Low-Cost Parallel Algorithms for 2:1 Octree Balance

Tobin Isaac*, Carsten Burstedde*†, Omar Ghattas*‡§
*Institute for Computational Engineering and Sciences (ICES)
The University of Texas at Austin, USA
Email: {tisaac,carsten,omar}@ices.utexas.edu
†Present address: Institut für Numerische Simulation (INS)
Rheinische Friedrich-Wilhelms-Universität Bonn, Germany
‡Jackson School of Geosciences, The University of Texas at Austin, USA
§Department of Mechanical Engineering, The University of Texas at Austin, USA

***Abstract*—The logical structure of a forest of octrees can be used to create scalable algorithms for parallel adaptive mesh refinement (AMR), which has recently been demonstrated for several petascale applications. Among various frequently used octree-based mesh operations, including refinement, coarsening, partitioning, and enumerating nodes, ensuring a 2:1 size balance between neighboring elements has historically been the most expensive in terms of CPU time and communication volume. The 2:1 balance operation is thus a primary target to optimize.**

One important component of a parallel balance algorithm is the ability to determine whether any two given octants have a consistent distance/size relation. Based on new logical concepts we propose fast algorithms for making this decision for all types of 2:1 balance conditions in 2D and 3D. Since we are able to achieve this without constructing any parent nodes in the tree that would otherwise need to be sorted and communicated, we can significantly reduce the required memory and communication volume. In addition, we propose a lightweight collective algorithm for reversing the asymmetric communication pattern induced by non-local octant interactions.

We have implemented our improvements as part of the open-source "p4est" software. Benchmarking this code with both synthetic and simulation-driven adapted meshes we are able to demonstrate much reduced runtime and excellent weak and strong scalability. On our largest benchmark problem with 5.13×10^{11} octants the new 2:1 balance algorithm executes in less than 8 seconds on 112,128 CPU cores of the Jaguar Cray XT5 supercomputer.

***Keywords*-Octrees, Adaptive mesh refinement, Parallel algorithms, Scientific computing, High performance computing**

I. Introduction

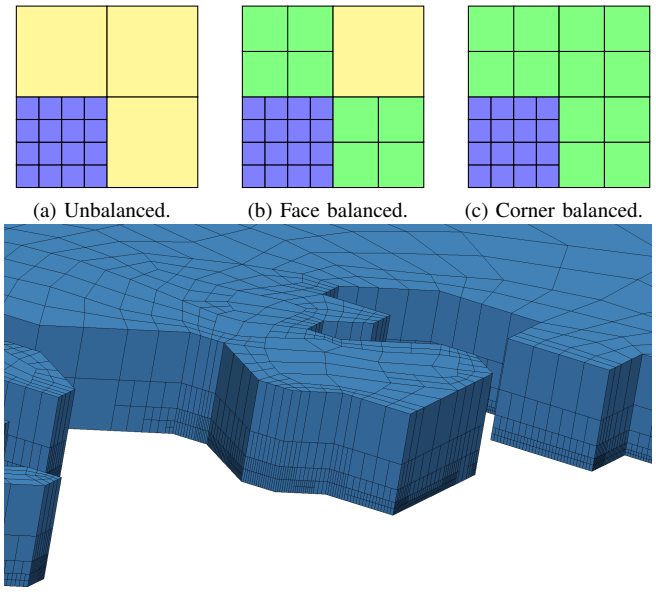
The accurate numerical solution of partial differential equations (PDEs) that give rise to multiscale phenomena is a long-standing subject of active research. A broad range of adaptive approaches has been developed over the years, for example using finite elements [1] or volumes [2], wavelets [3], meshfree methods [4], and many more.

We are concerned with numerical methods that use a forest-of-octrees computational mesh to cover the spatial domain of the PDE. Methods that use adaptive element sizes are often categorized as either block-structured (see e.g. [5]), where a hierarchy of overlapping and successively finer uniform grids is used, or unstructured (see e.g. [6]), where tetrahedra or hexahedra are connected through common faces into a graph. Parallel implementations have been developed for both block-AMR (see e.g. [7], [8], [9], [10], [11], [12]) and unstructured AMR (e.g. [13], [14]), and some have been evaluated at the petascale [15], [16].

Octree-based methods for adaptive mesh refinement are a hybrid construction in the sense that meshes are derived from uniform blocks by hierarchical splits, but do not produce overlap between elements of different size, see e.g. [17], [18], [19], [20], [21], [22], [23]. General domain shapes can be enabled by connecting multiple octrees into a "forest," optionally making use of mesh generation software to lay out the octrees in space. Using space-filling curves for encoding and partitioning of the mesh at the leaf level then essentially eliminates shared meta-data and enables very fast and lightweight load-balancing. This approach naturally accommodates generic finite elements when suitable interpolation operators are defined at T-intersections [24]. Forest-of-octrees AMR has been demonstrated on over 220,000 CPU cores [25] and is thus particularly well-suited for frequent (dynamic) adaptation at extremely large scales. It has recently been used in global-scale seismic wave propagation simulation [26] and enabled high-resolution scientific studies of mantle dynamics and plate tectonics [27].

A central building block for octree-based AMR is the 2:1 balance operation which is usually invoked after refinement and coarsening to reestablish well-defined size relations between neighboring elements (see Figure 1 for a schematic illustration). For highly graded meshes the balance operation can have long-range effects between non-neighboring processes which requires a careful design of any parallel balance algorithm. Experiments have shown that 2:1 balance is the most expensive octree-related algorithm (much more so than partitioning for example [25]), and while generally scaling to large clusters and being cheaper than numerical finite-element operations, still demands significant memory and communication resources.

In this paper, it is our objective to remedy this issue by analyzing and exploiting the logical structure of 2:1 balance



(a) Unbalanced. (b) Face balanced. (c) Corner balanced.

Figure 1. Top: 2:1 balance status for a 2D quadtree mesh. Balance across faces (b) ensures that T-intersections only occur once per face which is often required by numerical discretizations. Corner balance (c) produces a smoother grading of the mesh. Bottom: Part of a forest-of-octrees mesh of Antarctica, where AMR with 2:1 corner balance is used to achieve the resolution required for capturing physical processes at the base of the ice.

relations. After introducing the concept of parallel octree-based AMR and summarizing the current state of research in Section II, we address three main challenges in providing a low-cost 2:1 balance algorithm. In Section III we propose a new subtree balancing algorithm that is optimized for data structures that represent linear octrees. In Section IV we introduce new concepts and algorithms to determine whether two remote octants are balanced, which we use to greatly reduce the amount of both communication and computation required. Finally, in Section V we provide a lightweight divide-and-conquer algorithm for encoding the asymmetric communication pattern required for balancing that avoids resorting to Allgather-type collective communication calls. We evaluate the performance of our new algorithms in Section VI. Using our improvements we are able to demonstrate a 3.5× speedup and improved scalability up to half of the 2.33 petaflop Jaguar Cray XT5 supercomputer at Oak Ridge National Laboratories. To the best of our knowledge, this is now the fastest and most scalable 2:1 balance algorithm. We conclude in Section VII.

II. Overview of Octree-based Meshing

In this section we provide a brief exposition of some essential concepts related to parallel octree meshing. For details we refer the reader to the self-contained presentation [28]. In general, we rely on a 1-to-1 identification of an adaptive tree to a computational domain as sketched in

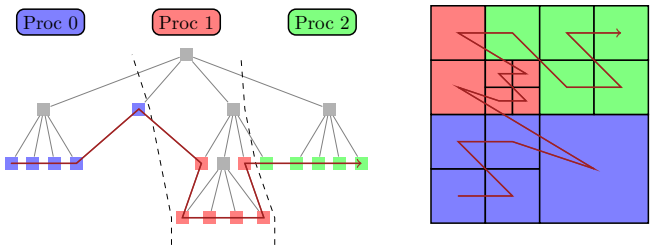


Figure 2. Identification of a quadtree and a corresponding volume mesh in 2D space. The octants are numbered left to right which establishes a total ordering of mesh elements with respect to a space-filling curve (dark red) that proceeds in a z-shaped recursive pattern. The curve is also used for equipartitioning the leaf octants, and thus the mesh elements, between different processes. The extension to octrees in 3D space is analogous.

Table I
Common octant relationships

size(o)	the length of o's sides is $2^{\text{size}(o)}$
parent(o)	the octant containing o that is twice as large
i-child(p)	the child of p that touches the ith corner of p
i-sibling(o)	i-child(parent(o))
family(o)	$\cup_i$ i-sibling(o)
child-id(o)	i such that i-child(parent(o)) $= o$
root	an octant representing a whole octree

Figure 2. The domain is divided into mesh elements, each of which corresponds to a leaf octant. By traversing all leaves of the tree left to right, we obtain a space-filling curve that establishes a total order among all mesh elements. This order can be used to partition the elements between processes, and for binary comparisons and searches of elements within any given process. In the following we denote the number of processes by $\mathcal{P}$, and the number of octants per process by $\mathcal{N}$.

A. Octants and Octrees

In this section we introduce some definitions that are essential to formulate our algorithms. Let the term *octant* denote a d-dimensional cube whose sides have length 2^l for some $l \in \mathbb{Z}$ and whose corners have coordinates that are all integer multiples of 2^l: we use the shorthand l-octant to indicate its size. The location of octant corners at regular intervals organizes octants into a tree structure, where an l-octant is contained in only one $(l+1)$-octant (its parent). For reference, we provide notation for common octant relationships is given in Table I.[1]

Non-overlapping octants can be ordered by a space-filling curve that traverses the descendants of octant o before the descendants of its sibling r if and only if child-id(o) $<$ child-id(r). We use $o \leq r$ to represent comparison using this ordering. By specifying $o < r$ if o is an ancestor of r, we establish that if an ordered array contains overlapping

[1]One often finds octants referred to by their level in the octree relative to the root. If the root is an L-octant, then an l-octant has level $(L-l)$.

octants, an octant precedes its descendants (this is equivalent to preorder traversal, often referred to as Morton order).

Octrees can be stored in different ways: by specifying pointers between parents and children [19], by compressing the pointers into index lists [21], or by storing only the leaves in space-filling curve order. The latter scheme is called a linear octree [22], which lends itself to formulating the relations in Table I via bitwise mask and shift operations on the integer octant coordinates [28].

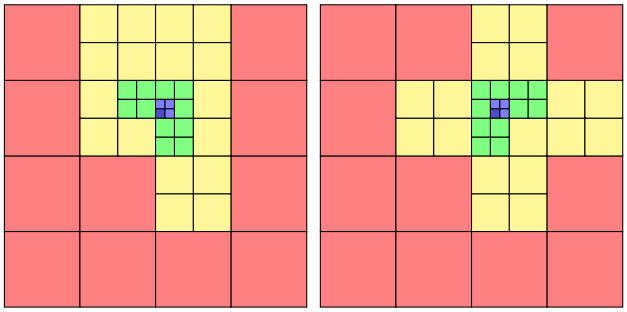
Geometries that are topologically more complex than a cube can be mapped by connecting multiple mapped cubes, where each cube is identified with an individual octree. This approach allows a-priori representations of primitive shapes, such as a torus or a hollow sphere. It also permits the translation of output from tetrahedral or hexahedral mesh generators into a macro-mesh that connects many thousand octrees (see Figure 1 for an example). We like to call such a construction a forest of octrees. Here, a given ordering between the trees can be used to connect the space-filling curves in the individual tree, creating a total octant order throughout the forest that can again be used for parallel partitioning and search.

To adhere to distributed memory parallelism, each compute core must only store a small fraction of the whole mesh and the associated numerical data. On an adapted forest of octrees this raises the issue of how to ensure equal load balance between the compute cores. To address this issue for possibly variable octant loads, a fast weighted partitioning scheme has been developed elsewhere [28] based on appropriate subdivisions of the space-filling curve (see Figure 2).

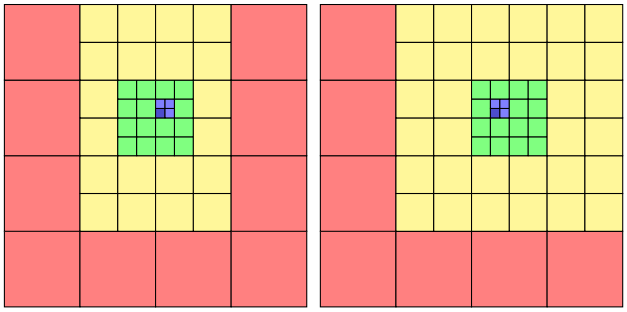
B. Algorithms for 2:1 Balance

Octree balance relates neighboring octants to each other. The definition of "neighboring" depends on the application, and each definition is associated with a certain balance condition, see also Figure 5. Many finite element codes use octrees whose octants must be balanced if they share a face or an edge. Methods that work with discontinuous functions, such as finite volume and discontinuous Galerkin methods, may only require 2:1 size faces. Balance across face, edges, and corners is rarely necessitated by numerical methods but may be useful for mesh smoothing or node-based operations. As shorthand, we refer to balance conditions by the number of boundary objects that require a 2:1 condition, e.g., 2-balance means balance across faces and corners in 2D and balance across faces and edges in 3D. When all pairs of neighboring octants in an octree are balanced, the octree is said to be balanced.

Given an arbitrary octant o and a prescribed balance condition k, there exists a unique octree $T_k(o)$ that contains o as a leaf, is k-balanced, and is the coarsest such octree, i.e., if any leaf octants were replaced with a coarser octant, the tree would become unbalanced. $T_k(o)$ is illustrated in Figure 3.



(a) $T_1(o)$ for $k = 1$ and two different choices of o.



(b) $T_2(o)$ for $k = 2$ and two different choices of o.

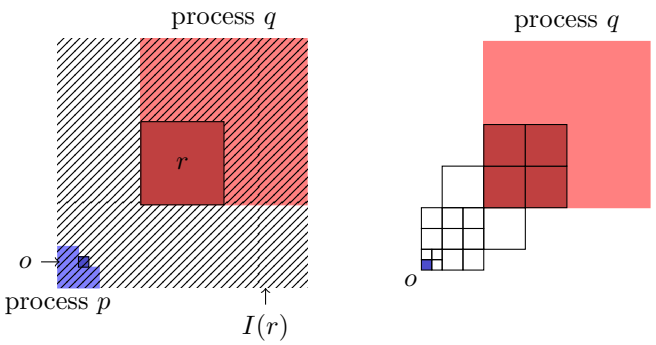
Figure 3. Examples of the coarsest balanced octree $T_k(o)$ for a given octant o in 2D for (a) $k = 1$ and (b) $k = 2$. In each figure o is blue, with other octants colored by size. Note that size increases outward in a ripple-like fashion, and that the shape of the ripple depends on the coordinates of o for each choice of k.

The idea of balance extends from neighboring to arbitrary octants by declaring two octants to be balanced if they are both the leaves of some balanced octree. A useful fact about two octants o and r is that they can be unbalanced only if o is contained in r's insulation layer $I(r)$, an envelope of 3^d like-sized octants, or vice versa. For a partitioned octree, the processes that must share information in order to balance the octree are determined by comparing insulation layers with process partitions, see Figure 4a.

If an octant is too coarse to be balanced with another octant, it can be *split*—replaced by its children—to bring the octree closer to being balanced. An algorithm that only compares neighbors when determining which octants to split is called a *ripple* algorithm, due to the fact that splitting one octant can cause another octant to split and so on. Parallel ripple algorithms only use communication between processes with neighboring partitions, so they generally requires multiple rounds of communication when an octant ultimately causes another octant on a remote process's partition to split.

A one-pass 2:1 balance algorithm can use insulation layer calculations to group all communication into one round only: a query and a response. This algorithm has four phases (see also [22], [28]):

1) *Local balance.* Each process balances the octants in its own partition with respect to each other.



(a) Insulation layer and partitions (b) Local rebalance

Figure 4. (a) Octant r is in process q's partition; octant o is in process p's partition and is inside r's insulation layer $I(r)$. Every balance algorithm directly or indirectly determines if o causes a split of r, so information must propagate from p to q. (b) Once process q knows of o, q must determine if its octants are balanced with o. If the balance of remote octants cannot be determined directly, then auxiliary octants must be constructed in between (empty squares which in this case cause a split).

2) *Query.* For each of its octants r, a process determines which processes have partitions that overlap $I(r)$ and adds r to a set of query octants sent to those processes.
3) *Response.* For each octant r received in a query, a process checks which of its octants might cause r to split and adds those octants to the response set sent to the owner of r.
4) *Local rebalance.* Each process balances its own partition with respect to the octants it has received, i.e., for every received octant o, a process q must determine which of its octants are coarser than the octants in $T_k(o)$ and thus need to be split.

Below we list three issues affecting the performance of such a one-pass 2:1 algorithm. We devote a section of this paper to each issue.

1) *Subtree balance.* When each process balances its own partition in the *Local balance* phase, a serial algorithm is required that is optimized for balancing arrays of octants that span a contiguous subset of the octree, which we call subtrees. In Section III we present an old algorithm for subtree balance and our new version.
2) *Balancing remote octants.* The basic operation in the *Local Rebalance* phase is for a process q to compute the portion of $T_k(o)$ that overlaps its partition, where o is an octant received from another process. Let us refer to this portion of $T_k(o)$ as S. If the octants in $T_k(o)$ can only be computed in a ripple-like algorithm that propagates octants outward from o, then before S can be computed we must compute *auxiliary octants*: octants in $T_k(o)$ that bridge the gap from o to q's partition (Figure 4b). Using auxiliary octants, the work performed by q to compute S depends not only on $|S|$, but on the distance between o and q's partition. In a worst case scenario, the work required to calculate S could be equivalent to the work required to calculate $T_k(o)$. In Section IV, we present a way to compute S whose performance is independent of the distance between o and q's partition.
3) *Encoding the communication pattern.* If no restrictions are placed on the way an octree is partitioned, then two octants in remote partitions may be unbalanced (Figure 4a). In contrast to the ripple algorithm for parallel balance, where all communication occurs in symmetric neighbor-to-neighbor exchanges, the communication pattern of the one-pass algorithm is situational and asymmetric: each process cannot determine on its own which other processes will send queries to it. In Section V, we present an old algorithm for determining this asymmetric communication pattern and our new version.

III. Subtree Balance

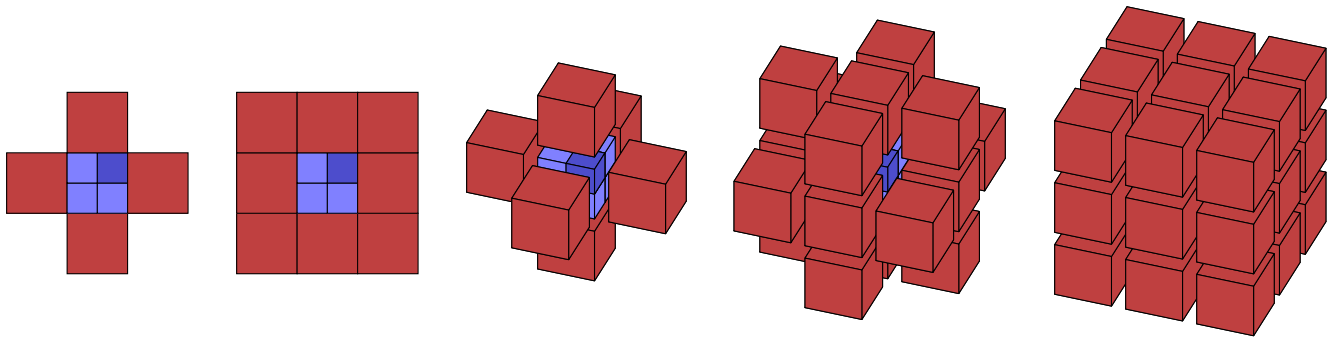
A subtree is a sorted array of octants with two additional properties: the subtree is *linear*, which means that no octant in the array is ancestor to any other (thus ruling out overlapping elements), and it is *complete*, which means that there are no missing leaves between two successive octants in the array (and thus no holes in the mesh). The root of an octree may be divided into multiple subtrees by drawing vertical separations in the tree diagram (see Figure 2 for a division of a tree into three subtrees). In this section we discuss our implementations of an existing 2:1 balance algorithm for subtrees and propose a new and faster variant.

For simplicity, we present our algorithms in terms of balancing an entire octree. These can be adapted to a subtree by treating the least common ancestor of the subtree as the root and removing octants before and after the subtree's octant range as a postprocessing step.

A. Existing Algorithm

Octants must be split to transform an arbitrary octree into a balanced octree: requiring that an octree remain complete and linear after each split would require frequent updates to the sorted array that represents it. In the old algorithm, we instead collect newly added octants in a hash table until we are sure that all new octants are present, at which point we combine and select old and new octants as a postprocessing step that removes the introduced overlap between elements.

The basic idea of the old algorithm is for each l-octant to attempt to add to the octree its family and the $(l+1)$-octants that neighbor its parent. We call the latter set the *coarse neighborhood* N of an octant, which depends on the type of 2:1 balance condition being enforced, as illustrated in Figure 5. Each octant that is added because it is a sibling or coarse neighbor of an existing octant also adds its siblings and coarse neighbors, and so on for every octant smaller than the root. At the conclusion of this procedure, every octant that will be in the balanced octree is present, in



(a) 2D, 1-balance. (b) 2D, 2-balance. (c) 3D, 1-balance. (d) 3D, 2-balance. (e) 3D, 3-balance.

Figure 5. The coarse neighborhoods $N(o)$ (red) around an octant o (blue) and its family (light blue) for all balance conditions in 2D and 3D (exploded view). If some or all octants of $N(o)$ extend beyond the boundaries of the root octree, they may influence a neighboring tree in the forest instead. In the old subtree balance algorithm, each octant o adds family(o) and $N(o)$ to the octree; in our new subtree balance algorithm, each octant o only adds 0-sibling(r) for each $r \in N(o)$.

Input: sorted array of octants S

$S^{\text{new}} \leftarrow \emptyset$
for all $o \in S \cup S^{\text{new}}$ **do**
  for all $s \in \text{family}(o) \cup N(o)$ **do**
    if $s \notin S \cup S^{\text{new}}$ **then**
      $S^{\text{new}} \leftarrow S^{\text{new}} \cup \{s\}$
    end if
  end for
end for
$S^{\text{final}} \leftarrow S \cup S^{\text{new}}$ {merge and sort}
return `Linearize`(S^{final})

Figure 6. High-level description of the old algorithm for balancing subtrees. This algorithm takes a sorted array of octants and produces the coarsest balanced octree for that set. Each octant iteratively adds its coarsest allowable neighborhood to the octree, resulting in many overlapping octants that will not be leaves in the final linear octree. These overlapping ancestor octants are removed by `Linearize`, an $\mathcal{O}(n)$ algorithm for removing coarse overlapping octants from a sorted array.

addition to many ancestors of these final octants, generally producing a large overlap between leaves and parent octants. In postprocessing, the new octants are merged with the input octants and sorted, and the set is linearized to retain only the leaves. Pseudocode for the old algorithm is given in Figure 6.

Because we use hash tables, determining if an octant is already among the newly added octants is an $\mathcal{O}(1)$ operation. Failing that, determining if an octant is in the original sorted set of octants is an $\mathcal{O}(\log \mathcal{N})$ operation.

The most costly part of postprocessing the algorithm is combining the old and new octants into a sorted array. In our implementation, this requires sorting a set the size of the output octree.

B. Octant Preclusion and New Algorithm

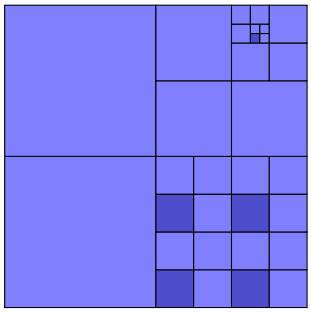
Our new variant of the previous algorithm is based on an octant relationship that we call *preclusion*, which captures a property essential for 2:1 balance. We say o precludes r, $r \prec o$ (or $r \preceq o$), if and only if parent(r) is ancestor (or equal) to parent(o). Preclusion defines a partial ordering of octants whose equivalence classes are families. Preclusion is a useful concept for compressing an octree: precluded octants can be removed from an octree, and the octree can be quickly recovered from the remaining octants by a completion algorithm that fills in the gaps between leaf octants in the coarsest possible way.

Input: sorted array of octants S

$R \leftarrow$ `Reduce`(S) {smaller equivalent set}
$R^{\text{new}} \leftarrow \emptyset$; $R^{\text{prec}} \leftarrow \emptyset$
for all $o \in R$ **do**
  for all $s \in N(o)$ **do**
    $s \leftarrow 0\text{-sibling}(s)$ {equivalent to s}
    if $s \notin R \cup R^{\text{new}}$ **then**
      $R^{\text{new}} \leftarrow R^{\text{new}} \cup \{s\}$
    end if
    if $s \prec o$ **then**
      $R^{\text{prec}} \leftarrow R^{\text{prec}} \cup \{s\}$ {tag precluded}
    else if there is $t \in R$ such that $t \prec s$ **then**
      $R^{\text{prec}} \leftarrow R^{\text{prec}} \cup \{t\}$ {tag precluded}
    end if
  end for
end for
$R \leftarrow R \backslash R^{\text{prec}}$; $R^{\text{new}} \leftarrow R^{\text{new}} \backslash R^{\text{prec}}$ {remove precluded}
$R^{\text{final}} \leftarrow R \cup R^{\text{new}}$ {merge and sort}
return `Complete`(R^{final})

Figure 7. High-level description of our new algorithm for balancing subtrees. This algorithm first reduces the input set to a compressed equivalent, and then has octants iteratively add a sparse set of octants that is equivalent under preclusion to their coarse neighborhoods. `Complete` is an $\mathcal{O}(n)$ algorithm for filling in the gaps in a sorted array. In our implementation, we use a hash table for R^{new}; our octant datatype has space for a tag that we use to mark inclusion in R^{prec}.

Our new algorithm (Figure 7) has much in common



Input: sorted array of octants S

1: $R[0] \leftarrow 0\text{-sibling}(S[0])$; $i \leftarrow 1$
2: **for** $0 \leq j < |S|$ **do**
3: $\quad s \leftarrow 0\text{-sibling}(S[j])$ {equivalent to $S[j]$}
4: $\quad r \leftarrow R[i-1]$ {last octant added}
5: $\quad$ **if** $r \prec s$ **then**
6: $\quad\quad R[i-1] \leftarrow s$ {replace r}
7: $\quad$ **else if** $s \not\preceq r$ **then**
8: $\quad\quad R[i] \leftarrow s$; $i \leftarrow i+1$ {append s}
9: $\quad$ **end if**
10: **end for**

Figure 8. Top: the sets $R = \texttt{Reduce}(S)$ (blue) and $S \backslash R$ (light blue). This figure illustrates the fact that R is the smallest subset of leaves that can be used to reconstruct a linear octree using a completion algorithm. Bottom: pseudocode for `Reduce`, which takes a sorted array representation of S and returns a sorted array representation of R.

with the old algorithm: new octants are kept in a hash table, and new and possibly overlapping octants are added iteratively. The new algorithm, however, first uses preclusion to compress the octree, and then each octant o adds a smaller set that is equivalent under preclusion to the coarse neighborhood $N(o)$ (the set of 0-siblings of octants in $N(o)$ is smaller than $N(o)$ because some of them are siblings themselves). The result of this process is still a reduced octree with gaps between leaves, so the result is then completed. Where the old algorithm tests for equality between new octants and input octants to avoid duplicates with a binary search of the input set, our new algorithm can determine whether a new octant precludes or is precluded by an octant in the reduced input with a single equivalent binary search. The benefit of working with the reduced octree is that the costliest step, namely sorting the union of old and new octants, is performed on the smallest possible set that can be completed to create the final balanced octree.

`Reduce` is our algorithm for removing precluded octants from a sorted array, for details see Figure 8. $R = \texttt{Reduce}(S)$ is sorted and if S is a complete octree then $|R| \leq |S|/2^d$. Most important for our new balancing algorithm is the fact that only one binary search is needed to determine if there is $t \in R$ such that $t \prec s$.

Our new algorithm is a drop-in replacement for the old algorithm and requires roughly 3 times fewer hash queries, smaller binary searches, and a reduction of the set that is sorted in postprocessing by the factor 2^d. In our tests in Section VI, we find that the savings in these operations more than offset the additional cost of reducing the input and completing the output.

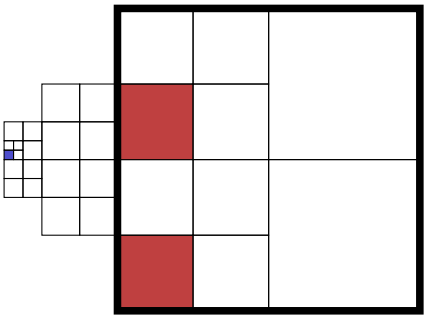


Figure 9. Seed octants (red) can reproduce the coarsest balanced octree $T_k(o)$ of a remote octant o (blue) within a specified query octant r (dark outline), when a balancing algorithm is run using the seed octants as inputs and the query octant as the root. The overlap of $T_k(o)$ and the octant r is the subtree S contained in the dark outline, and can be created from the seed octants only.

IV. Balancing Remote Octants

Recall from Section II-B that in the *Response* phase of the one-pass balance algorithm each process determines a subset of its octants which might cause a query octant to split, and sends those octants as a response to the query. At the end of that section we describe how the distance between a response octant o and a querying process q's partition can increase the work used by q in the old one-pass algorithm. This increase in work is a result of the fact that the old algorithm could not compute the overlap of o's coarsest balanced octree $T_k(o)$ and a remote partition without constructing auxiliary octants outside of that partition to bridge the gap (see Figure 4 for an illustration).

First, let us simplify this problem to the problem of computing the overlap of $T_k(o)$ with a single remote octant r, and let us call this overlapping set S. S itself is a subtree that has r as its root. The motivation for this section is that we would like for the querying process to perform work proportional to $|S|$ in computing S, in a way independent of the distance between o and r.

If we treat r as the root octant, then a subtree balancing algorithm can be adapted to reconstruct S from a subset $\tilde{S}$ of S (both the old and new algorithms for subtree balance in Section III also work efficiently on incomplete input sets). We call $\tilde{S}$ *seed octants* for S (Figure 9). If process q receives $\tilde{S}$ as a response instead of o, then it can perform work proportional to $|S|$ in computing S.

The problem then becomes the computation of seeds $\tilde{S}$ to stand in for o as a response to r. The old one-pass algorithm requires no such computation: once it is determined that o is within the insulation layer $I(r)$, it is sent as a response.

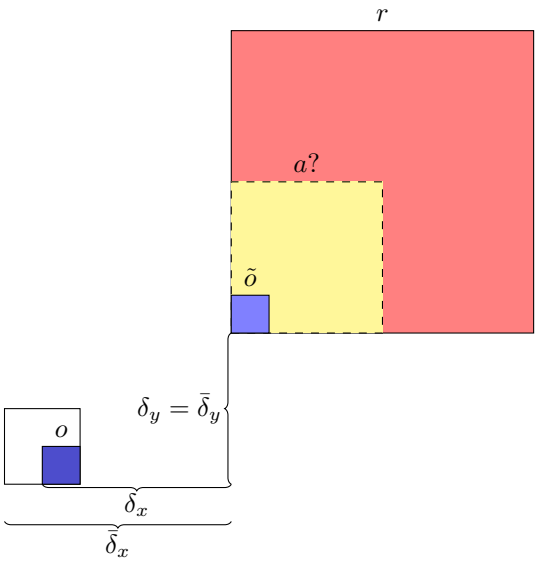


Figure 10. This figure illustrates the problem of determining a, the closest descendant of r that is in $T_k(o)$ and balanced with o. It involves the closest contained same-size octant $\tilde{o}$. The correct size of a can be computed based on the integers δ_x, $\bar{\delta}_x$, δ_y, and $\bar{\delta}_y$ which measure coordinate distances between octants.

Therefore the computation of $\tilde{S}$ should be $\mathcal{O}(1)$ for this replacement to maintain the same algorithmic efficiency.

It can be shown that for every o and r there is a set of seed octants $\tilde{S}$ such that $|\tilde{S}| \leq 3^{d-1}$. Although we do not include a proof here, the sketch of a constructive proof follows.

1) Because $T_k(o)$ can be created in ripple-like propagation outward from o, $\tilde{S}$ can be limited to octants inside of r that touch the boundary of r closest to o.
2) Because octants in $T_k(o)$ grow larger away from o, the closest octant to o in S will be the smallest. Let us call this octant a. We use a balancing algorithm to construct S from $\tilde{S}$, and balancing algorithms do not create octants smaller than their inputs, so a (or one of a's siblings) must be in $\tilde{S}$.
3) If an octant s is in S but is smaller then the overlapping octant in $T_k(a)$, then it (or one of its siblings) must be added to $\tilde{S}$. We can show that this only occurs if s is adjacent to family(a).
4) Starting with $\tilde{S} = \{a\}$, we take each octant s that is in $T_k(a)$, is adjacent to family(a), and touches the same portion of r's boundary as a and we test s against o. If s is unbalanced with o, we can compute the closest octant t in $T_k(o)$ that overlaps s in the same way that we compute a, and then add t to $\tilde{S}$. There are at most $(3^{d-1} - 1)$ octants that match the description of s, hence the limit on the size of S.

The core component of this method is determining a, the closest octant to o in S, in $\mathcal{O}(1)$ time. In the remainder of this section we show how this problem can be solved analytically using arithmetic and intrinsic binary operations on octant coordinates and sizes.

Table II
FUNCTION $\lambda(\bar{\boldsymbol{\delta}})$ SUCH THAT size$(a) = \lfloor \log_2 \lambda \rfloor$

k	1D	2D	3D
1	$\bar{\delta}$	$\bar{\delta}_x + \bar{\delta}_y$	$\texttt{Carry3}(\bar{\delta}_y + \bar{\delta}_z, \bar{\delta}_z + \bar{\delta}_x, \bar{\delta}_x + \bar{\delta}_y)$
2		$\max\{\bar{\delta}_x, \bar{\delta}_y\}$	$\texttt{Carry3}(\bar{\delta}_x, \bar{\delta}_y, \bar{\delta}_z)$
3			$\max\{\bar{\delta}_x, \bar{\delta}_y, \bar{\delta}_z\}$

Let us assume that size(r) > size(o). There exists a descendant $\tilde{o}$ of r that is closest to o and is the same size as o. The octant a we seek is the coarsest ancestor of $\tilde{o}$ that is balanced with o: once we know size(a), $\tilde{o}$'s coordinates can be masked to create a's. See an illustration in Figure 10.

Define $\boldsymbol{\delta} \in \mathbb{Z}^d$ to be the distance vector from o to $\tilde{o}$. We can assume $\delta_x, \delta_y, \delta_z \geq 0$. Our goal is to determine size(a) from $\boldsymbol{\delta}$. If o were replaced with a sibling s, size(a) would not change because $T_k(o) = T_k(s)$: this shows that the important vector is not $\boldsymbol{\delta}$ but $\bar{\boldsymbol{\delta}}$ that maps parent(o) to parent$(\tilde{o})$. We can show that if l = size(o) then the ith component of this vector is $\bar{\delta}_i = 2^{l+1}\lceil \delta_i / 2^{l+1} \rceil$. Vectors $\boldsymbol{\delta}$ and $\bar{\boldsymbol{\delta}}$ are also illustrated in Figure 10.

With these prerequisites we can propose a function $\lambda(\bar{\boldsymbol{\delta}})$ such that size$(a) = \lfloor \log_2 \lambda \rfloor$. Table II gives formulas for λ for $1 \leq d \leq 3$ for each k-balance condition (the function `Carry3` is described shortly).

The 1D result in Table II is fairly intuitive: as we move in one direction outward from o, the octants in $T_k(o)$ double in width as they double in distance from o, so size(a) should be roughly proportional to the logarithm of distance.

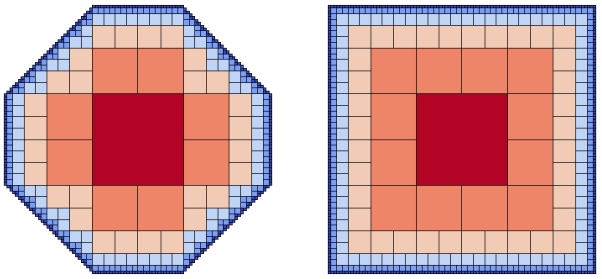
When $k = d$ the coarse neighborhood around an octant is cubic in shape (Figure 5), so that the sets of same-sized octants in $T_k(o)$ maintain concentric rectangular profiles (see Figure 3b). The octant a appears where one of these almost-cubic profiles first intersects r, which depends only on the largest component of $\bar{\boldsymbol{\delta}}$.

When $k < d$, the profiles of the sets of same-sized octants that make up $T_k(o)$ are not so easily described (see Figure 3a), hence the more complicated expressions in Table II. The function `Carry3` in that table is a form of adding three binary numbers that only carries a "1" to the next bit when there are at least three "1"s in the current bit. We only need the most significant bit of this procedure, so we can formulate it using bitwise OR by

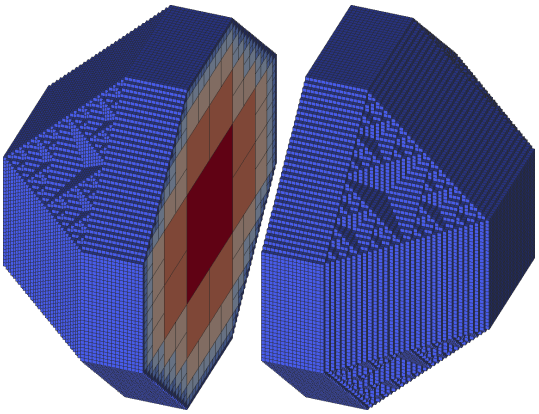
$$\texttt{Carry3}(\alpha, \beta, \gamma) \equiv \max\{\alpha, \beta, \gamma, \alpha+\beta+\gamma-(\alpha|\beta|\gamma)\}. \quad (1)$$

One can prove the correctness of the $k < d$ expressions inductively on the number of significant bits in the components in of $\bar{\boldsymbol{\delta}}$, but these proofs are too technical for inclusion here. We refer to the depictions in Figure 11 which we hope are illuminating nevertheless.

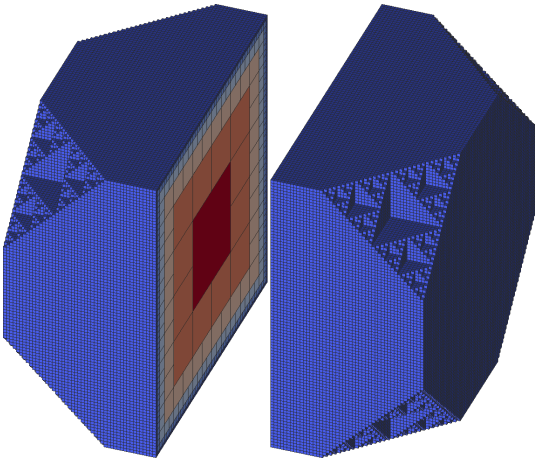
To summarize, the formulas in Table II help to determine a, the closest descendant of r that is in $T_k(o)$. In the same way that a was determined, a small set of octants around a



(a) 2D, 1-balance. (b) 2D, 2-balance.



(c) 3D, 1-balance.



(d) 3D, 2-balance.

Figure 11. This figure illustrates the process of iteratively adding fine balanced neighbors to a set, starting with a single octant: (a) and (b) show five layers of this process for 1- and 2-balance in 2D; (c) and (d) show the same in 3D. 3-balance in 3D is analogous to (b). The octants in each layer are as close as they can be to the central octant without causing it to split. In this sense the layers are like contours of $\lambda(\bar{\boldsymbol{\delta}})$: their coordinates represent values of $\bar{\boldsymbol{\delta}}$ such that if any component is reduced, the value of $\lambda(\bar{\boldsymbol{\delta}})$ in Table II decreases. In 2D these layers resemble the intersection of affine constraints, which is reflected in the values of the table. The cross sections in 3D show that if one component of $\bar{\boldsymbol{\delta}}$ is zero then $\lambda(\bar{\boldsymbol{\delta}})$ behaves like th 2D function of the remaining components for the same value of k. The Sierpinski-type fractal patterns in 3D imply that in the corner regions where all components of $\bar{\boldsymbol{\delta}}$ are nonzero, $\lambda(\bar{\boldsymbol{\delta}})$ cannot be calculated by intersecting affine functions, hence the bitwise operations in `Carry3` (1).

Input: receiver list R for process p

1: $N[0,\ldots,\mathcal{P}-1] \leftarrow$ `Allgather`$(|R|)$ {counts}
2: $O[q] \leftarrow \sum_{q'=0,\ldots,q-1} N[q']$ {offsets for all q}
3: $R[0,\ldots,\mathcal{P}-1] \leftarrow$ `Allgatherv`(R,N,O)
4: **for all** processes q, $q \neq p$ **do**
5: add q to sender list **if** $p \in R[q]$
6: **end for**

Figure 12. Naive implementation of pattern reversal. It determines a list of sending processes from a list of local receivers. The MPI routines correspond to those introduced in the standard [29]. The `Allgatherv` operation requires arrays of counts N and offsets O for all processes.

can be tested to see if they are also in $T_k(o)$. In a new version of the one-pass algorithm, these seed octants can then be sent as a response to r in the place of o. In the *Local rebalance* phase each process matches seed octants it has received to their respective query octants and balances each group separately, rather than rebalancing the whole partition and relying on external auxiliary octants. This change enables the most significant reduction in run time (see the timings in Section VI).

V. Encoding the Communication Pattern

At the beginning of the *Query* stage in the one-pass balance algorithm described in Section II-B every process can determine from its local octant partition which processes it will send messages to. Due to the locality of the space-filling curve this set is of size $\mathcal{O}(1)$ in the average case. The local process is however unable to infer which processes it will receive messages from. This dilemma must be addressed by a scheme to reverse the asymmetric communication pattern: given a list of receivers, determine a corresponding list of senders. Any such scheme necessarily involves communication, and we dedicate this section to describing a parallel algorithm that performs the reversal efficiently.

A naive procedure is sketched in Figure 12. It uses two collective communication calls, the latter of which operates on a variable buffer size. This approach makes use of standard collective MPI routines and is thus easy to implement. However, due to transporting large amounts of unnecessary information it is suboptimal in terms of both message number and volume. It is thus desirable to improve this scheme.

A first improvement has originally been implemented as follows: the senders are encoded in a given maximum number R of ranges by using one `Allgather` call operating on $2R$ integers. `Ranges` still has some drawbacks: the limited number of ranges may lead to the inclusion of processes that are not sending anything, thus creating zero-length messages, and the optimal data volume of the `Allgather` call may not be small (even though it is a fixed number of bytes per process).

We have thus devised a new scheme for pattern reversal based on the divide-and-conquer paradigm, using exclusively

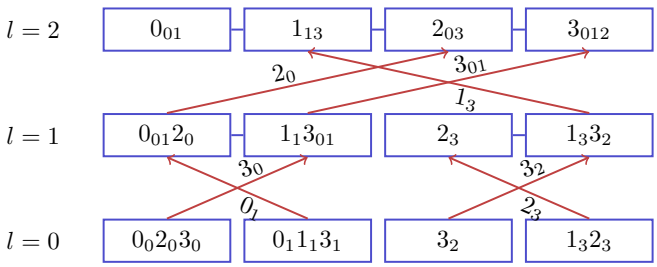


Figure 13. Example of executing the `Notify` algorithm for reversing a communication pattern. It proceeds bottom-up starting at $l = 0$ by crosswise exchange of messages (red arrows) between neighboring process groups (a group is indicated by the blue connecting line). The blue boxes represent processes where the numbers within denote receivers, each subscripted by the respective original senders. To ensure the invariant (2), each process p sends a message to its peer p **xor** 2^l. The end result of all senders to a process is available with the subscripts on the highest l.

point-to-point communication (shown in Figure 13). This `Notify` algorithm proceeds bottom-up to aggregate knowledge about senders in process groups of growing powers of two. The principal invariant is the following: at level l, each process p knows about messages addressed to processes q_i with

$$q_i \bmod 2^l = p \bmod 2^l. \tag{2}$$

Initially, $l = 0$ conforms to process p knowing all of its receivers. The invariant is then maintained by proceeding from $l = 0$ to the maximum l with $\mathcal{P} = 2^l$ (we extend the algorithm to non-powers of two below). At each level, every process exchanges a message with the corresponding process in the neighboring group, which leads to $\mathcal{O}(\mathcal{P} \log \mathcal{P})$ messages. The messages themselves are of variable size (including zero), but their order and sender/receiver pairs are deterministic and computable.

One way to generalize this algorithm to odd process counts would be to have the highest ranked process perform the part of the missing ones that range to the next largest power of two. However, this would introduce an obvious bottleneck in the highest process which in the worst case forces it to process a data size proportional to $\mathcal{O}(\mathcal{P})$ instead of $\mathcal{O}(1)$. Instead, we send to process $p - 2^l$ whenever the original peer does not exist, i.e., p **xor** $2^l \geq \mathcal{P}$. This balances the duplicate messages across peers on all levels while satisfying the invariant (2). We naturally demonstrate this feature in the numerical experiments below, given that the nodes of Jaguar XT5 contain 12 CPU cores each.

VI. Performance Evaluation

We have combined the algorithms proposed in the preceding three sections into a new version of the one-pass 2:1 balance function outlined in Section II-B. We have implemented it within `p4est`, a software library for parallel forest-of-octrees AMR that is publicly available [30]. The code to reproduce our results is included in the tagged revision `ipdps12-submission` and can be invoked by the `timings` example. All timings presented in this section are obtained on the Jaguar XT5 supercomputer at Oak Ridge National Laboratory using up to 112,128 compute cores.

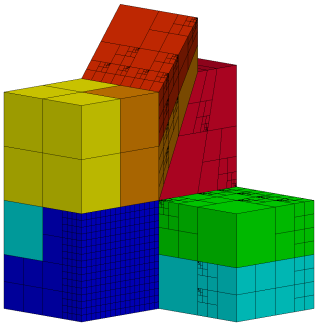


Figure 14. The forest of six octrees used in the weak scaling study, where colors indicate process partitions (the refinement used here is purely illustrative, see the exact refinement rule in the caption of Figure 15).

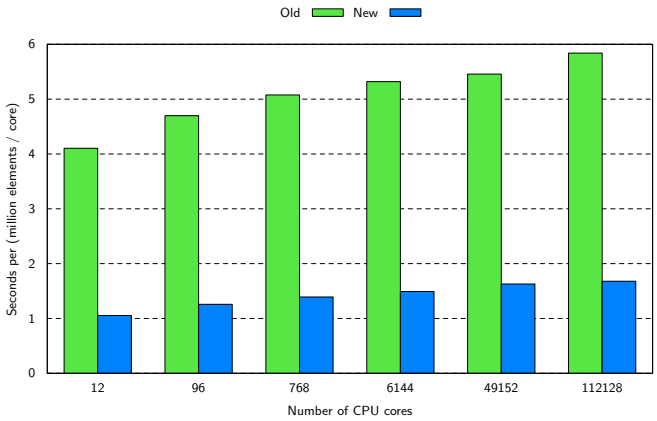
We compare the performance of the new 2:1 balance algorithm to the previous implementation for both isograngular ("weak") scaling on a synthetic example problem with mesh refinement prescribed by a recursive rule, and a fixed-size ("strong") scaling on a mesh of the Antarctic ice sheet with refinement driven by the physics of a finite-element simulation. In both sets of experiments the balance condition used is full corner balance. Both studies show that the new algorithm is faster than the old, often by a wide margin, and that its scalability is as good or better as well.

A. Weak scalability

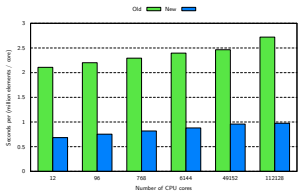
We use a six-octree forest (Figure 14) to study weak scalability for a fractal-type recursive refinement. Using the observation that an increase in the refinement level yields 8 times as many octants, we multiply the core count by 8 for each increment in level. This results in a problem size of approximately 1.3 million octants per core. The exception is the largest problem, which has 5.13×10^{11} octants but is tested on 112,128 cores (4.6 million octants per core). We display the measured runtimes of the full one-pass algorithm and of the component steps in Figure 15. The runtimes rise mildly from roughly 1 second per million octants for 12 cores to 1.7 seconds per million octants for 112,128 cores. Thus the parallel efficiency of the new algorithm is 63% for a 9,344-fold increase in core count. The total time taken for the largest problem is less than 8 seconds. The previously published benchmark for the exact same mesh was 21 seconds using almost twice as many cores [28], thus we achieve a speedup of over 5 on this largest test problem.

B. Strong scalability

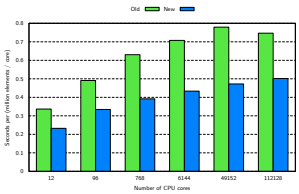
The mesh for the strong scaling study comes from a simulation of ice sheet dynamics. Due to its non-Newtonian properties the creeping flow of glaciers and ice sheets is



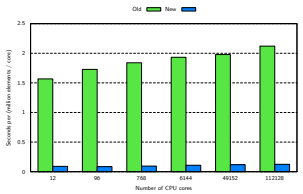
(a) Full one-pass algorithm.



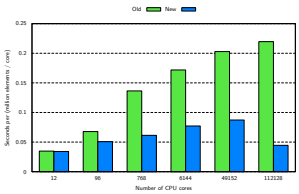
(b) Local balance.



(c) Query and Response.



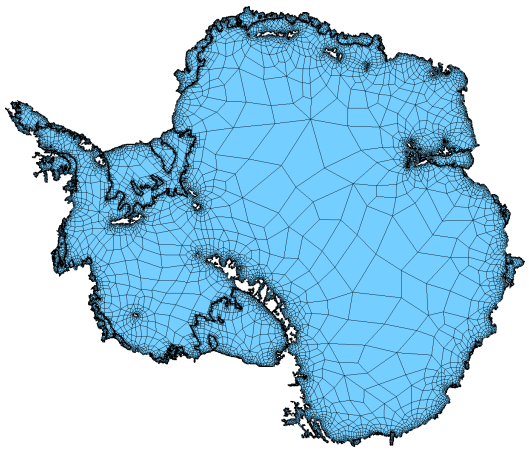
(d) Local rebalance.



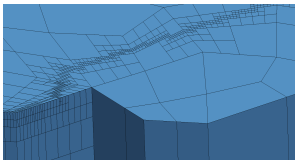
(e) Notify.

Figure 15. Weak scaling results up to 112,128 cores on *Jaguar*. The refinement is defined by choosing the forest in Figure 14, and recursively splitting octants with child identifiers 0, 3, 5 and 6 while not exceeding four levels of size difference in the forest. This leads to a fractal mesh structure. To scale from 12 to 112,258 cores the maximum refinement level is incremented by one while the number of cores is multiplied by 8 (except between 49,152 and 112,128). In each figure the performance of the old and new one-pass algorithms is assessed by normalizing the time spent by the number of octants per core (ideal scaling would result in bars of constant height). For all core counts the speedup for the full one-pass algorithm (a) is between 3.4 and 3.9. Roughly half of this speedup comes from the *Local balance* (b), *Query* and *Response* phases (c); the rest is mostly due to the dramatic 16x speedup in the *Local rebalance* phase, which no longer requires that full partitions be rebalanced. The smaller volume of communication in `Notify` (e) in comparison to `Ranges` (Section V) results in greatly improved scalability when encoding the communication pattern.

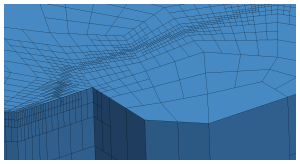
characterized by large regions of low velocity and localized streams of high velocity. The high velocity regions near the exterior of the ice sheet require greater resolution from numerical simulations than does the majority of the ice sheet. Localized stresses that can fracture the ice also develop at or near the boundary between grounded and floating ice. The location of this grounding line and the locations of ice streams shift over time. For ice sheet simulations mesh adaptivity is not a significant portion of computational time



(a) Antarctic ice sheet mesh.



(b) Detail of refinement.



(c) After 2:1 balance.

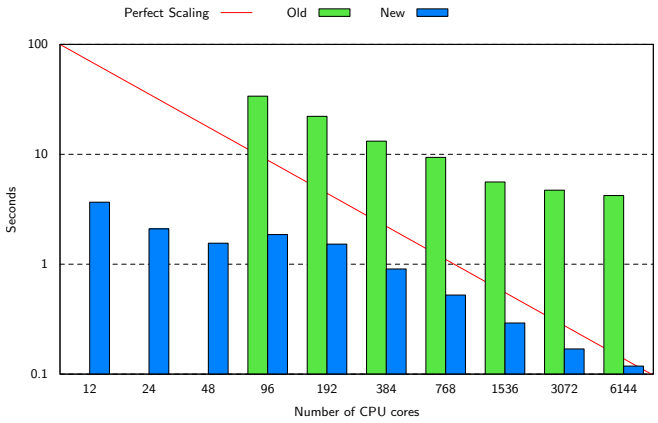
Figure 16. (a) Mesh of the Antarctic ice sheet used for the strong scaling study. The mesh is made up of more than 28,000 octrees. The mesh is refined until all octants that touch the boundary between floating and grounded ice are smaller than a given threshold size. Timings are all for the same initial refinement, which has 55 million octants and grows to 85 million octants once the mesh has been balanced. (b) A detail of the underside of the mesh, where the grounding line reaches the boundary. (c) The same region after the octrees have been balanced.

compared to floating point operations, but this highly graded mesh is a demanding test for the one-pass 2:1 balance algorithm.

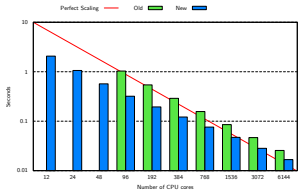
In the strong scaling study conducted on *Jaguar*, this mesh is balanced first on 12 cores, which results in 7 million octants per core, and then for core counts that increase by multiples of 2 until 6,144 cores, which results in 14,000 octants per core. The runtimes of the new and old versions of the full one-pass algorithm and its components are given in Figure 17. At 6,144 cores, the new algorithm balances the mesh in 0.12 seconds, where the old one requires 4.2 seconds. In comparison to the smallest number of cores on which both algorithms could run, the new algorithm scales with twice the parallel efficiency over a 64-fold increase in the number of cores.

VII. Conclusion

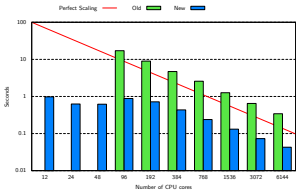
Motivated by the need for large-scale high-resolution simulations of ice sheet dynamics and other geophysical phenomena, we examine a key component of octree-based parallel adaptive mesh refinement (AMR), namely the 2:1 balance algorithm which has a history of being a notorious bottleneck.



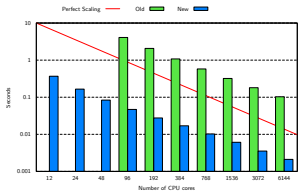
(a) Full one-pass algorithm.



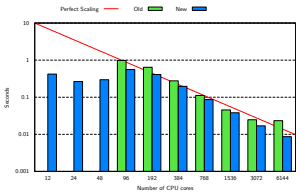
(b) Local balance.



(c) Query and Response.



(d) Local rebalance.



(e) Notify.

Figure 17. Strong scaling results up to 6,144 cores on *Jaguar*, comparing the old and new one-pass algorithms on the Antarctic ice sheet mesh (described in Figure 16). The red line in each plot indicates ideal parallel scalability, where the runtime is inversely related to the number of cores (note the logarithmic scale: lower bars are better). All balance phases exhibit excellent strong scalability, and the new algorithms are predictably faster than the old. In the timings for the *Query* and *Response* phases and for the `Notify` algorithm, we observe that the old algorithm exceeds the 16 GB memory per node and crashes at small core counts, whereas the new algorithm still works. The new *Local rebalance* phase shows the large speedup (nearly two orders of magnitude) versus the old algorithm that we obtained in the weak scaling study.

An analysis of the logical relations between octants suggests several concepts that relate to the performance of the general algorithm. We introduce the notion of octant preclusion and use it to eliminate redundancy and thus to reduce time spent in sorting and searching; we propose $\mathcal{O}(1)$ bitwise functions to exploit size relations between remote octants for fast rebalance; and we add algorithms to decrease the communication message count and volume.

We put the proposed algorithms to the test on the Jaguar XT5 supercomputer at Oak Ridge National Laboratories. To this end we use a synthetic mesh setup for weak scaling studies with up to 5.11×10^{11} octants on 112,128 cores, and a simulation-driven 85-million-octant Antarctica mesh for strong scaling experiments on up to 6,144 cores. We find that weak scaling times improve by a factor between 3 and 4, and that strong scaling improves by one to two orders of magnitude while at the same time requiring much less memory.

Beyond the immediate benefit of increased speed and robustness of the proposed 2:1 balance algorithm, which we make available as free software, we hope to illuminate further concepts behind octree-based AMR, and to provide algorithms (such as `Reduce` and `Notify`) that may be generally useful. Considering the current state of research, we believe that there is still a lot of structure to be found and exploited to reinforce the forest of octrees as a supremely scalable AMR technology.

Acknowledgments

The authors would like to thank Charles Jackson, Andrew Sheng, Georg Stadler, and Lucas C. Wilcox for useful comments and feedback. Tobin Isaac has been supported by the DOE Computational Science Graduate Fellowship. This work has been partially funded by NSF grants ARC-094167 and CMMI-1028889, DOE grant DE-SC0002710, and AFOSR grant FA9550-09-1-0608. We gratefully acknowledge the support by the National Center for Computational Sciences at Oak Ridge National Laboratory and the DOE ALCC GEO014 award for compute time on the *Jaguar* Cray XT5 supercomputer.

References

[1] I. Babuška, J. Chandra, and J. E. Flaherty, Eds., *Adaptive Computational Methods for Partial Differential Equations*. SIAM, 1983.

[2] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of Computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[3] W. Dahmen, "Wavelets and multiscale methods for operator equations," in *Acta Numerica 1997*, A. Iserles, Ed. Cambridge, London, New York: Cambridge University Press, 1997, pp. 55–228.

[4] I. Babuška, U. Banerjee, and J. E. Osborn, "Survey of meshless and generalized finite element methods," in *Acta Numerica 2003*, A. Iserles, Ed. Cambridge, London, New York: Cambridge University Press, 2003, pp. 1–126.

[5] M. J. Berger and R. J. LeVeque, "Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems," *SIAM Journal on Numerical Analysis*, vol. 35, no. 6, pp. 2298–2316, 1998.

[6] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, "Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws," *Journal of Parallel and Distributed Computing*, vol. 47, no. 2, pp. 139–152, 1997.

[7] A. C. Calder, B. C. Curtis, L. J. Dursi, B. Fryxell, G. Henry, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. X. Timmes, H. M. Tufo, J. W. Truran, and M. Zingale, "High-performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors," in *SC '00: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2000, pp. 56–56.

[8] G. L. Bryan, T. Abel, and M. L. Norman, "Achieving extreme resolution in numerical cosmology using adaptive mesh refinement: Resolving primordial star formation," in *SC '01: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2001.

[9] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf, "The Cactus framework and toolkit: Design and applications," in *Vector and Parallel Processing – VECPAR '2002, 5th International Conference*. Springer, 2003.

[10] B. W. O'Shea, G. Bryan, J. Bordner, M. L. Norman, T. Abel, R. Harkness, and A. Kritsuk, "Introducing Enzo, an AMR cosmology application," in *Lecture Notes in Computational Science and Engineering*, T. Plewa, T. Linde, and V. G. Weirs, Eds. Springer, 2004.

[11] J. Luitjens, B. Worthen, M. Berzins, and T. C. Henderson, "Scalable parallel AMR for the Uintah multiphysics code," in *Petascale Computing Algorithms and Applications*, D. A. Bader, Ed. Chapman and Hall/CRC, 2007.

[12] P. Colella, D. T. Graves, N. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen, *Chombo Software Package for AMR Applications. Design Document.*, Applied Numerical Algoirthms Group, NERSC Division, Lawrence Berkeley National Laboratory, Berkeley, CA, May 2007.

[13] C. D. Norton, J. Z. Lou, and T. A. Cwik, "Status and directions for the PYRAMID parallel unstructured AMR library," in *Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2001, p. 120.

[14] O. S. Lawlor, S. Chakravorty, T. L. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. V. Kalé, "ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications," *Engineering with Computers*, vol. 22, no. 3, pp. 215–235, 2006.

[15] J. Luitjens and M. Berzins, "Improving the performance of Uintah: A large-scale adaptive meshing computational framework," in *Proceedings of the 24th International Parallel and Distributed Processing Symposium*. IEEE, 2010.

[16] M. Zhou, O. Sahni, K. D. Devine, M. S. Shephard, and K. E. Jansen, "Controlling unstructured mesh partitions for massively parallel simulations," *SIAM Journal on Scientific Computing*, vol. 32, no. 6, pp. 3201–3227, 2010.

[17] S. Popinet, "Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries," *Journal of Computational Physics*, vol. 190, no. 2, pp. 572–600, 2003.

[18] J. R. Stewart and H. C. Edwards, "A framework approach for developing parallel adaptive multiphysics applications," *Finite Elements in Analysis and Design*, vol. 40, no. 12, pp. 1599–1617, 2004.

[19] T. Tu, D. R. O'Hallaron, and O. Ghattas, "Scalable parallel octree meshing for terascale applications," in *SC '05: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2005.

[20] D. Rosenberg, A. Fournier, P. Fischer, and A. Pouquet, "Geophysical-astrophysical spectral-element adaptive refinement (GASpAR): Object-oriented h-adaptive fluid dynamics simulation," *Journal of Computational Physics*, vol. 215, no. 1, pp. 59–80, 2006.

[21] W. Bangerth, R. Hartmann, and G. Kanschat, "deal.II – a general-purpose object-oriented finite element library," *ACM Transactions on Mathematical Software*, vol. 33, no. 4, p. 24, 2007.

[22] H. Sundar, R. Sampath, and G. Biros, "Bottom-up construction and 2:1 balance refinement of linear octrees in parallel," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2675–2708, 2008.

[23] J. J. Camata and A. L. G. A. Coutinho, "Parallel linear octree meshing with immersed surfaces," in *22nd International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2010.

[24] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, "Algorithms and data structures for massively parallel generic adaptive finite element codes," *ACM Transactions on Mathematical Software*, vol. 38, no. 2, 2011.

[25] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. C. Wilcox, "Extreme-scale AMR," in *SC10: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2010.

[26] C. Burstedde, M. Burtscher, O. Ghattas, G. Stadler, T. Tu, and L. C. Wilcox, "ALPS: A framework for parallel adaptive PDE solution," *Journal of Physics: Conference Series*, vol. 180, p. 012009, 2009.

[27] G. Stadler, M. Gurnis, C. Burstedde, L. C. Wilcox, L. Alisic, and O. Ghattas, "The dynamics of plate tectonics and mantle flow: From local to global scales," *Science*, vol. 329, no. 5995, pp. 1033–1038, 2010.

[28] C. Burstedde, L. C. Wilcox, and O. Ghattas, "`p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[29] M. P. I. Forum, "MPI: A message-passing interface standard," May 1994.

[30] C. Burstedde, "`p4est`: Parallel AMR on forests of octrees," 2010, http://www.p4est.org/.