Parthenon Notes

You

June 10, 2024

Here we collect some notes about the implementation of selected aspects of `Parthenon`.

Contents

1 Forest of Octrees

To allow more complex topological meshes than are currently supported[1], 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.

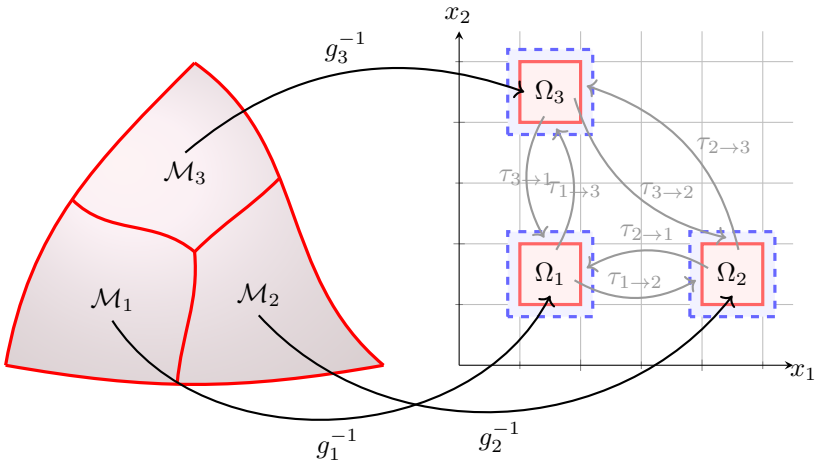
- `p4est` is a code that is similar to what we want to do, main reference is [4], others are [6, 3]
- We can get some geometric inspiration (e.g. how to deal with coordinates on different trees and their interelation) from hexahedral mesh building. Building hexahedral meshes is complex, recent review of hex meshing in [10]. [7] build meshes based on frame fields, not a great reference but where I started. [11] gives a mathematical description of n-symmetry direction fields (e.g. crosses) which are useful for meshing. [5] cool meshing algorithm based on insights from continuous differential geometry. [9] has a nice discussion of generalities of hexmesh topologies and integer grid maps. [2] widely cited reference with some more formal discussion of integer grid maps (which may be useful for thinking about how to define our coordinate charts and atlases). [8] nice description of a quadmeshing algorithm and discussion of how to define coordinate atlas and grids on discrete surfaces.

General thoughts:

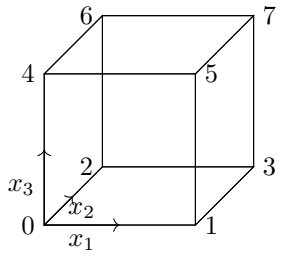
- We probably require skew coordinate systems to deal with odd valence points. For trees that are normal valence everywhere, we can probably assume our integer grid maps are conformal. What is the relationship between skew coordinates and singular points?

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

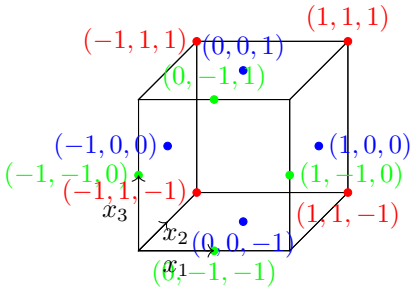
A forest of octrees covering a manifold $\mathcal{M}$ is defined by a set of nodes, $\mathcal{N}$, and a set of faces, $\mathcal{F}$ (in 2D) or cells $\mathcal{C}$ (in 3D) that each contain an octree. Each face corresponds to a 4-tuple of nodes, f_i and each cell corresponds to an 8-tuple of nodes, c_i. 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[2]. 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



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

[2]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.

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.

1.1 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).

1.1.1 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.

1.1.2 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

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

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

$$d_i(n_1, n_2) = \text{sign}(\mathcal{I}_{f_i}(n_2) - \mathcal{I}_{f_i}(n_1)) \left[1 + \log_2 |\mathcal{I}_{f_i}(n_2) - \mathcal{I}_{f_i}(n_1)|\right]. \tag{2}$$

If $\text{frac}(d(n_1, n_2)) \neq 0$, 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

$$\vec{O}_{f_i}[(n_0, ..., n_{n-1})] = \frac{1}{n} \sum_{j=0}^{n-1} \text{offset}(\mathcal{I}_{f_i}(n_j)). \tag{3}$$

$\vec{O} \cdot \vec{O}$ 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

$$\vec{u}_{n,\ell} = \tau_{o\to n}(\vec{u}_{o,\ell}) = R_{o\to n}(\vec{u}_{o,\ell} - 2^{\ell}\vec{t}), \tag{4}$$

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 $(n_a, n_b, n_c, n_d) \to (n_0, n_1, n_2, n_3)$.

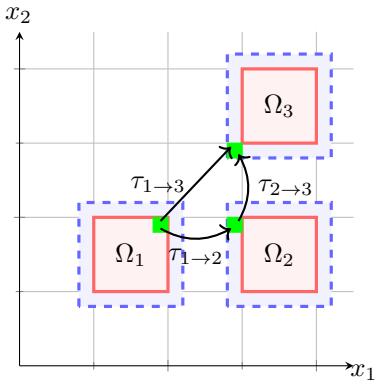
2. Set the tangential part of the coordinate transformation by associating $d_o(n_0, n_1)$ with $d_n(n_0, n_1)$. In 3D, also do this by associating $d_o(n_0, n_2)$ with $d_n(n_0, n_2)$. (note that both d_o are guaranteed to be positive by virtue of the sorting in step one)

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 $\vec{t} = \vec{O}_{f_o}[(n_0, ...)]$.

1.1.3 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 from the shared node 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

$$\tau_{o\to n}(\vec{u}_{o,\ell}) = \tau_{n'\to n}(\tau_{o\to n'}(\vec{u}_{o,\ell})) = R_{n'\to n}R_{o\to n'}[\vec{u}_{o,\ell} - 2^{\ell}(\vec{t}_{o\to n'} + R^{-1}_{o\to n'}\vec{t}_{n'\to n})]. \tag{5}$$

so that

$$R_{o\to n} = R_{n'\to n}R_{o\to n'} \tag{6}$$

$$t_{o\to n} = \vec{t}_{o\to n'} + R^{-1}_{o\to n'}\vec{t}_{n'\to n}. \tag{7}$$

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).

1.1.4 Boundary communication in general forests

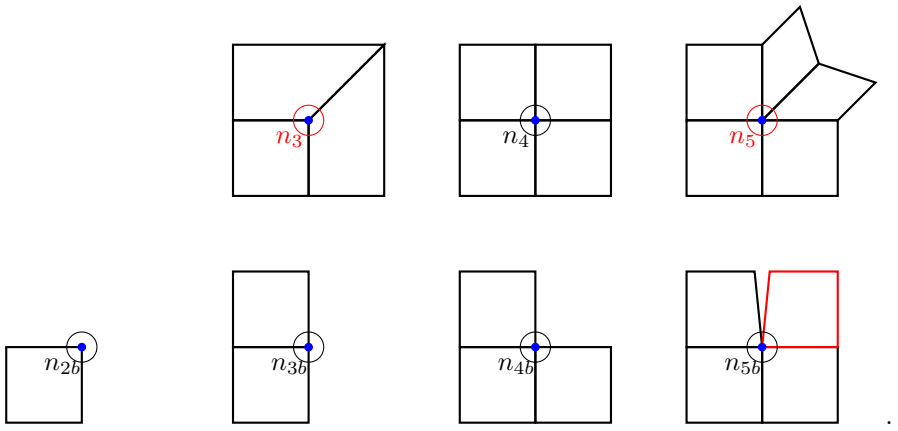
When transfering ghost data between blocks that are on separate trees, the buffer can be packed in a different order of positions than the receiving block expects due to the logical coordinate transformation between the trees. To fix this issue, when a receiving block calculates the index space into 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. Currently, all edge and face variables are assumed to be oriented so that their values can change signs in addition to having their indices permuted under some coordinate transformations. This is appropriate if the fields represent represent the normal/tangent components of vectors, but if they just represent a scalar defined on those topological elements there should not be sign changes under the transformation. We need to add a flag that indicates if a field is oriented.

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.

1.2 Singular valence points

For a 2D mesh (or an extruded 3D mesh), the allowed topological edge connectivities of each node are



It can be seen that a node can have either zero or two boundary edges that connect to it. Boundary communication across elements of blocks containing nodes of types n_4, n_{2b}, and n_{3b} is what is currently implemented in Parthenon. Nodes of type n_{4b} are non-singular, but the upper right corner boundary is not filled with using a physical boundary condition in the current infrastructure. Nodes of type n_{5b} also should more or less work with the current infrastructure, although corner data of all blocks at n_{5b} will be ambiguous

Internal nodes of type n_3 and n_5 are trickier to deal with. From a purely data communication perspective, points of type n_3 should just work, although all blocks will receive no data in the ghost corner corresponding to the n_3 node. Nevertheless, if the post communication data is used to prolongate naively using the current implementation the gradients will be calculated using invalid data in the corner, resulting in trash in the ghosts near the n_3 point. For nodes of type n_5, even data

communication does not work since two blocks will be sending data to the corner which would result in a race condition for what data is set there and would once again result in junk for prolongation.

Some thoughts about 3D meshes:

- For 3D, recognizing that a mesh defines a triangulation of an infinitesimal sphere surrounding any given internal node and using Euler's polyhedron formula, we find that the number of cells, C, adjacent to the node and the number of edges, E, emanating from the node are related by $C = 2(E-2)$ (see [9]). If we only allow 3, 4, and 5 valent internal edges, [9] shows that the maximum number of edges emanating from a node is 12 so that up to 20 cells can be associated with a node. This is too many, since a single cell associated with the node can only have three face neighbors and six edge neighbors that are all also associated with the node, leaving at least ten neighbor cells that are neighbors via the single vertex of the cell associated with the node. If we limit ourselves to seven or fewer edges emanating from a node, then there are at most three corner neighbors of any possible cell (there are in fact only five different possible topological configurations around a given node [9]).

- Another way to think about the allowable configurations is that every neighboring cell of a given cell must be accessible by stepping through at most three faces (similarly to how in 2D we demand that each neighbor face is accessible by stepping through at most two edges). This should limit the valence of the point. How does this look on the triangulation of the sphere mentioned in the previous point? Projected onto the sphere: each edge becomes a node, and each face becomes and edge, and each cell becomes a triangle. So we need to somehow figure out how to throw out triangulations that have more than three step minimal paths between any two triangles. Another way to think about this is how many ways can we triangulate the sphere where we limit ourselves to three, four, and five valent points on the sphere with the additional maximal path constraint.

- Possibly useful: Bowen & Fisk 1967 show a method for generating triangulations of the sphere from lower node count triangulations of the sphere. Probably can further constrain their algorithm and find the allowed node complexes. Boils down to $\sum_{v\in\{3,4,5,\ldots\}} N_v(6-v) = 12$ where N_v is the number of nodes with valence v in the triangulation. We would like to limit Some 3-tuples that satisfy this equation are not triangulations, so we need to generate the allowed triangulations with v

2 Geometric Multigrid

I think this is the original paper about MGCG [12]. The basic multigrid (MG) V-cycle is defined by algorithm 1. As is shown in the comment, MG represents a linear operator on the finest grid that takes the initial RHS $\vec{f}$ and returns an approximate solution on the finest level $\vec{u}$. All of the multigrid operators M_h are linear operators that approximately satisfy $M_h \approx A_h^{-1}$ (at least for good choices of $\tilde{A}_h$ or S_h). Then, we can clearly use a multigrid operator as a preconditioner

$$A_h\vec{x} = A_h M_h M_h^{-1}\vec{x} = (A_h M_h)\vec{y} = \vec{b} \tag{8}$$

with $\vec{x} = M_h\vec{y}$ or

$$(M_h A_h)\vec{x} = M_h\vec{b}. \tag{9}$$

2.1 Smoothing Operators

The smoothing operators S generally correspond to a number of iterations of some iterative solver. Here I write down some well known basics, just so I don't forget them. If we are trying to solve the matrix equation $A\vec{x} = \vec{f}$, we can split up the matrix as

$$A = P - D \tag{10}$$

so that

$$\vec{x} = \underbrace{P^{-1}\vec{f}}_{\vec{g}} + \underbrace{P^{-1}D}_{R}\vec{x} \tag{11}$$

Algorithm 1 Standard multigrid, which results in a matrix form

$$M_h = \begin{cases} \tilde{S}_{h,\text{post}} S_{h,\text{pre}} + S_{h,\text{post}} P_{2h} M_{2h} R_h (I - A_h S_{h,\text{pre}}), & \text{if } h < h_{\max} \\ A_h^{-1}, & \text{if } h \geq h_{\max}, \end{cases}$$

where $S_{h,\text{pre/post}}$ are smoothers, R_h is a restriction operator from a grid with cell-size h to a grid with cell-size $2h$, P_{2h} is a prolongation operator from $2h$ to h, and we have assumed we have a direct solver for our matrix A at the coarsest level.

procedure MULTIGRID$_h(\vec{f}_h)$
  if $h < h_{\max}$ **then**
    $\vec{u}_h \leftarrow S_{h,\text{pre}} f_h$ ▷ Requires communication of ghosts before each smoother iteration
    Communicate $\vec{u}_h$ ghosts
    $\vec{r}_h \leftarrow f_h - A_h \vec{u}_h$
    $\vec{r}_{2h} \leftarrow R_h \vec{r}_h$ ▷ Restriction doesn't reach across block boundaries
    $\vec{e}_{2h} \leftarrow \text{MultiGrid}_{2h}(\vec{r}_{2h})$
    Communicate $\vec{e}_{2h}$ ghosts
    $\vec{e}_h \leftarrow P_{2h} \vec{e}_{2h}$ ▷ Prolongation does reach across block boundaries
    $\vec{u}_h \leftarrow \vec{u}_h + \vec{e}_h$
    $\vec{u}_h \leftarrow S_{h,\text{post}} \vec{u}_h$ ▷ Requires communication of ghosts before each smoother iteration
  else
    $\vec{u}_h \leftarrow A_h^{-1} \vec{f}_h$ ▷ Direct solve on the lowest level
  end if
  return $\vec{u}_h$
end procedure

If we have an estimate $\vec{x}^{(i)}$ for the true solution $\vec{x}$, we can form a new estimate

$$\begin{aligned} \vec{x}^{(i+1)} &= \vec{g} + R\vec{x}^{(i)} \\ &= P^{-1}\left(\vec{f} + D\vec{x}^{(i)}\right), \end{aligned} \tag{12}$$

this is clearly stationary in the sense that if $\vec{x}^{(i)} = \vec{x}$, then $\vec{x}^{(i+1)} = \vec{x}^{(i)}$. We can define the error of the estimate as $\vec{\varepsilon}^{(i)} = \vec{x} - \vec{x}^{(i)}$, which gives

$$\vec{\varepsilon}^{(i+m)} = R^m \vec{\varepsilon}^{(i)} \tag{13}$$

Clearly, for iterations of R to converge to the stationary solution we must have

$$\lim_{m\to\infty} R^m \vec{\varepsilon}^{(0)} = \vec{0} = \sum_i \alpha_i \lambda_k^m \vec{\mathfrak{e}}_k \tag{14}$$

which can be stated as

$$\rho(R) = \max |\lambda_k| < 1, \tag{15}$$

where $\rho(R)$ is the spectral radius of R (here λ_k are the eigenvalues of R, $\vec{\mathfrak{e}}_k$ are the corresponding eigenvectors, and α_i are the expansion coefficients of $\vec{\varepsilon}^0$ in terms of $\vec{\mathfrak{e}}_k$). So for our relaxation method to theoretically work we require that our split is stationary and has spectral radius less than one. The first part occurs by construction, but the second part doesn't generically hold.

Assuming $\vec{x}^{(0)} = \vec{0}$, we can write

$$\vec{x}^{(m)} = \sum_{n=0}^{m} R^n \vec{g} = \underbrace{\left(\sum_{n=0}^{m} R^n\right) P^{-1}}_{S} \vec{f} \tag{16}$$

which makes it clear that this iterative procedure behaves as a linear operator.

Assume there is an exact solver for each block on on our grid that assumes any pieces of A that reach into ghost zones are zero. Then P could be taken as the combined block local matrices and D would contain the leftover terms of the matrix that couple the blocks through ghosts. Obviously on the root block $D_{\text{root}} = 0$ and we have an exact solver, so the MG hierarchy can terminate there.

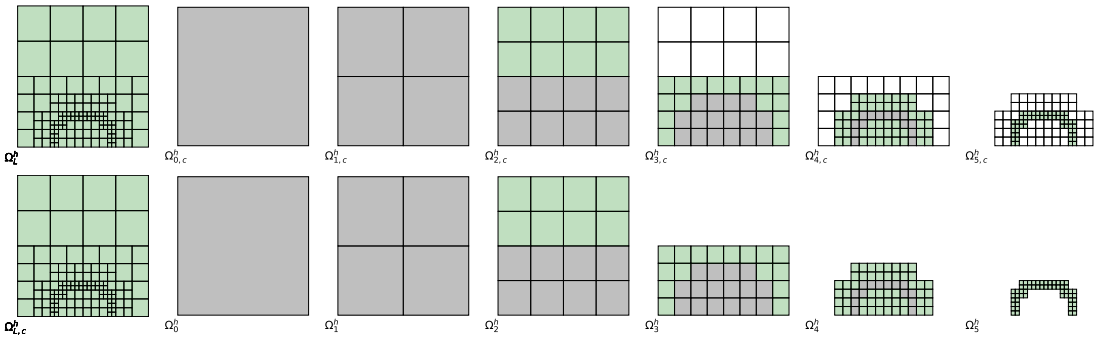


Figure 1: Diagram showing the different grids in a multi-grid grid hierarchy for an example Parthenon AMR grid. The squares correspond to blocks in the grid. We do not show grids for negative logical levels since they would appear the same as Ω_0^h (as coarsening on negative refinement levels occurs by reducing the number of zones per block rather than reducing the number of blocks). Green blocks correspond to leaf nodes, gray blocks correspond to internal nodes in the refinement tree, and uncolored blocks correspond to leaf blocks on the coarser level in a two-level composite grid. Non-course fine boundary neighbor coarse blocks have not been removed from the composite grids, even though they could be. We denote the composite grids by $\Omega_{\ell,c}^h$ instead of $\Omega_{\underline{\ell}}^h$ because of LaTeX issues in matplotlib.

2.2 Adaptive Multi-grid

Things become somewhat more complex when AMR is included and there are a number of related methods for Adaptive Multi-grid. We use an approach similar to that described in [13] Chapter 9 where finer levels do not cover the entire grid. To discuss our algorithm, we need to first formally define the grids that we are working on. Let the domain of our problem (w/o boundary) be Ω and the boundary of the domain be $\partial\Omega$. We denote a cell-centered grid covering a given domain Ω_x as Ω_x^h (and we are generally not careful about explicitly specifying the grid spacing h at different levels). We denote the same grid but including ghost zones as $\bar{\Omega}_x^h$. Then we can define some objects:

- *Leaf Grid* Ω_L^h: We refer to the pre-existing grid in Parthenon as the leaf grid since it includes all blocks corresponding to leaf nodes in the octree and no blocks corresponding to internal nodes of the refinement octree. Obviously $\Omega_L \cap \Omega = \Omega$.
- *Single-level Grid* Ω_ℓ^h: A single-level grid for refinement level ℓ includes blocks corresponding to all leaf *and internal* nodes of the octree at logical level ℓ. Clearly, for $\ell > \ell_r$ where ℓ_r is the level of the Parthenon root grid, we can have $\Omega_\ell \cap \Omega \neq \Omega$.
- *Two-level Composite Grid* $\Omega_{\underline{\ell}}^h$: A two-level composite grid for level ℓ that contains all blocks on level ℓ and leaf node blocks on level $\ell - 1$ that have neighbors on level ℓ.[3] To be clear, in our notation in the sub-region $\Omega_{\underline{\ell}} \cup \Omega_\ell$ only cell-centered points from Ω_ℓ^h are present while in $\Omega_{\underline{\ell}} \setminus \Omega_\ell$ only cell-centered points from $\Omega_{\ell-1}^h$ are present.

An example grid hierarchy is shown in figure 1.

We denote a field on level ℓ by $\vec{x}_\ell$. Such a field is defined over all of Ω_ℓ^h and has ghost cells in $\bar{\Omega}_\ell^h \setminus \Omega_\ell^h$. Such a field is only defined once, e.g. the value of $\vec{x}_\ell$ on overlapping regions Ω_ℓ^h, $\Omega_{\underline{\ell}}^h$, $\Omega_{\underline{\ell-1}}^h$, and Ω_L^h is always the same. One consequence of this is that setting the value of a field $\vec{x}_{\underline{\ell}}$ sets it on $\vec{x}_\ell$ as well. Said in terms of the Parthenon implementation, there is only one block allocated for each node in the tree and multiple blocklists corresponding to different grids can point to the same block.

The FAS based adaptive MG algorithm that is implemented in Parthenon is described in algorithm 2. I *think* this is essentially similar to the Fast Adaptive Composite (FAC) algorithm of McCormick, maybe even exactly the same, but it is challenging to translate some of the literature to cell-centered schemes and the block-based AMR language. Some specific details of the algorithm are:

[3] Currently, the implementation in Parthenon includes all leaf blocks on level $\ell - 1$ in $\Omega_{\underline{\ell}}^h$ but only because including them doesn't impact correctness and it was easier to build the block lists this way. There may be a slight performance boost if only the neighbor blocks are included, but communication on $\Omega_{\underline{\ell}}^h$ is currently set up so only coarse to fine communication occurs for level $\ell - 1$ blocks (and not level $\ell - 1$ block to $\ell - 1$ block communication).

Algorithm 2 V-cycle in the Full Approximation Scheme (FAS)

procedure SMOOTH$_\ell(\vec{u}_{\underline{\ell}},\ \vec{f}_{\underline{\ell}})$ ▷ See note 1.
  Boundary communication over $\Omega^h_{\underline{\ell}}$ ▷ See note 2.
  $\vec{t}_\ell \leftarrow A_\ell \vec{u}_\ell \in \Omega^h_\ell$ ▷ See note 3.
  $\vec{u}_\ell \leftarrow (1-\omega)D_\ell^{-1}\left(\vec{f}_\ell + D_\ell \vec{u}_\ell - \vec{t}_\ell\right) + \omega \vec{u}_\ell \in \Omega^h_\ell$ ▷ $\omega = 0$ for undamped Jacobi.
  return $\vec{u}_\ell$
end procedure
procedure FAS$_\ell(\vec{u}_{\underline{\ell}},\ \vec{f}_{\underline{\ell}})$
  $\vec{u}_\ell \leftarrow \text{SMOOTH}_\ell(\vec{u}_{\underline{\ell}}, \vec{f}_{\underline{\ell}}) \in \Omega^h_\ell$
  if $\ell = \ell_{min}$ **then**
    return $\vec{u}_\ell$
  end if
  $\vec{r}_\ell \leftarrow \vec{f}_\ell - A_\ell \vec{u}_\ell \in \Omega^h_\ell$
  $\vec{u}_{\underline{\ell-1}} \leftarrow \mathcal{R}^\ell_{\ell-1}\vec{u}_\ell \in (\Omega^h_{\underline{\ell-1}} \cap \Omega_\ell)$
  $\vec{u}_{\underline{\ell-1}} \leftarrow \vec{u}_{\underline{\ell-1},0} \in (\Omega^h_{\underline{\ell-1}} \setminus \Omega_\ell)$ ▷ See note 4.
  $\vec{f}_{\underline{\ell-1}} \leftarrow A_{\ell-1}\vec{u}_{\ell-1} + \mathcal{R}^\ell_{\ell-1}\vec{r}_\ell \in (\Omega^h_{\underline{\ell-1}} \cap \Omega_\ell)$ ▷ See note 5.
  $\vec{f}_{\underline{\ell-1}} \leftarrow \vec{f}_{\underline{\ell-1},0} \in (\Omega^h_{\underline{\ell-1}} \setminus \Omega_\ell)$ ▷ See note 4.
  $\vec{v}_{\ell-1} \leftarrow \text{FAS}_{\ell-1}(\vec{u}_{\underline{\ell-1}}, \vec{f}_{\underline{\ell-1}}) \in \Omega^h_{\ell-1}$
  $\vec{e}_{\ell-1} \leftarrow \vec{v}_{\ell-1} - \vec{u}_{\ell-1} \in \Omega^h_{\ell-1}$
  $\vec{u}_{\underline{\ell}} \leftarrow \vec{u}_{\underline{\ell}} + \mathcal{P}^{\ell-1}_{\underline{\ell}}\vec{e}_{\ell-1} \in \Omega^h_\ell$
  $\vec{u}_\ell \leftarrow \text{SMOOTH}_\ell(\vec{u}_\ell, \vec{f}_\ell) \in \Omega^h_\ell$
  return $\vec{u}_\ell$
end procedure

1. We write the smoother as a Jacobi iteration for simplicity of presentation, but in practice we usually use multiple scheduled Jacobi iterations where ω varies in a prescribed way with each iteration [14].

2. We fill all boundaries of blocks at level ℓ contained in $\Omega^h_{\underline{\ell}}$. Prolongating values from blocks on level $\ell-1$ to the ghost zones of blocks on level ℓ is key to achieving good convergence[4] with the multilevel algorithm. Since the ℓ boundary values are interpolated, they depend on both non-ghost cells in the neighboring $\ell-1$ block *and* non-ghost cells in the block on ℓ itself (think of the prolongation stencil).

3. The operator A_ℓ only returns an answer in the grid Ω^h_ℓ but it will almost always reach into ghost values in $\bar{\Omega}^h_\ell \setminus \Omega^h_\ell$. Because of interpolation into these coarse-fine boundaries, this introduces further dependence on values in Ω^h_ℓ near the boundary, as well as dependence on $\Omega^h_{\ell-1} \setminus \Omega_\ell$. As a result the true diagonal of the matrix may get an extra contribution from where the single level representation of A (that is A_ℓ) reaches into the boundaries. We have found that it is likely safe to neglect this extra contribution to D and use the value of D that would be found for a uniform grid.

4. Steps updating coarse zones on two-level composite grids are not explicitly included in the task list, since it is already performed for all levels by initializing $\vec{u}_L \leftarrow \vec{u}_{0,L}$ (and similarly for $\vec{f}_L$ before starting the FAS).

[4] Initially, we just tried setting the ghost zones of fine blocks at coarse-fine boundaries on composite grids to a constant value given by a initial communication over the entire leaf grid. Then, when prolongating the error from a coarser grid $\Omega^h_{\ell-1}$ to a finer grid Ω^h_ℓ, we reset the coarse-fine boundaries of the fine grid based on prolongating from leaf and internal blocks on $\Omega^h_{\ell-1}$. In both cases the coarse-fine boundaries were not updated during the smoothing operations. This resulted in an MG algorithm that converged similarly to the algorithm we settled on when doing raw V-cycles, but when used as a single v-cycle preconditioner with BiCGStab this algorithm failed miserably (i.e. it often did not converge at all on AMR problems that both MG alone and BiCGStab alone would converge on). It appears that the effective lag in updating the coarse-fine boundaries resulted in large residuals on the fine grid next to the coarse-fine boundaries.

5. We can cast this in terms of τ corrections $A_{\ell-1}\vec{u}_{\ell-1}+\mathcal{R}^{\ell}_{\ell-1}\vec{r}_{\ell} = \mathcal{R}^{\ell}_{\ell-1}\vec{f}_{\ell}+\tau^{\ell}_{\ell-1}$ where the correction to the restricted RHS is $\tau^{\ell}_{\ell-1} = A_{\ell-1}\mathcal{R}^{\ell}_{\ell-1}\vec{u}_{\ell} - \mathcal{R}^{\ell}_{\ell-1}A_{\ell}\vec{u}_{\ell}$. We can then consider the τ correction of the composite grid $\tau^{\underline{\ell}}_{\ell-1} = A_{\ell-1}\mathcal{R}^{\underline{\ell}}_{\ell-1}\vec{u}_{\underline{\ell}} - \mathcal{R}^{\underline{\ell}}_{\ell-1}A_{\underline{\ell}}\vec{u}_{\underline{\ell}}$, where $\mathcal{R}^{\underline{\ell}}_{\ell-1}$ is just the identity outside of Ω^h_{ℓ} and rows of $A_{\underline{\ell}}$ outside Ω^h_{ℓ} that do not reach into the fine region are the same as $A_{\ell-1}$. In fact for Poisson like problems, if we don't include flux correction, $A_{\underline{\ell}} = A_{\ell-1} \in \Omega^h_{\ell-1} \setminus \Omega_{\ell}$ (for rows). That suggests that if we do want to include flux correction in MG, it may be best done through a τ correction. Obviously, flux correction is not present in the single level operator A_{ℓ} but there would be a contribution from flux correction to $A_{\underline{\ell}}$ on the coarse side of the coarse-fine boundary inside the two-level grid. This would result in a non-zero value for $\tau^{\underline{\ell}}_{\ell-1}$ at the internal boundary of $\Omega^h_{\ell-1} \setminus \Omega_{\ell}$ and a slight correction to the input RHS on coarse-fine boundary leaf blocks on level $\ell - 1$. I also think that after every v-cycle, $\vec{f}_L$ would have to be reset to $\vec{f}_{0,L}$. An original reference for this is [1]. That being said, it appears that using un-flux corrected MG as a preconditioner for flux-corrected BiCGStab seems to work just fine so this addition seems unnecessary for the time being.

Some notes about the specific implementation in Parthenon:

1. Restriction is performed using the standard Parthenon `RestrictAverage` method, which is clearly linear. The default shared prolongation operator is not linear, since it is slope limited, so we use the prolongation operator `ProlongateSharedLinear` that performs bi/trilinear interpolation from a coarse grid to a fine grid. Fields involved in MG need to have this (or another) linear prolongation operator registered for the adaptive MG procedure to be linear. The default internal prolongation operator, `ProlongateInternalAverage`, is linear so no change is required there (and it also doesn't play a role for cell-centered fields). The linear prolongation operator should work with fields of any topological type.

2. Physical boundary conditions need to be carefully constructed so that they are implemented consistently on $\partial\Omega$ at all levels of the grid. Setting the ghosts to a constant value will not work since the position of the ghost points on the physical boundaries changes as the refinement level is changed. For Dirichlet boundary conditions, we find the choosing ghost points so that the value of the field linearly interpolated to the face between the ghost and the first active zone satisfies the Dirichlet condition. For nodal fields and face fields, it may be better to impose boundary conditions on the first layer of internal zones that sit on the boundary face[5].

3. Below the Parthenon root grid, we coarsen as many times as possible de-refining by a factor of two. The number of possible de-refinements from the root grid, α, is given by the minimum α_d amongst all directions d where α_d for each direction is the the maximum value of α_d for which $(n_{r,d} \times n_{z,d}) \bmod 2^{\alpha_d} = 0$. Here $n_{r,d}$ is the number of blocks in the d direction at the level of the root grid and $n_{z,d}$ is the number of zones in the d direction in blocks at or finer than the root grid level. Above the root grid, this can result in blocks that are not all the same shape, even across a single refinement level[6]. If α is larger than the level of the root grid, this leads us to define "negative" logical levels within our grid hierarchy. At logical level zero, a single block covers Ω. For negative levels, a single block still covers Ω but this block will have a factor of two coarser zones in every direction than the block on the next higher level.

2.3 Results

References

[1] D. Bai and A. Brandt. Local mesh refinement multilevel techniques. *SIAM Journal on Scientific and Statistical Computing*, 8(2):109–134, 1987. arXiv:https://doi.org/10.1137/0908025, doi:10.1137/0908025.

[5] There is probably room for some careful thinking about how we want to impose boundary conditions on non-cell centered fields in general.

[6] Dealing with this possibility generically requires the use of hierarchical parallelism within Parthenon kernels that loop over blocks, since the spatial index range can change amongst blocks in a single `MeshData` object. Obviously this is not required if grid sizes are chosen judiciously so blocks on a single level always have the same shape (even if the block size varies between refinement levels)

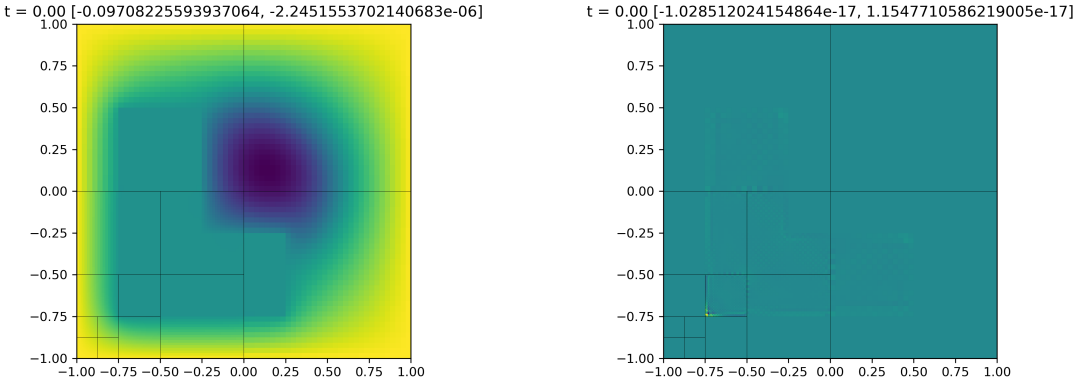


Figure 2: Example solution and residuals found for a discretized version of the equation $\nabla \cdot D\nabla u = f$, where $D = 1000$ in the L-shaped region and $D = 1$ elsewhere. The source function f is one inside a radius of 0.5 and zero elsewhere and $u = 0 \in \partial\Omega$.

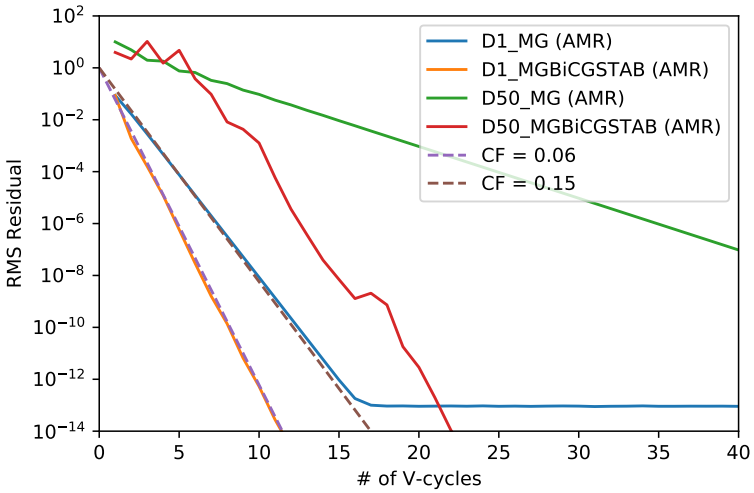


Figure 3: Convergence properties of the MG implementation in Parthenon for the problems similar to that shown in figure 2. D1 corresponds to $D = 1$ throughout the whole grid, while D50 corresponds to $D = 50$ within the L-shaped region.

[2] David Bommes, Marcel Campen, Hans-Christian Ebke, Pierre Alliez, and Leif Kobbelt. Integer-grid maps for reliable quad meshing. *ACM Trans. Graph.*, 32(4), jul 2013. doi:10.1145/2461912.2462014.

[3] Carsten Burstedde. Parallel tree algorithms for AMR and non-standard data access. *ACM Transactions on Mathematical Software*, 46(32):1–31, November 2020. doi:10.1145/3401990.

[4] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, January 2011. doi:10.1137/100791634.

[5] Etienne Corman and Keenan Crane. Symmetric moving frames. *ACM Trans. Graph.*, 38(4), jul 2019. doi:10.1145/3306346.3323029.

[6] Tobin Isaac, Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, 2015. doi:10.1137/140970963.

[7] N. Kowalski, F. Ledoux, and P. Frey. Block-structured hexahedral meshes for cad models using 3d frame fields. *Procedia Engineering*, 82:59–71, 2014. 23rd International Meshing Roundtable (IMR23). URL: https://www.sciencedirect.com/science/article/pii/S187770581401652X, doi:https://doi.org/10.1016/j.proeng.2014.10.373.

[8] Felix Kälberer, Matthias Nieser, and Konrad Polthier. Quadcover - surface parameterization using branched coverings. *Computer Graphics Forum*, 26(3):375–384, 2007. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01060.x, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01060.x, doi:https://doi.org/10.1111/j.1467-8659.2007.01060.x.

[9] Heng Liu, Paul Zhang, Edward Chien, Justin Solomon, and David Bommes. Singularity-constrained octahedral fields for hexahedral meshing. *ACM Trans. Graph.*, 37(4), jul 2018. doi:10.1145/3197517.3201344.

[10] Nico Pietroni, Marcel Campen, Alla Sheffer, Gianmarco Cherchi, David Bommes, Xifeng Gao, Riccardo Scateni, Franck Ledoux, Jean Remacle, and Marco Livesu. Hex-mesh generation and processing: A survey. *ACM Trans. Graph.*, 42(2), oct 2022. doi:10.1145/3554920.

[11] Nicolas Ray, Bruno Vallet, Wan Chiu Li, and Bruno Lévy. N-symmetry direction field design. *ACM Trans. Graph.*, 27(2), may 2008. doi:10.1145/1356682.1356683.

[12] Osamu Tatebe. The multigrid preconditioned conjugate gradient method. 1993.

[13] Ulrich Trottenberg, Cornelis W. Oosterlee, and Anton Schüller. *Multigrid*, volume 33 of *Texts in Applied Mathematics. Bd.* Academic Press, San Diego [u.a.], 2001. With contributions by A. Brandt, P. Oswald and K. Stüben.

[14] Xiang Yang and Rajat Mittal. Efficient relaxed-Jacobi smoothers for multigrid on parallel computers. *Journal of Computational Physics*, 332:135–142, March 2017. doi:10.1016/j.jcp.2016.12.010.