AN OPTIMIZED, PARALLEL COMPUTATION OF THE GHOST LAYER FOR ADAPTIVE HYBRID FOREST MESHES

JOHANNES HOLKE[*], DAVID KNAPP[†], AND CARSTEN BURSTEDDE[‡]

Abstract. We discuss parallel algorithms to gather topological information about off-process mesh neighbor elements. This information is commonly called the ghost layer, whose creation is a fundamental, necessary task in executing most parallel, element-based computer simulations. Approaches differ in that the ghost layer may either be inherently part of the mesh data structure that is maintained and modified, or kept separate and constructed/deleted as needed.

In this work, we present an updated design following the latter approach, which we favor for its modularity of algorithms and data structures. We target arbitrary adaptive, non-conforming forest-of-(oc)trees meshes of mixed element shapes, such as cubes, prisms, and tetrahedra, and restrict ourselves to face-ghosts. Our algorithm has low complexity and redundancy since we reduce it to generic codimension-1 subalgorithms that can be flexibly combined. We cover several existing solutions as special cases and optimize further using recursive, amortized tree searches and traversals.

Key words. Adaptive mesh refinement, parallel algorithms, forest of octrees, ghost layer

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

1. Introduction.

In the parallel mesh-based numerical solution of partial differential equations, the notion of a ghost or halo layer is ubiquitous. It refers to connectivity information about all elements owned by any remote process and directly adjacent to at least one process-local element. As such, it is implemented in many general purpose software packages; see for example [1, 9, 10, 26]. In practice, its effect is to guarantee identical results up to roundoff, independent of the parallel partitioning of the mesh.

If the numerical method only couples directly adjacent elements, which applies to most finite and spectral element methods, the combined set of variables on local and ghost elements suffices to complete a basic global step of the method, be it the assembly of a system matrix or an explicit or implicit solve. In particular with adaptive refinement, the ghost layer aides in globally numbering the degrees of freedom and in computing partition-independent refinement and coarsening indicators.

The concept of the ghost layer is widely applied due to several benefits it provides, such as the locality of parallel communication, the transparency to the discretization code, and the overlap of communication and computation it encourages. If the mesh structure is replicated in parallel, information on the individual process partition and the ghost elements is replicated, too, providing a global view of the partition data for every process. If, on the other hand, the mesh is distributed in parallel, constructing the ghost layer becomes a parallel algorithm in its own right.

When using unstructured meshes, the ghost layer is often part of the graph-based encoding of the mesh. Graph partitioners [8, 11, 18] can be executed and the result queried for both ghost and local elements, often encoded by lookup tables or other convenient data structures; see e.g. [21, 22, 25, 31, 32]. Tree-based meshes, on the other hand, often represent the ghost information implicitly using hierarchy and coordinates [27, 33]. In some approaches, the ghost layer is inherently part of the mesh data structure, which removes the need for its explicit computation but requires to keep the ghost layer synchronized for consistency [1, 28, 34].

---

[*] German Aerospace Center (DLR), Cologne, Germany (johannes.holke@dlr.de)

[†] Rheinische Friedrich-Wilhelms-Universität Bonn, Germany, and DLR, Cologne, Germany

[‡] Institut für Numerische Simulation (INS), Rheinische Friedrich-Wilhelms-Universität Bonn

The present work focuses on the alternative that the ghost layer is not considered first-class mesh data, but may be constructed when needed by executing a suitable algorithm [7]. Using this extra algorithm adds a cost, but simplifies the core mesh data structure as a benefit. In addition, it improves modularity and permits to optimize the ghost layer computation independently of other meshing algorithms [15], and it allows to omit the ghost construction altogether when it is not required by the numerical method due to more general alternatives [4].

The frame for our algorithm development is set by the forest-of-(oc-)trees approach to meshing [2, 7, 29], and the implementation is provided within the `t8code` software library [5, 14]. The unique property of this set of algorithms is that it operates on hybrid meshes, that is, one mesh may contain mixed shapes such as triangles and quadrilaterals in 2D or tetrahedra, prisms, and hexahedra in 3D. Refinement is tree-based and allows for hanging faces, which is unusual in a way but lends many of the benefits of hexahedral forest/tree-structures to the hybrid case.

1.1. Contributions.

In this paper we present two novel contributions. Firstly, we extend the computation of a (face-only) ghost layer for forest-based AMR to meshes with arbitrary element shapes, and in particular hybrid meshes. Secondly, we optimize the proposed algorithm to obtain optimal runtime.

To achieve the first goal, the most important step is the construction of (same-level) face-neighbors across tree boundaries. This is of particular interest if the connected trees have different shapes. The challenging part here is to perform the necessary transformations to account for tree-to-tree coordinate changes. Since the underlying low-level implementations for the element shapes should be exchangeable, it is crucial to avoid dependencies between these implementations. Such dependencies would for example arise if we directly transformed the coordinates of one element (for example a hexahedron) into coordinates of the neighbor element (for example a prism). Instead, our proposed approach is to construct the $(d-1)$-dimensional face element as an intermediate object. We then perform the necessary coordinate transformation in $d-1$ dimensions and extrude the resulting element into the desired d-dimensional face-neighbor.

To achieve the second goal, the optimization of runtime, we utilize recent developments of tree-based search routines [15] to exclude locally surrounded portions of the mesh and thus limit our computational effort to the partition boundary elements. For further technical details and background, as well as the in-depth discussion of triangular and tetrahedral space-filling curves, we refer to H.'s thesis [13].

1.2. Fundamental concepts.

Throughout this document, we assume a forest-of-trees mesh structure. The tree roots can be of any shape as long as their faces conform to all neighbor trees. For example, a hexahedron and a tetrahedron tree may both connect to a prism tree but not to each other. The trees are refined recursively, and the number of refinements from the root to an element is called its level. Thus, two elements may be a descendant or ancestor of each other (in fact both if they are equal) or unrelated. Given this generality, the number of child elements n may be a constant 4 (triangles/quadrilaterals) or 8 (tetrahedra/cubes/prisms), but also a different number, and even varying within the tree. For example, we might consider the one-dimensional line, $n = 2$, a Peano-style $1 : n = 3^d$ refinement of cubes [24, 35], or use a Peano refinement on even and Morton refinement on odd levels. Only the leaf elements of the forest are maintained in memory as true mesh elements, which is often described as linear tree storage [30]. The ghost layer will be assembled as a linear array as well, augmented with offset arrays to define ranges of ghost elements on the same process or in the same tree.

For each element, we assume that sub-algorithms exist to count and index its faces, to construct its parent or any of its children, et cetera. We consider these sub-algorithms an opaque, low-level functionality: They will vary by implementation and by shape, and we do not wish to depend on their internal mechanisms. Instead we impose abstract consistency requirements between the refinements of volumes, faces, and edges, within and between trees. For example, if we consider the faces of a tree as separate $(d-1)$-dimensional refinement trees, then the refinement of the volume cells restricted to a tree face must be a possible face refinement. This approach enables modularity and extensibility and keeps the technical complexity low [5].

In the forest, the connectivity between trees across tree-faces is a mesh of its own. This “genesis mesh” [29] or coarse mesh [2] is conforming even though the elements may become arbitrarily non-conforming by adaptive refinement. Throughout this document, we assume that we can access the coarse mesh information of each neighbor tree of a local tree. This is ensured since the coarse mesh is either replicated on all processes [7] or stores a layer of ghost trees, where a ghost tree is understood as the topological shell without regarding the elements in it. We have previously proposed sharp parallel algorithms to gather the ghost trees [6].

Definition 1. *A ghost element (or just ghost for short) of a process p in a forest $\mathscr{F}$ is a leaf element G of a process $q \neq p$, such that there exists a face-neighbor E of G that is a local leaf element of p.*

Definition 2. *We call a local leaf element E of a process p a partition boundary element if it has at least one face-neighbor that is a ghost element (the term “mirror element” has been used as well [12]). The remote processes to E are all processes $q \neq p$ that own ghost elements of E. The union of all remote processes over all local elements of p are the remote processes of p.*

Definition 3. *We say that an element is a locally surrounded element of process p if all of its leaf descendants and all of its leaf face-neighbors are owned by p.*

Definition 4. *By R_p^q we denote the set of partition boundary elements of process p that have process q as a remote process. The ghost layer for process p is thus*

$$\mathscr{G}_p = \cup_q R_p^q. \tag{1}$$

By construction, we have the following symmetry:

$$R_p^q \neq \emptyset \qquad \Leftrightarrow \qquad R_q^p \neq \emptyset. \tag{2}$$

Adaptation of the element mesh proceeds recursively from the root, which assigns a unique level $\ell \geq 0$ to each element. Note that for forests neither 2:1 balanced nor otherwise graded, the number of neighbors of an element E that are ghosts can be arbitrarily large. It is only bounded by the number of elements at maximum refinement level that can touch the faces of E. Therefore, the number of remote processes is not easily bounded from above.

1.3. Technical procedure. We will describe two variants of constructing the ghost layer. The algorithm `Ghost_tentative` is the first to create a ghost layer for arbitrary hybrid forests. Optimizations of its runtime lead to the final version `Ghost_optimized`.

When referring to the shape of an element, we refer to the associated low-level operations at the same time. In our reference implementation `t8code` [14] we provide line, quadrilateral, and hexahedral elements ordered by the Morton index [23], as well as triangular and tetrahedral elements using the tetrahedral Morton (TM-)index [5]. This also provides us with an implementation of prism elements, since we can model these as the cross product of a line and a triangle [19]. Additionally, in ongoing work we are developing and implementing pyramidal elements [20].

The basic idea of `Ghost_tentative` is to first identify all partition boundary elements and their remote processes, thus building the sets R_p^q and identifying the non-empty ones. In a second step, each process p sends all elements in R_p^q to q. The senders are known to the receivers due to the symmetry of the communication pattern (2). In the first step we iterate over all local leaves and for each over all of its faces. We then have to decide for each face F of a leaf E which processes own leaves that touch this face.

In `p4est`, the runtime is optimized by performing a so called (3×3)-neighborhood check of an element [15] inspired by the "insulation layer" concept [30]. For a local hexahedral/quadrilateral element, it is tested whether all possible same-level face- (or edge-/vertex-) neighbors would also be process-local and if so, the element is excluded from further processing. Since this check makes explicit use of the classical Morton code and its properties, it is difficult to generalize to hybrid meshes.

For `Ghost_optimized` we replace the iteration over all leaves with a neighbor-aware top-down forest traversal. While traversing, we exclude locally surrounded elements from the iteration, which is equivalent to an early pruning of the search tree. This approach supersedes the (3×3) test and improves the overall runtime over simpler, iterative algorithms.

We design our algorithms shape-independent from the beginning. To this end, we require a minimal set of element sub-algorithms that we develop in Section 2. We discuss fast algorithms of finding the owner process of neighbor elements in Section 3 and expose the high-level forest traversal algorithms to construct $\mathscr{G}_p$ beginning with Section 4.

2. Low-level element functions.

We will adhere throughout to the abstraction of low-level, per-element algorithms on the one hand and high-level, global parallel algorithms on the other. Introducing for example a new element shape like the pyramid, or an alternative space filling curve such as the Hilbert curve, will be implemented on the low-level side without requiring a change in the high-level algorithms. Conversely, improving the high-level algorithms further will be possible with or without defining new low-level interface functions, depending on the algorithmic idea.

We will use this section to add several low-level functions that we require for the high-level algorithms formulated later in this document. They all deal with direct element face neighbors. We will motivate and discuss the abstract interface first and then propose additional specifics for the (T-)Morton curves currently available in `p4est` and `t8code`. In doing so, we introduce conventions necessary for the reader to substitute their favorite element implementation if so desired.

An important part of any `Ghost` routine is to construct the same-level neighbor of a given element E across a face f. Here, to construct means to compute information defining this neighbor as a possible (hypothetical) element in a mesh, not necessarily as a leaf that exists on this or another process. The hypothetical element can than be compared with the existing local leaves (which may be descendants or ancestors) or the partition boundaries (which are encoded using deepest-level hypothetical elements).

As long as such a face-neighbor remains inside the same tree as E, this prob-

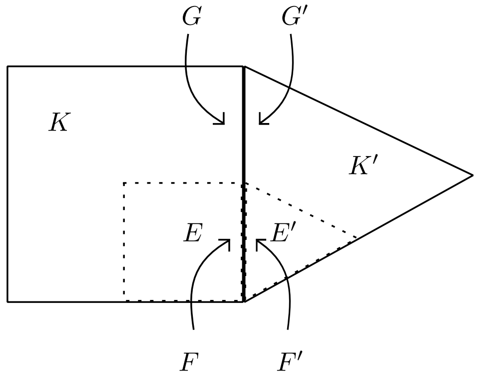


Fig. 1: A tree K, element E, and a face F of E that is a subface of a tree face. The task is to construct the face-neighbor element E'. A subtask is to identify the tree faces G and G', taking into account the coordinate systems of both trees.

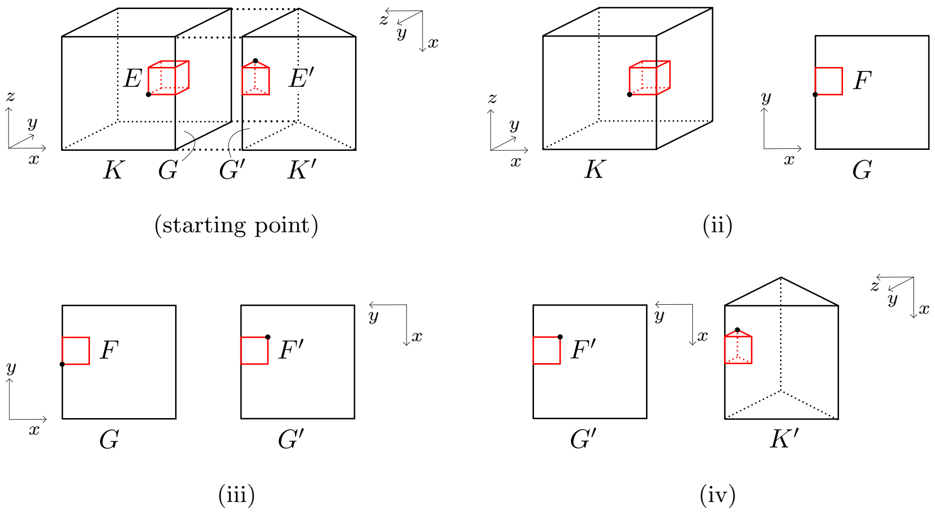
lem is addressed by the corresponding low-level function `t8_element_face_neighbor_inside`; see [7] for an implementation for the classical Morton index, [5, Algorithm 4.6] for the TM-index and [19] for prisms. It is more challenging to find element face-neighbors across tree boundaries. One reason is that neighbor trees may be rotated against each other. For hybrid meshes, a new challenge occurs in that multiple shapes of trees exist in the same forest.

To note down our proposed solution, we use capital letters (K, E, F, G) for entities such as trees, elements, and faces, and use lower case for indices (see Figure 1). Since the coordinate systems of neighbor trees may not be aligned, we must properly transform the $(d-1)$-dimensional coordinates of the faces between the two. To decouple neighbor trees of different shapes, we consider the face G of the tree K as a $(d-1)$-dimensional root element and explicitly construct the face F of E as a $(d-1)$-dimensional element descendant of G. Thus, we identify four major substeps in the computation of face-neighbors across tree boundaries (see Figure 2):

(i) From an element's face F at a tree boundary, identify the corresponding tree face G.
(ii) Construct the $(d-1)$-dimensional face element F.
(iii) Transform the coordinates of F to obtain the neighbor face element F'.
(iv) Extrude F' to the d-dimensional neighbor element E'.

Rationale 5. *We deliberately choose this method of using lower dimensional entities over directly transforming the tree coordinates from one tree to the other—as it is done for example in [7]—since our approach allows for maximum flexibility of the implementations of the different element shapes and SFC choices. This holds since all intermediate operations are either local to one element or change the dimension (i.e. hexahedra to quadrilaterals, tetrahedra to triangles, and back), but not both. Therefore, even if, for example, a hexahedron tree is neighbor to a prism tree, no function in the implementation of the hexahedral elements relies on knowledge about the implementation of the prism elements. Hence, it is possible to exchange the definition of the SFC for one element shape without changing the others.*

Remark 6. *We define a (relative) orientation of two neighboring trees. This def-*



(starting point) (ii)

(iii) (iv)

Fig. 2: A hexahedron and a prism element that are face-neighbors across tree boundaries. Constructing the face-neighbor E' of E across the face amounts to computing its anchor node (black) from the anchor node of E and the coarse mesh connectivity information about the two neighbor trees. Here, the coordinate systems of the two trees are rotated against each other. In step (ii) we construct the face element F from the element E. The coordinate system of the face root is inferred from that of the left tree. In step (iii) we transform F to the neighbor face element F'. In the last step (iv) we extrude the face-neighbor E' from the face element F'.

inition is shape-independent, and all low-level implementations must adopt (or translate into) it. The particulars follow in Section 2.2.

Remark 7. *The theory of the TM-index for triangles and tetrahedra uses the type to classify distinct sorts of occuring elements; see Figure 3. For triangles there are the two types 0 and 1, while for tetrahedra there are the types 0 through 5. By definition of the TM-index, the type of the root simplex is always 0 and the other types emerge on finer refinement levels. For more details see [5].*

2.1. (i) Identifying the tree face. The first subproblem is to identify the tree face G and its face number g from E, f, and the tree K. For this task we introduce a new low-level function:

$g \leftarrow$ `t8_element_tree_face (element` E`, face number` f`)`
The element face number f designates a subface of a tree face. Return the face number g of this tree face. Only valid if face f of E is on a tree boundary.

For lines, quadrilaterals, and hexahedra with the Morton index, the tree face indices are the same as the element's face indices [7] and thus `t8_element_tree_face` always returns $g = f$.

For simplices with the TM index [5], the enumeration of their faces depends on their simplex type. By convention, the face number i refers to the unique face that does not contain the vertex $\vec{x}_i$, and the vertex numbering relative to the surrounding

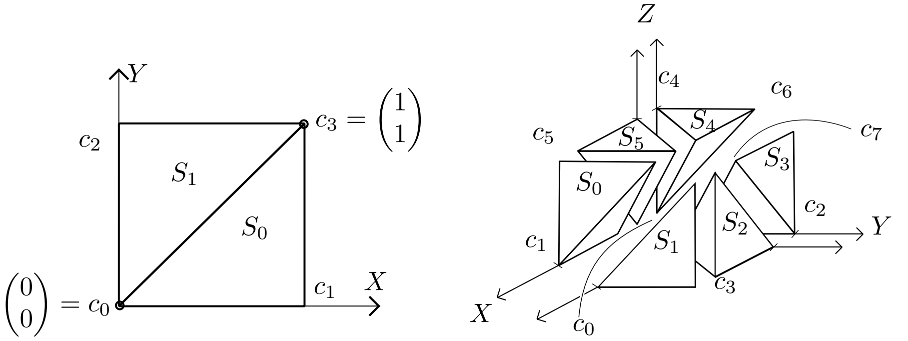


Fig. 3: The basic types i for triangles (2D) and tetrahedra (3D) S_i are obtained by dividing $[0,1]^d$ into simplices. Left: The unit square can be divided into two triangles sharing the diagonal edge from $(0,0)^T$ to $(1,1)^T$. The four corners of the square are numbered $c_0,\ldots,c_3$ in yx-order. Right (exploded view): The unit cube can be divided into six tetrahedra, all sharing the diagonal edge from the origin to $(1,1,1)^T$. The eight corners of the cube are numbered $c_0,\ldots,c_7$ in zyx-order. The tetrahedral Morton (TM-)index is constructed by bitwise interleaving the type with the coordinates of an element's lower left corner (drawings adapted by permission [3]).

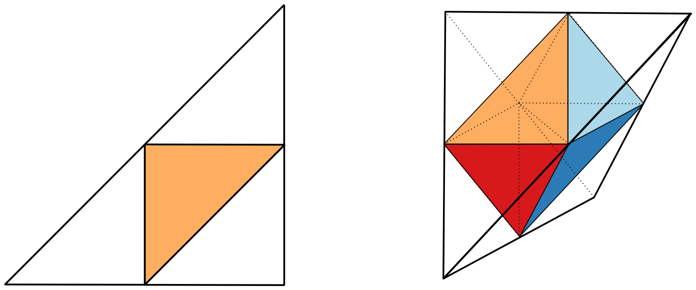


Fig. 4: First refinement level for triangles (left) and tetrahedra (right) with types color-coded. The TM-curve assumes a type 0 root element, and during refinement other types occur. For triangles the only other type is 1 (orange). For tetrahedra we get the types 0 (white), 1 (orange), 2 (light blue), 4 (dark blue) and 5 (red) on level 1, while type 3 occurs first on level 2 and never on the tree's boundary.

cube corners differs by type (Figure 3).

Since the root simplex always has type 0, for triangles and tetrahedra of type 0 the face number is the same as the face number of the root element. Triangles of type 1 and tetrahedra of type 3 cannot lie on the boundary of the tree and thus we never call `t8_element_tree_face` for these elements (Figure 4).

For each of the remaining four tetrahedron types there is exactly one face that can lie on the tree boundary. Face 0 of type 1 tetrahedra is a descendant of the root face 0; face 2 of type 2 tetrahedra is a descendant of the root face 1; face 1 of type 4 tetrahedra is a descendant of the root face 2. Finally, face 3 of type 5 tetrahedra is a descendant of the root face 3.

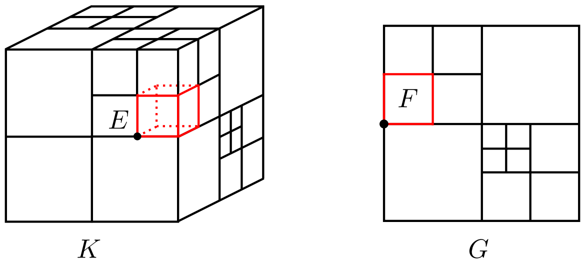


Fig. 5: Constructing the face element F to an element E at a tree face G. We can interpret the face of the 3D tree K as a 2D tree G. The face F of E is an element in this tree.

Note that for face indices f of faces that cannot lie on the tree boundary, calling `t8_element_tree_face` is illegal. This behavior is well-defined, since we ensure that the function is only called if the face f lies on the tree boundary.

2.2. (ii) Constructing the face element. As a next step, we build the face F as a $(d-1)$-dimensional element. We do this via the low-level function:

$F \leftarrow$ `t8_element_boundary_face (element` E`, face number` f`)`
Return the $(d-1)$-dimensional face element F of element E specified by the face number f. Required for all elements of positive dimension.

In other words, the lower dimensional face element F is created from E. For the Morton index this is equivalent to computing the coordinates of its anchor node and additionally its type for the TM-index. Hereby we interpret the tree face G as a $(d-1)$-dimensional root element of which F is a descendant element; see also Figure 5.

Remark 8. *Since we construct a lower-dimensional element as the face of a higher-dimensional one, there are two conditions that need to be satisfied for the implementations of the two element shapes involved.*

1. *The refinement pattern of a face of the higher dimensional elements must conform to the lower dimensional refinement pattern.*
2. *The maximum possible refinement level of higher dimensional elements must not exceed the one of the lower dimensional elements.*

If one or both of these conditions are not fulfilled, then there exist faces of the higher dimensional elements for which an interpretation as a lower dimensional element is not possible. For Morton-type SFCs, these two conditions are naturally fulfilled.

Remark 9. *For the simplicial and hexahedral Morton SFC with maximum refinement level* $\mathscr{L}$*, the anchor node coordinates of an element of level* ℓ *are integer multiples of* $2^{\mathscr{L}-\ell}$*. Suppose the maximum level of hexahedral elements is* $\mathscr{L}_1$ *and the maximum level of a face boundary quadrilateral element is* $\mathscr{L}_2 \geq \mathscr{L}_1$*, then we will have to multiply a hexahedral coordinate with* $2^{\mathscr{L}_2-\mathscr{L}_1}$ *to transform it into a quadrilateral coordinate. For simplicity, we reduce our presentation to the case that all element shapes have the same maximum possible refinement level and omit the scaling factor.*

For simplices with the TM-index, we note that we shall restrict ourselves to those combinations of element and face number that occur on the tree boundary. In particular, all possible faces are subfaces of the faces of the root simplex S_0.

Triangles of type 1 never lie on the tree boundary, hence we only need to consider type 0 triangles. The result solely depends on the face number f.

A tetrahedron that lies on the tree boundary has a type different from 3. In order to compute the boundary face, we distinguish two cases. Let g be the face of the root tetrahedron S_0 corresponding to the boundary face f of T.

1. $g = 0$ or $g = 1$. These faces of S_0 lie in the $(x = 0)$-plane or the $(x = z)$-plane of the coordinate system, and $(F.x, F.y) = (T.z, T.y)$.
2. $g = 2$ and $g = 3$. These faces lie in the $(y = 0)$-plane or the $(y = z)$-plane, and the anchor node of F is given by $(F.x, F.y) = (T.x, T.z)$.

2.3. (iii) Constructing F' from F. If we know the tree face number g, we can look up the corresponding face number g' of the face in K' from the coarse mesh connectivity [6].

In order to transform the coordinates of F to obtain F' we need to understand how the vertices of the face g connect to the vertices of the face g'. Each face's vertices form a subset of the vertices of the trees. Let $\{ v_0, \dots, v_{n-1} \}$ and $\{ v'_0, \dots, v'_{n-1} \}$ be these vertices for g and g' in ascending order, thus $v_i < v_{i+1}$ and $v'_i < v'_{i+1}$. The face-to-face connection of the two trees determines a permutation $\sigma \in S_n$ such that vertex v_i connects to vertex $v'_{\sigma(i)}$. In theory, there are $n!$ possible permutations. However, not all of them occur.

Definition 10. *Since we exclude trees with negative volume, there is exactly one way to connect two trees across the faces g and g' in such a way that the vertices v_0 and v'_0 are connected. We call the corresponding permutation σ_0.*

We obtain all other possible permutations σ by rotating the face g'. This rotation is encoded in the orientation information of the coarse mesh.

Definition 11 (From [6, Definition 2.2]). *The* orientation *of a face connection is the index j such that v_0 connects with v'_j. Thus,*

$$\text{orientation}(g, g', \sigma) = \sigma(0). \tag{3}$$

Remark 12. *If we look at the same face connection, but change the order of g and g', the permutation σ becomes σ^{-1}. In 3D $\sigma(0)$ is in general not equal to $\sigma^{-1}(0)$ and thus the orientation depends on the order of the faces g and g' (if unequal). In order to make the orientation unique, we use the following convention: If K and K' have the same shape then the smaller face is considered as g. If K and K' have different shapes, we consider g as the face of the smaller shape, regarding the order: hexahedron < prism < pyramid; tetrahedron < prism < pyramid [6].*

From the initial permutation σ_0 and the orientation we can reconstruct σ. σ_0 is determined by the shapes of K and K' and the face indices g and g'. In fact, since the orientation encodes the possible rotations, the only data we need to know is the sign of σ_0.

Definition 13. *Let K and K' be two trees of shapes t and t', and let g, g' faces of K and K' of the same element shape. We define the* sign *of g and g' as the sign of the permutation σ_0,*

$$\text{sign}_{t,t'}(g, g') := \text{sign}(\sigma_0). \tag{4}$$

REMARK 14. *This definition does not depend on the order of the faces g and g', since*

$$\operatorname{sign}_{t',t}(g', g) = \operatorname{sign}(\sigma_0^{-1}) = \operatorname{sign}(\sigma_0) = \operatorname{sign}_{t,t'}(g, g'). \tag{5}$$

Using the orientation, the sign, and the face number g', we transform the coordinates of F to obtain the corresponding face F' as a subface of the face G' of K'. For this task we introduce the low-level function

$F' \leftarrow$ `t8_element_transform_face`
(`face element` F, `orientation` o, `sign` s).

REMARK 15. *The transformation $o = i$, $s = -1$ is the same as first using $o = 0$, $s = -1$ and then $o = i$, $s = 1$. Thus, we only need to implement all cases with $s = 1$ and one additional case $o = 0$, $s = -1$.*

REMARK 16. *The sign is always 0 for the boundary of line elements. If the faces are lines there are two possible face-to-face connections and these are already uniquely determined by the orientation of the connection. Thus, for 1D and 2D trees (lines, quadrilaterals, and triangles) it is not necessary to use the sign.*

For hexahedra with the Morton index we compute the sign of two faces via the tables $\mathcal{R}, \mathcal{Q}, \mathcal{P}$ from [7, Table 3] as

$$\operatorname{sign}_{\text{hex,hex}}(g, g') = \operatorname{sign}\left(i \mapsto \mathcal{P}\left(\mathcal{Q}(\mathcal{R}(g, g'), 0), i\right)\right) = \neg\mathcal{R}(g, g'). \tag{6}$$

The permutation in the middle is exactly the permutation σ_0. The argument 0 of $\mathcal{Q}$ is the orientation of a face-to-face connection, but the result is independent of it, and we could have chosen any other value.

For the classical and tetrahedral Morton indices we need to compute the anchor node of F' from the anchor node of the input face F. For triangle and quadrilateral faces, we may do this explicitly for $o = 0$, $s = -1$ and then derive all other combinations by Remark 15. For the faces of quadrilaterals and hexahedra, `t8_element_transform_face` is equivalent to the internal coordinate transformation `p4est_transform_face` due to (6).

2.4. (iv) Constructing E' from F'. We now have E, F, F', K and K' and can construct the neighbor element E'. For this we introduce the function

$E' \leftarrow$ `t8_element_extrude_face`
(`face element` F', `tree` K', `face number` g').

This function has as input a face element and a tree face number and as output the element within the tree that has as a boundary face the given face element. How to compute the element from this data depends on the element shape and the tree face.

2.5. Supporting local surround. Our high-level ghost algorithm is an extension of a top-down tree traversal. The extension lies in the fact that we may prune subtrees that are locally surrounded, that is, their direct face neighbors are all process-local. To this end, we require nearest-neighbor context, which is not ordinarily available in a top-down recursion. Thus, we recreate parallel neighbor context in an optimized way in the high-level algorithms of Section 4, which requires two more low-level functions that we describe in the following.

$C[\,] \leftarrow$ `t8_element_children_at_face (element` E`, face number` f`)`
Returns an array of children of E that share a face with f.

$f' \leftarrow$ `t8_element_child_face`
`(element` E`, child index` i`, face number` f`)`
Given an element E, a child index i of E, and a face number f of some face F of E, compute the number f' of the child's face that is a subface of F. It is required that the child lies on the face F.

A typical implementation of `t8_element_children_at_face` would look up the child indices of these children in a table and then construct the children with these indices. The child indices can be obtained from the refinement pattern. For the quadrilateral Morton index, for example, the child indices at face $f = 0$ are 0 and 2. For a hexahedron the child indices at face $f = 3$ are 2, 3, 6, and 7. For the TM index these indices additionally depend on the type of the simplex.

The low-level algorithm `t8_element_child_face` can also be described via lookup tables. Its input is a parent element E, a face number f and a child index i, such that the child E_i of E has a subface of the face f. In other words, E_i is part of the output of `t8_element_children_at_face`. The return value of `t8_element_child_face` is the face number f_i of the face of $E[i]$ that is the subface of f.

For the classical Morton index, the algorithm is the identity on f, since the faces of child quadrilaterals/hexahedra are labeled in the same manner as those of the parent element. For the TM index for triangles, the algorithm is also the identity, since only triangle children of the same type as the parent can touch a face of the parent and for same type triangles the faces are labeled in the same manner.

For tetrahedra with the TM-index, the algorithm is the identity on those children that have the same type as the parent. However, for each face f of a tetrahedron T, there exists a child of T that has the middle face child of f as a face. This child does not have the same type as T. For this child the corresponding face value is computed as 0 if $f = 0$, 2 if $f = 1$, 1 if $f = 2$, or 3 if $f = 3$.

3. Owner processes of elements.

For any ghost algorithm, after we have successfully constructed an element's full-size or refined face-neighbor, we need to identify the owner process of this neighbor. We can use this information to shorten the list of potential neighbor processes, eventually arriving at the tightest possible set to communicate with.

DEFINITION 17. *Let E be an element in a (partitioned) forest. A process p is an* owner *of E if there exists a leaf L in the forest such that*

1. *L is in the partition of p, and*
2. *L is an ancestor or a descendant of E.*

Unique ownership is thus guaranteed for leaf elements and their descendants, but not for every ancestor element of a tree. In any case, each element has at least one owner.

DEFINITION 18. *The* first/last descendant *of an element E is the descendant of E of maximum refinement level with smallest/largest SFC index.*

Since first/last descendants cannot be refined further, they are either a leaf or descendants of a leaf. Hence, they have a unique owner process. (Note however that, depending on the chosen space filling curve, the first and last descendants of an element need not be placed at its corners.) Since a forest is always partitioned along the SFC in ascending order, it must hold for each owner process p of E that

$$p_{\text{first}}(E) \leq p \leq p_{\text{last}}(E). \tag{7}$$

Conversely, if a process p fulfills inequality 7 and its partition is not empty, then it must be an owner of E. Furthermore, we conclude that an element has a unique owner if and only if $p_{\text{first}}(E) = p_{\text{last}}(E)$.

Each process can compute the SFC index of the first descendant of its first local element. From these SFC indices we build an array of size P, which is the same on each process. We can then determine the owner process of any first or last descendant by performing a binary search in this array if we combine it with the array of per-process tree offsets [7]. We call this functionality `t8_forest_owner`, which in practice runs over a subwindow of the array narrowed down by the top-down tree traversal.

Algorithm 3.1: `t8_owners_at_face` (forest $\mathscr{F}$, element E, face number f)

```
Result: The set $P_E$ of all processes that own leaf elements that are
    descendants of $E$ and have a face that is a subface of $f$
$P_E \leftarrow \emptyset$
fd $\leftarrow$ t8_element_first_desc_face ($E$, $f$)          /* First and last */
ld $\leftarrow$ t8_element_last_desc_face ($E$, $f$)  /* descendant of $E$ at $f$ */
$p_{\text{first}} \leftarrow$ t8_forest_owner ($\mathscr{F}$, fd)        /* The owners of fd and ld */
$p_{\text{last}} \leftarrow$ t8_forest_owner ($\mathscr{F}$, ld)
if $p_{\text{first}} \in \{p_{\text{last}}, p_{\text{last}} - 1\}$ then          /* Only $p_{\text{first}}$ and $p_{\text{last}}$ are */
    return $\{p_{\text{first}}, p_{\text{last}}\}$                    /* owners of leaves at $f$ */
else          /* There may be other owners.  Enter the recursion */
    $C_f[] \leftarrow$ t8_element_children_at_face ($E$, $f$)
    for $0 \leq i <$ t8_element_num_face_children ($E$, $f$) do
        $j \leftarrow$ child_index ($C_f[i]$)       /* Child number relative to $E$ */
        $f' \leftarrow$ t8_element_child_face ($E$, $j$, $f$)
        $P_E \leftarrow P_E \cup$ t8_owners_at_face ($\mathscr{F}$, $C_f[i]$, $f'$)
    return $P_E$
```

For the `Ghost_tentative` algorithm—a generalization of [15] to hybrid shapes—we will have to identify all owners of leaves that are neighbors directly adjacent to a given element's face. Since **p4est**'s algorithm **find_range_boundaries** [15] is highly efficient, but specific to the Morton SFC and hypercubes, we introduce the Algorithm 3.1 **t8_owners_at_face**. Given an element E and a face f, it determines the set P_E of all processes that have leaf elements that are descendants of E and share a face with f.

Definition 19. *The first/last* face descendant *of an element E at a face f is the descendant of E of maximum refinement level that shares a subface with f and has smallest/largest SFC index.*

We denote the owner processes of an element's first and last face descendants by $p_{\text{first}}(E, f)$ and $p_{\text{last}}(E, f)$. If these are equal to the same process q, then q must be the single owner at that face.

As opposed to the owners of an element, not all nonempty processes in the range from $p_{\text{first}}(E, f)$ to $p_{\text{last}}(E, f)$ are necessarily owners of leaves at the face of E; see for example face $f = 0$ in Figure 6. It is thus not sufficient to determine all nonempty

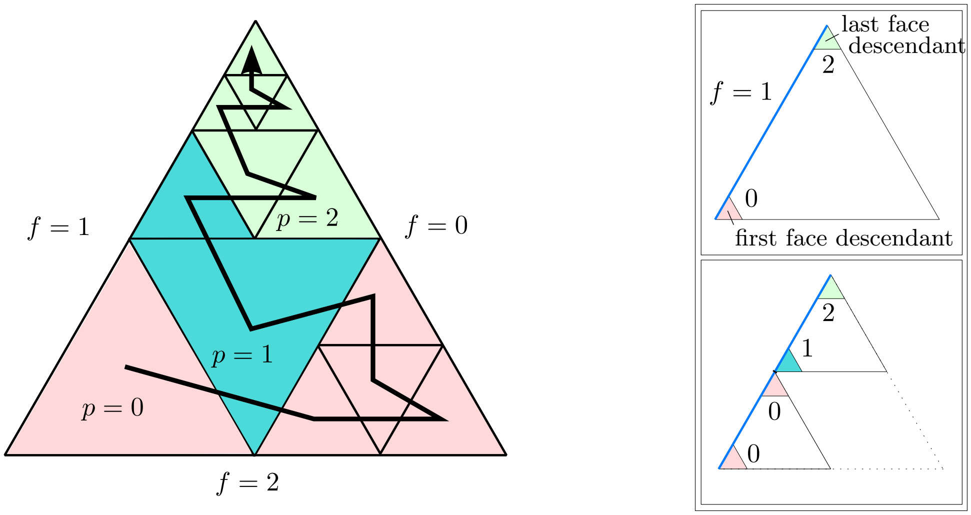


Fig. 6: An example for Algorithm 3.1, `t8_forest_owners_at_face`. Left: A triangle element E with the TM-index as SFC whose descendants are owned by three different processes: 0 (red), 1 (blue), and 2 (green). The owners at the faces are $\{0, 2\}$ at face 0, $\{0, 1, 2\}$ at face 1, and $\{0\}$ at face 2. Right: The iterations of `t8_forest_owners_at_face` at face $f = 1$. At first the first and last descendant of E at f are constructed. We compute their owner processes 0 and 2, and since their difference is greater one, we continue the recursion. In the second iteration the algorithm is called once for the lower left child and once for the upper child of E. We determine their first and last descendants at the respective subface of f. For the lower left child, the recursion stops since both face descendants are owned by process 0. For the upper child the owner processes are 1 and 2 and since there are no other possible owner processes in between, we stop the recursion as well.

processes between $p_{\text{first}}(E, f)$ and $p_{\text{last}}(E, f)$. Instead, if $p_{\text{first}}(E, f) < p_{\text{last}}(E, f) - 1$, we enter a recursion for each child of E that lies on the face f. We terminate early for elements whose descendants at the face f are all owned by a single process, or by two processes whose ranks differ by 1. In practice, this procedure prunes the search tree very quickly.

We optimize our implementation of `t8_owners_at_face` by taking into account that the first and last owners p_f and p_l at the current recursion step form lower and upper bounds for the first and last owners in any upcoming recursion step. Thus, we restrict the binary searches in `t8_forest_owner` to the interval $[p_{\text{first}}, p_{\text{last}}]$ instead of $[0, P - 1]$. We also exploit that the first descendant of an element E at a face f is at the same time the first face descendant of E's first child at f. The same holds for the last descendant and the last child at f. Thus, we reuse the first/last face descendants and owners of E when we enter the recursion with the first/last child at f.

4. The ghost algorithms.

In this section we propose updated ghost algorithms that are at least as scalable as their predecessors. On the one hand, they are currently more limited by being restricted to face neighbors. On the other, they are more general and extensible and applicable to fully hybrid adaptive meshes. In practice, we execute them on over 1e12 elements, which is a novel achievement for a hybrid AMR code.

4.1. Initial design. Our basic design, `Ghost_tentative`, follows the conception of [15] and operates on forests with arbitrary, non-graded refinements. In consequence, there are no assumptions on the relative sizes of neighbor leaves of element E across face F. Algorithmically, we begin by constructing the same-size face neighbor of E. We know that it is either a descendant of a forest leaf (including the case that it is a leaf itself), which then has a unique owner, or an ancestor of multiple forest leaves, which may all have different owners. Among those, we need to compute only the owners of descendant/ancestor forest leaves of E' that touch the face F. To this end, we call Algorithm 3.1, `t8_forest_owners_at_face`, that we describe above in Section 3. In addition, the `Ghost_tentative` Algorithm 4.1 invokes the function `dual_face` (not listed), which, given an element E and a face number f, returns the face number f' seen from the neighboring element.

Algorithm 4.1: `Ghost_tentative` (`forest` $\mathscr{F}$)

```
for $K \in \mathscr{F}$.trees do
    for $E \in K$.elements do
        for $0 \le f <$ t8_element_num_faces ($E$) do
            $E' \leftarrow$ t8_forest_face_neighbor ($\mathscr{F}$, $E$, $f$)
            $f' \leftarrow$ dual_face($E$, $E'$, $f$)
            $P_{E'} \leftarrow$ t8_forest_owners_at_face ($\mathscr{F}$, $E'$, $f'$)
            for $q \in P_{E'}$ do
                if $q \neq p$ then
                    $R_p^q = R_p^q \cup \{ E \}$
```

4.2. Optimizing the runtime. The `Ghost_tentative` Algorithm 4.1 iterates over all local leaf elements to identify the partition boundary leaves on the process. Thus, its runtime is proportional to the number of local leaves. However, for many meshes and partitions, only a small portion of the leaf elements are partition boundary elements, depending on the surface-to-volume ratio of the process's partition. Since the surface of a volume grows by one power less than the volume itself, the number of partition boundary leaves can become arbitrarily small in comparison to the number of all leaves. Ideally, the runtime of `Ghost` should be proportional to the number of partition boundary elements.

Our goal is therefore to improve the runtime of the algorithm by excluding non-boundary leaves from the iteration. In `p4est`, most locally surrounded leaves are excluded from the iteration by testing for each quadrilateral/hexahedron whether its 3×3 neighborhood [7, 30], namely all same-level face-(edge-/vertex-)neighbors, are process-local. However, since this approach uses particular geometrical properties of the quadrilateral/hexahedral shapes and of the Morton index, it is not practical for our hybrid, element-shape independent approach.

To exclude the locally surrounded leaves in `t8code`, we replace the leaf iteration with a top-town traversal, repurposing a recursive approach originally proposed for searches [15]. Starting with a tree's root element, we test whether it may have partition boundary leaf descendants, and if so, we create the children of the element and continue recursively. If we reach a leaf, we test whether it is a partition boundary element—and if so for which processes—in the way described in the previous section. This approach allows us to terminate the recursion as soon as we reach a locally surrounded element, thus saving the iteration over all descendant leaves of that element. Note, however, that each level of the recursion queries neighbors, thus it is not sufficient to rely on parent-child relations alone as in an ordinary top-down scheme.

4.2.1. A recursive top-down scheme. In [15] the authors present the recursive `Search` algorithm for octree AMR, which identifies process-local elements or subtrees and generalizes conceptually to hybrid-shape tree-based AMR. This mechanism is extended to identify remote process owners of abstract objects without further communication [4]. Semi-Lagrangian methods, for example, can effectively exploit both kinds of search to bound communication time and volume, in particular for arbitrary-CFL, ghost-free designs.

We suggest applying `Search` to the problem of identifying all local leaf elements at a process's partition boundary. The `Search` algorithm has been shown to be especially efficient when looking for multiple matching leaves at once [15], which is the case in our setting. The idea is starting with the root element of that tree and recursively creating its children until we identify a leaf element. On each intermediate element we call a user-provided callback function which returns true only if the search should continue with this element. Otherwise, the recursion for this element stops and all descendants are excluded from further processing.

For our version of the ghost algorithm, one of two tasks of the callback is to return false for locally surrounded elements, thus excluding possibly large areas of the mesh from further search and hence accelerating the computation. Once we reach a leaf element, the callback performs its second function, namely to examine the leaf's faces and to compute the leaf owners outside and adjacent to the respective neighbor faces. Effectively, we defer and reduce the inner for-loop of Algorithm 4.1, line 3, to the remaining cases visited by the search callback.

We show our version of `Search` in Algorithm 4.2. It is a simplified version of Algorithm 3.1 in [15] without point queries, since we do not need these for `Ghost_optimized`. We also use the function `split_array` from [15]. This function takes as input an element E and an array L of (process local) leaf elements in E, sorted in SFC order. It returns a set of arrays $\{ M[i] \}$ such that for the i-th child E_i of E the array $M[i]$ contains exactly the leaves in L that are also leaves of E_i, thus $L = \dot{\bigcup}_i M[i]$.

To cover the whole forest, we iterate over only the trees containing process-local leaves, and for each compute the finest element E such that all local leaves are still descendants of E. This identifies E as the nearest common ancestor of the first and last leaf element of the tree. With E and the leaf elements of the tree, we call the `element_recursion`; see Algorithm 4.3.

Algorithm 4.2: `element_recursion` (element E, leaves L, callback `Match`)

```
Require: The leaves in $L$ must be descendants of $E$ and ascending
isLeaf ← $L = \{ E \}$        /* Determine whether $E$ is a leaf element */
if Match($E$, isLeaf) and not isLeaf then
    $M[]$ ← split_array ($L$, $E$)     /* $M[i]$ are no-copy views onto $L$ */
    $C[]$ ← t8_element_children ($E$)
    for $0 \le i <$ t8_element_num_children ($E$) do
        if $M[i] \neq \emptyset$ then
            element_recursion ($C[i]$, $M[i]$, Match)
```

Algorithm 4.3: `t8_forest_search` (forest $\mathscr{F}$, callback `Match`)

```
for $K \in \mathscr{F}$.trees, $K$ contains local leaves do
    $E_1 \leftarrow$ first_tree_element ($\mathscr{F}$, $K$)        /* First and last local */
    $E_2 \leftarrow$ last_tree_element ($\mathscr{F}$, $K$)               /* leaf in the tree */
    $E \leftarrow$ t8_element_nearest_common_ancestor ($E_1$, $E_2$)
    $L \leftarrow$ tree_leaves ($\mathscr{F}$, $K$)        /* No-copy view of tree leaves */
    element_recursion ($E$, $L$, Match)            /* Top-down traversal */
```

Algorithm 4.4: `ghost_match` (element E, bool `isLeaf`)

Result: If E is a leaf, compute the owners of the face-neighbors and add to the sets R_p^q. If not, then terminate if E is a locally surrounded element

```
if isLeaf then            /* $E$ is a leaf.  Compute the owners at */
    for $0 \leq f <$ t8_element_num_faces ($E$) do           /* its faces */
        $E' \leftarrow$ t8_forest_face_neighbor ($\mathscr{F}$, $E$, $f$)
        $f' \leftarrow$ dual_face($E$, $E'$, $f$)
        $P_{E'} \leftarrow$t8_forest_owners_at_face ($\mathscr{F}$, $E'$, $f'$)
        for $q \in P_{E'}$ do
            if $q \neq p$ then
                $R_p^q = R_p^q \cup \{ E \}$
else                                              /* $E$ is not a leaf */
    $p_{\text{first}}(E) \leftarrow$ t8_element_first_owner ($E$)
    $p_{\text{last}}(E) \leftarrow$ t8_element_last_owner ($E$)
    for $0 \leq f <$ t8_element_num_faces ($E$) do
        $E' \leftarrow$ t8_forest_face_neighbor ($\mathscr{F}$, $E$, $f$)
        $f' \leftarrow$ dual_face($E$, $E'$, $f$)
        $p_{\text{first}}(E', f') \leftarrow$ t8_first_owner_at_face ($\mathscr{F}$, $E'$, $f'$)
        $p_{\text{last}}(E', f') \leftarrow$ t8_last_owner_at_face ($\mathscr{F}$, $E'$, $f'$)
        if $p_{\text{first}}(E', f') \neq p$ or $p_{\text{last}}(E', f') \neq p$ then
            return 1   /* Not all face-neighbor leaves owned by $p$ */
    if $p_{\text{first}}(E) = p_{\text{last}}(E) = p$ then
        return 0                           /* Terminate recursion */
return 1                                    /* Continue recursion */
```

4.2.2. The optimized `Ghost` algorithm. The algorithm `t8_forest_search` requires a callback function, in our case `ghost_match` (Algorithm 4.4), motivated as follows. If the element E passed to `ghost_match` is not a leaf, we must decide whether the element and all of its possible face-neighbors are owned by the current process. We opt not to call the function `t8_forest_owner` on E since it might return a large number of processes. We save runtime instead by computing the first and last process that owns leaves of the element and testing whether they are equal. We proceed in analogy for the owners at the neighbor faces. If for E the first and last process is p and at each face-neighbor the first and last owner at the corresponding face is also p, E is a locally surrounded element and cannot have any partition boundary leaves as descendants. Thus, we return 0 and the search does not continue for E's descendants.

If E is a leaf element, then it may or may not be a partition boundary element. Thus, we compute the owner processes of all face-neighbors using `t8_forest_owners_at_face` and add E as a partition boundary element to all of those that are not p.

Note that for each child C of an element E the ranks $p_{\text{first}}(E), p_{\text{last}}(E), p_{\text{first}}(E, f)$, and $p_{\text{last}}(E, f)$ serve as lower and upper bounds to the corresponding ranks for C. Hence in the `t8code` implementation of `ghost_match`, we remember these ranks for each recursion level reducing the search range from $[0, P-1]$ to $[p_{\text{first}}(E), p_{\text{last}}(E)]$ for C, and to $[p_{\text{first}}(E, f), p_{\text{last}}(E, f)]$ for the faces. To track these bounds correctly in practice, we always enter the `for`-loop on Line 12.

Algorithm 4.5: `Ghost_optimized` (forest $\mathscr{F}$)

```
t8_forest_search ($\mathscr{F}$, ghost_match)
```

The final `Ghost_optimized` Algorithm 4.5 is now expressed as a specialized search.

5. Numerical results.

In this section we present various runtime studies obtained using the Juqueen [16] and the Juwels [17] supercomputers at the FZ (Research Center) Jülich, Germany. Juqueen is an IBM BlueGene/Q system consisting of 28,675 compute nodes, each with 16 IBM PowerPC-A2 cores at 1,6 GHz. Each compute node has 16GB RAM. Juqueen was in operation until May 2018. Juqueen's successor Juwels is a Bull Sequana X1000 system, at the time consisting of 2,271 compute nodes, each node with 96 GB RAM and two 24-core Intel Xeon SC 8168 CPUs running at 2,7 GHz. We use one MPI rank per core throughout.

5.1. Comparing the different ghost versions.

We begin by verifying that the additional complexity of implementing the top-down search that sets `Ghost_optimized` apart from `Ghost_tentative` is worth the effort. To this end, we use two meshes on a unit cube geometry, where the first consists of a single hexahedron tree and the second of six tethrahedron trees with a common diagonal as shown in Figure 3. For each mesh we run two types of tests: In the first type we create a uniform level ℓ mesh and compute the ghost layer for it. In the second type, we start with a uniform level ℓ mesh and refine recursively every third third element (in SFC order) up to level $\ell + k$. After this refinement, we repartition the mesh and create a ghost layer; see Figure 7.

We use 64 compute nodes of Juqueen and display our results in Table 1. As expected, the iterative version `Ghost_tentative` scales linearly with the number of elements. In contrast, `Ghost_optimized` scales with the number of ghost elements, which grows less quickly compared to the number of local elements. The improved version shows overall a significantly better performance and is up to a factor of 23.7 faster (adaptive tetrahedra, level 8) than the iterative version. For smaller or degraded meshes where the number of ghosts is on the same order as the number of leaf elements, the improved version shows no disadvantage compared to the iterative version. This underlines that we do not lose runtime to the `Search` overhead, even when practically each element is a partition boundary element. For small meshes both algorithms show negligible runtimes (on the order of milliseconds).

We conclude that our `Ghost_optimized` algorithm based on the top-down search is the ideal choice and will further analyze it in the experiments presented in the following. We will just call it `Ghost`.

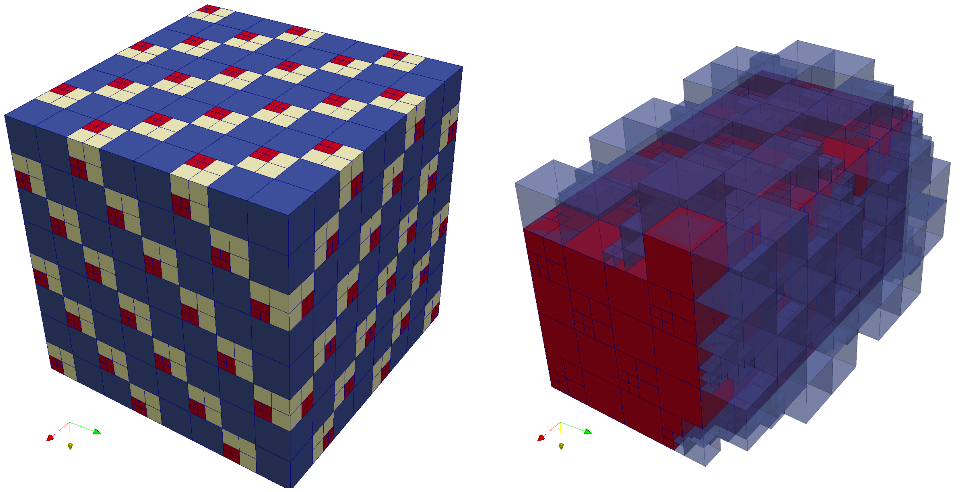


Fig. 7: We compare the different implementations of `Ghost` by testing them on a unit cube geometry with 1024 MPI ranks of Juqueen. Left: an adaptive mesh with minimum level $\ell = 3$ for one hexahedron tree. We refine every third element in SFC order and repeat the process a second time with the refined elements to reach level 5 (color by level). Right: for an illustrative example on 4 MPI ranks, we show the local leaf elements of the process with MPI rank 1 (red) and its ghost elements (blue).

tetrahedra						
	uniform			adaptive		
ℓ	9	8	4	8–10	7–9	3–5
elements/proc	786,432	98,304	24	1,015,808	126,976	31
ghosts/proc	32,704	8,160	30	31,604	8,137	56
`Ghost_tentative` [s]	129.6	16.19	5.93e-3	167.94	20.88	8.10e-3
`Ghost_optimized` [s]	7.41	1.75	5.01e-3	7.08	1.69	8.12e-3
hexahedra						
	uniform			adaptive		
ℓ	9	8	4	8–10	7–9	4–6
elements/proc	131,072	16,384	4	169,301	21,162	41
ghosts/proc	8,192	2,048	8	7,681	1,913	30
`Ghost_tentative` [s]	18.25	2.302	2.32e-3	23.79	2.964	8.01e-3
`Ghost_optimized` [s]	3.14	0.711	2.90e-3	2.81	0.649	8.12e-3

Table 1: Runtimes for the two different ghost algorithms on 1,024 MPI ranks of Juqueen. For tetrahedra and hexahedra we test a uniform level ℓ mesh and a mesh that adapts every third element of a uniform level ℓ mesh up to level $\ell+2$; cf. Figure 7. We observe that `Ghost_optimized` is superior to `Ghost_tentative` by a factor of up to 23 and scales with the number of ghost elements, not the number of leaves.

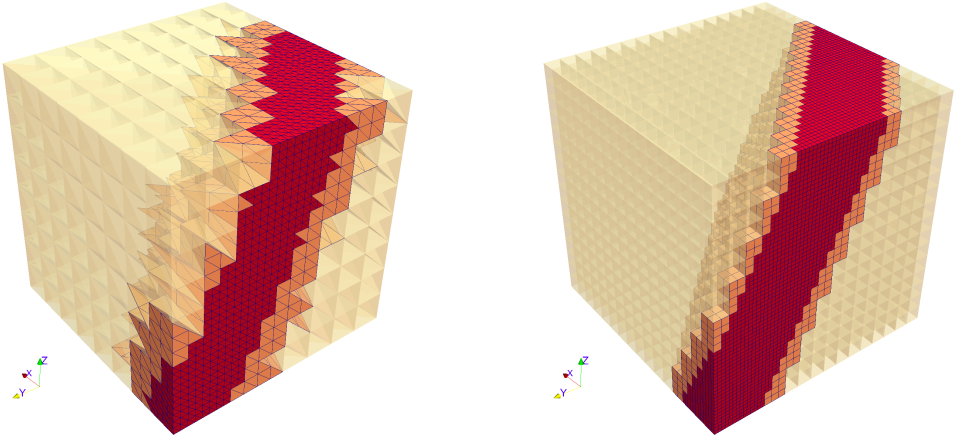


Fig. 8: On Juqueen, we test `Ghost` on a unit cube geometry consisting of six tetrahedral trees (left) or one hexahedral tree (right). Starting with a uniform level ℓ, we refine the forest in a band around a plane to level $\ell + k$. We then 2:1 balance the forest and create the ghost layer. In the next time step, the band moves in the direction of the plane's normal vector and we repeat the steps, coarsening previously fine forest elements if they now reside outside of the band. We show the forest after `Balance` and time step 2 for two different configurations. Left: tetrahedral elements with $\ell = 3$, $k = 2$. Right: hexahedral elements with $\ell = 4$, $k = 2$. We color by level.

5.2. A single-shape test case. In this test we use a setting similar to the tests in [6] for coarse mesh partitioning. We start with a uniform forest of level ℓ and refine it in a band parallel to a hyperplane to level $\ell + k$. We then establish a 2:1 balance among the elements (using a ripple propagation algorithm [34] not discussed here) and repartition the mesh using the `Partition` algorithm. Afterwards, we create a layer of ghost elements with `Ghost`. The interface moves through the domain in time in direction of the plane's normal vector. In each time step we adapt the mesh such that we coarsen elements outside of the band to level ℓ and refine within the band to level $\ell + k$. Then we repeat balance, partition, and `Ghost`. As opposed to the test in [6], we take the unit cube as our coarse mesh geometry. We run the test once with a hexahedral mesh consisting of one tree and once with a tetrahedral mesh of six trees forming a unit cube, similar to the previous section (see also Figure 8).

We choose the normal vector $\frac{3}{2}\begin{pmatrix}1, & 1, & \frac{1}{2}\end{pmatrix}^t$ and $\frac{1}{4}$ for the width of our refinement band. We move the refinement band with speed $v = \frac{1}{64}$ and scale the time step Δt with the refinement level as

$$\Delta t(\ell) = \frac{0.8}{2^\ell v}. \tag{8}$$

The constant 0.8 can be seen as width of the band of level ℓ elements that will be refined to level k in the next time step. We start the band at position $x_0(\ell) = 0.56 - 2.5\Delta t(\ell)$ and simulate up to 5 time steps. The strong and weak scaling results collected in the following are obtained with the `t8_time_forest_partition` example of `t8code` version 0.3 [14].

Tetrahedral case with $\ell = 8$, $k = 2$, $C = 0.8$ at $t = 4\Delta t$				
P	E/P	G/P	Time [s]	Par. Eff. [%]
8,192	234,178	17,946	3.25	100.0
16,384	117,089	11,311	2.12	96.6
32,768	58,545	7,184	1.27	102.4
65,536	29,272	4,560	0.79	104.5
131,072	14,636	2,859	0.52	99.5

Table 2: The results for strong scaling of `Ghost` with tetrahedral elements, $\ell = 8$, and $k = 2$. We show the runtimes at time $t = 4\Delta t$. The mesh consists of approximately 1.91e9 tetrahedra. In addition to the runtimes, we show the number of elements per process E/P, and ghosts per process G/P. The last column contains the parallel efficiency according to (9) in reference to the smallest run with 8,192 processes.

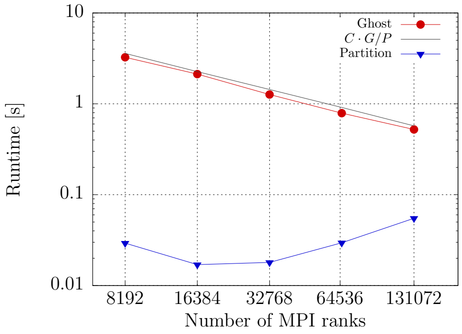


Fig. 9: Strong scaling with tetrahedral elements. We plot the runtimes of `Ghost` and `Partition` for the test case from Section 5.2 with $\ell = 8$, $k = 2$ at time step $t = 4\Delta t$. Ideally, `Ghost` scales with the number of ghost elements per process, G/P. This number is indicated by the black line.

5.2.1. Strong scaling.

We run a strong scaling test with tetrahedral elements and refinement parameters $\ell = 8$, $k = 2$ on 8,192 to 131,072 MPI ranks of Juqueen, increasing the process count by a factor of 2 in each step. We list the runtimes of `Ghost` at time $t = 4\Delta t$ in Table 2 and plot them together with those of `Partition` in Figure 9.

As we have already observed in Table 1, the runtime of `Ghost` depends linearly on the number of ghost elements per process. Consider two runs with P_1 and P_2 processes, respectively, and let G_1 and G_2 denote the numbers of ghost elements per process, then the parallel efficiency of the second run in relation to the first run is

$$e_{\text{Ghost}} = \frac{T_1 G_2}{T_2 G_1}. \tag{9}$$

Our results demonstrate that we achieve ideal strong scaling efficiency for `Ghost`.

5.2.2. Weak scaling.

For weak scaling we increase the global number of elements in proportion to the process count, keeping the local number of elements nearly

Tetrahedral case with $k = 2$, $C = 0.8$ at $t = 4\Delta t$					
P	ℓ	E/P	G/P	Time [s]	Par. Eff. [%]
8,192	8	234,178	17,946	3.25	100.0
65,536	9	233,512	18,282	3.76	88.2
458,752	10	266,494	20,252	3.79	96.8
2,048	7	117,630	10,999	1.99	100.0
16,384	8	117,089	11,311	2.12	96.5
131,072	9	116,756	11,478	2.18	95.2
Hexahedral case with $k = 2$, $C = 0.8$ at $t = 2\Delta t$					
P	ℓ	E/P	G/P	Time [s]	Par. Eff. [%]
8,192	9	309,877	34,600	6.79	100.0
65,536	10	310,163	35,136	6.85	100.7
458,752	11	354,746	38,833	7.86	96.9
2,048	8	156,178	21,536	4.18	100.0
16,384	9	155,702	22,036	4.25	100.6
131,072	10	155,460	22,284	4.36	98.9

Table 3: Weak scaling for `Ghost` with tetrahedral (top) and hexahedral elements (bottom). We increase the base level by one, keeping $k = 2$, while multiplying the process count by eight to maintain the same number of local elements per process. The sole exception is the highest process count of 458,752, seven times 65,536, resulting in $\approx 14\%$ more local elements. Similar to the strong scaling tests, the parallel efficiency is nearly ideal (see also Figure 10).

constant. Since with each refinement level ℓ the number of global elements grows by a factor of 8, we multiply the process count with 8 as well. We test the following configurations on Juqueen, again with $k = 2$:

- Tetrahedral elements with 8,192 processes, 65,536 processes, and 458,752 processes, with refinement levels $\ell = 8$, $\ell = 9$, $\ell = 10$. This amounts to about 235k elements per process. The largest run has about 108e9 elements.
- Tetrahedral elements with 2,048 processes, 16,384 processes, and 131,072 processes, with refinement levels $\ell = 8$, $\ell = 9$, $\ell = 10$. Here we have about 155k elements per process, summing up to 20.3e9 elements on 131,072 processes.
- Hexahedral elements with the same process counts and levels $\ell = 9$, $\ell = 10$ and $\ell = 11$ (162e9 elements in total).

Note that 458,752 is actually 7 times 65,536. We choose it since it is the maximum process count on Juqueen when assigning 16 processes per node, using all 28,672 compute nodes. The number of elements per process is thus about 14% greater than on the other process counts in the configuration. (9) still applies.

We show these results in Table 3 and Figure 10. We notice that `Ghost` for tetrahedra is faster than `Ghost` for hexahedra. The reason is the smaller number of ghosts due to less faces per element. In all tests we observe excellent strong and weak scaling with efficiencies on the order of 95%.

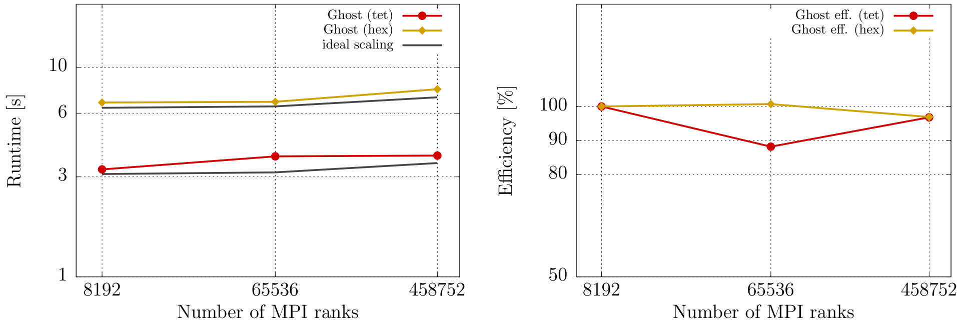


Fig. 10: Weak scaling results for tetrahedra with refinement levels 8, 9, and 10, and for hexahedra with refinement levels 9, 10, and 11, $k = 2$, on Juqueen. This amounts to 233k elements per process for tetrahedra and 310k elements per process for hexahedra. This number differs slightly for the 458,752 process runs, which is only seven times 65,536, while we increase the number of mesh elements by the factor 8. On the left-hand side we plot the runtimes of `Ghost` with the ideal scaling in black. On the right-hand side we plot the parallel efficiency in %. We list all values in Table 3.

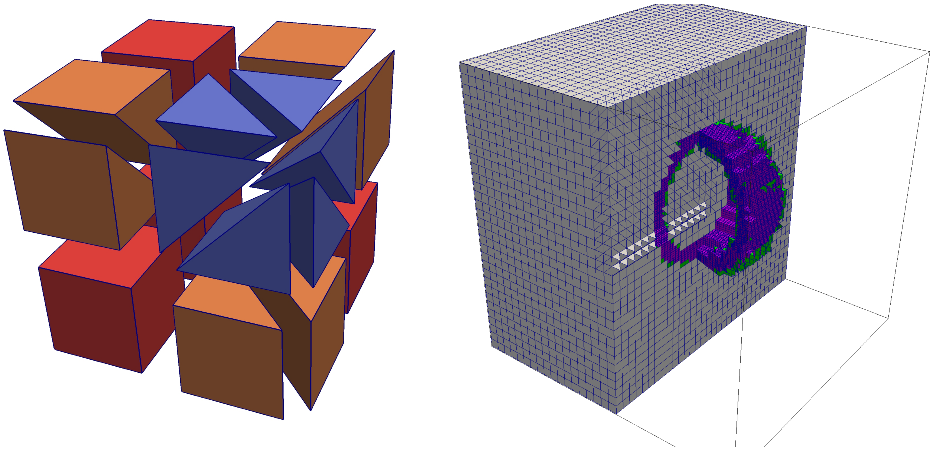


Fig. 11: For tests on Juwels we choose a hybrid cube mesh. Left: This coarse mesh consists of six tetrahedra, six prisms and four hexahedra. We rotate the trees such that every element in any refined mesh aligns at least two of its faces parallel to a coordinate plane. Right: We uniformly refine to a base level ℓ and then add $k = 2$ rounds of adaptive refinement near a spherical shell. The colors indicate different refinement levels; purple for level $\ell + 2$, green for level $\ell + 1$ and gray for level ℓ.

5.3. A hybrid test case. In our last test case we process hybrid meshes on the Juwels supercomputer. For the coarse mesh we model a cube with four hexahedra, six prisms and six tetrahedra (Figure 11). We refine this initial coarse mesh uniformly to level ℓ and then adaptively along a spherical shell to level $\ell + 2$. In this test series we do not enforce a 2:1 balance condition in the mesh, demonstrating the capability of our algorithm to handle unbalanced forests; cf. [15]. The results for weak and strong

P	ℓ	E/P	G/P	Time [s]	Par. Eff. [%]
192	8	1,601,702	70,467	0.384	–
1,536	9	1,601,702	74,586	0.363	112
12,288	10	1,601,702	76,514	0.355	117
98,304	11	1,601,702	77,454	0.369	114

Table 4: Weak scaling for the hybrid mesh with hexahedra, prisms and tetrahedra on Juwels; see also Figure 11. We increase the base level from $\ell = 8$ to $\ell = 11$ and the process count from 192 to 98,304. The largest mesh has approximately 157.5e9 elements. As before with non-hybrid meshes we observe excellent parallel efficiency.

P	ℓ	E/P	G/P	Time [s]	Par. Eff. [%]
12,288	11	12,813,617	305,290	1.428	–
24,576	11	6,406,808	194,919	0.912	99
49,152	11	3,203,404	121,987	0.550	103
98,304	11	1,601,702	77,454	0.369	97

Table 5: Strong scaling for the hybrid mesh with hexahedra, prisms and tetrahedra on Juwels; see also Figure 11. We fix the base level to $\ell = 11$ and increase the process count from 12,288 to 98,304. The mesh size is fixed at 157.5e9 elements. Absolute processing rates are above 200k ghost elements per second. We achieve nearly perfect parallel efficiency at sub-second full-system runtimes.

scaling are listed in Tables 4 and 5, respectively. Note that due to the relatively large amount of memory per node on Juwels we are able to create higher numbers of elements per process than on Juqueen, while the run time per element and process is much less.

As a final experiment we create a uniform level 12 mesh with more than 1e12 elements. Even for this exceptionally large mesh the runtime of our optimized `Ghost` algorithm is only 2.08 seconds (Table 6).

6. Conclusion.

In this paper, we present a parallel ghost layer assembly for adaptive forest meshes. It is general with respect to the shape of the trees, which may be cuboidal, simplicial, or of any other shape that yields a conforming coarse mesh. Furthermore, it is general in terms of non-conforming adaptivity by recursive refinement and works with and without a 2:1 balance property. The runtimes we show are proportional to the number of local ghost elements, which is the optimal rate to be expected, and our algortihm scales to over 1e12 elements total. Absolute runtimes are less than 5 μs per ghost element on the Xeon-based Juwels supercomputer.

We have limited the exposition to face-only connectivity, which suffices to implement flux- and mortar-based numerical methods, for example of finite volume, spectral element, or discontinuous Galerkin type. The extension of the algorithm to vertex and 3D edge connectivity is still open. Based on our experience with assembling the fully connected ghost layer for adaptive hexahedral meshes, we judge this extension to be feasible. In addition to implementing the necessary low-level functions to compute vertex and 3D edge neighbors within a tree, the major task will be

P	#Elements	`Ghost` [s]	`Partition` [s]
49,152	1,099,511,627,776	2.08	0.73

Table 6: The largest mesh that we create is uniform at level 12 and 1.1e12 elements.

to extend the tree-to-tree neighbor computation accordingly. To do so, the method that we describe in this paper, namely constructing the lower dimensional element, transforming it into its neighbor and then extruding it to the neighbor element, may be generalized to arbitrary codimension. The challenge compared to face-neighbors is that at a single inter-tree vertex/edge can connect to an arbitrary number of trees. Encoding and identifying these neighbor trees is a task that requires an appropriate extension of the coarse mesh connectivity data structure. Once this is accomplished, the neighbor elements can be constructed by following the techniques described in this paper.

The primary use for the presented algorithm is the support of element-based numerical methods, which can be realized in multiple ways. Since the ghost layer is an ordered linear structure, it can be binary searched directly by an application, for example to count and allocate the communication buffers for flux-based methods. Alternatively, it can be used in a combined local-and-ghost top-down traversal of the mesh to collect globally consistent face and node numbers for element-based methods and store them in a lookup table, which then serves as the interface to the numerical application. Other uses include CFL-limited particle tracking and semi-Lagrangian methods, which require quick owner search of points that leave the local partition.

In summary, we hope to have provided both an abstract technique and a usable software module that is indispensible to many numerical applications, and which hides the complexity of the algorithmic details behind a minimal interface.

Acknowledgements. The authors gratefully acknowledge the Gauß Centre for Supercomputing for funding the project chbn26 by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS supercomputers Juwels and Juqueen at Jülich Supercomputing Centre (JSC).

H. and B. gratefully acknowledge travel support by the Bonn Hausdorff Center for Mathematics (HCM) funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – GZ 2047/1, Project ID 390685813.

H. acknowledges additional financial support by the Bonn International Graduate School for Mathematics (BIGS) as part of HCM.

REFERENCES

[1] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, *Algorithms and data structures for massively parallel generic adaptive finite element codes*, ACM Transactions on Mathematical Software, 38 (2011), pp. 1–28, doi:10.1145/2049673.2049678.

[2] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.II – a general-purpose object-oriented finite element library*, ACM Transactions on Mathematical Software, 33 (2007), p. 24, doi:10.1145/1268776.1268779.

[3] J. Bey, *Der BPX-Vorkonditionierer in drei Dimensionen: Gitterverfeinerung, Parallelisierung und Simulation*, Universität Heidelberg, (1992). Preprint.

[4] C. Burstedde, *Parallel tree algorithms for AMR and non-standard data access*, ACM Transactions on Mathematical Software, 46 (2020), pp. 1–31, doi:10.1145/3401990.

[5] C. Burstedde and J. Holke, *A tetrahedral space-filling curve for nonconforming adaptive meshes*, SIAM Journal on Scientific Computing, 38 (2016), pp. C471–C503, doi:10.1137/15M1040049.

[6] C. Burstedde and J. Holke, *Coarse mesh partitioning for tree-based AMR*, SIAM Journal on Scientific Computing, 39 (2017), pp. C364–C392, doi:10.1137/16M1103518.

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

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

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

[10] A. Dedner, R. Klöfkorn, and M. Nolte, *The DUNE-ALUGrid module*, 2014, arXiv:1407.6954.

[11] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Computing in Science and Engineering, 4 (2002), pp. 90–97.

[12] A. Guittet, T. Isaac, C. Burstedde, and F. Gibou, *Direct numerical simulation of incompressible flows on parallel octree grids*. Manuscript, 2016.

[13] J. Holke, *Scalable algorithms for parallel tree-based adaptive mesh refinement with general element types*, PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2018, http://d-nb.info/1173789790/34.

[14] J. Holke and C. Burstedde, *`t8code`: Parallel AMR on hybrid non-conforming meshes*, 2015. http://github.com/holke/t8code, last accessed October 18th, 2019.

[15] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas, *Recursive algorithms for distributed forests of octrees*, SIAM Journal on Scientific Computing, 37 (2015), pp. C497–C531, doi:10.1137/140970963, arXiv:1406.0089.

[16] Jülich Supercomputing Centre, *JUQUEEN: IBM Blue Gene/Q supercomputer system at the Jülich Supercomputing Centre*, Journal of large-scale research facilities, 1 (2015), p. A1, doi:10.17815/jlsrf-1-18.

[17] Jülich Supercomputing Centre, *JUWELS: Modular tier-0/1 supercomputer at the Jülich Supercomputing Centre*, Journal of large-scale research facilities, 5 (2019), p. A135, doi:10.17815/jlsrf-5-171.

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

[19] D. Knapp, *Adaptive Verfeinerung von Prismen*, Bachelor's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2017.

[20] D. Knapp, *A space-filling curve for pyramidal adaptive mesh refinement*, Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2020.

[21] O. S. Lawlor, S. Chakravorty, T. L. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. V. Kalé, *ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications*, Engineering with Computers, 22 (2006), pp. 215–235, doi:10.1007/s00366-006-0039-5.

[22] Q. Liu, W. Zhao, J. Cheng, Z. Mo, A. Zhang, and J. Liu, *A programming framework for large scale numerical simulations on unstructured mesh*, in 2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS), April 2016, pp. 310–315, doi:10.1109/BigDataSecurity-HPSC-IDS.2016.12.

[23] G. M. Morton, *A computer oriented geodetic data base; and a new technique in file sequencing*, tech. report, IBM Ltd., 1966.

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

[25] 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*, Computing in Science Engineering, 16 (2014), pp. 13–21, doi:10.1109/MCSE.2014.75.

[26] C. Richardson and G. Wells, *High performance multi-physics simulations with FEniCS/DOLFIN*, Tech. Report eCSE03-10, CSE programme of the ARCHER UK National Supercomputing Service, 2016, doi:10.6084/M9.FIGSHARE.3406582.

[27] I. Sbalzarini, J. Walther, M. Bergdorf, S. Hieber, E. Kotsalis, and P. Koumoutsakos, *PPM—a highly efficient parallel particle-mesh library for the simulation of continuum systems*, Journal of Computational Physics, 215 (2006), pp. 566 – 588, doi:10.1016/j.jcp.2005.11.017.

[28] F. Schornbaum and U. Rüde, *Massively parallel algorithms for the lattice Boltzmann method on nonuniform grids*, SIAM Journal on Scientific Computing, 38 (2016), pp. 96–126.

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

[30] H. Sundar, R. Sampath, and G. Biros, *Bottom-up construction and 2:1 balance refinement of linear octrees in parallel*, SIAM Journal on Scientific Computing, 30 (2008), pp. 2675–2708, doi:10.1137/070681727.

[31] T. J. Tautges, J. A. Kraftcheck, N. Bertram, V. Sachdeva, and J. Magerlein, *Mesh interface resolution and ghost exchange in a parallel mesh representation*, in 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IEEE, 2012, pp. 1670–1679.

[32] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst, *MOAB: A mesh-oriented database*, SAND2004-1592, Sandia National Laboratories, Apr. 2004. Report.

[33] J. Teunissen and R. Keppens, *A geometric multigrid library for quadtree/octree AMR grids coupled to MPI-AMRVAC*, Computer Physics Communications, 245 (2019), p. 106866, doi:10.1016/j.cpc.2019.106866.

[34] T. Tu, D. R. O'Hallaron, and O. Ghattas, *Scalable parallel octree meshing for terascale applications*, in SC '05: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM/IEEE, 2005, pp. 4–4, doi:10.1109/SC.2005.61.

[35] T. Weinzierl and M. Mehl, *Peano—a traversal and storage scheme for octree-like adaptive Cartesian multiscale grids*, SIAM Journal on Scientific Computing, 33 (2011), pp. 2732–2760.

SUPPLEMENTARY MATERIAL

Appendix A. Low-level element functions in `t8code`. In this supplement we provide details for the implementation of the low-level algorithms needed for the face neighbor computation across tree boundaries that we discuss in Section 2 as well as the children-at-face computation that we need to identify owner processes. These are implementations for the classical Morton index for lines, quadrilaterals and hexahedra, and for the tetrahedral Morton index for triangles, tetrahedra and prisms.

For reference, we display our numbering convention of tree vertices and faces in Figure 12.

Algorithm A.2 defines our implementation of the face neighbor algorithm, which breaks the computation across tree boundaries into the following subalgorithms.

`t8_element_tree_face` (Section 2.1) computes the face number of a face of the tree given an element that shares a face with this tree. We provide a lookup table for the tetrahedron version in Table 7. The implementations of the other element shapes follow from the discussion in Section 2.1.

`t8_element_boundary_face` (Section 2.2) computes the $(d-1)$-dimensional face element corresponding to a given face of a d-dimensional element. The computation breaks down into projecting the anchor node coordinates of the element onto the correct $(d-1)$-dimensional plane. See Tables 8 and 9 for triangles, quads, hexahedra and tetrahedra. The implementation for lines is straightforward, and the implementation for prisms follows directly as a cross product of triangles and lines.

`t8_element_transform_face` (Section 2.3) transforms the coordinates of a $(d-1)$-dimensional face element from one reference tree into another. As input it requires the face as well as the orientations and sign of the tree-to-tree connection. In Remark 15 we argue that we only need to implement all cases with $s = 1$ and the additional case $o = 0, s = -1$. We display these implementations in Tables 10 and 11. The sign s can be determined by Table 12.

`t8_element_extrude_face` (Section 2.4) builds the d-dimensional element in the neighbor tree from the transformed face element. To this end, the anchor node of the new element is computed from the anchor node of the face element and the information on which tree face this element lies. We provide the details in Table 13.

Combining the above four subalgorithms, we may compute the face neighbor of an element across tree boundaries, which is a decisive element of the ghost layer assembly for multi-tree hybrid-shape meshes.

As a final addition to this supplement, we show the lookup tables for the implementation of the `t8_element_children_at_face` algorithm in Table 14. This algorithm is needed to identify owner processes of neighbor elements (Section 2.5).

Functional implementations of all proposed algorithms are available from the public `t8code` repository [14].

Algorithm A.2: Computing the face neighbor of an element

```
Result: The same-level face-neighbor $E'$ of $E$ across face $f$
if element_neighbor_inside_root ($E$, $f$) then
    $E' \leftarrow$ t8_element_face_neighbor_inside ($E$, $f$)
else
    $g \leftarrow$ t8_element_tree_face ($E$, $f$)      /* (i) Tree face no. and */
    $o \leftarrow$ face_orientation ($\mathscr{F}$, $K$, $g$)      /* relative orientation */
    $F \leftarrow$ t8_element_boundary_face ($E$, $f$)      /* (ii) Face element */
    $F' \leftarrow$ t8_element_transform_face ($F$, $o$)   /* (iii) Neighb. fc. */
    $g' \leftarrow$ tree_neighbor_face ($\mathscr{F}$, $K$, $g$)     /* Neighbor tree face */
    $E' \leftarrow$ t8_element_extrude_face ($F'$, $g'$)     /* (iv) and element */
```

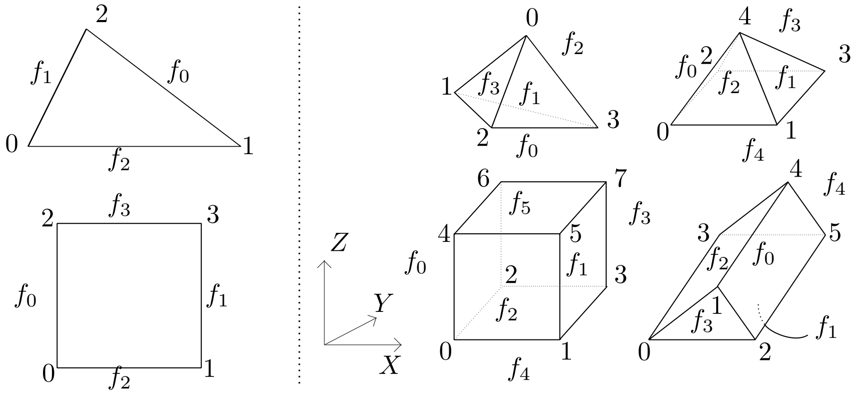


Fig. 12: The vertex and face labels of the 2D (left) and 3D (right) tree shapes. While the low-level functions for the pyramid are not yet released in `t8code`, this additional element shape is covered by our new high-level algorithms exactly like all others.

Tetrahedron					
type(T)	f	g	type(T)	f	g
0	i	i	3	–	–
1	0	0	4	1	2
2	2	1	5	3	3

Table 7: $g =$ `t8_element_tree_face` (T, f) for a tetrahedron T and a face f of T that lies on a tree face. Depending on T's type, all, exactly one, or none of its faces can be a subface of a face of the root tetrahedron. We show the tetrahedron's face number f and the corresponding face number g in the root tetrahedron.

Quadrilateral	
f	$F.x$
0, 1	$Q.y$
2, 3	$Q.x$

Hexahedron			
f	$(F.x, F.y)$	f	$(F.x, F.y)$
0	$(Q.y, Q.z)$	3	$(Q.x, Q.z)$
1	$(Q.y, Q.z)$	4	$(Q.x, Q.y)$
2	$(Q.x, Q.z)$	5	$(Q.x, Q.y)$

Table 8: `t8_element_boundary_face` for quadrilaterals and hexahedra. Left: For a quadrilateral Q with anchor node $(Q.x, Q.y)$ and a face f, the corresponding anchor node coordinate $F.x$ of the face line element. Right: For a hexahedron Q with anchor node $(Q.x, Q.y, Q.z)$ and a face f, the corresponding anchor node coordinates $(F.x, F.y)$ of the face quadrilateral element. In either case, computing the coordinates is equivalent to a projection.

Triangle		
type (T)	f	$F.x$
0	0	$T.y$
	1	$T.x$
	2	$T.x$

Tetrahedron				
type (T)	f	case	type (F)	$(F.x, F.y)$
0	0	1	0	$(T.z, T.y)$
	1	1	0	$(T.z, T.y)$
	2	2	0	$(T.x, T.z)$
	3	2	0	$(T.x, T.z)$
1	0	1	1	$(T.z, T.y)$
2	2	1	1	$(T.z, T.y)$
3	–	–	–	–
4	1	2	1	$(T.x, T.z)$
5	3	2	1	$(T.x, T.z)$

Table 9: `t8_element_boundary_face` $(T,\ f)$ for triangles and tetrahedra. Left: The x coordinate of the anchor node of the boundary line F at face f of a triangle T in terms of T's coordinates. Right: Two cases occur, which we list together with the type of the boundary triangle F at a face f of tetrahedron T and the anchor node coordinates $(F.x, F.y)$.

Triangle		Quadrilateral
type(F)	$\begin{pmatrix} F'.x \\ F'.y \end{pmatrix}$	$\begin{pmatrix} F'.x \\ F'.y \end{pmatrix}$
0	$\begin{pmatrix} F.x \\ F.x - F.y \end{pmatrix}$	$\begin{pmatrix} F.y \\ F.x \end{pmatrix}$
1	$\begin{pmatrix} F.x \\ F.x - F.y - h \end{pmatrix}$	

Table 10: Result of `t8_transform_face (`F`, `$o = 0$`, `$s = -1$`)` for triangles (left) and quadrilaterals (right). We compute any arbitrary combination of values for o with $s = -1$ by first applying `t8_transform_face (`F`, 0, `-1`)` and then `t8_transform_face (`F`, `o`, 1)` from Table 11.

Line	
o	$\begin{pmatrix} F'.x \end{pmatrix}$
0	$\begin{pmatrix} F.x \end{pmatrix}$
1	$\begin{pmatrix} 2^{\mathcal{L}} - F.x - h \end{pmatrix}$

Quadrilateral			
o	$\begin{pmatrix} F'.x \\ F'.y \end{pmatrix}$	o	$\begin{pmatrix} F'.x \\ F'.y \end{pmatrix}$
0	$\begin{pmatrix} F.x \\ F.y \end{pmatrix}$	2	$\begin{pmatrix} F.y \\ 2^{\mathcal{L}} - F.x - h \end{pmatrix}$
1	$\begin{pmatrix} 2^{\mathcal{L}} - F.y - h \\ F.x \end{pmatrix}$	3	$\begin{pmatrix} 2^{\mathcal{L}} - F.x - h \\ 2^{\mathcal{L}} - F.y - h \end{pmatrix}$

Triangle					
type(F)	o	$\begin{pmatrix} F'.x \\ F'.y \end{pmatrix}$	type(F)	o	$\begin{pmatrix} F'.x \\ F'.y \end{pmatrix}$
0	0	$\begin{pmatrix} F.x \\ F.y \end{pmatrix}$	1	0	$\begin{pmatrix} F.x \\ F.y \end{pmatrix}$
	1	$\begin{pmatrix} 2^{\mathcal{L}} - F.y - h \\ F.x - F.y \end{pmatrix}$		1	$\begin{pmatrix} 2^{\mathcal{L}} - F.y - h \\ F.x - F.y - h \end{pmatrix}$
	2	$\begin{pmatrix} 2^{\mathcal{L}} - F.x + F.y - h \\ 2^{\mathcal{L}} - F.x - h \end{pmatrix}$		2	$\begin{pmatrix} 2^{\mathcal{L}} - F.x + F.y \\ 2^{\mathcal{L}} - F.x - h \end{pmatrix}$

Table 11: Result of `t8_transform_face (`F`, `o`, `$s = 1$`)` for lines (top left), quadrilaterals (top right) and triangles (bottom) with sign 1. For values with $s = -1$ see Table 10 and Remark 15.

K and K' tetrahedra				
	g			
g'	0	1	2	3
0	-1	1	-1	1
1	1	-1	1	-1
2	-1	1	-1	1
3	1	-1	1	-1

K hexahedron, K' prism						
	g					
g'	0	1	2	3	4	5
0	1	-1	-1	1	1	-1
1	-1	1	1	-1	-1	1
2	1	-1	-1	1	1	-1

K tetrahedron, K' prism				
	g			
g'	0	1	2	3
3	-1	1	-1	1
4	1	-1	1	-1

K and K' prisms					
	g				
g'	0	1	2	3	4
0	-1	1	-1	–	–
1	1	-1	1	–	–
2	-1	1	-1	–	–
3	–	–	–	-1	1
4	–	–	–	1	-1

Table 12: Value of $\mathrm{sign}_{t,t'}(g, g')$ from Definition 13 for four possible tree-to-tree connections. We obtain these values from Figure 12. For two hexahedra, we refer to (6).

2D – coordinates			
g'	$\begin{pmatrix}E'.x\\E'.y\end{pmatrix}$	g'	$\begin{pmatrix}E'.x\\E'.y\end{pmatrix}$
Quadrilateral from line			
0	$\begin{pmatrix}0\\F'.x\end{pmatrix}$	2	$\begin{pmatrix}F'.x\\0\end{pmatrix}$
1	$\begin{pmatrix}2^{\mathcal{L}}-h\\F'.x\end{pmatrix}$	3	$\begin{pmatrix}F'.x\\2^{\mathcal{L}}-h\end{pmatrix}$
Triangle from line			
0	$\begin{pmatrix}2^{\mathcal{L}}-h\\F'.x\end{pmatrix}$	2	$\begin{pmatrix}F'.x\\0\end{pmatrix}$
1	$\begin{pmatrix}F'.x\\F'.x\end{pmatrix}$		

3D – types		
g'	type(F')	type(E')
Tetrahedron from triangle		
0	0	0
	1	1
1	0	0
	1	2
2	0	0
	1	4
3	0	0
	1	5
Prism from triangle or quad		
0	–	0
1	–	0
2	–	0
3	0	0
	1	1
4	0	0
	1	1

3D - coordinates			
g'	$\begin{pmatrix}E'.x\\E'.y\\E'.z\end{pmatrix}$	g'	$\begin{pmatrix}E'.x\\E'.y\\E'.z\end{pmatrix}$
Hexahedron from quad			
0	$\begin{pmatrix}0\\F'.x\\F'.y\end{pmatrix}$	3	$\begin{pmatrix}F'.x\\2^{\mathcal{L}}-h\\F'.y\end{pmatrix}$
1	$\begin{pmatrix}2^{\mathcal{L}}-h\\F'.x\\F'.y\end{pmatrix}$	4	$\begin{pmatrix}F'.x\\F'.y\\0\end{pmatrix}$
2	$\begin{pmatrix}F'.x\\0\\F'.y\end{pmatrix}$	5	$\begin{pmatrix}F'.x\\F'.y\\2^{\mathcal{L}}-h\end{pmatrix}$
Tetrahedron from triangle			
0	$\begin{pmatrix}2^{\mathcal{L}}-h\\F'.y\\F'.x\end{pmatrix}$	2	$\begin{pmatrix}F'.x\\0\\F'.y\end{pmatrix}$
1	$\begin{pmatrix}F'.x\\F'.y\\F'.x\end{pmatrix}$	3	$\begin{pmatrix}F'.x\\0\\F'.y\end{pmatrix}$
Prism from triangle or quad			
0	$\begin{pmatrix}2^{\mathcal{L}}-h\\F'.x\\F'.y\end{pmatrix}$	3	$\begin{pmatrix}F'.x\\F'.y\\0\end{pmatrix}$
1	$\begin{pmatrix}F'.x\\F'.x\\F'.y\end{pmatrix}$	4	$\begin{pmatrix}F'.x\\F'.y\\2^{\mathcal{L}}-h\end{pmatrix}$
2	$\begin{pmatrix}F'.x\\0\\F'.y\end{pmatrix}$		

Table 13: The computation of E' = `t8_element_extrude_face (`F'`, `T'`, `g'`)`. Depending on the anchor node coordinates of F' and the tree face number g' we determine the anchor node of the extruded element E'. For tetrahedra and prisms we additionally need to compute the type of E', which depends on g' and the type of the triangle F' (bottom left). In the case of a triangle the type of E' is always 0, since type 1 triangles cannot lie on a tree boundary. Hence, we do not show a table for this case. h refers to the length of the element E' (resp. F') and is computed as $2^{\mathcal{L}-\ell}$, where ℓ is the refinement level of E' and F'.

Triangle			
	f		
type(T)	0	1	2
0	1,3	0,3	0,1
1	2,3	0,3	0,2

Tetrahedron				
	f			
type(T)	0	1	2	3
0	1, 4, 5, 7	0, 4, 6, 7	0, 1, 2, 7	0, 1, 3, 4
1	1, 4, 5, 7	0, 5, 6, 7	0, 1, 3, 7	0, 1, 2, 5
2	3, 4, 5, 7	0, 4, 6, 7	0, 1, 3, 7	0, 2, 3, 4
3	1, 5, 6, 7	0, 4, 6, 7	0, 1, 3, 7	0, 1, 2, 6
4	3, 5, 6, 7	0, 4, 5, 7	0, 1, 3, 7	0, 2, 3, 5
5	3, 5, 6, 7	0, 4, 6, 7	0, 2, 3, 7	0, 1, 3, 6

Prism					
	f				
type(P)	0	1	2	3	4
0	1, 3, 5, 7	0, 3, 4, 7	0, 1, 4, 5	0, 1, 2, 3	4, 5, 6, 7
1	2, 3, 6, 7	0, 3, 4, 7	0, 2, 4, 6	0, 1, 2, 3	4, 5, 6, 7

Table 14: The child indices of all children of an element touching a given face for triangles, tetrahedra and prisms. These indices are needed for `t8_element_children_at_face`.