LLNL-PROC-610672

Scalable Mesh Management for Patch-based AMR

B. T. N. Gunney

January 16, 2013

Nuclear Explosive Code Development Conference
Livermore, CA, United States
October 22, 2012 through October 26, 2012

(U) Scalable Mesh Management for Patch-based AMR

[1]Brian T. N. Gunney
[1]Lawrence Livermore National Laboratory, Livermore, CA

Abstract

Patch-based adaptive meshes traditionally use sequential mesh management, where every process duplicates and works on the global mesh description. This approach greatly simplifies mesh management and is efficient at low process counts, but it scales poorly, becoming impractical at around 1K processes. The SAMRAI framework has adopted a distributed mesh management approach, where each process sees and operates on a small part of the mesh. The new approach uses information typically discarded by the traditional approach, avoiding unscalable global searches for proximity and the all-to-all communication required to acquire global mesh descriptions. We explain the core concepts, describe our implementation and show good scaling results on up to 256K processes.

SAMR and its traditional parallelization

Patch-based or structured adaptive mesh refinement (SAMR) [1] is an important technology used for complex multiscale multiphysics applications. Figure 1 illustrates a SAMR mesh and its associated hierarchy, consisting of a sequence of patch levels with increasing resolution. Each level is composed of structured grids called patches. We refer to **mesh management** as the generation, alteration, distribution, storage and processing of the mesh structure. Implementing SAMR is complex. Efforts to factor out this complexity have resulted in the development of framework libraries such as SAMRAI [2, 3], Chombo [4] and BoxLib [5]. Our work has been implemented in SAMRAI, but we believe the concepts are applicable to all similar SAMR implementations.

On distributed memory parallel computers, SAMRAI distributes the mesh one level at a time with each patch assigned to an MPI process, following the LPARX parallel distribution model [6]. Each process stores only the patches it owns and simulation data in those patches. However, for mesh management, the traditional approach stores a copy of all boxes in the mesh. This data facilitates the use of sequential algorithms for remeshing and detecting which patches are close to which. The cost for acquiring, storing and processing these boxes on each process is at least $O(P)$, where P is the number of processes. Although this cost grows linearly, the approach is justified originally because box data is so small compared with simulation data. However, for typical dynamically adaptive applications in continuum mechanics, we have observed that this cost overwhelms the cost of real computations around the range of $P = 1K$. This becomes a serious problem on current and upcoming parallel computers where P is several orders of magnitude greater than 1K.

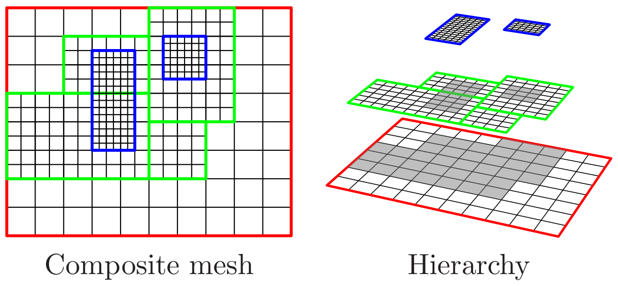


Figure 1: Patch-based Structed AMR mesh

Several approaches have aimed to address this cost. Stout, et. al. [7], and Burstedde, et. al. [8] used fixed-size patches with a 2-to-1 refinement ratio, managed by a parallel oct-tree. The oct-tree facilitated proximity detection. Luitjens, et. al. constrained the clustering algorithm to produce boxes aligned with predetermined uniform tiles [9] and used a parallel space-filling curve to distribute the fixed-size boxes [10]. However, the need to acquire and store a list of global patches remained when the individual parts were assembled into an application [11].

Our work focuses on the patch-based approach of Berger and Colella, which has the flexibility of variable-sized patches and variable refinement ratios. This flexibility helps to reduce the memory overhead for storing ghost data and facilitates ALE-AMR schemes requiring odd refinement ratios [12]. This paper describes the problem of parallel mesh management and a distributed mesh management approach that supports SAMR without requiring each process to have a copy of the global mesh. This approach is currently implemented in SAMRAI version 3.

Global metadata dilemma

We define **metadata** loosely as the data describing the mesh structure. It consists of boxes, box owner ranks when the boxes are distributed and other data computed from the boxes, such as nearby neighbors. While physics simulations operate on simulation data, mesh management operates on mesh metadata. The $O(P)$ data problem comes from from metadata. To ensure every process has some work to do, there must be at least P boxes. Any global description of the mesh will have size $O(P)$. To avoid the problems of $O(P)$ data, we've adopted a mesh management approach that distributes that data over multiple processes, in the same manner as we distribute patches. SAMRAI's **BoxLevel** is a data structure for storing a distributed set of boxes. Each box has an owner and each process stores only its local boxes. The absence of globalized boxes in the

BoxLevel creates multiple difficulties for remeshing.

To go from global metadata to distributed metadata, each process simply deletes the portions not local to itself. Doing this disables dynamic mesh adaptation, however, because key steps in adaptation rely on the global data. Consider mesh adaptation steps as originally described in [1] and adopted by SAMRAI. Mesh adaptation changes the mesh one level at a time. We generate a new level to replace an old one, transfer data to the new level and discard the old. Each time we do this, we perform four broad steps: clustering, box manipulation, partitioning and proximity detection.

The clustering step generates an initial set of boxes around cells that are tagged for refinement. SAMRAI uses a parallel clustering algorithm based on the Berger-Rigoutsos algorithm [13, 14]. It produces a set of boxes covering the regions the user wants refined, but the boxes are not necessarily configured to make a good, or even usable, patch level. It may contain cells that lie outside a non-convex domain, violate the proper nesting condition in [1] or have unbalanced work distribution. Working these boxes into a configuration that makes a good patch level without globalizing data requires a new set of algorithms.

The box manipulation step makes small changes to the extent of some boxes so that they can form a workable patch level. SAMRAI cuts off cells that lie outside the next coarser level and cells that violate certain nesting conditions. It extend boxes that are less than the maximum ghost data width from the domain boundary so that they touch the boundary. These steps require comparing two global sets of boxes for intersections. Upon examination, one can see that only comparisons of nearby neighbors affect the results. Most checks can be safely omitted. Nevertheless, in the absence of global metadata, finding nearby boxes without knowing if there are any–or which process owns them–is difficult.

Partitioning is the assignment of boxes to processes in a load-balanced manner, often breaking up some boxes to assign a part of them. Modern partitioning libraries such as Zoltan and PTScotch can partition without globalizing data, but to our knowledge, there is currently no partitioner that does this and also supports the box-breaking needed for effective SAMR partitioning. We can overcome this limitation by pre-breaking our boxes into smaller boxes before using a partitioning library, but this can produce an unnecessarily large numbers of small boxes just to ensure the result is well balanced.

Proximity detection finds the boxes' nearby neighbors for transferring simulation data. Applications may also need proximity information. As in the box manipulation step, this is difficult without searching global data.

Getting box proximity data is a major source of difficulty. We need this data during and after level generation. The new approach provides the data without resorting to globalization. The clustering and partitioning algorithms have been described in [14, 16]. For these steps, we describe the requirement that the new approach imposes on these algorithms to facilitate proximity detection without globalization.

Key Requirements

Our approach is built on two key requirements. The first is that boxes appear due to conditions in their locality, not at random or due exclusively to conditions far away. For example, a box generated by the clustering algorithm appears due to tagged cells that the box would cover. Box breaking in the partitioning step generates multiple smaller boxes from an existing large box. Every new box generated is based on conditions in existing boxes in the neighborhood of the new box. It is the processes owning these existing boxes that define and decide to create the box. It is therefore possible for those processes to find enough existing local boxes close to the new box. The owner of the new box, chosen from the processes participating in the box creation, can receive the new box's neighbor data sent from other participating processes. In this paper, we say that the new boxes and the ones that gave rise to them share a **derivative** relationship.

Whereas the traditional approach discards derivative relationships and does a global search for neighbors later, we save these relationships in a **Connector**. Borrowing from basic graph terminology, we say that a Connector is an adjacency list, storing directed edges in a format convenient for SAMRAI operations. Each Connector stores edges incident from a "**base**" BoxLevel to a "**head**" BoxLevel. For each base box, it can hold a set of related head boxes. We denote a Connector from base P to head Q as $\{P \rightarrow Q\}$, an expression that can be thought of as a set of edges from boxes in P to boxes in Q. The Connector pointing the opposite direction is written as $\{Q \rightarrow P\}$, and the pair is $\{P \Leftrightarrow Q\}$. Individual edges in the Connectors are written as $(p \rightarrow q)$, $(q \rightarrow p)$ and $(p \Leftrightarrow q)$, where p and q are boxes in P and Q respectively. Connector data is distributed like BoxLevels, and each process stores only relationships for its local base boxes. Neighbor sets may include remote boxes.

The new approach uses Connectors widely. Most Connectors hold proximity data. A **proximity Connector** $\{P \xrightarrow{\omega} Q\}$ has the following properties:

- The non-negative **Connector width** ω is a parameter of proximity or closeness. If a base box, grown in each direction by ω, overlaps a head box, they are close by definition. A width of zero means that boxes must overlap to be considered close. To avoid clutter in the writing, we may omit the width when its value is irrelevant or clear from context.
- All neighbor relationships must represent sufficient proximity.
- A proximity Connector is **complete** if all proximity relationships between its base and head are represented. SAMRAI provides complete proxmimity Connectors for applications to use. However, at certain points in mesh management operations, we can deduce that the missing relationships do not affect the outcome. At these points, we choose to work with incomplete ones to save the cost of ensuring completeness.

The second requirements is to efficiently compute proximity data for BoxLevels that do not share derivative relationships. For example, when updating a level, we use proximity data to transfer simulation data from the old level to the new level. The old level does not play a part in generating the new level, so proximity data between them is not available through derivative relationships. Also, the newly generated BoxLevel would inevitably change from the initial form given by the clustering algorithm, so we would have to recompute the initial Connectors, or at least update them. To achieve our goal, proximity data must be computed and updated without globalizing.

All approaches to SAMR mesh generation and adaptation that we are aware of satisfy the first requirement. The second requires new algorithms described in the next two sections.

Bridge algorithm

The bridge algorithm computes proximity data between two unrelated BoxLevels, A and B, using their relationships to a third BoxLevel, C. Figure 2a shows the inputs and outputs of the bridge operation. Algorithm 1 is a basic bridge implementation. Each process examines the A and B boxes that it knows about through $\{C \rightarrow A\}$ and $\{C \rightarrow B\}$. It finds A-B boxes that are within the Connector width of each other and sends that information to the owners of those boxes. The processes use $\{A \rightarrow C\}$ and $\{B \rightarrow C\}$ to determine what processes they have to receive from. All algorithms in this paper work on local data except where we explicitly specify communication.

Line 8 in Algorithm 1 introduces a new shorthand. A superscript applied to a box indicates a growing operation. Expression a^{Γ} means box a grown by Γ on each side.

The Connector width Γ, at line 4, depends on nesting parameters ν_a and ν_b, the amount that C must grow to nest A and B, respectively. These parameters can be computed from parameters constraining BoxLevels on the hierarchy, such as nesting requirements, and operations that change the amount of nesting between two BoxLevels, such as box growing. The exact formula for these parameters depend on the specific sets of boxes in specific SAMR implementations.

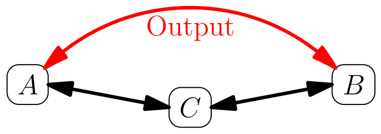
The output width value

$$\Gamma = \max(\Gamma_a - \nu_b, \Gamma_b - \nu_a) \tag{1}$$

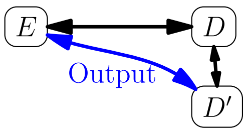
is the maximum width for which we can guarantee $\{A \overset{\Gamma}{\Longleftrightarrow} B\}$ is complete, if $\{A \overset{\Gamma_a}{\Longleftrightarrow} C\}$ and $\{C \overset{\Gamma_b}{\Longleftrightarrow} B\}$ are complete. Appendix A provides a proof of this.

In practice, either ν_a or ν_b has the value of zero for many specific bridge operations in the code. As long as one is known, the other can be assumed to be arbitrarily large without affecting Γ. If $\{A \Leftrightarrow C\}$ or $\{B \Leftrightarrow C\}$ is not available, it is always possible in SAMRAI to find a chain of proximity Connectors to which we can apply a sequence of bridge operations to eventually compute $\{A \Leftrightarrow B\}$.

Bridging is widely used in mesh adaptation, so its performance is multiplied many times with each regrid. Although Algorithm 1 is scalable for reasonably distributed metadata, it is important to have a well-optimized implementation. The alternate implementation, Algorithm 2, uses a number of tricks to run faster overall than Algorithm 1.



a) Bridge computes red proximity Connector using data in black proximity Connectors.



b) Modify computes blue proximity Connector given original $\{E \Leftrightarrow D\}$ and the mapping from D to D'.

Figure 2: Inputs and outputs of bridge and modify operations.

Algorithm 1 BasicBridge: Given proximity Connectors $\{A \overset{\Gamma_a}{\Longleftrightarrow} C\}$ and $\{C \overset{\Gamma_b}{\Longleftrightarrow} B\}$, compute proximity Connectors $\{A \overset{\Gamma}{\Longleftrightarrow} B\}$.

```
1: Post non-blocking receive for each remote neighbor owner in $\{A \rightarrow C\}$ and $\{B \rightarrow C\}$.
2: $\nu_a \leftarrow$ amount $C$ must grow to nest $A$
3: $\nu_b \leftarrow$ amount $C$ must grow to nest $B$
4: $\Gamma \leftarrow \max(\Gamma_a - \nu_b, \Gamma_b - \nu_a)$
5: for all local box $c$ in $C$ do
6:   for all neighbor $a$ of $c$ from $\{C \rightarrow A\}$ do
7:     for all neighbor $b$ of $c$ from $\{C \rightarrow B\}$ do
8:       if $a^\Gamma$ intersects $b$ then
9:         if $a$ is local then
10:          Store edge $(a \rightarrow b)$ in $\{A \rightarrow B\}$
11:        else
12:          Pack edges $(a \rightarrow b)$ into message for owners of $a$
13:        end if
14:        if $b$ is local then
15:          Store edge $(b \rightarrow a)$ in $\{B \rightarrow A\}$
16:        else
17:          Pack edges $(b \rightarrow a)$ into message for owners of $b$
18:        end if
19:      end if
20:    end for
21:  end for
22: end for
23: Post non-blocking sends of messages.
24: Receive and unpack messages into $\{A \Leftrightarrow B\}$
```

A major difference between Algorithm 2 and Algorithm 1 is that locally visible neighbors from BoxLevels A and B are placed into a tree for fast box intersection searches, changing the triple nested loop structure in Algorithm 1. Placing locally visible neighbors into set containers S_a and S_b eliminates duplicates. This in turn eliminates duplicate proximity checks on the same pair of boxes. Lumping all visible neighbors into the same container this way may add intersection checks that Algorithm 1 did not have to do. However, many of these checks can be quickly eliminated by the recursive binary box tree search algorithm [15]. Variables T_a and T_b are the tree versions of S_a and S_b. Function RecursiveBinaryBoxTreeSearch, called at lines 10 and 18, returns boxes in the tree (first argument) that intersect the box in the second argument.

Communication time can be overlapped with neighbor searches. The sets S_a and S_b order boxes by owner first, grouping all boxes with the same owner together. With a little logic to track iterator positions (omitted from the pseudocode for clarity), we can post the send to a neighbor owner before continuing to other owners.

Another optimization not shown in the pseudocode is that the looping does not strictly go in increasing numerical order of owner ranks. It starts with the first remote rank higher than the local rank, loops to the highest rank, continues at the lowest and ends with the last remote rank lower than the local rank. We do this to alleviate hot spots in communication network use that may appear around process 0 at the beginning and around the highest-rank process at the end.

Modify Algorithm

When we already have proximity Connectors between two BoxLevels but would like to change one of the BoxLevels, we need to modify the existing Connectors. Figure 2b shows the schematic for the modify operation wherein BoxLevel D would be replaced by D'. The Connectors between D and an existing BoxLevel E are to be modified. The modify operation is similar to bridging. In fact, we can bridge for Connectors $\{E \Leftrightarrow D'\}$ using D as the center BoxLevel. However, in practice the change from D to D' is often small. For example, in proper nesting enforcement, D is often already properly nested, so it doesn't change at all. We can write a specialized modify algorithm that uses less processing time, communication time and memory.

We define the following criteria for **mapping Connectors** $\{D \overset{\omega}{\Longleftrightarrow} D'\}$:

- $\{D \xrightarrow{\omega} D'\}$ has a neighbor set for $d \in D$ if and only if d is changed. Set the neighbors of d to the boxes that it becomes in D'. If d will be removed without being replaced by anything, it must have an empty neighbor set. This is in contrast to proximity Connectors, where the absence of a neighbor set and the presence of an empty set mean the same thing.
- $\{D' \xrightarrow{\omega} D\}$ has a neighbor set for $d' \in D'$ if and only if d' is a new box. For each new box d', set its neighbor to the box in D that it originated from.
- The non-negative Connector width ω is the amount that any box $d \in D$ would have to grow to nest its neighbors. A mapping Connector width represents how much the change may extend or move a box outside its old boundary.

Algorithm 2 BRIDGE: Given proximity Connectors $\{A \overset{\Gamma_a}{\Longleftrightarrow} C\}$ and $\{C \overset{\Gamma_b}{\Longleftrightarrow} B\}$, compute proximity Connectors $\{A \overset{\Gamma}{\Longleftrightarrow} B\}$.

```
1: Post non-blocking receive for each remote neighbor owner in $\{A \to C\}$ and $\{B \to C\}$.
2: $\nu_a \leftarrow$ amount $C$ must grow to nest $A$
3: $\nu_b \leftarrow$ amount $C$ must grow to nest $B$
4: $\Gamma \leftarrow \max(\Gamma_a - \nu_b, \Gamma_b - \nu_a)$
5: Insert neighbors from $\{C \to A\}$ into ordered set $S_a$
6: Insert neighbors from $\{C \to B\}$ into ordered set $S_b$
7: Build RecursiveBinaryBoxTree $T_a$ from $S_a$
8: Build RecursiveBinaryBoxTree $T_b$ from $S_b$
9: for all $a$ in $S_a$ do
10:    $\beta \leftarrow$ RECURSIVEBINARYBOXTREESEARCH($T_b, a^\Gamma$)
11:    if $a$ is local then
12:       Add edge $(a \to \beta)$ to $\{A \to B\}$
13:    else
14:       Add edge $(a \to \beta)$ to message to $a$'s owner
15:    end if
16: end for
17: for all $b$ in $S_b$ do
18:    $\alpha \leftarrow$ RECURSIVEBINARYBOXTREESEARCH($T_a, b^\Gamma$)
19:    if $b$ is local then
20:       Add edge $(b \to \alpha)$ to $\{A \to B\}$
21:    else
22:       Add edge $(b \to \alpha)$ to message to $b$'s owner
23:    end if
24: end for
25: Post non-blocking sends of messages.
26: Complete receives and unpack messages into $\{A \Leftrightarrow B\}$.
```

Algorithm 3 is the specially optimized algorithm for updating proximity Connectors when a BoxLevel changes.

Line 2 computes the output width by shrinking the input width by the mapping width. This is the maximum value for which we can guarantee completeness of $\{E \Leftrightarrow D'\}$ if $\{E \Leftrightarrow D\}$ is complete. If the mapping allows a box to go outside its old boundary, the box can get that much closer to the limit of how far out $\{D \rightarrow E\}$ can see and find neighbors. Thus, the distance over which we can guarantee seeing neighbors, relative to the boundaries of the new boxes, is smaller. In most cases, mapping Connectors have zero width, so multiple use of modify operations do not result in ever-shrinking widths for proximity Connectors.

Lines 14 and 15 introduce a new data structure holding the same information as a Connector but rearranged. The symbol $\overline{\{E \rightarrow D\}}$ represents a data structure storing, for each *visible neighbor* in E, the set of *local boxes* in D that have relationships to it. The symbol $\overline{\{D' \rightarrow D\}}$ is similar. Because the number of changed boxes in D is usually small, these data structures are also small. This re-arrangement of data allows us to write the loops through neighbors in E and D' that go through them grouped by their owners. The optimization used in the bridge algorithm for overlapping search and communication and for alleviating communication hot spots can thus be used in this algorithm.

Lines 18 and 28 each contains an implicit linear search loop. In practice, the loop length is usually short. In situations where a box may be mapped into a large number of boxes, these implicit loops can be slow. A common example is the mapping Connector for partitioning of the coarsest level, where the pre-partitioning BoxLevel often has everything on one process. (We note that this operation usually occurs only during the initial mesh generation, not during repeated adaptations.) In these cases, other measures must be taken to address the reason why such a drastic change is happening. Those measures logically belong in the algorithms that created such drastic changes and are therefore outside the scope of this paper. We justify expecting the creator of the Connectors to address the problem because when the Connectors are so big, their creation is also slow. A fast modify operation won't fix the cause of the performance problem, or the effects of storing such a large data structure. In addition, the creator of the Connectors possesses additional knowledge that could help fix the problem. The modify algorithm is generic and does not have that special knowledge.

The modify algorithm does not necessarily yield the same result that bridging for $\{E \Leftrightarrow D\}$ using Algorithm 2 would. Bridging Algorithm 2 tries to detect as many proximity relationships as possible, making up for missing edges where it can. The modify operation limits new relationships detected to those that follow from the $\{D \rightarrow D'\}$ mapping. This is a practical design decision based on the needs of low-level SAMRAI metadata operations. If the same result as bridging is desired, the modify algorithm can collect visible sets of neighbors in E and D' and search through them the way Algorithm 2 did with S_a and S_b.

Metadata Algorithm usage in Mesh Management

Algorithm 3 MODIFY: Given proximity Connectors $\{E \overset{\Gamma}{\Longleftrightarrow} D\}$ and mapping Connector $\{D \overset{\gamma}{\Longleftrightarrow} D'\}$, compute proximity Connectors $\{E \overset{\Gamma'}{\Longleftrightarrow} D'\}$.

```
1: Post non-blocking receive for each remote neighbor owner in $\{E \xrightarrow{\Gamma} D\}$ and $\{D' \xrightarrow{\gamma} D\}$.
2: $\Gamma' \leftarrow \Gamma - \gamma$
3: Copy neighbor data from $\{E \overset{\Gamma}{\Longleftrightarrow} D\}$ to $\{E \overset{\Gamma'}{\Longleftrightarrow} D'\}$
4: for all $d$ with neighbor sets in $\{D \to D'\}$ do
5:   Remove $d$'s neighbor set from $\{D' \to E\}$.
6:   for all neighbors $e$ of $d$ in $\{D' \to E\}$ do
7:     if $e$ is local then
8:       Remove $(e \to d)$ from $\{E \to D'\}$
9:     else
10:      Note removal of $(e \to d)$ in message for owner of $e$
11:    end if
12:  end for
13: end for
14: $\overline{\{E \to D\}} \leftarrow$ REARRANGECONNECTOR($\{D \to E\}$)
15: $\overline{\{D' \to D\}} \leftarrow$ REARRANGECONNECTOR($\{D \to D'\}$)
16: for all $e$ in $\overline{\{E \to D\}}$ do
17:   for all $d$ corresponding to $e$ in $\overline{\{E \to D\}}$ do
18:     $\delta' \leftarrow$ all $d'$ neighbors of $d$ intersecting $e^{\Gamma'}$
19:     if $e$ is local then
20:       Add $(e \to \delta')$ to $\{E \to D'\}$
21:     else
22:       Add $(e \to \delta')$ to message for $e$'s owner.
23:     end if
24:   end for
25: end for
26: for all $d'$ in $\overline{\{D' \to D\}}$ do
27:   for all $d$ corresponding to $d'$ in $\overline{\{D' \to D\}}$ do
28:     $\epsilon \leftarrow$ all $e$ neighbors of $d$ intersecting $d'^{\Gamma'}$
29:     if $d'$ is local then
30:       Add $(d' \to \epsilon)$ to $\{D' \to E\}$
31:     else
32:       Add $(d' \to \epsilon)$ to message for owner of $d'$.
33:     end if
34:   end for
35: end for
36: Post non-blocking sends of messages to owners of visible neighbors in $E$ and $D'$.
37: Complete receives, unpack messages into $\{E \Leftrightarrow D'\}$ and remove data for boxes that went away.
```

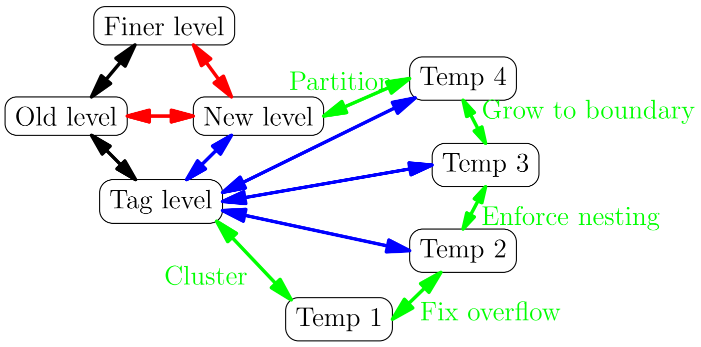


Figure 3: BoxLevels and Connectors during the generation of a new level to replace an old one. Connectors are colored by how they are computed. Black Connectors are pre-existing on the hierarchy. Green Connectors are derivative relationships. Blue Connectors come from modifying another Connector. Red Connectors are the results of bridging.

Figure 3 illustrates where bridging and modifying are used in regridding. Generation of the new level follows steps similar to those outlined in the original Berger-Colella paper [1]. Figure 3 denotes the level we wish to update as the "Old level." We want to create its replacement, labeled "New level." First, we tag cells on the next coarser level, labeled the "Tag level." The clustering step generates "Temp 1" and the Connector between "Temp 1" and "Tag level." Subsequent steps (fixing the overflow, enforcing proper nesting, growing some boxes to the domain boundary and finally partitioning) generate a chain of connected BoxLevels. Fixing overflow means removing cells that lie outside the Tag level, a nesting requirement that does not automatically follow from proper nesting. This is a new step, added to ensure nesting conditions for the bridge algorithm. Note that this step only removes extra cells added by the clustering algorithm, not cells that the user wants. We compute Connectors from "Tag level" to each BoxLevel in the chain using the modify operation. Bridging finds proximity between "New level" and other existing levels in the hierarchy.

In this paper, we have not discussed the details of SAMRAI's new clustering or partitioning algorithms. SAMRAI provides internal clustering and partitioning implementations and also supports external implementations at run time through virtual interfaces. As Figure 3 suggests, whatever clustering or partitioning algorithm used is expected to save derivative data in Connectors. The clustering implementation provides the initial proximity Connectors. The partitioning implementation has to generate mapping Connectors for use in the modify algorithm. If the implementations do not provide this data, SAMRAI can compute it automatically. Without knowledge from the implementations, however, the computation requires the unscalable step of globalizing metadata.

Outside of remeshing, mesh management occurs in generating communication **schedules** for data transfer. SAMRAI's communication schedules are data structures holding hierarchy-specific information for transferring simulation data within and between patch levels. Their creation requires proximity data, obviously. In past versions of SAMRAI, the slowest schedules to generate were schedules for recursive refinement, where we interpolate data from successively coarser levels to fill a fine level. Generating recursive refine schedules involves an arrangement of temporary levels and Connectors similar to remeshing. Bridging and modifying operations are intensely used.

Performance of new approach

To demonstrate performance of the new approach, we designed a benchmark simulation that emphasized remeshing cost. We chose a simple linear advection problem to have an inexpensive reference point for comparison. We remeshed each level after every four time steps on that level. Practical problems with more complex physics or less frequent remeshing will have higher computation-to-regrid cost ratios.

The benchmark used a 32x8x8 coarse grid domain covering a physical domain box from (0,0,0) to (8,2,2). We propagated a disturbance the shape of a sinusoidal wavy wall having a period of 4 in the j- and k-directions, oriented perpendicular to the i-direction. The amplitude of the sine waves was 0.5. We increased the problem size by tiling the domain, doubling in the j-direction first, then the k-direction, then the i-direction. Each time we doubled the i-direction, we doubled the number of wavy walls. Each wall was located 8 physical units from the next, in the i-direction. The mesh had three levels and used a refinement ratio of 2. We set SAMRAI's Berger-Rigoutsos efficiency tolerance to 0.8 and combine efficiency to 0.8. We used tagged cells where the wavy wall should be instead of where it is in the simulation. This gave us better control over the mesh size, which would otherwise be affected by numerical smearing of the solution. Level 0 had 2048 cells per process. Tags around the wavy wall were buffered to achieve about 4100-4600 cells per process on level 1 and about 20,000-22,000 cells per process on level 2. Figure 4 shows a characteristic box configuration. The advection velocity was (2,0.01,0.01). We run the problem for 25 coarse-mesh steps with a Courant number of 0.5, using time-refinement integration [1]. There were 2.5M cell updates per process, on average, over the 25 coarse time steps.

We ran the benchmark on two IBM BlueGene machines at Lawrence Livermore National Lab. The first plot in Figure 5 shows wall clock time, normalized by number of cell updates, for the weak-scaled benchmark on UBGL, an IBM BG/L computer. The benchmark problem size was increased by domain tiling to maintain nearly constant average work load per process. The problem scaled well past the 1K range, where the old approach failed to provide any further speed-up. The new approach replaced a few all-gather operations that were used to globalize metadata with many peer-to-peer operations. While all-gather operations are known to inherently scale like $O(P)$, the second plot in Figure 5 shows that cost of the bridge and modify operations scale well after an initial transient rise. The transient was probably due to the increasing number of processes that any process had to communicate with. In the beginning, this number was limited by P.

Figure 6 shows results of a similar benchmark run on Sequoia, an IBM BG/Q computer. Almost everything scaled reasonably to 256K processes. At around 64K processes, regridding began to

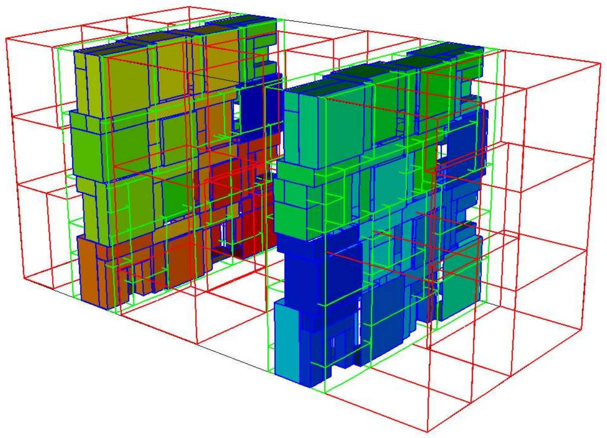


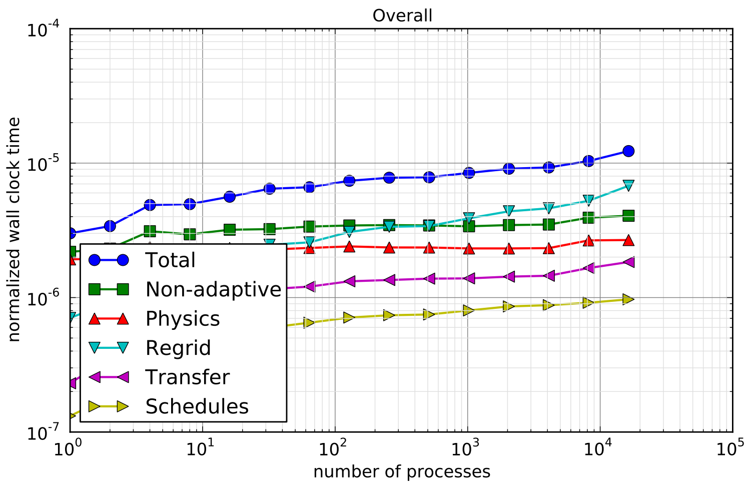
Figure 4: Wavy wall mesh for 32-process run. Levels 0 and 1 are drawn in wire-frame. Level 2 boxes are colored by owner process. Some missing boxes are caused by visualization software.

slow significantly. The middle plot shows the total cost of regridding and the cost of the individual constituent steps that we described earlier. It shows that the degradation is caused largely by the clustering step. The third plot shows that the bridge and modify algorithm performance continue to scale well. Clearly, more work is need to improve the clustering algorithm.

Summary and conclusion

Traditional parallel patch-based SAMR uses sequential mesh adaptation algorithms operating on global metadata. This simple approach works well for small numbers of processes but does not scale well on today's parallel supercomputers. Distributing the metadata as we distribute simulation data solves the data problem but complicates mesh adaptation. It makes it difficult to search for boxes that are close to each other. In addition, every major step in mesh adaptation requires global metadata or makes a change that requires a later step to search the global metadata.

We have developed a distributed mesh management approach that does not require global metadata. The new approach makes use of box proximity and box transformation data normally discarded by the traditional approach. We argue that because boxes in the new level are generated locally, it is possible to compute proximity data for new boxes without having the global picture. We maintain and update the distributed metadata, including proximity data, as we form the new patch level. When the new level is formed, we have all the proximity data to



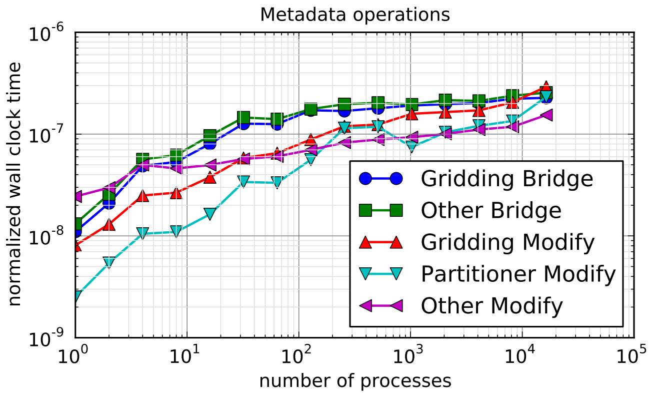


Figure 5: Performance on IBM BG/L

perform the simulation and further mesh adaptations without resorting to globalizing metadata.

We support the new approach with two new metadata algorithms. The bridge algorithm computes proximity data between two unrelated sets of boxes. The modify algorithm updates proximity data when some boxes are changed.

We demonstrated the scalability of the new approach with a benchmark designed to incur high remeshing cost, run on two BlueGene computers at Lawrence Livermore National Lab. The simulation did not exhibit the linear scaling problem associated with the traditional method. It scaled well for up to 256K processes, except for the clustering step which scaled well up to 64K processes. In particular, the metadata operations were fast compared with the simulation time and scaled well after an initial rise. More work is needed to improve the scaling performance of the clustering algorithm.

Appendix A: Bridge Completeness Proof

This appendix discusses the conditions under which the bridge algorithm guarantees outputs that are complete proximity Connectors and provides a proof of that completeness.

It is easy to see how the algorithm can miss proximity relationships. Consider BoxLevels $A = \{a\}$, $B = \{b\}$ and $C = \{c\}$. Each BoxLevel has exactly one box, each owned by a different process. Suppose boxes a and b overlap each other, but they lie so far from box c that $\{A \Leftrightarrow C\}$ and $\{B \Leftrightarrow C\}$ contain no relationship. No process sees both a and b, and no process can detect their proximity.

For bridging to detect all proximity relationships, there must be constraints on how far A and B can be from C. If A and B nest inside C, then the computed bridge $\{A \overset{\Gamma}{\Longleftrightarrow} B\}$ is a complete proximity Connector for some Γ. However, if neither A nor B nests inside C, $\{A \overset{\Gamma}{\Longleftrightarrow} B\}$ is not guaranteed to be complete. In such case, two questions arise. What conditions, if any will guarantee completeness of the bridge? What is the maximum width Γ for which we can guarantee completeness?

We answer these questions with the following Bridge Theorem:

- Let A, C and B be BoxLevels.
- Let $\{C \overset{\Gamma_a}{\Longleftrightarrow} A\}$ and $\{C \overset{\Gamma_b}{\Longleftrightarrow} B\}$ be complete proximity Connectors for the widths Γ_a and Γ_b, respectively.
- Let A nest in C grown by ν_a,
$$A \subset C^{\nu_a} \tag{2}$$
for some ν_a, meaning A is not arbitrarily far outside of C.
- Let
$$\Gamma_a \geq \nu_a \tag{3}$$
$$\Gamma_b \geq \nu_a \tag{4}$$
meaning both $\{C \xrightarrow{\Gamma_a} A\}$ and $\{C \xrightarrow{\Gamma_b} B\}$ see at least as far as A is outside of C.

Then $\{A \overset{\Gamma}{\Longleftrightarrow} B\}$ obtained by Algorithm 1 or 2 are complete for
$$\Gamma = \Gamma_b - \nu_a \tag{5}$$

Proof:

Assume that all boxes are single-cell. If the bridge algorithm works for single-cell boxes, then it will work for multi-cell boxes also. Finding neighbors for all single-cell boxes in a multi-cell box means that we can find neighbors for the multi-cell box containing those cells.

Let a and b be boxes in A and B, respectively. By definition of completeness, the existence of edge $(a \Leftrightarrow b) \in \{A \overset{\Gamma}{\Longleftrightarrow} B\}$ means that b intersects a^{Γ}. Because b is a single cell, it is also completely inside a^{Γ},

$$b \subset a^{\Gamma} \tag{6}$$

Algorithms 1 and 2 can find this edge if there is a $c \in C$ that sees both a and b. Formally,

$$(c \rightarrow a) \in \{C \xrightarrow{\Gamma_a} A\} \tag{7}$$

and

$$(c \rightarrow b) \in \{C \xrightarrow{\Gamma_b} B\} \tag{8}$$

From (2), there is at least one box $c \in C$ satisfying

$$a \subset c^{\nu_a} \tag{9}$$

Figure 7 shows this box and the grown version c^{ν_a}. Box a is drawn at the farthest distance it can be from box c and still satisfy (9).

From (3) and (9), we have

$$a \subset c^{\nu_a} \subset c^{\Gamma_a} \tag{10}$$

or $a \subset c^{\Gamma_a}$, which satisfies the requirement (7).

We now show that (8) is also satisfied. Because a and c are single-cell boxes, (9) gives

$$c \subset a^{\nu_a} \tag{11}$$

which can easily be seen in Figure 7. Since b is inside a^{Γ} (6), and c is inside a^{ν_a} (11), b and c can be no more than $\Gamma + \nu_a$ cells apart, or

$$b \subset c^{\Gamma+\nu_a} \tag{12}$$

Substituting (5), we have

$$b \subset c^{\Gamma_b} \tag{13}$$

which satisfies (8).

Having satisfied (7) and (8), we conclude that edges $(a \Leftrightarrow b)$ can be found for $\Gamma = \Gamma_b - \nu_a$.

In Figure 7, box b' is outside the range that c can see, so we cannot guarantee any process will check its proximity with a. However, the check is not needed because it is not in a^{Γ}. Avoiding proximity checks like this is actually desired, because it takes time but doesn't yield any new results.

If we switch the role of A and B, we can prove that the bridge is also complete for $\Gamma = \Gamma_a - \nu_b$. Either case would guarantee completeness, so we can claim completeness for the larger width value, $\Gamma = \max(\Gamma_b - \nu_a, \Gamma_a - \nu_b)$.

References

[1] Berger, M. J., and P. Colella Local Adaptive Mesh Refinement for Shock Hydrodynamics. *Journal of Computational Physics* **82**:64–84 (1989).

[2] Hornung, R. D., and S. R. Kohn Managing application complexity in the SAMRAI object-oriented framework. *Concurrency and Computation: Practice and Experience* **14**:347–368 (2002).

[3] SAMRAI Project Site. URL **http://www.llnl.gov/CASC/SAMRAI/** Center for Applied Scientific Computing. Lawrence Livermore Laboratory, Livermore, CA.

[4] Chombo - Software for Adaptive Solutions of Partial Differential Equations. URL **http://chombo.lbl.gov/** Applied Numerical Algorithms Group. Lawrence Berkeley Laboratory, Berkeley, CA.

[5] BoxLib. URL **https://ccse.lbl.gov/BoxLib/** Center for Computational Sciences and Engineering Lawrence Berkeley Laboratory, Berkeley, CA.

[6] Kohn, S. R. A Parallel Software Infrastructure for Dynamic Block-Irregular Scientific Calculations. *Ph.D. Thesis, University of California* (1995).

[7] Stout, Q. F., D. L. De Zeeuw, T. I. Gombosi, C. P. T. Groth, H. G. Marshall, and K. G. Powell Adaptive Blocks: A High Performance Data Structure. *Proceedings of the 1997 ACM/IEEE International Conference on Supercomputing* (2010).

[8] Burstedde, C., O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, L. C. Wilcox Extreme-Scale AMR. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010).

[9] Luitjens, J., and M. Berzins Scalable parallel regridding algorithms for block-structured adaptive mesh refinement. *Concurrency and Computation: Practice and Experience* **23**:1522–1537 (2011).

[10] Luitjens, J., M. Berzins, and T. Henderson Parallel space-filling curve generation through sorting. *Concurrency and Computation: Practice and Experience* **19**:1387–1402 (2007).

[11] Luitjens, J. P. The Scalability of Parallel Adaptive Mesh Refinement within Uintah. *Ph.D. Thesis, University of Utah* (2011).

[12] Anderson, R. W., N. S. Elliott, and R. B. Pember An arbitrary Lagrangian-Eulerian method with adaptive mesh refinement for the solution of the Euler equations. *Journal of Computational Physics* **199**:598–617 (2004).

[13] Berger, M., and I. Rigoutsos An Algorithm for Point Clustering and Grid Generation. *IEEE Transactions on Systems, Man and Cybernetics* **21**:1278–1286 (1991).

[14] Gunney, B. T. N., A. M. Wissink, and, D. A. Hysom Parallel Clustering Algorithms for Structured AMR. *Journal of Parallel and Distributed Computing* **66**:1419–1430 (2006).

[15] Wissink, A. M., D. Hysom, and, R. D Hornung Enhancing Scalability of Parallel Structured AMR Calculations. *Proceedings of the 17th ACM International Conference on Supercomputing* (2003).

[16] Bhatele, A., T. Gamblin, K. E. Isaacs, B. T. N. Gunney, M. Schulz, P. Bremer, and B. Hamann Novel Views of Performance Data to Analyze Large-scale Adaptive Applications. *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2012).

Prepared by LLNL under Contract DE-AC52-07NA27344

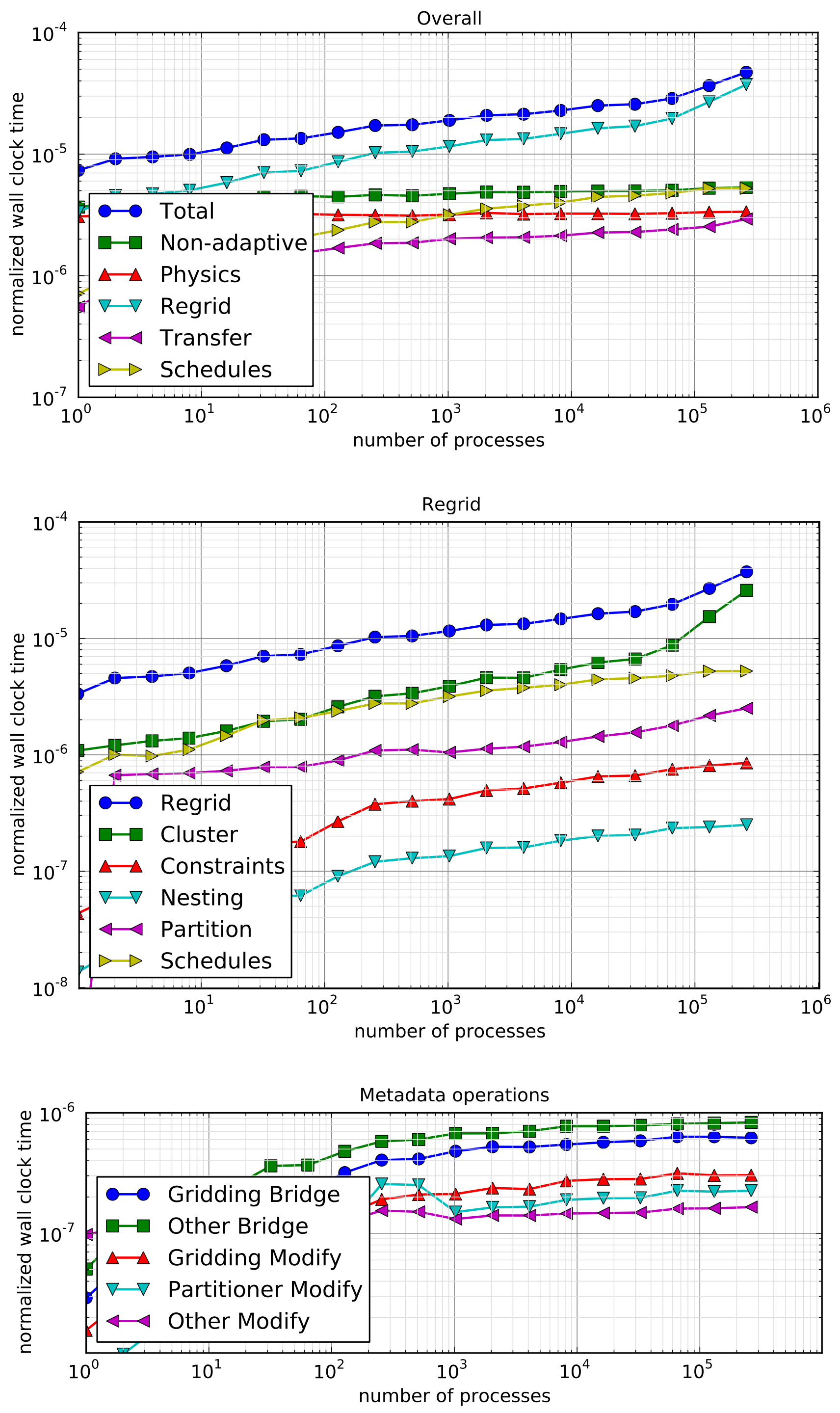


Figure 6: Performance on IBM BG/Q

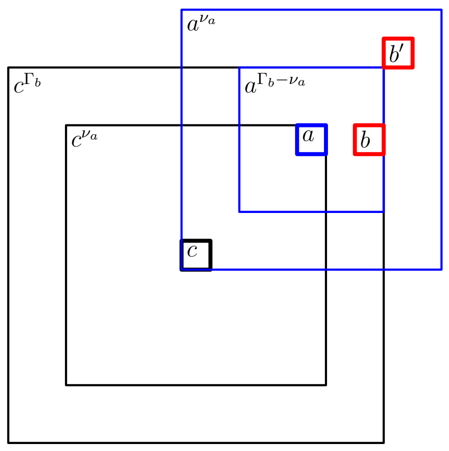


Figure 7: Boxes in completeness proof for bridge theorem.