LLNL-JRNL-668377

Advances in Patch-Based Adaptive Mesh Refinement Scalability

B. T. N. Gunney, R. W. Anderson

March 11, 2015

Journal of Parallel and Distributed Computing

Advances in Patch-Based Adaptive Mesh Refinement Scalability

Brian T. N. Gunney, Robert W. Anderson
Lawrence Livermore National Laboratory[*]
gunneyb@llnl.gov, anderson110@llnl.gov

December 2015

Abstract

Patch-based structured adaptive mesh refinement (SAMR) is widely used for high-resolution simulations. Combined with modern supercomputers, it could provide simulations of unprecedented size and resolution. A persistent challenge for this combination has been managing dynamically adaptive meshes on more and more MPI tasks. The distributed mesh management scheme in SAMRAI has made some progress SAMR scalability, but early algorithms still had trouble scaling past the regime of 10^5 MPI tasks. This work provides two critical SAMR regridding algorithms, which are integrated into that scheme to ensure efficiency of the whole. The clustering algorithm is an extension of the tile-clustering approach, making it more flexible and efficient in both clustering and parallelism. The partitioner is a new algorithm designed to prevent the network congestion experienced by its predecessor. We evaluated performance using weak- and strong-scaling benchmarks designed to be difficult for dynamic adaptivity. Results show good scaling on up to 1.5M cores and 2M MPI tasks. Detailed timing diagnostics suggest scaling would continue well past that.

1 Introduction

Patch-based structured adaptive mesh refinement (SAMR) was first introduced by Berger, Oliger, and Colella [BO84, BC89]. It is a popular approach for adaptive mesh refinement (AMR), with implementations in the BoxLib [Box15], Chombo [Cho15, CGJ+13], AMROC [AMR15], and SAMRAI [HK02, SAM15] libraries, as well as multiple applications that use these libraries.

The SAMR mesh is a hierarchy of levels with increasing resolution (Figure 1a). To refine a region, we tag cells that need refinement and generate the finer level that covers all of the tagged cells. The level generation procedure begins with a clustering step that finds a set of non-overlapping boxes covering the tags, using the Berger-Rigoutsos [BR91] algorithm or some other clustering procedure. It then applies small changes to the boxes to suit the particular constraints of the SAMR implementation, such as requiring the levels to fully nest at each level. Finally, it partitions the boxes by assigning them to MPI tasks, possibly modifying the boxes further in the process. To update a level in the course of a dynamic simulation, we regenerate the level, copy the data over, and discard the old level. Parallel distribution of meshes usually follows the LPARX model [KB96], with each patch assigned to one MPI process, but allowing for multiple patches per process. Hierarchies are distributed level by level to facilitate level operations prevalent in SAMR solvers.

While the concepts in this work probably apply to other SAMR implementations, the work was implemented and tested in the SAMRAI library. SAMRAI is unique in its software interface and flexibility [HK02]. For example, it integrates seamlessly with user data and algorithms without modifying or recompiling the library, and supports arbitrary refinement ratios, such as the odd ratios preferred by ALE-AMR [AEP04]. It was designed from the start to run on distributed memory parallel computers.

Mesh management for large-scale SAMR

[*]This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

emerged as a challenge in SAMRAI around 2003. Wissink, et. al., observed that as the number of MPI tasks exceeded $O(10^2)$, the cost of computing data dependencies via patch proximity overwhelmed everything else. Also, communication time in the Berger-Rigoutsos clustering implementation was slow [WHH03]. The first issue was resolved by replacing the quadratic search algorithm for detecting patch proximity with a tree-based search that had $O(K \log K)$ complexity, where K is the number of patches. The second issue was addressed by grouping MPI tasks to remove extraneous communications in the parallel Berger-Rigoutsos implementation.

Other advances followed in the SAMR field, including an asynchronous Berger-Rigoutsos clustering algorithm [GWH06], a tree partitioner [BGI+12], a tile-clustering approach [LB11, Lui11], a parallel space-filling curve partitioner [LBH07, Lui11] and distributed mesh management [Gun12]. Advances were also made in the oct-tree-based AMR framework ALPS [BGG+10].

These advances pushed AMR computations to the range of $O(10^5)$ CPU cores. Above $O(10^5)$ cores, we have seen limitations in the scalable ranges of the tree partitioner and the asynchronous Berger-Rigoutsos clustering. This work contributes clustering and partitioning algorithms that scale better.

The tree partitioner worked by first arranging MPI processes in a depth-first binary tree. Peer-to-peer communication along the tree's edges determined how much work should be given to or taken from each branch. Work moved up and down the tree, out of overloaded branches and into underloaded ones. The problem with this partitioner was that too much data was being pushed through a few links near the tree's root [BGI+12]. Although the data was tiny compared with the mesh size, it could remain a constant fraction of the mesh size and grow linearly with scale. Eventually, this was too much data to move quickly through a few links. Although we mitigated the problem in [BGI+12] by using more direct communications in one phase of the algorithm and increasing the minimum patch size, the problem was still inherently there, and would just emerge again in a larger simulation.

Clustering algorithms generate boxes covering cells tagged for refinement. The Berger-Rigoutsos algorithm works top-down, beginning with a global cluster around all tagged cells. In a recursive bisection scheme, it breaks the box into smaller parts that could exclude more and more untagged cells. Recursion stops when clustering efficiency reaches a given value. Evaluating the large clusters is communication-intensive, even in the asynchronous implementation.

In contrast, Luitjen's tile-clustering algorithm has no communication. The disadvantage of tiles is the competing influences on tile size. Large tiles are efficient to manage but less flexible for load balancing, and they tend to include more untagged cells, reducing the clustering efficiency. Smaller tiles are easier to partition and better at minimizing inclusion of untagged cells, but lead to more expensive mesh-management operations and require more ghost data and intra-level data transfer.

The current work contributes a completely new partitioner and extensions to tile clustering. The "cascading partitioner" made better use of the available communication network to avoid pushing too much data through a few links. Our extensions made tile clustering more flexible, combining the advantages of small and large tiles. Both algorithms were integrated it into SAMRAI's mesh-management system using scalable operations to ensure the whole regridding process remained scalable.

Much of the work in this paper is described in the context of its implementation; specifically, in SAMRAI's mesh-management context. Because SAMRAI's distributed mesh management is relatively new, we will review it in Section 2 before getting into the contributions of the current work. In Section 3, we describe partitioning principles and challenges relevant to this work. Sections 4 and 5 describe the cascade partitioner and the extended tile-clustering algorithm. Section 6 gives weak- and strong-scaling results and examines load balance and data locality related to the new algorithms. Conclusions are given in the last section.

2 Mesh management context

This section reviews the basic concepts and terminology in SAMRAI's mesh management [Gun12], including mesh metadata, `BoxLevels`, `Connectors`, and two important operations that we have called "bridge" and "modify." We will describe the novel environment in which the partitioning and clustering algorithms work and introduce terminology to discuss them in sufficient detail.

Mesh ***metadata*** is the data that describes the mesh. For a patch-based mesh distributed in parallel, this includes the boxes that form the patches and the owners of those patches. To facilitate referencing specific boxes, SAMRAI adds to each box an identifier that is globally unique in the container holding the box. Metadata also includes data derived from the boxes, such as which box is close to which. It can refer to similar data used during mesh generation and adaptation and other steps that manipulate distributed sets of boxes. ***Mesh management*** is the generation, acquisition (through communication), storage, and processing of metadata. The partitioning and clustering algorithms presented in this paper both exercise of mesh management.

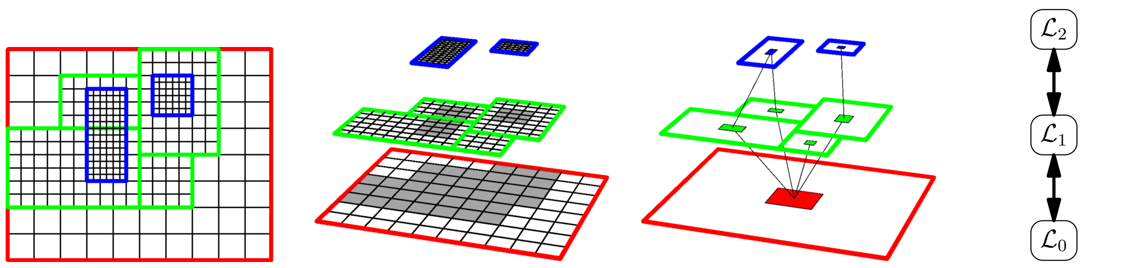
Early versions of SAMRAI used the traditional approach of storing global metadata on every process. This turned out to be prohibitive for large-scale computation. Starting in Version 3, SAMRAI uses distributed mesh management, storing metadata in two container classes, *`BoxLevel`* and *`Connector`*. A `BoxLevel` is a distributed box container (Figure 1b). Locally owned boxes are called ***local***, and others are ***remote***. Each process stores only local boxes in a `BoxLevel`. Relationships between two `BoxLevel`s are stored in `Connector`s(Figure 1c). A `Connector` holds references to two `BoxLevel`s, called the base and the head. For each local base box, a process can store in the `Connector` a set of neighboring head boxes. Borrowing from graph terminology, we can say that `Connector`s are adjacency lists distributed and stored in a data structure convenient for SAMRAI.

The neighbor of a local box can be local or remote. Relationships between two local boxes are called local relationships. Those between a local and a remote box are called ***semi-local*** relationships. There is usually no reason to deal with relationships between two remote boxes, which is costly due to the sheer number of remote boxes.

`Connector`s are simply containers of relationships. Although SAMRAI uses `Connector`s for many things, there are two common uses. The most common use is for holding ***proximity*** relationships, such as those in Figure 1c. For each base box, a proximity `Connector` stores head boxes that are close to the base box, typically nearest neighbors. The head and base `BoxLevel`s can be the same, in which case, the `Connector` stores intra-level proximity relationships. The second most common `Connector` use is to hold ***mappings*** that describe how a `BoxLevel` changes, such as when it undergoes partitioning. For each base box, a mapping `Connector` can hold the set of boxes that the base box will become when the change is applied. Boxes to be removed by the mapping are given empty neighbor sets, but we can omit neighbor sets outright for boxes that do not change. This omission provides for significant optimizations in mappings with minimal changes.

`Connector`s are filled and manipulated by a variety of algorithms that generate or find box relationships. Two generic utility-level algorithms commonly performed on `Connector`s has been factored out: the ***bridge*** algorithm and the ***modify*** algorithm. They are schematically illustrated in Figure 2. Both algorithms compute new proximity relationships but for different inputs. Bridging computes a new proximity `Connector` between `BoxLevel`s $\mathcal{A}$ and $\mathcal{B}$ if their proximity to a `BoxLevel` $\mathcal{C}$ is known. The modify operation updates a `Connector` if its head $\mathcal{D}$ is changing to $\mathcal{D}'$ according to a specified mapping `Connector`. Both algorithms have been described in [Gun12].

Figure 3 illustrates SAMRAI's use of bridge and modify during regridding, as a new level is generated to replace an old one. Our regridding process was similar to those described by Berger and Colella [BC89] but had more a few more steps for adjusting boxes. Each double-headed arrow in the figure represents two `Connector`s pointing in opposite directions. Green `Connector`s indicate derivative relationships, where a new `BoxLevel` was created by an operation on an existing level. For example, clustering derives `BoxLevel` "Temp 1" by operating on the ***tag level*** $\mathcal{L}_{n-1}$ (the level holding tags indicating which cells should be refined). This derivative relationship works as a proximity relationship. The generated `BoxLevel` then goes through a sequence of manipulations, as labeled in the figure (fix overflow, enforce nesting, grow to boundary, refine and partition), to make it suitable for constructing a patch level. Each step computes a new temporary `BoxLevel` and sets up a new derivative relationship that works as a



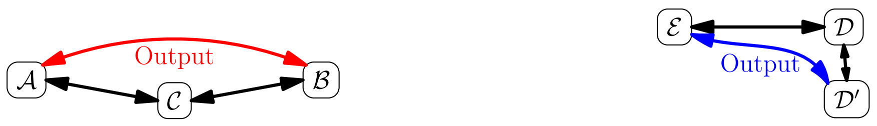
(a) Composite mesh

(b) Mesh as sequence of `BoxLevels`

(c) Inter-level proximity `Connectors`

(d) Symbolic representation

Figure 1: Representations of the same 3-level mesh. Patches are outlined in red, green and blue for levels 0, 1 and 2. Intra-level `Connectors` are not shown. The last image show symbolically the three levels and their inter-level `Connectors`.



(a) $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are `BoxLevels`. Bridge computes red proximity `Connector` using data in black proximity `Connectors`.

(b) Modify computes blue proximity `Connector` given original `Connectors` between $\mathcal{E}$ and $\mathcal{D}$ and the mapping from $\mathcal{D}$ to $\mathcal{D}'$.

Figure 2: Inputs and outputs of bridge and modify operations. (Recall that `BoxLevels` and `Connectors` are *containers*. Each `BoxLevel` symbol represents multiple boxes, and each arrow represents multiple relationships between boxes in connected levels.) Double-headed arrows represent transpose pairs, which are two `Connectors` pointing in opposite directions. See Section 2 for an overview of `BoxLevels`, `Connectors` and bridge and modify operations.

mapping. We apply the modify operations using those mappings to compute the updated proximity `Connectors` shown in blue. After the new level $\mathcal{L}_n$ is determined, along with its proximity to $\mathcal{L}_{n-1}$, bridging provides proximity `Connectors` shown in red. First, we bridge for the `Connector` between the old and new $\mathcal{L}_n$ levels. Then we use that result in another bridge, giving the `Connectors` between $\mathcal{L}_n$ and existing $\mathcal{L}_{n+1}$.

While the steps from clustering to partitioning, shown in Figure 3, are not generally new for patch-based AMR, traditional algorithms for performing these steps require globalized metadata. The new algorithms described in [Gun12] work for distributed metadata if the `Connectors` shown are available. So the `Connectors` in Figure 3 are not just illustrations, but essential data for creating the new mesh level with all of the necessary metadata for computation on the hierarchy.

2.1 Metadata notations

We will use script letters to symbolize `BoxLevels`, for example, $\mathcal{A}$ and $\mathcal{B}$. A `Connector` with base $\mathcal{A}$ and head $\mathcal{B}$ will be shown as $\{\mathcal{A} \to \mathcal{B}\}$. Borrowing again from graph terminology, we read this as "a set of directed edges from boxes in $\mathcal{A}$ to boxes in $\mathcal{B}$." A `Connector`'s ***transpose*** has the same edges, but they point in the opposite direction. The transpose has a different base, so its edges are distributed differently (unless it is an intra-level `Connector`). The transpose of $\{\mathcal{A} \to \mathcal{B}\}$ is written $\{\mathcal{B} \to \mathcal{A}\}$. A `Connector` and its transpose make a ***transpose pair***, and we represent it using a double-headed

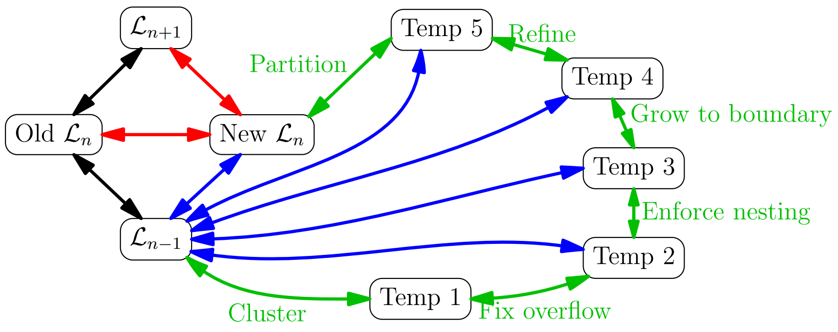


Figure 3: `BoxLevels` and `Connectors` during the generation of a new level to replace an old one. Double-headed arrows represent pairs of `Connectors` pointing in opposite directions. `BoxLevels` denoted by $\mathcal{L}$ are hierarchy levels, with the subscript denoting level number. `Connectors` are 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.

arrow, $\{\mathcal{A} \Leftrightarrow \mathcal{B}\}$. The double-headed arrows in Figures 2 and 3 all represent transpose pairs.

Each `Connector` has an associated non-negative ***connector width***, quantifying an important characteristic of its stored relationships. The width is an integer vector in the logical (i,j,k) space, but if all components are the same, we can use a single integer representation. The width of a proximity `Connector` specifies how close two boxes must be to each other to have a relationship. For a mapping `Connector`, it specifies how much the mapping may extend a level outside its old boundary. When the width is relevant and not clear from context, we add it to the `Connector` notation, as in $\{\mathcal{A} \overset{\omega}{\Longleftrightarrow} \mathcal{B}\}$. Otherwise, we leave it out. By convention, the width's unit is cells and always specified in the resolution of the `Connector`'s base `BoxLevel`. For a transpose pair, where the base of one `Connector` is the head of the other, the width is specified in the resolution of the left `BoxLevel`.

3 Partitioner challenges and design goals

The primary objective for SAMRAI's partitioning algorithm is to distribute equal work to all processes, so that no process has an excessive amount of work. The secondary objective is to reduce the cost of transferring data between patches. We accomplish this by reducing partition surface area and improving data locality. Reducing partition surface area reduces places where data has to be transferred during simulation. Some partitioning characteristics, such as low surface-to-volume ratio and contiguous partitions, correlate with reduced partitioning boundaries. Data locality refers to how close the data-exchanging partitions are to each other on the network. It is generally accepted that the closer they are, the less time the data would spend in transit.

For dynamically adaptive meshes, partitioners are run frequently, typically at every regrid, so they must be fast. This is in contrast to static meshes, which are partitioned only once, and for which partitioning speed is not as critical. Fast partitioning can be at odds with good partitionings. Time saved by an improved partition can be negated by time spent obtaining that improvement. Some compromise between partitioning quality and speed is often necessary because it is difficult to achieve both for general configurations.

A practical challenge for SAMR is that off-the-shelf partitioners are not very well suited for the task. Two popular categories of mesh partitioners are graph partitioners [DBH$^+$02, BDL$^+$15, KK15, KK99, CP08] and space filling curves (SFC) [LBH07]. Both types partition a set of work units (such as cells or fixed-sized boxes). Both view work units as atomic and assume they are fine-scaled. When treated as atomic, arbitrarily large boxes in SAMR make fine-scale adjustments impossible. Some patch-based partitioners eliminate large boxes by pre-cutting them and then applying an off-the-shelf partitioner. However, this introduces compromises similar to those discussed in choosing a tile size.

The primary design goals for our partitioner are the same as for all dynamic partitioners: balanced load, data locality, minimal surface area, and partitioning speed. Secondary objectives, which follow from the primary and apply specifically to patch-based AMR and SAMRAI, are given in the rest of this section.

Low box count: Fewer boxes means less fragmentation of patch levels, indirectly improving data locality because cells in the same box always reside on the same process. Lower box counts have the added benefit of requiring less data transfer between and within processes. They also have reduced mesh-management overhead because there are fewer boxes to manage. Since fewer boxes leads to more stride-one data layout, cache efficiency can also benefit. Ideally, each process would have just one box, though this is not achievable for arbitrary box configurations. A more reasonable goal is to limit the breaking of boxes as much as possible.

Low aspect ratio: Low aspect-ratio boxes correlate with low surface-to-volume ratios and lead to less partition surface area. When box cutting is necessary, we favor cuts that yield lower aspect ratios.

SAMRAI integration: Input and output must be distributed with each box in the local memory of its owner. In the SAMRAI framework, this implies `BoxLevel` format. If a partitioner requires additional information, such as a global view of the input, SAMRAI sees acquiring that information as a part of the partitioner cost. Likewise, a partitioner that does not naturally have every output box on its owner process must take steps to put them there. The placement of each box on its owner process facilitates scalable integration of mesh-management components.

Proximity to reference level: SAMRAI's distributed mesh management requires proximity data expressed through `Connector`s, as Figure 3 shows. SAMRAI provides the initial `Connector`s between the partitioner's input boxes and a reference level (usually the tag level) and expects the partitioner to update this proximity data when it changes the boxes. The partitioner, responsible for the decision to break up boxes and reassign them, is in the best position to update proximity relationships in a scalable way. (SAMRAI still works if this data is not provided, but a slower general search operation is required to find proximity relationships.)

4 Cascade partitioner

The cascade partitioner is a top-down box-partitioning algorithm. First, it shifts load between the two halves of the machine to balance their work. In this step, it does not try to redistribute loads within each half. To redistribute within each half, the same algorithm is applied recursively until the halves are single processes, thus allowing load to "cascade" to the final destination. We focus mainly on the algorithm for balancing the two halves of a contiguous set of process ranks.

Let us define a group as a contiguous range of MPI process ranks, written as {first rank-last rank}. We use the term ***weight*** to describe a total amount of work, which is a single number. We use the term ***load*** to describe not only the work but its representation as a collection of boxes. The distinction is important because in distributed mesh management, what a process can know about remote processes is the weight they hold, but not the load. Partitioning for a group means balancing its upper and lower halves but not balancing within each half.

Partitioning a group took four main steps.

1. Determine weight of each half.
2. Determine how much work a process in the heavy half should give up.
3. Determine send/receive pairs for inter-group load transfer.
4. Apportion and send (heavy half) or receive load (light half).

Figure 4 shows the groupings and communication patterns of this algorithm applied to an eight-process group. Step 1 starts with single-process group weights, which are simply the current local loads. These are the Grouping-3 groups in Figure 4. The weights of bigger groups are built bottom-up

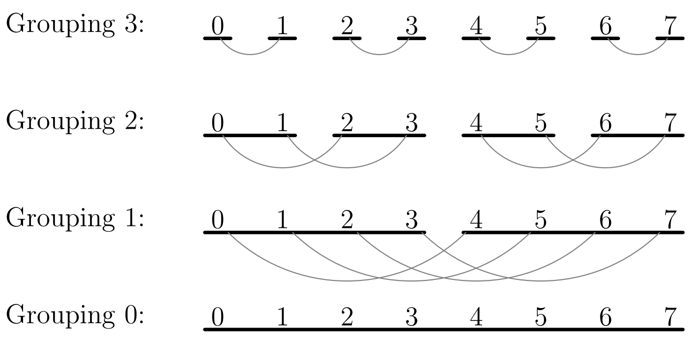


Figure 4: Cascade groups for 8-process machine, ranks 0-7. Each underscore indicates a group of processes. Recursive bisection of groups creates a binary tree where each node is a group. Each process belongs to 4 groups. Grouping 0 has the root node which includes all processes. Grouping 3 has single-process groups. Arcs connecting processes indicate inter-group communication pairs.

by a sequence of message exchanges by contacting pairs in the figure. The first exchange, using the pairs in Grouping 3, gives all processes the weights of their Grouping-3 sibling groups and, by summing, the weight of their Grouping-2 parent group. For example, process 0 would know the weights of groups {1} and {0-1}. The second exchange, using Grouping-2 contacting pairs, gives information on the Grouping-2 sibling groups and the Grouping-1 parent group. Process 0 would now know the weights of groups {2-3} and {0-3}. A third exchange gives all processes the weights of the two Grouping-1 groups, {0-3} and {4-7}, completing step 1.

After step 1, all processes can determine which half is too heavy and how much load should be transferred to the lighter half. As a by-product of this step, each process knows the weights of all groups it belongs to and the weights of their sibling groups. This data is used in the next step.

In Step 2, each half of the group has the obligation to take half of the group's weight, so the heavy half would ideally send the difference between its weight and its obligation. Each process in the heavy half must determine whether it should send work to the light half and if so, what is the ideal amount to send. Without knowing what other processes in its group are sending, the process can overcompensate for the imbalance by sending too much. To prevent this, we devised two rules for computing exactly how much a process should give up. First, processes closer to the recipient side (in rank space) had priority. A process farther away could give load only when all closer processes had given what they could and the light half still needed more. Second, each process should give as much as it can, up to a maximum. The maximum was the amount needed or the local surplus, whichever was less. Following these criteria and using the group weights acquired in Step 1, each process could compute exactly how much load other processes closer to the light side should give up and thus how much load it should give up. This scheme used data acquired in Step 1, so it required no additional communication.

In Step 3, each process in the heavy half is paired with a process in the light half, so they can send work they give up to the light half. For this step, we choose the same contacting pairs we used in Step 1. Note that each process exchanges data with a contact that has the same relative rank. A process's relative rank is its MPI rank minus the first MPI rank in its group. In this step, it does not matter if the receiver already has enough work, because we are not yet concerned about balancing within the halves.

Apportioning the load in Step 4 was a matter of bringing a container of loads to the correct weight by transferring load between it and a holding bin. The holding bin contains all boxes the local process has in its possession at that moment. We outline the load apportioning algorithm in Section 4.4. Once apportioned, the donor packs the work into a message for sending.

Note that moving load meant moving only metadata in the form of boxes, not voluminous mesh data. This allowed huge portions of the mesh to be moved without moving huge amounts of data. We would initialize the mesh data after connecting the new level to the hierarchy, which provides that data (Figure 3).

As mentioned above, the algorithm we just described only balanced the two halves of the machine. To complete partitioning, it must be applied recursively to each half. Conceptually, one could write a recursive function to perform this. However, we implemented it as a loop executed $\lceil \lg N \rceil$ times.

We can now address two details in this simplified description of the algorithm that require more attention: handling odd group sizes and dealing with imperfect load apportioning.

Handling odd group sizes required a small modification. For groups of size 2^n, such as those in Figure 4, the recursive bisection generated even-sized groups that were straightforward to split. Sibling groups are always the same size, so computing contacting partners was trivial. For odd group sizes, we split the group next to the middle rank and gave the middle rank to one half. The last rank in the smaller half had to contact the last *two* ranks in the larger half. If it had to send work, it did not matter which of the two contact to send to. We chose to send to the first contact. The rest of the algorithm remained unchanged.

Imperfect load apportioning refers to the inability to apportion arbitrary weights. The cause is the inability to cut a box into arbitrary sizes. Cuts must be along grid lines, and sometimes only along certain grid lines, a restriction coming from the need to generate "valid" SAMR meshes. SAMR implementations often allow users to specify a minimum patch size, further limiting the combination of cuts allowed. How should we compute the weight a process should give up when we did not know exactly what other processes were giving up? Our solution was to estimate what other processes were giving up based on what they were *expected* to give up, regardless of imperfect load apportioning. The resulting imbalance caused by estimating would be no worse than that caused directly by the inability to apportion the correct weight.

Imperfect load apportioning may produce group average weights higher than the global average because weight transferred was not exact. This could lead to high overloads if we allowed that surplus to be stuck on too few processes. To spread a group's surplus evenly, we only had to split it evenly among the group's two halves. To induce the algorithm to do this, we reset each half's obligation to half of the group's new weight before continuing the recursion.

4.1 Cascade partitioner complexity

The algorithm to balance a group's two halves has complexity $O(\lg N)$, where N is the number of MPI tasks. The individual steps have the following complexity. Step 1 uses a loop of length $\lg N$ to determine the weight of the halves being balanced. Messages exchanged in Step 1 have $O(1)$ lengths, and current supercomputers are expected to complete the communication in constant time. Step 2 examines the limited view of group weights acquired in Step 1, so it also has complexity $\lg N$. Steps 3 is independent of N and has constant complexity. Apportioning the load in Step 4 has complexity dependent on the size of the load container but independent of N. Sending the load also has complexity independent of N. Thus, the four steps together have complexity $\lg N$. These steps are repeated once for each of the $\lg N$ levels in Figure 4. Thus the complexity for partitioning every group has complexity $O(\lg^2 N)$.

4.2 Post-partition proximity data

SAMRAI provides the partitioner with `Connector` pair $\{\mathcal{R} \Leftrightarrow \mathcal{P}\}$ containing proximity relationships between the `BoxLevel` $\mathcal{P}$ to be partitioned and a reference `BoxLevel` $\mathcal{R}$. It expects the partitioner to update that data when changing $\mathcal{P}$. The standard way to update $\{\mathcal{R} \Leftrightarrow \mathcal{P}\}$ is to set up mapping `Connectors` $\{\mathcal{P} \Leftrightarrow \mathcal{P}'\}$ describing the redistribution of work and then use the modify operation to update $\{\mathcal{R} \Leftrightarrow \mathcal{P}\}$.

To build the mapping transpose $\{\mathcal{P}' \rightarrow \mathcal{P}\}$, processes must know the original form of each post-partition box they acquired. This can be easily accomplished by attaching to each box a copy of its originator. If a box gets broken up, every part gets a copy of the originator. Wherever the box goes, it carries the originator information. The boxes and their originator data provides all the information needed to populate the backward mapping $\{\mathcal{P}' \to \mathcal{P}\}$.

To build the forward mapping $\{\mathcal{P} \to \mathcal{P}'\}$, processes must know the final forms of the pre-partition boxes they originally owned. Building $\{\mathcal{P} \to \mathcal{P}'\}$ requires communication. Processes that give up their original boxes could not have known what became of the boxes, so they must receive information from the final owners. Though they do not know what senders to receive from, they can receive from any process until they have accounted for all the cells they gave up. Obviously, the final owners must send the final boxes to the original owners.

4.3 Correcting gross initial imbalance

Gross initial imbalance is when very few processes own virtually all the work in the pre-balance `BoxLevel`. Though this problem occurs most often during initial mesh generation, it can occur during regridding as well. When generating level 0, the few boxes defining the full domain are given to process 0 before partitioning. A consequence is that even if distributing the load were scalable, the representation of $\{\mathcal{P} \Leftrightarrow \mathcal{P}'\}$ would not be. Process 0 may have ended up with a reasonable share of the work, but it would have to store edges to every remote box in $\mathcal{P}'$. That data would have $O(N)$ size, and the modify operation applied to it would have $O(N)$ complexity. During regrid, this condition most commonly occurs when a tag level , due to its size, was distributed to a small fraction of a large MPI group. Consequently, a small number of tasks would own the resulting clusters. Figure 5a shows a simple example of gross imbalance with $N = 4096$.

To avoid forming a huge $\{\mathcal{P} \Leftrightarrow \mathcal{P}'\}$, we can gradually balance using multiple steps. With this technique, We transform the grossly imbalanced level $\mathcal{P}$ through a sequence of k states $\mathcal{P}, \mathcal{P}_1, \mathcal{P}_2, \ldots \mathcal{P}_k$, where $\mathcal{P}' = \mathcal{P}_k$. Figure 5b shows the 4096-task example with $k = 3$. The transformation can be performed through the mappings

$$\{\mathcal{P} \Leftrightarrow \mathcal{P}_1\}, \{\mathcal{P}_1 \Leftrightarrow \mathcal{P}_2\}...\{\mathcal{P}_{k-1} \Leftrightarrow \mathcal{P}_k\} \quad (1)$$

This changes $\{\mathcal{R} \Leftrightarrow \mathcal{P}\}$ through the sequence

$$\{\mathcal{R} \Leftrightarrow \mathcal{P}_1\}, \{\mathcal{R} \Leftrightarrow \mathcal{P}_2\}...\{\mathcal{R} \Leftrightarrow \mathcal{P}_k\} \quad (2)$$

with the last `Connector` being the final $\{\mathcal{R} \Leftrightarrow \mathcal{P}'\}$.

We limit the size of each mapping by limiting the number of processes to which loads from a single process can spread. Let $f = \frac{w_{\max}}{w_{\text{avg}}}$, where $w_{\max}$ is the maximum initial weight for any process and w_{avg} is the average weight. f is the greatest spreading an initial load must undergo if we used the single-step map $\{\mathcal{P} \Leftrightarrow \mathcal{P}'\}$. Let us limit to g the spread of any load by any mapping in (1). The first mapping would spread $w_{\max}$ out to g processes. The second would spread those g portions to g^2 processes, and so on. We could achieve the desired spread of f using k mappings if $g^k \geq f$. Thus, we replaced the single-step map with

$$k = \left\lceil \frac{\lg f}{\lg g} \right\rceil \quad (3)$$

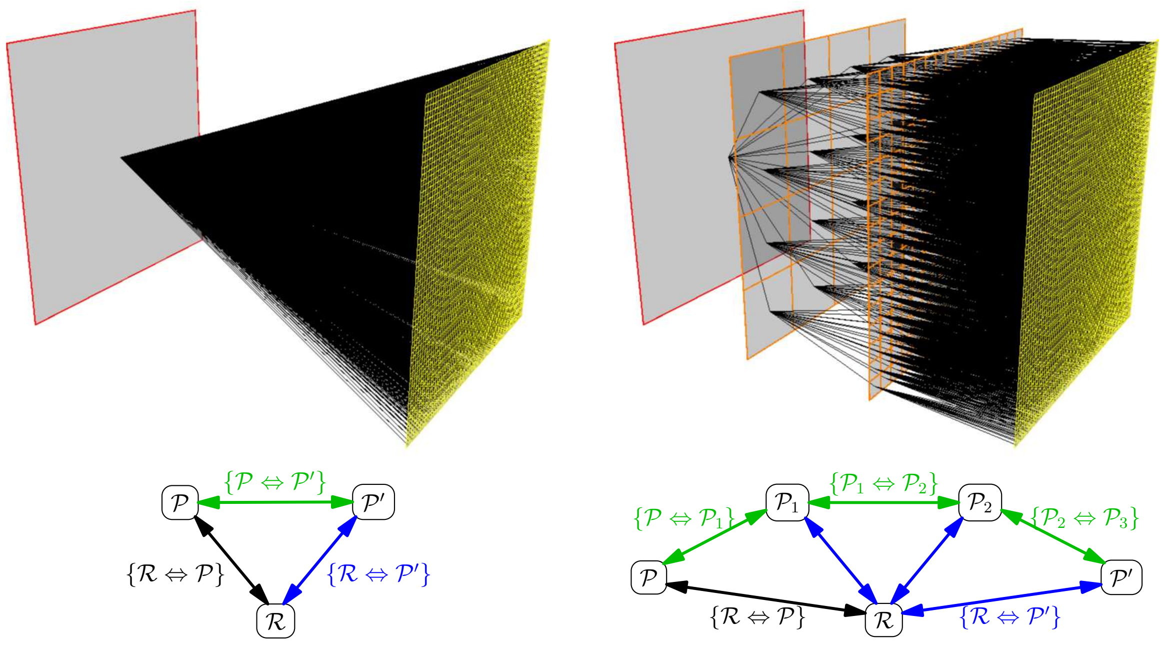
gradual mappings. For a run with N MPI ranks, the group tree (Figure 4) would have depth $m = \lceil \lg N \rceil$. We update $\mathcal{P}$ and $\{\mathcal{P} \Leftrightarrow \mathcal{R}\}$ after about every $\frac{m}{k}$ load exchanges, with the last update always occurring at the end. In addition to the reduction in storage space for the mapping `Connectors`, the multi-step process is also faster in time, because $kg \ll f$ in the typical cases we encounter when there is a large initial imbalance.

4.4 Load apportioning

Load apportioning refers to setting aside a given amount of work to be sent off. Given a load container with some weight (possibly zero), the load apportioning algorithm tries to shift load between it and another container, to bring the first container's weight to a specified amount. We shift weights in three main ways: by moving boxes from one container to the other; by swapping boxes between containers; and by cutting boxes to move a part of their load.

SAMRAI's apportioning algorithm is largely heuristic, based on a few strategies that guide low-level decisions to cut and move boxes.

1. We prefer moving whole boxes over breaking boxes, to reduce the creation of new box boundaries. New box boundaries add to data transfer costs and can reduce data locality by allowing the box fragments to move away from each other on the communication network.



$\{\mathcal{P} \Leftrightarrow \mathcal{P}'\}$
$\mathcal{P}$
$\mathcal{P}'$
$\{\mathcal{R} \Leftrightarrow \mathcal{P}\}$
$\{\mathcal{R} \Leftrightarrow \mathcal{P}'\}$
$\mathcal{R}$
$\{\mathcal{P}_1 \Leftrightarrow \mathcal{P}_2\}$
$\{\mathcal{P} \Leftrightarrow \mathcal{P}_1\}$
$\mathcal{P}_1$
$\mathcal{P}_2$
$\{\mathcal{P}_2 \Leftrightarrow \mathcal{P}_3\}$
$\mathcal{P}$
$\mathcal{P}'$
$\{\mathcal{R} \Leftrightarrow \mathcal{P}\}$
$\mathcal{R}$
$\{\mathcal{R} \Leftrightarrow \mathcal{P}'\}$

(a) Abrupt (b) Gradual

Figure 5: Abrupt vs. gradual partitioning. Upper images show an example with 4096 processes. In gradual partitioning, the single box partitioned into 16 boxes, then 256, then 4096. Schematics show how overlap connections between the partitioned `BoxLevel` $\mathcal{P}$ and a reference `BoxLevel` $\mathcal{R}$ were updated. Black `Connectors` are inputs. Green ones are mappings. Blue ones are results from modify operations. Partitioning transforms $\mathcal{P}$ to $\mathcal{P}'$ and $\{\mathcal{R} \Leftrightarrow \mathcal{P}\}$ to $\{\mathcal{R} \Leftrightarrow \mathcal{P}'\}$. Abrupt partitioning does it with a single mapping $\{\mathcal{P} \Leftrightarrow \mathcal{P}'\}$. When $\mathcal{P}$ is grossly imbalanced, the map's local size can be excessive (4096 edges in this example). Gradual partitioning forms multiple mappings that are much smaller locally (16 edges each). See Section 4.3.

2. When moving loads, we prefer moving fewer big boxes over more small boxes, to speed the partitioning algorithm and avoid load recipients getting large numbers of small boxes from multiple sources.

3. When breaking a box, we prefer cuts that create fewer new boundaries.

4. We use a weight tolerance for treating small imbalances as being "effectively zero". This avoids fragmenting the domain into small boxes that add mesh-management cost without significantly improving the balance. More importantly, it provides the flexibility needed to consider other partitioning objectives.

We believe this approach leads to generally good–though not necessarily optimal–partitioning. We did not systematically experiment with other strategies or combinations of strategies that may improve partition quality or partitioning speed. To be useful in dynamic mesh adaptation, such experiments should check that improvements gained were not negated by the cost of getting the improvements.

The apportioning algorithm uses a combination of swapping and box breaking to bring the main bin's weight to the target value. (Moving a box from one container to the other can be considered a degenerate case of swapping.) The algorithm first tries to swap boxes between the two containers until it cannot find any swap choices that would bring the weight closer to the target value. Next, it breaks up a single box to shift part of it over. If this succeeds, the change might open up more swap options, so the algorithm retries the swapping-breaking cycle. The cycle stops when breaking fails to make a change. After each swap or break, the algorithm checks the main bin's weight and stops if that weight is close enough to the target value.

There are numerous details of the load apportioning algorithm, such as how to choose box cutting directions and cutting planes. We will describe details that are most important to partitioner performance and partition quality. For lower-level details, most of which are heuristic, we refer the reader to the publicly available SAMRAI source code [SAM15].

We keep load containers sorted by the weights of individual items inside, to speed the search for swap pairs or boxes to break. Searching for a pair to swap has complexity $O(l_1+l_2)$, where the l terms are the lengths of the two containers.

To discourage cuts that would lead to small or high-aspect-ratio boxes, we evaluate boxes using a simple score, $\min\left(\frac{s_{\min}}{s_t}, 1\right)$; where $s_{\min}$ is the length of the box's shortest side; the threshold length $s_t = \sqrt[d]{V}$, where V is the volume associated with the ideal work per process and d is the spatial dimension. The scoring function is one for boxes larger than s_t and shrinks in proportion to the smallest side. Boxes wider than the threshold are not too small, so they are all equally favored with a score of one. When multiple configurations are found to satisfy a given weight shift, we choose the configuration with the highest box score.

We use two box-cutting schemes in SAMRAI: planar and cubic. The planar cut is a single cut across the box. We try to cut a longer side before considering a shorter one. The cubic cut aims to carve out a portion with "nearly" equal sides at one of the box's corners. The sides of the cubic cuts are limited by the size of the box being cut, of course, and could be highly unequal. When both planar and cubic cuts give work amounts within the acceptable range, we choose the one with the higher box score. Multiple leftover boxes are given a width score equal to the product of the individual box scores. This tends to favor planar cuts because they give at most one leftover box, though this would only differentiate boxes with a width below the threshold.

We use a weight tolerance, $w_t = \chi w_{\text{avg}}$, where χ is a small constant, to determine when an apportioned weight is close enough to its target weight and also for comparing multiple apportioning results. Apportioned weights within w_t of the target weight are considered close enough. Once within this range, the load apportioning algorithm returns, regardless of whether further improvements are possible. When comparing two apportioning results, all weights within w_t of the target are considered equivalent. Distinctions between apportioning results within this tolerance are based on box scores.

Note that our load-apportioning algorithm does not consider proximity of boxes in computational space, nor does it consider proximity of processes on the communication network. This is a limitation of this approach. However, it is somewhat mitigated by the resistance to worsening locality, e.g., by not cutting boxes until absolutely necessary. Input boxes tend to have good data locality, because a reasonably intelligent box-clustering implementation can easily give them the data locality characteristics of the tag level. We did see some increased communication times during data transfer when scaling our benchmarks to larger machines, but it was not clear at this point whether better locality would significantly improve the communication time.

5 Tile clustering

The main goal of clustering is to find a set of non-overlapping boxes that contains all cells the user has tagged to be refined and as few untagged cells as possible. However, performance considerations, both of the tiling process itself and for the resulting patch layout, may influence the best choice of clustering algorithm. In tile clustering, we select boxes from a pre-determined tile pattern rather than computing each one. Our tile-clustering algorithm was based on Luitjen's [LB11]. The essential approach is to overlay the rectangular (i,j,k) space with a grid of tiles of fixed size τ, measured in integer number of cells. If a tile overlaps any tagged cells, we generate a box for that tile.

Tile clustering is very simple and very fast. Unlike the Berger-Rigoutsos algorithm [BR91, GWH06], it avoids generating, acquiring, and searching through histograms of tags. This saves communication and–when the bounding box of the tags is very big–memory. Moreover, if the tag-level patch boundaries coincide with tile boundaries on the level being generated, each generated tile sits entirely inside its tag-level box, and the clustering process can be completely local. This is trivial to achieve by using the same logical tile size τ on tag and new levels, regardless of the different resolutions.

The tile approach, like the block-structured mesh approach, requires choosing a tile size that balances clustering efficiency with mesh-management speed. Smaller tile sizes improve clustering efficiency by excluding untagged cells at a finer granularity. However, it takes more small boxes to cover the tags, which slows down mesh-management operations and creates more data-transfer surfaces. In our implementation, we sought modifications that allowed us to use small tile sizes yet still reduce the number of boxes.

We extended Luitjen's scheme in three ways. First, our implementation worked when tag-level patches and tile boundaries did not line up, resulting in tiles overlapping multiple tag-level patches, including remote ones. We needed this extension because SAMRAI's gridding algorithm allows applications to use arbitrary partitioners through a virtual interface. Those partitioners may produce boxes that cross tile boundaries. Second, we kept tile size relatively small for better clustering efficiency, but we added the additional step of coalescing the generated boxes to reduce their numbers. Third, as with the partitioner, we must generate proximity data connecting the generated boxes to the tag level. The following two subsections describe the first two extensions. Generating proximity data was integrated into both extensions.

5.1 Non-local tile clustering

We began by applying Luitjen's local tiling algorithm without modification. Tags were stored in the tag level, represented by $\mathcal{T}$. Each process looped through its tag-level patches and generated a box matching a tile wherever it found tags overlapping the tile. These boxes (also called clusters in this context) were stored in a `BoxLevel`, $\mathcal{C}$. As we did this, we populated local proximity relationships in `Connector` $\{\mathcal{T} \overset{0}{\Longleftrightarrow} \mathcal{C}\}$. Up to this point, this was a completely local step, requiring no communication.

Two problems came up when tag-level patch boundaries did not align with tile boundaries. First, tiles that overlapped multiple patches may be duplicated because tags may exist on multiple patches. If those patches were on different processes, the processes would not know about the duplicates. Second, semi-local proximity relationships would be absent from $\{\mathcal{T} \overset{0}{\Longleftrightarrow} \mathcal{C}\}$. Neither problem could be fixed without communication.

We found missing semi-local relationships by bridging across $\{\mathcal{C} \overset{0}{\Longleftrightarrow} \mathcal{T}\}$ and $\{\mathcal{T} \overset{\Gamma}{\Longleftrightarrow} \mathcal{T}\}$ and reassigning the result back to $\{\mathcal{C} \overset{0}{\Longleftrightarrow} \mathcal{T}\}$. According to the bridge theorem in [Gun12], this bridge could find all missing relationships if $\Gamma \geq \tau$, where τ is the tile size. SAMRAI has mechanisms to ensure $\{\mathcal{T} \overset{\Gamma}{\Longleftrightarrow} \mathcal{T}\}$ is available.

Once we have semi-local relationships in $\{\mathcal{C} \overset{0}{\Longleftrightarrow} \mathcal{T}\}$, we could detect duplicate clusters and delete them without further communication. We scanned $\{\mathcal{C} \xrightarrow{0} \mathcal{T}\}$ to find all $c \in \mathcal{C}$ such that c overlaps multiple boxes in $\mathcal{T}$. For each set of duplicate clusters, we kept only the first one from the lowest-ranked process.

SAMRAI allows arbitrary-sized computational domains that do not necessarily conform to tile boundaries. This allowed clusters to extend outside not only the patch that generated them but also outside the mesh boundary. These were sheared off in a simple operation that did not require any communication. After the change, another modify operation updated $\{\mathcal{C} \overset{0}{\Longleftrightarrow} \mathcal{T}\}$. This simple shearing step also effectively removed pieces of boxes that crosses block boundaries in multi-block mode. (SAMRAI disallows patches crossing block boundaries.)

5.2 Coalescing clusters

Coalescing clusters, our second extension to Luitjen's tile-clustering scheme, reduced the number of boxes during regridding and eventually in the mesh. Solver performance typically also improved or at least did not noticeably degrade.

Coalescing need not be perfect to help, and it should not be treated as a global optimization problem. Even if we could coalesce to get a single cluster, it would be unwise to do it because we would have to cut up the cluster again during partitioning and deal with gross imbalances.

Rather than coalescing globally, each process coalesced its own clusters. This was a local operation, as was setting up the mapping `Connector` to describe the change. The only required communication added by coalescing was in the modify operation to update $\{\mathcal{C} \Leftrightarrow \mathcal{T}\}$.

SAMRAI's simple box-coalescing algorithm is implemented in the `BoxContainer` class method `coalesce`. It performs an exhaustive search for pairs of coalescible boxes. After each successful coalescing, the search restarts from the beginning. The algorithm has cubic complexity in the number of boxes being coalesced. This could take too long even when we limited the input to local clusters in $\mathcal{C}$, because many fixed-size tiles were required to cover the tagged regions in our benchmarks. We accelerated the coalescing step by sorting potentially coalescible boxes into a tree structure to limit the number of boxes subjected to the simple coalesce algorithm.

The new coalescing method is shown in Algorithm 1. The simple coalesce algorithm was used if there were sufficiently few boxes (line 1). Otherwise, we moved each box into one of two temporary containers. If the box had the majority of its volume in the lower half of the bounding box of all boxes in the original container, it went in the "lower" container; otherwise, it went into the upper container (line 8). The temporary containers were recursively coalesced then recombined. In the recombination step, boxes that were in one temporary container but touched the other container were first placed into a third temporary container, where they were coalesced among themselves before joining the other boxes. In this way, only boxes that had a chance of being coalesced are checked. Although there were still potentially coalescible boxes after recombining, we did not check for them. Our goal was speed, not perfection.

The if-block at line 9 prevented infinite recursion caused by rare cases when all boxes went into one container, leaving the other empty. We tried to reduce the container size by selecting some boxes to go into the empty container. If that did not work, we simply fell back on the simple coalesce algorithm.

Coalescing might arguably slow the solver by

Algorithm 1 COALESCETILES(`BoxContainer s`): Coalesce boxes in `s` using a tree structure to organize boxes into smaller groups for coalescing.

```
1:  if s.size() < $s_{\min}$ then
2:    s.coalesce() {Simple coalesce algorithm}
3:    return
4:  end if
5:  bb ← s.boundingBox() {bounding box for all
    boxes in container.}
6:  splitdir ← direction of longest side of bb
7:  BoxContainer lower, upper
8:  Split s into lower and upper, where lower con-
    tains boxes with more volume is in the lower
    half of bb along direction splitdir, and upper
    contains the rest of the boxes.
9:  if lower.empty() or upper.empty() then
10:   Move from the non-empty container to the
      empty container all boxes that crosses the
      mid-plane of bb
11:   if lower.empty() or upper.empty() then
12:     s.coalesce()
13:     return
14:   end if
15: end if
16: coalesceTiles(lower)
17: coalesceTiles(upper)
18: BoxContainer t
19: Move from upper and lower into t all boxes
    that touch the other container's bounding box.
20: t.coalesce() {Simple coalesce algorithm}
21: s.clear() {Clear out container.}
22: Move contents of upper, lower and t to s
23: return;
```

making patch data too big to fit into cache. To make sure they do fit, the boxes could be broken down to the right sizes after partitioning in a fast step involving breaking up local boxes, building a map, and applying a modify operation. This systematic approach would limit metadata size for as long as possible and reliably ensure all patches would be correctly sized. We did not explore cache efficiency because until this point, the primary problem had been mesh-management cost.

6 Results and discussion

We evaluated our algorithms using two performance test codes in SAMRAI, one for linear advection and one for Euler hydrodynamics. Both integrated conservation equations in the volume integral form,

$$\frac{\partial}{\partial t}\int_{\Omega} U d\Omega + \oint_{\partial\Omega} \mathbf{F}\cdot d\mathbf{S} = 0 \tag{4}$$

applied to the finite volume Ω with closed boundary $\partial\Omega$, where U is the conserved quantities and F is the corresponding flux. The differential area vector $d\mathbf{S}$ points outward on S.

The first performance test integrated the linear advection equation to move a scalar quantity u in a uniform constant velocity field $\mathbf{a}$. Thus, $U = u$ and $\mathbf{F} = \mathbf{a}u$ in equation (4). This test used the “LinAdv” performance code in SAMRAI.

The second performance test integrated the Euler equations for compressible hydrodynamics, with

$$U = \begin{pmatrix} \rho \\ \rho\mathbf{v} \\ \rho E \end{pmatrix} \tag{5}$$

where ρ is density, $\mathbf{v}$ is velocity and E is total energy. The nonlinear flux vector

$$\mathbf{F} = \begin{pmatrix} \rho\mathbf{v} \\ \rho\mathbf{v}\otimes\mathbf{v} + p\hat{S} \\ \rho E\mathbf{v} \end{pmatrix} \tag{6}$$

is composed of transport of mass, momentum and energy by velocity $\mathbf{v}$ and by pressure $p = (\gamma - 1)(\rho E - \rho|\mathbf{v}|^2)$ acting on $d\mathbf{S}$. $\gamma = 1.4$ is the specific heat ratio. This test used the “Euler” performance code in SAMRAI.

The LinAdv and Euler performance tests followed the Berger-Colella scheme [BC89] for integrating on the mesh hierarchy. This scheme integrates one level at a time, employing any level integrator suitable for single-level (non-AMR) grids. It links the level solutions together with two types of inter-level operations. To maintain conservation at coarse-fine mesh boundaries, it matches the flux on the coarse grid with that on the fine. Where coarse and fine solutions overlap, it synchronizes the coarser solution by replacing it with a coarsening (averaging) of the fine-level solution.

The level integrator used a finite-volume discretization of equation (4), storing the volume-averaged U in each cell. To get $\mathbf{F}$ on $d\Omega$, we used a Godunov scheme, computing the state on both sides of $d\Omega$ and using a Riemann solver to compute the state on $d\Omega$. For the advection equation, the state took on the upwind value. For the Euler equation, we used the Colella-Glaz approximate Riemann solver [CG85] to compute the state on $d\Omega$. We also used variants of Colella’s corner transport upwind scheme [Col90], which accounted for multi-dimensional effects at corners in $\partial\Omega$.

The levels were advanced in time using the explicit forward-Euler time-stepping scheme. Advancing the entire mesh used the time-refinement scheme of [Tra95, HK02], in which a finer level takes multiple but smaller time steps for each time step on a coarser level. The solution is synchronized at the end of the coarse level time step, when the solution time is the same for both coarse and fine levels.

The physics and numerics of linear advection were very simple. They serve as a familiar and well-understood point of reference with which to compare regridding costs. They make the benchmark a good stress test for regridding performance, because they provide little computation in which to hide the cost of regridding. In contrast, the Euler benchmark was had more computation, though it was still simple compared to current multi-physics simulations and high-order methods.

All performance tests were run on Lawrence Livermore National Laboratory’s Sequoia BGQ machine. Sequoia had 96K (98,304) compute nodes. Each node had 16 GB of memory shared among 16 CPU cores. The CPUs were PPC A2 chips running at 1.6 GHz. Each core had 4 hardware threads.

The benchmarks and SAMRAI code were not

threaded, but multiple threads could be used to run multiple MPI tasks. We typically saw a modest gain in numerical integration time at 2 tasks/core but lost it at higher thread use. The communication-dominated regridding components benefited little. The weak-scaling benchmarks in this work used 2 task/core, but the strong-scaling benchmarks used 1 task/core.

We ran the cascade partitioner with a spreading limit of $g = 500$, in equation (3) and weight tolerance factor $\chi = 0.05$ (Section 4.4). For the recursion limit in tile coalescing in Algorithm 1, we used $s_{\min} = 20$.

6.1 Weak-scaling linear advection results

Our first benchmark was the advection of sinusoidal wavy walls. Figure 6 shows a sample of the mesh from this benchmark. Without the waviness, the walls would be perpendicular to the x-axis. The exact x-coordinate of the nth wall was given by

$$\begin{aligned} x_w(y,z) &= x_0 + a_x t + n\lambda \\ &+ A\cos\left[k_y(y + y_0 - a_y t)\right]\cos\left[k_z(z + z_0 - a_z t)\right] \end{aligned} \tag{7}$$

where (x_0, y_0, z_0) were initial displacements of the walls, (a_x, a_y, a_z) were components of advection velocity $\mathbf{a}$, (k_y, k_z) were wave numbers of the sinusoidal function in the y- and z-directions, and λ was the distance between consecutive walls in the x-direction. The exact solution $u(x, y, z, t)$ was constant between walls and jumped by a value of one across each wall. We used physical parameters of $\lambda = 8$, $k_y = k_z = \frac{2\pi}{8}$, $\mathbf{a} = (2, .55, .55)$, $A = \frac{1}{2}$, and $(x_0, y_0, z_0) = (3, 0, 0)$.

Level 0 had mesh size of 24x3x3 cells/core, covering a physical region from the origin to (8,1,1). We increased work by expanding the computational domain. We grew it in the y- and z-directions as core count grew from 1 to 64. At 64 cores, the domain was 8x8x8, big enough to fit a whole period of the wavy wall. After this point, we increased the domain in the x-, y- and z-directions, one at a time. Each time we doubled the domain in the x-direction, the number of walls doubled. At 1M cores, there were 32 walls. The refinement ratios were 3 in each coordinate direction. We used tile sizes of 3x3x3 on the coarsest level and 9x9x9 on other levels. We ran the simulation with a Courant number of 0.43 for six Level-0 time steps. We regridded each level (except level 0) after every 9 time steps on that level.

Weak-scaling performance studies require precise control of the number of cells in the mesh. This turned out to be difficult because the number of cells could not be specified directly (except on level 0). Mesh size could only be indirectly controlled by tuning error estimates and thresholds. For the needs of this study, the mesh size was too sensitive to the error threshold, and it was virtually impossible to find a single threshold value to give the mesh both the size and depth the study required. Moreover, the region tagged using the error estimate tended to grow with solution time due to numerical diffusion of the discontinuity.

To avoid some of the difficulties in mesh size control, it was necessary to tag cells using the exact solution. We tagged cells that had any node within a "buffer distance" of the exact location of a wall, in any coordinate direction. The buffer distances were 0.15, 0.045, 0.015, 0.005, and 0.0015 when tagging on levels 0 through 4, respectively. The box configuration in Figure 6 was generated with these buffer distances. Although this controlled the tags directly, the mesh was still indirectly controlled, so its size varied slightly. Actual mesh sizes from the benchmark runs are given at the top of Figure 7. Higher resolution allowed more exacting refinement leading to smaller variations on finer levels. We chose the midpoints between the minimums and maximums as the representative sizes. The cell counts came from the 5-level run, but they were similar for other hierarchy depths. Cells-per-core was $22.2{\times}10^3$, $195{\times}10^3$, and $1.640{\times}10^6$ for the 3-, 4-, and 5-level meshes respectively.

We scaled the Wall benchmark from 1 to 1M (1,048,576) cores, with 2 MPI tasks/core. The three timing plots in the middle of Figure 7 show overall performance on 3-, 4- and 5-level meshes. Raw timer values were the maximum timer value across all processes. To remove the effects of mesh-size variations, we normalized the times by the number of cell updates performed. Major components contributing to the total time are shown in the three timing plots in the middle of the figure. "Numerical Kernels" refers to the time spent executing the sequential numerical kernels on structured grids (patches). "Advance Levels" includes the kernels plus filling physical and coarse-fine

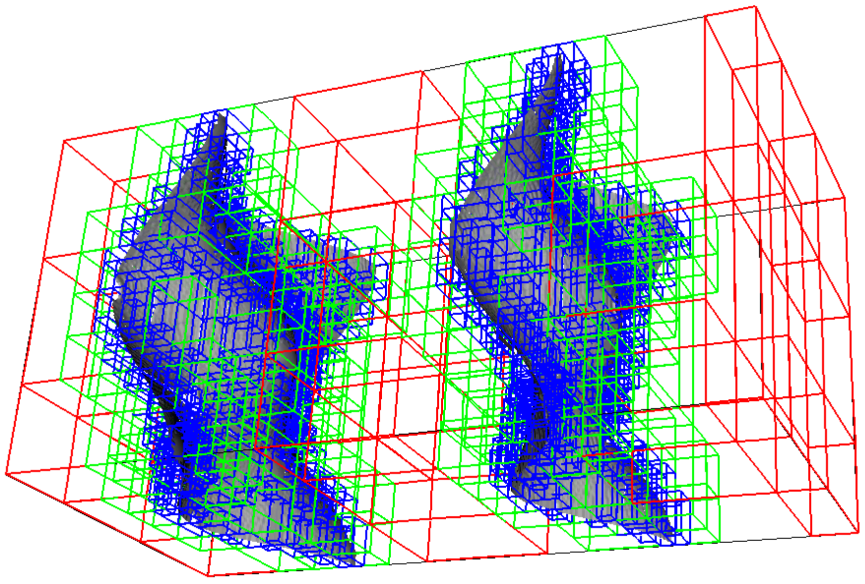


Figure 6: Wall configuration with three levels at 128-cores, showing two wavy walls. Without the waviness, the walls would be perpendicular to the x-direction. Patch outlines are red, green and blue for levels 0, 1 and 2, respectively. To increase the problem size, we doubled the domain by in the y-, z-then x-direction, rotating through them in order. Each time the domain doubled in the x-direction, the number of walls doubled.

boundaries, matching flux at coarse-fine boundaries and computing stable global time steps on each level. We will use this as the static solver cost (the total cost had there been no regridding). Technically, static solver cost should also include level synchronization, but level advance cost made a good approximation because synchronization cost tended to be very low. "Regrid" is the total cost of regridding, including steps in Figure 3, populating mesh data and recomputing data transfer schedules for the new hierarchy configuration. Total cost includes advancing levels, regridding, and synchronizing levels.

The numerical kernel code was sequential, so its time was unaffected by communication performance. The jump occurring between 8 and 16 cores in the timing plots were likely due to using threads to run MPI tasks. Though we specified 16 MPI tasks when running the problem on 8 cores, the 16-core runs were the first in each scaling sequence where each cores actually had to run two MPI tasks, changing the timing trend at that point. (This jump did not appear when we ran with 1 task/core.) From 16 cores on, Numerical Kernels times were very steady. It was nearly 2 microseconds per cell update on the 3-level mesh and 1.5 microseconds per cell updates on the two bigger meshes.

The Advance Level curves dipped slightly between 1 and 4 cores, jumped as the numerical kernels did at 16 cores, then dropped again between 32 and 64 cores. The dipping behavior was caused primarily by the exact boundary conditions. They used sine functions, which turned out be much more expensive when compared with the numerically simple cell updates. Below 32 cores, all physical boundaries were exact. At 32 cores, the j-boundaries switched from exact to periodic and stopped using sine functions. At 64 cores, the k-boundaries did the same. The i-boundaries were always exact because the solution was not periodic in i, but only level 0 touched these boundaries, so the time spent on them was not discernible in the plots. Surface-to-volume ratio of the computational domain also contributed to the subtle dip. The domain was a slender rod at one core and had a high surface-to-volume ratio, amplifying boundary condition costs. As it grew, the high aspect ratio and surface-to-volume ratio went away.

After early transient changes, the Advance Level curve started a slightly rising but steady, non-accelerating trend. The last two plots in Figure 7 show parallel speed-up and efficiency of the Advance Level step and the overall run. Both were referenced to the 64-core run to avoid the early transients. Overall efficiency at 1M cores was still high: 72%, 79% and 84%, depending on the amount of work per core. Without gridding, the overall time would be close to the Advance Level time, which had efficiency of 74%, 85% and 88%. At 1M cores, the efficiency loss rate was between 3% and 5% every 10X increase in core count and showed no sign of accelerating.

Adding levels increased floating point computation as well as regridding cost because of the additional levels and boxes. However, regridding cost rose more slowly, leading to higher parallel efficiency for meshes with more levels. At 1M cores, the ratio of regridding cost to level advance cost was 1.35 for the 3-level configuration. It was about $\frac{1}{2}$ for 4 levels and about $\frac{1}{4}$ for 6 levels.

Complex and communication-intensive, regridding naturally scaled worse than integrating the solver. However, Figure 7 shows that it remained a manageable overhead cost when there was reasonable computational work. Even with little computation, regridding cost did not blow up or exhibit the $O(N)$ scaling characteristic of handling globalized data. Figure 8a shows the major mesh-management timings in detail for the 4-level Wall benchmark. The top plot shows the cost of regridding components. Though we had not discussed all operations shown in the plot, it was important to reveal their trends to ensure no poorly scaling components would dominate as we scaled up. Even fast components would eventually do this if their slope on the log-log plot was too high. The "Compute Boxes" time includes everything required to compute boxes that form the new levels and connect them to the tag level.

The initial rise of nearly all curves in regridding was typical for operations dominated by peer-to-peer communication, as the number of possible peers increased. This increase should not scale with MPI task count, and the leveling out of the curves shows that it did not. Clear and steady trends in Figures 7 and 8 suggest that scaling trends could continue well past 10M cores.

Partitioning was the worst scaling component in regridding seen in Figure 8a, likely due to its $O(\lg^2 N)$ complexity, but its slope was mild. At 1M cores, it amounted to only 12% of the cost of computing boxes, 5% of regridding cost and 2% of the total time in simulations with only 195×10^3 unknowns per core. Clustering was three times faster and rose even more slowly, with a curve shape typical of peer-to-peer communications. The Bridge and Modify curves represented all the bridge and modify operations in Figure 3, except for those within clustering and partitioning. Both had characteristics associated with peer-to-peer communication, but bridging was significantly more expensive. The number of bridge and modify operations was roughly the same so the difference between the two was due to individual operation times.

Figure 8b shows clustering component costs for the 4-level Wall configuration. Coalescing scaled exceptionally well and adjusting $\{\mathcal{C} \Leftrightarrow \mathcal{T}\}$ for the change was even faster. The higher coalesce-adjustment cost starting at 64 cores was likely due to the extra work the modify operation had to do to find overlap relationships across periodic domain boundaries. The most expensive clustering component was increasing the width w of the $\{\mathcal{C} \overset{w}{\Longleftrightarrow} \mathcal{T}\}$ `Connector`. The algorithm we described in Section 5 only computed proximity for a zero width. SAMRAI's gridding algorithm and communication schedules require a finite width, which was computed from operations anticipated during and after regridding. To increase w to the correct value, we did another bridge operation. Increasing width actually happened outside of clustering, so it was not a part of the Total Clustering time.

Figure 8c shows the cascade partitioner component for the 4-level Wall configuration. The most expensive component was using the change maps to update $\{\mathcal{R} \Leftrightarrow \mathcal{P}\}$. The fastest-growing cost was the $O(\lg^2 N)$ operation to combine children groups to get weights for the next-bigger group, but at 1M cores, it was only half the cost of using the map and about 1.5% of total regridding cost.

The 3-level result may be roughly compared to our former wall propagation results in [Gun12], if we account for some differences first. The current

Mesh size for Wall benchmark

Level number	$\frac{h_0}{h}$	Cells per core: on level	Cells per core: incl. coarser
0	1	216	216
1	3	2,267 ± 3.5%	2,483
2	9	19,848 ± 6.3%	22.2×10^3
3	81	172,766 ± 5.9%	195×10^3
4	243	1,445,150 ± 0.6%	1.640×10^6

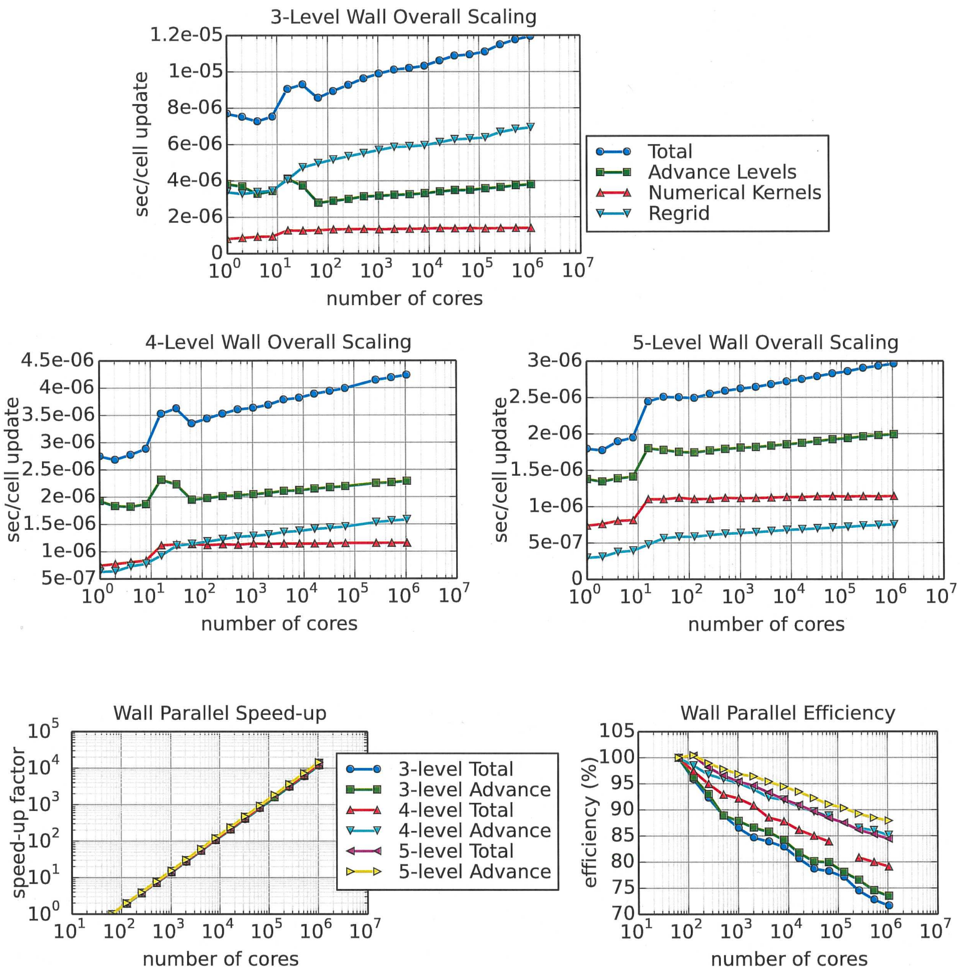
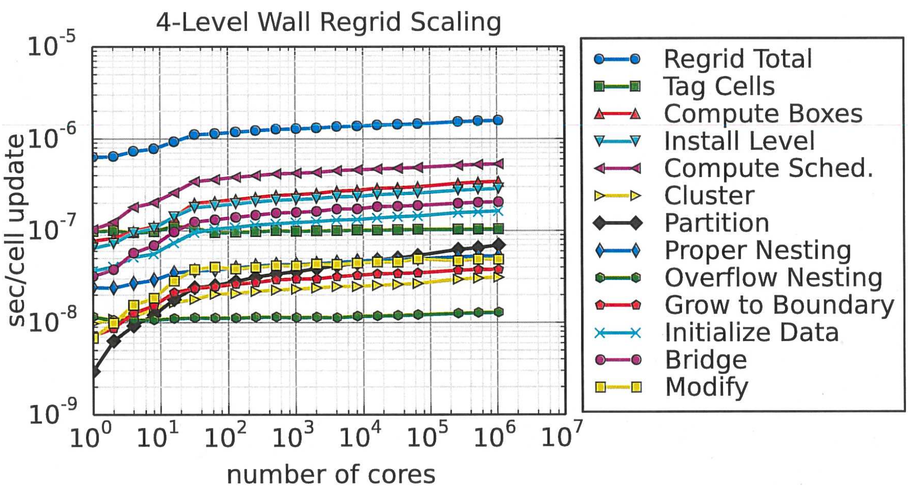
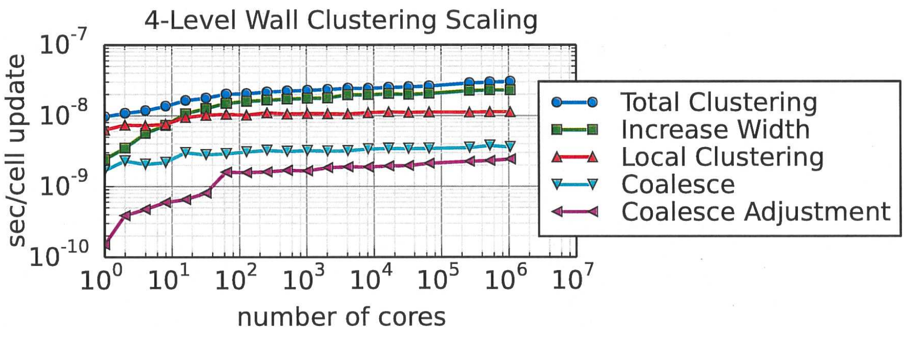


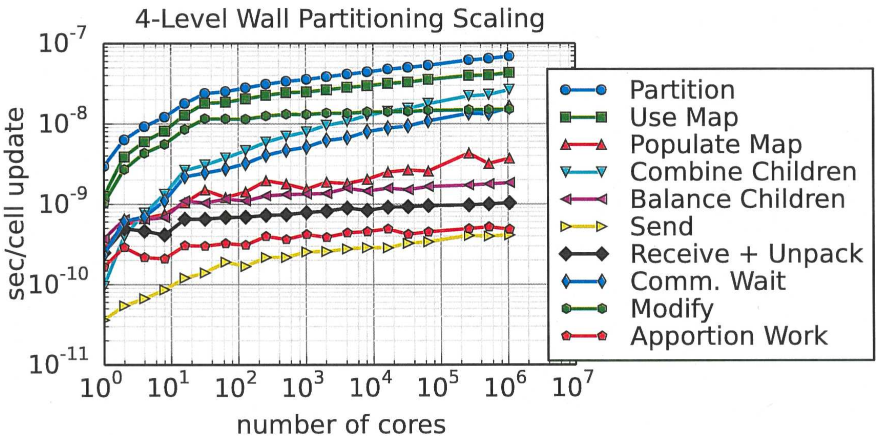
Figure 7: Wall benchmark mesh size and weak-scaling results: Full mesh size for a given depth includes finest level and all coarser levels. Most cells are on the finest level. The three timing plots use the first legend and give results for hierarchy depths of 3, 4, and 5 levels. Parallel speed-up and efficiency use the second legend. They are in reference to the 64-core run, to avoid the initial transient behavior caused by expensive exact boundary conditions.



(a) Regridding component times.



(b) Clustering component times.



(c) Partitioning component times.

Figure 8: Weak-scaling Wall benchmark mesh-management scaling.

work used a refinement ratios of 3, regrid interval of 9 time steps and averaged 22.2K cells/core. The former results used a refinement ratio of 2, regrid interval of 4 time steps and averaged 27.3K cell/core. The current normalized regridding time, if multiplied by $\frac{9}{4} \times \frac{22.2}{27.3} \simeq 1.83$, may be directly compared to the former regridding time. (The older benchmark was run on a smaller BGQ machine and there had been system upgrades, but these were likely to have only a secondary affect on performance.) Figure 9 shows the current and former regridding time plots. The table at the bottom of the figure shows the hypothetical time reduction had we used the new algorithms. All measures showed significant improvement, and clustering time practically disappeared. However, the most important difference was that while old regridding time began to increase linearly after 64K cores, the current result maintained its steady trend, making it usable for much larger scales.

6.2 Strong-scaling linear advection results

The Sphere advection test was the first of two strong-scaling benchmarks. We used the spherical feature because of its contrast to the wavy wall. While the walls were thin features that spread out cross the mesh, the sphere was thicker and compact. The sphere had an inner radius of 0.9 and outer radius of 1.1, propagating through a cubic domain with corners at (0,0,0) and (16,16,16) in physical space. The initial center of the sphere was (6,6,6) in physical space, and the propagation velocity $\mathbf{a} = (1.0, 0.5, 0.5)$. The sphere was sufficiently far from domain boundaries so only level 0 touched the physical boundaries and exact boundary conditions were not required. We tagged cells that were within certain distances of the sphere, either inside or outside of it, in any coordinate direction. The distances were 0.07, 0.03125, 0.00625, 0.00125, 0.00025, and 0.00005 for tagging on levels 0 through 5, respectively. The hierarchies were 3 to 6 levels deep, with refinement ratios of 4, giving a mesh spacing ratio of 1024 from coarsest to finest. Each level took 4 time steps for every time step taken on the next-coarser level. A level was adapted once every 8 time steps, except for the static level 0.

Strong-scaling did not require a sequence of meshes with specific sizes, so we did not use domain expansion. We set up level 0 with a 32x32x32 grid and increased mesh sizes by adding levels.

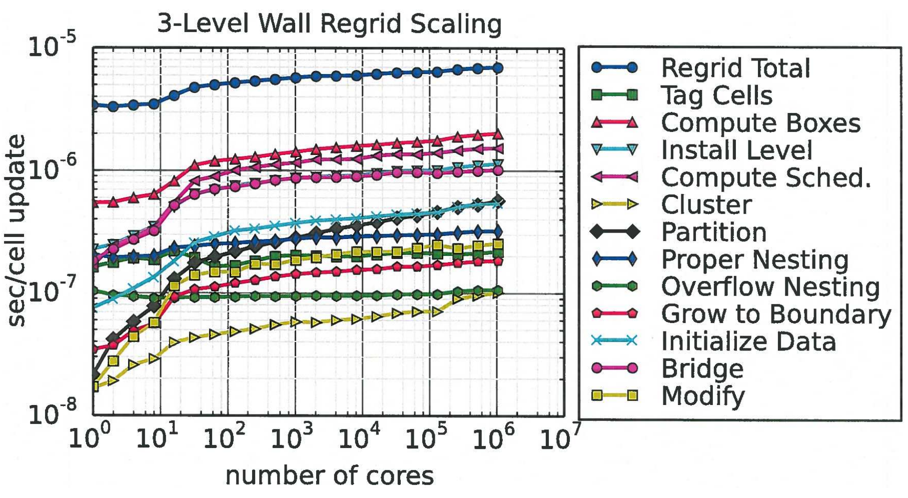
The table at the top of Figure 10 shows representative mesh size on each level for the Sphere benchmark. Mesh-size variations were even smaller for this benchmark than for the Wall benchmark. We used the midpoints between minimum and maximum values as the representative sizes. Global mesh size ranged from 971×10^3 cells for the 3-level mesh to 24.5×10^9 cells for the 6-level mesh, covering a range of 4 orders of magnitude.

We ran this benchmark up to 1.5M cores. The four timing plots in Figure 10 show overall scaling performance. The left sides of these plots show that regridding the Sphere benchmark took as little as 5% to 10% of the total run time. Regridding lost speed-up well before the solver did. (The regridding curves leveled out earlier.) Eventually, regridding costs overtook non-regridding costs, but the point at which the curves cross was further out the bigger the mesh grew.

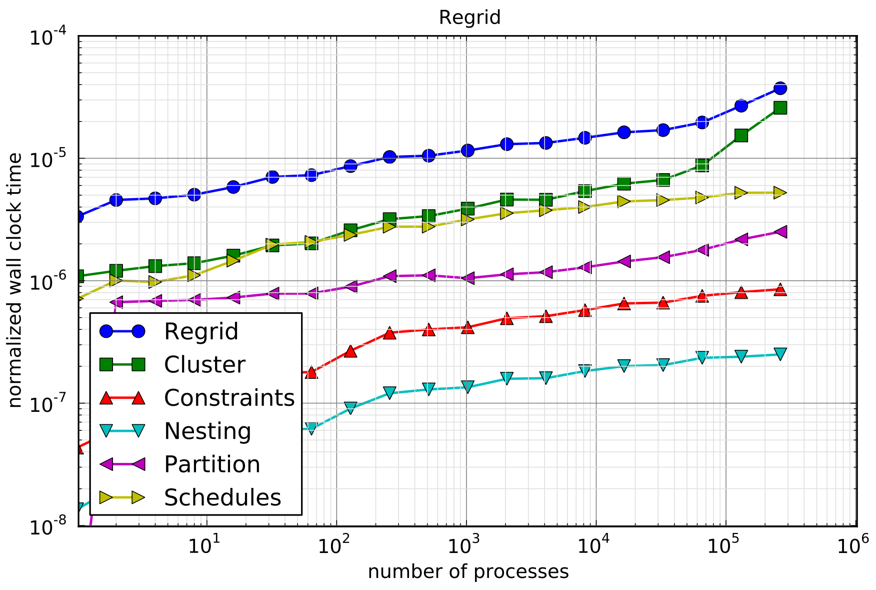
The sequential numerical kernel was the best scaling component, as expected. It stopped speeding up and abruptly leveled out when there was not enough work to occupy every MPI task, on the 3- and 4-level meshes. When this happened, the other major timers also leveled out, because metadata also leveled out. However, on the 5- and 6-level cases where the mesh continued to be divided into smaller parts to use the additional cores, metadata continued to grow, and regridding eventually reversed direction. It is important to note that regridding remained relatively flat even after reversing directions. This indicates that simple changes such as reducing regrid frequency could extend the scalable range.

The last two plots in Figure 10 show parallel speed-up and efficiency the Sphere benchmark. For each mesh, we chose as reference the smallest core count that the mesh could run on without running out of memory. An exception was for the 4-level mesh, which ran down to 1 core, but that did not reflect the available memory because the core had access to the node's entire 16GB of memory. We chose the 4-core run because it had cell/core count in line with the reference runs for the bigger meshes. So the reference runs were on 1, 4, 128 and 8K cores for the 3-, 4-, 5-, and 6-level meshes, respectively.

The 3-level mesh, with only a quarter of the work per core of the deeper meshes on reference runs, lost



(a) Three-level Wall regridding time.



(b) Regridding time plot from [Gun12], which used a similar 3-Level wall benchmark.

	Regrid		Clustering		Partitioning	
Number of cores	64K	256K	64K	256K	64K	256K
Time reduction	43%	68%	99%	100%	58%	62%

(c) Hypothetical regridding component time reductions by new algorithms at 64K and 256K cores (after incorporating the 1.83 factor).

Figure 9: Regridding time comparison between 3-Level wall benchmarks in this work vs earlier work [Gun12]. Times in plot (a) should be multiplied by 1.83 to compare to times in (b).

efficiency fastest. Its overall speed-up was about 11. The other three meshes, with 3-4 million cells/core in the reference run, managed overall speed-up in the range of 40-60. The solver saw nearly an order of magnitude more speed-up.

The 6-level run exhibited an unexpected super-linear speed-up between 8K and 64K cores, visible in the efficiency plot and also discernible in the 6-level timing plot. The 5-level run also experienced super-linear scaling, but to a smaller degree. This behavior started in the numerical kernels, so it was not likely due to communication issues. It could be due to patches shrinking enough to fit into cache.

The point at which a problem fails strong-scaling is difficult to define and is a stronger function of problem size than efficient algorithms. For a more precise way to compare regridding strong-scalability, we look at the ratio of regridding overhead cost to the more scalable, predictable and better understood sequential computation cost. We quantify regrid scalability using the work per core at the point where they are equal. The intersection of the Regrid and Numerical Kernels curves for the 3-, 4-, 5-, and 6-level meshes occurred around 18, 280, 6,400, and 270,000 cores, corresponding to 54, 58, 80, and 91 thousand cells/core. This parameter could be used to compare results across the four orders of magnitude in problem size. Moreover, it was unaffected by the super-linear speed-up and reference choice, unlike speed-up and efficiency. The parameter revealed something interesting. The smaller meshes may have stopped scaling orders of magnitude earlier, but they actually scaled to a point of less work per core before regridding overhead became significant. The degraded values for the larger problems might reflect the overhead of using more tasks. Because these problems all used the same computational kernels, computational work could not be the reason for the difference. However, the Euler benchmark does have more computational work, allowing it to run more efficiently at the higher task counts.

Figure 11a shows the breakdown of regridding for the 4-level Sphere benchmark. Partitioning took less than 10% of regridding time, and clustering took less than 1%. Most components did speed up between 1 and 128 cores but had no significant gains beyond that. Partitioning and enforcing proper nesting times did not shrink much before growing, but they also started out very small. At the high end, bridging took the longest, twice as long as all the steps in computing boxes. As with the weak-scaling case, bridging was significantly more expensive than modifying. No individual component dominated the timing or threatened to do so.

Figure 11b shows the breakdown of the partitioner timings. Although little partitioning speed was gained with more cores, the speed loss at the higher end was mild. No component dominated the partitioning cost or threatened to do so.

Figure 11c shows the breakdown of the clustering timings. Local clustering and coalescing were local operations, so they scaled well until they ran out of work and leveled out abruptly. Increasing width and adjusting for coalesced tiles both required communication, but increasing width was significantly slower and scaled less well. The main difference might be because increasing width used a bridge while coalesce adjustment used a modify operation.

6.3 Strong scaling Euler Results

We remarked that there was little numerical computation in the linear advection benchmarks, making them good stress tests for regridding performance. With an Euler benchmark we could contrast the physically simple linear advection with more complex physics of hydrodynamics. This benchmark used the same spatial discretization and time stepping scheme, but it had 5 unknowns per cell and more complex, non-linear physics.

The simulation was for an expanding spherical shock wave. The physical domain, a box from (0,0,0) to (2,2,2), was initialized with a high-pressure spherical region centered at one corner. The sphere radius was 1. Pressure inside the sphere was 1140.35×10^5 times the outside pressure. Otherwise, $\rho = 1$ and $\mathbf{u} = (0, 0, 0)$ everywhere initially.

The Euler benchmark used 4 levels, with 244x244x244 cells on Level 0. Refinement ratio was 4, and tile size was 8x8x8. The domain boundary did not match tile boundaries, but our implementation did not require it to. We ran the problem at a Courant number of 0.9 for two Level-0 time steps (equivalent to 128 steps on the finest level). Each level was adapted after every 8 steps (except level 0).

Figure 12 shows the mesh sizes and overall scaling results of the Euler benchmark. The mesh grew

Mesh size for Sphere Benchmark

Level number	$\frac{h_0}{h}$	Global cell count: on level		Global cell count: incl. coarser
0	1	32768		32768
1	4	131072		163840
2	16	806912	$\pm$ 5.6%	970752
3	64	15.325×10^6	$\pm$ 0.15%	16.30×10^6
4	256	492.617×10^6	$\pm$ 0.10%	508.9×10^6
5	1024	24.0931×10^9	$\pm$ 0.0055%	24.60×10^9

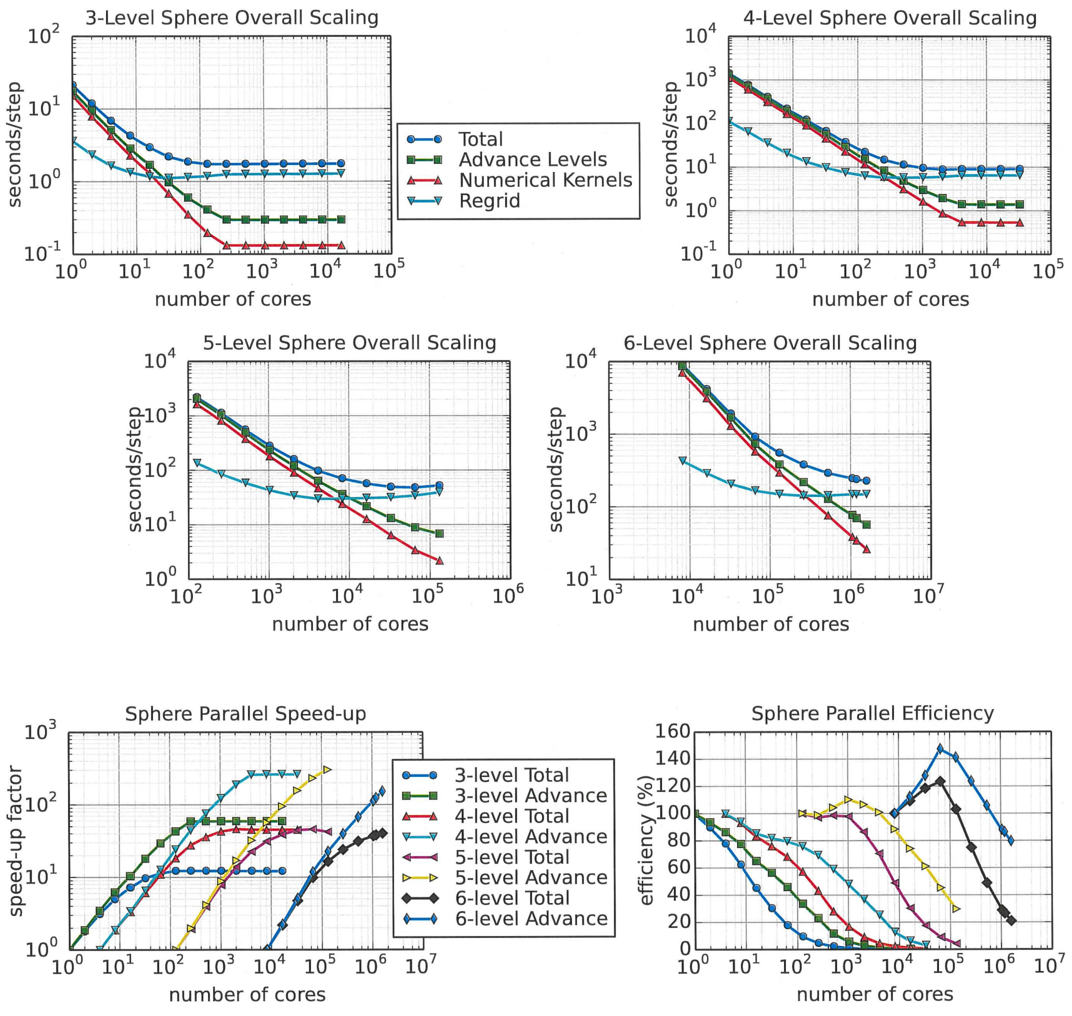
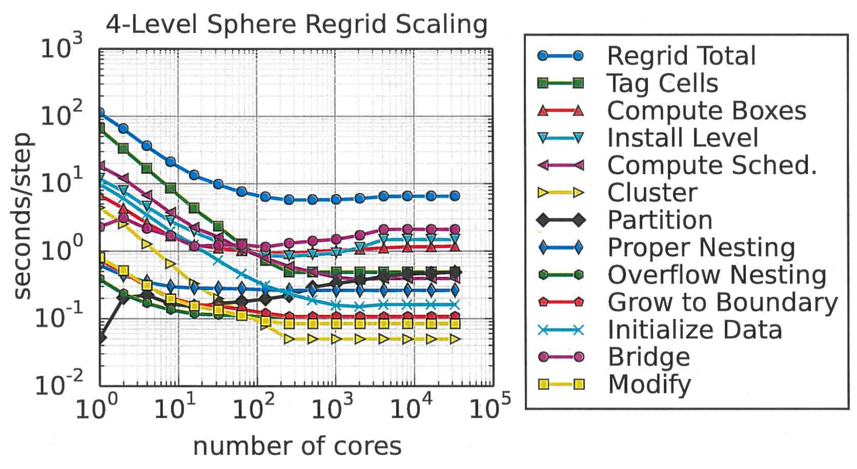
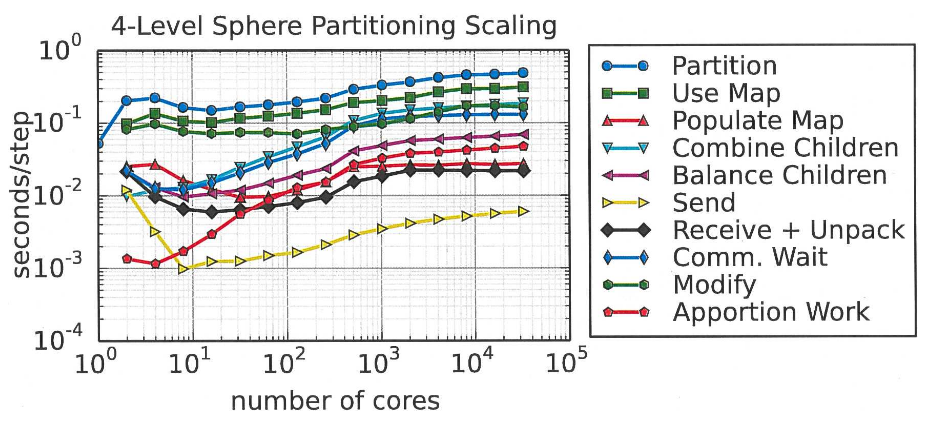


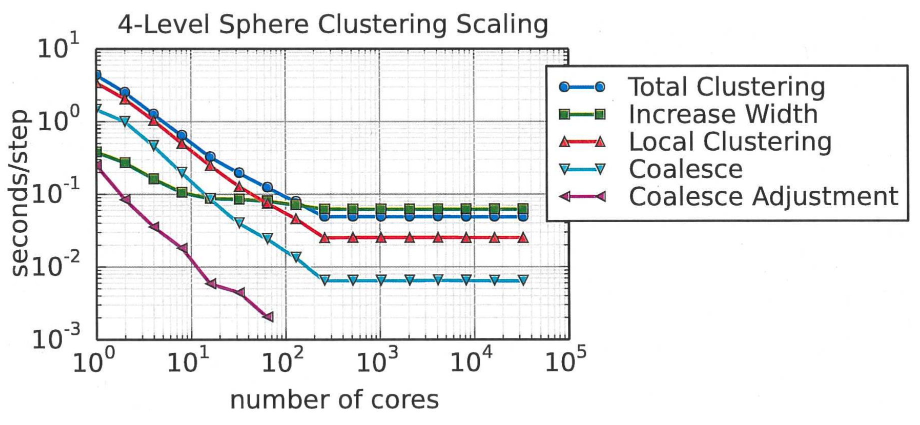
Figure 10: Sphere benchmark mesh size and strong-scaling results: See Figure 7 for explanation. Parallel speed-up and efficiency are referenced to 1 core for the 3-level mesh, 4 cores for the 4-level, 128 cores for the 5-level, and 8K cores for the 6-level.



(a) Regridding component times.



(b) Clustering component times.



(c) Partitioning component times.

Figure 11: Scaling of regrid in Sphere problem.

significantly due to the growing shock wave and expansion fan. It started with 4.31×10^9 cells and doubled to 8.74×10^9, almost all on the finest level.

The last two plots in Figure 12 show speed-up and efficiency for the Euler benchmark. The speed-up factor was 60 and still climbing at 1.5M cores, though efficiency dropped to 32%. The non-regridding portion had sped up more than 100 times and an efficiency of 56%.

Regridding cost overtook numerical kernel cost around the last data point, 1.57×10^6 cores, corresponding to mesh size of 2.6-5.3 thousand cells/core, or 13-27 thousand unknowns/core. Comparing this with the 54-91 thousand unknowns/core for the Sphere benchmark shows that the Euler benchmark not only overcame the overheads of using on more tasks, it scaled to another factor of four reduction in work per core. We believe this provides a concrete measurement for comparing strong-scaling efficiency. Note that it is valid to compare these two benchmarks because both had equally good load balancing (see Section 6.5) and the same time integrator and regridding frequency.

Figure 13 shows the breakdown of regridding for the 4-level Euler benchmark. At 1.5M cores, partitioning took less than 10% of regridding time and clustering took less than 2%.

6.4 Choices affecting relative regridding cost

Regridding is an overhead that is commonly viewed in terms of what fraction of the overall time it took. Though relative regridding cost is a very useful metric, by itself, it does not describe a complete picture. Choices we made could easily have a significant effect on this metric.

We showed above that increasing mesh size and depth decreased relative regridding cost for the weak-scaling benchmark, even though mesh management and computational work both increased. Using higher-order discretizations, non-linear discretizations, multi-stage integrators, or solving more complex physics all would increase computation without a corresponding increase in meta-data, resulting in lowering the relative regridding cost. Even poor load balance and data locality could help hide regridding cost by increasing the integrator cost.

Changing regridding frequency would have an immediate proportional effect on total regridding time. This choice is usually accompanied by growing finer levels in space to ensure important features would remain well resolved until the next regrid, creating more computational work that further reduces relative regridding cost but possibly increases total cost. Similarly, using large tiles would reduce mesh-management cost, but leads to more unnecessary refinement, possibly resulting in more cost.

We believe that the benchmarks we chose were challenging for mesh-management algorithms, in that there was a lot of regridding and little computation to hide that cost. Most problems in practice would have significantly more complex physics and computations to hide mesh-management costs. On the other hand, the BGQ computer, on which we ran the benchmark, had a high ratio of communication speed to processor speed. On another computer, the relative cost of communication-intensive regridding could be higher.

6.5 Load balance

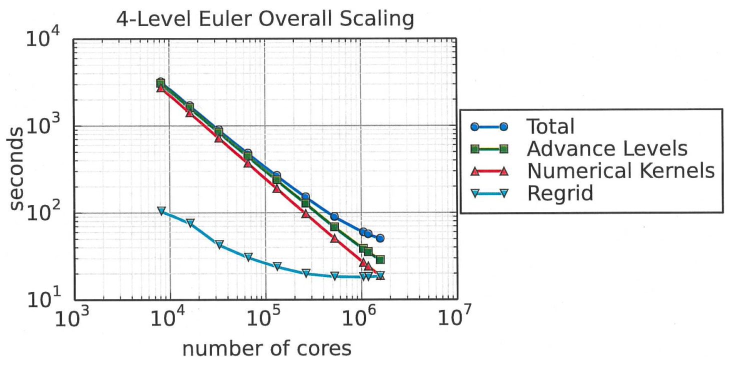
Load balance and data locality are qualities that can help a solver run faster. In SAMRAI, they are the result of the clustering and partitioning operations. This subsection examines how effectively the new algorithms balance loads. Section 6.6 examines data locality.

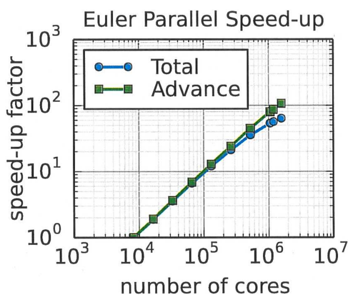
Table 1 shows cell and box counts per task for the weak-scaling Wall benchmark. Max and average values refer to the maximum and average over all MPI tasks. The $\frac{\text{max}}{\text{avg}}$ ratio is the key parameter for evaluating the degree of overload affecting the slowest process and consequently the whole. The partitioner managed to keep the max overload to about 10% where possible (on levels 2-4). On level 0, the max was 9 times the average, but this figure is misleading. The max load was only 846 cells more than the average. Similarly, on level 1, the max load was 33% more than the average, but the overload was less than a single tile. Overload that amounted to small actual work surplus did not have a significant effect on speed. In our discussion, we will disregard overloads caused by lack of work rather than poor balancing.

Although box count is of secondary importance compared with cell count, it is important to keep it low. Excessive box counts slow the search for data

Mesh size for Euler benchmark

Level number	$\frac{h_0}{h}$	Global cells ($\times 10^6$)	
		initial	final
0	1	14.53	14.53
1	4	15.72	15.72
2	16	245.1	284.7
3	64	4032	8099
total		4037	8413





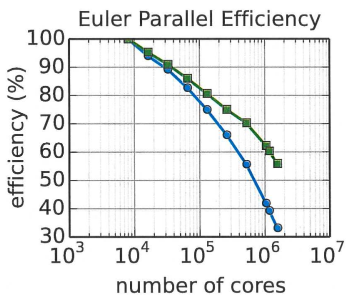


Figure 12: Euler benchmark mesh size and strong-scaling results. See Figure 7 for explanation. Parallel speed-up and efficiency are referenced to 8K cores.

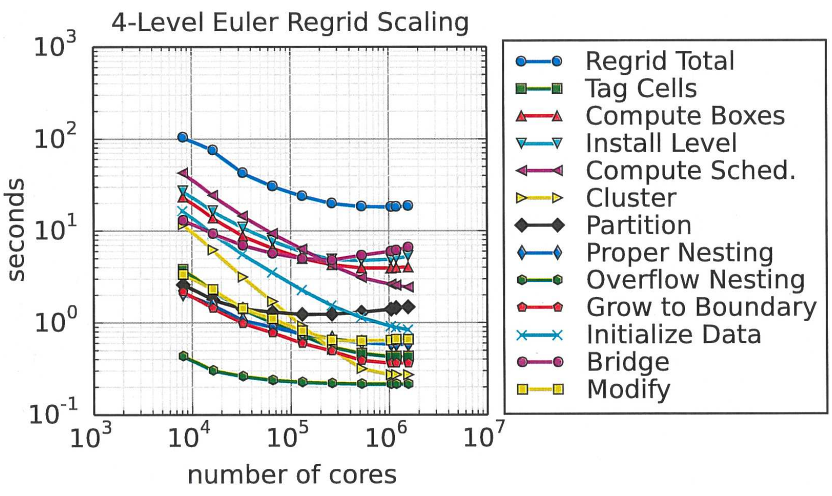


Figure 13: Regrid of strong-scaling Euler Problem.

dependency and create too many places where we have to transfer data. The right side of Table 1 shows box-count statistics. As we noted, Level 0 was short on work. It averaged 0.15 boxes per process. The $\frac{\max}{\text{avg}}$ ratio of 6.67 was high, but again, this was due to lack of work, and we will disregard these cases. Average box counts for levels 1-3 were 1.02, 3.63, and 20.98 boxes/process, growing with work amount but otherwise reasonable. Level 4 averaged 160.6 boxes. While this seems large, it came from discretizing the expansive, non-grid-aligned walls at high resolution. The box count was actually very low compared to the nearly 1,000 boxes it would have taken had we not coalesced the tiles. Level 4 averaged 4,513 cells/box, or about 6 times the tile size. Only 6% of these boxes on this level came from the partitioning step. The partitioner effectively limited extraneous box cutting.

Table 2 shows size characteristics for the strong-scaling Sphere benchmarks. The characteristics changed as we distributed the fixed-sized mesh on different process counts, so we present typical results on small, medium and large process counts: 8K, 128K, and 1.5M. As we saw with the weak-scaling benchmark, when there was enough work (at least 1 tile) per process, the partitioner managed to keep the max overload to around 10%.

At 1.5M processes, the Sphere benchmark's Level 5 flipped between dominant overload characteristics of 7% and 14% but averaged about 10% overload. We show both characteristics at the bottom of the table. The large overload in this case was caused by an average work load that had gotten too small compared with the tile size. The biggest overload of 2040 cells was equivalent to about half a tile. This illustrates the desirability of smaller tile sizes for fine-scale balancing.

Box counts averaged less than 4 per process for most Sphere benchmark meshes. The exception was on the finest level at 8K processes, which averaged 18.71 boxes/process. As in the Wall benchmark, this was caused by the number of cells required to discretize a large feature at a fine scale. Only 2.5% of the boxes came from the partitioner. The average box contained 157,162 cells, or about 38 16x16x16 tiles. The 18.71 boxes compared well with the 718 it would have taken had we not coalesced tiles. Though higher box counts contributed to more mesh-management work, in this case, they were necessary and correlated with high cell counts, so their cost remained low compared to computations, as we saw in the scaling plots of Figure 10.

Results for the Euler benchmark, shown in Table 3, were similar. The largest average box count was 123, for Level 3 on 8K processes. That number compares well with the 1446 tiles that would have been required had we not coalesced.

Level num.	cells/task			boxes/task			avg cells per box
	avg	max-avg	$\frac{\max}{\text{avg}}$	avg	max	$\frac{\max}{\text{avg}}$	
0	108	864	9.00	0.15	1	6.67	721
1	1,094	364	1.33	1.02	2	1.97	1076
2	10,525	1,139	1.11	3.63	11	3.03	2786
3	91,239	10,092	1.11	20.98	44	2.10	4348
4	724,558	59,117	1.08	160.6	311	1.94	4513

Table 1: Weak-scaling Wall benchmark typical cell and box counts (per MPI task): This representative data came from a snapshot of the 1M-core run. Tile size for this benchmark was 9x9x9, or 729 cell/tile. The clustering and partitioning algorithms effectively balanced load while keeping box counts low.

Core count	Level num.	cells/task			boxes/task			avg cells per box
		avg	max-avg	$\frac{\max}{\text{avg}}$	avg	max	$\frac{\max}{\text{avg}}$	
8K	0	4	4,092	1024	$\ll 1$	1	*	4096
	1	16	4092	1024	$\ll 1$	1	*	4096
	2	93	4,003	44.0	0.02	1	*	4096
	3	1,856	2,240	2.21	0.45	1	2.21	4096
	4	60,095	2,881	1.05	3.25	11	3.39	18,343
	5	2,940,995	274,364	1.09	18.71	92	4.92	157,162
128K	0	$\ll 1$	4,096	*	$\ll 1$	1	*	4096
	1	1	4095	4096	$\ll 1$	1	*	4096
	2	6	4,090	683	0.0015	1	*	4096
	3	116	3,980	35.4	0.028	1	*	4096
	4	3,756	341	1.09	0.92	3	3.27	4089
	5	183,819	16,885	1.09	3.68	17	4.62	49,941
1.5M	0	$\ll 1$	4,096	*	$\ll 1$	1	*	4096
	1	$\ll 1$	4,096	*	$\ll 1$	1	*	4096
	2	1	4,095	*	$\ll 1$	1	*	4096
	3	10	4,086	426	.0023	1	*	4096
	4	313	3,783	13.1	0.077	1	*	4088
	5	15,318	2,090	1.14	1.61	5	3.10	9486
			1,066	1.07				9493

Table 2: Typical cell and box counts for strong-scaling Sphere benchmark on 8K, 128K, and 1.5M cores. Asterisks denote large $\frac{\max}{\text{avg}}$ values caused by small averages and are therefore not meaningful. Tile size for this benchmark was 16x16x16, or 4096 cells/tile. Level 5 had two dominant overload characteristics at 1.5M processes.

Core count	Level num.	cells/task			boxes/task			avg cells per box
		avg	max-avg	$\frac{\max}{\text{avg}}$	avg	max	$\frac{\max}{\text{avg}}$	
8K	0	1,773	531	1.30	1.62	3	1.85	1095
	1	1,919	2,048	4.27	1.59	3	2.52	1209
	2	29,923	2,845	1.10	8.36	27	3.23	3581
	3	740,482	64,894	1.09	123	272	2.21	6060
128K	0	111	1617	15.6	0.21	1	4.85	538
	1	120	392	4.27	0.23	1	4.27	512
	2	1,870	178	1.10	1.58	4	2.54	1186
	3	46,280	4,408	1.10	9.13	38	4.16	5068
1.5M	0	9	1,719	187	0.017	1	*	538
	1	10	502	51.2	0.019	1	*	512
	2	156	356	3.29	0.30	1	3.25	512
	3	3,857	239	1.06	2.55	8	3.14	1515

Table 3: Typical load balance statistic for the strong-scaling Euler benchmark on 8K, 128K, and 1.5M cores. The Euler mesh changed rapidly. This data was taken at the end of one Level-0 time step. Asterisks denote large $\frac{\max}{\text{avg}}$ values caused by small averages and are therefore not meaningful. Tile size for this benchmark was 8x8x8, or 512 cells/tile.

6.6 Data locality

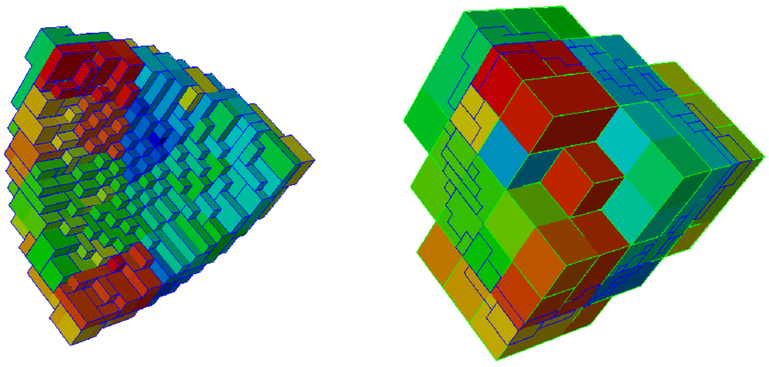
We knew of no generally acceptable metric for comparing data locality, but we attempted three evaluations: patch ownership image, `Connector` statistics, and MPI wait time during mesh data transfer.

Images of the mesh colored by ownership provided an initial qualitative evaluation. Figure 14 shows this for the Euler benchmark. For this illustration, we dropped the mesh down to 3 levels and ran on 64 MPI tasks. (The 4-level mesh in the benchmark was challenging to plot, due to its size.) The ownership images of levels 1 and 2 showed that patches close together in physical space tend to be close together in MPI rank space. This translated to being close on the communication network of BGQ machines and others where proximity in MPI rank space correlated with network proximity. There was also a degree of inter-level proximity, suggested by the ownership correlation between the two levels. Coarse-level patches tend to overlap fine-level patches with similar color, as seen in the third image.

Statistics of proximity `Connector`s provided our second indicator of data locality. As a reference for comparing intra-level locality, we use a hypothetical parallelpiped domain cut through by grid planes. In the 3D limit away from level boundaries, this produces boxes with 26 neighbors each. In the 2D limit, it is 8 neighbors. If we size the boxes to fit the average workload, each process is given a single box. Intra-level data transfer requires communicating with 26 other processes in the 3D limit.

Tables 4 and 5 show key statistics from `Connector`s to Level 5 ($\mathcal{L}_5$) of the Sphere benchmark. $\mathcal{L}_5$ was large and had the greatest risk of fragmentation and loss of locality. (The Wall benchmark had a larger level, but its periodic features could lead to an artificial mesh-dependent pattern on the edges statistics.) Fragmentation risk increased with more MPI tasks, so we present the data for 8K, 128K, and 1.5M tasks.

The intra-level proximity `Connector` $\{\mathcal{L}_5 \xrightarrow{5} \mathcal{L}_5\}$ in Table 4 had a width of 5 cells, the maximum that SAMRAI required for this particular `Connector`. There was an average of $13.5\frac{\text{edge}}{\text{box}}$ at 8K tasks, increasing to 22.5 at 1.5M tasks. This was significantly better than the 26 edges in the hypothetical partitioning. Although the spherical feature was about 410 cells across and appeared 3D from on the scale of the cells, it was only 25.6 tiles across and only 3.37 averaged boxes across. On the scale of boxes, the feature appeared more 2D and thus the statistics was closer to those for the 2D hypothetical partitioning. In this case,


(a) Level 2. (b) Level 1 with Level 2 in outline.

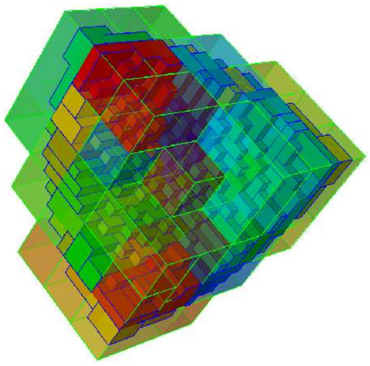

(c) Levels 2 and 1 superimposed.

Figure 14: Patch owners for Euler benchmark: This 3-level, reduced-size, 64-process configuration illustrates typical proximity characteristics. Colors reflect patch ownership, changing from zero (blue) to 63 (red). Physically close patches tend to be close in rank space for both inter-level intra-level patches, suggesting significant data locality.

coalescing tiles greatly improved data locality by creating boxes on the order of the feature size.

At 8K tasks, $\{\mathcal{L}_5 \xrightarrow{5} \mathcal{L}_5\}$ averaged 93.7 neighbors per task. A neighbor is a head box with an edge from a local base box. If there were multiple base boxes with edges to a neighbor, we still counted it as one neighbor. Of the 93.7 neighbors, 75 were remote, and these were owned by an average of 34.4 tasks. This was worse than the 26 remote tasks for the hypothetical partitioning, particularly considering that we were close to the 2D limit. However, with $252\frac{\text{edge}}{\text{task}}$ average, limiting them to only 34.4 remote tasks indicates significant locality. Moreover, 18.7 (20%) of the neighbors were local, which could provide some work to keep the process busy while waiting for asynchronous messages during data transfer. We do not, however, know how important that effect is.

At 1.5M tasks, $\mathcal{L}_5$ was more finely partitioned and less coalesced. Boxes were smaller and the feature appeared more 3D. The edge statistics reflected this. Each task averaged 31.4 neighbors in $\{\mathcal{L}_5 \xrightarrow{5} \mathcal{L}_5\}$, and almost all of them (29.7) were remote. With an average of 1.61 boxes per task on $\mathcal{L}_5$, one could not expect multiple local intra-level neighbors on the level. On average, these neighbors were owned by 25.8 remote tasks, very close to the 26 of the hypothetical partitioning.

Table 5 shows statistics for the coarse-to-fine `Connector` $\{\mathcal{L}_4 \xrightarrow{3} \mathcal{L}_5\}$. Inter-level `Connectors` tend to have different characteristics than intra-level `Connectors` because (1) inter-level patches can overlap, (2) coarser level patches tend to be physically much larger and (3) coarse-to-fine

`Connector`s tend to be physically wider. The $\{\mathcal{L}_4 \xrightarrow{3} \mathcal{L}_5\}$ `Connector`'s width was equivalent to 12 $\mathcal{L}_5$ cells. Relative to intra-level `Connector`s, coarse-to-fine `Connector`s contains many more relationships for fewer base boxes. Fine-to-coarse `Connector`s have more base boxes but fewer relationships per box, which does not shed much insight on data locality.

At 8K MPI tasks, for $\{\mathcal{L}_4 \xrightarrow{3} \mathcal{L}_5\}$, each task averaged 101 edges, connecting to 81.6 neighbors. On average, 16.2 edges pointed to local boxes. Given the average of 18.71 $\frac{\text{box}}{\text{task}}$ on $\mathcal{L}_5$, 16.2 local neighbors was a high degree of locality. The 64.4 remote neighbors lived on 29.4 remote tasks, which is similar to the 26 tasks for the hypothetical partitioning.

At 1.5M tasks, only 8% of tasks had any edges in $\{\mathcal{L}_4 \xrightarrow{3} \mathcal{L}_5\}$, which could not make for meaningful statistics. For example, there was an average of $96.9\frac{\text{edge}}{\text{box}}$ but the deceptive $7.4\frac{\text{edge}}{\text{task}}$ was due to the vast majority of task having no edge. We will instead examine the 128K-task run. It had a low $0.92\frac{\text{box}}{\text{task}}$, but that was sufficiently far from zero to provide meaningful numbers. At 128K tasks, $\{\mathcal{L}_4 \xrightarrow{3} \mathcal{L}_5\}$ averaged $35.8\frac{\text{edge}}{\text{task}}$, connecting to 35.8 neighbors, so no neighbors were connected multiple times in the average case. Of these neighbors, 2.4 were local and 35.8 were remote. Given only $3.68\frac{\text{box}}{\text{task}}$ in $\mathcal{L}_5$, 2.4 local neighbors represents significant locality.

High data locality helps data transfer by reducing internal boundaries, reducing message transit times, or eliminating transit time completely in cases where source and destination are both local. It also helps computing transfer schedules that specify which data should be transferred where. Poor data locality could increase data-transfer time, so our third locality evaluation examines data-transfer time. Figure 15 shows how data-transfer components scaled for the benchmarks run with four levels. We used the sequential Numerical Kernels time for comparison. In the weak-scaling benchmark all components scaled well, exhibiting characteristics seen in regridding operations with peer-to-peer communication. They initially rose due to the increasing number of available tasks to communicate with, but they leveled out before the point of 100 tasks.

In the weak-scaling benchmark, constructing and executing refine schedules each took about a third of the time numerical kernels did. Though this seems high, recall that the kernel for linear advection was very simple. Executing refine schedules took about as much time as exact boundary conditions did up to 32 cores. Bridge and modify operations were used in refine schedule construction, where temporary supplemental levels were computed and connected. Their good weak-scaling characteristics suggest fast metadata operations. The two MPI wait timings for mesh data transfer also scaled well, again suggesting good data locality. Coarsen schedules were simpler and used less often, so their cost was well below that of refine schedules.

Data transfer characteristics for strong-scaling benchmarks were more difficult to evaluate, because locality effects were subtle compared with those due to rapidly shrinking local work. However, the ordering of the components was very similar to that of the weak-scaling case, suggesting similar effects of data locality. In the Euler benchmark, where there was more significant computation, relative cost of data transfer was lower for all components.

7 Summary and conclusion

We developed and benchmarked two significant algorithms for patch-based AMR regridding: a flexible tile-clustering algorithm and a cascading partitioning algorithm. Integrated into SAMRAI's distributed mesh-management scheme, they required no unscalable communication to work. Our test benchmarks were designed to be challenging for dynamic regridding, using simple physics and numerics, frequent regridding, and a broad range of mesh sizes. They scaled well to 1.5M cores and to 2M MPI tasks.

Our weak-scaling benchmarks achieved efficiencies of 72% to 84% at 2M tasks. Strong-scaling benchmarks achieved 11X to 60X speed-up factors. More importantly, detailed timing of all operations showed smooth, steady trends that suggest we could continue into significantly higher core counts. Varying mesh size and physics complexity showed that relative regridding cost would be significantly lower for problems that had more computational work than our benchmarks did.

The partitioner kept maximum overloads to

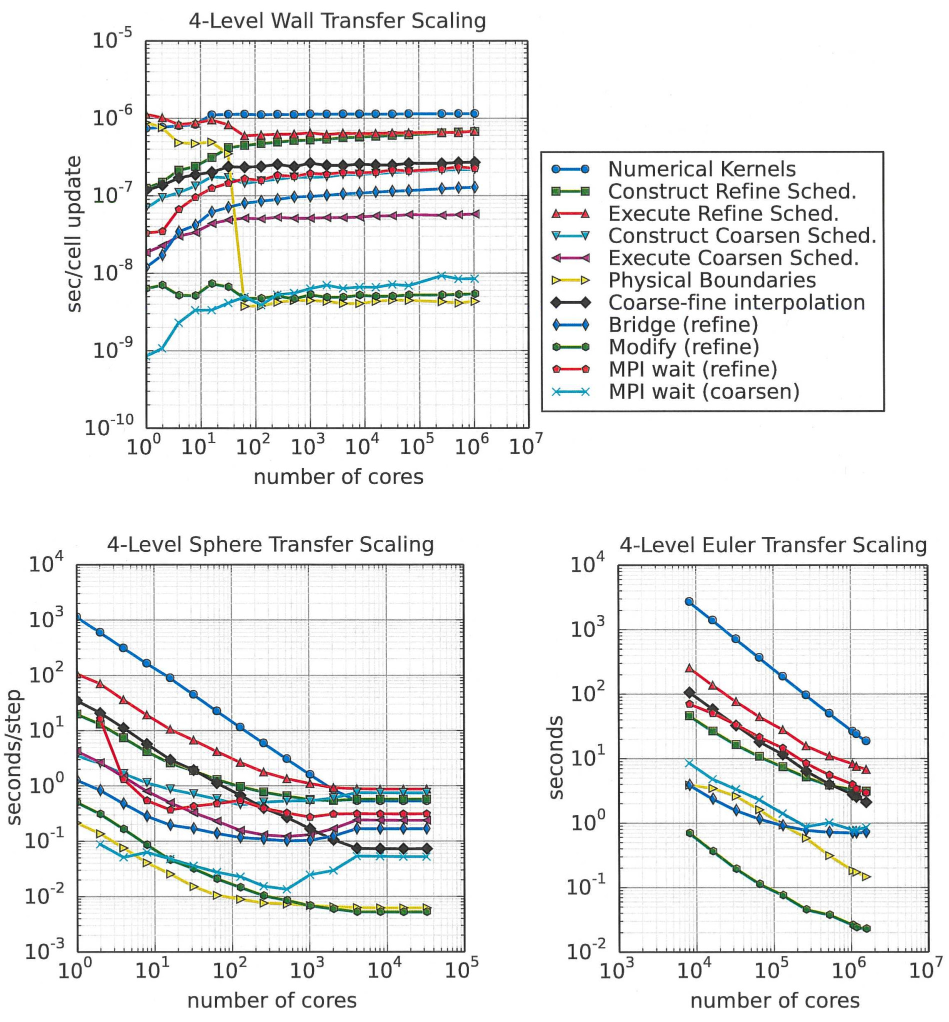


Figure 15: Data transfer cost for weak-scaling advection, strong-scaling advection and strong-scaling Euler benchmarks: For reference, the cost of the serial numerical kernels is also shown. (Coarsen schedule times are missing from the Euler benchmark because the timers were inadvertently disabled.)

$\{\mathcal{L}_5 \xrightarrow{5} \mathcal{L}_5\}$ intra-level statistics	Task count		
	8K	128K	1.5M
edges/box	13.5	19.2	22.5
edges/task	252	70.6	36.3
local neighbors/task	18.6	3.7	1.6
remote neighbors/task	75	44.4	29.7
remote neighbor owners/task	34.4	31.5	25.8

Table 4: Data locality indicators from intra-level `Connector` for Level 5 of Sphere benchmark at time 0.129.

$\{\mathcal{L}_4 \xrightarrow{3} \mathcal{L}_5\}$ inter-level statistics	Task count		
	8K	128K	1.5M
edges/box	31.0	38.9	96.9
edges/task	101	35.8	7.4*
local neighbors/task	17.3	2.4	0.1*
remote neighbors/task	64.4	33.4	7.3*
remote neighbor owners/task	29.4	24.9	6.2*

Table 5: Data locality indicators for coarse-to-fine (Level 4 to Level 5) `Connector` for Sphere benchmark at time 0.129. The `Connector` was wider than that in Table 4. It was 3 Level-4 cells (12 Level-5 cells). Values with * are not useful because they are averages over tasks, but only 8% of tasks owned boxes on $\mathcal{L}_4$.

about 10% using minimal box cutting to prevent over-fragmenting levels. The clustering algorithm did a good job coalescing tiles–averaging as high as 38 tiles per box–to reduce metadata size and mesh-management cost, increase data locality, reduce memory used by ghost cells, and speed up data transfer. We compared the different benchmarks by examining the amount of local computation at the point where the regridding time equals sequential computation time. This removed the more dominant effects of work load differences and demonstrated that problems with more computational work could scale to a smaller amount of work per process.

Further research could include extensions to the partitioner to support non-uniform work function, a load-apportioning scheme that considers locations to further improve data locality. Since regridding time and over-fragmentation seems to be under control, a fast operation to limit patch sizes could speed up solvers by improving cache efficiency. Threading and graphics processing units (GPU) appear to be the next big step in computational speed, so finding a way to thread mesh-management operations will be important. This would be very challenging because mesh management has so little metadata to work on and it is very communication- and logic-intensive.

References

[AEP04] R. W. Anderson, 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(2):598–617, 2004.

[AMR15] AMROC - blockstructured adaptive mesh refinement in object-oriented c++, 2015. http://amroc.sourceforge.net/.

[BC89] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, 1989.

[BDL+15] Erik Boman, Karen Devine, Vitus Leung, Sivasankaran Rajamanickam, Michael Wolf, and Umit Catalyurek. Zoltan home page. http://www.cs.sandia.gov/Zoltan, 2015.

[BGG+10] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Tobin Isaac, Georg Stadler, Tim Warburton, and Lucas Wilcox. Extreme-scale amr. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[BGI+12] Abhinav Bhatele, Todd Gamblin, Katherine E. Isaacs, Brian T. N. Gunney, Martin Schulz, Peer-Timo Bremer, and Bernd Hamann. Novel views of performance data to analyze large-scale adaptive applications. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12. IEEE Computer Society, November 2012. LLNL-CONF-554552.

[BO84] Marsha J. Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[Box15] Boxlib web site, 2015. https://ccse.lbl.gov/BoxLib/.

[BR91] M. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man and Cybernetics*, 21(5):1278–1286, September/October 1991.

[CG85] Phillip Colella and Harland M. Glaz. Efficient solution algorithms for the riemann problem for real gasses. *Journal of Computational Physics*, 59:264–289, 1985.

[CGJ+13] P. Colella, D. T. Graves, J. N .Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, T. D. Sternberg, B. Serafini, and B. Van Straalen. *Chombo Software Package for AMR Applications – Design Document*. Applied Numerical Algorithms Group, Computational Research Division, Lawrence Berkeley National Laboratory, October 2013. http://seesar.lbl.gov/anag/chombo/ChomboDesign-3.1.pdf.

[Cho15] Chombo - software for adaptive solutions of partial differential equations, 2015. http://chombo.lbl.gov/.

[Col90] Phillip Colella. Multidimensional upwind methods for hyperbolic conservation laws. *Journal of Computational Physics*, 87:171–200, 1990.

[CP08] C. Chevalier and F. Pelegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34:318–331, 2008.

[DBH+02] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[Gun12] Brian T. N. Gunney. Scalable Mesh Management for Patch-based AMR. In *NECDC 2012 Proceedings*, October 2012.

[GWH06] Brian T. N. Gunney, Andrew M. Wissink, and David A. Hysom. Parallel clustering algorithms for structured AMR. *Journal of Parallel and Distributed Computing*, 66(11):1419–1430, November 2006.

[HK02] Richard D. Hornung and Scott R. Kohn. Managing application complexity in the SAMRAI object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14:347–368, 2002.

[KB96] Scott R. Kohn and Scott B. Baden. Irregular coarse-grain data parallelism under LPARX. *Scientific Programming*, 5(3):185–201, 1996.

[KK99] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[KK15] George Karypis and Vipin Kumar. Family of graph and hypergraph partitioning software. http://www.cs.umn.edu/ metis, 2015.

[LB11] J. Luitjens and M. Berzins. Scalable parallel regridding algorithms for block-structured adaptive mesh refinement. *Concurr. Comput. : Pract. Exper.*, 23(13):1522–1537, September 2011.

[LBH07] J. Luitjens, M. Berzins, and T. Henderson. Parallel space-filling curve generation through sorting: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(10):1387–1402, July 2007.

[Lui11] Justin Paul Luitjens. *The Scalability of Parallel Adaptive Mesh Refinement within Uintah.* PhD thesis, University of Utah, May 2011.

[SAM15] SAMRAI project web site, 2015. https://computation.llnl.gov/project/SAMRAI/.

[Tra95] John A. Trangenstein. Adaptive mesh refinement for wave propagation in nonlinear solids. *SIAM Journal on Scientific Computing*, 16(4):819–839, July 1995.

[WHH03] Andrew M. Wissink, David Hysom, and Richard D. Hornung. Enhancing scalability of parallel structured AMR calculations. In *Proceedings of the 17th ACM International Conference on Supercomputing (ICS03)*, pages 336–347, San Francisco, June 2003.