Data structures and algorithms for high-dimensional structured adaptive mesh refinement

Magnus Grandin

Uppsala University
magnus.grandin@it.uu.se

Abstract. Spatial discretization of high-dimensional partial differential equations requires data representations that are of low overhead in terms of memory and complexity. Uniform discretization of computational domains quickly grows out of reach due to an exponential increase in problem size with dimensionality. Even with spatial adaptivity, the number of mesh data points can be unnecesarily large if care is not taken as to where refinement is done. We propose an adaptive scheme that generates the mesh by recursive bisection, allowing mesh blocks to be arbitrarily anisotropic to allow for fine structures in some directions without over-refining in those directions that suffice with less refinement. We describe in detail how the mesh blocks are organized in a kdtree and the algorithms that update the mesh as is necessary for preserved accuracy in the solution. Algorithms for refinement, coarsening and 2:1 balancing of a mesh hierarchy are derived, and we describe how information is retrieved from the tree structure by means of binary search. To show the capabilities of our framework, we present results showing examples of generated meshes and evaluate the algorithmic scalability on a suite of test problems. In summary, we conclude that although the worst-case complexity of sorting the nodes and building the node map index is n^2, the average runtime scaling in our examples is no worse than $n \log n$.

1 Introduction

Structured adaptive mesh refinement (SAMR) is an active area of research within the scientific computing community [18]. By adjusting the resolution of the computational mesh dynamically to features in the solution or the computational domain, widely varying scales of resolution can be represented simultaneously. In effect, the computational efficiency of a simulation is improved, possibly by orders of magnitude, allowing for larger computations and/or shorter execution times due to a reduction in the number of gridpoints [4]. For problems in two and three dimensions, there are efficient algorithms and data structures available, relying on quad/octrees for structuring the mesh blocks [4, 12, 22, 10]. However, extending quadtrees and octrees to higher dimensional trees is problematic since they yield a fan-out of $2^{\mathcal{D}}$ nodes at every branch, which leads to an exponential increase in the potential number of tree nodes to handle. The contribution of this paper is a framework capable of generating and propagating meshes of arbitrary dimensionality. Our approach is based on recursive bisection and generates far fewer mesh nodes compared to $2^{\mathcal{D}}$-trees of corresponding refinement.

In order to construct a practical refinement scheme that works well even in higher dimensions, we build our framework on a structured block-based refinement strategy [18]

allowing blocks to be refined *anisotropically* and maintain the mesh nodes and their mutual relationships in a *kdtree* [3]. With anisotropic refinement, a block is not restricted to be refined equally in all dimensions, potentially leading to a more efficient discretization in terms of the number of created blocks since mesh blocks are refined only in the dimensions in which they would benefit from finer resolution. We refine the grid successively by dividing blocks in half, dimension by dimension. If a block needs refinement in more than one dimension, this is done by subsequent division in several steps. In this paper, we do not consider the details on error estimation and how to determine when refinement/coarsening is required. For the anisotropic refinement strategy to be useful though, the error estimator must be able to detect the discretization error per dimension. An example of such an error estimator is given in [16].

A kdtree is a binary tree representation of a hierarchical subdivision of a $\mathcal{D}$-dimensional hyperrectangle by recursive bisection [3, 8, 9, 11, 24]. The interior nodes of a kdtree represent hyperplane cuts, aligned with the cartesian coordinate axes, and the leaf nodes contain the actual data. In a general kdtree, a cut can be placed anywhere along the split dimension of a block. However, by restricting the cuts to always be placed in the middle of the block (which we do by imposing a halving of the blocks on each refinement) the scheme is simplified significantly. Furthermore, implementations of kdtrees usually assign split dimensions to nodes cyclically, such that a node at tree level l is split in dimension (l mod $\mathcal{D}$). In our implementation, we relax this restriction and allow nodes to be split arbitrarily without any intermediate refinement in the other dimensions. Thus, blocks can become as elongated as is needed and in principle there is no restriction on the aspect ratio of the blocks.

It is often motivated for reasons of efficiency and memory requirements that a pointer representation of a tree structure should be avoided. By storing locality information in each mesh node and structuring the nodes in a linear order according to this information, the internal structure of a tree is available implicitly and no pointers are needed for searching and navigating it [4, 5, 12, 22]. With the leaf nodes stored in linear order (e.g. the Morton order space-filling curve [15]), tree search is replaced by binary search, which is further advantageous in terms of search complexity; a tree representing an adaptively refined mesh is potentially very unbalanced with a search complexity approaching $\mathcal{O}(n)$ in the number of leaf nodes, whereas binary search in the linear representation is always $\mathcal{O}(\log_2 n)$ [22]. We follow the approach taken in previous work by other authors [4, 22] and build linear Morton-order trees, generalizing the data structures and algorithms to enable an extension to higher dimensionalities. However, the anisotropic node refinement does not map directly to an efficient consecutive order of elements solely from the locality information in the leaf nodes. In order to alleviate this, we extend the linear tree representation with a lightweight representation of the internal node structure (a *node map index*), corresponding to a *hierarchical* Morton order of nodes. The node map index is constructed at the same time as the nodes are sorted and stored in a compact array representation for efficient traversal.

The remainder of this paper is organized as follows. In Section 2, we discuss some related work. A key concept in our implementation is the notion of location codes, which

we describe in Section 3. In Section 4 we derive the construction of a hierarchical Morton order index together with some implementation details. The algorithms for tree search are discussed in Section 5, followed by the adaptive mesh algorithms in Section 6. Finally, Section 7 gives some results of our implementation and Section 8 concludes the paper.

2 Related Work

To our knowledge, there are no dynamically adaptive frameworks available that supports anisotropic SAMR on hyper-rectangular domains in $\mathcal{D}$ dimensions. Klöfkorn and Nolte described the implementation of the `SPGrid` interface of the DUNE framework [17, 14], which allows anisotropic structured grids of higher dimensionality but only handles static meshes. An alternative approach to tackling higher dimensions is to discretize the domain into structured simplex meshes [2, 7, 19, 23]. In [24], a two-level approach of combining hypercubes and simplices is presented.

The most typical uses of kdtrees are within domain decomposition for clustering of scattered point data, nearest neighbor search and raytracing, although they were originally developed for efficient searching in multidimensional data bases [3]. It is straightforward to adopt the kdtree structure to any form of domain decomposition problem. In this paper we show how kdtrees can be used for adaptive mesh refinement by subdividing the computational domain until every block in the domain constitutes a fine enough grid resolution for the error in that part of the grid to be below some tolerance. Although presented in this setting, our techniques can potentially be useful in more general kdtrees and for other applications.

Samet and Tamminen [20, 21] developed a tree structure that is closely related to the kdtree, which they refer to as the binary image tree (bintree). It is a pointerless tree structure developed for connected component labeling [21] of d-dimensional images, connecting pixels of an image into a hierarchy of larger objects. The tree structure representation of the pointerless bintree is similar to the way we represent our trees, and their algorithms for connected component labeling share some features with our search algorithms.

We have chosen the Morton order space-filling curve for linear indexing of mesh elements, due to its simplicity and direct correspondence to recursive bisection and binary trees. Other alternatives have better locality properties, e.g. the Hilbert-curve and the Peano-curve [1], but are more complex to compute and not straightforward to use with the anisotropic refinement we present in this paper.

3 Location Codes

A mesh block represents a hyper rectangle (orthotope) subdomain of the discretized $\mathcal{D}$-dimensional space. The *location code* [4, 22] of an orthotope is a unique identifier that encodes the location and refinement of the corresponding mesh element. The location code has two components; (1) a coordinate in space with respect to the orthotope anchor, and

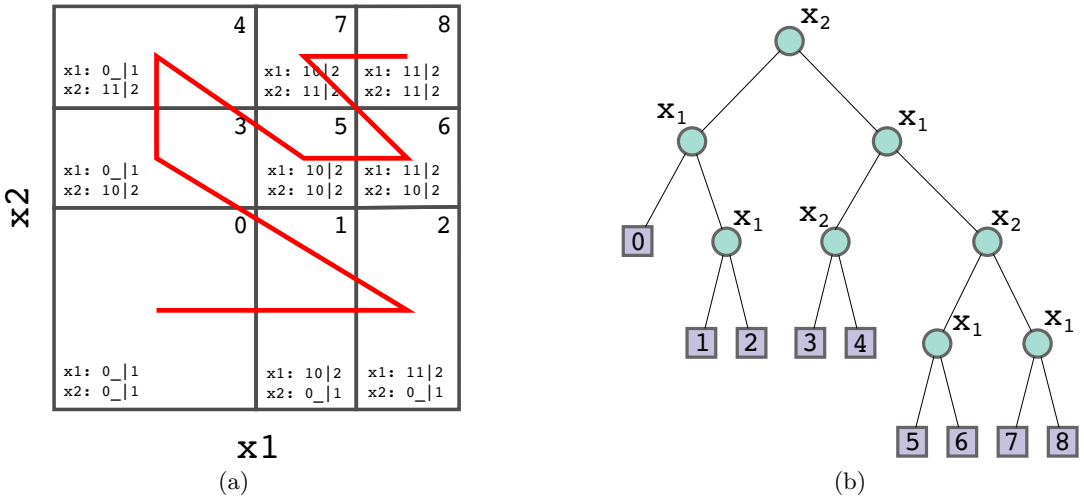


Fig. 1. Anisotropic mesh with coordinates and refinement levels indicated for each node. (a) The linear Morton order of the nodes is indicated in red, starting in the lower left corner. (b) Corresponding node hierarchy, internal nodes specifying the split dimension at each level.

(2) refinement level information (per dimension). The anchor of an orthotope is chosen by convention to be the leftmost/lowermost corner in every direction, given by $\mathcal{D}$ integer coordinates. Figure 1 depicts an example node hierarchy and the corresponding mesh with location codes indicated for each mesh node. The integer representation of coordinates directly correspond to the orthotope's location in the unit hypercube and allows for efficient bitwise integer operations. The actual coordinates of mesh blocks in the domain can be obtained by scaling the unit hypercube coordinates to the domain boundaries. Due to the anisoptropic refinement, we also need to store separate refinement levels in each dimension ($\mathcal{D}$ integers). This is a generalization of the location codes used with linear quad/octrees, which only have one integer component for the refinement level due to their isotropic nature.

The coordinates of an orthotope are constructed and stored as follows [4]. Each coordinate is represented by $\mathcal{B}$ bits, for a total of $\mathcal{D}\mathcal{B}$ bits for the entire coordinate set of an orthotope. Subdividing an orthotope at level l in dimension x_i updates the lth bit in the ith coordinate accordingly. The leftmost bit of a coordinate is the most significant and corresponds to the first subdivision in the corresponding dimension, the second leftmost bit corresponds to the second subdivision, and so forth. The bits that are below the refinement level of an orthotope are set to 0. Thus, the integer size $\mathcal{B}$ relates to the maximum level of refinement in any dimension and requires only $\lceil \log_2 \mathcal{B} \rceil$ bits per dimension for storage of the refinement levels. It is straightforward to combine the coordinates and refinement levels in a single integer per dimension and given 32-bit integers for the storage; this still allows for 27 levels of refinement per dimension to be represented.

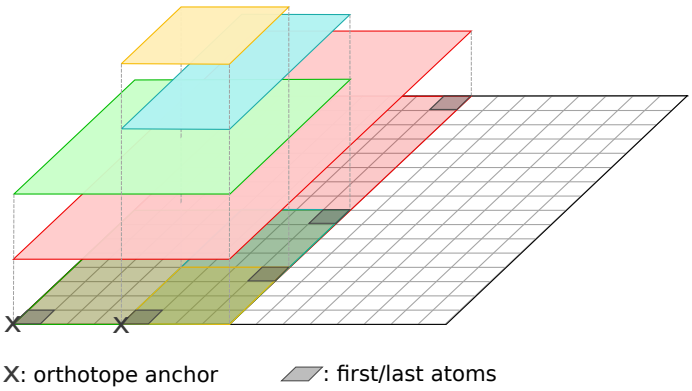


Fig. 2. Hierarchy of orthotopes on top of bottom-level grid of atoms. Further indicated are the anchors and the first/last atoms of each orthotope.

3.1 Atoms

For some operations on the tree it is necessary to reason about the smallest possible domain units in the mesh. We refer to these components as atoms (also referred to as smallest or least descendants by other authors [4, 22]). By definition, an atom is of refinement level $\mathcal{B}$ in all dimensions. The domain and all its subcomponents can be viewed as overlying a fine grid of atoms, such that an orthotope is spanned by its first and last atoms (i.e. the first and last atoms in the range of linear indices that the orthotope covers). Figure 2 illustrates how an orthotope depends on a hierarchy of coarser orthotopes and how these are mapped onto the grid of atoms. Further indicated in the figure are the anchors and the first and last atoms of each orthotope.

Property 1. Given a fully covered domain with no overlapping mesh elements, any atom in the domain will overlap exactly one mesh element.

3.2 Surfaces

Another important type of orthotopes are surfaces. A surface is an orthotope of maximum refinement level $\mathcal{B}$ in a subset of the dimensions, but not all, such that it stretches out along e.g. an edge or a face. More specifically, an m-face surface (m-surface for short) is an orthotope corresponding to an m-dimensional mesh element in $\mathcal{D}$-dimensional space. Thus, in an m-surface, $\mathcal{D} - m$ of the orthotope's refinement indicators are of maximum level $\mathcal{B}$. Atoms are special cases of surfaces (0-surfaces, or points).

The m-face surfaces of an orthotope describe its corresponding m-dimensional boundaries. As such, they are useful for many purposes, e.g. for obtaining information about an orthotopes neighborhood.

Algorithm 1: `mFaceSurface(orthotope` o`, int` f`[`$\mathcal{D}$`])`

```
1:  q ← o
2:  for d ∈ 0, ..., D − 1 do
3:      if not f[d] = 0 then
4:          h ← 2^(B − o.level[d])
5:          if f[d] > 0 then
6:              q.coords[d] ← o.coords[d] + (h − 1)
7:          end
8:          q.level[d] ← B
9:      end
10: end
11: return q
```

Algorithm 2: `mFaceNeighbor(orthotope` o`, int` f`[`$\mathcal{D}$`])`

```
1:  q ← o
2:  for d ∈ 0, ..., D − 1 do
3:      h ← 2^(B − o.level[d])
4:      if f[d] < 0 then
5:          q.coords[d] ← o.coords[d] − h
6:      else if f[d] > 0 then
7:          q.coords[d] ← o.coords[d] + h
8:      q.level[d] ← o.level[d]
9:  end
10: return q
```

3.3 Operations on Location Codes

Boundary elements Given an orthotope in the mesh, we are often interested in determining its boundaries (corners, edges, faces, etc.). Following the reasoning in Section 3.2, we can derive each boundary element as an m-dimensional surface, $0 \leq m < \mathcal{D}$. To see how an m-surface is obtained from an orthotope's location code, consider Algorithm 1. The surface of orthotope o that we are interested in is indicated by a $\mathcal{D}$-vector f of face directions; in each dimension d there are three possibilities

- $f[d] = 0$: The surface covers o in its full length in d; location code of surface in d (coordinate as well as level) is equal to location code of o.
- $f[d] < 0$: The surface is pushed to the left in d; coordinate of surface in d is equal to coordinate of o, but the level is set to $\mathcal{B}$.
- $f[d] > 0$: The surface is pushed to the right in d; coordinate of surface in d is pushed to the last possible coordinate within o, level is set to $\mathcal{B}$.

Neighboring orthotopes We define a *logical neighbor* of an orthotope o to be an orthotope of identical shape and size, adjacent to o across some shared boundary element.

Algorithm 3: `overlap(orthotope` o_1`, orthotope` o_2`)`

```
1:  r ← True
2:  for d ∈ 0, ..., D − 1 do
3:      m ← min(o1.level[d], o2.level[d])
4:      mask ← (111...11) ≪ (B − m)
5:      if (o1.coords[d] ⊕ o2.coords[d]) & mask then
6:          r ← r & False
7:      else
8:          r ← r & True
9:      end
10: end
11: return r
```

Algorithm 4: `intersection(orthotope` o_1`, orthotope` o_2`)`

```
1:  if not overlap(o1,o2) then
2:      return NIL
3:  end
4:  for d ∈ 0, ..., D − 1 do
5:      m ← max(o1.level[d], o2.level[d])
6:      mask ← (111...11) ≪ (B − m)
7:      q.coords[d] ← (o1.coords[d] | o2.coords[d]) & mask
8:      q.level[d] ← m
9:  end
10: return q
```

A logical neighbor does not necessarily exist in the actual mesh, but can be used as an intermediate search key in order to find the true neighboring nodes.

In order to find the neighbor of o across an m-face, we follow a similar approach to how we find boundary elements (Algorithm 1). The procedure to determine neighbors is formalized in Algorithm 2. The face directions in f are specified as in Algorithm 1, but the location codes are updated in a different manner. Here, the coordinates are updated at the level of o's refinement (determined by h), and the values in the level array are copied since the neighbor is of identical shape.

Overlap and intersection Testing two orthotopes for overlap and finding the orthotope corresponding to the intersection area are key operations in navigation a kdtree. They are listed in Algorithms 3 and 4 respectively. The methodology is similar in both cases: Find a refinement level encapsulating the information we are after, mask all irrelevant bits away, and update the result accordingly. In `overlap`(), we take the minimum level in each dimension and perform a bitwise `XOR` on the corresponding bits. This evaluates the common bits at the coarsest level of refinement; for the nodes to be overlapping we require that all common bits are equal. In `intersection`() on the other hand, we look at

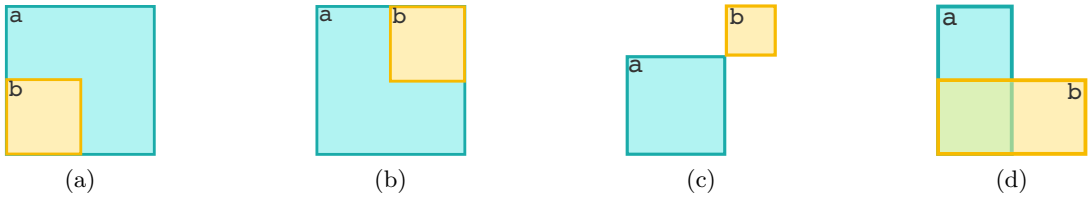


Fig. 3. Examples of how orthotopes are ordered in the linear Morton order. In examples (a) – (c), a precedes b in the linear order; in (a), the coordinates of a and b are the same but the refinement level of a is less than the refinement level of b; in (b) and (c), the coordinates of a precedes the coordinates of b. In (d), the coordinates of a and b are the same but a unique order cannot be determined.

the mutual maximum refinement level instead, since we want to extract the coordinates on the finest common level and update the corresponding coordinate to the highest of the two within the relevant active bits (which we do by a bitwise `OR` on the respective coordinates).

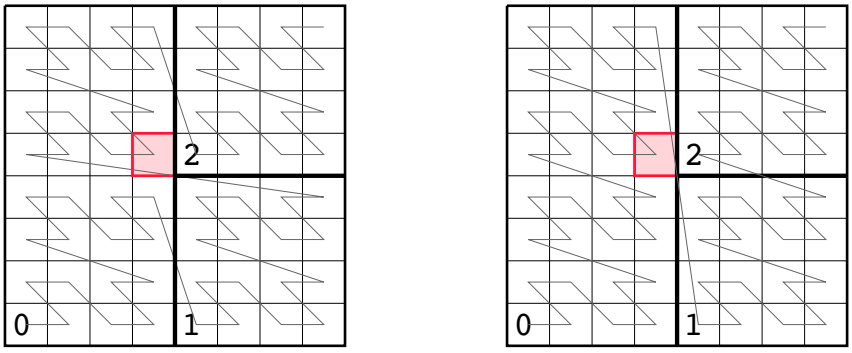
Comparison Two orthotopes are considered equal if and only if their coordinates are equal and they are of equal refinement. If the indices are equal but the refinement differs, then the following rules determine the order (the linear order of nodes is not based directly on the coordinates of each orthotope, but on the hierarchical Morton index; see Section 4)

1. An orthotope precedes all the orthotopes that it completely overlaps.
2. The last atom of an orthotope follows all the orthotope's contained nodes.
3. If neither orthotope completely overlaps the other, their respective order is undefined.

Figure 3 summarizes the properties of the linear order. Note that the ambiguity in condition 3 above is not a problem in practice, since nodes in an actual mesh are required to be non-overlapping.

4 Hierarchical Morton Order

In order to understand why it is problematic to omit the internal node hierarchy in anisotropic domains, consider the example in Figure 4. If there is no information available regarding the hierarchical structure of node subdivisions in the kdtree, all we can get is an ordering based directly on the location code information, corresponding to the Morton ordering in Figure 4(a). Here, the ordering of atoms clearly violates the order of the structure of the mesh blocks, which is an undesirable feature. For instance, consider searching for a mesh block overlapping the element highlighted in red in Figure 4. With the regular Morton ordering, the search could break since it would be led astray by the inconsistency in the element ordering. Instead, we propose a hierarchical Morton ordering based on the internal structure of the tree, where the coarser level structure changes the order in which the bits are interleaved, which would make the linear order correspond to Figure 4(b).



(a) Regular Morton order (b) Hierarchical Morton order

Fig. 4. An anisotropic mesh with 3 blocks, showing the linear order of atoms following different indexing algorithms. In (a), the chain of indices in block 0 is disrupted by block 1. In (b), the ordering has been fixed such that all blocks cover a consecutive sequence of indices.

4.1 Structure of the node index map

The internal node hierarchy (the *node index map*) is stored in a binary tree, as illustrated in Figure 1(b). Given a location code coordinate, we generate its corresponding hierarchical Morton index by traversing the node index map from the root, updating the index bit-by-bit from left to right according to each branch we follow on the path to the corresponding leaf (`0`/`1` for left/right branch). In order to avoid the overhead of random access due to pointer chasing in the tree traversal, we store the tree in an array corresponding to a depth-first traversal of the nodes [21]. The only data we need to store in each node is the dimension of the split it represents. Storing the nodes in depth-first order implies that the left child of a node is the next node in the array whereas the right child of a node is some offset distance away (corresponding to the total number of nodes in its left subtree). Thus, explicitly storing the offset to the right branch, the array representation of the binary tree requires a total storage of $2n$ nodes, with each node holding 2 integers. Although the worst-case complexity of traversing the node index map is still $\mathcal{O}(n)$, it is a sequential data structure that is fast to traverse.

4.2 Sorting the nodes and constructing the node index map

Updating the mesh may reorder the leaf nodes such that they are no longer arranged in proper hierarchical Morton order. Thus, as soon as we update the mesh, we must assure that the sorted order is restored. Furthermore, the node index map must be updated to reflect the changes in the mesh layout. Keeping the node hierarchy up-to-date during mesh modifications is cumbersome and involves complex chains of dependent tree updates. Instead, we propose a combined sort-and-build-map procedure that rebuilds the node map index completely on each mesh update, building the map iteratively during sorting.

The procedure to build and sort the map resembles the structure of in-place Quicksort [6]. The following steps summarize the algorithm:

1. i Find a hyperplane h and an array index i_{split} that partitions the nodes in the array index range $[i_{begin}, i_{end}]$ such that h does not intersect any node.
   ii Rearrange the nodes with respect to h such that all nodes to the left of h are in the index interval $[i_{begin}, i_{split}]$ and all nodes to the right of h are in $(i_{split}, i_{end}]$.
   iii Append the split dimension corresponding to h to the node index map.
   iv Recursively partition and sort the nodes in $[i_{begin}, i_{split}]$.
   v Recursively partition and sort the nodes in $(i_{split}, i_{end}]$.
2. Traverse the nodes in the node index map; examine the total size of the left subtree of each node and update the right child offset accordingly (one pass over the array).

The worst-case complexity of the scheme above is $\mathcal{O}(n^2)$ (just as for Quicksort [6]), a situation that will occur when the kdtree is maximally unbalanced. On average in practical computations we expect the runtime complexity to be much closer to the best-case, which is $\mathcal{O}(n \log_2 n)$ [6]. We further stress that this procedure operates directly on the elements in the leaf node array, rearranging the elements in-place. All comparisons and the handling of hyperplanes are integer bit-operations, and thus of low overhead.

5 Search

The basic algorithms `binarySearchMatching` and `binarySearchOverlapping` perform binary search of the linear kdtree with a location code as search key. We do not list the search routines here since they are straightforward to implement. The search key is translated to a linear index which is used for comparisons in the binary search, whereas the search key itself is used to indicate a search hit. In `binarySearchMatching` an identical match to the search key is looked for, whereas `binarySearchOverlapping` returns the first node that is found that overlaps the search key. When an atom is passed as search key, `binarySearchOverlapping` is guaranteed to find a unique node due to Property 1.

Based on `binarySearchOverlapping` described above, we outline a new search algorithm, `findOverlappingNodes` (listed in Algorithm 5), which collects the full set of mesh nodes overlapping a given location code. It uses binary search with overlap to find all the overlapping nodes available in a (sub)tree by exploration; starting with the first atom of the overlap orthotope the search is expanded in the upward direction in each dimension. Explored nodes that are within the overlap region are put on a stack of nodes to revisit, whereas the nodes that fall outside are ignored, effectively pruning the search. When the stack turns up empty, this indicates completion of the search. This is a similar approach to the active border traversal that is described in [21], with the addition that we limit the search to stay within a given region and discard all nodes that do not at least partially lie within this region.

Algorithm 5: `findOverlappingNodes(kdtree` $\mathcal{T}$`, orthotope` o`)`

```
1:  L ← UniqueSet(∅)                                   ▷ Unique set of overlapping nodes
2:  S ← Stack(∅)                                       ▷ Stack of nodes to visit
3:  f ← firstAtom(o)
4:  i_first ← binarySearchOverlapping(T, f)            ▷ Leaf index of first node in search range
5:  S.push(i_first)                                    ▷ Initialize stack with first node
6:  while not S.isempty() do
7:      i_next = S.pop()
8:      q = T.leaves[i_next].index                     ▷ Location code of next leaf to explore
9:      r = intersection(o, q)                         ▷ Search is guided by intersection
10:     for d ∈ 0, ..., D − 1 do
11:         f = [−1, −1, ..., −1]; f[d] = 1            ▷ Request "first" corner in dimension d
12:         u = mFaceSurface(r, f)                     ▷ Corner atom of intersection
13:         v = nextAtom(u, d)                         ▷ Next atom in dimension d from corner atom
14:         if overlap(o, v) then
15:             k = binarySearchOverlapping(T, v)
16:             L.insert(k)
17:             if k was inserted in L then
18:                 S.push(k)                          ▷ Push leaf index onto stack if not already in set
19:             end
20:         end
21:     end
22: end
23: return L
```

6 Adaptive Mesh Algorithms

This section describes the key algorithms for mesh adaptation; refinement, coarsening and 2:1 balancing. All of these algorithms are of worst-case complexity $\mathcal{O}(n^2)$, where n is the number of leaf nodes in the tree (thus equal to the number of mesh elements). The actual run-time complexity will depend on how unbalanced the refinement tree is and the number of mesh nodes that need to be updated. In practical computations on localized solutions, we expect only a small portion of the mesh nodes to be updated.

In describing these algorithms, we omit the details on how to know when a mesh node should be refined or coarsened. We associate with each mesh node fields `refineFlags` and `coarsenFlags`; these are lists of dimensions in which the corresponding node has been flagged for refinement or coarsening. These flags are assumed to be set by the numerical software from which our framework is used after measuring the error.

6.1 Refinement

Our algorithm for refinement is built upon the refine-procedure of Burstedde *et al.* [4]. In their implementation, an auxilliary by-pass queue is used to perform the refinement in-place with little extra storage. The refinement in our scheme allows for more flexibility, with a

Algorithm 6: `refine(kdtree` $\mathcal{T}$`)`

```
1:  L ← T.leaves
2:  Q ← Deque()                          ▷ Bypass-queue (double-ended) for intermediate storage of nodes
3:  n ← length(L)
4:  i ← 0,  p ← 0
5:  while  i < n + p do
6:      q ← L[i]
7:      Q.pushBack(q)                    ▷ Append node at current position to end of queue
8:      o ← Q.popFront()                 ▷ Pop next node from beginning of queue
9:      for  s ∈ o.refineFlags do
10:         (l, r) ← o.split(s)
11:         Q.pushFront(r)               ▷ Prepend right child node to queue
12:         o ← l                        ▷ Left child node to be treated next
13:         p ← p + 1
14:     end
15:     L[i] ← o
16:     i ← i + 1                        ▷ Propagate point-of-traversal
17: end
18: T.sortAndBuildMap()
```

binary split of nodes and with no restriction in the order of the nodes' split-dimensions. Due to this flexibility, the linear order among the nodes might be broken if implemented straight from [4], and we must extend the scheme such that the linear order is maintained or restored at completion of the refinement step. Our sorting algorithm is described in Section 4.2.

Given a linear kdtree $\mathcal{T}$, `refine()` (Algorithm 6) makes one linear pass over the nodes. In each step of the procedure (lines 5–17), the point of traversal (node index i) is propagated forward and the node at the current position ($L[i]$) is appended to the end of the by-pass queue. The node at the beginning of the queue (the head node) is popped and its refinement flags examined. If refinement is needed, the node is split, and its right child is prepended to the deque. The left node from the split is now the new head node and the pop–refine–prepend sequence is repeated as long as more refinement is needed. When the head node needs no more refinement, it is inserted into the current position of the array. When all nodes have been visited, the array of nodes has increased its size from n to $n + p$.

6.2 Coarsening

As discussed in Section 4.2, we avoid dynamic updates to the node index map. Coarsening in particular is cumbersome in this regard, since merging two nodes that are not children of the same node in the node index map would require a dimension reordering, possibly leading to long chains of updates in the node hierarchy. For these reasons, we restrict coarsening to nodes that are siblings at the leaf level, sharing a common parent node in the node index map. Under some circumstances, this may lead to situations where merges

Algorithm 7: `coarsen(kdtree` $\mathcal{T}$`)`

```
1:  L ← T.leaves
2:  for i ∈ 0, ..., length(L) − 1, do
3:      o ← L[i]
4:      while o needs more merging do
5:          s ← split dimension of o's parent
6:          if o can merge in s and o is left child in dimension s then
7:              q ← sibling of o in dimension s
8:              if q can merge in s and q exists in T then
9:                  p ← merge(o, q, s)
10:                 if merge was successful then
11:                     o ← p                    ▷ Overwritte left (lesser) node with parent
12:                     q ← ∅                                    ▷ Remove right node
13:                 end
14:             end
15:         end
16:     end
17: end
```

between nodes that do not share a common ancestor are hindered, even though they would be feasible with a different ordering of the dimensions. We expect these merges to resolve eventually as the mesh is propagated in time, although suboptimal merging might infer a small runtime performance penalty.

The scheme for coarsening is presented in Algorithm 7. We iterate over the nodes in reverse order, since this gives us the opportunity to perform several subsequent merges on one node before putting it back in the leaf array. The left node in each merge-pair is the node triggering the merge, and upon a successful merge it is replaced by its parent (since the left child and the parent have the same coordinates and Morton index). The right child in the merge pair is removed. Note that we do not need to sort and rebuild the map after coarsening, since this operation does not change the order of the nodes. However, we might want to compress and rebuild the map since coarsening will remove nodes from it, leaving empty space in the map since it is not updated dynamically.

6.3 2:1 balancing

After the mesh has been adjusted according to the refine/coarsen flags, it must be balanced with respect to the 2:1 refinement constraint. We list our balancing procedure in Algorithm 8. Balancing of the refinement is done iteratively on a block-by-block basis. For each block in the mesh, we compare the block's refinement levels against its neighbors'. Any neighbor that is more than twice coarser than the block under introspection is marked for refinement accordingly.

The potential number of neighbors of a block in an unbalanced mesh is extremely large, and searching for all all neighbors of all blocks might become impractical. We resort to the

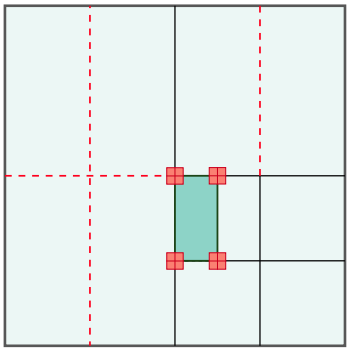


Fig. 5. Illustration of the balancing search strategy in 2 dimensions, enforcing a fully balanced mesh across all m-faces. Highlighted in red are the 2×2 corner boxes of atoms adjacent to each corner of the orthotope to balance. The red dashed lines indicate the refinements that are enforced by the balancing.

much more efficient approach of searching only for those neighbors of a block that are coarser than the block itself [22], which limits the search to a constant number of searches per node. We devise our neighbor search strategy by considering a $2^{\mathcal{D}}$ box of atoms around each corner atom of the node to balance. For a fully balanced mesh across all m-faces (illustrated in Figure 5), the overlapping nodes of all atoms in the corner box are searched for (this can be optimized slightly by excluding the atom overlapping the node itself from the search). In order to obtain a k-balanced mesh [24], $0 < k < \mathcal{D}$, we can further remove atoms from the corner box to cover only the search directions we need.

Due to the ripple effect [22], balancing a particular block might require adjacent blocks to be balanced as well. Thus, we need to keep track of the blocks that have been balanced such that we can propagate the balancing to their neighbors (this is what we call rebalancing). To this end, we maintain two sets of nodes, a *balance list*, L, and a *rebalance set* of unique entries, B. Algorithm 8 operates in rounds. The balance list contains the nodes that are considered for balancing in the current round, and each node that is found to be too coarse is marked for refinement and inserted in the rebalance set. At the end of each balance round, we move the nodes in the rebalance set together with all nodes they overlap (i.e. the nodes that were refined in the present round) to the balance list and continue with the next round.

7 Results

7.1 Mesh Propagation

In order to study how the mesh adaptation performs, we construct examples of different dimensionalities, 2D and 4D, and study the mesh behaviour. In each experiment, a function is generated and propagated in time, and the mesh is evolved with it. Here, we do not actually solve a PDE; instead, we use an analytic function to which we know the values

Algorithm 8: `balance(kdtree` $\mathcal{T}$`)`

```
1:  L ← T.leaves                                  ▷ Initial list of nodes to balance is all leaf nodes
2:  R ← UniqueSet(∅)                              ▷ Rebalance set, initially empty
3:  while not L.isempty() do
4:      R ← ∅                                     ▷ Clear rebalance set
5:      for q ∈ L do
6:          for each corner atom c of q do
7:              v ← Neighbor of c across corner
8:              b ← Minimum bounding box of c and v (2^d atoms)
9:              for each atom u of corner box b do
10:                 o ← binarySearchOverlapping(T, u)        ▷ Find neighbor
11:                 t ← q.level − o.level                    ▷ Compare refinement of q and o
12:                 for {d ∈ D | t[d] > 1} do
13:                     Mark o for refinement in d
14:                     R.insert(o)                          ▷ Neighbor node must be rebalanced
15:                     R.insert(q)                          ▷ The node itself also needs rebalancing
16:                 end
17:             end
18:         end
19:     end
20:     refine(T)                                            ▷ Refine the balanced nodes
21:     L ← ∅                                                ▷ Clear list of nodes to balance
22:     for r ∈ R do                                         ▷ Copy nodes from rebalance set to balance list
23:         V ← findOverlappingNodes(T, r)                   ▷ Overlapping nodes include the newly refined ones
24:         L.insert(V)
25:     end
26: end
```

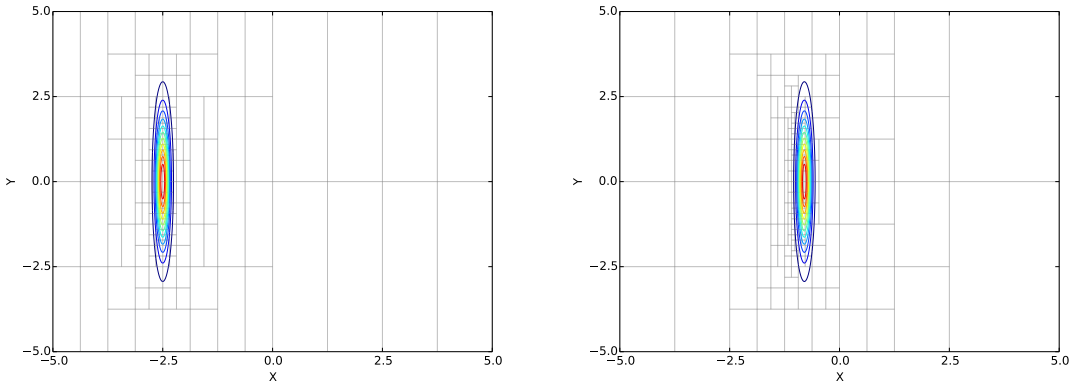
at time t and update the mesh points with the exact values in each time instance. This approach eliminates the impact of all exterior factors, such as interpolation and rounding errors, but still exercises all the important aspects of the mesh generation and adaptation. In each time step, the local truncation error is estimated in each block (using the normalized gradient [13]) and the mesh is updated from the existing mesh according to the estimated error. Note that the mesh is never rebuilt; once initialized we propagate it by adding and removing blocks dynamically.

As analytic functions, we use time-shifted gaussians

$$f(x_1, \ldots, x_d, t) = \prod_{i=1,\ldots,d} \exp\left(-\frac{(x_i - x_{i,0} - k_i t)^2}{2\sigma_i^2}\right), \tag{1}$$

since these are straightforward to extend to arbitrary dimensions.

Example: mesh propagation in 2D We first study a 2-dimensional example, given by the analytic function in (1) with parameters $\sigma_1 = 0.1$, $\sigma_2 = 1.2$, $k_1 = 5$, $k_2 = 0$, $x_{1,0} = 2.5$,



(a) $t = 0$, 228 mesh blocks.

(b) $t = 0.34$, 246 mesh blocks.

(c) $t = 0.68$, 229 mesh blocks.

(d) $t = 1.0$, 221 mesh blocks.

Fig. 6. Time-propagation of the 2-dimensional gaussian example.

Table 1. Execution times for the propagation experiments in Section 7.1. In each experiment, 100 time steps were taken. The propagation times are averaged over the time steps, and further broken down into the amounts of time spent in each subroutine; refine, coarsen and balance. Times are in seconds.

		2D	4D
Initialization		0.146	26.8
Propagation (avg. per time step)		0.015	1.89
Propagation (break down)	*refine*	$8.42 \cdot 10^{-4}$	$2.90 \cdot 10^{-2}$
	coarsen	$2.57 \cdot 10^{-3}$	$7.79 \cdot 10^{-2}$
	balance	$1.19 \cdot 10^{-2}$	3.97

$x_{2,0} = 0$. In this example, we use a blocksize of 16^2 mesh points. At time $t = 0$, this corresponds to the stretched gaussian function in Figure 6(a), shown together with the mesh that is adapted to it. Figures 6(b)-(d) show snapshots of the propagated solution and corresponding meshes at three following time instances up until $t = 1.0$.

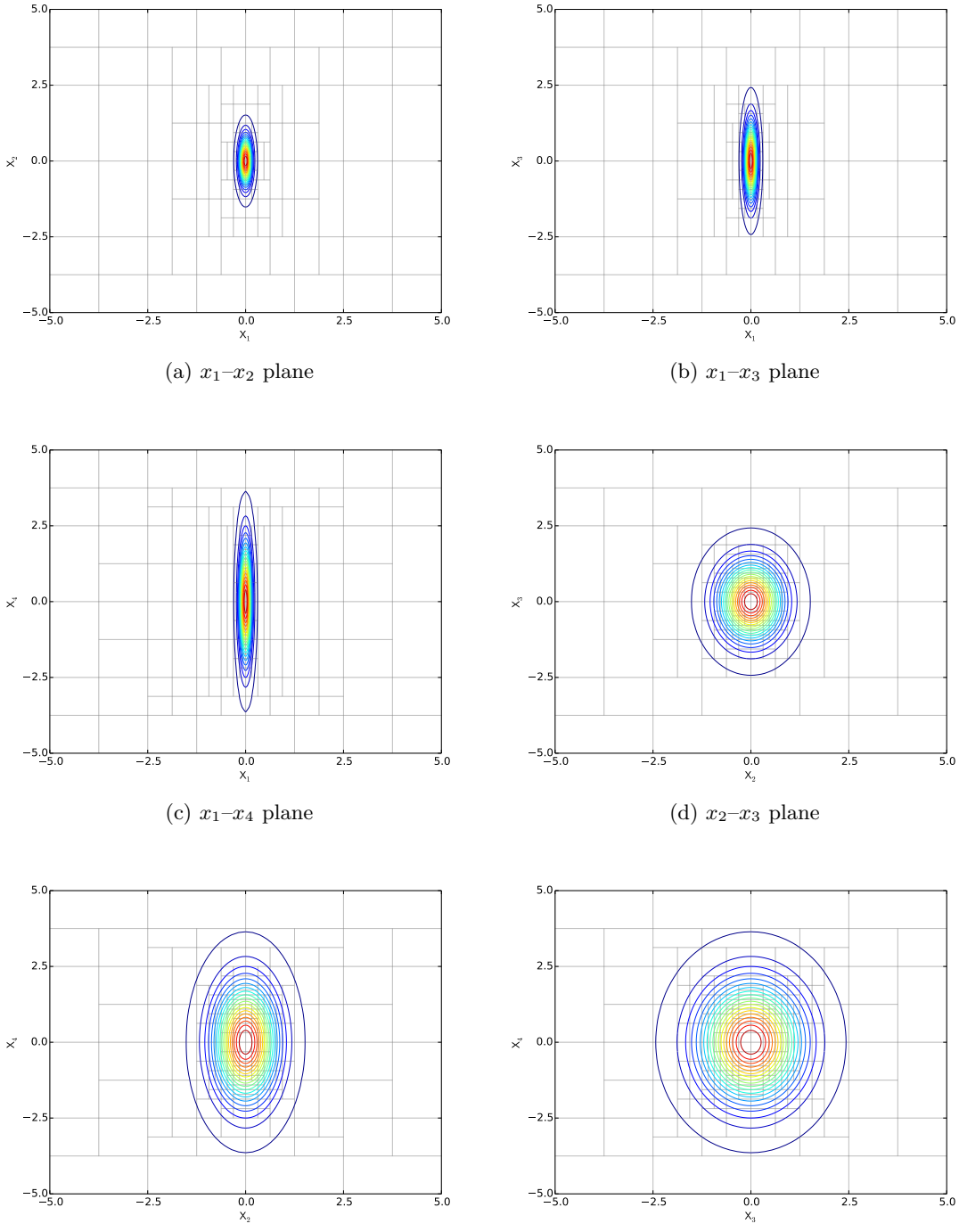
The mesh adapts well to the propagated solution, and we can clearly see the effects of anisotropic refinement strategy close to the function. These features are moved along with the solution, and we can also see that the mesh is properly coarsened as the solution moves out of regions that were previously refined. We also note that the number of mesh blocks during the simulation is close to constant throughout each time step.

Example: mesh propagation in 4D Next, we have a look at a 4-dimensional problem. We use a time-shifted gaussian (1) given by parameters $k_1 = 2.5$, $k_2 = 1.25$, $k_3 = k_4 = 0$, $\sigma_1 = 0.1$, $\sigma_2 = 0.5$, $\sigma_3 = 0.8$, $\sigma_4 = 1.2$, initially centered at the origin of the domain (thus $x_{1,0} = x_{2,0} = x_{3,0} = x_{4,0} = 0$). Each block stores 8^4 mesh points. The initial mesh is shown in Figure 7 as 2-dimensional cross-sections trough the origin (6 planes in total). Furthermore, in Figure 8, we plot the number of mesh nodes in each time step.

In this example too, we see that the initial mesh is well adapted to the function, with more anisotropic features in the dimensions where the function is elongated than in the dimensions where it is more uniform. From Figure 8 we draw the conclusion that the strategy to only merge nodes at the leaf level seems to lead to a somewhat suboptimal mesh contraction, with an initial increase in mesh nodes as the mesh moves, but after some time the total number of nodes stabilizes around an equilibrium.

7.2 Performance

We present two performance tests: a time breakdown of the mesh adaptation procedures (refine, coarsen and balance), based on the experiments in Section 7.1, and a more detailed performance and scalbility analysis of the balancing step which is the most time consuming part of the dynamic mesh adaptation.



(a) x_1–x_2 plane

(b) x_1–x_3 plane

(c) x_1–x_4 plane

(d) x_2–x_3 plane

(e) x_2–x_4 plane

(f) x_3–x_4 plane

Fig. 7. Initial function for the 4-dimensional mesh example, presented as separated cross-sections through the origin (a total of 8248 mesh blocks).

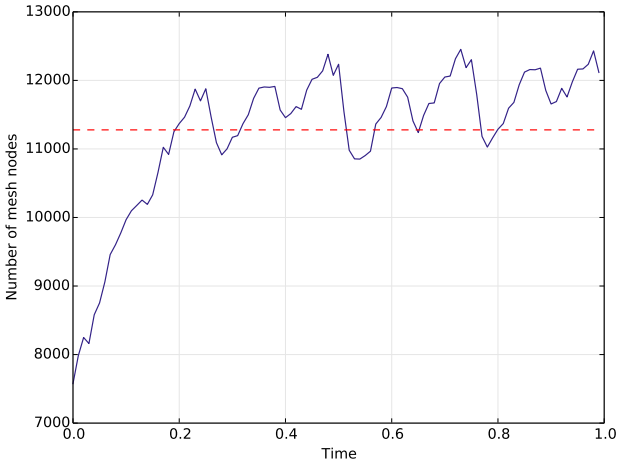


Fig. 8. Mesh node counts over the time steps for the 4D mesh propagation example. The dashed red line shows the average node count over all time steps.

All code is implemented in C++, and the experiments were performed on a workstation equipped with an Intel Core i5-2500 processor; 4 cores running at 3.3 GHz, 6 MB L3 cache, 4×256 kB L2 cache, 4×32 kB L1 instruction/data caches, and 16 GB of RAM. We used the GNU C++ compiler, version 4.6.1, with flags `-O3 -funroll-loops`. The code makes use of a template constant `<DIMENSIONS>` in order to statically unroll loops over dimensions.

Mesh propagation performance analysis We study performance data from the mesh propagation experiments in Section 7.1, separated into respective timings for each adaptation step; initialization, refinement, coarsening and balancing. The results are found in Table 1. We present the total time for initializing the mesh and the average propagation time per time step. Propagation includes refinement, coarsening and balancing; the average propagation time per time step is further broken down into the time spent in each of these procedures. The time taken to initialize a mesh is given separately since it is a one-time cost, amortized over a large number of time steps.

Performance of 2:1 balancing From the timings in Table 1 we observe that the balancing step is the most time consuming part of the dynamic mesh propagation. Each node in the mesh has a large number of dependencies to neighboring nodes that must be searched for and examined with respect to refinement ratios. Furthermore, due to the ripple propagation described in 6.3, nodes that have been balanced need to be rebalanced iteratively. In this section we study the performance of the balance step in more detail. We are particularly interested in how it scales with dimensionality, since the complexity of the balance

algorithm is directly dependent on the number of potential neighbors of each block to search for.

We study two types of synthetic benchmark meshes:

- Randomly refined meshes of different node counts and dimensionalities. We repeatedly iterate over the nodes and (with a probability of $p_r = 0.5$) refine each node in a random dimension. This creates meshes that are very badly structured and difficult to balance, but serves as a good worst-case scenario for analysis. In this type of meshes, we expect to see a large portion of anisotropic mesh elements.
- Meshes where one of the innermost corners is repeatedly refined isotropically, which creates a subregion with 2^d nodes refined to a certain refinement level (down to maximum refinement level $\mathcal{B}$), cornered by nodes of refinement level 1. We will not encounter any anisotropic mesh elements in this case, but we will see deep chains of rippling refinements due to the imbalance in the initial mesh.

The performance results are presented in Figure 9. We see an exponential increase in runtime as the dimensionality is increased, but the performance penalty for each dimensionality is constant. Within each given dimensionality, the scaling with the number of nodes is better than $n \log n$. The staggered plot in Figure 9 is due to the vast difference in the numbers of nodes generated for each dimensionality in this example. Although there are jumps between the lines in the plot, the trend is clear and each line follows a leaner slope.

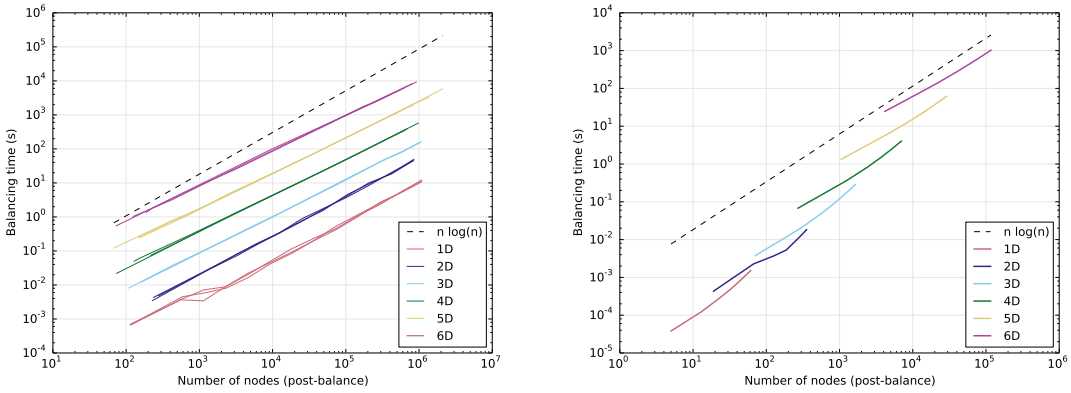


Fig. 9. Execution time for balancing of two types of meshes. (a) Randomly generated meshes; individual timings on 3 different randomizations. (b) Mesh with innermost corner refined down to maximum level. The number of nodes on the x-axis refers to the number of nodes in the mesh after balancing. The reference line shows $n \log n$ scaling.

8 Conclusion

In this paper, we have presented an efficient scheme for managing adaptively refined meshes with anisotropic features. Our framework is built around a linear representation of kdtrees, with algorithms for refinement, coarsening and 2:1 balancing. Our linear kdtree data structure can be seen as a generalization of the quadtree/octree data structures for higher dimensionalities, with added flexibility in terms of layout and shape of grid elements. Compared to a binary tree representation of a kdtree, the linear tree encoding is more efficient in terms of storage and searching in unbalanced trees. In order to obtain a more robust ordering of the nodes, we store a lightweight representation of the internal node structure; a node index map. With the information available in this structure, we obtain a hierarchical Morton index, by which we linearize the nodes. Our performance results show scaling trends that are close to or better than $\mathcal{O}(n \log n)$ in practical examples, although parts of the algorithms scale as $\mathcal{O}(n^2)$ in the worst case. This paper has focused on the algorithmic foundation and properties of the adaptive grids. In a follow-up paper we will extend the scheme with distributed parallel functionality in order to scale the code to massively parallel computers.

References

1. M. Bader. *Space-filling curves: An introduction with applications in scientific computing.* Springer Berlin Heidelberg, 2013.
2. M. Bader, S. Schraufstetter, C. A. Vigh, and J. Behrens. Memory efficient adaptive mesh generation and implementation of multigrid algorithms using sierpinski curves. *International Journal of Computational Science and Engineering*, 4(1):12–21, 2008.
3. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
4. C. Burstedde, L. C. Wilcox, and O. Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.
5. C. Chuanbo and Z. Haiming. Linear binary tree. In *[1988 Proceedings] 9th International Conference on Pattern Recognition*, pages 576–578. IEEE Computuer Society Press, 1988.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT Press, Cambridge, 2nd edition, 2001.
7. A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive discretization schemes: Abstraction principles and the DUNE-FEM module. *Computing*, 90(3-4):165–196, 2010.
8. R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.
9. J. H. Freidman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.
10. S. F. Frisken and R. N. Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7(3):1–11, 2002.
11. G. Gan, C. Ma, and J. Wu. *Data clustering: Theory, algorithms and applications.* Society for Industrial and Applied Mathematics, 2007.
12. I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.

13. J. Jessee, W. A. Fiveland, L. H. Howell, P. Colella, and R. B. Pember. An adaptive mesh refinement algorithm for the radiative transport equation. *Journal of Computational Physics*, 139(2):380–398, 1998.
14. R. Klöfkorn and M. Nolte. Performance pitfalls in the DUNE grid interface. In A. Dedner, B. Flemisch, and R. Klöfkorn, editors, *Advances in DUNE*, pages 45–58. Springer Berlin Heidelberg, 2012.
15. G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.
16. A. Nissen, K. Kormann, M. Grandin, and K. Virta. Stable difference methods for block-oriented adaptive grids. *arXiv:1405.0735 [math.NA]*, May 2014.
17. M. Nolte. *Efficient numerical approximation of the effective hamiltonian*. PhD thesis, Institute of mathematics, University of Freiburg, Dec. 2011.
18. J. Rantakokko and M. Thuné. Parallel structured adaptive mesh refinement. In R. Trobec, M. Vajteršic, and P. Zinterhof, editors, *Parallel Computing*, pages 147–173. Springer London, 2009.
19. M.-C. Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM Journal on Numerical Analysis*, 21(3):604–613, 1984.
20. H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.
21. H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):579–586, 1988.
22. H. Sundar, R. S. Sampath, and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.
23. C. T. Traxler. An algorithm for adaptive mesh refinement in n dimensions. *Computing*, 59(2):115–137, 1997.
24. K. Weiss and L. De Floriani. Bisection-based triangulations of nested hypercubic meshes. In S. Shontz, editor, *Proceedings of the 19th International Meshing Roundtable*, pages 315–333. Springer Berlin Heidelberg, 2010.