SIAM J. SCI. COMPUT.
Vol. 38, No. 5, pp. C471–C503

A TETRAHEDRAL SPACE-FILLING CURVE FOR NONCONFORMING ADAPTIVE MESHES*

CARSTEN BURSTEDDE† AND JOHANNES HOLKE‡

Abstract. We introduce a space-filling curve for triangular and tetrahedral red-refinement that can be computed using bitwise interleaving operations similar to the well-known Z-order or Morton curve for cubical meshes. To store sufficient information for random access, we define a low-memory encoding using 10 bytes per triangle and 14 bytes per tetrahedron. We present algorithms that compute the parent, children, and face-neighbors of a mesh element in constant time, as well as the next and previous element in the space-filling curve and whether a given element is on the boundary of the root simplex or not. Our presentation concludes with a scalability demonstration that creates and adapts selected meshes on a large distributed-memory system.

Key words. forest of octrees, parallel adaptive mesh refinement, Morton code, high performance computing, nonconforming simplicial mesh, space-filling curve

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

DOI. 10.1137/15M1040049

1. Introduction.

Conforming adaptive mesh refinement for simplicial (triangular and tetrahedral) meshes is one of the most successful concepts in numerical mathematics and computational science and engineering; see, e.g., [3, 21, 50]. Simplices provide high flexibility in meshing to arbitrary domain geometries [46, 47] and can be mapped to a reference simplex using an elementwise constant Jacobian in most cases, which allows for an efficient numerical implementation. Discretization and integration methods of various kinds are available for both low and high orders of accuracy.

Large-scale scientific computing requires fast and scalable algorithms for (1) adaptive refinement and coarsening (AMR) as well as (2) parallel partitioning. One class of methods for AMR exploits the properties of Delaunay triangulations [23, 45, 26], while partitioning of unstructured meshes is often approached by formulating the mesh topology as a graph. These triangulations are usually conforming; that is, elements intersect only along whole faces and edges. Common application codes such as FEniCS [33], PLUM [35, 36], OpenFOAM [37], or MOAB from the SIGMA toolkit [53] delegate graph partitioning to third-party software like ParMETIS [29] or Scotch [20]. Graph-based algorithms have been advanced to target millions of processes and billions of elements [18, 42, 49]. Still, increasing the scalability and decreasing the absolute runtime and memory demands of distributed implementations remains a

*Submitted to the journal's Software and High-Performance Computing section September 15, 2015; accepted for publication (in revised form) June 21, 2016; published electronically September 7, 2016.

http://www.siam.org/journals/sisc/38-5/M104004.html

Funding: This work was supported by the Bonn Hausdorff Center for Mathematics (HCM) funded by the Deutsche Forschungsgemeinschaft (DFG). The second author was also supported by the Bonn International Graduate School for Mathematics (BIGS) as part of HCM. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under contract DE-AC02-06CH11357.

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

challenge, and the lack of an obvious parent-child structure in many unstructured meshing approaches prevents certain use cases.

Nonconforming AMR in combination with recursive refinement makes refinement and coarsening nearly trivial operations. The additional mathematical logic to enable hanging faces and edges is well understood for both continuous and discontinuous discretizations [43, 22, 30, 1]. It is local to the loop over the finite elements or volumes and transparent to most of the numerical pipeline, thus offering the possibility to extend existing conforming codes. The resolution may be as coarse as any chosen root mesh (a mesh which is not intended for further coarsening), which poses only a slight limit to geometric flexibility.

The challenge of efficient partitioning of meshes may be addressed using space-filling curves (SFCs). Instead of working on the NP-hard graph partitioning problem [12], SFCs are used to approximately solve the partitioning problem in linear runtime [4, 57]. Common SFCs establish an equivalence between the adaptive mesh and a tree, where the mesh elements correspond one-to-one to the leaves of the tree. They also define a total order among the leaves that can be used to store application data linearly in the same order as the elements [15]. The original concept is specified for a cubic domain and can be traced back to over a hundred years ago [39, 25]. The Sierpinski curve [48] and generalized versions of it are the most common SFCs on simplicial meshes. These were described by Bänsch [8], Kossaczkỳ [31], and others; see, e.g., [44, 4]. The computation of parents and children, for example, usually involves a loop over the refinement levels, equivalent to following the path between the tree root and a leaf.

When using hypercubes as mesh primitives, the logic of common SFCs can be formulated with remarkable simplicity due to the local tensor product structure [39, 51, 54, 52, 57]. It is also well-suited to compute topological entities like face-neighbors, children, and parents of given mesh elements. The Morton curve [34], in particular, allows one to design algorithms for these tasks (see, e.g., [16]), whose runtime is level-independent, i.e., does not depend on the depth of the refinement tree. Moreover, SFCs are memory efficient, since for a mesh element only the coordinates of one anchor node plus its refinement level have to be stored. Today, this principle has found its way into widely used software libraries [6, 5]. Memory usage can be further optimized by incremental encoding along the SFC [13, 55, 4].

Using SFCs on hexahedral meshes is exceptionally fast and scalable [41, 14, 28]. This fact has not only been exploited in writing simulation codes using hexahedral meshes, but also by approaches that use the hexahedral SFC as an instrument to partition simplices, mapping them into a surrounding cube [2]. However, hexahedral AMR is more limited geometrically than simplicial AMR. One indication for this is the (lack of) availability of (open-source) mesh generators that operate exclusively on cubes. Furthermore, assuming an existing simplicial code, rewriting it in terms of cubical meshes will be out of the question if the code is of sufficient size, complexity, or maturity. For these reasons, in this paper, we attempt to establish a new SFC for triangles and tetrahedra that has many of the favorable properties known for hexahedra.

Our starting point is to divide the simplices in a refined mesh into two (two dimensions, 2D), respectively, six (three dimensions, 3D), different types and selecting for each type a specific reordering for Bey's red-refinement [9]. This type and the coordinates of one vertex serve as a unique identifier, the Tet-id, of the simplex in question. In particular, we do not require storing the type of all parent simplices to the root, as one might naively imagine. We then propose a Morton-like coordinate

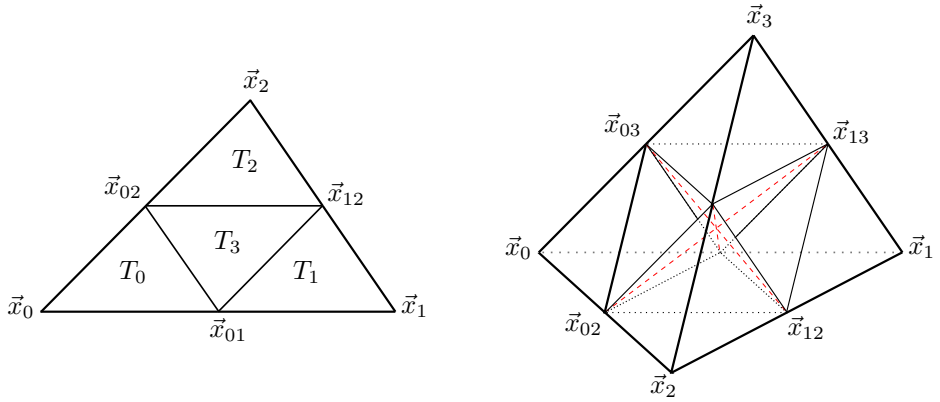


FIG. 1. *Left: The refinement scheme for triangles in two dimensions. A triangle* $T = [\vec{x}_0, \vec{x}_1, \vec{x}_2] \subset \mathbb{R}^2$ *is refined by dividing each face at the midpoints* $\vec{x}_{ij}$*. We obtain four smaller triangles, all similar to* T*. Right: The situation in three dimensions. If we divide the edges of the tetrahedron* $T = [\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3] \subset \mathbb{R}^3$ *in half, we get four smaller tetrahedra (similar to* T*) and an inner octahedron. By dividing the octahedron along any of its three diagonals (shown dashed) we finally end up with partitioning* T *into eight smaller tetrahedra, all having the same volume. The refinement rule of Bey is obtained by always choosing the diagonal from* $\vec{x}_{02}$ *to* $\vec{x}_{13}$ *and numbering the corners of the children according to* (2).

mapping that can be computed from the Tet-id and gives rise to an SFC. Based on this logic, we develop constant-time algorithms (that is, not requiring a loop over the refinement levels) to compute the tetrahedral-Morton identifiers of any parent, child, face-neighbor, and SFC-successor/predecessor and to decide whether for two given elements one is an ancestor of the other. We conclude with scalability tests of mesh creation and adaptation with over 8.5×10^{11} tetrahedral mesh elements on up to 131,072 cores of the JUQUEEN supercomputer and 786,432 cores of MIRA.

2. Mesh refinement on simplices. Our aim is to define and examine a new SFC for triangles and tetrahedra by adding ordering prescriptions to the nonconforming Bey-refinement (also called red-refinement) [9, 10, 56]. We briefly restate the red-refinement in this section and contrast it with the well-known conforming (or red/green) refinement.

We refer to triangles and tetrahedra as d-simplices, where $d \in \{2, 3\}$ specifies the dimension. It is sometimes convenient to drop d from this notation. A d-simplex $T \subseteq \mathbb{R}^d$ is uniquely determined by its $d+1$ affine-independent corner nodes $\vec{x}_0, \ldots, \vec{x}_d \in \mathbb{R}^d$. Their order is significant, and therefore we write

$$T = [\vec{x}_0, \vec{x}_1, \vec{x}_2] \quad \text{in 2D}, \tag{1a}$$

$$T = [\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3] \quad \text{in 3D}. \tag{1b}$$

We define $\vec{x}_0$ as the *anchor node* of T. By $\vec{x}_{ij}$ we denote the midpoint between $\vec{x}_i$ and $\vec{x}_j$, and thus $\vec{x}_{ij} = \frac{1}{2}(\vec{x}_i + \vec{x}_j)$.

2.1. Bey's refinement rule. Bey's rule is a prescription for subdividing a simplex. It is one instance of the so-called red-refinement, where all faces of a simplex are subdivided simultaneously.

DEFINITION 1. *Given a* d*-simplex* $T = [\vec{x}_0, \ldots, \vec{x}_d] \subset \mathbb{R}^d$*, the refinement rule of Bey consists of cutting off four subsimplices at the corners (as in Figure* 1*). In* 3*D*

the remaining octahedron is then divided along the diagonal from $\vec{x}_{02}$ *to* $\vec{x}_{13}$*. Bey numbers the* 2^d *resulting subsimplices as follows:*

$$2D: \quad \begin{array}{ll} T_0 := [\vec{x}_0, \vec{x}_{01}, \vec{x}_{02}], & T_1 := [\vec{x}_{01}, \vec{x}_1, \vec{x}_{12}], \\ T_2 := [\vec{x}_{02}, \vec{x}_{12}, \vec{x}_2], & T_3 := [\vec{x}_{01}, \vec{x}_{02}, \vec{x}_{12}], \end{array} \tag{2a}$$

$$3D: \quad \begin{array}{ll} T_0 := [\vec{x}_0, \vec{x}_{01}, \vec{x}_{02}, \vec{x}_{03}], & T_4 := [\vec{x}_{01}, \vec{x}_{02}, \vec{x}_{03}, \vec{x}_{13}], \\ T_1 := [\vec{x}_{01}, \vec{x}_1, \vec{x}_{12}, \vec{x}_{13}], & T_5 := [\vec{x}_{01}, \vec{x}_{02}, \vec{x}_{12}, \vec{x}_{13}], \\ T_2 := [\vec{x}_{02}, \vec{x}_{12}, \vec{x}_2, \vec{x}_{23}], & T_6 := [\vec{x}_{02}, \vec{x}_{03}, \vec{x}_{13}, \vec{x}_{23}], \\ T_3 := [\vec{x}_{03}, \vec{x}_{13}, \vec{x}_{23}, \vec{x}_3], & T_7 := [\vec{x}_{02}, \vec{x}_{12}, \vec{x}_{13}, \vec{x}_{23}]. \end{array} \tag{2b}$$

DEFINITION 2. *In this paper, a* refinement *of a d-simplex S denotes a set* $\mathscr{S}$ *of d-dimensional subsimplices of S that can be constructed from S via successive refinement, where only the finest simplices belong to the actual refinement. Thus all refinements can be obtained by applying the following rules:*

- $\mathscr{S} = \{ S \}$ *is a refinement of S.*
- *If* $\mathscr{S}'$ *is a refinement of S, then* $\mathscr{S} = (\mathscr{S}' \setminus \{ T \}) \cup \{ T_0, \ldots, T_{2^d-1} \}$ *is a refinement for every* $T \in \mathscr{S}'$*.*

We explicitly allow nonuniform refinements and thus nonconforming faces and edges.

DEFINITION 3. *The* T_i *from* (2) *are called the* children *of T, and T is called the* parent *of the* T_i*, written* $T = P(T_i)$*. Therefore, we also call the* T_i siblings *of each other. If a d-simplex T belongs to a refinement of another d-simplex S, then T is a* descendant *of S, and S is an* ancestor *of T. The number* ℓ *of refining steps needed to obtain T from S is unique and called the* level *of T (with respect to S); we write* $\ell = \ell(T)$*. Usually S is clear from the context, and therefore we omit it in the notation. By definition, T is an ancestor and descendant of itself.*

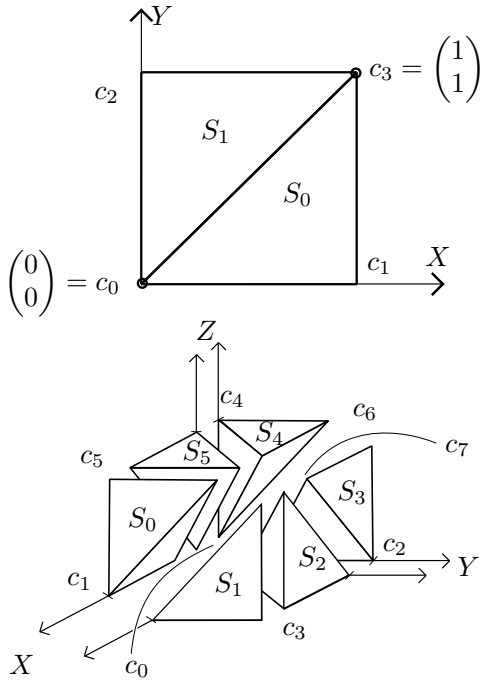
Consider the six tetrahedra $S_0, \ldots, S_5 \subset \mathbb{R}^3$ displayed in Figure 2. These tetrahedra form a triangulation of the unit cube. The results and algorithms in this paper rely on the following property [9].

PROPERTY 4. *Refining the six tetrahedra from the triangulation of the unit cube simultaneously to level* ℓ *results in the same mesh as first refining the unit cube to level* ℓ *and then triangulating each smaller cube with the six tetrahedra* $S_0, \ldots, S_5$*, scaled by a factor of* $2^{-\ell}$ *(see Figure 3). The same behavior can be observed in 2D when the unit square is divided into two triangles.*

2.2. Removal of hanging nodes using red/green refinement. It is worth noting that, although the methods and algorithms presented in this paper apply to red-refined meshes with hanging nodes, it is possible to augment them to create meshes without hanging nodes. For this we may use red/green or red/green/blue refinement methods [7, 17].

After the red-refinement step we may add an additional and possibly nonlocal refinement operation that ensures a maximum level difference of 1 between neighboring simplices. Such an operation is also called 2:1 balance [27, 52, 54]; its full description, however, would exceed the scope of the present paper. Hanging nodes are then resolved by bisecting those simplices with hanging nodes (green/blue refinement) [4, section 12.1.3]. The 2D case is shown in Figure 4.

If one of the newly created simplices shall be further refined, the bisection is reversed, the original simplex is red-refined, and the balancing and green-refinement is repeated. This may void the nesting property of certain discrete function spaces,



(3a)

2D	$\vec{x}_0$	$\vec{x}_1$	$\vec{x}_2$
S_0	c_0	c_1	c_3
S_1	c_0	c_2	c_3

(3b)

3D	$\vec{x}_0$	$\vec{x}_1$	$\vec{x}_2$	$\vec{x}_3$
S_0	c_0	c_1	c_5	c_7
S_1	c_0	c_1	c_3	c_7
S_2	c_0	c_2	c_3	c_7
S_3	c_0	c_2	c_6	c_7
S_4	c_0	c_4	c_6	c_7
S_5	c_0	c_4	c_5	c_7

FIG. 2. *The basic triangle (2D) and tetrahedra types (3D) obtained by dividing* $[0,1]^d$ *into simplices of varying types, denoted by a subscript. Top left: The unit square can be divided into two triangles sharing the edge from* $(0,0)^T$ *to* $(1,1)^T$*. We denote these triangles by* S_0 *and* S_1*. The four corners of the square are numbered* $c_0,\ldots,c_3$ *in yx-order. Top right: The corner nodes of* S_0 *and* S_1 *in terms of the square corners. Bottom left (exploded view): In 3D the unit cube can be divided into six tetrahedra, all sharing the edge from the origin to* $(1,1,1)^T$*. We denote these tetrahedra by* $S_0,\ldots,S_5$*. The eight corners of the cube are numbered* $c_0,\ldots,c_7$ *in zyx-order (redrawn and modified with permission* [9]*). Bottom right: The corner nodes of the six tetrahedra* $S_0,\ldots,S_5$ *in terms of the cube corners.*

yet applications may still prefer this approach over the manual implementation of hanging node constraints.

3. A tetrahedral Morton index. The Morton index or Z-order for a cube in a hexahedral mesh is computed by bitwise interleaving the coordinates of the anchor node of the cube [34]. In this section we present an index for d-simplices that also uses the bitwise interleaving approach, the tetrahedral Morton index (TM-index). To define the TM-index we look at refinements of a reference simplex, which we discuss in section 3.1 below. For each d-simplex in a refinement of the reference simplex we define a unique identifier, the so-called Tet-id, which serves as the input to compute the TM-index and for all algorithms related to it. This Tet-id consists of the coordinates of the anchor node of the considered simplex plus one additional number, the type of the simplex. We define the Tet-id and type in section 3.2. We then define the TM-index in section 3.3 and in the following subsections. We show that it possesses locality properties similar to those of the Morton index. One novel aspect of this construction lies in logically including the types of the simplex and all its parents in the interleaving, while only using the type of the simplex itself in the algorithms.

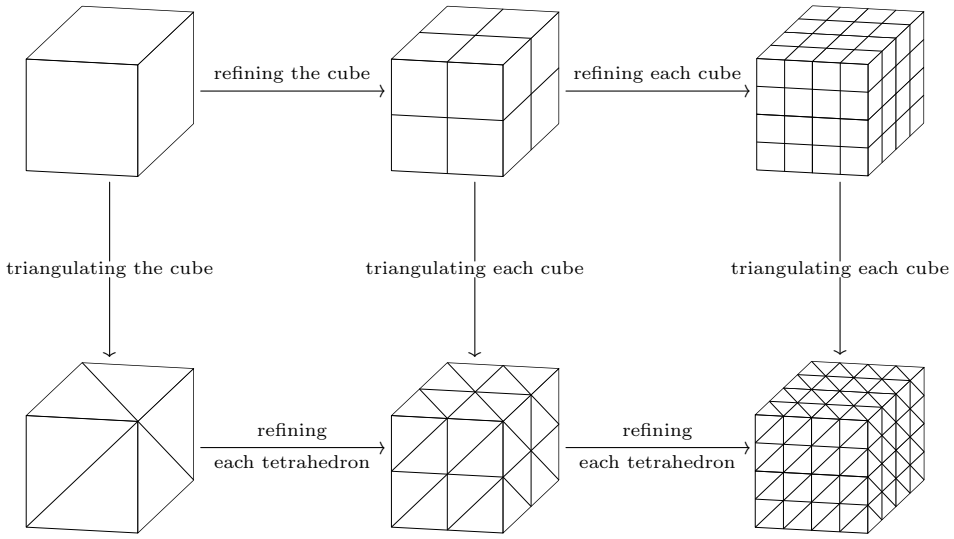


FIG. 3. *Triangulating a cube according to Figure* 2 *and then refining the tetrahedra via Bey's refinement rule results in the same mesh as first refining the cube into eight subcubes and afterward triangulating each of these cubes. Each occurring tetrahedron is uniquely determined by the subcube it lies in plus its type. The same situation can be observed in* 2*D if we restrict our view to one side of the cube.*

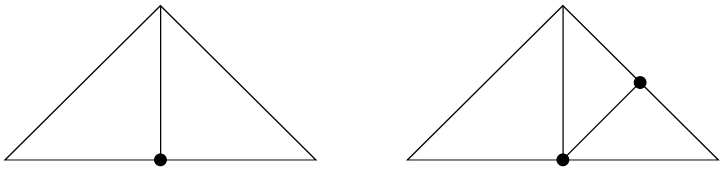


FIG. 4. *To resolve hanging nodes we can execute an additional step of green- or blue-refinement after the last refinement step* [7, 17]. *Here we show the* 2*D refinement rules. Left: green (*1 *hanging node). Right: blue (*2 *hanging nodes). If a triangle has* 3 *hanging nodes, it is red-refined.*

3.1. The reference simplex. Throughout the rest of this paper, let $\mathcal{L}$ be a given maximal refinement level. Instead of the unit cube $[0,1]^d$, we consider the scaled cube $[0,2^{\mathcal{L}}]^d$, ensuring that all node coordinates in a refinement up to level $\mathcal{L}$ are integers. Suppose we are given some d-simplex $T \subset \mathbb{R}^d$ together with a refinement $\mathscr{S}$ of T. By mapping T affine-linearly to $2^{\mathcal{L}}S_0$ the refinement $\mathscr{S}$ is mapped to a refinement $\mathscr{S}'$ of $2^{\mathcal{L}}S_0$. Therefore, to examine SFCs on refinements of T, it suffices to examine SFCs on $2^{\mathcal{L}}S_0$. Thus, we only consider refinements of the d-simplex $T_d^0 := 2^{\mathcal{L}}S_0$. Let $\mathcal{T}_d$ be the set of all possible descendants of this d-simplex with level smaller than or equal to $\mathcal{L}$; thus

$$\mathcal{T}_d = \left\{ T \,|\, T \text{ is a descendant of } T_d^0 \text{ with } 0 \le \ell(T) \le \mathcal{L} \right\}. \tag{4}$$

Any refinement (up to level $\mathcal{L}$) of T_d^0 is a subset of $\mathcal{T}_d$, and for each $T \in \mathcal{T}_d$ there exists at least one refinement $\mathscr{T}$ of T_d^0 with $T \in \mathscr{T}$. In this context, we refer to T_d^0 as the *root simplex*. Furthermore, let $\mathbb{L}^d$ denote the set of all possible anchor node coordinates of d-simplices in $\mathcal{T}_d$; thus

$$\begin{aligned} \mathbb{L}^2 &= \left\{ [0,2^{\mathcal{L}})^2 \cap \mathbb{Z}^2 \,|\, y \le x \right\}, \\ \mathbb{L}^3 &= \left\{ [0,2^{\mathcal{L}})^3 \cap \mathbb{Z}^3 \,|\, y \le z \le x \right\}. \end{aligned} \tag{5}$$

TABLE 1

For a d-simplex T of type b the table gives the types $Ct(T_0), \ldots, Ct(T_{2^d-1})$ of T's children. The corner-children T_0, T_1, T_2 (and in 3D also T_3) always have the same type as T.

Ct		Child			
2D		T_0	T_1	T_2	T_3
b	0	0	0	0	1
	1	1	1	1	0

Ct		Child				
3D		$T_0, \ldots, T_3$	T_4	T_5	T_6	T_7
b	0	0	4	5	2	1
	1	1	3	2	5	0
	2	2	0	1	4	3
	3	3	5	4	1	2
	4	4	2	3	0	5
	5	5	1	0	3	4

Note that we could have chosen any other of the S_i (scaled by $2^{\mathcal{L}}$) as the root simplex, and we do not see any advantage or disadvantage in doing so.

3.2. The type and Tet-id of a *d*-simplex. Making use of Property 4, we define the following.

DEFINITION 5. *Each d-simplex $T \in \mathcal{T}_d$ of level ℓ lies in a d-cube of the hexahedral mesh that is part of a uniform level ℓ refinement of $[0, 2^{\mathcal{L}}]^d$. This specific cube is the* associated cube *of T and denoted by Q_T. The d-simplex T is a scaled and shifted version of exactly one of the six tetrahedra S_i that constitute the unit cube, and we define the* type *of T as this number,* $\text{type}(T) := i$.

The anchor node of a subcube of level ℓ is the particular corner of that cube with the smallest x-, y- (and z-) coordinates. This means that for each simplex T in the refinement from Figure 3 the anchor node of T and the anchor node of its associated cube coincide. Any two d-simplices in $\mathcal{T}_d$ with the same associated cube are distinguishable by their type.

From Bey's observation from Figure 3 it follows that any simplex in $\mathcal{T}_d$ can be obtained by specifying a level ℓ, then choosing one level ℓ subcube of the root cube, and finally fixing a type. This provides motivation for the following definition.

DEFINITION 6 (Tet-id). *For $T = [\vec{x}_0, \ldots, \vec{x}_d] \in \mathcal{T}_d$ we define the* Tet-id *of T as the tuple of its anchor node and type; thus*

$$\textit{Tet-id}(T) := (\vec{x}_0, \text{type}(T)). \tag{6}$$

COROLLARY 7. *Let $T, T' \in \mathcal{T}_d$. Then $T = T'$ if and only if their Tet-ids and levels are the same.*

Note that in an arbitrary adaptive mesh there can be simplices with different levels, and each simplex T has an associated cube of level $\ell(T)$. In particular, simplices with the same anchor node can have different associated cubes if their levels are not equal.

Since any simplex in $\mathcal{T}_d$ can be specified by the Tet-id and level, the Tet-id provides an important tool for our work. The construction of the TM-index in the next section and the algorithms that we present in section 4 rely on the Tet-id as the fundamental data of a simplex. All information about a mesh can be extracted from the Tet-id and level of each element.

Since the root simplex has type 0, in a uniform refinement more simplices have type 0 than any other type. However, a close examination of Table 1 together with a short inductive argument leads to the following proposition.

PROPOSITION 8. *In the limit $\ell \to \infty$ the different types of simplices in a uniform level ℓ refinement of $\mathcal{T}_d$ occur in equal ratios.*

3.3. Encoding of the tetrahedral Morton index. In addition to the anchor coordinates the TM-index also depends on the types of all ancestors of a simplex. In order to define the TM-index we start by giving a formal definition of the interleaving operation and some additional necessary information.

DEFINITION 9. *We define the* interleaving $a \dot{\perp} b$ *of two n-tuples $a = (a_{n-1}, \ldots, a_0)$ and $b = (b_{n-1}, \ldots, b_0)$ as the $2n$-tuple obtained by alternating the entries of a and b:*

$$a \dot{\perp} b := (a_{n-1}, b_{n-1}, \ldots, a_0, b_0). \tag{7}$$

The interleaving of more than two n-tuples $a^1, \ldots, a^m$ is defined analogously as the mn-tuple

$$a^1 \dot{\perp} \cdots \dot{\perp} a^m := (a^1_{n-1}, a^2_{n-1}, \ldots, a^m_{n-1}, a^1_{n-2}, \ldots, a^{m-1}_0, a^m_0). \tag{8}$$

Remark 10. The TM-index of a d-simplex $T \in \mathcal{T}_d$ that we are going to define is constructed by interleaving $d+1$ $\mathcal{L}$-tuples, where the first d are the binary representations of the coordinates of T's anchor node, and the last is the tuple consisting of the types of the ancestors of T.

DEFINITION 11. *Let $T \in \mathcal{T}_3$ be a tetrahedron of refinement level ℓ with anchor node $\vec{x}_0 = (x, y, z)^T \in \mathbb{L}^3$. Since $x, y, z \in \mathbb{N}_0$ with $0 \le x, y, z < 2^{\mathcal{L}}$, we can express them as binary numbers with $\mathcal{L}$ digits, writing*

$$x = \sum_{j=0}^{\mathcal{L}-1} x_j 2^j, \quad y = \sum_{j=0}^{\mathcal{L}-1} y_j 2^j, \quad z = \sum_{j=0}^{\mathcal{L}-1} z_j 2^j. \tag{9}$$

We define the $\mathcal{L}$-tuples X, Y, and Z as the $\mathcal{L}$-tuples consisting of the binary digits of x, y, and z; thus,

$$X = X(T) := (x_{\mathcal{L}-1}, \ldots, x_0), \tag{10a}$$

$$Y = Y(T) := (y_{\mathcal{L}-1}, \ldots, y_0), \tag{10b}$$

$$Z = Z(T) := (z_{\mathcal{L}-1}, \ldots, z_0). \tag{10c}$$

In 2D we get the same definitions with X and Y, leaving out the z-coordinate.

DEFINITION 12. *For a $T \in \mathcal{T}_d$ of level ℓ and each $0 \le j \le \ell$ let T^j be the (unique) ancestor of T of level j. In particular, $T^\ell = T$. We define $B(T)$ as the $\mathcal{L}$-tuple consisting of the types of T's ancestors in the first ℓ entries, starting with T^1. The last $\mathcal{L} - \ell$ entries of $B(T)$ are zero:*

$$B = B(T) := \left(\underbrace{\mathrm{type}(T^1), \mathrm{type}(T^2), \ldots, \mathrm{type}(T)}_{\ell \text{ entries}}, 0, \ldots, 0 \right). \tag{11}$$

Thus, if we write B as an $\mathcal{L}$-tuple with indexed entries b_i,

$$B = B(T) = (b_{\mathcal{L}-1}, \ldots, b_0) \in \{0, \ldots, d! - 1\}^{\mathcal{L}}, \tag{12}$$

then the ith entry b_i is given as

$$b_i = \begin{cases} \mathrm{type}(T^{\mathcal{L}-i}), & \mathcal{L} - 1 \ge i \ge \mathcal{L} - \ell, \\ 0, & \mathcal{L} - \ell > i \ge 0. \end{cases} \tag{13}$$

DEFINITION 13 (tetrahedral Morton index). *We define the* tetrahedral Morton index *(*TM-index*)* $m(T)$ *of a* d*-simplex* $T \in \mathcal{T}_d$ *as the interleaving of the* $\mathcal{L}$*-tuples* Z *(for tetrahedra),* Y, X*, and* B*. Thus,*

$$m(T) := Y \mathbin{\dot\perp} X \mathbin{\dot\perp} B \tag{14a}$$

for triangles and

$$m(T) := Z \mathbin{\dot\perp} Y \mathbin{\dot\perp} X \mathbin{\dot\perp} B \tag{14b}$$

for tetrahedra.

This index resembles the well-known Morton index or Z-order for d-dimensional cubes, which we denote by $\widetilde{m}$ here. For such a cube Q the Morton index is usually defined as the bitwise interleaving of its coordinates. Thus $\widetilde{m}(Q) = Z \mathbin{\dot\perp} Y \mathbin{\dot\perp} X$, respectively, $\widetilde{m}(Q) = Y \mathbin{\dot\perp} X$; see [34, 52, 16].

As we show in section 4, the TM-index can be computed from the Tet-id of T with no further information given. Thus, in an implementation it is not necessary to store the $\mathcal{L}$-tuple B.

The TM-index of a d-simplex builds up from packs of d bits z_i (for tetrahedra), y_i, and x_i followed by a type $b_i \in \{0, \dots, d! - 1\}$. Since $d! = 2 < 4$ for $d = 2$, we can interpret the 2D TM-index as a quarternary number with digits $(y_i x_i)_2$ and b_i:

$$\begin{aligned} m(T) &= ((y_{\mathcal{L}-1} x_{\mathcal{L}-1})_2, b_{\mathcal{L}-1}, \dots, (y_0 x_0)_2, b_0)_4 \\ &= \sum_{i=0}^{\mathcal{L}-1} \left((2y_i + x_i) 4^{2i+1} + b_i 4^{2i} \right). \end{aligned} \tag{15a}$$

Similarly, we can interpret it as an octal number with digits $(z_i y_i x_i)_2$ and b_i for $d = 3$, since then $d! = 6 < 8$:

$$\begin{aligned} m(T) &= ((z_{\mathcal{L}-1} y_{\mathcal{L}-1} x_{\mathcal{L}-1})_2, b_{\mathcal{L}-1}, \dots, (z_0 y_0 x_0)_2, b_0)_8 \\ &= \sum_{i=0}^{\mathcal{L}-1} \left((4z_i + 2y_i + x_i) 8^{2i+1} + b_i 8^{2i} \right). \end{aligned} \tag{15b}$$

The entries in these numbers are only nonzero up to the level ℓ of T, since $x_{\mathcal{L}-i} = y_{\mathcal{L}-i} = (z_{\mathcal{L}-i} =) b_{\mathcal{L}-i} = 0$ for all $i > \ell$. The octal/quarternary representation (15) directly gives an order on the TM-indices, and therefore it is possible to construct an SFC from it, which we examine further in section 3.6. We use $m(T)$ to denote both the $(d+1)\mathcal{L}$-tuple from (14) and the number given by (15).

Let us look at Figure 5 for a short example to motivate this definition of the TM-index. Since the anchor coordinates and the type together with the level uniquely determine a d-simplex in $\mathcal{T}_d$, one could ask why we do not define the index to be $((Z \mathbin{\dot\perp}) Y \mathbin{\dot\perp} X, \mathrm{type}(T))$, a pair consisting of the Morton index of the associated cube of T and the type of T. This index was introduced for triangles in a slightly modified version as semiquadcodes in [38] and would certainly require less information since the computation of the sequence B would not be necessary. However, it results in an SFC that traverses the leaf cubes of a refinement in the usual Z-order, and inside of each cube it traverses the $d!$ different simplices in the order $S_0, \dots, S_{d!-1}$. As a result, there can be simplices T whose children are not traversed as a group, which means that there is a tetrahedron T', which is not an ancestor or descendant of T, such that some child T_i of T is traversed before T' and T' is traversed before another child T_j

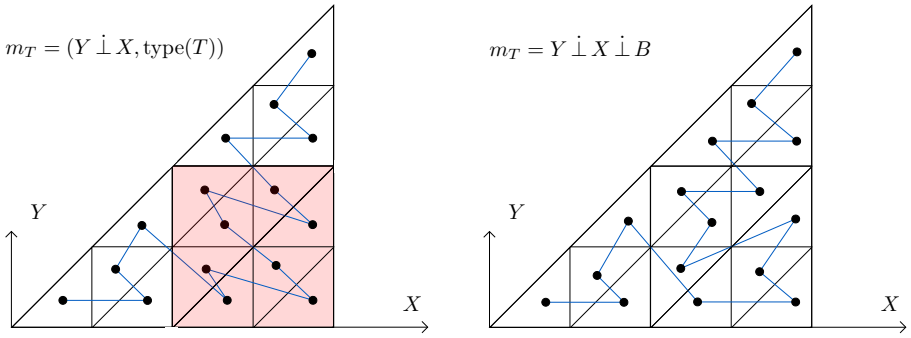


FIG. 5. *Comparing a straightforward definition of a Morton-type SFC with our approach. Left: The SFC arising from taking the Morton order of the quadrants and only dividing into triangles on the last level. Thus the index is* $(Y \dot{\perp} X, \text{type}(T))$. *As we see on the two coarse triangles that are shaded, the children of a level* 1 *triangle are not necessarily traversed before any other triangle is traversed. Such a locality property would be desirable, and therefore this index is not suitable for our purposes. Right: The SFC arising from the TM-index from our Definition* 13. *We see that for each level* 1 *triangle its four children are traversed as a group. Theorem* 16 *states that this property holds for any parent triangle/tetrahedron. The order in which the children are traversed depends (only) on the type of the parent and is different from Bey's order given by* (2).

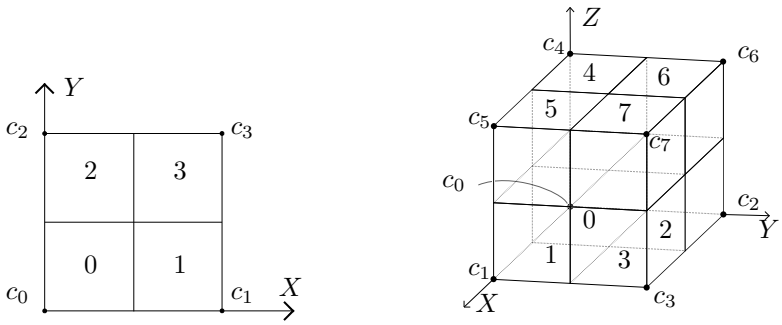


FIG. 6. *Left: A square is refined to four children, each of which corresponds to a corner of the square. The number of the corner is the cube-id of that child. Right: In* 3*D a cube is refined to eight children. Their cube-ids and corner numbers are shown as well.*

of T. Theorem 16 states that this problem does not occur with the TM-index that we have defined above. Figure 5 compares the two approaches for a uniform level 2 refinement of T_2^0.

3.4. A different approach to deriving the TM-index. There is another interpretation of the TM-index which is particularly useful for the AMR algorithms presented in section 4. In order to define it we introduce the concept of the so-called cube-id. According to Figure 2 we number the 2^d corners of a d-dimensional cube by $c_0, \ldots, c_{2^d-1}$ in a zyx-order (x varies fastest). When refining a cube to 2^d children, each child has exactly one of the c_i as a corner, and it is therefore convenient to number the children by $c_0, \ldots, c_{2^d-1}$ as well. For the child c_i we call the number i the *cube-id* of that child; see Figure 6 for an illustration. Since each cube Q that is not the root has a unique parent, it also has a unique cube-id. This cube-id can easily be computed by interleaving the last significant bits of the z- (in 3D), y-, and x-coordinates of Q's anchor node.

DEFINITION 14. *Because each d-simplex $T \in \mathcal{T}_d$ of level ℓ has a unique associated cube, we define the* cube-id *of T to be the cube-id of the associated cube of T, that is, the d-cube of level ℓ that has the same anchor node as T.*

If X, Y (and Z) are as is in Definition 11, then we can write the cube-id of T's ancestors as

$$\begin{aligned} &\text{cube-id}(T^i) = (y_i x_i)_2 \quad &&\text{in 2D},\\ &\text{cube-id}(T^i) = (z_i y_i x_i)_2 \quad &&\text{in 3D}, \end{aligned} \tag{16}$$

and therefore, using (15), we can rewrite the TM-index of T as

$$m(T) = (\text{cube-id}(T^1), \text{type}(T^1), \ldots, \text{cube-id}(T^\ell), \text{type}(T^\ell), 0, \ldots, 0)_{2^d}. \tag{17}$$

This resembles the Morton index of the associated cube Q_T of T, since we can write this as

$$\widetilde{m}(Q_T) = (\text{cube-id}(Q^1), \ldots, \text{cube-id}(Q^\ell), 0, \ldots, 0)_{2^d}. \tag{18}$$

3.5. Properties of the TM-index. In this section we show that the TM-index has properties similar to that of the Morton index for cubes. We are particularly interested in locality properties stating that refining a simplex only locally changes the order given by the TM-index. We begin by stating the uniqueness of the TM-index if additionally a level ℓ is given.

PROPOSITION 15. *Together with a refinement level ℓ, the TM-index $m(T)$ uniquely determines a d-simplex in $\mathcal{T}_d$.*

Proof. If $\ell = 0$, then there is only one simplex of level ℓ in $\mathcal{T}_d$, which is T_d^0. So let $\ell > 0$ and $m = m(T)$ be given as in (14), and let ℓ be the level of T. From m we can compute the x-, y- (and z-) coordinates of the associated cube of T. We can also compute the type of T from the TM-index. By Corollary 7 this information uniquely determines T. ∎

For the Morton index $\widetilde{m}$ for cubes the following important properties are known [52]:

(i) The Morton indices of the descendants of a parent cube are larger than or equal to the index of the parent cube.
(ii) A Morton index of a cube Q is the prefix of an index of a cube P of higher level than Q if and only if P is a descendant of Q.
(iii) Refining only changes the SFC locally. Thus, if Q is a cube and P is a cube with $\widetilde{m}(Q) < \widetilde{m}(P)$ and P is not a descendant of Q, then $\widetilde{m}(Q') < \widetilde{m}(P)$ for each descendant Q' of Q.

Property (iii) defines a hierarchic invariant of the SFC that is specific to our construction (see Figure 5). We show below that properties (i), (ii), and (iii) hold for d-simplices and the TM-index described by (14).

THEOREM 16. *For arbitrary d-simplices $T \neq S \in \mathcal{T}_d$ the TM-index satisfies the following:*

(i) *If S is a descendant of T, then $m(T) \leq m(S)$.*
(ii) *If $\ell(T) < \ell(S)$, then $m(T)$ is a prefix of $m(S)$ if and only if S is a descendant of T.*
(iii) *If $m(T) < m(S)$ and S is no descendant of T, then for each descendant T' of T we have*

$$m(T) \leq m(T') < m(S). \tag{19}$$

The proof of Theorem 16 requires some work, and we need to show a technical result first. Hereby, we consider only the 3D case, since for 2D the argument is completely analogous. We define an embedding of the set of all TM-indices into the set of Morton indices for 6D cubes. Since the properties (i)–(iii) hold for these cubes it follows that they hold for tetrahedra as well. To this end, for a given tetrahedron $T \in \mathcal{T}_3$ we interpret each entry b_j of $B(T)$ as a 3-digit binary number

$$b_j = (b_j^2\, b_j^1\, b_j^0)_2, \tag{20}$$

which is possible since $b_j \in \{0, \ldots, 5\}$. We obtain three new $\mathcal{L}$-tuples B^2, B^1, B^0 satisfying

$$B = B^2 \mathbin{\dot\perp} B^1 \mathbin{\dot\perp} B^0, \tag{21}$$

and thus we can rewrite the TM-index as

$$m(T) = Z \mathbin{\dot\perp} Y \mathbin{\dot\perp} X \mathbin{\dot\perp} B^2 \mathbin{\dot\perp} B^1 \mathbin{\dot\perp} B^0. \tag{22}$$

Note that we can interpret each B^i as an $\mathcal{L}$-digit binary number for which we have $0 \le B^i < 2^{\mathcal{L}}$. Now let $\mathcal{Q}$ denote the set of all 6D cubes that are a child of the cube $Q_0 := [0, 2^{\mathcal{L}}]^6$:

$$\mathcal{Q} = \{\, Q \,|\, Q \text{ is a descendant of } Q_0 \text{ of level } 0 \le \ell \le \mathcal{L} \,\}. \tag{23}$$

Since a cube $Q \in \mathcal{Q}$ is uniquely determined by the six coordinates $(x_0, \ldots, x_5)$ of its anchor node plus its level ℓ, we also write $Q = Q_{(x_0,\ldots,x_5),\ell}$. Note that the Morton index for a cube can be defined as the bitwise interleaving of its anchor node coordinates [34]:

$$\widetilde{m}(Q) = X^5 \mathbin{\dot\perp} X^4 \mathbin{\dot\perp} X^3 \mathbin{\dot\perp} X^2 \mathbin{\dot\perp} X^1 \mathbin{\dot\perp} X^0. \tag{24}$$

PROPOSITION 17. *The map*

$$\begin{array}{rccl} \Phi\colon & \mathcal{T}_3 & \longrightarrow & \mathcal{Q}, \\ & T & \longmapsto & Q_{(B^0(T),B^1(T),B^2(T),x(T),y(T),z(T)),\ell(T)} \end{array} \tag{25}$$

is injective and satisfies

$$\widetilde{m}(\Phi(T)) = m(T). \tag{26}$$

Furthermore, it fulfills the property that T' is a child of T if and only if $\Phi(T')$ is a child of $\Phi(T)$.

Proof. The equation $m(T) = \widetilde{m}(\Phi(T))$ follows directly from the definitions of the TM-indices on $\mathcal{T}_3$ and $\mathcal{Q}$. From Proposition 15 we conclude that Φ is injective. Now let $T', T \in \mathcal{T}_3$, where T' is a child of T. Furthermore, let $\ell = \ell(T)$. We know that $Q' := \Phi(T')$ is a child of $Q := \Phi(T)$ if and only if for each $i \in \{0, \ldots, 5\}$ it holds that

$$x_i(Q') \in \left\{ x_i(Q), x_i(Q) + 2^{\mathcal{L}-(\ell+1)} \right\}. \tag{27}$$

Because of the underlying cube structure (compare Figure 3) we know that the x-coordinate of the anchor node of T' satisfies

$$x(T') \in \left\{ x(T), x(T) + 2^{\mathcal{L}-(\ell+1)} \right\}, \tag{28}$$

and likewise for $Y(T')$ and $Z(T')$. Therefore, (27) holds for $i = 3, 4, 5$. By definition $B^j(T')$ is the same as $B^j(T)$ except at position $\mathcal{L} - (\ell + 1)$, where

$$B^j(T')_{\mathcal{L}-(\ell+1)} = b^j_{\mathcal{L}-(\ell+1)}(T') \in \{\, 0, 1 \,\} \tag{29}$$

and

$$B^j(T)_{\mathcal{L}-(\ell+1)} = 0. \tag{30}$$

Hence, we conclude that (27) also holds for $i = 0, 1, 2$. So $\Phi(T')$ is a child of $\Phi(T)$.

To show the other implication, let us assume that $\Phi(T')$ is a child of $\Phi(T)$. Since $\ell(T') = \ell(\Phi(T')) > 0$, T' has a parent and we denote it by P. In the argument above we have shown that $\Phi(P)$ is the parent of $\Phi(T')$, and because each cube has a unique parent the identity $\Phi(P) = \Phi(T)$ must hold. Therefore, we get $P = T$ since Φ is injective; thus, T' is the child of T. ∎

Inductively we conclude that T' is a descendant of T if and only if $\Phi(T')$ is a descendant of $\Phi(T)$. Now Theorem 16 follows, because the desired properties (i)–(iii) hold for the Morton index of cubes.

3.6. The space-filling curve associated to the TM-index. By interpreting the TM-indices as 2^d-ary numbers as in (15) we get a total order on the set of all possible TM-indices, and therefore it gives rise to an SFC for any refinement $\mathscr{S}$ of T_d^0. In this section we further examine the SFC derived from the TM-index. We give here a recursive description of it, similarly to how it is done for the Sierpinski curve and other cubical SFC by Haverkort and van Walderveen [24].

Part (iii) of Theorem 16 tells us that the descendants of a simplex T are traversed before any other simplices with a higher TM-index than T are traversed. However, the order that the children of T have relative to each other can be different from the order of children of another simplex T'. In particular the order of the simplices defined by the TM-index differs from the order (2) defined by Bey. We observe this behavior in 2D in Figure 5 on the right-hand side: For the level 1 triangles of type 0 the children are traversed in the order

$$T_0, T_1, T_3, T_2, \tag{31}$$

and the children of the level 1 triangle of type 1 are traversed in the order

$$T_0, T_3, T_1, T_2. \tag{32}$$

In fact, the order of the children of a simplex T depends only on the type of T, as we show in the following proposition.

PROPOSITION 18. *If $T, T' \in \mathcal{T}_d$ are two d-simplices of given type $b = \mathrm{type}(T) = \mathrm{type}(T')$, then there exists a unique permutation $\sigma \equiv \sigma_b$ of $\{\, 0, \ldots, 2^d - 1 \,\}$ such that*

$$\begin{aligned} m(T_{\sigma(0)}) &< m(T_{\sigma(1)}) < \cdots < m(T_{\sigma(2^d-1)}), \\ m(T'_{\sigma(0)}) &< m(T'_{\sigma(1)}) < \cdots < m(T'_{\sigma(2^d-1)}). \end{aligned} \tag{33}$$

Thus, the children of T and the children of T' are in the same order with respect to their TM-index.

Proof. By ordering the children of T and T' with respect to their TM-indices, we obtain σ and σ' with

$$\begin{aligned} m(T_{\sigma(0)}) < m(T_{\sigma(1)}) < \cdots < m(T_{\sigma(2^d-1)}), \\ m(T'_{\sigma'(0)}) < m(T'_{\sigma'(1)}) < \cdots < m(T'_{\sigma'(2^d-1)}). \end{aligned} \tag{34}$$

These permutations are well-defined and unique with this property because different simplices of the same level never have the same TM-index; see Proposition 15. It remains to show that $\sigma' = \sigma$. Let $\ell = \ell(T)$ and $\ell' = \ell(T')$. Since the TM-indices of the children of T do all agree up to level ℓ, we see, using the notation from (15), that their order σ depends only on the $d+1$ numbers (z is omitted for $d = 2$)

$$z_{\mathcal{L}-(\ell+1)}(T_i), \quad y_{\mathcal{L}-(\ell+1)}(T_i), \quad x_{\mathcal{L}-(\ell+1)}(T_i), \quad \text{and} \quad b_{\mathcal{L}-(\ell+1)}(T_i). \tag{35}$$

The same argument applies to σ' and ℓ'. From now on we carry out the computations for $d = 3$. Since $\mathrm{type}(T) = \mathrm{type}(T')$, we can write

$$T = \lambda T' + \vec{c}, \tag{36}$$

with

$$\lambda = 2^{\ell'-\ell}, \quad \vec{c} = \begin{pmatrix} x(T) - x(T') \\ y(T) - y(T') \\ z(T) - z(T') \end{pmatrix}. \tag{37}$$

Since the refinement rules (2) commute with scaling and translation, we also obtain

$$T_i = \lambda T'_i + \vec{c} \tag{38}$$

for the children of T and T', and therefore

$$b_{\mathcal{L}-(\ell+1)}(T_i) = \mathrm{type}(T_i) = \mathrm{type}(T'_i) = b_{\mathcal{L}-(\ell'+1)}(T'_i) \tag{39}$$

for $0 \le i < 2^d$. Furthermore, we have

$$x_{\mathcal{L}-(\ell+1)}(T_i) = (x(T_i) - x(T))2^{-\mathcal{L}+(\ell+1)}, \tag{40}$$

from which we derive

$$\begin{aligned} x_{\mathcal{L}-(\ell+1)}(T_i) &= \lambda(x(T'_i) - x(T'))2^{-\mathcal{L}+(\ell+1)} \\ &= 2^{\ell'-\ell}(x(T'_i) - x(T'))2^{-\mathcal{L}+(\ell+1)} \\ &= (x(T'_i) - x(T'))2^{-\mathcal{L}+(\ell'+1)} \\ &= x_{\mathcal{L}-(\ell'+1)}(T'_i), \end{aligned} \tag{41}$$

and analogously

$$\begin{aligned} y_{\mathcal{L}-(\ell+1)}(T_i) = y_{\mathcal{L}-(\ell'+1)}(T'_i), \\ z_{\mathcal{L}-(\ell+1)}(T_i) = z_{\mathcal{L}-(\ell'+1)}(T'_i). \end{aligned} \tag{42}$$

This shows that the tetrahedral Morton order of the children of T and T' is the same and σ' must equal σ. ∎

TABLE 2

The local index for the children of a d-simplex T according to the TM-ordering. For each type b, the 2^d children $T_0, \ldots, T_{2^d-1}$ of a simplex of this type can be ordered according to their TM-indices. The position of T_i according to the TM-order is the local index $I_{\mathrm{loc}}(T_i) = \sigma_b(i)$.

I_{loc} 2D		Child T_0	T_1	T_2	T_3
b	0	0	1	3	2
	1	0	2	3	1

I_{loc} 3D		Child T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7
b	0	0	1	4	7	2	3	6	5
	1	0	1	5	7	2	3	6	4
	2	0	3	4	7	1	2	6	5
	3	0	1	6	7	2	3	4	5
	4	0	3	5	7	1	2	4	6
	5	0	3	6	7	2	1	4	5

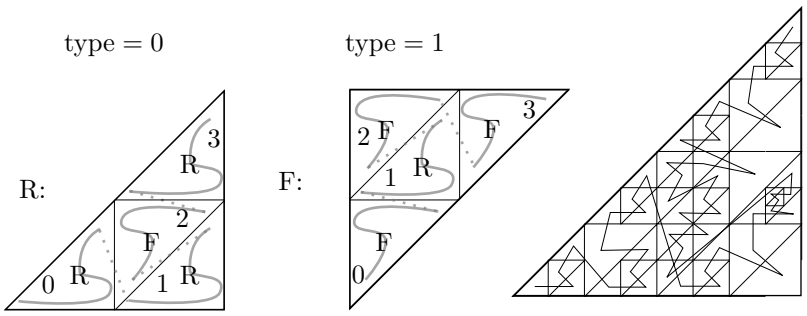


FIG. 7. *Left: Using the notation from* [24] *we recursively describe the SFC arising from the TM-index for triangles. The number inside each child triangle is its local index. R denotes the refinement scheme for type* 0 *triangles, and F for type* 1 *triangles. This pattern can be obtained from Tables* 1 *and* 2*. Right: The SFC for an example adaptive refinement of the root triangle.*

DEFINITION 19. *Let $T \in \mathcal{T}_d$ such that T's parent P has type b and T is the ith child of P according to Bey's order* (2)*, $0 \le i < 2^d$. We call the number $\sigma_b(i)$ the* local index *of the d-simplex T and use the notation*

$$I_{\mathrm{loc}}(T) := \sigma_b(i) \tag{43}$$

to denote the child number in the TM-ordering, subsequently written TM-child. By definition, the local index of the root simplex is zero, $I_{\mathrm{loc}}(T_d^0) := 0$. Table 2 *lists the local indices for each parent type.*

Thus, we know for each type $0 \le b < d!$ how the children of a tetrahedron of type b are traversed. This gives us an approach for describing the SFC arising from the TM-index in a recursive fashion [24]. By specifying for each possible type b the order and types of the children of a type b simplex, we can build up the SFC. In Figure 7 we describe the SFC for triangles in this way. In 3D it is not convenient to draw the six pictures for the different types, but the SFC can be derived similarly from Tables 1 and 2.

4. Algorithms on simplices. In this section we present fundamental algorithms that operate on d-simplices in $\mathcal{T}_d$. These algorithms include computations of parent and child simplices, computation of face-neighbors, and computations involved with the TM-index. To simplify the notation we carry out all algorithms for tetrahedra and then describe how to modify them for triangles. We introduce the data type `Tet` and do not distinguish between the abstract concept of a `Tet` and the geomet-

ric object (tetrahedron or triangle) that it represents. The data type `Tet` T has the following members:

- $T.\ell$ — the refinement level of T;
- $T.\vec{x} = (T.x, T.y, T.z)$ — the x-, y-, and z-coordinates of T's anchor node, also sometimes referred to as $T.x_0$, $T.x_1$, and $T.x_2$;
- $T.b$ — the type of T.

In 2D computations the parameter $T.z$ is not present. To avoid confusion we use the notation $\vec{x}$ to denote vectors in $\mathbb{Z}^d$ and x (without arrow) for integers, thus numbers in $\mathbb{Z}$. From Corollary 7 we know that the values stored in a `Tet` suffice to uniquely identify a d-simplex $T \in \mathcal{T}$.

Remark 20 (storage requirement). The algorithms that we present in this section only need this data as input for a simplex resulting in a fixed storage size per `Tet`. If, for example, the maximum level $\mathcal{L}$ is 32 or less, then the coordinates can be stored in one 4-byte integer per dimension, while the level and type occupy one byte each, leading to a total storage of

$$\begin{aligned} &2 \times 4 + 1 + 1 = 10 \quad \text{bytes per } \texttt{Tet} \text{ in 2D,} \\ &3 \times 4 + 1 + 1 = 14 \quad \text{bytes per } \texttt{Tet} \text{ in 3D.} \end{aligned} \tag{44}$$

Remark 21 (runtime). Most of these algorithms run in constant time independent of the maximum level $\mathcal{L}$. The only operations using a loop over the level $\mathcal{L}$ or $T.\ell$, thus having $\mathcal{O}(\mathcal{L})$ runtime, are computing the consecutive index from a `Tet` and initializing a `Tet` according to a given consecutive index. Hence, we show how to replace repetitive calls of these relatively involved algorithms by more efficient constant-time ones.

4.1. The coordinates of a *d*-simplex. The coordinates of the $d+1$ nodes of a d-simplex T can be obtained easily from its Tet-id, the relation (3), and simple arithmetic: If T is a d-simplex of level ℓ, type b, and anchor node $\vec{x}_0 \in \mathbb{Z}^d$, then

$$T = 2^{\mathcal{L}-\ell} S_b + \vec{x}_0. \tag{45}$$

Hence, in order to compute the coordinates of T we can take the coordinates of S_b, as given in (3), and then use relation (45). A closer look at (3) reveals that it is not necessary to examine all coordinates of S_b in order to compute the x_i, but that they can also be computed arithmetically. This computation is carried out in Algorithm 4.1.

4.2. Parent and child. In this section we describe how to compute the Tet-ids of the parent $P(T)$ and of the 2^d children T_i, $0 \le i < 2^d$, of a given d-simplex $T \in \mathcal{T}_d$. Computing the anchor node coordinates of the parent is easy, since their first $T.\ell - 1$ bits correspond to the coordinates of T's anchor node and the rest of their bits is zero. For computing the type of $P(T)$, we need the function

$$\begin{aligned} \mathrm{Pt}\colon \{0, \dots, 2^d - 1\} \times \{0, \dots, d! - 1\} &\longrightarrow \{0, \dots, d! - 1\}, \\ (\text{cube-id}(T), T.b) &\longmapsto P.b, \end{aligned} \tag{46}$$

giving the type of T's parent in dependence of its cube-id and type. In Figure 8 we list all values of this function for $d \in \{2, 3\}$.

The algorithm `Parent` to compute the parent of T now puts these two ideas together, computing the coordinates and type of $P(T)$. Algorithm 4.3 shows an implementation. It uses Algorithm 4.2 to compute the cube-id of a d-simplex.

Algorithm 4.1: `Coordinates`(`Tet` T)

$X \leftarrow (T.\vec{x}, 0, 0, 0)$;
$h \leftarrow 2^{\mathcal{L}-\ell}$;
$i \leftarrow \lfloor \frac{T.b}{2} \rfloor$; `/* Replace with` $i \leftarrow T.b$ `for 2D */`
if $T.b \,\%\, 2 = 0$ **then**
  $j \leftarrow (i+2) \,\%\, 3$
else
  $j \leftarrow (i+1) \,\%\, 3$
$X[1] \leftarrow X[0] + he_i$;
$X[2] \leftarrow X[1] + he_j$; `/* Replace with` $X[2] \leftarrow X[0] + (h,h)^T$ `for 2D */`
$X[3] \leftarrow X[0] + (h,h,h)^T$; `/* Remove this line for 2D */`
return X;

Pt(c,b) 2D		b 0	b 1
c	0	0	1
	1	0	0
	2	1	1
	3	0	1

Pt(c,b) 3D		b 0	b 1	b 2	b 3	b 4	b 5
c	0	0	1	2	3	4	5
	1	0	1	1	1	0	0
	2	2	2	2	3	3	3
	3	1	1	2	2	2	1
	4	5	5	4	4	4	5
	5	0	0	0	5	5	5
	6	4	3	3	3	4	4
	7	0	1	2	3	4	5

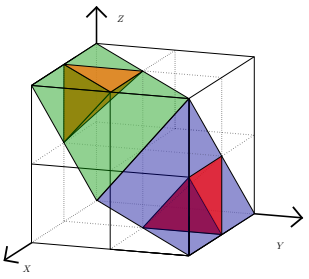


FIG. 8. *The type of the parent of a d-simplex T can be determined from T's cube-id c and type b. Left: The values of Pt from (46) in the 2D case. Middle: The values of the function Pt in the 3D case. Right: Two examples showing the computation in 3D. (1) The small tetrahedron in the top left corner (orange) has cube-id 5 and type 4, and its parent (green) can be seen to have type 5. (2) The small tetrahedron at the bottom right (red) has cube-id 3 and type 3, and its parent (blue) has type 2.*

Algorithm 4.2: `c-id`(`Tet` T,int ℓ)

$i \leftarrow 0$, $h \leftarrow 2^{\mathcal{L}-\ell}$
$i \mid= (T.x \,\&\, h)\,?\,1 : 0$
$i \mid= (T.y \,\&\, h)\,?\,2 : 0$
$i \mid= (T.z \,\&\, h)\,?\,4 : 0$ `/* Remove this line for 2D */`
return i

Algorithm 4.3: `Parent`(`Tet` T)

$h \leftarrow 2^{\mathcal{L}-T.\ell}$
$P.\ell \leftarrow T.\ell - 1$
$P.x \leftarrow T.x \,\&\, \neg h$
$P.y \leftarrow T.y \,\&\, \neg h$
$P.z \leftarrow T.z \,\&\, \neg h$ `/* Remove this line for 2D */`
$P.b \leftarrow \mathrm{Pt}(\texttt{c-id}(T, T.\ell), T.b)$ `/* See (46) and Figure 8 for Pt */`
return P

For the computation of one child T_i of T for a given $i \in \{0, \ldots, 2^d - 1\}$ we look at Bey's definition of the subsimplices in (2) and see that in order to compute the anchor node of the child we need to know some of the node coordinates $\vec{x}_0, \ldots, \vec{x}_d$ of the parent simplex T. These can be obtained via Algorithm 4.1. However, it is more efficient to compute only those coordinates of T that are actually necessary. To compute the Tet-id of T_i we also need to know its type. The type of T_i depends only on the type of T, and in the algorithm we use the function `Ct` (children type) to compute this type. `Ct` is effectively an evaluation of Table 1. Algorithm 4.4 shows now how to compute the coordinates of the ith child of T in Bey's order.

When we would like to compute the ith child of a d-simplex T of type b with respect to the tetrahedral Morton order (thus the child T_k of T with $I_{\mathrm{loc}}(T_k) = i$) we just call Algorithm 4.4 with $\sigma_b^{-1}(i)$ as input. The permutations σ_b^{-1} are available from Table 2; see (43) and Algorithm 4.5.

Algorithm 4.4: `Child`(`Tet` T,int i)

```
1 X ← Coordinates(T)
2 if i = 0 then j ← 0
3 else if i ∈ {1, 4, 5} then j ← 1
4 else if i ∈ {2, 6, 7} then j ← 2
5 else if i = 3 then j ← 3                    /* If i = 3 then j ← 1 for 2D */
6 T_i.x⃗ ← (1/2)(X[0] + X[j])
7 T_i.b ← Ct(T.b, i)                                        /* See Table 1 */
```

Algorithm 4.5: `TM-Child`(`Tet` T,int i)

```
1 return Child (T, σ^{-1}_{T.b}(i))                         /* See Table 2 */
```

4.3. Neighbor simplices. Many applications—e.g., finite element methods—need to gather information about the face-neighboring simplices of a given simplex in a refinement. In this section we describe a level-independent constant-runtime algorithm to compute the Tet-id of the same level neighbor along a specific face f of a given d-simplex T. This algorithm is very lightweight since it only requires a few arithmetic computations involving the Tet-id of T and the number f. In comparison to other approaches to finding neighbors in constant time [32, 11], our algorithm does not involve the computation of any of T's ancestors.

The $d+1$ faces of a d-simplex $T = [\vec{x}_0, \ldots, \vec{x}_d]$ are numbered $f_0, \ldots, f_d$ in such a way that face f_i is the face not containing the node $\vec{x}_i$. To examine the situation where two d-simplices of the same level share a common face, let $\mathcal{T}_d^\ell$ denote a uniform refinement of T_d^0 of a given level $0 \le \ell \le \mathcal{L}$,

$$\mathcal{T}_d^\ell := \{\, T \,|\, T \text{ is a descendant of } T_d^0 \text{ of level } \ell \,\} \subset \mathcal{T}_d. \tag{47}$$

$\mathcal{T}_d$ can be seen as the disjoint union of all the $\mathcal{T}_d^\ell$:

$$\mathcal{T}_d = \bigcup_{\ell=0}^{\mathcal{L}} \mathcal{T}_d^\ell. \tag{48}$$

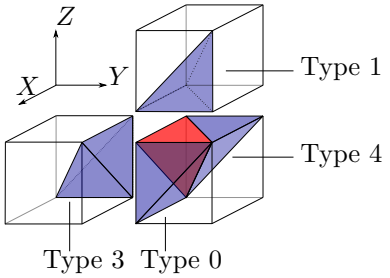


FIG. 9. *A tetrahedron T of type 5 (in the middle, red) and its four face neighbors (blue) of types 1, 0, 4, and 3, drawn with their associated cubes (exploded view). We see that the type of T's neighbors depends only on its type, while their node coordinates relative to T's depend additionally on T's level. (Color available online.)*

TABLE 3

Face neighbors in 2D. For each possible type $b \in \{0,1\}$ of a triangle T and each of its faces $f = f_i$, $i \in \{0,1,2\}$, the type, anchor node coordinates, and corresponding face number $\tilde{f}$ of T's neighbor across f are shown. In the 2D case we can directly compute $\mathcal{N}.b = 1 - T.b$ and $\tilde{f} = 2 - f$. Here, $h = 2^{\mathcal{L}-\ell}$ refers to the length of one square of level ℓ.

2D	f	0	1	2
$T.b = 0$	$\mathcal{N}_f.b$	1	1	1
	$\mathcal{N}_f.x$	$T.x + h$	$T.x$	$T.x$
	$\mathcal{N}_f.y$	$T.y$	$T.y$	$T.y - h$
	$\tilde{f}$	2	1	0
$T.b = 1$	$\mathcal{N}_f.b$	0	0	0
	$\mathcal{N}_f.x$	$T.x$	$T.x$	$T.x - h$
	$\mathcal{N}_f.y$	$T.y + h$	$T.y$	$T.y$
	$\tilde{f}$	2	1	0

Given a d-simplex $T \in \mathcal{T}_d^\ell$ and a face number $i \in \{0,\dots,d\}$, denote T's neighbor in $\mathcal{T}_d^\ell$ across face $f = f_i$ by $\mathcal{N}_f(T)$, and denote the face number of the neighbor simplex $\mathcal{N}_f(T)$ across which T is its neighbor by $\tilde{f}(T)$. Hence, the relation

$$\mathcal{N}_{\tilde{f}(T)}(\mathcal{N}_f(T)) = T \tag{49}$$

holds for each face f of T.

Our aim is to compute the Tet-id of $\mathcal{N}_f(T)$ and $\tilde{f}(T)$ from the Tet-id of T. Using the underlying cube structure, this problem can be solved for each occurring type of d-simplex separately, and the solution scheme is independent of the coordinates of T and of ℓ. In Figure 9 the situation for a tetrahedron of type 5 is illustrated, and in Tables 3 and 4 we present the general solution for each type.

Using these tables, a constant-time computation of the Tet-id of $\mathcal{N}_f(T)$ and of $\tilde{f}(T)$ from the Tet-id of T is possible, and the 3D case is carried out in Algorithm 4.6. Note that this algorithm uses arithmetic expressions in $T.b$ to avoid the sixfold distinction of cases.

Remark 22. To find existing neighbors in a nonuniform refinement we use Algorithm 4.6 in combination with `Parent` and `TM-Child` and comparison functions.

Of course it is possible that $\mathcal{N}_f(T)$ does not belong to $\mathcal{T}_d^\ell$ any longer. If this is the case, then f was part of the boundary of the root simplex T_d^0. We describe in the next section how we can decide in constant time whether $\mathcal{N}_f(T)$ is in T_d^ℓ or not.

TABLE 4

Face neighbors in 3D. For each possible type $b \in \{0,1,2,3,4,5\}$ of a tetrahedron T and each of its faces $f = f_i$, $i \in \{0,1,2,3\}$, the type $\mathcal{N}_f(T).b$ of T's neighbor across f, its coordinates of the anchor node $\mathcal{N}_f(T).x$, $\mathcal{N}_f(T).y$, $\mathcal{N}_f(T).z$, and the corresponding face number $\tilde{f}(T)$, across which T is $\mathcal{N}_f(T)$'s neighbor, are shown.

3D	f	0	1	2	3
$T.b = 0$	$\mathcal{N}_f.b$	4	5	1	2
	$\mathcal{N}_f.x$	$T.x + h$	$T.x$	$T.x$	$T.x$
	$\mathcal{N}_f.y$	$T.y$	$T.y$	$T.y$	$T.y - h$
	$\mathcal{N}_f.z$	$T.z$	$T.z$	$T.z$	$T.z$
	$\tilde{f}$	3	1	2	0
$T.b = 1$	$\mathcal{N}_f.b$	3	2	0	5
	$\mathcal{N}_f.x$	$T.x + h$	$T.x$	$T.x$	$T.x$
	$\mathcal{N}_f.y$	$T.y$	$T.y$	$T.y$	$T.y$
	$\mathcal{N}_f.z$	$T.z$	$T.z$	$T.z$	$T.z - h$
	$\tilde{f}$	3	1	2	0
$T.b = 2$	$\mathcal{N}_f.b$	0	1	3	4
	$\mathcal{N}_f.x$	$T.x$	$T.x$	$T.x$	$T.x$
	$\mathcal{N}_f.y$	$T.y + h$	$T.y$	$T.y$	$T.y$
	$\mathcal{N}_f.z$	$T.z$	$T.z$	$T.z$	$T.z - h$
	$\tilde{f}$	3	1	2	0

3D	f	0	1	2	3
$T.b = 3$	$\mathcal{N}_f.b$	5	4	2	1
	$\mathcal{N}_f.x$	$T.x$	$T.x$	$T.x$	$T.x - h$
	$\mathcal{N}_f.y$	$T.y + h$	$T.y$	$T.y$	$T.y$
	$\mathcal{N}_f.z$	$T.z$	$T.z$	$T.z$	$T.z$
	$\tilde{f}$	3	1	2	0
$T.b = 4$	$\mathcal{N}_f.b$	2	3	5	0
	$\mathcal{N}_f.x$	$T.x$	$T.x$	$T.x$	$T.x - h$
	$\mathcal{N}_f.y$	$T.y$	$T.y$	$T.y$	$T.y$
	$\mathcal{N}_f.z$	$T.z + h$	$T.z$	$T.z$	$T.z$
	$\tilde{f}$	3	1	2	0
$T.b = 5$	$\mathcal{N}_f.b$	1	0	4	3
	$\mathcal{N}_f.x$	$T.x$	$T.x$	$T.x$	$T.x$
	$\mathcal{N}_f.y$	$T.y$	$T.y$	$T.y$	$T.y - h$
	$\mathcal{N}_f.z$	$T.z + h$	$T.z$	$T.z$	$T.z$
	$\tilde{f}$	3	1	2	0

Algorithm 4.6: `Face-neighbor`(`Tet` T,int f)

```
1  b ← T.b, x_0 ← T.x_0, x_1 ← T.x_1, x_2 ← T.x_2
2  if f = 1 or f = 2 then
3  |  f̃ ← f if (b % 2 = 0 and f = 2) or (b % 2 ≠ 0 and f = 1) then
4  |  |  b ← b + 1
5  |  else
6  |  |  b ← b − 1
7  else
8  |  f̃ ← 3 − f
9  |  h ← 2^(L − T.ℓ)
10 |  if f = 0 then                                   /* f = 0 */
11 |  |  i ← b div 2
12 |  |  x_i ← T.x_i + h
13 |  |  b ← b + (b % 2 = 1 ? 2 : 4)
14 |  else                                           /* f = 3 */
15 |  |  i ← (b + 3) % 6 div 2
16 |  |  x_i ← T.x_i − h
17 |  |  b ← b + (b % 2 = 0 ? 2 : 4)
18 return (x_0, x_1, x_2, b % 6, f̃)
```

For completeness, we summarize the geometry and numbers of d-simplices touching each other via a corner ($d = 2$ or $d = 3$) or edge (only $d = 3$). In this paper we do not list these neighboring tetrahedra in detail.

For $d = 3$ each corner in the mesh $\mathcal{T}_3^\ell$ has 24 adjacent tetrahedra; thus each tetrahedron has at each corner 23 other tetrahedra that share this particular corner. For $d = 2$ the situation is similar, with six triangles meeting at each corner. To

examine the number of adjacent tetrahedra to an edge we distinguish three types of edges in $\mathcal{T}_d^\ell$:

1. edges that are also edges in the underlying hexahedral mesh;
2. edges that are the diagonal of a side of a cube in the hexahedral mesh;
3. edges that correspond to the inner diagonal of a cube in the hexahedral mesh.

Edges of the first and third kinds have six adjacent tetrahedra each, and edges of the second kind do have four adjacent tetrahedra each.

4.4. The exterior of the root simplex. When computing neighboring d-simplices it is possible that the neighbor in question does not belong to the root simplex T_d^0 but lies outside of it. If we look at face-neighbors of a d-simplex T, the fact that the considered neighbor lies outside means that the respective face was on the boundary of T_d^0. In order to check whether a computed d-simplex is outside the base simplex, we investigate a more general problem: Given anchor node coordinates $(x_0, y_0)^T \in \mathbb{Z}^2$, respectively, $(x_0, y_0, z_0)^T \in \mathbb{Z}^3$, of type b and level ℓ, decide whether the corresponding d-simplex N lies inside or outside of the root tetrahedron T_d^0: $N \in \mathcal{T}_d^\ell$ or $N \notin \mathcal{T}_d^\ell$. At the end of this section we furthermore generalize this to the problem of deciding for any two d-simplices N and T whether or not N lies outside of T. We solve this problem in constant time and independent of the levels of N and T.

We examine the 3D case. Looking at T_3^0 we observe that two of its boundary faces correspond to faces of the root cube, namely, the intersections of T_3^0 with the $y = 0$ and the $x = 2^\mathcal{L}$ planes. The other two boundary faces of T_3^0 are the intersections with the $x = z$ and the $y = z$ planes. Thus, the boundary of T_3^0 can be described as the intersection of T_3^0 with those planes. We refer to the latter two planes as E_1 and E_2.

Let N be a tetrahedron with anchor node $(x_0, y_0, z_0)^T \in \mathbb{Z}^3$ of type b and level ℓ, and denote with $(x_i, y_i, z_i)^T$ the coordinates of node i of N. Since $(x_i, y_i, z_i)^T \geq (x_0, y_0, z_0)^T$ (componentwise), we directly conclude that if $x_0 \geq 2^\mathcal{L}$ or $y_0 < 0$, then $N \notin \mathcal{T}_3$. Because the outer normal vectors of T_3^0 on the two faces intersecting with E_1 and E_2 are

$$\vec{n}_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} \quad \text{and} \quad \vec{n}_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \tag{50}$$

we also conclude that $N \notin \mathcal{T}_3$ if $z_0 - x_0 > 0$ or $y_0 - z_0 > 0$. Now we have already covered all the cases except those where the anchor node of N lies directly in E_1 or E_2. In these cases we cannot solve the problem by looking at the coordinates of the anchor node alone, since there exist tetrahedra $T' \in \mathcal{T}_3$ with anchor nodes lying in one of these planes (see Figure 10 for an illustration of the analogous case in 2D). This depends on the type of the tetrahedron in question. We observe that a tetrahedron $T' \in \mathcal{T}_3$ with anchor node lying in E_1 can have the types $0, 1$, or 2, and a tetrahedron with anchor node lying in E_2 can have the types $0, 4$, or 5. We conclude that to check whether N is outside of the root tetrahedron we have to check if any one of six conditions is fulfilled. In fact these conditions fit into the general form below with $x_i = x$, $x_j = y$, $x_k = z$, and T as the root tetrahedron; thus $T.x = T.y = T.z = 0$ and $T.b = 0$.

These generalized conditions solve the problem to check for any two given tetrahedra N and T, whether N lies outside of T or not.

PROPOSITION 23. *Given two d-simplices N, T with $N.\ell \geq T.\ell$, then N is outside of the simplex T—which is equivalent to saying that N is no descendant of T—if and*

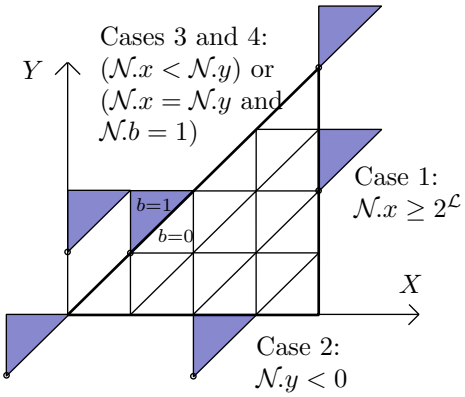


FIG. 10. *A uniform level 2 refinement of the triangle T^0 in 2D and triangles lying outside of it with their anchor nodes marked. When deciding whether a triangle with given anchor node coordinates lies outside of T^0 there are four cases to consider, one for each face of T^0. For the two faces lying parallel to the X-axis, respectively, Y-axis, it suffices to check whether the x-coordinate is greater than or equal to $2^{\mathcal{L}}$, or whether the y-coordinate is smaller than 0. Similarly one can conclude that the triangle lies outside of T^0 if its x-coordinate is smaller than its y-coordinate. If both coordinates agree (and none of the previous cases applies), then the given triangle is outside T^0 if and only if its type is 1.*

only if at least one of the following conditions is fulfilled.

For 2D,

$$N.x_i - T.x_i \geq 2^{T.\ell}, \tag{51a}$$

$$N.x_j - T.x_j < 0, \tag{51b}$$

$$(N.x_j - T.x_j) - (N.x_i - T.x_i) > 0, \tag{51c}$$

$$N.x_i - T.x_i = N.x_j - T.x_j \text{ and } N.b = \begin{cases} 1 & \text{if } T.b = 0, \\ 0 & \text{if } T.b = 1. \end{cases} \tag{51d}$$

For 3D,

$$N.x_i - T.x_i \geq 2^{T.\ell}, \tag{52a}$$

$$N.x_j - T.x_j < 0, \tag{52b}$$

$$(N.x_k - T.x_k) - (N.x_i - T.x_i) > 0, \tag{52c}$$

$$(N.x_j - T.x_j) - (N.x_k - T.x_k) > 0, \tag{52d}$$

$$N.x_k - T.x_k = N.x_i - T.x_i \quad \text{and} \quad N.b \in \begin{cases} \{T.b+1, T.b+2, T.b+3\} & \text{if } T.b \text{ is even}, \\ \{T.b-1, T.b-2, T.b-3\} & \text{if } T.b \text{ is odd}, \end{cases} \tag{52e}$$

$$N.x_j - T.x_j = N.x_k - T.x_k \quad \text{and} \quad N.b \in \begin{cases} \{T.b-1, T.b-2, T.b-3\} & \text{if } T.b \text{ is even}, \\ \{T.b+1, T.b+2, T.b+3\} & \text{if } T.b \text{ is odd}. \end{cases} \tag{52f}$$

The coordinates x_i, x_j, and x_k are chosen in dependence of the type of T according to Table 5.

Proof. By shifting N by $(-T.\vec{x})$ we reduce the problem to checking whether the shifted d-simplex lies outside of a d-simplex with anchor node $\vec{0}$, level $T.\ell$, and type

TABLE 5

Following the general scheme described in this section to compute whether a given d-simplex N lies outside of another given d-simplex T, we give the coordinates x_i, x_j, and x_k in dependence of the type of T.

	$T.b$	
2D	0	1
x_i	x	y
x_j	y	x

	$T.b$					
3D	0	1	2	3	4	5
x_i	x	x	y	y	z	z
x_j	y	z	z	x	x	y
x_k	z	y	x	z	y	x

$T.b$. For $d = 3$ the proof is analogous to the above argument, where we considered the case $b = 0$ and $\ell = 0$. In 2D the situation is even simpler, since there exists only one face of the root triangle that is not a coordinate axis (see Figure 10). ∎

4.5. A consecutive index. In contrast to the Morton index for cubes, the TM-index for d-simplices does not produce a consecutive range of numbers. Therefore, two simplices T and T' of level ℓ that are direct successors/predecessors with respect to the tetrahedral Morton order do not necessarily fulfill $m(T) = m(T') \pm 2^{d(\mathcal{L}-\ell)}$ or $m(T) = m(T') \pm 1$. For $d = 3$ this follows directly from the fact that each b^j occupies three bits, but there are only six values that each b^j can assume, since there are only six different types. In 2D this follows from the fact that not every combination of anchor node coordinates and type can occur for triangles in $\mathcal{T}_2$, the triangle with anchor node $(0, 0)$ and type 1 being one example. This also means that the largest occurring TM-index is bigger than $2^{d\mathcal{L}} - 1$.

Constructing a consecutive index that respects the order given by the TM-index is possible, as we show in this section. Since in practice it is more convenient to work with this consecutive index instead of the TM-index, our aim is to construct for each level ℓ a consecutive index $I(T) \in \{0, \dots, 2^{d\ell} - 1\}$, such that

$$\forall\, T, S \in \mathcal{T}_d^\ell\colon \quad I(T) < I(S) \;\Leftrightarrow\; m(T) < m(S). \tag{53}$$

This index can also be understood as a bijection

$$I\colon \{\, m \in \mathbb{N}_0 \,|\, m = m(T) \text{ for a } T \in \mathcal{T}_d^\ell \,\} \xrightarrow{\cong} \{0, \dots, 2^{d\ell} - 1\}, \tag{54}$$

mapping the TM-indices of level ℓ d-simplices to a consecutive range of numbers. It is obvious that $I(T_d^0) = 0$. The index $I(T)$ can be easily computed as the ℓ-digit 2^d-ary number consisting of the local indices as digits, thus

$$I(T) = (I_{\text{loc}}(T^1), \dots, I_{\text{loc}}(T^\ell))_{2^d}. \tag{55}$$

Algorithm 4.7 shows an implementation of this computation. It can be done directly from the Tet-id of T, and thus it is not necessary to compute the TM-index of T first.

The inverse operation of computing T from $I(T)$ and a given level ℓ can be carried out in a similar fashion; see Algorithm 4.8. For each $0 \le i \le \ell$ we look up the type b and the cube-id of T^i from $I_{\text{loc}}(T^i)$ and the type of $\texttt{Parent}(T^i) = T^{i-1}$ (starting with $\text{type}(T^0) = 0$) via Tables 7 and 8. From the cube-ids we can build up the anchor node coordinates of T. The last computed type is the type of T. The runtime of this algorithm is $\mathcal{O}(T.\ell)$.

Similar to Algorithm 4.7 is Algorithm `Is_valid`, which decides whether a given index $m \in [0, 2^{6\mathcal{L}}) \cap \mathbb{Z}$ is in fact a TM-index for a tetrahedron. Thus, in the spirit of

Algorithm 4.7: $I(\texttt{Tet}T)$

```
I ← 0, b ← T.b
for T.ℓ ≥ i ≥ 1 do
    c ← c-id(T, i)
    I ← I + 8^i I^b_loc(c)        /* See Table 6; multiply with 4^i for 2D */
    b ← Pt(c, b)
return I
```

TABLE 6
The local index of a tetrahedron $T \in \mathcal{T}$ in dependence of its cube-id c and type b.

$I^b_{\mathrm{loc}}(c)$ 2D		cube-id c			
		0	1	2	3
b	0	0	1	1	3
	1	0	2	2	3

$I^b_{\mathrm{loc}}(c)$ 3D		cube-id c							
		0	1	2	3	4	5	6	7
b	0	0	1	1	4	1	4	4	7
	1	0	1	2	5	2	5	4	7
	2	0	2	3	4	1	6	5	7
	3	0	3	1	5	2	4	6	7
	4	0	2	2	6	3	5	5	7
	5	0	3	3	6	3	6	6	7

Algorithm 4.8: $\texttt{T}$(consecutive index I,int ℓ)

```
T.x, T.y, T.z ← 0, b ← 0
for 1 ≤ i ≤ ℓ do
    Get I_loc(T^i) from I                          /* See (55) */
    c ← c-id(T^i), b ← T^i.b                       /* See Tables 7 and 8 */
    if c & 1 then T.x ← T.x + 2^{L−i}
    if c & 2 then T.y ← T.y + 2^{L−i}
    if c & 4 then T.z ← T.z + 2^{L−i}              /* Remove this line for 2D */
T.b ← b
return T
```

TABLE 7
For a tetrahedron $T \in \mathcal{T}$ of local index I_{loc} whose parent P has type $P.b$ we give the cube-id of T.

cube-id(T) 2D		$I_{\mathrm{loc}}(T)$			
		0	1	2	3
$P.b$	0	0	1	1	3
	1	0	2	2	3

cube-id(T) 3D		$I_{\mathrm{loc}}(T)$							
		0	1	2	3	4	5	6	7
$P.b$	0	0	1	1	1	5	5	5	7
	1	0	1	1	1	3	3	3	7
	2	0	2	2	2	3	3	3	7
	3	0	2	2	2	6	6	6	7
	4	0	4	4	4	6	6	6	7
	5	0	4	4	4	5	5	5	7

section 3.5 we can decide whether a given 6D cube is in the image of map (25) that embeds $\mathcal{T}_3$ into the set of 6D subcubes of $[0, 2^{\mathcal{L}}]^6$. The runtime of `Is_valid` is $\mathcal{O}(\mathcal{L})$.

The consecutive index simplifies the relation between the TM-index of a simplex and its position in the SFC. In the special case of a uniform mesh, the consecutive index and the position are identical.

TABLE 8
For a tetrahedron $T \in \mathcal{T}$ of local index I_{loc} whose parent P has type $P.b$ we give the type of T.

$T.b$ 2D		$I_{\text{loc}}(T)$ 0	1	2	3
$P.b$	0	0	0	1	0
	1	1	0	1	1

$T.b$ 3D		$I_{\text{loc}}(T)$ 0	1	2	3	4	5	6	7
$P.b$	0	0	0	4	5	0	1	2	0
	1	1	1	2	3	0	1	5	1
	2	2	0	1	2	2	3	4	2
	3	3	3	4	5	1	2	3	3
	4	4	2	3	4	0	4	5	4
	5	5	0	1	5	3	4	5	5

Algorithm 4.9: `Is_valid`$(m \in [0, 2^{6\mathcal{L}}) \cap \mathbb{Z}, \ell)$

```
1 I ← 0
2 k ← 6(L − i)
3 for ℓ ≥ i ≥ 1 do
4 |   b ← (m_k, m_{k+1}, m_{k+2})_8
5 |   c ← (m_{k+3}, m_{k+4}, m_{k+5})_8
6 |   k ← 6(L − i + 1)
7 |   if (m_k, m_{k+1}, m_{k+2})_8 ≠ Pt(c, b) then      /* Take (0,0,0)_8 if i = 1 */
8 |   |   return False
9 return True
```

4.6. Successor and predecessor. Calculating the TM-index corresponding to a particular consecutive index is occasionally needed in higher-level algorithms. This is relatively expensive, since it involves a loop over all refinement levels, thus some 10 to 30 in extreme cases. However, often the task is to compute a whole range of d-simplices. This occurs, for example, when creating an initial uniform refinement of a given mesh (see Algorithm `New` in section 5.1). That is, for a given consecutive index I, a level ℓ, and a count n, find the n level-ℓ simplices following the d-simplex corresponding to the consecutive index I, that is, the d-simplices corresponding to the n consecutive indices $I, I+1, \ldots, I+n-1$. Ideally, this operation should run linearly in n, independent of ℓ, but if we used Algorithm 4.8 to create each of the $n+1$ simplices, we would have a runtime of $\mathcal{O}(n\mathcal{L})$. In order to achieve the desired linear runtime we introduce the operations `Successor` and `Predecessor` that run in average $\mathcal{O}(1)$ time. These operations compute from a given d-simplex T of level ℓ with consecutive index I_T the d-simplex T' whose consecutive index is $I_T + 1$, respectively, $I_T - 1$. Thus, T' is the next level ℓ simplex in the SFC after T (resp., the previous one). Algorithm 4.10, which we introduce to solve this problem, does not require knowledge about the consecutive indices I_T and $I_T \pm 1$ and can be computed significantly faster than Algorithm 4.8; see Lemma 24.

To compute the predecessor of T we only need to reverse the sign in line 3 in the `Successor_recursion` subroutine of Algorithm 4.10.

LEMMA 24. *Algorithm* 4.10 *has constant average runtime (independent of $\mathcal{L}$).*

Proof. Because each operation in the algorithm can be executed in constant time, the average runtime is nc, where c is a constant (independent of $\mathcal{L}$) and $n-1$ is the number of average recursion steps. Since in consecutive calls to the algorithm the variable i cycles through 0 to $2^d - 1$, we conclude that the recursion is on average

Algorithm 4.10: `Successor`(`Tet` T)

```
1  return Successor_recursion(T, T, T.ℓ)

   Function Successor_recursion(Tet T, Tet T′, int ℓ)
1  c ← c-id(T, ℓ)
2  From c and b look up i := I_loc(T^ℓ)                    /* See Table 6 */
3  i ← (i + 1) % 8
4  if i = 0 then                          /* Enter recursion (in rare cases) */
5  |  T′ ← Successor_recursion (T, T′, ℓ − 1)
6  |  b̂ ← T′.b                              /* b̂ stores the type of T′^{ℓ−1} */
7  else
8  |  b̂ ← Pt(c, b)
9  From b̂ and I_loc = i look up (c′, b′)                   /* See Tables 7 and 8 */
10 Set the level ℓ entries of T′.x, T′.y and T′.z to c′
11 T′.b ← b′
12 return T′
```

executed in every 2^dth step, allowing for a geometric series argument. ∎

We see in Algorithm 4.10 the usefulness of the consecutive index. Because we are using this index instead of the TM-index, computing the index of the successor/predecessor only requires adding/subtracting 1 to the given index. On the other hand, computing the TM-index of a successor/predecessor would involve more subtle computations.

5. High-level AMR algorithms. To develop the complete AMR functionality required by numerical applications, we aim at a forest of quad-/octrees in the spirit of [16, 28]. Key top-level algorithms are as follows:

- `New`. Given an input mesh of conforming simplices, each considered a root simplex, generate an initial refinement.
- `Adapt`. Adapt (refine and coarsen) a mesh according to a given criterion.
- `Partition`. Partition a mesh among all processes such that the load is balanced, possibly according to weights.
- `Balance`. Establish a 2:1 size condition between neighbors in a given refined mesh. The levels of any two neighboring simplices must differ by at most 1.
- `Ghost`. For each process, assemble the layer of directly neighboring elements owned by other processes.
- `Iterate`. Iterate through the local mesh, executing a callback function on each element and on all interelement interfaces.

Since partitioning via SFC only uses the SFC index as information, we refer to already existing descriptions of `Partition` for hexahedral or simplicial SFCs [40, 16]. `Balance`, `Ghost`, and `Iterate` are sophisticated parallel algorithms and require additional theoretical work, which is beyond the scope of this paper.

Here, we briefly describe `New` and `Adapt`. In the forest-of-trees approach we model an adaptive mesh by a coarse mesh of level 0 d-simplices, the *trees*. Such a coarse mesh could be specified manually for simple geometries, or obtained from executing a mesh generator. Each level 0 simplex is identified with the root simplex T_d^0 and then refined adaptively to produce the fine and potentially nonconforming mesh of

d-simplices. These simplices are partitioned among all processes; thus each process holds a range of trees, of which the first and last may be incomplete: Their leaves are divided between multiple processes.

An entity $\mathcal{F}$ of the structure `forest` consists of the following entries:

- $\mathcal{F}.C$ — the coarse mesh;
- $\mathcal{F}.\mathcal{K}$ — the process-local trees;
- $\mathcal{F}.\mathcal{E}_k$ — for each local tree k the list of process-local simplices in tetrahedral Morton order.

We acknowledge that `New` and `Adapt` are essentially communication-free, but still serve well to exercise some of the fundamental algorithms described earlier.

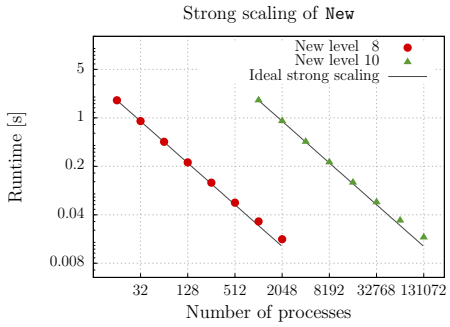
5.1. New. The `New` algorithm creates a partitioned uniform level ℓ refined forest from a given coarse mesh. To achieve this, we first compute the first and last d-simplices belonging to the current process p. From this range we can calculate which trees belong to p and, for each of these trees, the consecutive index of the first and last d-simplices on this tree. We then create the first simplex in a tree by a call to `T` (Algorithm 4.8). In contrast to the `New` algorithm in [16] we create the remaining simplices by calls to `Successor` instead of `T` to avoid the $\mathcal{O}(\ell)$ runtime of `T` in the case of simplices. Our numerical tests, displayed in Figure 11, show that the runtime of `New` is in fact linear in the number of elements and does not depend on the level ℓ. Within the algorithm, K denotes the number of trees in the coarse mesh and P the number of processes.

Algorithm 5.1: `New`(`Coarse Mesh` C, int ℓ)

```
1  n ← 2^{dℓ}, N ← nK ;        /* d-simplices per tree and global number of
     d-simplices */
2  g_first ← ⌊Np/P⌋, g_last ← ⌊N(p+1)/P⌋ − 1 ;   /* Global numbers of first
     and last..  */
3  k_first ← ⌊g_first/n⌋ , k_last ← ⌊g_last/n⌋;   /* ..local simplex and local tree
     range */
4  for t ∈ { k_first, ..., k_last } do
5  |  e_first ← (t = k_first) ? g_first − nt : 0;
6  |  e_last ← (t = k_last) ? g_last − nt : n − 1;
7  |  T ←T (e_first, ℓ);                          /* Call Algorithm 4.8 */
8  |  E_k ← { T };
9  |  for e ∈ { e_first, ..., e_last − 1 } do
10 |  |  T ← Successor (T);
11 |  |  E_k ← E_k ∪ { T }
```

After `New` returns, the process local number of elements is known, and per-element data can be allocated linearly in an array of structures, or a structure of arrays, depending on the specifics of the application.

5.2. Adapt. The `Adapt` algorithm modifies an existing forest by refining and coarsening the d-simplices of a given forest according to a callback function. It does this by traversing the d-simplices of each tree in tetrahedral Morton order and passing them to the callback function. If the current d-simplex and its $2^d - 1$ successors form a family (all having the same parent), then the whole family is passed to the callback.



Runtime tests for `New`

#Procs	Level	#Tetrahedra	Runtime [s]	Factor
64	7	1.073e9	0.059	–
	8	8.590e9	0.451	7.64
	9	6.872e10	3.58	7.94
	10	5.498e11	28.6	7.99
256	8	8.590e9	0.116	–
	9	6.872e10	0.898	7.74
	10	5.498e11	7.15	7.96

FIG. 11. *Runtime tests for* `New` *on JUQUEEN. Left: Two strong scaling studies. A new uniform level* 8 *(circles) and level* 10 *(triangles) refinement of a coarse mesh of* 512 *root tetrahedra, carried out with* 16 *up to* 2,048 *processes and* 1,024 *up to* 131,072 *processes with* 16 *processes per compute node. Right: The data shows that the runtime of* `New` *is linear in the number of generated elements and does not additionally depend on the level. The uniform refinement is created from a coarse mesh of* 512 *root tetrahedra. For the first computation on* 64 *processes we use* 1 *process per compute node, and for the computation on* 256 *processes we use* 2 *processes per node.*

This callback function accepts either one or 2^d d-simplices as input plus the index of the current tree. In both cases, a return value greater than zero means that the first input d-simplex should be refined, and thus its 2^d children are added in tetrahedral Morton order to the new forest. Additionally, if the input consists of 2^d simplices, they form a family, and a return value smaller than zero means that this family should be coarsened, thus replaced by their parent. If the callback function returns zero, the first given d-simplex remains unchanged and is added to the new forest, and `Adapt` continues with the next d-simplex in the current tree. The `Adapt` algorithm creates a new forest from the given one and can handle recursive refinement/coarsening. For the recursive part we make use of the following reasonable assumptions:

- A d-simplex that was created in a refine step will not be coarsened during the same adapt call.
- A d-simplex that was created in a coarsening step will not be refined during the same adapt call.

From these assumptions we conclude that for recursive refinement we only have to consider those d-simplices that were created in a previous refinement step and that we only have to care about recursive coarsening directly after we processed a d-simplex that was not refined and could be the last d-simplex in a family. If refinement and coarsening are not done recursively, the runtime of `Adapt` is linear in the number of d-simplices of the given forest.

An application will generally project or otherwise transform data from the previous to the adapted mesh. This can be done within the adaptation callback, which is known to proceed linearly through the local elements, or after `Adapt` returns if a copy of the old mesh has been retained. In the latter case, one would allocate element data for the adapted mesh and then iterate over the old and the new data simultaneously, performing the projection in the order of the SFC. Once this is done, the old data and the previous mesh are deallocated [15].

6. Performance evaluation. Given the design of the algorithms discussed in this paper, we expect runtimes that are precisely proportional to the number of elements and independent of the level of refinement. To verify this, we present scal-

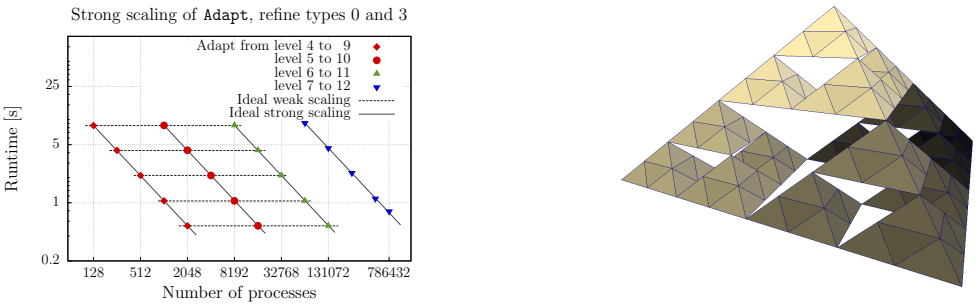


FIG. 12. *Strong scaling for* `Adapt` *with a fractal refinement pattern. Starting from an initial level* k *on a coarse mesh of* 512 *tetrahedra we refine recursively to a maximal final level* $k + 5$. *The refinement callback is such that only subtetrahedra of types* 0 *and* 3 *are refined. Left: Strong and weak scaling on JUQUEEN with up to* 131,072 *processes and strong scaling on MIRA with up to* 786,432 *processes. On both systems we use* 16 *processes per compute node. The level* 12 *mesh consists of* 858,588,635,136 *tetrahedra. Right: An initial level* 0 *and final level* 3 *refinement according to the fractal pattern. The subtetrahedra of levels* 1 *and* 2 *are transparent.*

ing and runtime tests[1] for `New` and `Adapt` on the JUQUEEN supercomputer at the Forschungszentrum Juelich [19], an IBM BlueGene/Q system with 28,672 nodes consisting of 16 IBM PowerPC A2 @ 1.6 GHz and 16 GB Ram per node. We also present one runtime study on the full MIRA system at the Argonne Leadership Computing Facility, which has the same architecture as JUQUEEN and 49,152 nodes. The biggest occurring number of mesh elements is around 8.5×10^{11} tetrahedra with 13 million elements per process.

The first two tests are a strong scaling (up to 131k processes) and a runtime study of `New` in 3D, shown in Figure 11. For both tests we use a coarse mesh of 512 tetrahedra. We time the `New` algorithm with input level 8 (resp., level 10 for higher numbers of processes). We execute the runtime study to examine whether `New` has the proposed level-independent linear runtime in the number of generated tetrahedra, which can be read from the results presented in the table in Figure 11.

The last test is `Adapt` with a recursive nonuniform refinement pattern. The starting point for all runs is a mesh obtained by uniformly refining a coarse mesh of 512 tetrahedra to a given initial level k. This mesh is then refined recursively using a single `Adapt` call, where only the tetrahedra of types 0 and 3 whose level does not exceed the fine level $k + 5$ are refined recursively. The resulting mesh on each tetrahedron resembles a fractal pattern similar to the Sierpinski tetrahedron. We perform several strong and weak scaling runs on JUQUEEN starting with 128 processes and scaling up to 131,072. The setting is 16 processes per compute node. We finally do another strong scaling run on the full system of the MIRA supercomputer at the Argonne Leadership Computing Facility with 786,432 processes and again 16 processes per compute node. Figure 12 shows our runtime results.

7. Conclusion. We present a new encoding for adaptive nonconforming triangular and tetrahedral mesh refinement based on Bey's red-refinement rule. We identify six different types of tetrahedra (and two types of triangles) and prescribe an ordering of the children for each of these types that differs from Bey's original order. By introducing an embedding of the mesh elements into a Cartesian coordinate structure, we define a tetrahedral Morton index that can be computed using bitwise interleaving

[1] Version `v0.1` is available at https://bitbucket.org/cburstedde/t8code.git.

similar to the Morton index for cubes. This tetrahedral Morton index shares some properties with the well-known cubical one and allows for a memory-efficient random access storage of the mesh elements.

Exploiting the Cartesian coordinate structure, we develop several constant-time algorithms on simplices. These include computing the parent, the children, and the face-neighbors of a given mesh element, as well as computing the next and previous elements according to the SFC.

In view of providing a complete suite of parallel dynamic AMR capabilities, the constructions and algorithms described in this paper are just the beginning. A repartitioning algorithm following our SFC, for example, is easy to imagine, but challenging to implement if the tree connectivity is to be partitioned dynamically, and if global shared metadata shall be reduced from being proportional to the number of ranks to the number of compute nodes. The present paper provides atomic building blocks that can be used in high-level algorithms for 2:1 balancing [27] and the computation of ghost elements and generalized topology iteration [28]; regardless, these algorithms still have to be written and are likely to raise questions that we will have to address in future work. Notwithstanding the above, we believe that the choices presented in this paper are sustainable for maintaining extreme scalability in the long term.

Acknowledgments. We would like to thank Tobin Isaac (University of Chicago) for his thoughts on software interfacing.

We also 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 [19] at the Jülich Supercomputing Centre (JSC). GCS is the alliance of the three national supercomputing centers HLRS (Universität Stuttgart), JSC (Forschungszentrum Jülich), and LRZ (Bayerische Akademie der Wissenschaften), funded by the German Federal Ministry of Education and Research (BMBF) and the German State Ministries for Research of Baden-Württemberg (MWK), Bayern (StMWFK), and Nordrhein-Westfalen (MIWF).

We are indebted to two anonymous reviewers for their careful reading of the manuscript and their thoughtful and constructive comments.

REFERENCES

[1] V. Akçelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. R. O'Hallaron, T. Tu, and J. Urbanic, *High resolution forward and inverse earthquake modeling on terascale computers*, in SC03: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM/IEEE, 2003.

[2] F. Alauzet and A. Loseille, *On the use of space filling curves for parallel anisotropic mesh adaptation*, in Proceedings of the 18th International Meshing Roundtable, B. W. Clark, ed., Springer, 2009, pp. 337–357, doi:10.1007/978-3-642-04319-2_20.

[3] I. Babuška and W. C. Rheinboldt, *Error estimates for adaptive finite element computations*, SIAM J. Numer. Anal., 15 (1978), pp. 736–754, doi:10.1137/0715049.

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

[5] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, *Algorithms and data structures for massively parallel generic adaptive finite element codes*, ACM Trans. Math. Software, 38 (2011), pp. 14:1–14:28.

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

[7] R. E. Bank, A. H. Sherman, and A. Weiser, *Some refinement algorithms and data structures*

for regular local mesh refinement, in Scientific Computing, Applications of Mathematics and Computing to the Physical Sciences, Vol. 1, R. S. Stapleman, ed., IMACS/North-Holland, 1983.

[8] E. Bänsch, *Local mesh refinement in* 2 *and* 3 *dimensions*, IMPACT Comput. Sci. Engrg., 3 (1991), pp. 181–191, doi:10.1016/0899-8248(91)90006-G.

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

[10] J. Bey, *Simplicial grid refinement: On Freudenthal's algorithm and the optimal number of congruence classes*, Numer. Math., 85 (2000), pp. 1–29, doi:10.1007/s002110050475.

[11] K. Brix, R. Massjung, and A. Voss, *Refinement and connectivity algorithms for adaptive discontinuous Galerkin methods*, SIAM J. Sci. Comput., 33 (2011), pp. 66–101, doi:10.1137/090767418.

[12] T. N. Bui and C. Jones, *Finding good approximate vertex and edge partitions is NP-hard*, Inform. Process. Lett., 42 (1992), pp. 153–159, doi:10.1016/0020-0190(92)90140-Q.

[13] H.-J. Bungartz, M. Mehl, and T. Weinzierl, *A parallel adaptive Cartesian PDE solver using space-filling curves*, in Euro-Par 2006 Parallel Processing, Springer-Verlag, New York, 2006, pp. 1064–1074.

[14] 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, ACM/IEEE, 2010.

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

[16] 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, doi:10.1137/100791634.

[17] C. Carstensen, *An adaptive mesh-refining algorithm allowing for an H^1 stable L^2 projection onto Courant finite element spaces*, Constr. Approx., 20 (2004), pp. 549–564.

[18] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, *Hypergraph-based dynamic load balancing for adaptive scientific computations*, in Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS'07), IEEE, 2007.

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

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

[21] W. Dörfler, *A convergent adaptive algorithm for Poisson's equation*, SIAM J. Numer. Anal., 33 (1996), pp. 1106–1124, doi:10.1137/0733054.

[22] P. F. Fischer, G. W. Kruse, and F. Loth, *Spectral element methods for transitional flows in complex geometries*, J. Sci. Comput., 17 (2002), pp. 81–98, doi:10.1023/A:1015188211796.

[23] N. Golias and R. Dutton, *Delaunay triangulation and* 3*D adaptive mesh generation*, Finite Elem. Anal. Des., 25 (1997), pp. 331–341, Adaptive Meshing, Part 2, doi:10.1016/S0168-874X(96)00054-6.

[24] H. Haverkort and F. van Walderveen, *Locality and bounding-box quality of two-dimensional space-filling curves*, Comput. Geom., 43 (2010), pp. 131–174.

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

[26] Ø. Hjelle and M. Dæhlen, *Triangulations and Applications*, Math. Visualization, Springer, New York, 2006, http://www.springer.com/us/book/9783540332602.

[27] T. Isaac, C. Burstedde, and O. Ghattas, *Low-cost parallel algorithms for* 2:1 *octree balance*, in Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium, IEEE, 2012, doi:10.1109/IPDPS.2012.47.

[28] 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, doi:10.1137/140970963.

[29] 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.

[30] D. A. Kopriva, S. L. Woodruff, and M. Y. Hussaini, *Computation of electromagnetic scattering with a non-conforming discontinuous spectral element method*, Internat. J. Numer. Methods Engrg., 53 (2002), pp. 105–122, doi:10.1002/nme.394.

[31] I. Kossaczkỳ, *A recursive approach to local mesh refinement in two and three dimensions*, J. Comput. Appl. Math., 55 (1994), pp. 275–288, doi:10.1016/0377-0427(94)90034-5.

[32] M. Lee, L. De Floriani, and H. Samet, *Constant-time neighbor finding in hierarchical tetra-*

hedral meshes, in SMI 2001 International Conference on Shape Modeling and Applications, 2001, pp. 286–295, doi:10.1109/SMA.2001.923400.

[33] A. Logg, K.-A. Mardal, and G. N. Wells, eds., *Automated Solution of Differential Equations by the Finite Element Method*, Lect. Notes Comput. Sci. Eng. 84, Springer, New York, 2012, doi:10.1007/978-3-642-23099-8.

[34] G. M. Morton, *A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*, tech. report, IBM Ltd., Ottawa, ON, Canada, 1966.

[35] L. Oliker and R. Biswas, *PLUM: Parallel load balancing for adaptive unstructured meshes*, J. Parallel Distrib. Comput., 52 (1998), pp. 150–177, doi:10.1006/jpdc.1998.1469.

[36] L. Oliker, R. Biswas, and H. N. Gabow, *Parallel tetrahedral mesh adaptation with dynamic load balancing*, Parallel Comput., 26 (2000), pp. 1583–1608, doi:10.1016/S0167-8191(00)00047-8.

[37] OpenCFD, *OpenFOAM – the Open Source CFD Toolbox – User's Guide*, 1.4 ed., OpenCFD Ltd., United Kingdom, 2007.

[38] E. J. Otoo and H. Zhu, *Indexing on spherical surfaces using semi-quadcodes*, in Advances in Spatial Databases, Lecture Notes in Comput. Sci. 692, D. Abel and B. Chin Ooi, eds., Springer, Berlin, 1993, pp. 510–529, doi:10.1007/3-540-56869-7_29.

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

[40] J. R. Pilkington and S. B. Baden, *Partitioning with Spacefilling Curves*, tech. report, Dept. of Computer Science and Engineering, University of California, San Diego, 1994.

[41] 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 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, 2010, pp. 1–11.

[42] 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. Engrg., 16 (2014), pp. 13–21, doi:10.1109/MCSE.2014.75.

[43] 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, doi:10.1145/355887.355891.

[44] H. Sagan, *Space-Filling Curves*, Springer, New York, 1994.

[45] J. Shewchuk, *Delaunay Refinement Mesh Generation*, Ph.D. thesis, CS tech. report CMU-CS-97-137. Carnegie Mellon University, Pittsburgh, 1997,

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

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

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

[49] 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.

[50] 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.

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

[52] 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, doi:10.1137/070681727.

[53] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst, *MOAB: A Mesh-Oriented Database*, tech. report SAND2004-1592, Sandia National Laboratories, Albuquerque, NM, 2004.

[54] 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, doi:10.1109/SC.2005.61.

[55] 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, doi:10.1137/100799071.

[56] S. Zhang, *Successive subdivisions of tetrahedra and multigrid methods on tetrahedral meshes*, Houston J. Math., 21 (1995), pp. 541–556.

[57] G. W. Zumbusch, *Load balancing for adaptively refined grids*, Proc. Appl. Math. Mech., 1 (2002), pp. 534-537.