Scalable algorithms for adaptive mesh refinement with arbitrary element types

Johannes Holke (German Aerospace Center DLR, Cologne)
Carsten Burstedde (University of Bonn)
Who am i and why am i here?

- 2014 Master of Science, Mathematics, University of Bonn
- 2014-2018 PhD under Carsten Burstedde, Institute for Numerical Simulation, Bonn
- Since 2018 at DLR, Simulation and Software Technology | High-performance Computing

I like: Adaptive meshes, tetrahedra, large scale computing
Who am i and why am i here?
Who am i and why am i here?

We are getting scaling problems because of mesh partitioning (parMETIS) - At CoSaS 2018
Outline

Adaptive Meshes

Trees and space-filling curves

The TM-curve

Arbitrary element types

Recursive search

Numerical results
Adaptive Meshes

Uniform

Adaptive
Adaptive Meshes
Adaptive Meshes

Adaptive Refinement

Only refine the mesh where needed.

- The same computational error with less elements
- Mesh management becomes more complicated
Dynamical AMR

The mesh changes (frequently) during the simulation (i.e. every $n$ time steps).
AMR Algorithms

- Input: Coarse mesh
- New
- Adapt
- Balance
- Partition
- Ghost
AMR Algorithms

- Input: Coarse mesh
- New
- Adapt
- Balance
- Partition
- Ghost
AMR Algorithms

- Input: Coarse mesh
- New
- Adapt
- Balance
- Partition
- Ghost

Adapt + Balance
AMR Algorithms

- Input: Coarse mesh
- New
- Adapt
- Balance
- Partition
- Ghost
AMR Algorithms

- Input: Coarse mesh
- New
- Adapt
- Balance
- Partition
- Ghost

Repeat
Trees and space-filling curves

Unstructured

Structured
Trees and space-filling curves
Unstructured meshes

Arbitrary connectivity. All element neighbors and grid coordinates need to be stored explicitly.
Unstructured meshes

Arbitrary connectivity. All element neighbors and grid coordinates need to be stored explicitly.

- Full geometric flexibility
- Works with all element shapes → hybrid meshes
Unstructured meshes

Arbitrary connectivity. All element neighbors and grid coordinates need to be stored explicitly.

- Full geometric flexibility
- Works with all element shapes → hybrid meshes
- Large memory footprint
- Long runtimes of AMR algorithms
Unstructured meshes

Arbitrary connectivity. All element neighbors and grid coordinates need to be stored explicitly.

- Full geometric flexibility
- Works with all element shapes $\rightarrow$ hybrid meshes
- Large memory footprint
- Long runtimes of AMR algorithms

(Re-)Partition

Often with graph based methods (parMETIS/SCOTCH).
1.000 - 10.000 elements/s per process\(^a\)

\(^a\)Smith, Rasquin, Shephard et. Al. Application specific mesh partition improvement. 2015
Tree based AMR

Idea: The geometrical resolution needed is often much coarser than the numerical resolution.
Tree based AMR

Idea: The geometrical resolution needed is often much coarser than the numerical resolution.
Tree based AMR

Idea: The geometrical resolution needed is often much coarser than the numerical resolution.

Left geometry: 11k spheres, 383 million tetrahedra
Right adaptive refinement: 167 billion tetrahedra
Tree based AMR

Idea: The geometrical resolution needed is often much coarser than the numerical resolution.
Tree based AMR
Start with (Unstructured) input mesh modelling the geometry
**Tree based AMR**

Start with (Unstructured) input mesh modelling the geometry

Use (structured) refinement rule within every coarse mesh cell.
Tree based AMR
Start with (Unstructured) input mesh modelling the geometry

Use (structured) refinement rule within every coarse mesh cell.

Applying recursively leads to refinement tree.
Space-filling curves (SFC)
Linear order of the leaves of each tree.
Space-filling curves (SFC)

Linear order of the leaves of each tree.

Efficient partitioning (weights are possible as well)
Two meshes: Coarse mesh that describes the geometry.
Fine mesh for the computation.
Two meshes: Coarse mesh that describes the geometry.
  Fine mesh for the computation.

Each coarse mesh cell is a refinement tree.
Initial partition of coarse mesh (i.e. METIS) part of preprocessing.
Trees and space-filling curves

Properties

- Geometric flexibility
- Works with all shapes that have an SFC $\rightarrow$ Hybrid meshes are possible
- Low memory footprint (Coordinates and neighbors per tree)
- Shorter runtimes and better scalability
Trees and space-filling curves

Properties

- Geometric flexibility
- Works with all shapes that have an SFC \(\rightarrow\) Hybrid meshes are possible
- Low memory footprint (Coordinates and neighbors per tree)
- Shorter runtimes and better scalability

(Re-)Partition

Unstructured\(^a\)  1.000 - 10.000  Elements/s per prozess
Trees and space-filling curves

Properties

- Geometric flexibility
- Works with all shapes that have an SFC → Hybrid meshes are possible
- Low memory footprint (Coordinates and neighbors per tree)
- Shorter runtimes and better scalability

(Re-)Partition

<table>
<thead>
<tr>
<th>Type</th>
<th>Unstructured</th>
<th>Tree based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements/s per process</td>
<td>1.000 - 10.000</td>
<td>700.000 - 5,000.000</td>
</tr>
</tbody>
</table>

\[a\] Smith, Rasquin, Shephard et al. Application specific mesh partition improvement. 2015

\[b\] Burstedde, Holke, Coarse mesh partitioning for tree-based AMR, 2017
Trees and space-filling curves

Properties

- Geometric flexibility
- Works with all shapes that have an SFC $\rightarrow$ Hybrid meshes are possible
- Low memory footprint (Coordinates and neighbors per tree)
- Shorter runtimes and better scalability

(Re-)Partition

Unstructured$^a$ 1,000 - 10,000 Elements/s per prozess
Tree based$^b$ 700,000 - 5,000,000

- Cache efficient
- Recursive search
- Possibly unconnected partitions

---

$^a$ Smith, Rasquin, Shephard et. Al. *Application specific mesh partition improvement*. 2015

$^b$ Burstedde, Holke, *Coarse mesh partitioning for tree-based AMR*, 2017
Experience: Tree based AMR does pay off especially when the mesh changes frequently.
Interlude

Hanging nodes?

To resolve hanging nodes one could either

1. Interpolate in the solver routines.
2. Resolve them with one green refinement step after Adapt and Balance.
**Interlude**

**Hanging nodes?**

To resolve hanging nodes one could either

1. Interpolate in the solver routines.
2. Resolve them with one green refinement step after Adapt and Balance.
Interlude

Hanging nodes?

To resolve hanging nodes one could either

1. Interpolate in the solver routines.
2. Resolve them with one green refinement step after Adapt and Balance.
Existing frameworks

Tree based AMR libraries

- Quad/Hex, Peano curve - Peano
- Quad/Hex, Morton curve - p4est (Burstedde et al)
- Triangles, Sierpinski curve - sam(oa)²
Existing frameworks

Tree based AMR libraries

- Quad/Hex, Peano curve - Peano
- Quad/Hex, Morton curve - p4est (Burstedde et. al)
- Triangles, Sierpinski curve - sam(oa)$^2$
- Tetrahedra - ?
Existing frameworks

Tree based AMR libraries

- Quad/Hex, Peano curve - Peano
- Quad/Hex, Morton curve - p4est (Burstedde et al.)
- Triangles, Sierpinski curve - sam(oa)^2
- Tetrahedra/triangles - t8code
Existing frameworks

Tree based AMR libraries

- Quad/Hex, Peano curve - Peano
- Quad/Hex, Morton curve - p4est (Burstedde et. al)
- Triangles, Sierpinski curve - sam\(^2\) (oa)
- Tetrahedra/triangles - t8code
- Hybrid - ?
  - Prisms - ?
  - Pyramids - ?
Existing frameworks

Tree based AMR libraries

- Quad/Hex, Peano curve - Peano
- Quad/Hex, Morton curve - p4est (Burstedde et. al)
- Triangles, Sierpinski curve - sam\(^2\) (oa)
- Tetrahedra/triangles - t8code
- Hybrid - ?
  - Prisms - ?
  - Pyramids - ?
The Morton curve

Morton curve

The Z-curve for quadrilateral/hexahedral meshes is constructed via the Morton code (Morton 1966). It is the bitwise interleaving of an element's anchor coordinates.
The Morton curve

Morton curve

The Z-curve for quadrilateral/hexahedral meshes is constructed via the Morton code (Morton 1966). It is the bitwise interleaving of an elements anchor coordinates.

\[ x(Q) = 6 = (0110)_2 \]
\[ y(Q) = 8 = (1000)_2 \]
\[ \ell(Q) = 3 \Rightarrow m(Q) = (10010100)_2 = (2 \times 1 \times 1 \times 0)_{10} = 148 \]

Fix a maximum refinement level \( \mathcal{L} \), i.e. \( \mathcal{L} = 4 \).
The Morton curve

Morton curve

The Z-curve for quadrilateral/hexahedral meshes is constructed via the Morton code (Morton 1966). It is the bitwise interleaving of an element’s anchor coordinates.

Fix a maximum refinement level \( L \), i.e. \( L = 4 \).
The Morton curve

Using the Morton index, many low-level algorithms can be computed efficiently. Example:
The Morton curve

Using the Morton index, many low-level algorithms can be computed efficiently. Example:

**Computing face-neighbors (within tree)**

Add or subtract $h = 2^{\ell - L}$ from the appropriate coordinate.
The Morton curve

Advantageous features of the Morton code

- Computable in a fast manner via bitwise interleaving.
- Easy to implement.
- Memory efficient
  - storage per element: coordinates of one node, level \( (4d + 1 \text{ Bytes}) \).
- Children, parent, face-neighbor, etc. computed in constant time.
- 2D and 3D follow the same logic.
We embed this triangle in the square $[0, 2^L]^2$, which is divided along its diagonal; obtaining a second triangle $S_1$. 
Bey’s observation:
When refining $S_0$, each occurring triangle $T$ is equivalent to $S_0$ or $S_1$. 

The TM-curve
The TM-curve

Bey’s observation:
When refining $S_0$, each occurring triangle $T$ is equivalent to $S_0$ or $S_1$. 
Bey’s observation:
When refining $S_0$, each occurring triangle $T$ is equivalent to $S_0$ or $S_1$.

Definition
$\text{Type}(T) = i \iff T \simeq S_i$. 

The TM-curve
The TM-curve

Furthermore, there is also an underlying quadrilateral mesh.
A triangle $T$ in a refinement is uniquely identified by the coordinates of one node plus its level plus its type.
The TM-curve

The same constructions applies in 3d. Here we have six different types.

Each tetrahedron $T$ in a refinement of $S_0$ is equivalent to one of $S_0, \ldots, S_5$. 
The TM-curve

The same constructions applies in 3d. Here we have six different types.

Each tetrahedron $T$ in a refinement of $S_0$ is equivalent to one of $S_0, \ldots, S_5$. 
The TM-curve

The same constructions applies in 3d. Here we have six different types.

Each tetrahedron $T$ in a refinement of $S_0$ is equivalent to one of $S_0, \ldots, S_5$. It is uniquely identified by the coordinates of one node plus its level plus its type.
The TM-curve

Definition

The tetrahedral Morton index of a simplex $T$ is constructed by interleaving $(z, y, x)$ coordinates of the anchor node of $T$ with $B$.

$$(y_{L-1}, \ldots, y_0)_2 \quad (x_{L-1}, \ldots, x_0)_2 \quad (b_{L-1}, \ldots, b_0)_4$$

$$m(T) = (y_{L-1}x_{L-1}, b_{L-1}, \ldots, y_0x_0, b_0)_4$$
The TM-curve
Computing face-neighbors, an example
Computing face-neighbors, an example

Computation of N0
Add $h$ to z-coordinate, change type to 1.
Computing face-neighbors, an example

Computation of N1
Change type to 0.
Computing face-neighbors, an example

Computation of N2
Change type to 4.
Computing face-neighbors, an example

Computation of N3

Subtract $h$ from $y$-coordinate, change type to 3.
Computing face-neighbors

We do this once for each type and obtain a look-up table. Computing face-neighbors is as easy as looking up the values in the table.
Computing face-neighbors

We do this once for each type and obtain a look-up table. Computing face-neighbors is as easy as looking up the values in the table.

Parent, children and node coordinates can be computed similarly.
The TM-curve

Advantageous features of the tetrahedral Morton code

- Computable in a fast manner via bitwise interleaving.
- Easy to implement.
- Memory efficient
  - storage per element: coordinates of one node, level \((4d + 1)\) Bytes.
- Children, parent, face-neighbor, etc.
  - computed in constant time.
- 2D and 3D follow the same logic.
Tree based AMR libraries

- Quad/Hex, Peano curve - *Peano*
- Quad/Hex, Morton curve - p4est, *t8code*
- Triangles, Sierpinski curve - *sam*(oa)$^2$
- Tetrahedra/triangles - *t8code*
- Hybrid - ?
  - Prisen - ?
  - Pyramiden - ?
Arbitrary element types

Core algorithms (high-level)

- New
- Adapt
- Partition
- Ghost
- Balance
Arbitrary element types

Core algorithms (high-level)
- New
- Adapt
- Partition
- Ghost
- Balance

low-level
- element_refine
- element_parent
- element_id
- element_face_neighbor
  .
  .
  .
Arbitrary element types

Core algorithms (high-level)

- New
- Adapt
- Partition
- Ghost
- Balance

low-level

- element_refine
- element_parent
- element_id
- element_face_neighbor

- Decouple high-level and low-level algorithms
- Define API for element-local (low-level) functions
- Low-level functions can be exchanged arbitrarily without affecting high-level logic
Example: Face-neighbors across tree boundaries

Avoid couplings

Hex ↔ Prism
Prism ↔ Tet
Quad ↔ Tri
...
Example: Face-neighbors across tree boundaries

Avoid couplings

- Hex ↔ Prism
- Prism ↔ Tet
- Quad ↔ Tri
- ...
Example: Face-neighbors across tree boundaries
Example: Face-neighbors across tree boundaries

1: Build 2D boundary element
Example: Face-neighbors across tree boundaries

1: Build 2D boundary element

2: Transform coordinates
Example: Face-neighbors across tree boundaries

1: Build 2D boundary element

2: Transform coordinates

3: Extrude boundary element
Arbitrary element types

Do whatever you want

The decoupling of high- and low-level functions allows the user to implement its own refinement pattern and SFC.

search

Example: $N$ particles in the domain. For each, find the containing element and execute a callback function.
search

Example: $N$ particles in the domain. For each, find the containing element and execute a callback function.

More general: Find elements that match a certain condition, and execute a callback.
search

Example: $N$ particles in the domain. For each, find the containing element and execute a callback function.

More general: Find elements that match a certain condition, and execute a callback.

With recursive top-down search in a tree, we can handle all conditions (i.e. points) and elements in one run. And, we can exclude complete subtrees from the search.
search
search
search
search
search
search
search
search
search
search
search
search
search
search

Example: Ghost

Find all elements at our domain boundary. Communicate them with the neighboring process.

<table>
<thead>
<tr>
<th></th>
<th>tetrahedra</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>uniform</td>
<td>adaptive</td>
<td>uniform</td>
<td>adaptive</td>
<td>uniform</td>
<td>adaptive</td>
<td></td>
</tr>
<tr>
<td>Level</td>
<td>uniform</td>
<td>adaptive</td>
<td>uniform</td>
<td>adaptive</td>
<td>uniform</td>
<td>adaptive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>8–10</td>
<td>7–9</td>
<td>3–5</td>
<td></td>
</tr>
<tr>
<td>elements/proc</td>
<td>786,432</td>
<td>98,304</td>
<td>24</td>
<td>1,015,808</td>
<td>126,976</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>ghosts/proc</td>
<td>32,704</td>
<td>8,160</td>
<td>30</td>
<td>31,604</td>
<td>8,137</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>Ghost [s]</td>
<td>129.6</td>
<td>16.19</td>
<td>5.93e-3</td>
<td>167.94</td>
<td>20.88</td>
<td>8.10e-3</td>
<td></td>
</tr>
<tr>
<td>Ghost with search [s]</td>
<td>7.41</td>
<td>1.75</td>
<td>5.01e-3</td>
<td>7.08</td>
<td>1.69</td>
<td>8.12e-3</td>
<td></td>
</tr>
</tbody>
</table>

Up to 24 times speed-up
Numerical results
Strong scaling, JUQUEEN

2 billion tetrahedra
Parallel efficiency of Ghost $\geq 96.6\%$
Weak scaling, JUQUEEN

Ghost with $\sim 233k$ Elements/process (Tet), $\sim 310k$ elements/process (Hex).
Largest mesh: 162 billion Hexaedra, 108 billion tetrahedra
96.8% efficiency on 458,752 processes
Large scale mesh partition

Coarse mesh: 325 million Tets
Fine mesh: 167 billion Tets
Configuration: Juqueen, 458,752 MPI ranks
Large scale mesh partition

Coarse mesh partition on 458,752 MPI ranks

<table>
<thead>
<tr>
<th>t</th>
<th>mesh size</th>
<th>trees (ghosts) sent</th>
<th>shared trees</th>
<th>run time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>324,766,336</td>
<td>704 (2444)</td>
<td>280,339</td>
<td>0.207</td>
</tr>
<tr>
<td>2</td>
<td>324,766,336</td>
<td>708 (2456)</td>
<td>281,694</td>
<td>0.204</td>
</tr>
<tr>
<td>3</td>
<td>324,766,336</td>
<td>707 (2458)</td>
<td>281,900</td>
<td>0.204</td>
</tr>
</tbody>
</table>

Fine mesh partition on 458,752 MPI ranks

<table>
<thead>
<tr>
<th>t</th>
<th>mesh size</th>
<th>elements sent</th>
<th>run time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>167,625,595,829</td>
<td>362,863</td>
<td>0.522</td>
</tr>
<tr>
<td>2</td>
<td>167,709,936,554</td>
<td>364,778</td>
<td>0.578</td>
</tr>
<tr>
<td>3</td>
<td>167,841,392,949</td>
<td>365,322</td>
<td>0.567</td>
</tr>
</tbody>
</table>

Tabelle: Run times for coarse mesh and fine mesh partition for the brick with holes on 458,752 MPI ranks. Forest level 3 to 4 (365k elements/proc).
Advection solver
Advection solver
Advection solver
Strong scaling, JUQUEEN

2,9 billion tetrahedra, $\ell = 5, r = 6$. 
Conclusion

- Constructed and investigated new SFC for tets and triangles
- Develop and evaluate new massively scaling algorithms (t8code)
- New method of coarse mesh partitioning
- Easily extendable with new element types/SFCs (prisms in a Bachelor’s thesis)
- First application of tree based AMR with tetrahedra and hybrid meshes.
- See github.com/holke/t8code, GPL v2

Wish list

- Currently limited to face-connectivity
- Balance is a bottleneck
- Pyramids
- ...
Conclusion

- Constructed and investigated new SFC for tets and triangles
- Develop and evaluate new massively scaling algorithms (t8code)
- New method of coarse mesh partitioning
- Easily extentable with new element types/SFCs (prisms in a Bachelor’s thesis)
- First application of tree based AMR with tetrahedra and hybrid meshes.
- See github.com/holke/t8code, GPL v2

Wish list

- Currently limited to face-connectivity
- Balance is a bottleneck
- Pyramids
- ...

Thank you for your attention.