Modularity of Lowlevel Forest-of-octree Libraries

Johannes Holke (German Aerospace Center DLR, Cologne)
Carsten Burstedde (University of Bonn)

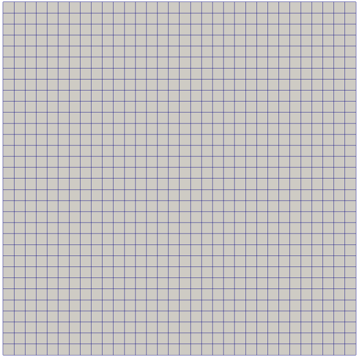
Outline

Adaptive Mesh Refinement

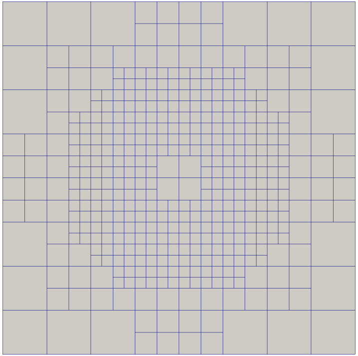
`p4est` and `t8code`

Arbitrary element types

Adaptive Meshes

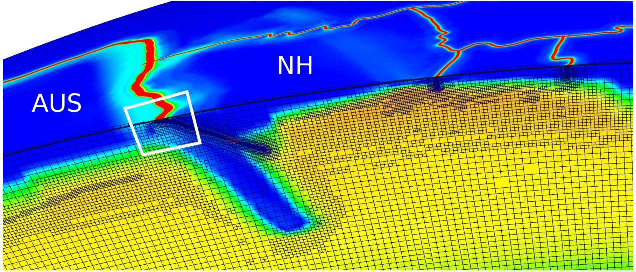


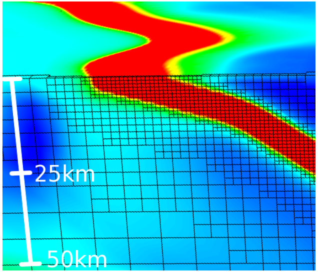
Uniform



Adaptive

Adaptive Meshes





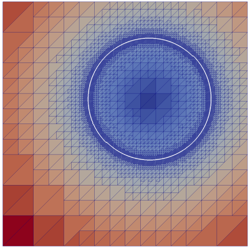
Source: *The dynamics of plate tectonics and mantle flow: From local to global scales.*, G. Stadler, M. Gurnis, C. Burstedde, L. C. Wilcox, L. Alisic, and O. Ghattas. Science, 329(5995):1033–1038, 2010

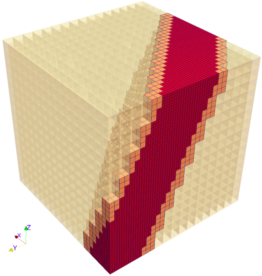
Adaptive Meshes

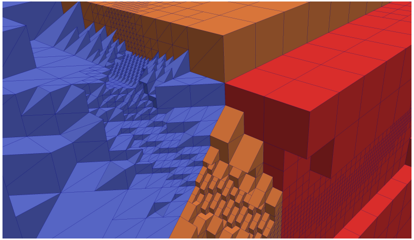
Adaptive Refinement

Only refine the mesh where needed.

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

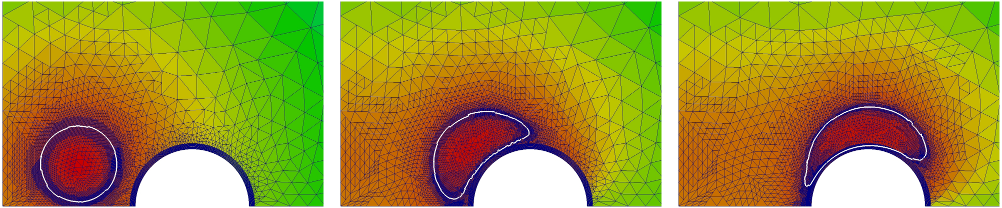






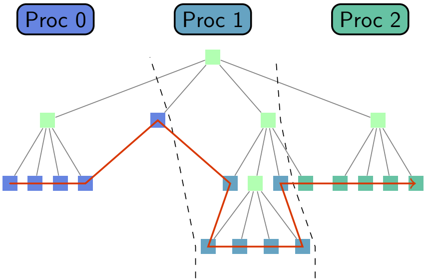
Dynamical AMR

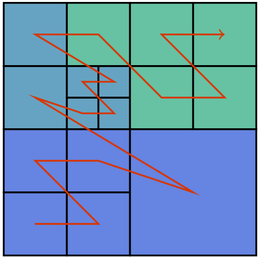
The mesh changes (frequently) during the simulation (i.e. every n time steps).



AMR needs to be fast and scalable!

Refinement tree and SFC

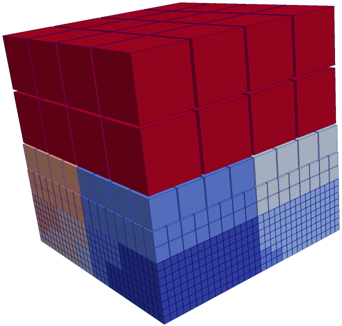




Forest of trees

From tree...

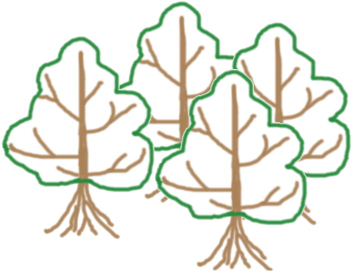
=



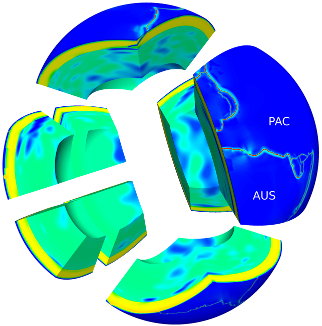
- Limitation: Cube-like geometric shapes

Forest of trees

…to forest

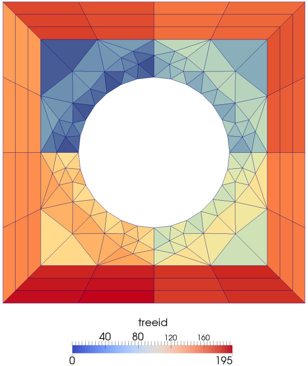


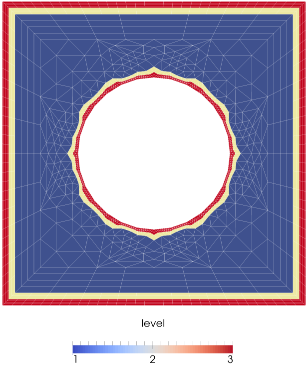
=



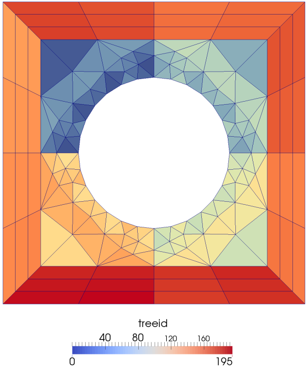
Science 329 (5995), p. 1033-1038

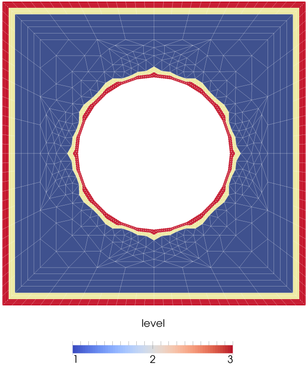
- Advantage: Geometric flexibility
- Challenge: Non-matching coordinate systems between octrees





Two meshes: Unstructured **coarse mesh** that describes the geometry.
Fine mesh for the computation.





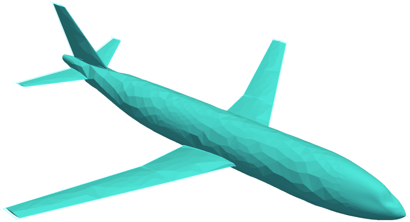
Two meshes: Unstructured **coarse mesh** that describes the geometry.
Fine mesh for the computation.

Each coarse mesh cell is a refinement tree.
Initial partition of coarse mesh (i.e. METIS) part of preprocessing.

AMR Algorithms

- `Input: Coarse mesh`
- `New`
- `Adapt`
- `(Balance)`
- `Partition`
- `Ghost`
- `(Iterate)`
- `(Search)`

Repeat

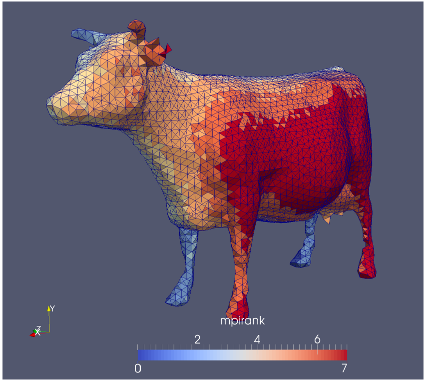


Coarse Mesh

AMR Algorithms

- `Input: Coarse mesh`
- `New`
- `Adapt`
- `(Balance)`
- `Partition`
- `Ghost`
- `(Iterate)`
- `(Search)`

Repeat

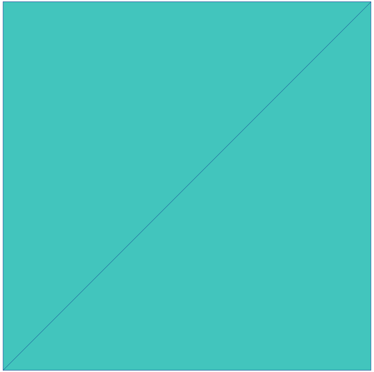


Coarse Mesh

AMR Algorithms

- `Input: Coarse mesh`
- `New`
- `Adapt`
- `(Balance)`
- `Partition`
- `Ghost`
- `(Iterate)`
- `(Search)`

Repeat

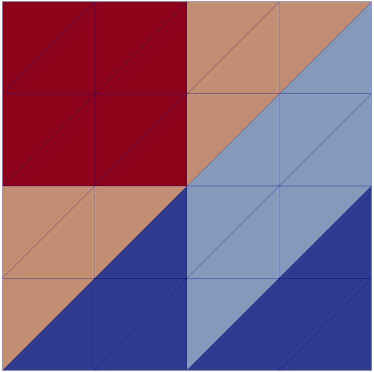


Coarse Mesh

AMR Algorithms

- `Input: Coarse mesh`
- `New`
- `Adapt`
- `(Balance)`
- `Partition`
- `Ghost`
- `(Iterate)`
- `(Search)`

Repeat

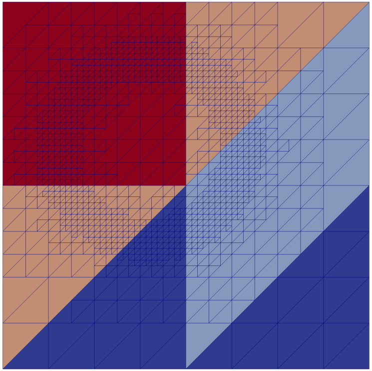


`New`

AMR Algorithms

- Input: Coarse mesh
- New
- Adapt
- (Balance)
- Partition
- Ghost
- (Iterate)
- (Search)

Repeat

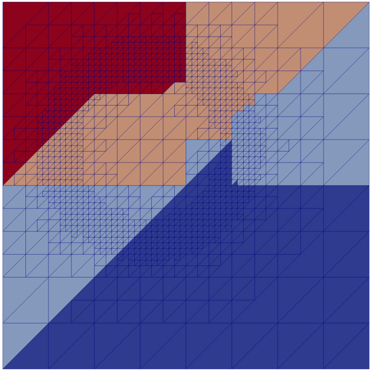


Adapt + Balance

AMR Algorithms

- `Input: Coarse mesh`
- `New`
- `Adapt`
- `(Balance)`
- `Partition`
- `Ghost`
- `(Iterate)`
- `(Search)`

Repeat

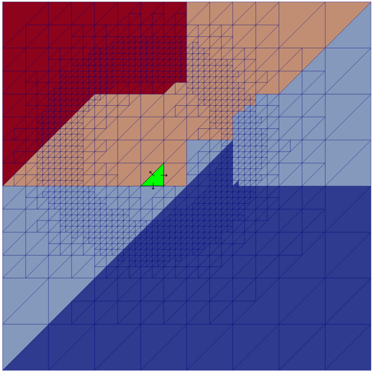


`Partition`

AMR Algorithms

- `Input: Coarse mesh`
- `New`
- `Adapt`
- `(Balance)`
- `Partition`
- `Ghost`
- `(Iterate)`
- `(Search)`

Repeat

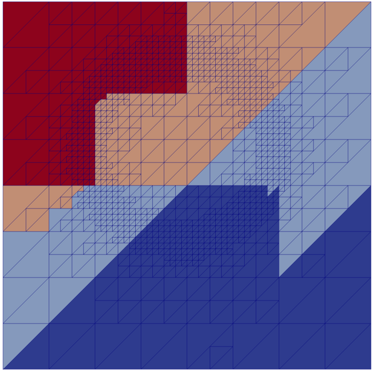


`Iterate`

AMR Algorithms

- Input: Coarse mesh
- New
- Adapt
- (Balance)
- Partition
- Ghost
- (Iterate)
- (Search)

Repeat



Repeat

`p4est` **and** `t8code`

`p4est`

- Developed by C. Burstedde[a], T. Isaac[b], L. C. Wilcox[c] and others since $\sim$ 2007
- Quads (2D) and Hexes (3D), Z-Curve

- Face/Edge/Corner connections
- Coarse mesh replicated on each rank
- Scales to $> 10^6$ MPI ranks, 10^{12} elements
- Used in `deal.ii`, `petsc`, `ForestClaw`, `Rhea`, `ParFlow`, ...

---

[a] University of Bonn, [b] Georgia Tech, [c] Naval Postgraduate School

`p4est` and `t8code`

`p4est`

- Developed by C. Burstedde[a], T. Isaac[b], L. C. Wilcox[c] and others since $\sim$ 2007
- Quads (2D) and Hexes (3D), Z-Curve
- Face/Edge/Corner connections
- Coarse mesh replicated on each rank
- Scales to $> 10^6$ MPI ranks, 10^{12} elements
- Used in `deal.ii`, `petsc`, `ForestClaw`, `Rhea`, `ParFlow`, ...

`t8code`

- Developed by C. Burstedde and J. Holke since 2014
- Points/Lines/Quads/Tris/Hexes/Tets/Prisms hybrid, extendable in a modular fashion
- Currently Face connections only
- Coarse mesh partitioned
- Scales to $> 10^6$ MPI ranks, 10^{12} elements
- Performance comparable to `p4est`

[a]University of Bonn, [b] Georgia Tech, [c] Naval Postgraduate School

Our philosophy

"Do one thing and do it well."

We do

- Fast and scalable AMR algorithms
- Exchange of communication buffers (packed by application)
- Mesh elements as black box for anything that the application wants
- Topology information
- Iterators over mesh elements and element-to-element interfaces
- Fast local and global search algorithms
- Full flexibility regarding refinement criteria

We do not

- PDE solvers
- Tell you what data to put on mesh elements
- packing/unpacking of data
- Geometry information
- Interpolating data between meshes (but we give you the meshes and corresponding elements)
- Refinement criteria
- Time stepping

Arbitrary element types

Core algorithms (high-level)

- `New`
- `Adapt`
- `Partition`
- `Ghost`
- `Balance`
- `Iterate`

Arbitrary element types

Core algorithms (high-level)

- `New`
- `Adapt`
- `Partition`
- `Ghost`
- `Balance`
- `Iterate`

low-level

- `element_refine`
- `element_parent`
- `element_id`
- `element_face_neighbor`

$\vdots$

Arbitrary element types

Core algorithms (high-level)

- `New`
- `Adapt`
- `Partition`
- `Ghost`
- `Balance`
- `Iterate`

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
- Realized via `C++` pure virtual base class

Low-level API

Virtual base class

```
/** This struct holds virtual functions for a particular element class. */
struct t8_eclass_scheme
{
  /** This scheme defines the operations for a particular element class. */
protected:
public:
   t8_eclass_t eclass;      /**< The element class */

  /** Return the maximum allowed level for any element of a given class.
   * \return                      The maximum allowed level for elements of class \b ts.
   */
  virtual int         t8_element_maxlevel (void) = 0;

  /** Return the level of a particular element.
   * \param [in] elem    The element whose level should be returned.
   * \return             The level of \b elem.
   */
  virtual int         t8_element_level (const t8_element_t * elem) = 0;

  /** Copy all entries of \b source to \b dest. \b dest must be an existing
   *  element. No memory is allocated by this function.
   * \param [in] source The element whose entries will be copied to \b dest.
   * \param [in,out] dest This element's entries will be overwritted with the
   *                    entries of \b source.
   * \note \a source and \a dest may point to the same element.
   */
  virtual void        t8_element_copy (const t8_element_t * source,
                                       t8_element_t * dest) = 0;


  /** Compute the parent of a given element \b elem and store it in \b parent.
   *  \b parent needs to be an existing element. No memory is allocated by this function.
   *  \b elem and \b parent can point to the same element, then the entries of
   *  \b elem are overwritten by the ones of its parent.
   * \param [in] elem   The element whose parent will be computed.
   * \param [in,out] parent This element's entries will be overwritten by those
   *                    of \b elem's parent.
   *                    The storage for this element must exist
   *                    and match the element class of the parent.
   *                    For a pyramid, for example, it may be either a
   *                    tetrahedron or a pyramid depending on \b elem's childid.
   */
  virtual void        t8_element_parent (const t8_element_t * elem,
                                         t8_element_t * parent) = 0;
```

Tetrahedra

```
/** This data type stores a tetrahedron. */
typedef struct t8_dtet
{
  /** The refinement level of the tetrahedron relative to the root at level 0. */
  int8_t              level;

  /** Type of the tetrahedron in 0, ..., 5. */
  t8_dtet_type_t      type;

  t8_dtet_coord_t     x;        /**< The x integer coordinate of the anchor node. */
  t8_dtet_coord_t     y;        /**< The y integer coordinate of the anchor node. */
  t8_dtet_coord_t     z;        /**< The z integer coordinate of the anchor node. */
}
t8_dtet_t;
```

```
void
t8_default_scheme_tet_c::t8_element_parent (const t8_element_t * elem,
                                            t8_element_t * parent)
{
  const t8_default_tet_t *t = (const t8_default_tet_t *) elem;
  t8_default_tet_t   *p = (t8_default_tet_t *) parent;

  T8_ASSERT (t8_element_is_valid (elem));
  T8_ASSERT (t8_element_is_valid (parent));

  t8_dtri_cube_id_t   cid;
  t8_dtri_coord_t     h;

  T8_ASSERT (t->level > 0);

  h = T8_DTRI_LEN (t->level);
  /* Compute type of parent */
  cid = compute_cubeid (t, t->level);
  parent->type = t8_dtri_cid_type_to_parenttype[cid][t->type];
  /* Set coordinates of parent */
  parent->x = t->x & ~h;
  parent->y = t->y & ~h;
  parent->z = t->z & ~h;
  parent->level = t->level - 1;
}
```

Low-level API

Virtual base class

```
/** This struct holds virtual functions for a particular element class. */
struct t8_eclass_scheme
{
  /** This scheme defines the operations for a particular element class. */
protected:
public:
   t8_eclass_t eclass;      /**< The element class */

  /** Return the maximum allowed level for any element of a given class.
   * \return                      The maximum allowed level for elements of class \b ts.
   */
  virtual int         t8_element_maxlevel (void) = 0;

  /** Return the level of a particular element.
   * \param [in] elem    The element whose level should be returned.
   * \return             The level of \b elem.
   */
  virtual int         t8_element_level (const t8_element_t * elem) = 0;

  /** Copy all entries of \b source to \b dest. \b dest must be an existing
   *  element. No memory is allocated by this function.
   * \param [in] source The element whose entries will be copied to \b dest.
   * \param [in,out] dest This element's entries will be overwritted with the
   *                   entries of \b source.
   * \note \a source and \a dest may point to the same element.
   */
  virtual void        t8_element_copy (const t8_element_t * source,
                                       t8_element_t * dest) = 0;


  /** Compute the parent of a given element \b elem and store it in \b parent.
   *  \b parent needs to be an existing element. No memory is allocated by this function.
   *  \b elem and \b parent can point to the same element, then the entries of
   *  \b elem are overwritten by the ones of its parent.
   * \param [in] elem   The element whose parent will be computed.
   * \param [in,out] parent This element's entries will be overwritten by those
   *                   of \b elem's parent.
   *                   The storage for this element must exist
   *                   and match the element class of the parent.
   *                   For a pyramid, for example, it may be either a
   *                   tetrahedron or a pyramid depending on \b elem's childid.
   */
  virtual void        t8_element_parent (const t8_element_t * elem,
                                         t8_element_t * parent) = 0;
```

Hexahedra

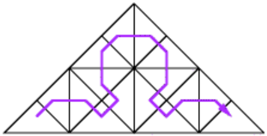
```
/** The structure holding a hexahedral element in the default scheme.
 * We make this definition public for interoperability of element classes.
 * We might want to put this into a private, scheme-specific header file.
 */
typedef p8est_quadrant_t t8_phex_t;
```

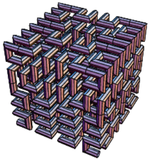
```
void
t8_default_scheme_hex_c::t8_element_parent (const t8_element_t * elem,
                                              t8_element_t * parent)
{
  T8_ASSERT (t8_element_is_valid (elem));
  T8_ASSERT (t8_element_is_valid (parent));
  p8est_quadrant_parent ((const p8est_quadrant_t *) elem,
                         (p8est_quadrant_t *) parent);
}
```

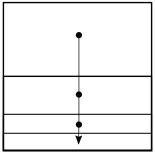
Arbitrary element types

Do whatever you want

The decoupling of high- and low-level functions allows the user to implement their own refinement patterns and SFCs.



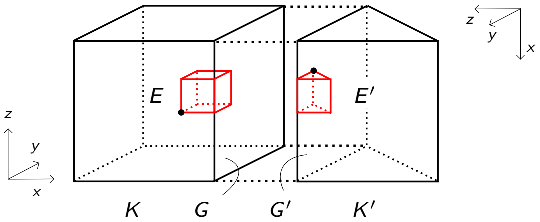




Sources: M. Bader: Space-Filling Curves,
`http://mathworld.wolfram.com/HilbertCurve.html`

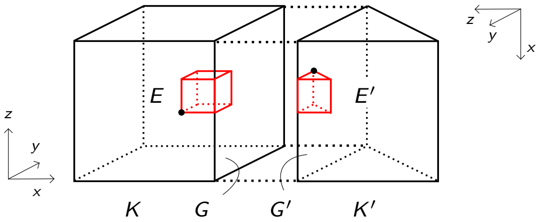
Element type independency introduces some challenges

Example: Face-neighbors across tree boundaries



Element type independency introduces some challenges

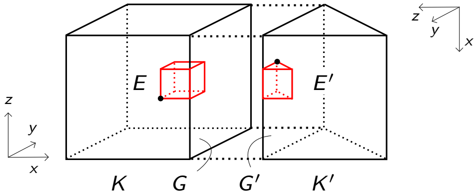
Example: Face-neighbors across tree boundaries



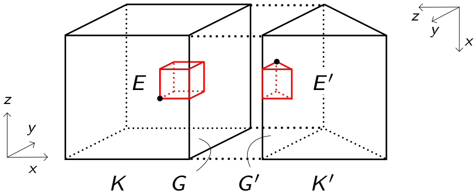
Avoid couplings

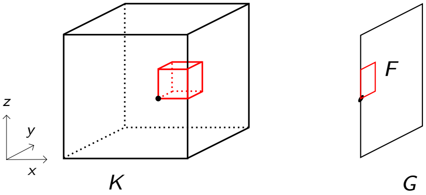
Hex↔Prism
Prism↔Tet
Quad↔Tri
⋮

Example: Face-neighbors across tree boundaries



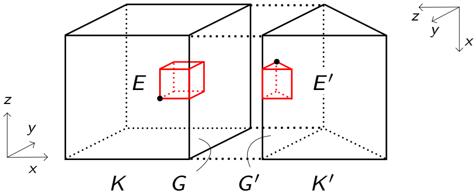
Example: Face-neighbors across tree boundaries

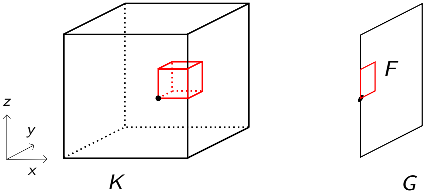




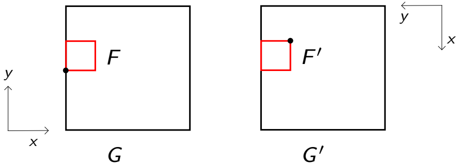
1: Build 2D boundary element

Example: Face-neighbors across tree boundaries



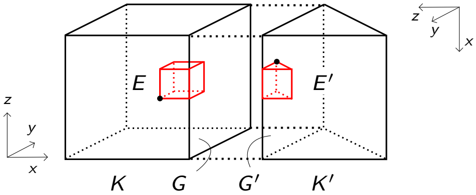


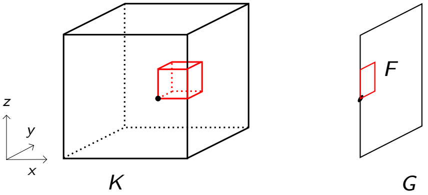
1: Build 2D boundary element



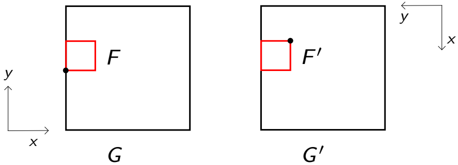
2: Transform coordinates

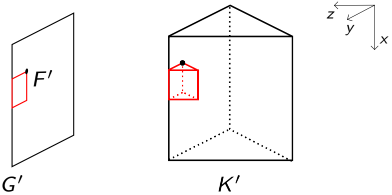
Example: Face-neighbors across tree boundaries





1: Build 2D boundary element





2: Transform coordinates

3: Extrude boundary element

Thank you for your attention.

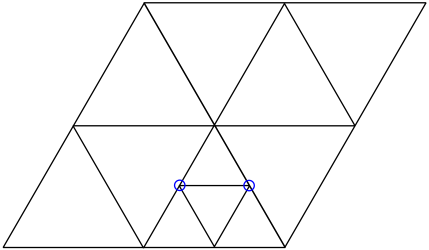
- `p4est.org`
- `github.com/cburstedde/p4est`
- `github.com/holke/t8code`
- My Thesis *Scalable Algorithms for Parallel Tree-based Adaptive Mesh Refinement with General Element Types* on Arxiv

Do not be afraid of hanging nodes

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`.

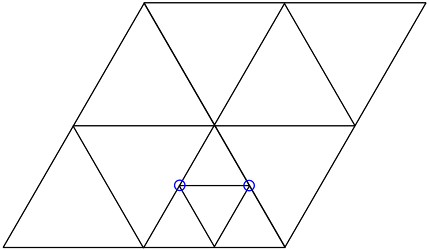


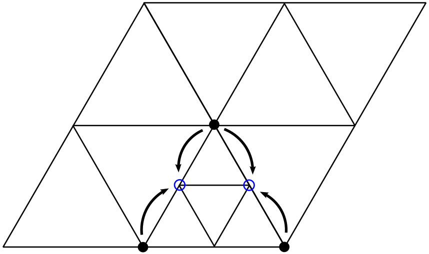
Do not be afraid of hanging nodes

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`.





Do not be afraid of hanging nodes

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`.