SIAM J. SCI. COMPUT.
Vol. 39, No. 5, pp. C364–C392

COARSE MESH PARTITIONING FOR TREE-BASED AMR*

CARSTEN BURSTEDDE† AND JOHANNES HOLKE‡

Abstract. In tree-based adaptive mesh refinement, elements are partitioned between processes using a space-filling curve. The curve establishes an ordering between all elements that derive from the same root element, the tree. When representing complex geometries by connecting several trees, the roots of these trees form an unstructured coarse mesh. We present an algorithm to partition the elements of the coarse mesh such that (a) the fine mesh can be load-balanced to equal element counts per process regardless of the element-to-tree map, and (b) each process that holds fine mesh elements has access to the meta data of all relevant trees. As an additional feature, the algorithm partitions the meta data of relevant ghost (halo) trees as well. We develop in detail how each process computes the communication pattern for the partition routine without handshaking and with minimal data movement. We demonstrate the scalability of this approach on up to 917e3 MPI ranks and 371e9 coarse mesh elements, measuring run times of one second or less.

Key words. adaptive mesh refinement, coarse mesh, mesh partitioning, parallel algorithms, forest of octrees, high-performance computing

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

DOI. 10.1137/16M1103518

1. Introduction.

Adaptive mesh refinement (AMR) has a long tradition in numerical mathematics. Over the years, the technique has evolved from a pure concept to a practical method, where the transition occurred somewhere in the mid 1980s; see, for example, [6]. A second transition from serial to parallel computing, eventually parallelizing the storage of the mesh itself, occurred around the turn of the millennium (see, e.g., [23]). Different flavors of the approach have been studied, varying in the shape of the mesh elements used (triangles, tetrahedra, quadrilaterals, hexahedra) and in the logical grouping of the elements.

Elements actually may not be grouped at all but rather assembled into an unstructured mesh; some recent references are [36, 40, 55]. The connectivity of elements can be modeled as a graph, and the partitioning of elements between parallel processes can be translated into partitioning the graph. Finding an approximate solution to this problem has been the subject of extensive research, some of which has led to the development of software libraries [17, 18, 30]. In practice, the graph approach is often augmented by diffusive migration of elements. All in all, partitioning times of roughly 1e3–10e3 elements per second per process have been measured [16, 19, 48]. Thus we may think of approaching the partitioning problem differently, trading the generality of the graph for a mathematical structure that offers rates of maybe

*Submitted to the journal's Software and High-Performance Computing section November 15, 2016; accepted for publication (in revised form) July 24, 2017; published electronically October 10, 2017.

http://www.siam.org/journals/sisc/39-5/M110351.html

Funding: This work was supported by the Bundesministerium für Bildung und Forschung, the Deutsche Forschungsgemeinschaft, the Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg, and the Rheinische Friedrich-Wilhelms-Universität Bonn. The second author was supported by the Bonn International Graduate School for Mathematics (BIGS) as part of HCM.

†Institut für Numerische Simulation (INS) and Hausdorff Center for Mathematics (HCM), Rheinische Friedrich-Wilhelms-Universität Bonn, 53115 Bonn, Germany (burstedde@ins.uni-bonn.de).

‡Corresponding author. Institut für Numerische Simulation (INS) and Hausdorff Center for Mathematics (HCM), Rheinische Friedrich-Wilhelms-Universität Bonn, 53115 Bonn, Germany (holke@ins.uni-bonn.de).

100e3–1e6 elements per second per process.

When we group elements by their size h, a particularly strict ansatz is the hierarchical, $h = 2^{-\ell}$, using some refinement level $\ell \in \mathbb{Z}$. For quadrilateral/hexahedral elements, we may introduce a rectangular grid of equal-sized ones and reduce the assembly of the mesh to the question of arranging such grids relative to one another. Block-structured methods do just that in varying instances of generality, with some allowing free rotation and overlapping of blocks of any level [47], while others impose strict rules of assembly (such as not allowing rotation but only shifts in multiples of h [5]). Another such rule interprets elements as nodes of a tree, where the level ℓ takes on a second meaning as the depth of a node below the root [41].

Tree-based AMR can be implemented for any shape of element, where special 2D solutions exist [32] as well as the popular rectangles (2D) and cubes (3D) approach. The tree root is identified with the largest possible element and thus inherits the element's shape as a volume in space. This points directly to a practical limitation: how do we represent domain geometries that have more complex shapes? One answer is to consider multiple tree roots and arrange them in the fashion of unstructured AMR, giving rise to a forest of elements [4, 49, 50]. The mesh of trees is sometimes called the coarse mesh, which is created a priori to map the topology and geometry of the domain with sufficient fidelity. Elements may then be refined and coarsened recursively, changing the mesh below the root of each tree.

Many simulations require only one tree, and the concept of the forest is not called upon. Popular forest AMR codes respect this fact by operating with no or minimal overhead in the special case of a one-tree forest. When multiple trees are needed, their number is limited by the available memory of contemporary computers to roughly one million per process [15]. While this number allows us to execute most coarse meshing tasks in serial, we may still question how to work with coarse meshes of say one or more billion trees total; such numbers are common in industrial and medical unstructured meshing [21, 26, 40]. In this case, the coarse mesh needs to be partitioned between the processes either by means of file I/O [8] or in-core, as we will discuss in this paper.

Most tree-based AMR codes make use of a space-filling curve (SFC) to order the elements (as well as points or other primitives) within a tree [24, 39, 51, 52]. Two main approaches for partitioning a forest of elements have been discussed [56], namely (a) assigning each tree and thus all of its elements to one owner process [9, 28, 43], or (b) allowing a tree to contain elements belonging to multiple processes [4, 15]. The first approach offers a simpler logic but may not provide acceptable load balance when the number of elements differs vastly between trees. The second allows for perfect partitioning of elements by number (the local numbers of elements between processes differ by at most one) but presents the issue of trees that are shared between multiple processes.

We choose paradigm (b) for speed and scalability, challenging ourselves to solve an n-to-m communication problem for every coarse mesh element. Thus the objective of this paper is to develop how to do this without handshaking (i.e., without having to determine separately which process receives from which) and with a minimal number of senders, receivers, and messages. One cornerstone is to avoid identifying a single owner process for each tree and instead treat all its sharer processes as algorithmically active, under the premise that they produce a disjoint union of the information necessary to be transferred. In particular, each process shall store the relevant tree meta data to be readily available, eliminating the need to transfer this data from a single owner process.

In a typical application, the forest mesh is refined and coarsened repeatedly,

sometimes once per time step or even more frequently [33]. The algorithms we propose have the character of subroutines called from a user's application. The criteria for mesh adaptation may be computed by the application in arbitrary ways, which is often done by evaluating some kind of error indicator; see, e.g., [1]. Refinement, coarsening, and smoothing/grading the refinement rely on these criteria. The repartitioning of elements between processes follows the updated per-process element counts, optionally taking into account user-defined weights. This is the starting point for the partitioning of the coarse mesh described here, ensuring that each process that owns an element in the new partition will have access to the tree meta data for every local element.

In this paper, we also integrate the parallel transfer of ghost trees. The reason for this is that each process will eventually collect ghost elements, i.e., remote elements adjacent to its own. Although we do not discuss such an algorithm here, we note that ghost elements of any process may be part of trees that are not in its local set. To disconnect the ghost element transfer from identifying and transferring ghost trees, we perform the latter as part of the coarse mesh partitioning, presently across tree faces. We study in detail what information we must maintain to reference neighbor trees of ghost trees (that may themselves be either local, ghost, or neither) and propose an algorithm with minimal communication effort.

We have implemented the coarse mesh partitioning for triangles and tetrahedra using the SFC designed in [14], and for quadrilaterals and hexahedra exploiting the logic from [15]. To demonstrate that our algorithms are safe to use, we verify that (a) small numbers of trees require run times on the order of milliseconds and thus present no noticeable overhead compared to a serial coarse mesh, and (b) the coarse mesh partitioning adds only a fraction of run time compared to the partitioning of the forest elements, even for extraordinarily large numbers of trees. We show a practical example of 3D dynamic AMR on 8e3 processes using 383e6 trees and up to 25e9 elements. To investigate the ultimate limit of our algorithms, we partition coarse meshes of up to 371e9 trees on a Blue Gene/Q system using 917e3 processes, obtaining a total run time of about 1.2s and a rate of 340e3 trees per second per process. On 131e3 processes we obtain rates as high as 750e3 trees per second per process.

We may summarize our results by saying that partitioning the trees can be made even less costly than partitioning the elements and often executes so fast that it does not make a difference at all. This allows a forest code that partitions both trees and elements dynamically to treat the whole continuum of forest mesh scenarios, from one tree with nearly trillions of elements on the one extreme to billions of trees that are not refined at all on the other, with comparable efficiency.

2. Tree-based AMR. We understand the connectivity of a forest as a mesh of root elements, which is effectively the coarsest possible mesh. We will simply call it *coarse mesh* and call the root elements *trees* in the following. Each tree may be subdivided recursively into elements, replacing one element by four (triangles and quadrilaterals) or eight (tetrahedra and hexahedra). For our purposes, the subdivision may be chosen freely by an application. Thus, in our context an element has two roles: one geometric as a volume in space and one logical as a node of a tree. The leaf elements of the forest compose the *forest mesh* used for computation; see also Figures 1 and 2 and Table 1. The (leaf) elements may be used in a classical finite element/finite volume/discontinuous Galerkin (dG) setting or as a meta structure, for example to manage a subgrid in each leaf element [11, 20, 31, 42].

We use SFCs to order the elements within each tree. SFCs map the d-dimensional elements of a refinement tree to an interval by assigning a unique integer index $m(E)$

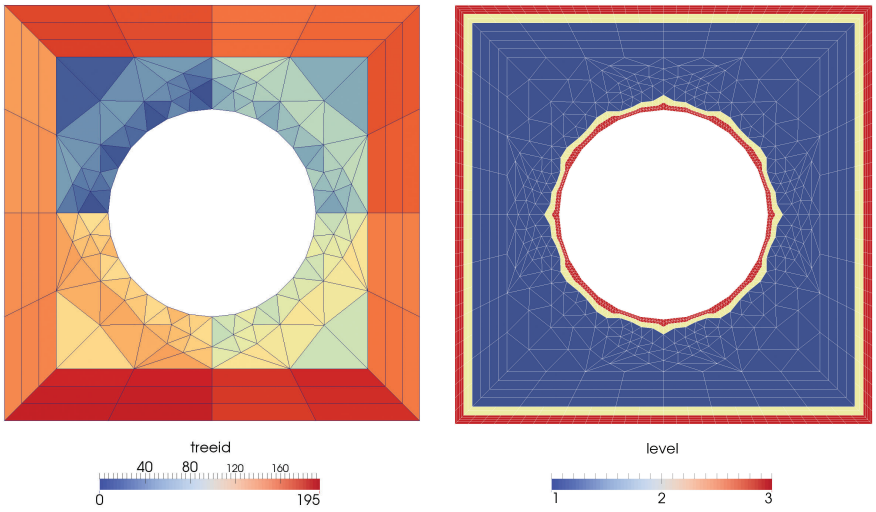


FIG. 1. *A mesh consists of two structures, the coarse mesh (left) that represents the topology of the domain, and the forest mesh (right) that consists of the leaf elements of a refinement and is used for computation. In this example the domain is a unit square with a circular hole. The color coding in the coarse mesh displays each tree's unique and consecutive identifier, while the color coding in the forest mesh represents the refinement level of each element. In this example we choose an initial global level* 1 *refinement and a refinement of up to level* 3 *along the domain boundary.*

to each element E. Thus, we can order all elements of that refinement tree linearly in an array. As in [51], we do not store the internal (nonleaf) nodes of the tree.

The choice of SFC affects the ordering of these elements of the forest mesh and thus the parallel partition of elements. Possibilities include, but are not limited to, the Hilbert, Peano, or Morton curves for quadrilaterals and hexahedra [25, 35, 37, 53], as well as the Sierpiński curve for triangles [3, 46] and the Morton curve extensions for triangles and tetrahedra [14].

A global order of elements is established first by tree and then by their index with respect to an SFC (see also [2]): We enumerate the K trees of the coarse mesh by $0, \ldots, K-1$ and call the number k of a tree its *global index*. With the global index we naturally extend the SFC order of the leaves: Let a leaf element of the tree k have SFC index I (within that tree); then we define the combined index (k, I). This index compares to a second index (k', J) as

$$(k, I) < (k', J) \quad :\Leftrightarrow \quad k < k' \text{ or } (k = k' \text{ and } I < J). \tag{1}$$

In practice we store the mesh elements local to a process in one contiguous array per locally nonempty tree in precisely this order.

The algorithms and techniques discussed in this paper assume an SFC induced order among the elements, but they are not affected by the particular choice of SFC. In the `t8code` software used for the demonstrations in this paper, we have so far implemented Morton SFCs for quadrilaterals and hexahedra via the `p4est` library [10] and the tetrahedral Morton SFC for tetrahedra and triangles. These curves compute the index $m(E)$ of an element via bitwise interleaving the coordinates of its lower left vertex in a suitable reference tree; see also Figure 2 for an illustration of the curve on triangles. Other SFC schemes may be added to the `t8code` in a modular fashion.

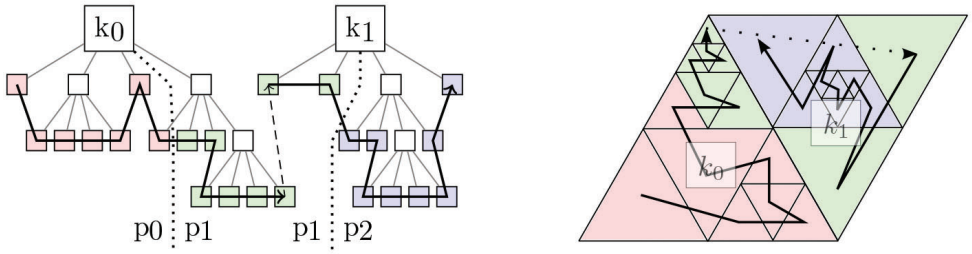


FIG. 2. *We connect multiple trees to model complex geometries. Here, we show two trees k_0 and k_1 with an adaptive refinement. To enumerate the forest mesh, we establish an a priori order between the two trees and use an SFC within each tree. On the left-hand side of the figure the refinement tree and its linear storage are shown. When we partition the forest mesh to P processes (here, $P = 3$), we cut the SFC in P equally sized parts and assign part i to process i.*

TABLE 1
The basic definitions for the coarse mesh and the forest mesh and their elements. Throughout, we refer to the neighbor information of the trees as connectivity.

Coarse mesh	Conforming mesh of tree roots
Tree	An element of the coarse mesh
Forest mesh	The adaptive mesh of elements (leaves of the trees)
Element/leaf	Each element of the forest mesh is the leaf of a tree

2.1. The tree shapes. The trees of the coarse mesh can be of arbitrary shape as long as they are all of the same dimension and fit together along their faces. In particular, we identify the following tree shapes:

- Points in 0D.
- Lines in 1D.
- Quadrilaterals and triangles in 2D.
- Hexahedra and tetrahedra in 3D.
- Prisms and pyramids in 3D.

Coarse meshes consisting solely of prisms or pyramids are quite uncommon; these tree shapes are used primarily to transition between hexahedra and tetrahedra in hybrid meshes.

2.2. Encoding of face-neighbors. The connectivity information of a coarse mesh includes the neighbor relation between adjacent trees. Two trees are considered neighbors if they share at least one lower dimensional face (vertex, face, or edge). Since all of this connectivity information can be inferred from codimension-1 neighbors, we restrict ourselves to those, denoting them uniformly by face-neighbors. This choice does not lessen the generality of the partitioning algorithms to follow and avoids a significant jump in complexity of the element-neighbor code.

An application often requires a quick mechanism to access the face-neighbors of a given forest mesh element. If this neighbor element is a member of the same tree, the computation can be carried out via the SFC logic, which involves only a few bitwise operations for the hexahedral and tetrahedral Morton curves [14, 15, 35, 51]. If, however, the neighbor element belongs to a different tree, we need to identify this tree, given the parent tree of the original element and the tree face at which we look for the neighbor element. It is thus advantageous to store the face-neighbors of each tree in an array that is ordered by the tree's faces. To this end, we fix the enumeration of faces and vertices relative to each other as depicted in Figure 3.

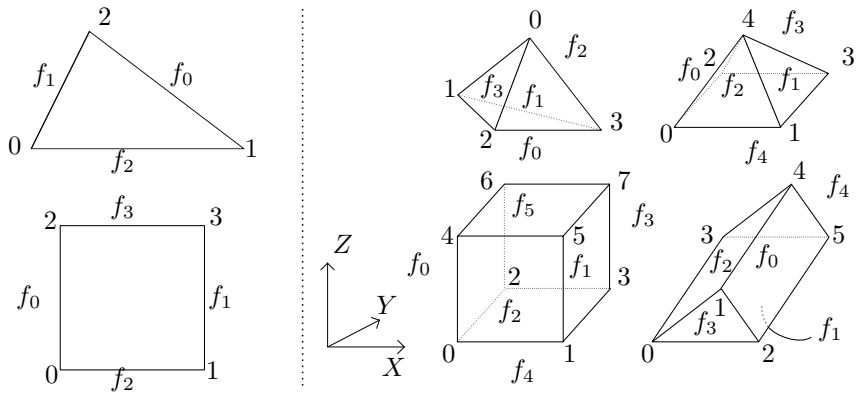


FIG. 3. *The vertex and face labels of the* 2*D (left) and* 3*D (right) tree shapes.*

2.3. Orientation between neighbors. In addition to the global index of the neighbor tree across a face, we describe how the faces of the tree and its neighbor are rotated relative to each other. We allow all connectivities that can be embedded in a compact 2- or 3-manifold in such a way that each tree has positive volume. This includes the Moebius strip and Klein's bottle and other quite exotic meshes, e.g., a hexahedron whose one face connects to another in some rotation. We obtain two possible orientations of a line-to-line connection, three for a triangle-to-triangle connection, and four for a quadrilateral-to-quadrilateral connection.

We would like to encode the orientation of a face connection analogously to the way it is handled in `p4est`: At first, given a face f, its vertices are a subset of the vertices of the whole tree. If we order them accordingly and renumerate them consecutively starting from zero, we obtain a new number for each vertex that depends on the face f. We call it the *face corner number*. If now two faces f and f' meet, the corner 0 of the face with the smaller face number is identified with a face corner k in the other face. In `p4est` this k is defined to be the orientation of the face connection.

In order for this operation to be well-defined, it must not depend on the choice of the first face when the two face numbers are the same, which is easily verified for a single tree shape. When two trees of different shapes meet, we generalize to determine which face is the first one.

DEFINITION 2.1. *We impose a semiorder on the* 3*-dimensional tree shapes as follows:*

$$\begin{matrix} Hexahedron \\ \\ Tetrahedron \end{matrix} \begin{matrix} \diagdown \\ < \\ \diagup \end{matrix} Prism < Pyramid. \tag{2}$$

This comparison is sufficient since a hexahedron and a tetrahedron can never share a common face. We use it as follows.

DEFINITION 2.2. *Let* t *and* t' *denote the tree shapes of two trees that meet at a common face with respective face numbers* f *and* f'*. Furthermore, let* ξ *be the face corner number of* f' *matching corner* 0 *of* f*, and let* ξ' *be the face corner number of*

f matching corner 0 *of* f'. *We define the orientation of this face connection as*

$$\texttt{or} := \begin{cases} \xi & \text{if } t < t' \text{ or } (t = t' \text{ and } f \le f'), \\ \xi' & \text{otherwise.} \end{cases} \tag{3}$$

We now encode the face connection in the expression $\texttt{or} \cdot F + f'$ from the perspective of the first tree and $\texttt{or} \cdot F + f$ from the second, where F is the maximal number of faces over all tree shapes of this dimension.

3. Partitioning the coarse mesh. As outlined above, tree-based AMR methods partition mesh elements with the help of an SFC. By cutting the SFC into as many equally sized parts as processes and assigning part i to process i, the repartitioning process is distributed and runs in linear time. Weighted partitions with a user-defined weight per leaf element are also possible and practical [15, 38].

If the physical domain has a complex shape such that many trees are required to optimally represent it, it becomes necessary to also partition the coarse mesh in order to reduce the memory footprint. This is even more important if the coarse mesh does not fit into the memory of one process, since such problems are not even computable without coarse mesh partitioning.

Suppose the forest mesh is partitioned among the processes. Since the forest mesh frequently references connectivity information from the coarse mesh, a process that owns a leaf e of a tree k also needs the connectivity information of the tree k to its neighbor trees. Thus, we maintain information on these so-called ghost neighbor trees.

There are two traditional approaches to partition the coarse mesh. In the first approach [9, 54], the coarse mesh is partitioned, and the owner process of a tree will own all elements of that tree. In other words, it is not possible for two different processes to own elements of the same tree, which can lead to highly imbalanced forest meshes. Furthermore, if there are fewer trees than processes, there will be idle processes assigned zero elements. In particular, this approach prohibits the frequent special case of a single tree.

The second approach [56] is to first partition the forest mesh and then deduce the coarse mesh partition from that of the forest. If several processes have leaf elements from the same tree, then the tree is assigned to one of these processes, and whenever one of the other processes requests information about this tree, communication is invoked. This technique has the advantage that the forest mesh is load-balanced much better, but it introduces additional synchronization points in the program and can lead to critical bottlenecks if a lot of processes request information on the same tree.

We propose another approach, which is a variation of the second, that overcomes the communication issue. If several processes have leaf elements from the same tree, we duplicate this tree's connectivity data and store a local copy of it on each of the processes. Thus, there is no further need for communication, and each process has exactly the information it requires. Since the purpose of the coarse mesh is not to store data that changes during the simulation but to store connectivity data about the physical domain, the data on each tree is persistent and does not change during the simulation. Certainly, this concept poses an additional challenge in the (re)partitioning process, because we need to manage multiple copies of trees without producing redundant messages.

As an example, consider the situation in Figure 4. Here the 2D coarse mesh consists of two triangles 0 and 1, and the forest mesh is a uniform level 1 mesh

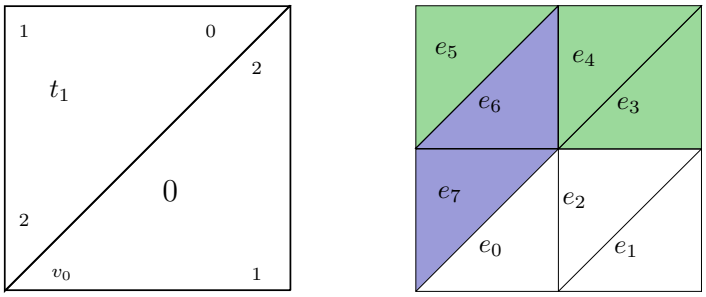


FIG. 4. *A coarse mesh of two trees and a uniform level* 1 *forest mesh. If the forest mesh is partitioned to three processes, each tree of the coarse mesh is requested by two processes. The numbers in the tree corners denote the position and orientation of the tree vertices, and the global tree ids are the numbers in the center of each tree. As an example SFC we take the triangular Morton curve from* [14]*, but the situation of multiple trees per process can occur for any SFC.*

consisting of 8 elements. Elements $0, 1, 2, 3$ belong to tree 0 and elements $4, 5, 6, 7$ to tree 1. If we load-balance the forest mesh to three processes with ranks 0, 1, and 2, then a possible forest mesh partition arising from an SFC could be

(4)

rank	elements
0	0, 1, 2
1	3, 4, 5
2	6, 7,

leading to the coarse mesh partition (5)

rank	trees
0	0
1	0, 1
2	1.

Thus, each tree is stored on two processes.

3.1. Valid partitions. We allow arbitrary partitions for the forest, as long as they are induced by an SFC. This gives us some information on the type of coarse mesh partitions that we can expect.

DEFINITION 3.1. *In general, a* partition *of a coarse mesh of* K *trees* $\{0, \dots, K-1\}$ *to* P *processes* $\{0, \dots, P-1\}$ *is a map* f *that assigns each process a certain subset of the trees,*

$$f\colon \{0, \dots, P-1\} \longrightarrow \mathcal{P}\{0, \dots, K-1\}, \tag{6}$$

and whose image covers the whole mesh:

$$\bigcup_{p=0}^{P-1} f(p) = \{0, \dots, K-1\}. \tag{7}$$

Here, $\mathcal{P}$ *denotes the set of all subsets (power set). We call* $f(p)$ *the* local trees *of process* p *and explicitly allow that* $f(p) \cap f(q)$ *may be nonempty. If so, the trees in this intersection are* shared *between processes* p *and* q*.*

The above definition includes all possibilities of attaching trees to processes. In particular, we allow the same tree to reside on multiple processes. For example, it is a partition in the above sense if all trees are partitioned to all processes, expressed by $f(p) = \{0, \dots, K-1\}$ for all p. Any other arbitrary mapping is possible, with no restriction on the number of trees per process, where empty processes with $f(p) = \emptyset$ are included.

For our method, we will not consider every possible partition of a coarse mesh. Since we assume that a forest mesh partition comes from an SFC, we restrict ourselves to a subset of partitions. In particular, the SFC order of a forest imposes the restriction that the order of trees is linear among the processes. Thus if tree k is on process p, then every tree $l > k$ must be partitioned to a process $q \geq p$.

DEFINITION 3.2. *Consider a partition f of a coarse mesh with K trees. We say that f is a* valid partition *if there exist a forest mesh with N leaves and a (possibly weighted) SFC partition of it that induces f. Thus, for each process p and each tree k, we have $k \in f(p)$ if and only if there exists a leaf e of the tree k in the forest mesh that is partitioned to process p. Processes without any trees are possible; in this case $f(p) = \emptyset$.*

We denote by k_p the local tree on p with the lowest global index and denote by K_p the local tree with the highest global index.

This, for example, excludes meshes where the processes are not mapped to the trees in ascending order. A simple example of a partition of two trees on two processes that is not valid is given by $f(0) = \{ 1 \}$ and $f(1) = \{ 0 \}$.

The definition of valid partitions requires a forest mesh and a specific SFC-induced partition of it. Since this is not convenient for theoretical investigations, we deduce three properties that characterize valid partitions independently of a forest mesh and SFCs.

PROPOSITION 3.3. *A partition f of a coarse mesh is valid if and only if it fulfills the following properties.*

(i) *The tree indices of a process's local trees are consecutive; thus*

$$f(p) = \{ k_p, k_p + 1, \ldots, K_p \} \text{ or } f(p) = \emptyset. \tag{8}$$

(ii) *A tree index of a process p may be smaller than a tree index on a process q only if $p \leq q$:*

$$p \leq q \Rightarrow K_p \leq k_q \quad (\text{if } f(p) \neq \emptyset \neq f(q)). \tag{9}$$

(iii) *The only trees that can be shared by process p with other processes are k_p and K_p:*

$$f(p) \cap f(q) \subseteq \{ k_p, K_p \} \text{ for } p \neq q. \tag{10}$$

Proof. We show the only-if direction first. Let an arbitrary forest mesh with SFC partition be given such that f is induced by it. In the SFC order the leaves are sorted according to their SFC indices. If (i, I) denotes the leaf corresponding to the Ith leaf in the ith tree and tree i has N_i leaves, then the complete forest mesh consists of the leaves

$$\{ (0,0), (0,1), (0, N_0 - 1), (1,0), \ldots, (K-1, N_{K-1} - 1) \}. \tag{11}$$

The partition of the forest mesh is such that each process p gets a consecutive range

$$\{ (k_p, i_p), \ldots, (K_p, i'_p) \} \tag{12}$$

of leaves, where (k_{p+1}, i_{p+1}) is the successor of (K_p, i'_p) and the k_p and the K_p form increasing sequences with $K_p \leq k_{p+1}$. The coarse mesh partition is then given by

$$f(p) = \{ k_p, k_p + 1, \ldots, K_p \} \text{ for all } p, \tag{13}$$

which shows properties (i) and (ii). To show (iii) we assume that $f(p)$ has at least three elements; thus $f(p) = \{ k_p, k_p + 1, \dots, K_p \}$. However, this means that in the forest mesh partition each leaf of the trees $\{ k_p + 1, \dots, K_p - 1 \}$ is partitioned to p. Since the forest mesh partitions are disjoint, no other process can hold leaf elements from these trees, and thus they cannot be shared.

To show the if-direction, suppose the partition f fulfills (i), (ii), and (iii). We construct a forest mesh with a weighted SFC partition as follows. Each tree that is local to a single process is not refined and thus contributes a single leaf element to the forest mesh. If a tree is shared by m processes, then we refine it uniformly until we have more than m elements. It is now straightforward to choose the weights of the elements such that the corresponding SFC partition induces f. ∎

We directly conclude the following.

COROLLARY 3.4. *In a valid partition, each pair of processes can share at most one tree; thus*

$$|f(p) \cap f(q)| \leq 1 \tag{14}$$

for each $p \neq q$.

Proof. Supposing the contrary, with (10) we know that there would exist two processes p and q with $p < q$ such that $f(p) \cap f(q) = \{ k_p, K_p \} = \{ k_q, K_q \}$ and $k_p \neq K_p$. Thus $K_p > k_p = k_q$, which contradicts property (9). ∎

COROLLARY 3.5. *If in a valid partition* f *of a coarse mesh the tree* k *is shared between processes* p *and* q*, then for each* $p < r < q$,

$$f(r) = \{ k \} \quad \text{or} \quad f(r) = \emptyset. \tag{15}$$

Proof. We can directly deduce this from (8), (9), and Corollary 3.4. ∎

In order to properly deal with empty processes in our calculations, we define start and end tree indices for these as well.

DEFINITION 3.6. *Let* p *be an empty process in a valid partition* f*; thus* $f(p) = \emptyset$. *Furthermore, let* $q < p$ *be maximal such that* $f(q) \neq \emptyset$. *Then we define the start and end indices of* p *as*

$$k_p := K_q + 1, \tag{16a}$$

$$K_p := K_q = k_p - 1. \tag{16b}$$

If no such q *exists, then no rank lower than* p *has local trees, and we set* $k_p = 0$, $K_p = -1$. *With these definitions,* (8) *and* (9) *are valid if any of the processes are empty.*

From now on, all partitions in this paper are assumed valid even if not stated explicitly.

3.2. Encoding a valid partition. A typical way to define a partition in a tree-based code is to store an array `O` of tree offsets for each process, that is, the global index of the first tree local to each process. The range of local trees for process p can then be computed as $\{ \mathtt{O}[p], \dots, \mathtt{O}[p+1] - 1 \}$. However, for valid partitions in the coarse mesh setting, this information would not be sufficient because we would not know which trees are shared. We thus modify the offset array by adding a negative sign when the first tree of a process is shared.

DEFINITION 3.7. *Let f be a valid partition of a coarse mesh, with k_p being the index of p's first local tree. Then we store this partition in an array $\mathtt{O}$ of length $P+1$, where for $0 \le p < P$,*

$$\mathtt{O}[p] = \begin{cases} k_p & \text{if } k_p \text{ is not shared with the next smaller} \\ & \text{nonempty process or } f(p) = \emptyset, \\ -k_p - 1 & \text{if it is.} \end{cases} \tag{17}$$

Furthermore, $\mathtt{O}[P]$ shall store the total number of trees.

Because of the definition of k_p, we know that $\mathtt{O}[0] = 0$ for all valid partitions.

LEMMA 3.8. *Let f be a valid partition, and let $\mathtt{O}$ be as in Definition 3.7. Then*

$$k_p = \begin{cases} \mathtt{O}[p] & \text{if } \mathtt{O}[p] \ge 0, \\ |\mathtt{O}[p] + 1| & \text{if } \mathtt{O}[p] < 0, \end{cases} \tag{18}$$

and

$$K_p = |\mathtt{O}[p+1]| - 1. \tag{19}$$

Proof. The first statement follows since (17) and (18) are inverses of each other. For (19) we distinguish two cases. First, let $f(p)$ be nonempty. If the last tree of p is not shared with $p+1$, then it is $k_{p+1} - 1$ and $\mathtt{O}[p+1] = k_{p+1}$, and thus we have

$$K_p = k_{p+1} - 1 = |\mathtt{O}[p+1]| - 1. \tag{20}$$

If the last tree of p is shared with $p+1$, then it is k_{p+1}, the first local tree of $p+1$, and thus $\mathtt{O}[p+1] = -k_{p+1} - 1$ and

$$K_p = k_{p+1} = |-k_{p+1}| = |\mathtt{O}[p+1]| - 1. \tag{21}$$

Now let $f(p) = \emptyset$. If k_{p+1} is not shared, then $k_{p+1} = k_p = K_p + 1$ by Definition 3.6, and $\mathtt{O}[p+1] = k_{p+1}$ by (17). Thus,

$$K_p = k_p - 1 = k_{p+1} - 1 = |\mathtt{O}[p+1]| - 1. \tag{22}$$

If k_{p+1} is shared, then again by Definition 3.6, $k_{p+1} = k_p - 1 = K_p$ and $\mathtt{O}[p+1] = -k_{p+1} - 1$ such that we obtain

$$K_p = k_{p+1} = k_{p+1} + 1 - 1 = |\mathtt{O}[p+1]| - 1. \tag{23}$$

∎

COROLLARY 3.9. *In the setting of Lemma 3.8 the number n_p of local trees of process p fulfills*

$$n_p = |\mathtt{O}[p+1]| - k_p = \begin{cases} |\mathtt{O}[p+1]| - \mathtt{O}[p] & \text{if } \mathtt{O}[p] \ge 0, \\ |\mathtt{O}[p+1]| - |\mathtt{O}[p] + 1| & \text{else.} \end{cases} \tag{24}$$

Proof. This follows from the identity $n_p = K_p - k_p + 1$. ∎

Lemma 3.8 and Corollary 3.9 show that for valid partitions, the array $\mathtt{O}$ carries the same information as the partition f.

3.3. Ghost trees. A valid partition provides information on the local trees of a process. These trees are all trees of which a forest has local elements. In many applications it is common to collect a layer of ghost (or halo) elements of the forest to support the exchange of data with neighboring processes. Since these ghost elements may be descendants of nonlocal trees, we store their trees as ghost trees. We want to confine this logic to the coarse mesh to be independent of a forest mesh, and thus we propose to store each nonlocal face-neighbor tree as a ghost tree. This means possibly storing more ghost trees than needed by a particular forest. However, this only affects the first and the last local tree of a process, which bounds the overhead. Since we restrict the neighbor information to face-neighbors, we also restrict ourselves to face-neighbor ghosts in this paper. However, an extension to edge and vertex neighbor ghosts is planned for the future. These will prompt a somewhat more elaborate discussion, since an arbitrary number of trees can be neighbored across a vertex/edge. There exist known algorithms for quadrilaterals and hexahedra [27], which we believe can be modified to extend to simplices, prisms, and pyramids.

DEFINITION 3.10. *Let f be a valid partition of a coarse mesh. A* ghost tree *of a process p is any tree k such that*

- *$k \notin f(p)$, and*
- *there exists a face-neighbor k' of k such that $k' \in f(p)$.*

If a coarse mesh is partitioned according to f, then each process p will store its local trees and its ghost trees.

3.4. Computing the communication pattern. Suppose a coarse mesh is partitioned among the processes $\{0, \ldots, P-1\}$ according to a partition f. The input of the partition algorithm is this coarse mesh and a second partition f', and the output is a coarse mesh that is partitioned according to the second partition.

Apart from these partitions being valid, no other restrictions are imposed on the partitions f and f'. Thus, we include the trivial case $f = f'$ as well as extreme cases. An example for such a case is an f that concentrates all trees on one process and an f' that assigns them to another (or distributes them evenly among all processes). For dynamic forest repartitioning it is not unusual for almost all of the elements to change their owner process [13, 34]. We expect similar behavior for the trees, especially when the number of trees is on the order of the number of processes or higher.

We suppose that in addition to its local trees and ghost trees, each process knows the complete partition tables f and f', for example, in the form of offset arrays. The task is now for each process to identify the processes to which it needs to send local and ghost trees and then to execute the sending. A process also needs to identify the processes from which it receives local and ghost trees and to execute the receiving. We discuss here how each process can compute this information from the offset arrays without further communication.

It will become clear in section 3.5 that ghost trees need not yet be discussed at this point. Thus, it is sufficient to concentrate on the local trees for the time being.

3.4.1. Ownership during partition. The fact that trees can be shared between multiple processes poses a challenge when repartitioning a coarse mesh. Suppose we have a process p and a tree k with $k \in f'(p)$, and k is a local tree for more than one process in the partition f. We do not want to send the tree multiple times, so how do we decide which process sends k to p?

A simple solution would be that the process with the smallest index to which k is a local tree sends k. This process is unique and can be determined without

communication. However, suppose that the two processes p and $p-1$ share the tree k in the old partition, and p will also have this tree in the new partition. Then $p-1$ would send the tree to p even though this message would not be needed.

We resolve this issue by only sending a local tree to a process p if this tree is not already local on p.

PARADIGM 3.11. *When repartitioning with $k \in f'(p)$, the process that sends k to p is*

- *p if k already is a local tree of p, or else*
- *q with q minimal such that $k \in f(q)$.*

We acknowledge that sending from p to p in the first case is just a local data movement not involving communication.

DEFINITION 3.12. *When repartitioning, given a process p we define the sets S_p and R_p of processes to and from which p sends and receives local trees, respectively, and thus*

$$S_p := \{\, 0 \le p' < P \,|\, p \text{ sends local trees to } p' \,\}, \tag{25a}$$

$$R_p := \{\, 0 \le p' < P \,|\, p \text{ receives local trees from } p' \,\}. \tag{25b}$$

Both sets may include the process p itself. Furthermore, we establish the notation for the smallest and largest ranks in these sets, understanding that they depend on p:

$$s_{\text{first}} := \min S_p, \quad s_{\text{last}} := \max S_p, \tag{26a}$$

$$r_{\text{first}} := \min R_p, \quad r_{\text{last}} := \max R_p. \tag{26b}$$

If S_p is empty, we set $s_{\text{first}} = -1$ and $s_{\text{last}} = -2$, and likewise for R_p.

S_p and R_p are uniquely determined by Paradigm 3.11.

3.4.2. An example. We discuss a small example; see Figure 5. Here, we repartition a partitioned coarse mesh of five trees among three processes. The initial partition f is given by

$$\texttt{O} = \{\, 0, -2, 3, 5 \,\} \tag{27}$$

and the new partition f' by

$$\texttt{O'} = \{\, 0, -3, -4, 5 \,\}. \tag{28}$$

Thus, initially tree 1 is shared by processes 0 and 1, while in the new partition, tree 2 is shared by processes 0 and 1, and tree 3 is shared by processes 2 and 3. We arrange the local trees that each process will send to every other process in a table, where the set in row i, column j is the set of local trees that process i sends to process j:

(29)

	0	1	2
0	$\{0,1\}$	$\emptyset$	$\emptyset$
1	$\{2\}$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\{3,4\}$

This leads to the following sets S_p and R_p:

$$S_0 = \{\, 0 \,\}, \qquad R_0 = \{\, 0, 1 \,\}, \tag{30a}$$

$$S_1 = \{\, 0, 1 \,\}, \qquad R_1 = \{\, 1, 2 \,\}, \tag{30b}$$

$$S_2 = \{\, 1, 2 \,\}, \qquad R_2 = \{\, 2 \,\}. \tag{30c}$$

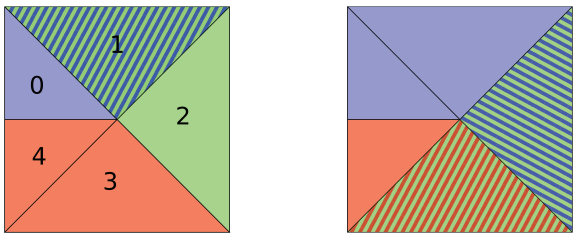


FIG. 5. *A small example. The coarse mesh consists of five trees (numbered) and is partitioned among three processes (color coded). Left: the initial partition* `O`*. Right: the new partition* `O'`*. The colors of the trees encode the processes that have a tree as local tree. Process* 0 *is drawn in blue, process* 1 *in green, and process* 2 *in red. Initially, tree* 1 *is shared among processes* 0 *and* 1*, while in the new partition, tree* 2 *is shared among processes* 0 *and* 1*, and tree* 3 *is shared among processes* 1 *and* 2*. In* (27)–(30) *we list the sets* `O` *and* `O'`*, the trees that each process sends, and the sets* S_p *and* R_p*.*

We see that process 1 keeps the tree 2 that is also needed by process 0. Thus, process 1 sends tree 2 to process 0. Process 0 also needs tree 1, which is local on process 1 in the old partition. But, since it is also local to process 0, process 1 does not send it.

3.4.3. Determining S_p and R_p. In this section we show that each process can compute the sets S_p and R_p from the offset array without further communication.

PROPOSITION 3.13. *A process p can calculate the sets S_p and R_p without further communication from the offset arrays of the old and new partitions. Once the first and last elements of each set are known, process p can determine in constant time whether any given rank is in any of those sets.*

We split the proof into two parts. First, we discuss how a process can compute s_{first}, s_{last}, r_{first}, and r_{last}, and then show how it can decide for two processes $\tilde{p}$ and q whether $q \in S_{\tilde{p}}$. In particular, we will apply this decision to each of the process numbers between the first and last elements of S_p and R_p.

We begin by determining s_{first} and s_{last} for $S_p \neq \emptyset$. For s_{first} we consider two cases. First, if the first local tree of p is not shared with a smaller rank, then s_{first} is the smallest process q that has this tree in the new partition and either is p itself or did not have it in the old one. We can find q with a binary search in the offset array.

Second, if the first tree of p is shared with a smaller rank, then p only sends it in the case that p keeps this tree in the new partition. Then $s_{\text{first}} = p$. Otherwise, we consider the second tree of p and proceed with a binary search as in the first case.

To compute s_{last}, we notice that among all ranks that have p's old last tree in the new partition and did not already have it, s_{last} is the largest (except when p itself is this largest rank, in which case it certainly had the last tree). We can determine this rank with a binary search as well. If no such process exists, we proceed with the second-to-last tree of p, for which we know that such a process must exist.

Remark 3.14. The special case $S_p = \emptyset$ occurs in the following situations:

1. p does not have any local trees.
2. p has one local tree that is shared with a smaller rank, and p does not have this tree as a local tree in the new partition.
3. p has two trees, the first of which case 2 holds for. The second (last) tree is shared with a set Q of bigger ranks, and there is no process $q \notin Q$ that has this tree as a local tree in the new partition.

These conditions can be queried before computing s_{first} and s_{last}. To check condition 3, we need to perform one binary search, while we evaluate conditions 1 and 2 in constant time.

Similarly, to compute R_p, we first look at the smallest and largest elements of this set. These are the first and last processes from which p receives trees. r_{first} is the smallest rank that had p's new first tree as a local tree in the old partition, or it is p itself if this tree was also a local tree on p. Also r_{last} is the smallest rank greater than or equal to r_{first} that had p's new last local tree as a local tree in the old partition, or p itself. We can find both of these with a binary search in the offset array of the old partition.

Remark 3.15. R_p is empty if and only if p does not have any local trees in the new partition.

LEMMA 3.16. *Given any two processes $\tilde{p}$ and q, the process p can determine in constant time whether $q \in S_{\tilde{p}}$. Moreover, p can determine for a given tree k whether $\tilde{p}$ sends k to q. In particular, this includes the cases $\tilde{p} = p$ and $q = p$.*

Proof. Let $\hat{k}_{\tilde{p}}$ be the first nonshared local tree of $\tilde{p}$ in the old partition. If such a tree does not exist, then $S_{\tilde{p}} = \emptyset$ or $S_{\tilde{p}} = \{\tilde{p}\}$. Let $\hat{K}_{\tilde{p}}$ be the last local tree of $\tilde{p}$ in the old partition if it is not the first local tree of q in the old partition, and let it be the second-to-last local tree otherwise. If such a second-to-last local tree does not exist, we conclude that $\tilde{p}$ has only one tree in the old partition, and q also has this tree in the old partition. Thus $q \notin S_{\tilde{p}}$. Furthermore, let $\hat{k}_q$ and $\hat{K}_q$ be the first and last local trees of q in the new partition. We add 1 to $\hat{k}_q$ if q sends its first local tree to itself, and this tree is also the new first local tree of q. We claim that $q \in S_{\tilde{p}}$ if and only if all of the four inequalities

$$\hat{k}_{\tilde{p}} \le \hat{K}_{\tilde{p}}, \quad \hat{k}_{\tilde{p}} \le \hat{K}_q, \quad \hat{k}_q \le \hat{K}_{\tilde{p}}, \quad \text{and} \quad \hat{k}_q \le \hat{K}_q \tag{31}$$

hold. The only-if direction follows, since if $\hat{k}_{\tilde{p}} > \hat{K}_{\tilde{p}}$, then $\tilde{p}$ does not have trees to send to q. If $\hat{k}_{\tilde{p}} > \hat{K}_q$, then the last new tree on q is smaller than the first old tree on $\tilde{p}$. If $\hat{k}_q > \hat{K}_{\tilde{p}}$, then the last tree that $\tilde{p}$ could send is smaller than the first new local tree of p. Also if $\hat{k}_q > \hat{K}_q$, then q does not receive any trees from other processes. Thus, $\tilde{p}$ cannot send trees to q if any of the four conditions is not fulfilled. The if-direction follows, since if all four conditions are fulfilled, there exists at least one tree k with

$$\hat{k}_{\tilde{p}} \le k \le \hat{K}_{\tilde{p}} \quad \text{and} \quad \hat{k}_q \le k \le K_q. \tag{32}$$

Any tree with this property is sent from $\tilde{p}$ to q. Process p can compute the four values $\hat{k}_{\tilde{p}}, \hat{K}_{\tilde{p}}, \hat{k}_q$, and $\hat{K}_q$ from the partition offsets in constant time. ∎

Remark 3.17. Let p be a process that is not empty in the new partition. For symmetry reasons, R_p contains exactly those processes $\tilde{p}$ with $r_{\text{first}} \le \tilde{p} \le r_{\text{last}}$ and $p \in S_{\tilde{p}}$.

Thus, in order to compute S_p, we can compute s_{first} and s_{last} and then check for each rank q in between whether or not the conditions of Lemma 3.16 are fulfilled with $\tilde{p} = p$. For each process this check takes only constant run time.

Now, to compute R_p we can compute r_{first} and r_{last} and then check for each rank q in between whether or not $p \in S_q$.

These considerations complete the proof of Proposition 3.13.

3.5. Face information for ghost trees. We identify the following five different types of possible face connections in a coarse mesh:

1. Local tree to local tree.
2. Local tree to ghost tree.
3. Ghost tree to local tree.
4. Ghost tree to ghost tree.
5. Ghost tree to nonlocal and nonghost tree.

There are several possible approaches to which of these face connections of a local coarse mesh we could actually store. As long as each face connection between any two neighbor trees is stored at least once globally, the information of the coarse mesh over all processes is complete, and a single process could reproduce all five types of face connection at any time, possibly using communication. Depending on which of these types we store, the pattern for sending and receiving ghost trees during repartitioning changes. Specifically, the tree that will become a ghost on the receiving process may be either a local tree or a ghost on the sending process.

When we use the maximum possible information of all five types of connections, we have the most data available and can minimize the communication required. In particular, from the nonlocal neighbors of a ghost and the partition table, a process can compute which other processes this ghost is also a ghost of and of which it is a local tree. With this information we can ensure that a ghost is sent only once and only from a process that also sends local trees to the receiving process.

The outline of the sending/receiving phase then looks like the following:

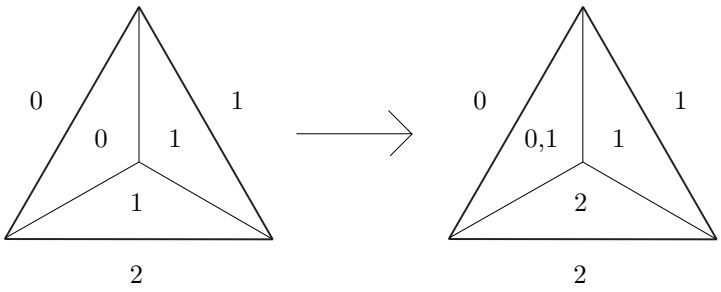
1. For each $q \in S_p$, send local trees that will be owned by q (following Paradigm 3.11).
2. Consider sending a neighbor of these trees to q if it will be a ghost on q. Send one of these neighbors if either $p = q$ or both of the following conditions are fulfilled:
   - p is the smallest rank among those that consider sending this neighbor as a ghost, and
   - $p \neq q$ and q does not consider sending this neighbor as a ghost to itself.
3. For each $q \in R_p$, receive the new local trees and ghosts from q.

In item 2 a process needs to know, given a ghost that is considered for sending to q, which other processes consider sending this ghost to q. This can be calculated without further communication from the face-neighbor information of the ghost. Since we know for each ghost the global index of each of its neighbors, we can check whether any of these neighbors is currently local on a different process $\tilde{p}$ and will be sent to q by $\tilde{p}$. If so, we know that $\tilde{p}$ considers sending this ghost to q.

Using this method, each local tree and ghost is sent only once to each receiver, and only those processes send ghosts that send local trees anyway, leading to minimal message numbers and message sizes. Storing less information would increase either the number of communicating processes or the amount of data that is communicated.

Supposing we did not store the face connection type 5, for ghost trees we would not have the information about to which nonlocal trees they connect. With this face information we could use a communication pattern such that each ghost is received only once by a process q, by sending the new ghost trees from a process that currently has it as a local tree (taking into account Paradigm 3.11). However, that process might not be an element of R_q, in which case additional processes would communicate.

If we stored only the local tree face information (types 1 and 2), then we would have minimal control over the ghost face connections. Nevertheless, we could define the partition algorithm by specifying that if a process p sends local trees to a process q, it will send all neighbors of these local trees as potential ghosts to q. The process q



	1, 2			1, 2, 3, 4			1, 2, 3, 4, 5		
p	0	1	2	0	1	2	0	1	2
0	0(1,2)	0(2)	—	0	0	(0)	0(1,2)	0	—
1	—	1(2)	2(0,1)	(1,2)	1(2)	2(1)	—	1(2)	2 (0,1)

FIG. 6. *Repartitioning example of a coarse mesh showing the communication patterns controlled by the amount of face information available. Top: A coarse mesh of three trees is repartitioned. The numbers outside of the trees are their global indices. The numbers inside of each tree denote the processes that have this tree as a local tree. At first process* 0 *has tree* 0*, process* 1 *has trees* 1 *and* 2*, and process* 2 *has no local trees. After repartitioning process* 0 *has tree* 0*, process* 1 *has trees* 0 *and* 1*, and process* 2 *has tree* 2 *as local trees. Bottom: The table shows for each usage of face connection types which processes send which data. The row of process* i *shows in column* j *which local trees* i *sends to* j*, and—in parentheses—which ghosts it sends to* j*. Using face connection types* 1–4 *we use more communication partners (process* 0 *sends to process* 2 *and process* 1 *to process* 0*) than with all five types. Using types* 1 *and* 2 *only, duplicate data is sent (process* 0 *and process* 1 *both send the ghost tree* 2 *to process* 1*).*

is then responsible for deleting those trees that it received more than once. With this method the number of communicating processes would be the same but the amount of data communicated would increase.

In Figure 6, we give an example comparing the three choices. To minimize the communication and overcome the need for postprocessing steps, we propose to store all five types of face connection.

4. Implementation. Let us begin by outlining the data structures for trees, ghosts, and the coarse mesh, and continue with a section on how to update the local tree and ghost indices. After this we present the partition algorithm to repartition a given coarse mesh according to a precalculated partition array. We emphasize that the coarse mesh data stores pure connectivity. In particular, it does *not* include the forest information, i.e., leaf elements and per-element payloads, which are managed by separate, existing algorithms.

4.1. The coarse mesh data structure. Our data structure `cmesh` that describes a (partitioned) coarse mesh has the following entries:

- `O`: An array storing the current partition table; see Definition 3.7.
- n_p: The number of local trees on this process.
- n_{ghosts}: The number of ghost trees on this process.
- `trees`: A structure storing the local trees in order of their global indices.
- `ghosts`: A structure storing the ghost trees in no particular order.

We use 64-bit integers for global counts in `O` and use 32-bit signed integers for the local tree counts in `trees` and `ghosts`. This limits the number of trees per process to

$2^{31} - 1 \cong 2 \times 10^9$. However, even with an overly optimistic memory usage of only 10 bytes per tree, storing that many trees would require about 18.6 GB of memory per process. Since on most distributed machines the memory per process is indeed much smaller, choosing 32-bit integers does not effectively limit the local number of trees. In presently unimaginable cases, we could still switch to 64-bit integers.

We call the index of a local tree inside the `trees` array the *local index* of this tree. Analogously, we call the index of a ghost in `ghosts` the *local index* of that ghost. On process p, we compute the global index k of a tree in `trees` from its local index ℓ and compute the global index k_p of the first local tree and vice versa, since

$$k = k_p + \ell. \tag{33}$$

This allows us to address local trees with their local indices using 32-bit integers.

Each `tree` in the array `trees` stores the following data:

- `eclass`: The tree's shape as a small number (triangle, quadrilateral, etc.).
- `tree_to_tree`: An array storing the local tree and ghost neighbors along this tree's faces. See section 2.2 and the text below.
- `tree_to_face`: An array encoding for each face the face-neighbor's face number and the orientation of the face connection. See section 2.3.
- `tree_data`: A pointer to additional data that we store with the tree, for example, geometry information or boundary conditions defined by an application.

The ith entry of `tree_to_tree` encodes the tree number of the face-neighbor at face i using an integer k with $0 \leq k < n_p + n_{\text{ghosts}}$. If $k < n_p$, the neighbor is the local tree with local index k. Otherwise, the neighbor is the ghost with local index $k - n_p$.

We do not allow a face to be connected to itself. Instead, we use such a connection in the face-neighbor array to indicate a domain boundary. However, a tree can be connected to itself via two different faces. This allows for one-tree periodicity, as say in a 2D torus consisting of a single quadrilateral tree.

The `tree_data` field can contain arbitrary data. An application can use these, for example, to store higher-order geometry data per tree in order to account for curved boundaries of the coarse mesh. Refined elements in the forest can then be snapped to the curved boundaries by evaluating the `tree_data` field [12]. `tree_data` is partitioned to the processes together with the trees; thus possible duplicate copies of it can exist.

Each `ghost` in the array `ghosts` stores the following data:

- `Id`: The ghost's global tree index.
- `eclass`: The shape of the ghost tree.
- `tree_to_tree`: An array giving for each face the global number of its face-neighbor.
- `tree_to_face`: As above.

Since a ghost stores the global number of all its face-neighbor trees, we can locally compute all other processes that have this tree as a ghost by combining the information from `0` and `tree_to_tree`.

4.2. Updating local indices. After partitioning, the local indices of the trees and ghosts change. The new local indices of the local trees are determined by subtracting the global index of the first local tree from the global index of each local tree. The local indices of the ghosts are given by their positions in the data array.

Since the local indices change after repartitioning, we update the `tree_to_tree` entries of the local trees to store those new values. Because a neighbor of a tree can be

either a local tree or a ghost on the previous owning process $\tilde{p}$ and become either local or a ghost on the new owning process p, there are four cases that we shall consider.

We handle these four cases in two phases, the first phase being carried out on process $\tilde{p}$ before the tree is sent to p. In this phase we change all neighbor entries of the trees that become local. The second phase executes on p after the tree has been received from $\tilde{p}$. At this point we change all neighbor entries belonging to trees that become ghosts.

In the first phase, $\tilde{p}$ has information about the first local tree on $\tilde{p}$ in the old partition, its global number being $k_{\tilde{p}}$. Via `O'` it also knows k_p^{new}, the global index of p's first tree in the new partition. Given a local tree on $\tilde{p}$ with local index $\tilde{k}$ in the old partition, we compute its new local index k on p as

$$k = k_{\tilde{p}} + \tilde{k} - k_p^{\text{new}}, \tag{34}$$

which is its global index minus the global index of the new first local tree. Given a ghost g on $\tilde{p}$ that will be a local tree on p, we compute its local tree number as

$$k = g.\text{Id} - k_p^{\text{new}}. \tag{35}$$

In the second phase, p has received all its new trees and ghosts and thus can give the new ghosts local indices to be stored in the `neighbors` fields of the trees. We do this by parsing its ghosts for each process $\tilde{p} \in R_p$ (in ascending order) and incrementing a counter. For each ghost, we parse its neighbors for local trees, and for any of these we set the appropriate value in its `neighbors` field.

Note that these four cases apply in the special case $\tilde{p} = p$ as well.

4.3. `Partition_cmesh`: Algorithm 4.1. The input is a partitioned coarse mesh C and a new partition layout `O'`, and the output is a new coarse mesh C' that carries the same information as C and is partitioned according to `O'`.

This algorithm follows the method described in section 3.5 and is separated into two main phases, the `sending phase` and the `receiving phase`. In the former we iterate over each process $q \in S_p$ and decide which local trees and ghosts we send to q. Before sending, we carry out phase one of the update of the local tree numbers. Subsequently, we receive all trees and ghosts from the processes in R_p and carry out phase two of the local index update.

In the `sending phase` we iterate over the trees that we send to q. For each of these trees we check for each neighbor (local tree and ghost) whether we send it to q as a ghost tree. This is the second item in the list of section 3.5. The function `Parse_neighbors` decides for a given local tree or ghost neighbor whether it is sent to q as a ghost.

5. Numerical results. The run time results that we present here have been obtained with version 0.2 of `t8code`[1] using the JUQUEEN supercomputer at Forschungszentrum Jülich, Germany. It is an IBM BlueGene/Q system with 28,672 nodes consisting of IBM PowerPC A2 processors at 1.6 GHZ with 16 GB RAM per node [29]. Each compute node has 16 cores and is capable of running up to 64 MPI processes using multithreading.

5.1. How to obtain example meshes. To measure the performance and memory consumption of the algorithms presented above, we would like to test the algorithms on coarse meshes that are too big to fit into the memory of a single process,

[1] https://github.com/cburstedde/t8code

Algorithm 4.1: `Partition_cmesh`(`cmesh` C, `partition O'`)

```
1  p ← this process
2  From C.O and O' determine S_p and R_p.          /* see section 3.4.3 */
   /* Sending phase                                                     */
3  for each q ∈ S_p do
4  |  G ← ∅                                  /* trees p sends as ghosts to q */
5  |  s ← first local tree to send to q.
6  |  e ← last local tree to send to q.
7  |  T ← { C.trees[s], ..., C.trees[e] }   /* local trees p sends to q */
8  |  for k ∈ T do
9  |  |  Parse_neighbors (C, k, q, G, O', s, e)
10 |  update_tree_ids_phase1 (T ∪ G)  /* see equations (34) and (35) */
11 |  Send T ∪ G to process q
   /* Receiving phase                                                   */
12 for each q ∈ R_p do
13 |  Receive T[q] ∪ G[q] from process q
14 C'.trees ← ⋃_{R_p} T[q]                    /* new array of local trees */
15 C'.ghosts ← ⋃_{R_p} G[q]                   /* new array of ghost trees */
16 update_tree_ids_phase2 (C'.ghosts)
17 C'.O ← O'
18 return C'
```

```
   /* decide which neighbors of k to send as a ghost to q               */
1  Function Parse_neighbors(cmesh C, tree k, process q, ghosts G,
     partition O', tree_indices s, e)
2  for u ∈ k.tree_to_tree \ { s, ..., e } do
3  |  if 0 ≤ u < n_p and Send_ghost(C, ghost(u), q, O') then
4  |  |  if u + k_p ∉ f'(q) then
5  |  |  |  G ← G ∪ { ghost(u) }    /* local tree u becomes ghost of q
   |  |  |    */
6  |  else                                                  /* n_p ≤ u */
7  |  |  g ← C.ghosts[u − n_p]
8  |  |  if g.Id ∉ f(q) and Send_ghost(C, g, q, O') then
9  |  |  |  G ← G ∪ { g }                          /* g is a ghost of q */
```

```
   /* Subroutine to decide whether to send a ghost or not               */
1  Function Send_ghost(cmesh C, ghost g, process q, partition O')
2  S ← ∅
3  for u ∈ g.tree_to_tree do
4  |  for q' with u is a local tree of q' do
5  |  |  if q' sends u to q then                    /* See Lemma 3.16 */
6  |  |  |  S ← S ∪ { q' }
7  if q ∉ S and p = min S then
8  |  return true     /* p is the smallest rank sending trees to q */
9  else
10 |  return false
```

which is 1 GB on JUQUEEN if we use 16 MPI ranks per node. We consider the following three approaches to construct such meshes:

1. Use an external parallel mesh generator.
2. Use a serial mesh generator on a large-memory machine, transfer the coarse mesh to the parallel machine's file system, and read it using (parallel) file I/O.
3. Create a large coarse mesh by forming the disjoint union of smaller coarse meshes over the individual processes.

Due to a lack of availability of parallel open source mesh generators, we restrict ourselves to the second and third methods. These have the advantage of being started with initial coarse meshes that fit into a single process's memory such that we can work with serial mesh generating software. In particular, we use `gmsh`, `TetGen`, and `Triangle` [22, 44, 45].

The third method is especially well suited for weak scaling studies, since the individual small coarse meshes can be created programmatically and communication-free on each process. They may be of the same size or different sizes among the processes.

We discuss two examples below. In the first we examine purely the coarse mesh partitioning without regard for a forest and its elements (using hexahedral meshes), and in the second we drive the coarse mesh partitioning by a dynamically changing forest of elements (using tetrahedral meshes). The latter example produces shared trees and thus fully executes the algorithmic ideas put forward above.

5.2. Disjoint bricks. In our first example we conduct both strong and weak scaling studies of coarse mesh partitioning and test the maximal number of (hexahedral) trees that we can support before running out of memory. For our weak scaling results, we keep the same per-process number of trees while increasing the total number of processes, which we achieve by constructing an $n_x \times n_y \times n_z$ brick of trees on each process using three constant parameters n_x, n_y, and n_z. We repartition this coarse mesh once, by the rule that each rank p sends 43% of its local trees to the rank $p + 1$ (except the biggest rank $P - 1$, which keeps all its local trees). We choose this odd percentage to create nontrivial boundaries between the regions of trees to keep and trees to send. See Figure 7 for a depiction of the partitioned coarse mesh on six processes. The local bricks are created locally as `p4est` connectivities with `p4est_connectivity_new_brick` and are then reinterpreted in parallel as a distributed coarse mesh data structure.

We perform strong and weak scaling studies on up to 917,504 MPI ranks and display our results in Figures 8 and 9 and Table 2. We show the results of one study with 16 MPI ranks per compute node, thus 1 GB available memory per process, and one with 32 MPI ranks per compute node, leaving half of the memory per process. In both cases we measure run times for different mesh sizes per process. We observe that even for the biggest meshes of 405e3 and 810e3 trees per process the absolute run times of partition are below 1.2 seconds. Furthermore, we measure a weak scaling efficiency of 97.4% for the 810e3 mesh on 262,144 processes and 86.2% for the 405e3 mesh on 458,752 processes. The biggest mesh that we created is partitioned between 917,504 processes and uses 405e3 trees per process for a total of over 371e9 trees.

We notice a drop off in the scaling behavior when the number of trees per process is about 100e3 and smaller. At this stage the time for local computation is small relative to the time for communication. Since in many cases the number of trees per process will likely be even smaller, we add Table 3. It documents running `Partition_cmesh`

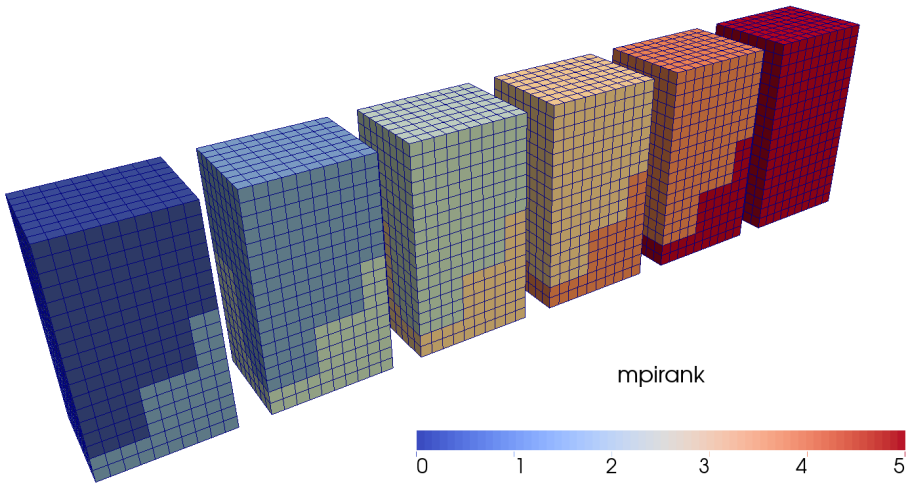


FIG. 7. *The structure of the coarse mesh that we use to measure the maximum possible mesh sizes and scalability for an example with six processes. Before partitioning, the coarse mesh local to each process is created as one $n_x \times n_y \times n_z$ block of hexahedral trees. We repartition the mesh such that each process sends 43% of its local trees to the next process. The picture shows the resulting partitioned coarse mesh with parameters $n_x = 10, n_y = 18$, and $n_z = 8$ and color coded MPI rank.*

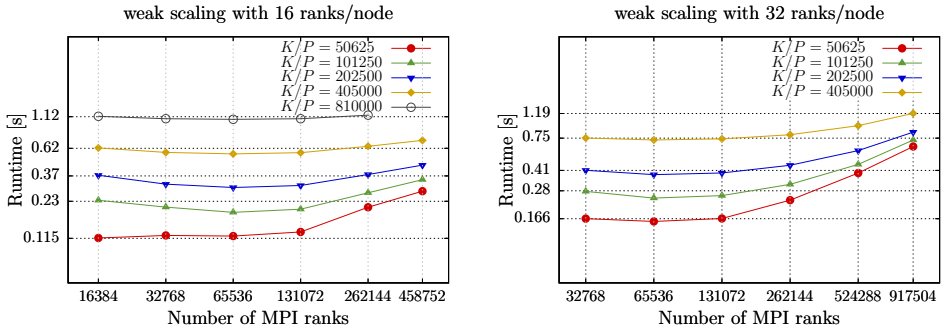


FIG. 8. *Weak scaling of* `Partition_cmesh` *with disjoint bricks. Left:* 16 *ranks per node. Right:* 32 *ranks per node. We show the run times for the baseline on the y-axis and provide graphs for different ratios between total coarse cells K and MPI processes P. On the left-hand side the time for the largest* 458,752 *process run is* 0.72 *seconds; on the right-hand side the time for the largest* 917,504 *process run is* 1.19 *seconds. We obtain efficiencies of* $0.62/0.72 = 86\%$ *and* $0.75/1.19 = 63\%$ *compared to the baselines of* 16,384/32,768 *MPI ranks, respectively (yellow lines). The time for the* $262,144$ *process run with* $810e3$ *trees per process (black line) increases from* 1.12 *to* 1.15 *seconds, which translates into a weak scaling efficiency of* 97.4%. *(See online version for color.)*

with small coarse meshes, using a number of trees on the order of the number of processes. These tests show that for such small meshes the run times are on the order of milliseconds. Hence, for small meshes there is no disadvantage in using a partitioned coarse mesh over a replicated one (i.e., each process holding a full copy).

5.3. An example with a forest. In this example we partition a tetrahedral coarse mesh according to a parallel forest of fine elements. While we pushed the maximum number of trees in the previous example, we now consider mesh sizes that occur in more common usage scenarios.

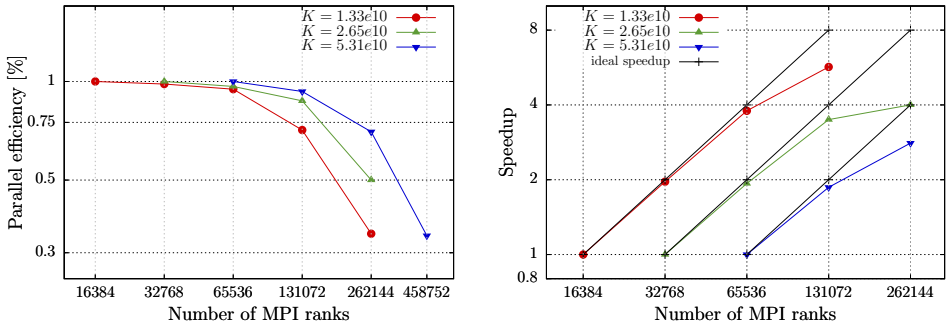


FIG. 9. *Strong scaling of* `Partition_cmesh` *for the disjoint bricks example on JUQUEEN with* 16 *ranks per compute node, for three runs with* 13.3e9, 26.5e9, *and* 53.1e9 *trees. We show the parallel efficiency on the left and the speedup on the right. The absolute run times for* 262,144 *processes are* 0.21, 0.27, *and* 0.38 *seconds.*

TABLE 2

The run times of `Partition_cmesh` *for* 131,072 *processes with* 16 *processes per node (top) and* 917,504 *processes with* 32 *processes per node (bottom). The largest coarse mesh that we created during the tests has* 371e9 *trees. In the middle column we list the average number of trees (ghosts) that each process sends to another process. The last column is the quotient of the current run time divided by the previous run time. Since we double the mesh size in each step, we expect an increase in run time of a factor of* 2, *which hints at parallel overhead becoming negligible in the limit of many trees per process.*

Run time tests for `Partition_cmesh`

131,072 MPI ranks (16 ranks per node)					
Mesh size	Per rank	Trees (ghosts) sent		Time [s]	Factor
6.635e9	50,625	21,767	(3,414)	0.13	–
13.27e9	101,250	43,536	(5,504)	0.20	1.53
26.54e9	202,500	87,074	(6,607)	0.31	1.56
53.08e9	405,000	174,149	(11,381)	0.57	1.85
106.2e9	810,000	348,297	(22,335)	1.08	1.89
917,504 MPI ranks (32 ranks per node)					
Mesh size	Per rank	Trees (ghosts) sent		Time [s]	Factor
46.45e9	50,625	21,768	(3,413)	0.64	–
92.90e9	101,250	43,537	(5,504)	0.72	1.13
185.8e9	202,500	87,075	(6,607)	0.84	1.12
371.6e9	405.000	174,150	(11,383)	1.19	1.42

TABLE 3

Run times for `Partition_cmesh` *for relatively small coarse meshes. The bottom two rows of the table was not computed on JUQUEEN but on a local institute cluster of* 78 *nodes with* 8 *Intel Xeon CPU E5-2650 v2 @ 2.60GHz each.*

#MPI ranks	#trees	Run time [s]
1,024	4,096	0.00136
1,024	8,192	0.00149
1,024	16,384	0.00142
64	105	0.00122
32	105	0.00789
64	3,200	0.000293
64	19,200	0.000865

When simulating shock waves or two-phase flows, there is often an interface along which a finer mesh resolution is desired in order to minimize computational errors. Motivated by this example, we create the forest mesh as an initial uniform refinement of the coarse mesh with a specified level ℓ and refine it in a band along an interface defined by a plane in $\mathbb{R}^3$ up to a maximum refinement level $\ell + k$. As the refinement rule we use 1:8 red refinement [7] together with the tetrahedral Morton SFC [14]. We move the interface through the domain with a constant velocity. Thus, in each time step the mesh is refined and coarsened, and therefore we repartition it to maintain an optimal load balance. We measure run times for both coarse mesh and forest mesh partitioning for three time steps.

Our coarse mesh consists of tetrahedral trees modeling a brick with spherical holes in it. To be more precise, the brick is built out of $n_x \times n_y \times n_z$ tetrahedralized unit cubes, and each of those has one spherical hole in it; see Figures 10 and 11 for a small example mesh.

We create the mesh in serial using the generator `gmsh` [22]. We read the whole file on a single process, and thus use a local machine with 1 terabyte memory for preprocessing. On this machine we partition the coarse mesh to several hundred processes and write one file for each partition. This data is then transferred to the supercomputer. The actual computation consists of reading the coarse mesh from files, creating the forest on it, and partitioning the forest and the coarse mesh simultaneously. To optimize memory while loading the coarse mesh, we open at most one partition file per compute node.

The coarse mesh that we use in these tests has parameters $n_x = 26, n_y = 24, n_z = 18$ and thus 11,232 unit cubes. Each cube is tetrahedralized with about 34,150 tetrahedra, and the whole mesh consists of 383,559,464 trees. In the first test, we create a forest of uniform level 1 and maximal refinement level 2, and in the second, we create a forest of uniform level 2 and maximal refinement level 3. The forest mesh in the first test consists of approximately 2.6e9 elements. In the second test, we also use a broader band and obtain a forest mesh of 25e9 tetrahedra.

In Table 4 we show the run time results and further statistics for coarse mesh partitioning and in Table 5 show results for forest partitioning . We observe that the run time for `Partition_cmesh` is between 0.10 and 0.13 seconds, and that about 88% or 98%, respectively, of all processes share local trees with other processes. The run times for forest partition are below 0.22 seconds for the first example and below 0.65 seconds for the second example.

We also run a third test on 458,752 MPI ranks and refine the forest to a maximum level of four. Here, the forest mesh has 167e9 tetrahedra, which we partition in under 0.6 seconds. The coarse mesh partition routine runs in about 0.2 seconds. Approximately 60% of the processes have a shared tree in the coarse mesh. In Table 6 we show the results from this test.

6. Conclusion. In this paper we propose an algorithm that executes dynamic and in-core coarse mesh partitioning. In the context of tree-based adaptive mesh refinement (AMR), the coarse mesh defines the connectivity of tree roots, which is used in all neighbor query operations between elements. This development is motivated by simulation problems on complex domains that require large input meshes. Without partitioning of the tree meta data, we will run out of memory around one million trees, and with static or out-of-core partitioning, we might not have the flexibility to transfer the tree meta data as required by the change in process ownership of the trees' elements, which occurs in every AMR cycle. With the approach presented

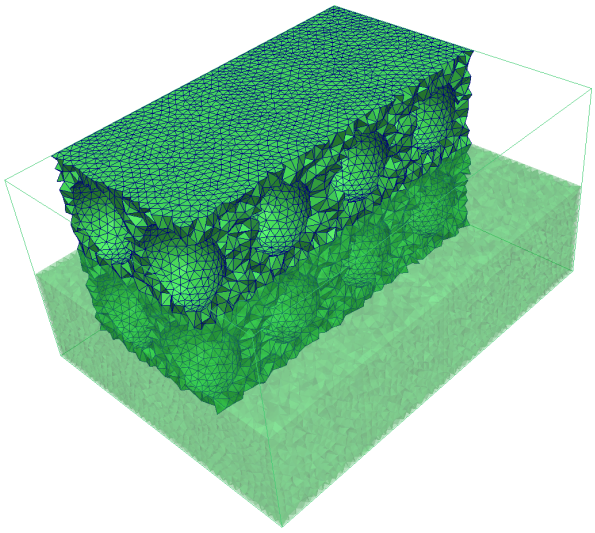


FIG. 10. *The coarse mesh connectivity that we use for the partition tests motivated by an adapted forest. It consists of $n_x \times n_y \times n_z$ cubes, with each cube having one spherical hole. For this picture we use $n_x = 4$, $n_y = 3$, $n_z = 2$, and each cube is triangulated with approximately* 7,575 *tetrahedra. For illustration purposes we show some parts of the mesh as opaque and other parts as invisible.*

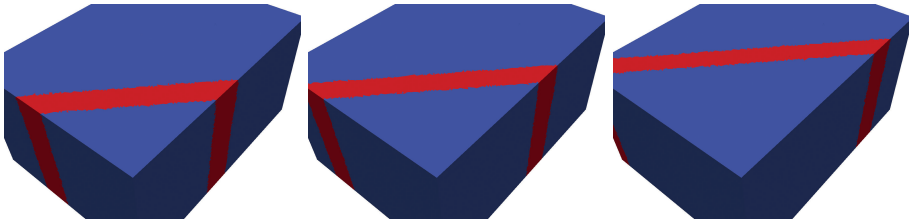


FIG. 11. *An illustration of the band of finer forest mesh elements in the example. The region of finer mesh elements moves through the mesh in each time step. From left to right we see $t = 1$, $t = 2$, and $t = 3$. In this illustration, elements of refinement level* 1 *are blue and elements of refinement level* 2 *are red.*

here, this can be performed with run times that are significantly smaller than those for partitioning the elements, even considering that SFC methods for the latter are exceptionally fast in absolute terms. Thus, we add little to the run time of all AMR operations combined.

Our algorithm guarantees that each process can provide the tree meta data for each of its fine mesh elements that are themselves distributed using a SFC. We handle the communication without handshaking and develop a communication pattern that minimizes data movement. This pattern is calculated by each process individually, reusing information that is already present.

Our implementation scales up to 917e3 MPI processes and up to 810e3 trees per process, where the largest test case consists of 371e9 trees. What remains to be done is extending the partitioning of ghost trees to edge and corner neighbors, since only face-neighbor ghost trees are presently handled. It appears that the structure of the algorithm will allow this with little modification.

TABLE 4

Coarse mesh partition on 8,192 *MPI ranks.* `Partition_cmesh` *with a coarse mesh of* 324,766,336 *tetrahedral trees on* 8,192 *MPI ranks. We measure the duration of mesh repartitioning for three time steps. For each one, we show process average values of the number of trees (ghosts) and the total number of bytes that each process sends to other processes. The average number of other processes to which a process sends ($|S_p|$) is below three in each test. We also provide the total number of shared trees in the mesh, where* 8,191 *is the maximum possible value.*

t	Trees (ghosts) sent	Data sent [MiB]	$\lvert S_p \rvert$	Shared trees	Run time [s]
1	25,117 (11,948)	4.95	2.27	7,178	0.103
2	34,860 (16,854)	6.88	2.75	7,176	0.110
3	36,386 (17,568)	7.18	2.82	7,182	0.112
1	39,026 (18,334)	7.67	2.97	8,096	0.128
2	38,990 (18,268)	7.66	2.95	8,085	0.129
3	38,942 (18,074)	7.64	2.93	8,085	0.128

TABLE 5

Forest mesh partition on 8,192 *MPI ranks. For the same example as in Table* 4 *we display statistics and run times for the forest mesh partition. We show the total number of tetrahedral elements and the average count of elements and bytes that each process sends to other processes (their count is the same as in Table* 4*).*

t	Mesh size	Elements sent	Data sent [MiB]	Run time [s]
1	2,622,283,453	203,858	3.49	0.215
2	2,623,842,241	281,254	4.82	0.215
3	2,626,216,984	293,387	5.03	0.214
1	25,155,319,545	3,013,230	46.6	0.642
2	25,285,522,233	3,008,800	46.5	0.640
3	25,426,331,342	2,991,990	46.2	0.645

TABLE 6

Coarse and forest mesh partitions on 458,752 *MPI ranks. Run times for coarse mesh and forest partition for the brick with holes on* 458,752 *MPI ranks. The setting and the coarse mesh are the same as in Table* 4 *except that for the forest we use an initial uniform level three refinement with a maximum level of four.*

t	Trees (ghosts) sent	Data sent [MiB]	$\lvert S_p \rvert$	Shared trees	Run time [s]
1	704 (2,444)	0.267	2.99	280,339	0.207
2	707 (2,456)	0.269	3.00	281,694	0.204
3	708 (2,458)	0.269	3.00	281,900	0.204

t	Mesh size	Elements sent	Data sent [MiB]	Run time [s]
1	167,625,595,829	362,863	5.55	0.522
2	167,709,936,554	364,778	5.58	0.578
3	167,841,392,949	365,322	5.59	0.567

Acknowledgments. The authors gratefully acknowledge travel support by the Bonn Hausdorff Center for Mathematics (HCM). We use the interface to the MPI3 shared array functionality written by Tobin Isaac, to be found in the files `sc_shmem.{c,h}` of the `sc` library at https://github.com/cburstedde/libsc. The authors would like to thank the Gauss Centre for Supercomputing (GCS) for providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS share of the supercomputer JUQUEEN at Jülich Supercomputing Centre (JSC). GCS is the alliance of the three national supercomputing centres HLRS (Universität Stuttgart), JSC (Forschungszentrum Jülich), and LRZ (Bayerische Akademie der Wissenschaften).

REFERENCES

[1] I. Babuška and W. Rheinboldt, *A posteriori error estimates for the finite element method*, Int. J. Numer. Methods Eng., 12 (1978), pp. 1597–1615.

[2] M. Bader, *Space-Filling Curves: An Introduction with Applications in Scientific Computing*, Texts in Comput. Sci. Eng., Springer, 2012.

[3] M. Bader and Ch. Zenger, *Efficient storage and processing of adaptive triangular grids using Sierpinski curves*, in Proc. International Conference on Computational Science, Lecture Notes in Comput. Sci. 3991, Springer, 2006, pp. 673–680, https://doi.org/10.1007/11758501_90.

[4] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.II—a general-purpose object-oriented finite element library*, ACM Trans. Math. Software, 33 (2007), 24, https://doi.org/10.1145/1268776.1268779.

[5] M. J. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, J. Comput. Phys., 82 (1989), pp. 64–84, https://doi.org/10.1016/0021-9991(89)90035-1.

[6] M. J. Berger and J. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comput. Phys., 53 (1984), pp. 484–512, https://doi.org/10.1016/0021-9991(84)90073-1.

[7] J. Bey, *Der BPX-Vorkonditionierer in drei Dimensionen: Gitterverfeinerung, Parallelisierung und Simulation*, preprint, Universität Heidelberg, 1992.

[8] G. Bryan, *Enzo* 2.5 *Documentation*, Laboratory for Computational Astrophysics, University of Illinois, http://enzo.readthedocs.io/en/latest/ (accessed Oct 30, 2016).

[9] A. Burri, A. Dedner, R. Klöfkorn, and M. Ohlberger, *An efficient implementation of an adaptive and parallel grid in DUNE*, in Computational Science and High Performance Computing II, Springer, 2006, pp. 67–82, https://doi.org/10.1007/3-540-31768-6_7.

[10] C. Burstedde, *p4est: Parallel AMR on Forests of Octrees*, http://www.p4est.org/ (accessed February 27, 2015).

[11] C. Burstedde, D. Calhoun, K. T. Mandli, and A. R. Terrel, *Forestclaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws*, in Parallel Computing: Accelerating Computational Science and Engineering (CSE), M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. Peters, eds., Advances in Parallel Computing 25, IOS Press, 2014, pp. 253–262, http://doi.org/10.3233/978-1-61499-381-0-253.

[12] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. C. Wilcox, *Extreme-scale AMR*, in SC10: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2010, pp. 1–12.

[13] C. Burstedde, O. Ghattas, G. Stadler, T. Tu, and L. C. Wilcox, *Towards adaptive mesh PDE simulations on petascale computers*, in Proc. Teragrid '08, 2008.

[14] C. Burstedde and J. Holke, *A tetrahedral space-filling curve for nonconforming adaptive meshes*, SIAM J. Sci. Comput., 38 (2016), pp. C471–C503, https://doi.org/10.1137/15M1040049.

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

[16] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, *Hypergraph-based dynamic load balancing for adaptive scientific computations*, in Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), IEEE, 2007, https://doi.org/10.1109/IPDPS.2007.370258.

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

[18] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Comput. Sci. Eng., 4 (2002), pp. 90–97.

[19] H. Digonnet, T. Coupez, and L. Silva, *A massively parallel multigrid solver using PETSc for unstructured meshes on Tier*0 *supercomputer,* Presentation at PETSc User Meeting, 2016, https://www.mcs.anl.gov/petsc/meetings/2016/slides/digonnet.pdf.

[20] J. Dreher and R. Grauer, *Racoon: A parallel mesh-adaptive framework for hyperbolic conservation laws*, Parallel Comput., 31 (2005), pp. 913–932.

[21] D. Feng, C. Tsolakis, A. N. Chernikov, and N. P. Chrisochoides, *Scalable* 3*d hybrid parallel Delaunay image-to-mesh conversion algorithm for distributed shared memory architectures*, Computer-Aided Design, 85 (2017), pp. 10–19.

[22] C. Geuzaine and J.-F. Remacle, *Gmsh: A* 3*-D finite element mesh generator with built-in pre- and post-processing facilities*, Int. J. Numer Methods Engrg., 79 (2009), pp. 1309–1331, https://doi.org/10.1002/nme.2579.

[23] M. Griebel and G. Zumbusch, *Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves*, Parallel Comput., 25 (1999), pp. 827–843.

[24] M. Griebel and G. Zumbusch, *Hash based adaptive parallel multilevel methods with space-filling curves*, in Proc. NIC Symposium 2001, H. Rollnik and D. Wolf, eds., NIC Ser. 9, Forschungszentrum Jülich, 2002, pp. 479–492.

[25] D. Hilbert, *Über die stetige Abbildung einer Linie auf ein Flächenstück*, Math. Ann., 38 (1891), pp. 459–460.

[26] D. A. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard, *Pumi: Parallel unstructured mesh infrastructure*, ACM Trans. Math. Software, 42 (2016), 17, https://doi.org/10.1145/2814935.

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

[28] Y. Ito, A. M. Shih, A. K. Erukala, B. K. Soni, A. Chernikov, N. P. Chrisochoides, and K. Nakahashi, *Parallel unstructured mesh generation by an advancing front method*, Math. Comput. Simul., 75 (2007), pp. 200–209.

[29] Jülich Supercomputing Centre, *JUQUEEN: IBM Blue Gene/Q supercomputer system at the Jülich supercomputing centre*, J. Large-Scale Res. Facilities, 1 (2015), A1, https://doi.org/10.17815/jlsrf-1-18.

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

[31] R. Maxwell, S. Kollet, S. Smith, C. Woodward, R. Falgout, I. Ferguson, N. Engdahl, L. Condon, B. Hector, S. Lopez, J. Gilbert, L. Bearup, J. Jefferson, C. Collins, I. de Graaf, C. Prubilick, C. Baldwin, W. Bosl, R. Hornung, and S. Ashby, ParFlow *User's Manual*, Report GWMI 2016-01, Integrated Ground Water Modeling Center, 2016.

[32] O. Meister, K. Rahnema, and M. Bader, *Parallel memory-efficient adaptive mesh refinement on structured triangular meshes with billions of grid cells*, ACM Trans. Math. Software, 43 (2017), 19, https://doi.org/10.1145/2947668.

[33] M. Mirzadeh, A. Guittet, C. Burstedde, and F. Gibou, *Parallel level-set methods on adaptive tree-based grids*, J. Comput. Phys., 322 (2016), pp. 345–364.

[34] W. F. Mitchell, *A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids*, J. Parallel Distrib. Comput., 67 (2007), pp. 417–429, https://doi.org/10.1016/j.jpdc.2006.11.003.

[35] G. M. Morton, *A Computer Oriented Geodetic Data Base, and a New Technique in File Sequencing*, Tech. report, IBM Ltd., 1966.

[36] C. D. Norton, G. Lyzenga, J. Parker, and R. E. Tisdale, *Developing Parallel GeoFEST(P) Using the PYRAMID AMR Library*, Tech. report, Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2004.

[37] G. Peano, *Sur une courbe, qui remplit toute une aire plane*, Math. Ann., 36 (1890), pp. 157–160.

[38] A. Pinar and C. Aykanat, *Fast optimal load balancing algorithms for* 1*D partitioning*, J. Parallel Distrib. Comput., 64 (2004), pp. 974–996, https://doi.org/10.1016/j.jpdc.2004.05.003.

[39] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, et al., *Petascale direct numerical simulation of blood flow on* 200*k cores and heterogeneous architectures*, in Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2010, pp. 1–11.

[40] M. Rasquin, C. Smith, K. Chitale, E. S. Seol, B. A. Matthews, J. L. Martin, O. Sahni, R. M. Loy, M. S. Shephard, and K. E. Jansen, *Scalable implicit flow solver for realistic wing simulations with flow control*, Comput. Sci. Eng., 16 (2014), pp. 13–21, https://doi.org/10.1109/MCSE.2014.75.

[41] W. C. Rheinboldt and C. K. Mesztenyi, *On a data structure for adaptive finite element mesh refinements*, ACM Trans. Math. Software, 6 (1980), pp. 166–187, https://doi.org/10.1145/355887.355891.

[42] H.-Y. Schive, Y.-C. Tsai, and T. Chiueh, *Gamer: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics*, Astrophys. J. Suppl. Ser., 186 (2010), pp. 457–484.

[43] P. M. Selwood and M. Berzins, *Parallel unstructured tetrahedral mesh adaptation: Algorithms, implementation and scalability*, Concurrency Practice Experience, 11 (1999), pp. 863–884, https://doi.org/10.1002/(SICI)1096-9128(19991210)11:14<863::AID-CPE464>3.0.CO;2-T.

[44] J. R. Shewchuk, *Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator*, in Applied Computational Geometry: Towards Geometric Engineering, M. C. Lin and D. Manocha, eds., Lecture Notes in Comput. Sci. 1148, Springer, 1996, pp. 203–222.

[45] H. Si, *TetGen—A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator*, Weierstraß Institute for Applied Analysis and Stochastics, Berlin, 2006.

[46] W. Sierpiński, *Sur une nouvelle courbe continue qui remplit toute une aire plane*, Bull. Acad. Sci. Cracovie Sér. A, 1912, pp. 462–478.

[47] W. Skamarock, J. Oliger, and R. L. Street, *Adaptive grid refinement for numerical weather prediction*, J. Comput. Phys., 80 (1989), pp. 27–60, https://doi.org/10.1016/0021-9991(89)90089-2.

[48] C. W. Smith, M. Rasquin, D. Ibanez, K. E. Jansen, and M. S. Shephard, *Application Specific Mesh Partition Improvement*, Tech. report 2015-3, Rensselaer Polytechnic Institute, 2015, https://www.scorec.rpi.edu/REPORTS/2015-3.pdf.

[49] D. A. Steinman, J. S. Milner, C. J. Norley, S. P. Lownie, and D. W. Holdsworth, *Image-based computational simulation of flow dynamics in a giant intracranial aneurysm*, Amer. J. Neuroradiology, 24 (2003), pp. 559–566.

[50] J. R. Stewart and H. C. Edwards, *A framework approach for developing parallel adaptive multiphysics applications*, Finite Elements Anal. Design, 40 (2004), pp. 1599–1617, https://doi.org/10.1016/j.finel.2003.10.006.

[51] 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, https://doi.org/10.1137/070681727.

[52] T. Tu, D. R. O'Hallaron, and O. Ghattas, *Scalable parallel octree meshing for TeraScale applications*, in SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, ACM/IEEE, 2005, https://doi.org/10.1109/SC.2005.61.

[53] T. Weinzierl and M. Mehl, *Peano—a traversal and storage scheme for octree-like adaptive Cartesian multiscale grids*, SIAM J. Sci. Comput., 33 (2011), pp. 2732–2760, https://doi.org/10.1137/100799071.

[54] Y. Yilmaz, C. Özturan, O. Tosun, A. H. Özer, and S. Soner, *Parallel Mesh Generation, Migration and Partitioning for the Elmer Application*, Tech. report, PREMA-Partnership for Advanced Computing in Europe, 2010.

[55] M. Zhou, O. Sahni, K. D. Devine, M. S. Shephard, and K. E. Jansen, *Controlling unstructured mesh partitions for massively parallel simulations*, SIAM J. Sci. Comput., 32 (2010), pp. 3201–3227, https://doi.org/10.1137/090777323.

[56] G. Zumbusch, *Parallel Multilevel Methods. Adaptive Mesh Refinement and Load Balancing*, Vieweg+Teubner Verlag, 2003.