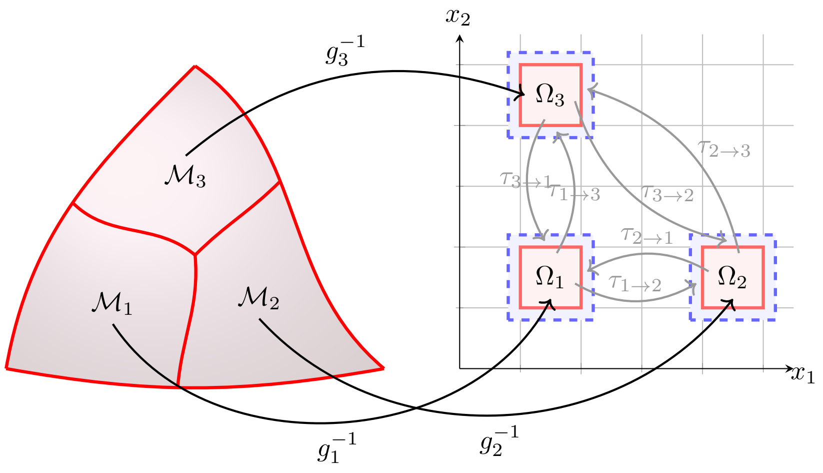
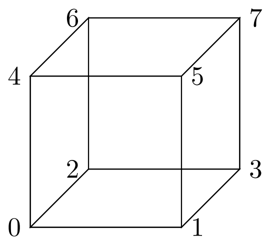
6 Forest of Octrees

To allow more complex topological meshes than are currently supported[5], we would like to go to a forest of octrees type approach. We can then think of a topological macromesh of trees that share faces, edges, and nodes. The goal is to support an arbitrarily connected hexahedral macrosmesh, which implies that trees can have one tree neighbor per face, but an arbitrary number of edge and node connections (including zero even on internal elements). Additionally, the logical coordinate systems of the trees do not necessarily align, so it is necessary to encode information about the relative connectivity of the blocks with shared elements.

A forest of octrees covering a manifold $\mathcal{M}$ is defined by a set of nodes, $\mathcal{N}$, and a set of faces, $\mathcal{F}$ (in 2D) or cells $\mathcal{C}$ (in 3D) that each contain an octree. Each face corresponds to a 4-tuple of nodes, f_i and each cell corresponds to an 8-tuple of nodes, c_i. For now, we do not attach any further geometric information to the forest. Rather, we just assume that there exist integer grid maps $g_i : \Omega_i \to \mathcal{M}_i$, where $\mathcal{M}_i \in \mathcal{M}$ is the region of the manifold covered by face (cell) i and $\Omega_i \in \Omega$ is a square (cubic) region of an integer grid Ω that is bounded by integer grid lines (planes) separated by one unit. The ghost halos and interiors of the the trees are connected by logical coordinate transformations $\tau_{i\to j}$.



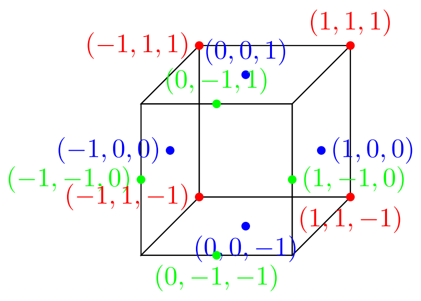
By construction, these maps take the nodes of a 4-tuple (8-tuple) defining a face (cell) to the corners of a square (cube) in morton-order. We define logical coordinates to be coordinates defined on this integer grid[6]. The index of each node in an 8-tuple defining a block are



The node indices of a 4-tuple defining a face are just the indices of the bottom face. We also require offsets relative to the cell center that are defined by 3-tuples of integers, some examples of offsets are

[5]The `Athena++` implementation of octree AMR limits domains to having a logically hyper-rectangular shape.

[6]Formally it is probably necessary to define numerous coordinate systems on the integer grid, both associated with each block (which differ by only a translation) and with each octree level in a block (which differ by a scaling factor of 2^ℓ where ℓ is the refinement level of the octree), but we aren't explicit about that here.



where offsets corresponding to faces are blue, edges are green, and nodes are red.

Each refinement level, ℓ, of the octree has a coordinate system that runs from 0 to 2^ℓ from one side of the face to the other in each direction. We refer to positions one unit outside of $[0, 2^\ell]^D$ as the ghost halo of the tree. A `LogicalLocation` as defined in `Parthenon` is an integer D-tuple at level ℓ defining the position of the lower left hand corner of a unit cube. If this cube is in the interior of the tree, this level and location can be transformed into a Morton number. This gives a nearly perfect hash for `LogicalLocation`, which we use to allow us to represent trees as hash maps of leaf node `LogicalLocation`s using `std::unordered_map`.

Similarly to blocks in an octree, it is necessary to allow for each octree itself to have a ghost halo. If a block is refined at the boundary of one octree, this can trigger refinement in a neighboring octree as a result of demanding a properly nested grid. The simplest way to deal with this is to transform the logical location of the newly refined block in the logical coordinate system of the origin block to the logical location in the logical coordinate system of the neighbor block. Therefore, it is necessary to find transformation maps $\vec{u}_{j,\ell} = \tau_{i\to j}(\vec{u}_{i,\ell})$. Here $\vec{u}_{i,\ell}$ is a coordinate vector in the logical coordinate system of block i at level ℓ.

6.1 Forest Neighbor Connectivity

To build a `Mesh` based on a general forest, the unstructured macro-grid of octrees in the forest must first be constructed. In 2D, this macro grid consists of a set of `std::shared_ptr<forest::Node>`s and a set of `std::shared_ptr<forest::Face>`s constructed from 4-tuples of these nodes. This pointer graph is then used to find the neighboring faces and logical coordinate transformations of every face in the forest and associate boundary conditions with edges that are unshared. Then, a `forest::Forest` object can be built that includes a set of `forest::Tree`s that store the neighbor connection information. `Mesh` contains a constructor that takes a `ForestDefinition` (which just contains a list of faces, boundary conditions, initial refinement locations, and the physical coordinates of each tree) and builds a `Mesh` based on this general topology. Since the macro-mesh is not adaptive, this procedure occurs only once at the beginning of a simulation (or at the beginning of a restart).

6.1.1 Finding Neighbors

Finding node, edge and face neighbors in the forest is very straightforward. First, associate every face that contains a given node with that node (in practice this is done by giving each node a vector of pointers to faces that contain it). Then for each face (cell) i, go through each node in f_i and get the set of associated faces $\{f_j\}/f_i$ (cells $\{c_j\}/c_i$). Take the intersection of f_i with each f_j. If there is one node in the intersection, f_j is a corner neighbor, if there are two nodes in the intersection and they correspond to an edge of face i and of face j the two are edge neighbors, etc. There are a number of checks that can be done here for the realizability of the mesh.

6.1.2 Finding face neighbor coordinate transforms

We define a map that takes a node in a face and returns its index in the 4-tuple defining the face

$$\mathcal{I}_{f_i} : f_i \to \{0, 1, 2, 3\}. \tag{16}$$

Because of our choice of Morton ordering of the corners, it is easy to see by inspection that the direction of an oriented edge $(n_1, n_2) \in \mathcal{N}^2$ is given by

$$d_i(n_1, n_2) = \mathrm{sign}(\mathcal{I}_{f_i}(n_2) - \mathcal{I}_{f_i}(n_1)) \left[1 + \log_2 |\mathcal{I}_{f_i}(n_2) - \mathcal{I}_{f_i}(n_1)|\right]. \tag{17}$$

If $\mathrm{frac}(d(n_1, n_2)) \neq 0$, then this is not an edge of the face (i.e. n_1 and n_2 are on opposite corners of the face), otherwise the edge is aligned in the $\hat{\imath}$ direction if $|d| = 1$ or the $\hat{\jmath}$ direction if $|d| = 2$ (or the $\hat{k}$ direction if $|d| = 3$ in a 3D forest). If $d > 0$, the edge points in the same direction as the associated unit vector, otherwise it points in the opposite direction.

Define the offsets of the nodes of a face as is normally done in Parthenon (e.g. the node at position zero in the tuple has offset $(-1, -1, -1)$ from the center of the tree in 3D). Then, the offset of any component of any component of a cell or face defined from an n-tuple can be found from

$$\vec{O}_{f_i}[(n_0, ..., n_{n-1})] = \frac{1}{n}\sum_{j=0}^{n-1} \mathrm{offset}(\mathcal{I}_{f_i}(n_j)). \tag{18}$$

$\vec{O} \cdot \vec{O}$ can easily be checked to make sure that chosen set of nodes actually corresponds to a face (so that $\vec{O} \cdot \vec{O} = 1$) or an edge (so that $\vec{O} \cdot \vec{O} = 1$ or 2 in 2- and 3D).

Using these results, the procedure for finding the coordinate transformation from an origin octree o to a neighbor octree n

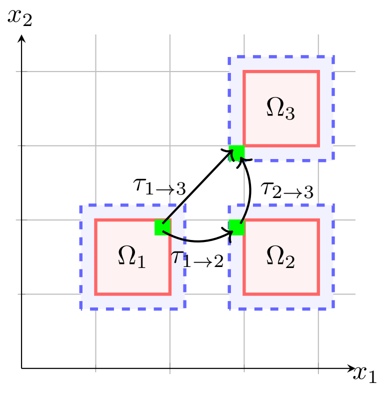
$$\vec{u}_{n,\ell} = \tau_{o\to n}(\vec{u}_{o,\ell}) = R_{o\to n}(\vec{u}_{o,\ell} - 2^\ell \vec{t}), \tag{19}$$

where $R_{o\to n}$ is a 3x3 orthogonal matrix contained in the full octahedral symmetry group (which has 48 members). R is not explicitly built in `Parthenon`, rather it is implicitly stored in arrays mapping directions in one coordinate system to directions in the other in the class `forest::LogicalCoordinateTransformation`. The translation vector $\vec{t}$ is defined for coordinate systems at $\ell = 0$. This transformation map is associated with a shared edge (n_a, n_b) or a shared face (n_a, n_b, n_c, n_d) is

1. Re-orient the shared edge by sorting the tuple based on $\mathcal{I}_{f_o}(n_i)$ so that $(n_a, n_b) \to (n_0, n_1)$ or $(n_a, n_b, n_c, n_d) \to (n_0, n_1, n_2, n_3)$.
2. Set the tangential part of the coordinate transformation by associating $d_o(n_0, n_1)$ with $d_n(n_0, n_1)$. In 3D, also do this by associating $d_o(n_0, n_2)$ with $d_n(n_0, n_2)$. (note that both d_o are guaranteed to be positive by virtue of the sorting in step one)
3. Set the normal part of the coordinate transformation by associating the direction of the non-zero component of $\vec{O}_{f_o}[(n_0, ...)]$ with the direction of the non-zero component of $\vec{O}_{f_i}[(n_0, ...)]$ and setting the sign of the direction positive if the offsets have different sign and negative otherwise.
4. Set the translation $\vec{t} = \vec{O}_{f_o}[(n_0, ...)]$.

6.1.3 Finding node neighbor coordinate transforms (2D)

Because node neighbors only share a single node, there is not enough information to determine a coordinate transformation from one face to the other alone. Nevertheless, if another face also includes the node and is an edge neighbor to both faces that are corner neighbors, the coordinate transformation between the node neighbors can be found by composition:



The composition of the coordinate transforms from the origin block o to the share edge neighbor n' followed by the transformation from n' to the corner neighbor n is given by

$$\tau_{o\to n}(\vec{u}_{o,\ell}) = \tau_{n'\to n}(\tau_{o\to n'}(\vec{u}_{o,\ell})) = R_{n'\to n}R_{o\to n'}[\vec{u}_{o,\ell} - 2^\ell(\vec{t}_{o\to n'} + R^{-1}_{o\to n'}\vec{t}_{n'\to n})]. \tag{20}$$

so that

$$R_{o\to n} = R_{n'\to n} R_{o\to n'} \tag{21}$$

$$t_{o\to n} = \vec{t}_{o\to n'} + R^{-1}_{o\to n'} \vec{t}_{n'\to n}. \tag{22}$$

Therefore, it becomes a straightforward task to build node neighbor coordinate transforms. After sweeping through all faces and calculating and storing edge neighbor transformations, a second sweep through all faces is performed looking for node neighbors. Other faces associated with the shared node are then checked to see if both of the node neighbors are their edge neighbors. Then compositions of the two related coordinate transformations are performed using the routine `forest::ComposeTransformations`. Note that if we allow for more than five-valent nodes, some transformations require more than a single composition (but the translation should be neglected after the first transformation).

6.1.4 Boundary communication in general forests

When transfering ghost data between blocks that are on separate trees, the buffer can be packed in a different order of positions than the receiving block expects because of the logical coordinate transformation between the trees. To fix this issue, when a receiving block calculates the index space into which it is receiving data it calculates the index space in it own coordinate system then transforms this index range back into the coordinate system of the neighbor block (this transformation only uses the coordinate rotation $R^{-1}_{o\to n}$ since the translation is already taken care of in the indexing routines). Then, the flattened version of this transformed index space corresponds correctly to the one-dimensional index space of the packed buffer. When the buffer is unpacked, the position of each point in the buffer is transformed back to a position in the logical coordinates of the receiving block using $R_{o\to n}$. In this way, boundary data is communicated into the correct positions.

Additionally, the logical coordinate transformations should act on the components of tensorial objects, like some face and edge fields as well as objects with `Metadata::Vector` and `Metadata::Tensor`. The code for doing these transformations exists and face- and edge- variables are properly transformed, but transformations are not yet performed on `Metadata::Vector/Tensor` variables.

Five-valent points pose a problem, since two neighbor blocks will want to write data to the corner of a block at a five-valent corner. This has not been dealt with yet in the code, so we are really limited to three- and four-valent corners for now. In the future, the extra data will be stored in a extra index of the variable and be made accessible to downstream codes. For finite-volume codes that reconstruct along coordinate directions, the corner data should never need to be directly accessed but prolongation needs to use it internally at least.