problems and should prove to be equally beneficial for a large class of multiscale problems governed by PDEs.

**Nomenclature.**

$\mathcal{CT}_{k,c}$ Neighboring octrees at corner $c$ of octree $k$
$\mathcal{CC}$ Computed nonredundant interoctree corner connections $C \in \mathcal{CC}$
$\mathcal{ET}_{k,e}$ Neighboring octrees at edge $e$ of octree $k$
$\mathcal{EC}$ Computed nonredundant interoctree edge connections $E \in \mathcal{EC}$




| #Refine | #octants |
|---|---|
| 0 | 13 |
| 1 | 55 |
| 2 | 433 |
| 3 | 3289 |
| 4 | 25465 |
| 5 | 200185 |
| 6 | 1587193 |
| 7 | 12640249 |
| 8 | 100892665 |
| 9 | 605554585 |
| 10 | 2857338721 |
| 11 | 8609009569 |

FIG. 11. *Fixed-size ("strong") scaling on* Jaguar *for the mesh generation process of an earthquake simulation where the mesh is refined throughout the globe to match the local wave speeds. Left: Speedup for three different problem sizes induced by earthquake frequencies* 0.2 *Hz,* 0.5 *Hz, and* 1 *Hz (the octant number grows with the third power of the frequency). Right: The mesh generation process for the largest run at* 1 *Hz creates* $8.6 \times 10^9$ *octants. The common starting point for all frequencies is a macro mesh consisting of* 13 *root octants arranged as described in Figure* 10 *that are then* `Refine`*'d,* `Partition`*'ed, and* `Balance`*'d iteratively. The process involves* 11 *calls to* `Refine` *and* `Partition` *and* 10 *calls to* `Balance` *and yields a mesh that does not depend on the number of processes. While for the* 1 *Hz meshing example the speedup is largest for* 65,536 *processes, we run this example successfully on up to* 220,320 *processes, where only cycles* 6 *and onward have at least one octant per process; this demonstrates why it is necessary to allow empty processes in* `p4est`*. For* 220,320 *processes the complete mesh generation still takes only* 32 *seconds.*

$\mathcal{I}_o$ Insulation layer of octant $o$
$P$ Number of processes
$p$ Index of local process (rank), $p \in \{0, \ldots, P-1\}$
$N_p$ Number of local octants on process $p$ counting across all local octrees
$G_p$ Global number of first octant on process $p$
$L_p$ Global number of first independent node on process $p$
$\mathcal{T}$ All octrees
$k_p$ First nonempty octree on process $p$
$o_p$ First level-$b$ descendant $o_p$ of first local octant on process $p$
$k_{\text{last}}$ Last nonempty octree on process $p$
$\mathcal{T}_{\text{local}} = (k_p, \ldots, k_{\text{last}})$ Nonempty process-local octrees
$\mathcal{O}_k$ Process-local octants in octree $k$
$\mathcal{O}_{\text{local}}$ All process-local octants $\cup_k \mathcal{O}_k$

**Acknowledgments.** The authors would like to thank Tobin Isaac for additional debugging and for writing the "brick" connectivity used in Figure 8. We enjoyed rewarding discussions with Rahul Sampath, Georg Stadler, and Tiankai Tu. We also thank Georg Stadler for generating the plots used in Figure 12. We appreciate the feedback provided by Wolfgang Bangerth and Timo Heister while interfacing `deal.II` to `p4est`. The 2D and 3D connectivities used in Figure 9 are extracted from `deal.II` and were originally created by Mike Anderson and Benjamin Kirk, respectively. The authors thank the Texas Advanced Computing Center (TACC) for providing us with access to the Ranger supercomputer under TeraGrid award MCA04N026, and the National Center for Computational Science (NCCS) for early-user access to the *Jaguar* Cray XT5 supercomputer.

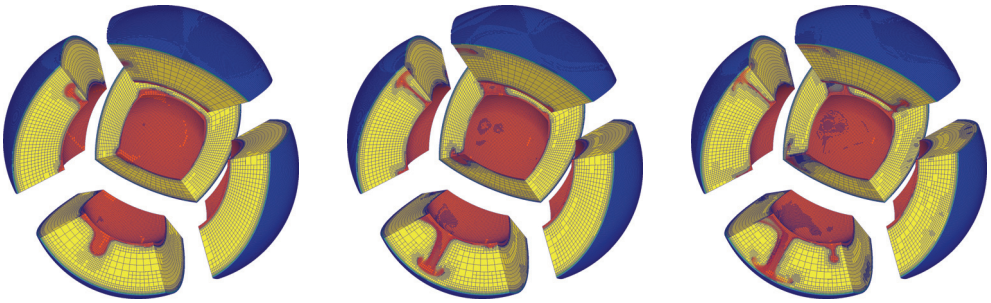


FIG. 12. *Exploded views of a mantle convection simulation. We split earth's mantle into six spherical caps of* $2 \times 2$ *octrees each (five of these caps are shown above). This creates a total of* 24 *octrees of an approximately uniform aspect ratio. Perturbations in the initial condition grow over time into plumes of hot material (red) which rise to the top of the mantle. The mesh is adapted frequently to track moving regions of high thermal gradients, which leads to sharply localized refinement. From left to right we show snapshots at* 395, 1000, *and* 1800 *time steps. (Simulation performed using* `p4est` *via Rhea* [7]. *Published under licence in* Journal of Physics: Conference Series *by IOP Publishing Ltd.)*

## REFERENCES


[1] V. AKÇELIK, J. BIELAK, G. BIROS, I. EPANOMERITAKIS, A. FERNANDEZ, O. GHATTAS, E. J. KIM, J. LOPEZ, D. R. O'HALLARON, T. TU, AND J. URBANIC, *High resolution forward and inverse earthquake modeling on terascale computers*, in SC '03: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, New York, 2003.

[2] W. BANGERTH, C. BURSTEDDE, T. HEISTER, AND M. KRONBICHLER, *Algorithms and data structures for massively parallel generic adaptive finite element codes*, ACM Trans. Math. Software, to appear.

[3] W. BANGERTH, R. HARTMANN, AND G. KANSCHAT, *deal.*II*—a general-purpose object-oriented finite element library*, ACM Trans. Math. Software, 33 (2007), article 24.

[4] B. K. BERGEN, F. HÜLSEMANN, AND U. RÜDE, *Is* $1.7 \times 10^{10}$ *unknowns the largest finite element system that can be solved today?*, in SC '05: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, New York, 2005.

[5] M. J. BERGER AND J. OLIGER, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comput. Phys., 53 (1984), pp. 484–512.

[6] G. L. BRYAN, T. ABEL, AND M. L. NORMAN, *Achieving extreme resolution in numerical cosmology using adaptive mesh refinement: Resolving primordial star formation*, in SC '01: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, New York, 2001.

[7] C. BURSTEDDE, M. BURTSCHER, O. GHATTAS, G. STADLER, T. TU, AND L. C. WILCOX, *ALPS: A framework for parallel adaptive PDE solution*, J. Phys. Conf. Ser., 180 (2009), 012009.

[8] C. BURSTEDDE, O. GHATTAS, M. GURNIS, E. TAN, T. TU, G. STADLER, L. C. WILCOX, AND S. ZHONG, *Scalable adaptive mantle convection simulation on petascale supercomputers*, in SC '08: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, New York, 2008.

[9] C. BURSTEDDE, O. GHATTAS, G. STADLER, T. TU, AND L. C. WILCOX, *Towards adaptive mesh PDE simulations on petascale computers*, in Proceedings of Teragrid '08, Las Vegas, NV, 2008.

[10] C. BURSTEDDE, O. GHATTAS, G. STADLER, T. TU, AND L. C. WILCOX, *Parallel scalable adjoint-based adaptive solution for variable-viscosity Stokes flows*, Comput. Methods Appl. Mech. Engrg., 198 (2009), pp. 1691–1700.

[11] A. C. CALDER, B. C. CURTIS, L. J. DURSI, B. FRYXELL, G. HENRY, P. MACNEICE, K. OLSON, P. RICKER, R. ROSNER, F. X. TIMMES, H. M. TUFO, J. W. TRURAN, AND M. ZINGALE, *High-performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors*, in SC '00: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, New York, 2000, pp. 56–56.

[12] P. Colella, J. Bell, N. Keen, T. J. Ligocki, M. Lijewski, and B. Van Straalen, *Performance and scaling of locally-structured grid methods for partial differential equations*, J. Phys. Conf. Ser., 78 (2007), pp. 1–13.

[13] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, *New challenges in dynamic load balancing*, Appl. Numer. Math., 52 (2005), pp. 133–152.

[14] R. A. Finkel and J. L. Bentley, *Quad trees A data structure for retrieval on composite keys*, Acta Inform., 4 (1974), pp. 1–9.

[15] 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*, J. Parallel and Distributed Computing, 47 (1997), pp. 139–152.

[16] U. Frisch, *Turbulence: The Legacy of A. N. Kolmogorov*, Cambridge University Press, Cambridge, UK, 1995.

[17] J.-L. Gailly and M. Adler, *A massively spiffy yet delicately unobtrusive compression library*, http://zlib.net/.

[18] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf, *The Cactus framework and toolkit: Design and applications*, in Vector and Parallel Processing—VECPAR '2002, 5th International Conference, Springer, New York, 2003.

[19] G. Karypis and V. Kumar, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, J. Parallel and Distributed Computing, 48 (1998), pp. 71–95.

[20] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, *`libMesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations*, Engineering with Computers, 22 (2006), pp. 237–254.

[21] A. Laszloffy, J. Long, and A. K. Patra, *Simple data management, scheduling and solution strategies for managing the irregularities in parallel adaptive hp finite element simulations*, Parallel Computing, 26 (2000), pp. 1765–1788.

[22] O. S. Lawlor, S. Chakravorty, T. L. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. V. Kalé, *ParFUM: A parallel framework for unstructured meshes for scalable dynamic physics applications*, Engineering with Computers, 22 (2006), pp. 215–235.

[23] J. Luitjens, B. Worthen, M. Berzins, and T. C. Henderson, *Scalable parallel AMR for the Uintah multiphysics code*, in Petascale Computing Algorithms and Applications, D. A. Bader, ed., Chapman and Hall/CRC, Boca Raton, FL, 2007.

[24] D. Meagher, *Geometric modeling using octree encoding*, Computer Graphics and Image Processing, 19 (1982), pp. 129–147.

[25] R. J. Meyers, T. J. Tautges, and P. M. Tuchinsky, *The hex-tet hex-dominant meshing algorithm as implemented in CUBIT*, in Proceedings of the 7th International Meshing Roundtable, Sandia National Laboratory, Albuquerque, NM, 1998, pp. 151–158.

[26] G. M. Morton, *A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*, Technical report, IBM Ltd., Armonk, NY, 1966.

[27] C. D. Norton, J. Z. Lou, and T. A. Cwik, *Status and directions for the pyramid parallel unstructured amr library*, in Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2001, p. 120.

[28] M. Paszyński, D. Pardo, C. Torres-Verdín, L. Demkowicz, and V. Calo, *A parallel direct solver for the self-adaptive hp-finite element method*, J. Parallel and Distributed Computing, 70 (2010), pp. 270–281.

[29] S. Popinet, *Gerris: A tree-based adaptive solver for the incompressible Euler equations in complex geometries*, J. Comput. Phys., 190 (2003), pp. 572–600.

[30] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros, *`Dendro`: Parallel algorithms for multigrid and AMR methods on* 2:1 *balanced octrees*, in SC '08: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, New York, 2008.

[31] R. S. Sampath and G. Biros, *A parallel geometric multigrid method for finite elements on octree meshes*, SIAM J. Sci. Comput., 32 (2010), pp. 1361–1392.

[32] G. Stadler, M. Gurnis, C. Burstedde, L. C. Wilcox, L. Alisic, and O. Ghattas, *The dynamics of plate tectonics and mantle flow: From local to global scales*, Science, 329 (2010), pp. 1033–1038.

[33] J. R. Stewart and H. C. Edwards, *A framework approach for developing parallel adaptive multiphysics applications*, Finite Elem. Anal. Des., 40 (2004), pp. 1599–1617.

[34] H. Sundar, R. S. Sampath, and G. Biros, *Bottom-up construction and* 2:1 *balance refinement of linear octrees in parallel*, SIAM J. Sci. Comput., 30 (2008), pp. 2675–2708.

[35] H. Sundar, R. S. Sampath, S. S. Adavani, C. Davatzikos, and G. Biros, *Low-constant parallel algorithms for finite element simulations using linear octrees*, in SC '07: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, New York, 2007.

[36] H. Tropf and H. Herzog, *Multidimensional range search in dynamically balanced trees*, Angewandte Informatik, 2 (1981), pp. 71–77.

[37] T. Tu, D. R. O'Hallaron, and O. Ghattas, *Scalable parallel octree meshing for terascale applications*, in SC '05: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, New York, 2005.

[38] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron, *From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing*, in Proceedings of SC2006, Tampa, FL, 2006.

[39] L. C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas, *A high-order discontinuous Galerkin method for wave propagation through coupled elastic-acoustic media*, J. Comput. Phys., 229 (2010), pp. 9373–9396.