Using a Multiblock Hierarchy in *SAMRAI*

Noah S. Elliott

1 Introduction

The multiblock functionality in *SAMRAI* is intended to allow the use of *SAMRAI*'s structured AMR infrastructure on problem domains that have one or more singularity points of reduced or enhanced connectivity but can be decomposed into logically rectangular subdomains, or blocks. This document describes the steps needed to set up a multiblock domain, how the hierarchy that represents the domain can be used, and finally the usage of multiblock-specific communication classes in *SAMRAI*.

2 Creating a `MultiblockPatchHierarchy`

This section describes how to set up an object of class `MultiblockPatchHierarchy`, which manages a domain for a multiblock problem. Figure 1 shows an example domain, consisting of five blocks, that will be used for reference in this section.

2.1 Defining the Index Spaces for the Blocks

`MultiblockPatchHierarchy` manages a domain for a multiblock problem, and should be used in a manner analagous to the way that `PatchHierarchy` is used on an ordinary rectangular domain. The constructor for `MultiblockPatchHierarchy` takes an array of pointers to `PatchHierarchy`, each of which represents one block of the multiblock domain. The `PatchHierarchy` for each block must be constructed from a `GridGeometry` object that defines an index space which is independent from all other blocks. After the construction of each `PatchHierarchy`, pointers to each one can be placed in an array and passed into the `MultiblockPatchHierarchy` constructor. The order of the array can be arbitrary but is important to note, as each block will be identified by its index in the array.

In the input file, the index spaces for each block are specified in the input for each `GridGeometry`, such as the following for the domain shown in Figure 1:

```
SkeletonGridGeometry0 {
   domain_boxes = [ (0,0) , (7,8) ]
}
SkeletonGridGeometry1 {
   domain_boxes = [ (0,0) , (6,8) ]
}
SkeletonGridGeometry2 {
   domain_boxes = [ (0,0) , (6,4) ]
}
SkeletonGridGeometry3 {
   domain_boxes = [ (0,0) , (4,6) ]
}
```

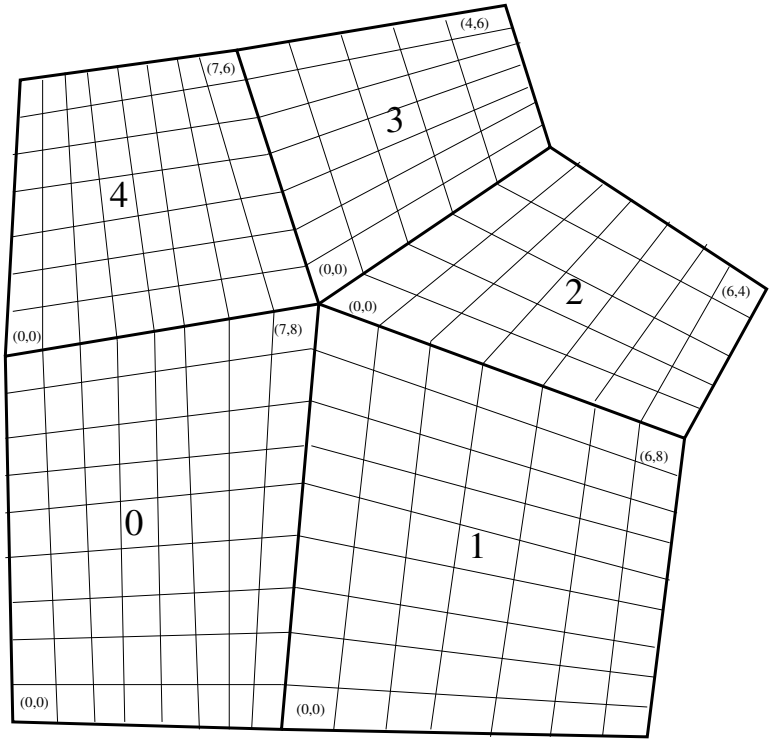

Figure 1: Example 5 block domain

```
SkeletonGridGeometry4 {
   domain_boxes = [ (0,0) , (7,6) ]
}
```

2.2 Input to describe singularities

In addition to an array of `PatchHierarchy` pointers, `MultiblockPatchHierachy` also requires input that is used to define the relationships and relative locations of the different blocks of each domain. First, the number of blocks must be specified, and then a block of input is required to describe the location of each singularity point.

```
MultiblockPatchHierarchy {
   num_blocks = 5

   Singularity0 {
      blocks = 0,1,2,3,4

      sing_box_0 = [(8,9),(8,9)]
      sing_box_1 = [(-1,9),(-1,9)]
      sing_box_2 = [(-1,-1),(-1,-1)]
      sing_box_3 = [(-1,-1),(-1,-1)]
      sing_box_4 = [(8,-1),(8,-1)]
   }
   ...
}
```

The num_blocks symbol is self-explanatory. For each singularity point, the `blocks` input specifies which blocks touch the singularity, and the `sing_box_*` inputs tell where each block abuts the singularity. In two dimensions, the required input is the single-cell box that lies immediately outside the block and touches the block only at the singularity point. In three dimensions, a singularity can occur at either a corner or an edge. If the singularity is a corner, then the input is the same as in two dimensions. If it is on an edge, the the input box must be a box that runs along the singularity edge and is width one in the other two directions. This box also must be located immediately outside the block and touch the block only along the singularity edge.

2.3 Defining Neighbor relationships

The remaining input for `MultiblockPatchHierarchy` describes the relationships between neighboring blocks. Blocks are said to be neighbors if they abut each other at any point, edge or face. Every pair of neighbors must be specified in the `MultiblockPatchHierarchy` input. Shown here are some of the `BlockNeighbors` entries needed for the Figure 1 example.

```
MultiblockPatchHierarchy {

   ...

   BlockNeighbors0 {
      block_a = 0
      block_b = 1

      rotation_b_to_a = "I_UP", "J_UP"
      point_in_a_space = 8,0
      point_in_b_space = 0,0
   }

   BlockNeighbors1 {
      block_a = 0
      block_b = 2

      rotation_b_to_a = "I_UP", "J_UP"
      point_in_a_space = 8,9
      point_in_b_space = 0,0
   }

   BlockNeighbors2 {
      block_a = 1
      block_b = 2

      rotation_b_to_a = "I_UP", "J_UP"
      point_in_a_space = 0,9
      point_in_b_space = 0,0
   }

   BlockNeighbors3 {
      block_a = 2
      block_b = 3

      rotation_b_to_a = "J_DOWN", "I_UP"
```

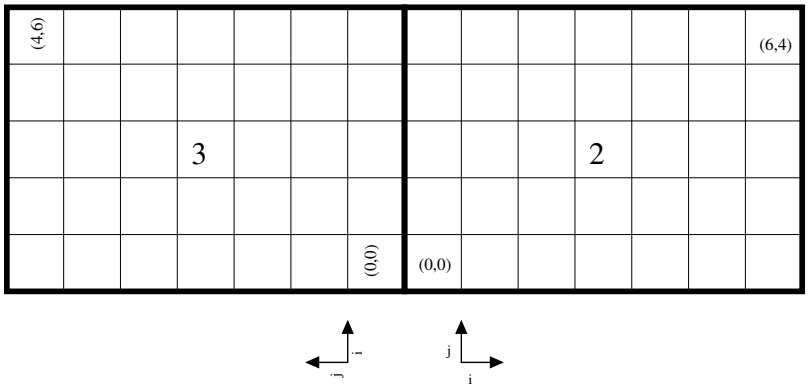

Figure 2: Index spaces of blocks 2 and 3

```
      point_in_a_space = -1,0
      point_in_b_space = 0,0
   }

   ...

}
```

In each `BlockNeighbors` entry, one block is arbitrarily chosen to be `block_a` and the other `block_b`. `rotation_b_to_a` is used to specify how block b's index space is aligned in comparison to block a's. As an example, for the `BlockNeighbors3` entry above, we consider only the two blocks in question, 2 and 3, as if the other blocks did not exist. Figure 2 shows the index spaces of the blocks 2 and 3 and the respective alignments of their i-j axes.

The positive i direction on block 2 is equivalent to the negative j direction on block 3. Thus the first entry for `rotation_b_to_a` is `"J_DOWN"`. Likewise the positive j direction on block 2 is equivalent to the positive i direction on block 3, so the second entry is `"I_UP"`. In general, the entries for `rotation_b_to_a` are determined by travelling in the positive direction for each dimension in block a, and identifying the equivalent axis and direction on block b.

For the entries `point_in_a_space` and `point_in_b_space`, we choose any cell-centered point in block a's index space and assign it to `point_in_a_space`. It does not matter whether or not the point is in the interior of block a's domain. Then we identify the index location of that same point in block b's index space, and assign it to `point_in_b_space`.

The three entries `rotation_b_to_a`, `point_in_a_space`, and `point_in_b_space` are sufficient to describe a unique neighbor relationship between two blocks.

3 Usage of `MultiblockPatchHierarchy` and other multiblock classes

`MultiblockPatchHierarchy` is one of several classes that serve as multiblock versions of classes that previously existed in *SAMRAI*. These classes provide an extension of the concepts used in regular single-block AMR problems to problems on multiblock domains.

3.1 Other multiblock classes

Just as `PatchLevel` is a representation of a single level of refinement within a `PatchHierarchy`, the class `MultiblockPatchLevel` represents a single level of refinement within a `MultiblockPatchHierarchy`. A `MultiblockPatchLevel` consists of an array of pointers to `PatchLevel` and can be retrieved using the member function `getPatchLevel()` in `MultiblockPatchHierarchy`. Depending on the state of the problem, the regions of a particular level of refinement may not exist on all of the blocks of the domain. In such a case, a `MultiblockPatchLevel` will still have a pointer to `PatchLevel` for every block of the domain, but the pointers corresponding to blocks having no refinement at that particular level will be null.

`MultiblockGriddingAlgorithm` is a multiblock extension of the original `GriddingAlgorithm` class. The multiblock version can be used almost the same as the original class, but it differs in that it uses the multiblock communication schedules (see section 4) to transfer cell-tagging information across block boundaries.

3.2 Using multiblock classes in high-level algorithm classes

Each of `MultiblockPatchHierarchy`, `MultiblockPatchLevel`, and `MultiblockGriddingAlgorithm`, along with their corresponding original non-multiblock classes, inherit from virtual base classes that allow interfaces with *SAMRAI*'s high-level classes in the algorithm package. For example, `TimeRefinementIntegrator`'s constructor takes as arguments pointers to `BasePatchHierarchy` and `BaseGriddingAlgorithm` and stores them as private data. The `TimeRefinementIntegrator` does not need to know whether its patch hierarchy pointer points to a `PatchHierarchy` or a `MultiblockPatchHierarchy`, because the time integrator consists of high-level operations such as initializing and advancing the entire hierarchy. The details of these operations are delegated to lower-level classes, such as a level integrator class that is a problem-appropriate implementation of the `TimeRefinementLevelStrategy` virtual base class.

4 Multiblock communication algorithms and schedules

Multiblock versions of the refine and coarsen algorithms and schedules have been added to *SAMRAI* in order to handle data transfer and parallel communication on multiblock hierarchies. They can be used similarly to the algorithms and schedules that exist for problems on rectangular domains.

4.1 `MultiblockRefineAlgorithm`

Like `RefineAlgorithm`, `MultiblockRefineAlgorithm` is used to register and manage a set of refinement operations and to create schedules that will control the movement of data. `MultiblockRefineAlgorithm` is constructed with a pointer to a `RefineAlgorithm` and will manage all refinement operations that have been registered with the `RefineAlgorithm`. Additional refinement operations can be added by calling the method `MultiblockRefineAlgorithm::registerRefine()`.

`MultiblockRefineAlgorithm` is used to create `MultiblockRefineSchedule` objects, which will execute the communications operations that are registered with the refine algorithm. The overloaded versions of `MultiblockRefineAlgorithm::createSchedule()` are analagous to the versions of `createSchedule()` existing in `RefineAlgorithm`. All versions of `createSchedule()` take a pointer to `MultiblockPatchStrategy` (see Subsection 4.3), which is a virtual base class that inherits from `RefinePatchStrategy`. The user must create a class that implements the physical-boundary filling interface from `RefinePatchStrategy` as well as an interface defined in `MultiblockPatchStrategy` for filling boundary data around singularities. The `MultiblockPatchStrategy` may be null, in which case no data will be filled in physical boundary ghost zones nor in ghost zones around a singularity.

4.2 MultiblockRefineSchedule

The most significant member function of `MultiblockRefineSchedule` is `fillData()`, which, like the member of the same name in `RefineSchedule`, executes the communication operations. On each block of the destination level, data is filled in the block interior, then ghost zones on block boundaries are filled from patches lying on other blocks, then ghost data around the singularity and physical boundaries is filled. `fillData()` takes an optional boolean argument `do_physical_boundary_fill`, with which the filling of physical boundary ghost data can be turned off.

4.3 MultiblockPatchStrategy

`MultiblockPatchStrategy` is a virtual base class that inherits from `RefinePatchStrategy` and adds one more pure virtual interface: `fillSingularityBoundaryConditions()`.

```
virtual void fillSingularityBoundaryConditions(
   hier::Patch<DIM>& patch,
   tbox::List<typename MultiblockRefineSchedule<DIM>::SingularityPatch>&
      singularity_patches,
   const double fill_time,
   const hier::Box<DIM>& fill_box,
   const hier::BoundaryBox<DIM>& boundary_box) = 0;
```

The `MultiblockRefineSchedule` will call this function to fill ghost data around a singularity. Figure 3 shows a `Patch` that has a corner which touches a singularity point where three blocks meet. Since every `Patch` in *SAMRAI* is still defined on a logically rectangular index space, there is data allocated for the ghost zones at the upper right corner of this `Patch`, even though those zones do not represent any physical space in the multiblock domain. The interface for `fillSingularityBoundaryConditions()` allows the user to fill these ghost zones in a problem-specific manner.

If the singularity is a point of enhanced connectivity, as in Figure 4, then there are two or more sets of data that can represent the corner ghost region of the `Patch` to be filled. The struct `SingularityPatch` within `MultiblockRefineSchedule` is used to handle this case. A list of `SingularityPatch` is passed from `MultiblockRefineSchedule` into `fillSingularityBoundaryConditions()`. The list will have one item for each block that abuts the `Patch` at the singularity (the list will be empty in cases of reduced connectivity). `SingularityPatch` is a struct that contains a `Patch` and an integer identifier of the block from which it came. The `Patch` covers the index space of the ghost region that is to be filled, and it has pointers to `PatchData` allocated for each data component that needs to be filled. The `PatchData` is filled with data from the block indicated by the integer identifier. Thus there is access to data from all neighboring blocks in the user-defined implementaion of `fillSingularityBoundaryConditions()`.

4.4 MultiblockCoarsenAlgorithm and MultiblockCoarsenSchedule

In ordinary usage, the original `CoarsenAlgorithm` and `CoarsenSchedule` classes coarsen data from a fine `Patch` onto a coarse `Patch` representing the same physical space, and no ghost data is used, thus no parallel communication is necessary. If that same behavior is all that is needed in a multiblock problem, the same original coarsen classes can be used. However, in cases where ghost data is required to be filled on the coarse level prior to the execution of the coarsen operator, `MultiblockCoarsenAlgorithm` and `MultiblockCoarsenSchedule` must be used. The usage of these classes is almost the same as that of the original coarsen classes, but if `MultiblockCoarsenAlgorithm` is constructed with the optional boolean argument `fill_coarse_data` set to true, then the instances of `MultiblockCoarsenSchedule` that will be created will use `MultiblockRefineSchedule` to pre-fill the coarse levels. Thus `MultiblockCoarsenAlgorithm`'s

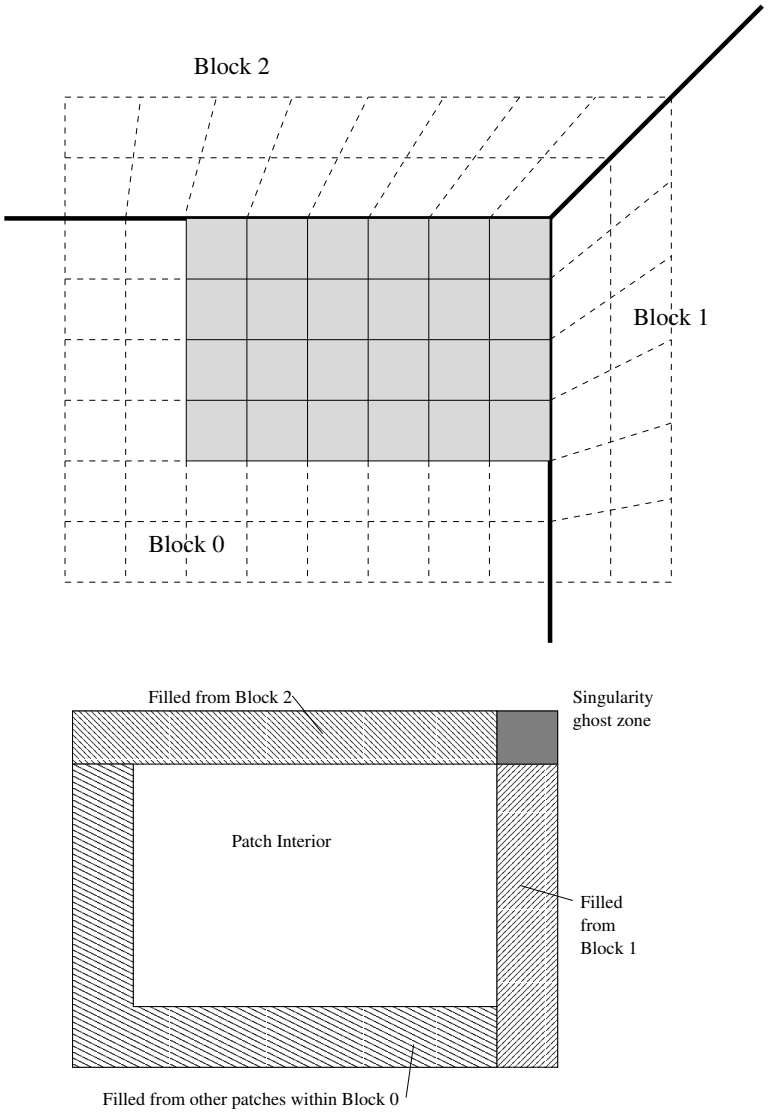

Figure 3: A `Patch` that needs ghost zones filled at a singularity. The first picture shows a `Patch` where it is located in a multiblock domain, and the second shows that the `Patch` is still defined in terms of a rectangular index space. The ghost zones on the top and the right can be filled from the neighboring blocks, but the zones in the upper right do not have corresponding cells on the other blocks.

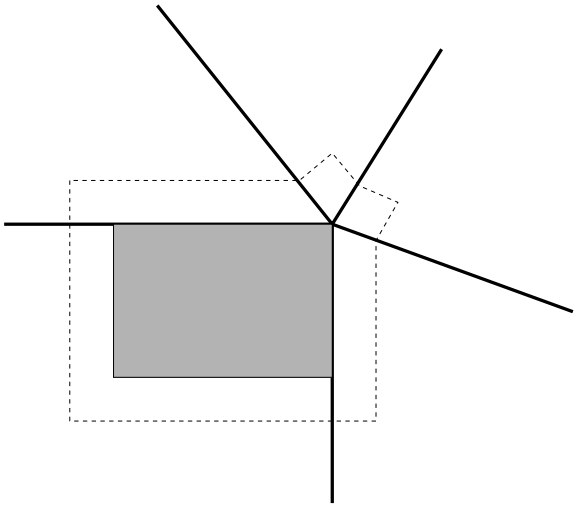

Figure 4: A `Patch` that needs ghost zones filled at a singularity with enhanced connectivity. There are two blocks that can be used to fill ghost data in the upper right corner.

`coarsenSchedule()` routine takes pointers to both `CoarsenPatchStrategy` and `MultiblockPatchStrategy`. The `CoarsenPatchStrategy` pointer is used for user-defined functions related to the the coarsen operation, just as with an ordinary `CoarsenSchedule`, while the `MultiblockPatchStrategy` pointer is needed to allow for the user-defined functions for refinement and boundary-filling that will be required when a `MultiblockRefineSchedule` is used by `MultiblockCoarsenSchedule`.