Vis Comput
DOI 10.1007/s00371-014-1018-2

ORIGINAL ARTICLE

Interpolation and parallel adjustment of center-sampled trees with new balancing constraints

Byungmoon Kim · Panagiotis Tsiotras · Jeong-Mo Hong · Oh-young Song

Abstract We present a novel tree balancing constraint that is slightly stronger than the well-known 2-to-1 balancing constraint used in octree data structures (Tu and O'hallaron, Balanced refinement of massive linear octrees. Tech. Rep. CMU-CS-04-129. Carnegie Mellon School of Computer Science, Pennsylvania, 2004). The new balancing produces a limited number of local cell connectivity types (stencils): 5 for a quadtree and 21 for an octree. Using this constraint, we interpolate the data sampled at cell centers using weights pre-computed by interpolation or by generating interpolation codes for each stencil. In addition, we develop a parallel tree adjustment algorithm, and show that the imposed balancing constraint is satisfied even when the tree is adjusted in parallel. We also show that the adjustment has high parallelization performance. We finally apply the new balancing scheme to level set image segmentation and smoke simulation problems.

Keywords Octree · Quadtree · Balanced tree · Interpolation · Parallelization · Smoke simulation · Segmentation

Electronic supplementary material The online version of this article (doi:10.1007/s00371-014-1018-2) contains supplementary material, which is available to authorized users.

B. Kim
Adobe Systems, San Jose, USA

P. Tsiotras
Georgia Institute of Technology, Atlanta, USA

J.-M. Hong
Dongguk University, Seoul Campus, Seoul, South Korea

O. Song (✉)
Sejong University, Seoul, South Korea
e-mail: oysong@sejong.ac.kr

1 Introduction

Continued increase in media image resolution is powered by the rapid advances in computing technologies, which allow us to enjoy highly detailed visual effects in entertainment films, or to confidently analyze detailed medical images. In many cases, the quality of digital content depends on solving systems of partial differential equations (PDEs), such as when using level set methods to solve fluid equations on high-resolution grids.

It is well known that the memory and computational cost barriers of high-resolution grids can be overcome by adaptive octree or quadtree grids that use high resolution only at those locations demanding high accuracy.

Tree grids store data samples such as velocity, level set values, density, etc. at the *centers* or the *corners* of the grid cells. In corner sampling, data structures for corners are needed. This not only complicates implementation but also consumes a significant amount of memory, often overwhelming the memory occupied by the data samples themselves. For example, suppose we want to represent a 32 bit scalar-valued field using an octree. Since there are 8 corners per cell, and since corners are shared amongst cells, corner sampling would require 8 corner indices (or pointers) (8×32 or 64 bits) per cell. In this case, each cell occupies 8 or 16 times larger memory space than the samples themselves. In contrast, storing the samples at the cell centers does not need corner indices or pointers. We simply need two indices for parents and the first child to represent the tree.

Since corner samples do not have parents or children, it is difficult to average finer level samples and then store the average at a coarser level. Therefore, corner sampling does not favor multi-resolution processing such as those encountered in multi-grid or pyramid algorithms. Another difficulty is that corner samples do not have a sampling area or volume

Published online: 04 September 2014

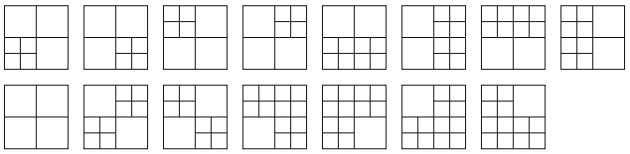


Fig. 1 The new balancing yields only 15 quadtree stencils, only 5 of which are unique under rotational and reflection symmetry

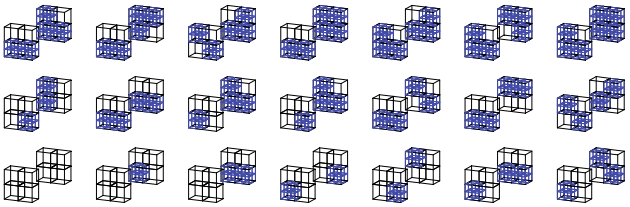


Fig. 2 The new balancing yields only 255 octree stencils, only 21 of which are unique under rotational and reflection symmetry

defined in a natural way. In contrast, a cell-centered sample represents a node in a quadtree/octree structure, naturally.[1] This is useful in performing multi-resolution region-based level set segmentation [7], since one can average image colors or any other statistics in a low-resolution cell that spans multiple image pixels. On the other hand, storing samples at the cell centers makes interpolation hard.

In this paper, we solve this problem by introducing a novel tree constraint that greatly simplifies the number of local connectivity types. The new constraint forces a leaf's same-depth neighbors to be leaves or have leaf children, as illustrated in Fig. 1. This new constraint is, in some sense, a strong 2-to-1 balancing scheme since the 2-to-1 ratio of cell resolutions is applied not only to adjacent cells, but slightly beyond. Therefore, tree resolution varies more smoothly than for a 2-to-1 balanced tree. This produces solution fields (such as smoke and level set fields) that vary more smoothly compared to traditional 2-to-1 balanced trees [31]. To satisfy the new balancing constraint in quadtrees, only 15 local connectivity types (stencils) are produced. Using rotational symmetry, we can further reduce this down to only five types. In the octree case, 255 stencils are possible, but by grouping all rotational symmetries, we obtain only 21 stencil types as shown in Fig. 2. Thanks to this constraint, we could restrict the number of stencils small. For each interpolation stencil, we develop an interpolation procedure in Sect. 3.

In many applications, trees are used to handle very large amounts of data. Recent computers with multiple cores and a large amount of memory, for example 12 physical cores and 48 GB of memory, are becoming in general use. Therefore, shared memory model parallel processing of large octree data is nowadays feasible. In general, the tree adjustment stage is not trivially parallelizable since tree balancing constraints tend to introduce new challenges. For example, subdividing a cell may require subdividing its neighbors to satisfy the constraint. Since another thread may be processing each neighbor, deleting a cell may be deferred; see, for instance, [35]. In this case, it is unclear whether the resulting mesh will satisfy the imposed constraints or not, since a cell and its neighbors can be adjusted by multiple threads, each of which performs the adjustment without knowing the outcome(s) of other threads. In addition, adjusting trees implies up-sampling or down-sampling of the data, which should be performed in an order-independent manner to produce deterministic behavior regardless of the number of threads used. To address these issues, in Sect. 3, we develop a multi-pass parallel tree adjustment algorithm, and show that the algorithm maintains the new balancing constraints.

The novel contributions presented in this paper are: (1) an easy-to-implement center-sampled tree scheme without additional data structure, which naturally suitable for multi-resolution applications; (2) a new 2-to-1 balancing constraint providing significant performance gain; (3) a highly scalable parallel algorithm for tree-balancing on shared memory system that enforces this new constraint.

1.1 Previous work

Storing values at cell corners has been used for implicit representation of shapes using distance fields [12,14,33,34], medical volume image segmentation [2], or in numerical PDE solvers [13,16,20]. In these works, samples stored at the cell centers are not interpolated. Only samples stored at the corners are interpolated.

Interpolation or approximation for a center-sampled tree has also studied previously. In [18], samples were first interpolated at the corners of a cell, which introduced smoothing, and then interpolated at any location in the cell similarly to the corner sampling case. However, the final value computed at the cell center may be different from the center samples. Consequently, this is best characterized as an approximation, rather than an interpolation scheme. Another approach is to use triangulation/tetrahedralization [30]. These are expensive operations of complexity order $O(n \log n)$. Complexity may be reduced to $O(n)$ using precomputed stencils, but this approach still needs a triangular or tetrahedral mesh, thus requiring additional memory along with complex computations to find the specific triangle or tetrahedron that contains the sample point. In contrast, in this paper, we develop a method to interpolate cell-centered samples without using triangulation or tetrahedralization.

Often, when cell-centered samples are used, interpolation across different resolution levels is simply avoided. Instead, uniform resolution is forced at those locations requiring

[1] Whereas the term "node" means a corner of a cell in the previous paper [16], we use the term as a tree node in a quadtree/octree structure. In addition, the term is equivalent to "cell" in our cell-centered tree.

interpolation [3,10,36]. To prevent large accuracy jumps caused by large resolution differences, trees are often constrained by 2-to-1 balancing that allows only one level difference between adjacent cells [16,31]. Although balancing appears to be adding complexity and producing a larger number of cells (some authors indeed do not use balancing constraints at all [8,13,19–21]), the disadvantages of adding constraints are compensated: while constraints complicate the tree adjustment stage, they simplify interpolation, differentiation, finding the neighborhood, and many other operations. In addition, the increase in the number of cells due to balancing is often insignificant, since 2-to-1 balancing coarsens cells to one level, which already reduces the number of cells down to 1/8. This leaves small room for improvement by coarsening to even lower resolution cells.

Parallelization of balanced trees has also been studied previously. In [35], the modification of nodes in the tree is deferred for later steps. In this paper, we use a similar approach. However, since we introduce a new balancing scheme, it is unclear whether this balancing scheme is not hampered by the parallelization. In Sect. 3, we show this is indeed true.

Parallelization of balanced tree adjustment for non-shared memory models has been studied in [31]. In the non-shared memory model case, the 2-to-1 balancing constraint introduces another problem: recursive refinement to neighbors causes *ripples* that propagate outside the domain of a processor core. In [31], after observing that ripple sets are typically small, the ripple region is loaded into the memory and then processed. In contrast, in the shared memory case, where entire domain is loaded in the main memory, ripples are not problematic.

The preliminary version of this work was presented in [15]. In the current article, we give a significantly improved exposition of the technique as well as extend the technique: (1) The analysis of the proposed tree-balancing constraint is now more rigorous, and as a result, greatly simplifies the number of local connectivity types (stencils), from 255 to 21 only, in octree data structures. (2) By developing a new parallel tree adjustment algorithm, and proving that the constraint is satisfied even when the tree is modified in parallel, we improve the scalability of the algorithm and its applicability. (3) In addition, we perform a new large-scale smoke simulation experiment to demonstrate that the proposed method has high parallelization performance.

2 Octree/quadtree constraints

While trees with constraints have been used in the past, little attention has been paid to the number of resulting stencils. In this paper, we observe that a tree grid with a small number of stencils may allow the design of the interpolation of cell-center samples. Based on this idea, we develop a new constraint that yields a small number of stencils.

2.1 Octree/quadtree constraints

Our first goal is to limit the number of stencils. To this end, we propose the following constraint:

$$\text{Leaf cell's same-depth neighbors are } \begin{cases} \text{leaves, or} \\ \text{parents of leaves.} \end{cases} \quad (1)$$

Same-depth neighbors are illustrated in Fig. 3: in (a), the cells L and E have same-depth neighbors marked by S. In (b), (c), and (d), the green cell's same-depth neighbors are the blue cells.

Note that (1) implies:

1. A leaf's same-depth neighbor may not exist in all directions. For example, in Fig. 3a, the leaf L has neighbors C that are leaves, but have depth one less than L.
2. A cell's same-depth and coarser neighbors enclose the cell, constructing the Moore neighborhood.
3. A coarser neighbor of a leaf should only be one resolution coarser. For example, in Fig. 3a, leaf L has coarser neighbors C, which have only one depth less. This can be seen by applying (1) to C. Since C's same-depth neighbor should be the parent of the leaf L, and must not be a parent of any non-leaf, C can only be one level coarser than L.

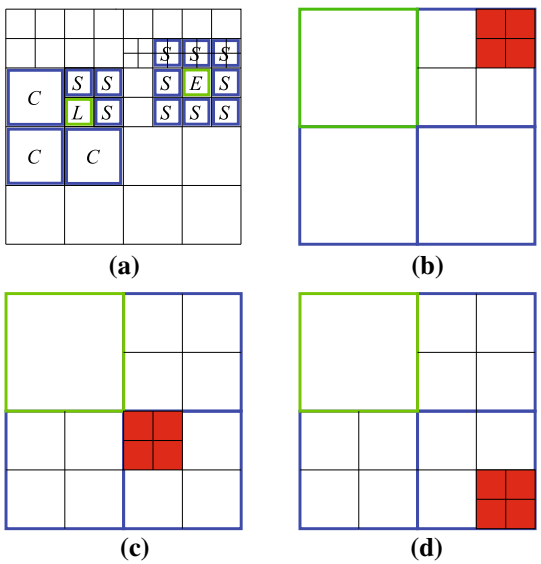


Fig. 3 **a** a quadtree connectivity that satisfies constraint (1). The leaves L and E have same-depth neighbors, denoted by S. Since S are leaves or have leaf children only, L satisfies the constraint (1). In contrast, the stencils in **b**, **c**, and **d** are not valid, since the *green* cell's same-depth neighbors are the three *blue* cells, one of which contains a non-leaf child (in **d**, for example, the lower right child is not a leaf, but has four *red* children)

4. 2-to-1 balancing is guaranteed. Only one depth difference between adjacent cells is possible. For example, in Fig. 3c the green and the red cells have depths differing by two. This is not allowed, since the green cell's same-depth neighbors (the blue cells) that contain the red cells are not parents of leaves, hence (1) is violated.
5. Only one depth difference is allowed between a cell and the neighbor's children. For example, the red cells in Fig. 3d violate (1) even though resolution changes by one between all neighboring pairs.

As a result, the constraint (1) produces a small number of stencils, which for a quadtree are illustrated in Fig. 1. In all quadtree nodes represented by cells, we can find the four neighboring same-depth nodes that are either leaf nodes or have four leaf children. Since each of the four nodes can have children, there can be 16 different cases. However, if all of the four nodes have leaves, then the stencil is equivalent to the case in which all of the four nodes are leaves. Therefore, there are 15 different local stencils. Similarly, for an octree, we can always find eight nodes that are either leaf nodes or have leaf children. Therefore, there exist 255 different stencils. The number of stencils can be further reduced down to 21 by removing symmetric cases. Thus, only a small number of stencils are possible.

The implementation of a tree adjustment algorithm satisfying (1) is not trivial. In particular, note that if we refine a cell, the resulting cell may break the constraints until neighbors are refined. In addition, the adjustment must be independent of the order the tree nodes are visited. In addition—and more importantly—the adjustment algorithm should be easily parallelizable. To meet these goals, we develop an efficient, yet simple-to-implement, tree adjustment strategy in Sect. 4.2.

3 Interpolation on quadtree and octree grids

3.1 Interpolation

In an octree or a quadtree with the constraint (1) applied, we consider how to interpolate cell-centered data. We use axis-aligned interpolation boxes. In the interpolation subroutine, we first find an interpolation box that contains the sample, compute the values at the corners of the interpolation box, and then perform bilinear or trilinear interpolation. For example, in Fig. 4a or b, suppose we want to interpolate values at the Y mark. By comparing the coordinates at Y and the coordinates of the neighboring cell centers, we immediately know that bilinear interpolation can be done in the shown green box, and we can perform bilinear interpolation with the four samples at the red corners of the green box.

To interpolate at the point marked by X in Fig. 4a, we split the blue box as shown by the black dotted lines. This

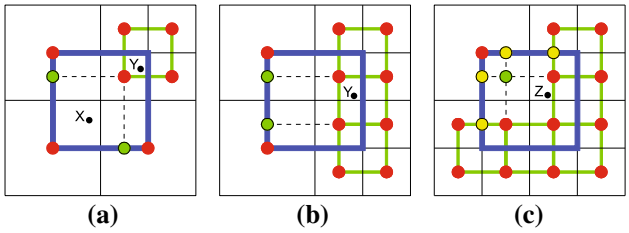


Fig. 4 Various interpolation boxes and their sub-boxes. The *red circles* are cell centers that hold sample values, which are interpolated to the*green circles*during the interpolation. At the points Y in **a** and **b**, bilinear interpolation is performed in the small *green boxes* that contain Y. At the point X in **a**, bilinear interpolation is performed in the lower-left sub-box of the *blue box*. At the point Z in **c**, bilinear interpolation is performed in the centered sub-box of the *blue box*

step produces the three sub-boxes shown in Fig. 4a. Again, by comparing the coordinates of the point X and the dotted lines, we can identify the sub-box that contains the point X. However, to perform bilinear interpolation in this sub-box, we must compute the interpolated value at the green circles in Fig. 4a as weighted sums of the values at the red circles. We simply pre-compute these interpolation weights for the green markers for all the 15 or 255 different box types for quadtree or octree, respectively. In this way, we can quickly compute the values at the green circles. Now we have all values at the four corners of the sub-box that contains the interpolation point. Finally, we can perform bilinear interpolation in each sub-box. The interpolation steps are as follows, for example, at X or Z:

1. Consider the location marked by X in Fig. 4a. Since the location is in the upper right corner of the cell, we look at the neighbors at the right and upper sides and choose the blue interpolation box. In contrast, if we want to interpolate at the point Z in Fig. 4c, which is in the lower left inside the smaller cell that contains Z, we cannot simply look at the lower left neighbors. Instead, we look at the lower left neighbors of the parent.
2. Check whether the four cells at the four corners of the interpolation box are leaves or not. Then, for the point X in Fig. 4a, we can identify that the connectivity type is the third one in Fig. 1.
3. From the relative location inside the blue interpolation box, we identify the sub-box that contains the interpolation point. Using the pre-computed interpolation weight table entry that corresponds to the stencil, we can compute the interpolated values at the corners of the sub-box.
4. By performing bilinear interpolation in the interpolation sub-box, we can compute the interpolated value.

This interpolation is continuous, since interpolation (sub-)boxes do not overlap, and interpolations are consistent

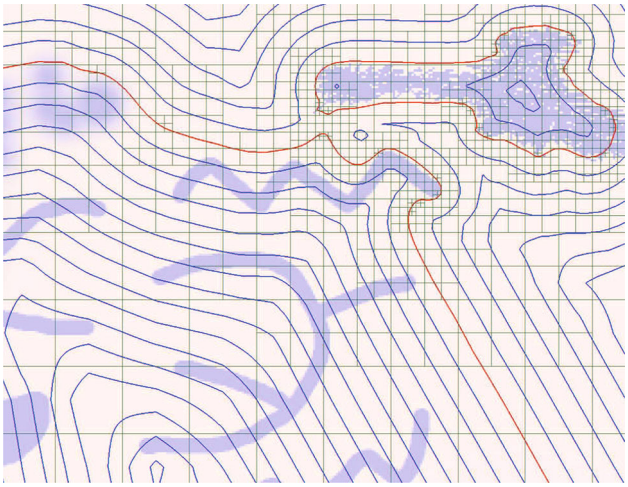


Fig. 5 A snapshot of a level set in the middle of segmentation of the background image with various *blue shapes*. *Red and blue lines* are isocontours (the zero level set $\{\phi = 0\}$ on *red lines*, i.e., the interfaces). Level set values sampled at cell centers are interpolated continuously. This continuous interpolation produces smooth contour lines, which are continuous across differently sized cells

at the (sub-)box boundaries. See Fig. 5 for an interpolation example. As shown in this figure, interpolation is continuous across different resolutions. In particular, the interpolated contours of the zero level set $\{\phi = 0\}$, i.e., the interfaces, shown with red lines, are a high quality curve, which are efficiently represented by the proposed adaptive grid.

3.2 Implementation and performance

We have implemented the proposed interpolation in three different ways. The first method (Method I) performs the interpolation in a single function, in which we compute the stencil type and sub-box location, and then perform the bilinear interpolation using interpolation weights, which are precomputed for each stencil and sub-box. This method can be implemented using a code shorter than Method II or III, but needs a large weight table to handle all cases. The second method (Method II) does not use a weight table to handle all cases, but rather uses individual interpolation functions for all stencil types (15 for quadtree and 255 for octree). Since interpolation weights are embedded in the code, and the handling of each stencil type is hard-coded, the code is more optimal than the one of Method I. However, code size can be large in the octree case, and the instruction cache miss rate can be high. Therefore, we designed a third method (Method III) that reduces the code size using rotational symmetry. To this end, we first map the interpolation coordinates to 5 or 21 stencil types, and then perform the interpolation for each of them, similarly to Method II. Since we have 5 or 21 types only, instead of 15 or 255, the resulting code size and the number of branches are greatly reduced. As shown in Fig. 6, Methods II and III show the best interpolation performance. Method II is better in the quadtree case. In the octree case, Method II is slightly better than Method III in cache-warmed case, but Method III is better than Method II

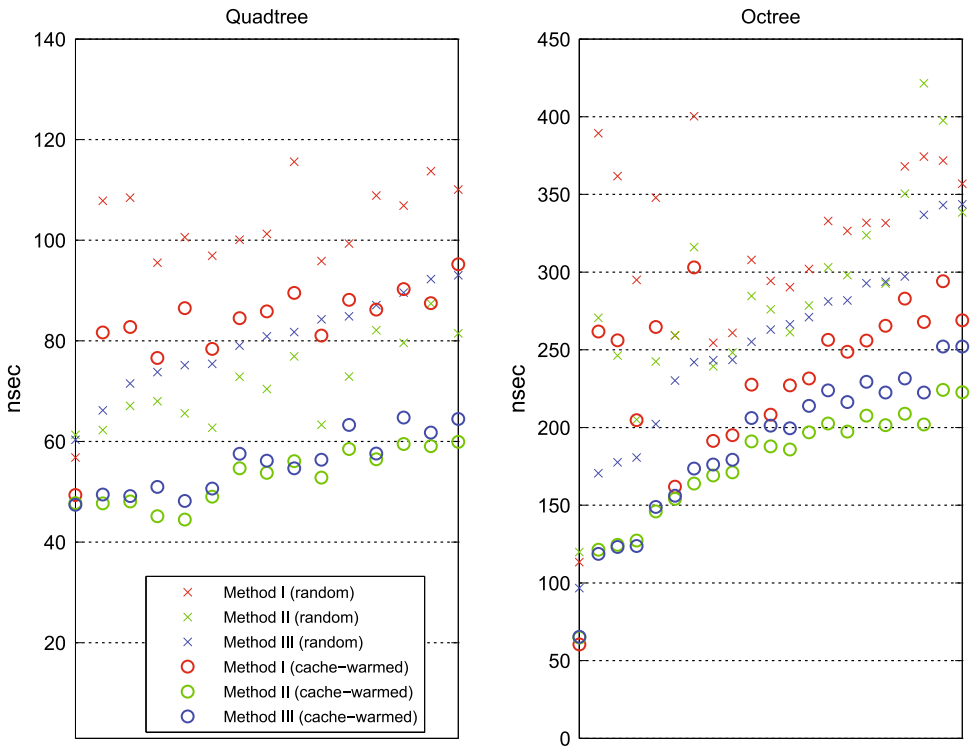


Fig. 6 Interpolation performance for different stencils. *X*-axis locations are 15 quadtree (*left*) and 21 octree (*right*) stencils. The performance is measured using Xeon 2.66 GHz CPU

for random sampling. For warm-up the cache, we measured the second interpolation at the same location.

We tested the proposed interpolation scheme in a smoke advection setting with CFL = 4. Since CFL is large, a large number of semi-Lagrangian samples are located outside the cell. This requires traversing to a parent that contains the semi-Lagrangian sample and then traverse down to a leaf that contains the sample. Thus, on average, 4.69 levels were traversed in the smoke advection test. Interestingly, since expensive stencil types are rather rare in practice, the interpolation time per sample was 167 nanoseconds on average. This is about 11 times slower than in the uniform grid case. The result for trilinear interpolation on uniform 3D array was 15 nanoseconds (cache-warmed case). In a similar 2D test, we achieved 11 nanoseconds for bilinear interpolation on a 2D uniform grid. Since we achieved about 50 nanoseconds for quadtree interpolation, it is seen that interpolation on quadtrees is about five times slower than uniform grid interpolation.

All the previously quoted results are with the same (or fewer) depth neighbors precomputed. Specifically, we precomputed four neighbors in the quadtree and six neighbors in the octree. We also precomputed the interpolation stencil type per cell. The neighbor table consumes four or six integers for a quadtree or an octree node, respectively. Stencil types require half or a single byte per quadtree or octree node, respectively. We can save memory overhead by computing the neighbors and the interpolation stencil type on-the-fly inside the interpolation routine. This adds an additional tree traversal cost to parents and children. By computing neighbor and stencil types on-the-fly, we obtained 236 nanoseconds. With only stencil types precomputed, we obtain 218 nanoseconds, and with only neighbors precomputed, we obtained 177 nanoseconds. With both the neighbors and stencil types, we obtained 167 nanoseconds.

We precomputed neighbors and stencil types for fluid simulations since the additional memory required for neighbors and stencil types is relatively insignificant since the simulation needs space for multiple scalar and vector variables, temporary variables, and sparse matrices. In contrast, to represent a single scalar field such as in a level set case, one may choose not to use precomputed neighbors to reduce the memory cost.

4 Tree adjustment

Recall that explicit time integration of PDEs is known to be trivially parallelizable, especially on regular grids. However, this is no longer true for adaptive grids; in fact, this is an area of active research [25]. For adaptive grids, the first problem is to decompose the tree domain so that each processor core gets a similar number of nodes to process. This problem can be solved by mapping all tree nodes into an 1D sequence and then decompose that sequence in equal length [24,26]. Once the domain is decomposed, the grid adjustment step must be parallelized. Parallel grid adjustment is not trivial since each core must refine/coarsen a grid cell, while maintaining the tree constraints, for which a CPU core must check neighboring grid cells and refine them, if necessary. However, the neighboring grid cell may have been assigned to a different CPU core, and hence must not be refined or coarsened while the other CPU core is working on that cell.

4.1 Criteria for refining and coarsening

Mesh resolutions can be computed in various ways depending on the application. In level set applications, the resolution may be a function of distance to the interface, and/or the curvature at the interface; in smoke simulations, the resolution can be a function based on smoke density and density gradient. Based on a criterion that fits the application needs, one can compute the desired tree depth d_{desired} at every leaf node. Then, one can refine the tree so that every leaf node satisfies $d \geq d_{\text{desired}}$.

4.2 Refining and coarsening with constraints

The algorithm coarsens a cell only if the balancing constraint is not violated. On the other hand, if the desired grid resolution at a cell's center is greater than the current resolution of the cell, we always refine that cell, and force the constraints by refining its neighbors, if necessary. Thus, we refine a cell whenever the cell needs higher grid resolution, but coarsen a cell only if the constraint (1) is satisfied after the coarsening.

Suppose a cell C is a leaf-parent with all children having $d_{\text{desired}} < d$, and hence C may be coarsened. In this case, to check if C can be coarsened or not, as shown in Fig. 9, we check whether a cell's same-depth neighbors are leaves or leaf-parents. As illustrated in Fig. 7, if a cell C has a same-depth neighbor L that is a leaf, then C can be coarsened without breaking the constraint, since L has depth lower or the same as C. Therefore, children of C can be merged, making C a leaf cell. On the other hand, if C has a same-depth neighbor P_L that is a leaf-parent, C can also be coarsened

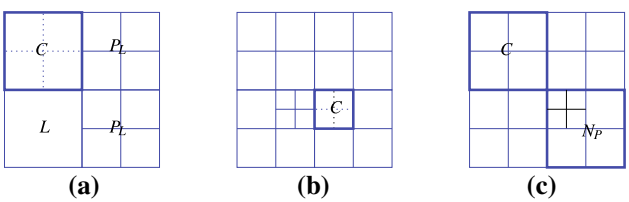


Fig. 7 In **a** and **b**, the cell C can be coarsened since the same depth neighbors are leaves or parents of leaves. In **c**, the cell C cannot be coarsened since a same depth neighbor N_p has a non-leaf child

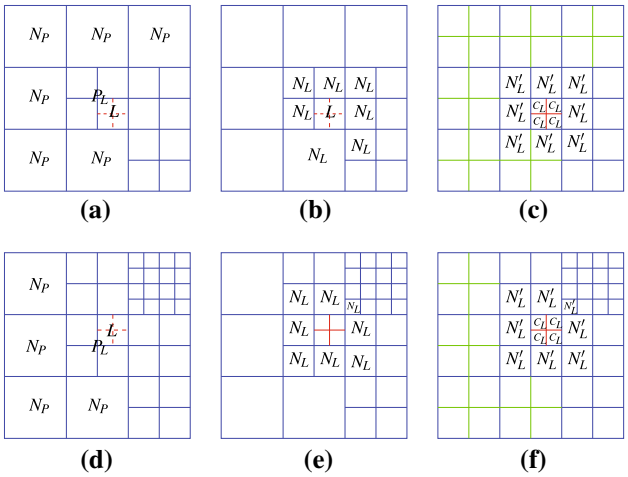


Fig. 8 **a** To satisfy the constraints (1), when a leaf L is refined, we must refine the cells N_P that are non-leaf neighbors of P_L (the parent of L). **b** Neighbors of L before refining L and N_P. **c** Neighbors of L after refining L and N_P. **d**–**f** are other examples

(i.e., the children of C can be merged) since the depth difference between C and the children of P_L will be one. In contrast, in Fig. 7c, the cell C cannot be coarsened, since the same-depth neighbors N_P are not a leaf or a leaf-parent.

In contrast to coarsening, we always refine a leaf L if higher grid resolution is needed ($d_{\text{desired}} > d$). To enforce the constraints, we must force-refine the neighbors. As illustrated in Fig. 8, let P_L be the parent of L. When L is refined, we refine P_L's same or lesser-depth leaf neighbors. As shown in Fig. 10, the refinement is recursively applied, i.e., when we refine a neighbor of P_L, we refine the neighbors of the parents of P_L's neighbor. Therefore, at each recursion, forced refinement depth decays by one.

Note that we refine only one depth at a grid adjustment pass. This is a negligible limitation of the algorithm since PDE solvers take a large number of time steps, while the depth of the tree is a small number, such as 10 for a 1024^3-equivalent grid. If the coarsest resolution is 5, at a location where the highest resolution is desired the maximum depth can be reached in 5 time steps since we simply perform the adjustment every time step.

With the algorithm in Fig. 9, we coarsen cells only if same-depth neighbors are leaves or leaf-parents. After coarsening, the new leaf has same-depth neighbors that are leaves or leaf parents, still satisfying (1). We now show that (1) holds after a tree is refined by Fig. 10.

Theorem 1 (Tree Constraint) *Let a tree T_0 satisfy the balancing constraint* (1)*. Let T_1 be T_0 refined by the algorithm in Fig. 10. Then, T_1 satisfies* (1)*.*

Proof Suppose we are refining a leaf L. Let the set of leaves in the neighborhood of a cell C be $\mathcal{N}(C) = \{\text{leaves in region covered by } C \text{ and } C\text{'s same-or-lesser depth neighbors}\}$. For

```
coarsen()
  if (isLeafParent()==false) return
  for each same-depth neighbor
    if (neighbor.isLeafOrLeafParent()==false) return
  deleteChildren()
```

Fig. 9 By coarsening only if same-depth neighbors are leaves or have leaf children, the tree does not break the balancing constraint. Refer to Sect. 5.1.4 for deferred deletion strategy for parallelization

```
refine()
  createSubcells();
  treeNode p = getParent()
  for each same-depth neighbor of p
    if (neighbor.isLeaf()) neighbor.refine()
```

Fig. 10 By refining current cell and recursively force-refining parent's leaf neighbors, the tree holds the balancing constraint by Theorem 1

example, in Fig. 8a or d, $\mathcal{N}(P_L)$ is the set of all leaves in the domain.

Let P_L be the parent of L. Since P_L is a leaf parent, P_L cannot have a coarser leaf neighbor (see Fig. 8a or d). In addition, leaves inside P_L have the same depth as L. Therefore,

$$\text{for all leaves } C \in \mathcal{N}(P_L), \quad \text{depth}(C) = \begin{cases} \text{depth}(L) - 1, & \text{or} \\ \text{depth}(L), & \text{or} \\ \text{depth}(L) + 1. \end{cases} \tag{2}$$

After L is refined by the algorithm in Fig. 10, all leaves $C \in \mathcal{N}(P_L)$ with $\text{depth}(C) = \text{depth}(P_L) = \text{depth}(L) - 1$ will be refined. Therefore, all leaves in $\mathcal{N}(P_L)$ will have depth greater or equal to L, i.e., after refinement,

$$\text{for all leaves } C \in \mathcal{N}(P_L), \quad \text{depth}(C) = \begin{cases} \text{depth}(L), & \text{or} \\ \text{depth}(L) + 1. \end{cases} \tag{3}$$

For example, in Fig. 8c and f, all leaves have depth the same as or one more than L.

Let N_{C_L} be a leaf neighbor of C_L. Since $N_{C_L} \in \mathcal{N}(P_L)$, (3) is applied:

$$\text{depth}(N_{C_L}) = \begin{cases} \text{depth}(L) = \text{depth}(C_L) - 1, & \text{or} \\ \text{depth}(L) + 1 = \text{depth}(C_L). \end{cases} \tag{4}$$

Therefore, if $\text{depth}(N_{C_L}) = \text{depth}(C_L)$, N_{C_L} has the same-depth leaf C_L, and if $\text{depth}(N_{C_L}) = \text{depth}(C_L) - 1$, then N_{C_L} has the same-depth neighbor L that is a leaf-parent. In addition, C_L has the same or one lesser depth leaf neighbors only. Therefore, C_L's same depth neighbors are leaves or leaf parents. Thus, the constraint (1) is satisfied between C_L and its neighbors.

5 Parallel and consistent tree adjustment with continuous interpolation

In this section, we develop a tree adjustment algorithm that is easy to be parallelized and has consistent adjustment that is independent of the tree-visit order. In addition, the values at refined cells are obtained by continuous interpolation.

For parallelization, a computational domain should be mapped to a one-dimensional list by space filling curves (SFC) [26]. The list is then divided and assigned to multiple threads. We list the tree nodes in the depth first order, and then divide the list by the number of threads. Since computations are performed on leaf-only, we divide the list in multiple ways so that decomposed domains contain the same number of leaves.

5.1 Parallel and consistent mesh adjustment

Cell-centered trees can be coarsened by merging children, or refined by subdividing a cell into children. Since such adjustments should satisfy the balancing constraint that requires reading from neighborhoods, and since subdividing requires interpolation at each child's center, the mesh adjustment step is a computationally expensive step, often becoming a performance bottleneck. Therefore, parallel refinement can greatly improve the overall performance. However, the mesh adjustment step is not trivial to be parallelized. When a cell is refined, the cell may require refining a neighboring cell, which may belong to a different core, and conversely, a core may delete a cell that another core is processing. To resolve this difficulty, we use a multi-pass adjustment strategy, rather than adjusting the tree in a single pass. An important question is whether the constraint (1) is satisfied or not after the parallel refinement.

5.1.1 Pass 1: criteria evaluation

In the first pass, each thread scans its domain, computes d_{desired} and marks cells as *merge* if $d > d_{\text{desired}}$, or *refine* if $d < d_{\text{desired}}$. When we mark a cell as *refine*, we also visit this cell's neighbors by the algorithm in Fig. 10, and if a neighbor must be refined to satisfy the constraint, we mark the neighbor as *refine* as well. Of course, this procedure may continue to distant neighbors until no cell violates the constraints, as shown in Fig. 10. This recursion normally occurs in a small number of cells [31], but this recursion makes each thread mark cells in other thread's domain. Thus, cells near the domain boundary may be marked multiple times by different threads. However, this does not cause any problem since marking is a write-only operation. Therefore, the first pass is trivial to be parallelized. Once this pass is done, we can refine/coarsen the cells.

5.1.2 Pass 2: data interpolation and child generation

The second pass is to perform the interpolation and create new children nodes or subcells. Therefore, this pass is also trivial to be parallelized. In this pass, we do not yet insert the nodes to the tree. This is an important part of the algorithm that allows us to perform a consistent and continuous interpolation. Otherwise, if we insert newly created nodes in the second pass, the tree balancing constraint may be temporarily broken until we refine all neighbors to reinforce the constraints. The dilemma is that we cannot subdivide neighboring cells using the continuous interpolation. Even if we can solve this problem, interpolated values at new cells will depend on the order in which we visit tree nodes, since interpolation will use values at the new cells. Then, interpolation results will change slightly in a non-deterministic manner due to parallelization. Therefore, we propose to use a separate third pass that actually inserts the children generated in the second pass to the tree.

5.1.3 Pass 3: insertion of children

In this pass, we insert the new children nodes generated from pass 2 to the tree. After this pass, the tree is finally modified (refined). The question is whether the above three-stage refinement passes 1, 2, and 3, each of which is performed in parallel, still maintain the constraint (1) or not. We now show this is indeed true.

Theorem 2 *Let a tree T_0 satisfy the constraint* (1)*. Let the tree T_1 be T_0 refined by the three-stage refinement passes 1, 2, and 3. Then, T_1 satisfies* (1)*.*

Proof Suppose a leaf L_0 meets the refinement condition $d < d_{\text{desired}}$. We want to refine L_0. In addition, we search the neighborhood of L_0 to refine neighboring cells denoted by $L_{N_0}, L_{N_1}, \ldots$, to enforce the constraints (Theorem 1). Let the set of all such cells we want to refine by

$$\mathcal{R}_0 = \{L_0, L_{N_0}, L_{N_1}, \ldots, L_{N_m}\}. \tag{5}$$

Similarly, suppose that another leaf L_1 satisfies the refinement criteria $d < d_{\text{desired}}$. For this leaf, we can also find $\mathcal{R}_1$ that contains all leaves that must be refined to satisfy the constraints. We can continue this process and imagine all $\mathcal{R}_i$ for all L_i, where $i = 0, 1, 2, \ldots, n$. Notice that each $\mathcal{R}_i$ is computed from the tree T_0, and $\mathcal{R}_i$ and $\mathcal{R}_j$ may overlap.

The first pass is to compute all the cells that must be refined, i.e., the first pass computes $\mathcal{R}$ from

$$\mathcal{R} = \mathcal{R}_0 \bigcup \mathcal{R}_1 \bigcup \cdots \bigcup \mathcal{R}_n. \tag{6}$$

Note that, in the first pass, cells belonging to an overlap, for example $\mathcal{R}_i \bigcap \mathcal{R}_j$, will be marked twice, but this is not

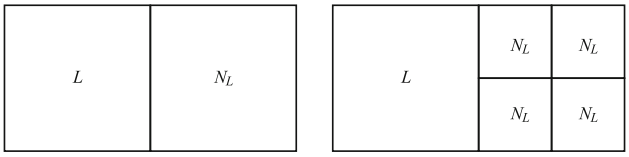


Fig. 11 When a leaf L is refined, if L's neighbor N_L is not force refined, then depth(N_L) $\geq$ depth(L)

different from being marked once. At the end of the first pass, we only know $\mathcal{R}$, and do not have any information on $\mathcal{R}_i$.

We now want to show that refining all cells in $\mathcal{R}$ satisfies the constraint (1). First, note that for an arbitrary j, refining cells in $\mathcal{R}_j$ satisfies the constraint by construction (Theorem 1). However, since some neighbors of $\mathcal{R}_j$ can be refined by $\mathcal{R}_k$ for some k, the question is whether $\mathcal{R}_j \bigcup \mathcal{R}_k$ still satisfies the constraints (1) or not.

Consider a leaf $L \in \mathcal{R}_j$ (e.g., the cell L in Fig. 8b or e). Suppose that a neighbor of L, denoted by N_L, is refined by $\mathcal{R}_k$ only, i.e., $N_L \notin \mathcal{R}_j$ and $N_L \in \mathcal{R}_k$. Since $N_L \notin \mathcal{R}_j$, the depth of N_L should be the same as or one more than the depth of L as illustrated in Fig. 11. When N_L and L are refined, we have

$$\text{depth}(C_{N_L}) = \begin{cases} \text{depth}(C_L), & \text{or} \\ \text{depth}(C_L) + 1, \end{cases} \tag{7}$$

where C_L and C_{N_L} are the new children of L and N_L, respectively. Thus, C_L's same depth neighbor will be N_L (leaf-parent) or N_L's children (leaves), and C_{N_L}'s neighbor will be C_L. Therefore, cells refined by $\mathcal{R}_k$ does not violate the constraint (1) for the cells in $\mathcal{R}_j$. Since j and k are arbitrary, the constraint (1) is still satisfied after all cells in $\mathcal{R}_j \bigcup \mathcal{R}_k$ are refined. Since we can union $\mathcal{R}_j \bigcup \mathcal{R}_k$ with any other $\mathcal{R}_i$ for any i still holding the constraint (1), refining cells in the union set $\mathcal{R}$ satisfies the constraint (1).

Thus, the constraint (1) is satisfied by Theorem 2. Moreover, since each pass 1, 2, or 3 depends only on T_0, and not on any intermediate results in the pass, the refinement passes 1, 2, and 3 are order independent. Therefore, the refinement passes 1, 2, and 3 are also trivial to be parallelized, and produce consistent results. We experimentally verify the balancing constraint while a massive tree with more than 40 million cells is adjusted, creating 7.2 million cells and deleting 4.4 million cells.

5.1.4 Pass 4: coarsening

Finally, in the fourth pass, we scan the tree to coarsen cells that are marked 'merge' in the first pass. In the fourth pass, we check if the merging violates the constraints or not.

Consider a cell C marked *merge* and its neighbor N, which is also marked *merge*. Note that merging children of C do not require clearing the *merge* mark for N. Therefore, while a thread is checking for the constraint (1) for a cell and merging its children, any other thread can merge any cells. Thus, merging is also trivial to be parallelized. However, unlike the refinement step, coarsening may not be order independent if cells are merged in each thread, since after coarsening a cell, i.e., merging its children, a neighboring cell that could not be merged due to the constraint (1) may be merged. Moreover, in practice, we found that deleting children nodes in each thread can invalidate the domain decomposition data structure that contains starting and ending nodes for each thread. In addition, since deletion should enter a critical section in the memory manager, contention occurs. Therefore, during merging, we only do computations such as averaging children's sample values and store them to the parent. During coarsening a cell, we do not delete its children; instead, we just collect indices to cells to be deleted in an array dedicated to each thread, and then delete the cells later in serial in the main thread. This eliminates the need of a critical section, and simplifies the parallelization domain handling. In addition, the deletion becomes order independent. For example, we acquired exactly the same number of cells created and deleted always and regardless of the number of threads. See the Sect. 5.2 below.

In summary, actual tree adjustments are performed at the final two passes. This multi-pass strategy plays an essential role: it makes the adjustment independent of the order we visit the tree, it keeps the tree connectivity unchanged while the continuous interpolation is performed, and it makes parallelization easy and consistent.

5.2 Results

We have evaluated the performance of the parallel refinement in smoke simulations shown in Fig. 12. The left image is start, and the right image is five frames later. Note that in the right image, smoke has risen slightly. We measure the computation time for these five frames. We used a machine with two Intel Xeon X5650 2.67GHz processors (12 cores). As shown in Table 1, we obtain a speedup factor of 9.2, which is similar to the speedup factors obtained for embarrassingly parallel semi-Lagrangian advection (CIR) steps. The simulation started from 42,771,176 cells. During the five simulation steps, 7,176,624 cells are created and 4,402,472 cells are deleted. Since the algorithm is independent of the number of threads, the number of cells created and deleted do not depend on the number of threads.

6 Applications

6.1 Segmentation using level sets

Since their introduction [22,23,28], level sets have been used in a large number of applications, including image segmen-

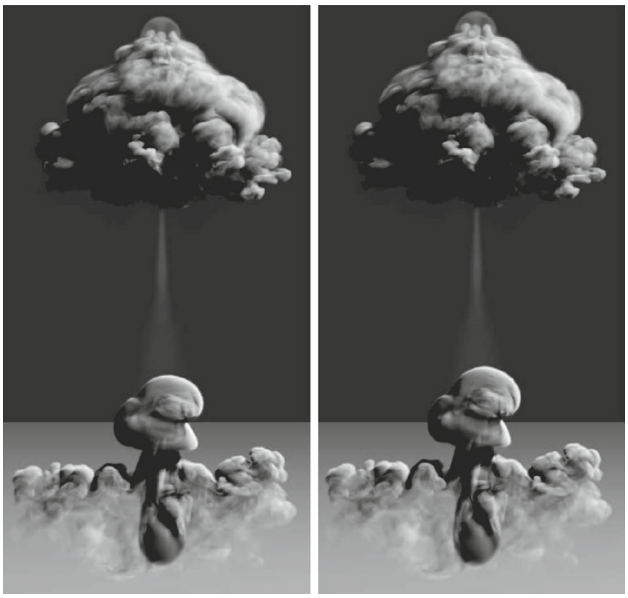


Fig. 12 The first and the last (five frames later) smoke simulation frames used for the parallel refinement benchmark. The grid resolution is equivalent to $1{,}024^3$ with 42.77 million cells. During the simulation, 7.2 million cells are created and 4.4 million cell are deleted. Tree connectivity and smoke field require 5.2 and 167 MB, respectively, in disk without compression

Table 1 Computation times and speedup factors for parallel adjustment of an octree grid during five smoke simulation frames (Fig. 12) on a 12 core machine

Threads	Octree adjustment		Velocity advection		Smoke Advection	
	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
1	29.04	1.00	47.60	1.00	41.83	1.00
2	14.57	1.99	24.05	1.98	21.07	1.98
3	10.62	2.74	16.53	2.88	14.34	2.92
4	8.18	3.55	12.84	3.71	11.14	3.75
5	6.62	4.39	10.78	4.42	9.19	4.55
6	5.74	5.06	9.20	5.17	7.83	5.35
7	4.94	5.87	8.05	5.91	6.80	6.16
8	4.43	6.55	7.23	6.58	6.10	6.85
9	3.95	7.36	6.62	7.19	5.56	7.52
10	3.62	8.01	6.11	7.79	5.08	8.23
11	3.39	8.57	5.60	8.50	4.64	9.02
12	3.14	9.24	5.32	8.94	4.39	9.52

The time for multi-pass adjustment is smaller than a simple scalar field (smoke) CIR advection

tation, which is performed either using edge-based level set methods [5,17], or region-based level set methods [7]. While edge-based methods are used to segment a feature that has a well-defined boundary, region-based methods can be used when a feature in an image has statistics that are different from the background. Examples of such statistics are intensity [7], texture [4], or motion [9]. Region-based level set methods can also be used to segment vector-valued images [6], or to segment an image into more than two regions using multi-phase level sets [4,32]. In particular, region-based level set segmentations are suitable to adaptive tree grids since a tree node can contain statistics inside the cell. Therefore, we choose to work with region-based level set segmentation methods.

Each cell-centered sample can represent the image statistics of all image pixels in the cell, which can be stored in the image pyramid. This allows us to perform segmentation in various resolutions, adaptively adjusted to the size of the region and curvatures. The grid is refined using level set and curvature as:

$$d_{\text{desired}} = d_{\max} - \left(\frac{\phi}{\sigma d_{\max}} + \frac{C_{\max}}{\pi/2} \tan^{-1} \left(\tan \left(\frac{1}{C_{\max}} \right) \frac{\kappa_C}{\kappa} \right) \right), \tag{8}$$

where $C_{\max}, d_{\max}, \sigma$, and κ_C are user defined parameters. $C_{\max}$ is the maximum coarseness at interface. We choose $C_{\max} = 3, 4$, or 5. $d_{\max}$ is the maximum tree depth, σ is the thickness of interface within which the tree is refined and outside which the tree always tries to be coarsened as long as the constraint (1) is not violated. We choose σ to be a few maximum resolution grid cells. Finally, κ_C is a parameter we choose to be $0.5 d_{\text{cell}}$, where d_{cell} is the width of the cell. $\kappa = \nabla \cdot \frac{\nabla \phi}{||\nabla \phi||}$ is the curvature computed at cell centers. In this way, at locations with high curvature, κ_C / κ becomes small. When $\phi \to 0$ (near interface) and $\kappa \to \infty$ (high curvature), the tree is refined to higher resolution, i.e., $d_{\text{desired}} \to d_{\max}$.

Figure 13 shows the segmentation result on a 2D image. As shown in Figs. 5 and 13, the level set interfaces remained smooth during the image segmentation processes, without developing any artifacts.

Figure 14 shows the segmentation result for a volumetric cell image taken from the CCDB site (http://ccdb.ucsd.edu). Since the cell shape has high curvature regions, cells are refined to high depths. As a result, a naive integration with maximum tree depth of 9 took a very long time (30 min). By segmenting in a low resolution first, and then moving to higher resolutions, the segmentation time is reduced to approximately 40 sec on an Intel Q6600 CPU. Further speedups may be achieved when combined with narrow band methods [1].

Note that the number of segmentation time steps at each resolution is manually chosen. A future direction would be to evaluate the convergence properties at interface locations, and then, for the region that achieved convergence, to stop the computation, and for the region that has not yet converged, to automatically unlock higher resolutions with smaller time steps.

To visualize the 3D level set interface, we used the dual-grid marching cube idea [27]. Owing to the limited number of

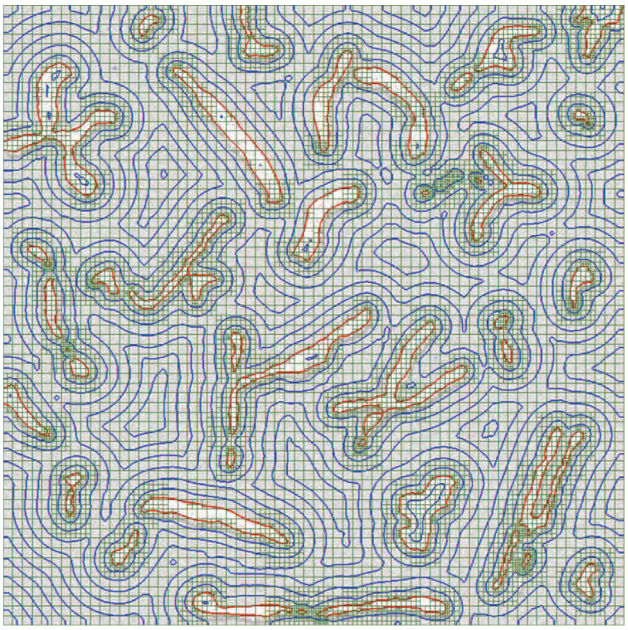


Fig. 13 Segmentation of a 512×512 image with grid resolutions high only at high curvature interfaces

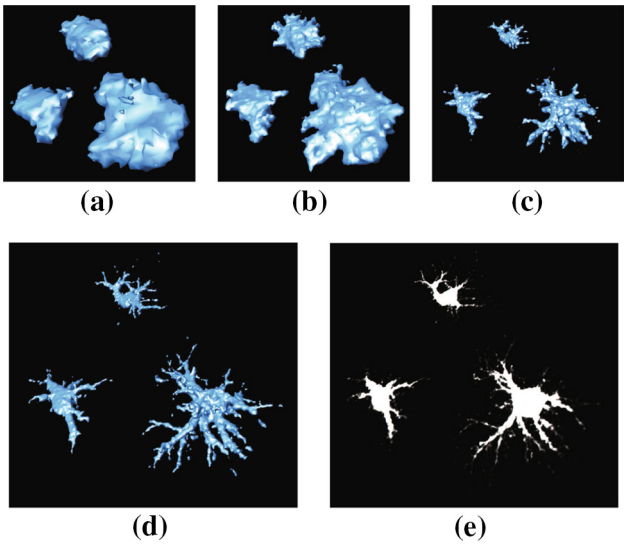


Fig. 14 Segmentation of a cell image data of size 512 × 512 × 64 took about 30 min by naively solving the PDE. The segmentation time can be reduced to 40 s (Intel Q6600 CPU) by first segmenting in a low resolution and then moving on to higher resolutions. Images **a**–**d** are segmentation results in 64 × 64 × 8, 128 × 128 × 16, 256 × 256 × 32, and 512 × 512 × 64 resolutions, respectively. Volume data shown in **e** are courtesy of CCDB (http://ccdb.ucsd.edu)

stencils, it is straightforward to generate the dual-grid cubes, which may have some edges or faces collapsed to a point.

6.2 Smoke simulation

Simulation of smoke is ample in modern films and is a problem studied broadly in the graphics community, e.g., [11,29].

In this paper, we perform the smoke simulation by solving the Navier–Stokes and smoke equations using the operator splitting and pressure projection [29]

$$\dot{\mathbf{u}} = -\mathbf{u} \cdot \nabla \mathbf{u} + \mu \nabla^2 \mathbf{u} + \frac{1}{\rho} \nabla P + \mathbf{f}, \tag{9}$$

$$\dot{s} = -\mathbf{u} \cdot \nabla s + \mu_s \nabla^2 s, \tag{10}$$

on the adaptive grid we have developed in this paper. In the previous equations, $\mathbf{u}$ is velocity, P is pressure, ρ is fluid density, μ, μ_s are the viscosities and smoke diffusion coefficients, $\mathbf{f}$ is a force term that includes gravity and artificial smoke buoyancy force, and s is the smoke density. We use the sample perturbation approach [16] to construct pressure and diffusion matrices. The grid is refined using smoke density as follows:

$$d_{\text{desired}} = C_s \, d_{\max} \, |\nabla s| \; + D_s \; \frac{s(1-s)}{1+100s^2}. \tag{11}$$

Here, we chose $C_s = 0.1$. The second term is introduced simply to tune the adjustment behavior in low density regions where $|\nabla s|$ is too small. We have used $D_s = 44.25$. We rendered the smoke with a ray tracer that samples the octree grid using the proposed interpolation. The uncompressed tree consumes 5 to 500 MB in disk depending on the number of cells. The smoke field samples consume 4 bytes per cell, and the tree connectivity consumes one bit per cell that represents whether the cell is a leaf or not. If a uniform grid is used, 8GB will be needed for smoke samples.

The smoke simulation and ray tracing results are shown in Fig. 15. The simulation time step is 1/30. The simulation for 20 s (600 frames) took 8 h to finish.

7 Discussion and conclusions

The new 2-to-1 balancing constraint would create smoother variation in node depth and as a result, produces slightly smoother solution fields. This, however, would not make a significant difference compared to the original 2-to-1 balancing scheme in the aspect of visual quality. The biggest advantage of the new constraint is simply that the tree stencil is limited to a small number of cases. This is suitable for variety of applications using multi-grid or pyramid approaches since we can explore directly interpolating tree center sampled data. To our knowledge, no prior art achieved this. Only two related works [18,30] are to tetrahedralize entire domain, and approximation (not interpolating) using corner locations.

We developed a new tree balancing scheme that results in only a few stencils. For each stencil, we pre-compute the interpolation weights and the required procedures, which allows us to interpolate data sampled at cell centers. We believe that other operations such as tetrahedralization of the domain can be precomputed or simplified. In addition,

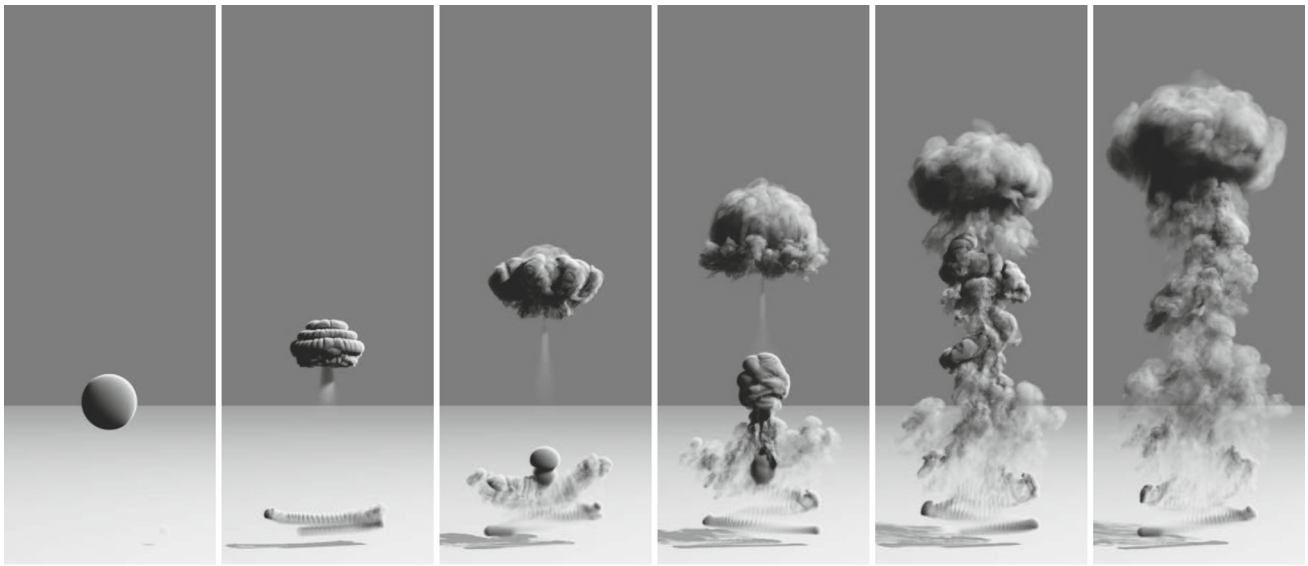


Fig. 15 Smoke simulation and ray tracing using an octree grid: 16 × 32 × 16 uniform array of roots, each of which is refined up to 6 levels. This grid is equivalent to 1,024 × 2,096 × 1,024 resolution

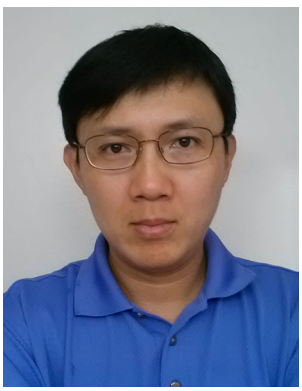
we developed new algorithms that adjust trees in parallel, and showed that the proposed balancing constraints are satisfied by the parallel adjustment. Finally, we applied the tree balancing and parallel adjustment algorithms to smoke simulation and image segmentation problems, and showed that the trees work well for such applications, and scale well with the number of processors.

Acknowledgments The second author was supported by the US National Science Foundation, award CMMI-0856565. The third author was supported by National Research Foundation of Korea (NRF) (Grant NRF-2011-0023134).

References

1. Adalsteinsson, D., Sethian, J.: Fast level set method for propagating interfaces. J. Comput. Phys. **118**, 269–277 (1995)
2. Bai, Y., Han, X., Prince, J.L.: Octree grid topology preserving geometric deformable model for three-dimensional medical image segmentation. In: Information Processing in Medical Imaging (IPMI 2007), pp. 20:556–68 (2007)
3. Benson, D., Davis, J.: Octree textures. ACM Transactions on Graphics. In: Proc. of SIGGRAPH, 21, pp. 785–790 (2002)
4. Brox, T., Weickert, J.: Level set segmentation with multiple regions. IEEE Trans. Image Process. **15**(10), 3213–3218 (2006)
5. Caselles, V., Kimmel, R., Sapiro, G.: Geodesic active contours. In: International Conference of Computer Vision (ICCV), pp. 694–699 (1995)
6. Chan, T.F., Sandberg, B.Y., Vese, L.A.: Active contours without edges for vector-valued images. J. Vis. Commun. Image Represent. **11**(2), 130–141 (2000)
7. Chan, T.F., Vese, L.A.: Active contours without edges. IEEE Trans. Image Process. **10**(2), 266–277 (1999)
8. Chen, H., Min, C.H., Gibou, F.: A supra-convergent finite difference scheme for the poisson and heat equations on irregular domains and non-graded adaptive cartesian grids. J. Sci. Comput. **31**, 19–60 (2007)
9. Cremers, D.: A variational framework for image segmentation combining motion estimation and shape regularization. In: IEEE International Conference on Computer Vision and Pattern Recognition (CVPR), pp. 53–58 (2003)
10. DeBry, D., Gibbs, J., Petty, D.D., Robins, N.: Painting and rendering textures on unparameterized models. ACM Transactions on Graphics. In: Proc. of SIGGRAPH, 21, pp. 763–768 (2002)
11. Foster, N., Fedkiw, R.: Practical animation of liquids. In: ACM SIGGRAPH, pp. 15–22 (2001)
12. Frisken, S.F., Perry, R.N., Rockwood, A.P., Jones, T.R.: Adaptively sampled distance fields: a general representation of shape for computer graphics. In: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '00, pp. 249–254. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000). doi:10.1145/344779.344899
13. Gibou, F., Min, C.H., Ceniceros, H.: Finite difference schemes for incompressible flows on non-graded adaptive cartesian grids. Fluid Dyn. Mater. Process. **154**, 199–208 (2007)
14. Ju, T., Losasso, F., Schaefer, S., Warren, J.: Dual contouring of hermite data. ACM Transactions on Graphics. In: Proc. of SIGGRAPH, 21(3), pp. 339–346 (2002)
15. Kim, B., Tsiotras, P.: Image segmentation on cell-center sampled quadtree and octree grids. In: Proceedings of SPIE Electronic Imaging / Wavelet Applications in Industrial Processing VI, pp. 265–278 (2009)
16. Losasso, F., Gibou, F., Fedkiw, R.: Simulating water and smoke with an octree data structure. In: ACM SIGGRAPH, pp. 457–462 (2004)
17. Malladi, R., Sethian, J.A., Vemuri, B.C.: Evolutionary fronts for topology-independent shape modeling and recovery. In: Proceedings of the third European conference on Computer vision, pp. 1–13 (1994)
18. Milne, B.: Adaptive Level Set Methods Interfaces. PhD thesis, Dept. of Mathematics, University of California, Berkeley, CA (1995)
19. Min, C.H., Gibou, F.: A second order accurate projection method for the incompressible navier-stokes equations on fully adaptive grids. J. Comput. Phys. **219**, 912–929 (2006)

20. Min, C.H., Gibou, F.: A second order accurate level set method on non-graded adaptive grids. J. Comput. Phys. **225**, 300–321 (2007)
21. Min, C.H., Gibou, F., Ceniceros, H.: A supra-convergent finite difference scheme for the variable coefficient poisson equation on fully adaptive grids. J. Comput. Phys. **202**, 577–601 (2006)
22. Osher, S., Sethian, J.A.: Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations. J. Comput. Phys. **79**, 12–49 (1988)
23. Osher, S.J., Fedkiw, R.P.: Level Set Methods and Dynamic Implicit Surfaces. Springer, Berlin. ISBN 0-387-95482-1 (2002)
24. Parashar, M., Browne, J.C.: Distributed dynamic data-structures for parallel adaptive mesh-refinement. In: Proceedings of the international conference for high performance computing (1995)
25. Plewa, T., Linde, T., Weirs (Editors), V.G.: Adaptive Mesh Refinement - Theory and Applications. In: Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, Sept. 3–5, 2003. Lecture Notes in Computational Science and Engineering, Vol. 41. Springer, Berlin (2003)
26. Sagan, H.: Space-Filling Curves. Springer, Berlin (1994)
27. Schaefer, S., Warren, J.: Dual marching cubes: primal contouring of dual grids. In: Proceedings of Pacific Graphics, pp. 70–76 (2004)
28. Sethian, J.A.: Level Set Methods and Fast Marching Methods. Cambridge University Press, Cambridge. ISBN 0-521-64557-3 (1999)
29. Stam, J.: Stable fluids. In: ACM SIGGRAPH, pp. 121–128 (1999)
30. Strain, J.: Fast tree-based redistancing for level set computations. J. Comput. Phys. **152**(2), 648–666 (1999)
31. Tu, T., O'hallaron, D.R.: Balanced refinement of massive linear octrees. Tech. Rep. CMU-CS-04-129, Carnegie Mellon School of Computer Science, Pennsylvania (2004)
32. Vese, L.A., Chan, T.F.: A multiphase level set framework for image segmentation using the mumford and shah model. IEEE Trans. Image Process. **50**(3), 271–293 (2002)
33. Westermann, R., Kobbelt, L., Ertl, T.: Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces. Vis. Comput. **15**(2), 100–111 (1999)
34. Wilhelms, J., Gelder, A.V.: Octrees for faster isosurface generation. ACM Trans. Graph. **11**(3), 201–227 (1991)
35. Yerry, M.A., Shephard, M.S.: Automatic three-dimensional mesh generation by the modified-octree technique. Int. J. Numer. Methods Eng. **20**(11), 1965–1990 (1984)
36. Zhou, K., Gong, M., Huang, X., Guo, B.: Data-parallel octrees for surface reconstruction. IEEE Trans. Vis. Comput. Graph. **177**(5), 681–699 (2011)



Byungmoon Kim completed Ph.D. in computer science at the Georgia Institute of Technology, where he researched on high-order advection methods (BFECC and MacCormack) for fluid simulations, bubbles in foam simulations using the volume control method, a new mesh filter transfer function and its coefficient computation (filter design). While studying in computer science, Byungmoon Kim also worked on shadow volume, mesh editing, video effects, haptics, and continuous collision detection, and received MS in computer science. Byungmoon Kim also completed MS in Mathematics and MS in Aerospace Engineering, both at Georgia Tech. Byungmoon Kim is now a researcher at Imagination Lab in Adobe Research, where he is working on computer graphics and simulation problems.

Panagiotis Tsiotras received his Ph.D. degree in Aeronautics and Astronautics from Purdue University in 1993. He also holds degrees in Mechanical Engineering and Mathematics. Currently, he is College of Engineering Dean's Professor in the School of Aerospace Engineering and a member of the Institute of Robotics and Intelligent Machines at the Georgia Institute of Technology. His research interests are in theoretical optimal and nonlinear control and vehicle autonomy, with applications to aerospace and mechanical systems. He is a recipient of the NSF CAREER Award. He is a fellow of AIAA and a senior member of IEEE.

Jeong-Mo Hong is an assistant professor at Computer Science and Engineering Department. He received the B.S. degree in 2000 and the M.S. degree in 2002 in mechanical engineering from the Korea Advanced Institute of Science and Technology (KAIST), South Korea. After obtaining the Ph.D. degree in computer science from the Korea University in 2005, he developed several simulation techniques for visual effects as a research fellow at the Computer Science Department of Stanford University.

Oh-young Song is an associate professor in the Department of Digital Contents at Sejong University, Korea. He has been working at Sejong University since 2006. His primary research interests are physics-based animation and character animation. He received the B.S., M.S., and Ph.D. degrees in the School of Electrical Engineering and Computer Science from Seoul National University, Korea, in 1998, 2000, and 2004, respectively. He was a postdoctoral fellow in the School of Electrical Engineering and Computer Science at Seoul National University, Korea from 2004 to 2006.