A Stack-Free Parallel h-Adaptation Algorithm for Dynamically Balanced Trees on GPUs

LIXIN REN, Institute of Software, Chinese Academy of Sciences, China
XIAOWEI HE*, Institute of Software, Chinese Academy of Sciences, China
SHUSEN LIU, Institute of Software, Chinese Academy of Sciences, China
YUZHONG GUO, Institute of Software, Chinese Academy of Sciences, China
ENHUA WU, Key Laboratory of System Software (Chinese Academy of Sciences) and SKLCS, Institute of Software, Chinese Academy of Sciences, China and FST, University of Macau, China

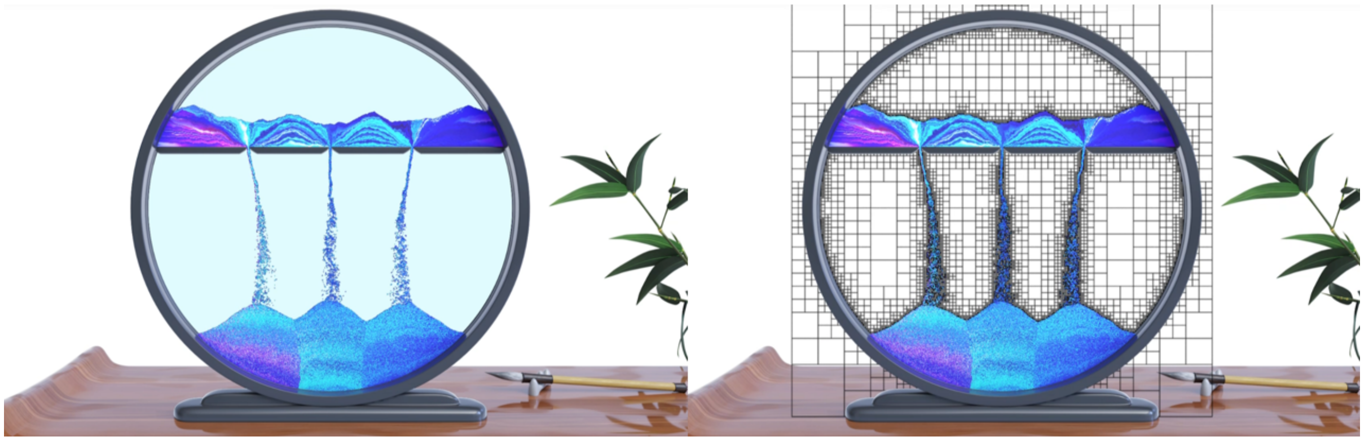


Fig. 1. Left: A three-hole hourglass simulation coupling two different fluids, where water is represented by the octree and sand is represented by particles. Right: The 2-balanced octree utilized in the simulation has a maximum level of $l_{max} = 9$ and $l_{num} = 9$ and consists of 150,000 leaf nodes. The average computational cost for constructing the 2-balanced octree is around 0.77ms.

Prior research has demonstrated the efficacy of balanced trees as spatially adaptive grids for large-scale simulations. However, state-of-the-art methods for balanced tree construction are restricted by the iterative nature of the ripple effect, thus failing to fully leverage the massive parallelism offered by modern GPU architectures. We propose to reframe the construction of balanced trees as a process to merge N-balanced Minimum Spanning Trees (N-balanced MSTs) generated from a collection of seed points. To ensure optimal performance, we propose a stack-free parallel strategy for constructing all internal nodes of a specified N-balanced MST. This approach leverages two 32-bit integer registers as buffers rather than relying on an integer array as a stack during construction, which helps maintain balanced workloads across different GPU threads. We then propose a dynamic update algorithm utilizing refinement counters for all internal nodes to enable parallel insertion and deletion operations of N-balanced MSTs. This design achieves significant efficiency improvements compared to full reconstruction from scratch, thereby facilitating fluid simulations in handling dynamic moving boundaries. Our approach is fully compatible with GPU implementation and demonstrates up to an order-of-magnitude speedup compared to the state-of-the-art method [Wang et al. 2024]. The source code for the paper is publicly available at https://github.com/peridyno/peridyno.

*Corresponding authors

SKLCS: State Key Laboratory of Computer Science.
FST: Faculty of Science and Technology.
Authors' addresses: Lixin Ren, Institute of Software, Chinese Academy of Sciences, Beijing, 100190, China, renlixin@iscas.ac.cn; Xiaowei He, Institute of Software, Chinese Academy of Sciences, Beijing, China, xiaowei@iscas.ac.cn; Shusen Liu, Institute of Software, Chinese Academy of Sciences, Beijing, China, liushusen@iscas.ac.cn; Yuzhong Guo, Institute of Software, Chinese Academy of Sciences, Beijing, China, guoyuzhong@iscas.ac.cn; Enhua Wu, Key Laboratory of System Software (Chinese Academy of Sciences) and SKLCS, Institute of Software, Chinese Academy of Sciences, Beijing, China and FST, University of Macau, Macao SAR, China, ehwu@umac.mo.

0730-0301/2025/12-ART180
https://doi.org/10.1145/3763349

CCS Concepts: • **Computing methodologies → Physical simulation**.

Additional Key Words and Phrases: Balanced trees, dynamic update, GPU, fluid simulation

ACM Reference Format:
Lixin Ren, Xiaowei He, Shusen Liu, Yuzhong Guo, and Enhua Wu. 2025. A Stack-Free Parallel h-Adaptation Algorithm for Dynamically Balanced Trees on GPUs. *ACM Trans. Graph.* 44, 6, Article 180 (December 2025), 15 pages. https://doi.org/10.1145/3763349

1 INTRODUCTION

Balanced trees, in which adjacent nodes differ by no more than one level in resolution, are commonly used in fluid simulation [Aanjaneya et al. 2017; Ando and Batty 2020; Goldade et al. 2019]. As shown in Figure 2, balanced trees offer adaptability and facilitate

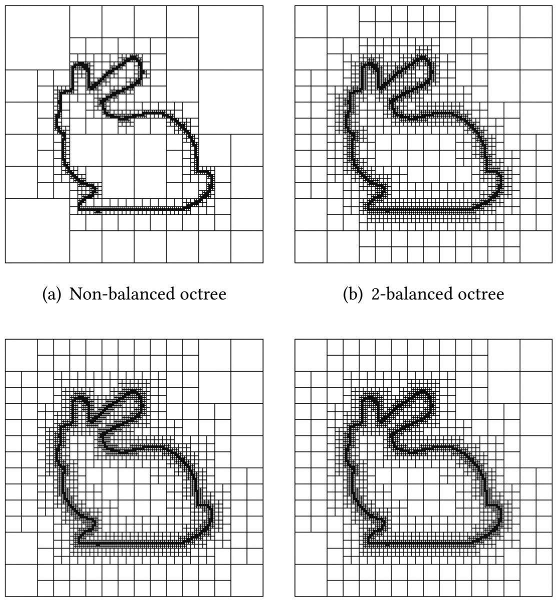


Fig. 2. Cross-sectional views of the octree representation of the Stanford Bunny model under different balance constraints.

a smooth increase in resolution, providing significant advantages in solving partial differential equations. The resolution restriction among cells in a balanced tree is commonly referred to as the 2:1 balance constraint. Constructing a balanced tree typically involves two stages: the first stage consists of creating a tree without constraint, while the second stage focuses on refining the 2:1 balance [Sundar et al. 2008; Wang et al. 2024].

For constructing non-balanced trees, the common practice is to use either a bottom-up or top-down approach. In recent years, several studies [Ajmera et al. 2008; Lauterbach et al. 2009; Schwarz and Seidel 2010; Zhou et al. 2010] have explored the efficient construction of GPU-based octrees and quadtrees. However, constructing balanced trees presents a significant challenge for parallel implementation due to the balance constraint. This challenge arises from the well-known *ripple effect*: the refinement of a leaf node—i.e., subdividing it into eight child nodes (four in 2D)—can trigger the refinement of other nodes, including those that are not immediate neighbors. The ripple effect also creates a bottleneck for fully GPU-based balanced tree construction. Wang et al. [2024] recently proposed a GPU-based algorithm for constructing balanced octrees. Their algorithm uses a bottom-up approach, enforcing balance refinement level by level, from the finest to the coarsest resolution. However, the iterative nature remains, and the overall performance is significantly impacted by the repeated processes of sorting and removing duplicate nodes.

In this work, we reframe the procedure of constructing balanced trees. Unlike the canonical method which balances a tree starting from the finest resolution and repeat the same process for coarser resolutions one by one [Isaac et al. 2012; Wang et al. 2024], we articulate to construct the balanced trees in two stages: In the initial stage, we build the coarsest balanced tree (which is later referred to as the N-balanced Minimum Spanning Tree or N-balanced MST) for each seed point in parallel. Subsequently, in the second stage, we propose to merge those trees into a single one. Nevertheless, in practical implementation, the two stages can be executed concurrently with high efficiency due to several significant findings and technical innovations. By identifying several crucial properties of the N-balanced MST for a leaf node containing seed points, we propose a stack-free internal node traversal strategy that utilizes only two 32-bit integer registers rather than using an integer array as a stack to balance the workload across different GPU threads. Then, we propose a dynamic update algorithm based on refinement counters to support the insertion and deletion of MSTs. Our method ensures high performance in constructing various balanced trees based on input seed points, which are generated from input primitives—such as surface meshes, point clouds, or analytical representations—according to the specified sampling requirements.

To summarize, our primary contributions are:

- A new framework to construct balanced trees satisfying the 2:1 balance constraint, including N-balanced octrees, quadtrees or binary trees;
- A stack-free parallel algorithm for constructing N-balanced MSTs relies solely on two 32-bit integer registers, eliminating the need for a stack during GPU implementation;
- A dynamic update algorithm utilizing refinement counters for all internal nodes, enabling both efficient parallel insertion and deletion operations.

2 RELATED WORK

There has been considerable parallel research on constructing spatial hierarchical data structures on GPUs, such as kd-trees [Zhou et al. 2008], bounding volume hierarchies (BVHs) [Lauterbach et al. 2009] and octrees [Karras 2012]. The common construction strategies employed in these works include top-down [Zhou et al. 2008] and bottom-up [Zhou et al. 2010] approaches. Lauterbach et al. [2009] proposed to achieve a certain degree of parallelism by constructing the octree in a top-down layered manner. Zhou et al. [2008] proposed to construct the kd-tree using a top-down recursive splitting approach. Lauterbach et al. [2009] presented a method that constructs BVHs in a top-down recursive manner based on the sorted space-filling Morton curve. Ajmera et al. [2008] proposed the idea that the octree can be viewed as multiple space-filling curves at different resolutions, and thus the octree can be constructed in a bottom-up manner based on the sorted space-filling curve (SFC), but did not provide a specific implementation. Zhou et al. [2010] provided the specific implementation process of constructing the octree in a bottom-up layered manner, and used the constructed octree for surface reconstruction. Schwarz and Seidel [2010] and Liu and Kim [2013] adopted a similar method to Zhou et al. [2010] to construct the octree, but without the need to build additional complex data structures for the octree nodes, and further applied the octree structure to solid voxelization and signed distance fields

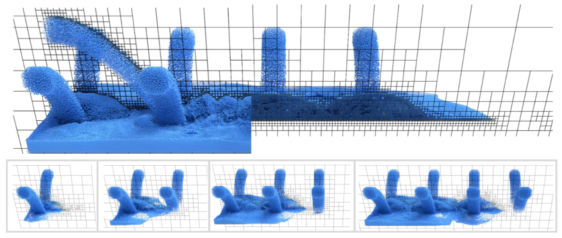
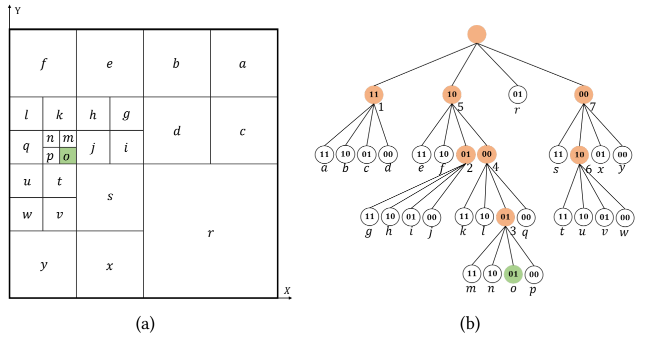


Fig. 3. Two-dimensional cross-sectional representations of octrees in the water pouring scenario.



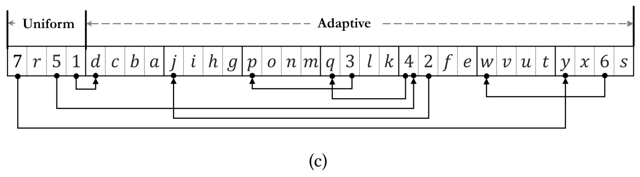


Fig. 4. Example of a *1*-balanced Minimum Spanning Tree $\mathcal{T}_1(o)$ for a quadrant o in the quadtree. (a) A representation of the corresponding spatial grid of $\mathcal{T}_1(o)$; (b) A representation of $\mathcal{T}_1(o)$ in its postorder traversal; (c) A representation of $\mathcal{T}_1(o)$ in GPU memory.

computation. Karras [2012] proposed a parallel algorithm for constructing binary radix trees, and used it as a basis to construct BVHs, kd-trees, and octrees. Morrical and Edwards [2017] constructed quadtrees based on the method in Karras [2012], and further adjusted the quadtrees to separate, or resolve, collections of closely spaced objects. Chen et al. [2022] constructed the BVH based on the method in Karras [2012], and then traversed the BVH in a top-down manner to build the octree. The above works involved the parallel construction of octrees, but did not address the implementation of balance constraint.

In the field of computational physics, there has been some work exploring parallel construction methods for balanced octrees, with a focus on distributed memory machines [Burstedde et al. 2011; Isaac et al. 2012; Sundar et al. 2008]. Sundar et al. [2008] proposed several important ideas, such as the linear octree, where leaf octants are stored in a contiguous array. This work suggested a bottom-up approach to octree construction, which enables uniform load distribution across processors. It also introduced a two-stage balancing scheme, where local balancing is first performed on each processor, followed by balancing across processor boundaries to avoid iterative communication. Burstedde et al. [2011] developed a fast weighted partitioning scheme based on space-filling curves with appropriate subdivisions, in order to achieve better load balancing across processors. The research from Isaac et al. [2012] proposed a method to reduce communication overhead by determining whether any two given octants maintain a consistent distance or size relationship. More recently, Wang et al. [2024] presented a GPU-parallel implementation of the Sundar et al. [2008] approach, incorporating several optimizations leveraging GPU hardware, such as the use of shared memory.

In the fields of computer graphics as well as computational fluid dynamics, there are many works that utilize graded octrees, also referred to as balanced octrees. Popinet [2003] introduced the definition of graded octree and the constraints it should satisfy, and initially applied it to the solution of incompressible fluids. Ferstl et al. [2014] proposed a hexahedral finite element solver based on graded octree. Aanjaneya et al. [2017] proposed the use of power diagrams on graded octree to handle resolution transitions reaching the free surface. Goldade et al. [2019] presented a graded octree-based adaptive variational finite-difference framework that significantly accelerates the solution of the free surface viscosity equations. Ando and Batty [2020] proposed a graded octree-based fluid solver with adaptive capability for liquid surfaces. The aforementioned works focus on graded octree-inspired fluid simulation, without addressing the efficiency issues related to graded octree construction. Kim et al. [2015] proposed a constraint that is slightly stronger than the 2:1 balance constraint and developed a parallel tree adjustment algorithm suitable for multi-core CPUs. This algorithm was subsequently applied to volumetric painting [Kim et al. 2018]. There are also a series of works [Batty 2017; Guittet et al. 2015; Losasso et al. 2006, 2004; Sousa et al. 2019] that strive to overcome the numerical and computational difficulties associated with non-graded octree data structures, which are not the main focus of this paper. However, our method may also provide benefits to those works, as our approach can be easily extended to the efficient parallel construction of non-graded octrees.

3 OVERVIEW

A tree of octants or quadrants is considered complete if every internal node has precisely eight children in the case of an octree, or four in the case of a quadtree. However, when used in fluid simulation, the tree needs to be 2:1 balanced, meaning that the size difference between any adjacent nodes cannot exceed 2:1. Given a set of scattered points, the top-down approach starts from the root node and gradually subdivides the nodes until the finest level is reached. During this procedure, the 2:1 constraint can be easily fulfilled by checking the size difference between adjacent nodes that share either an edge or a face [Li et al. 2016]. However, the top-down approach does not fit well into the GPU implementation due to issues with data locality and dynamic memory overhead. In contrast, the bottom-up approach starts with the finest leaf nodes containing the given points and simultaneously constructs additional nodes from the bottom level upwards. Enforcing the 2:1 constraint during this procedure is challenging, as prior to the completion of the tree

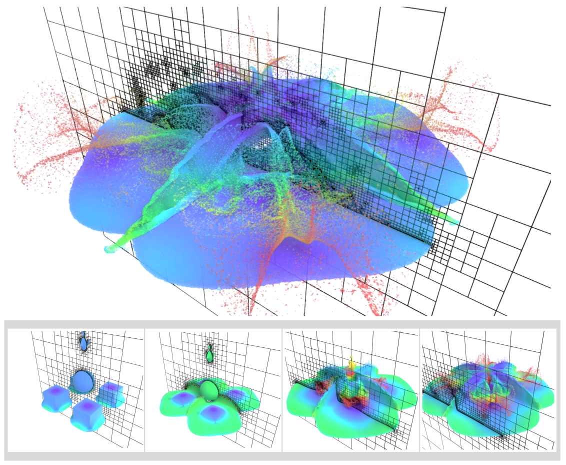


Fig. 5. Two-dimensional cross-sectional representations of octrees in the dambreak scenario.

construction, internal nodes may lack a full set of eight children (four in the case of a quadtree), thereby leading to the presence of holes. The primary objective of this study is to create a GPU-friendly bottom-up algorithm for constructing complete balanced trees from a specified collection of seed points.

3.1 Motivation

Before proceeding, let us first clarify the definition of the balance constraint for different trees. Following the classification in [Sundar et al. 2008], we refer to octrees that are balanced across faces as ***2-balanced***, octrees that are balanced across both edges and faces as ***1-balanced***, and octrees that are balanced across corners, edges, and faces as ***0-balanced***. In the context of quadtrees, a ***1-balanced*** quadtree refers to one that is balanced across its edges, while a ***0-balanced*** quadtree is balanced across both corners and edges. A ***0-balanced*** binary tree refers to one that is balanced across its corners.

Table 1. Notations.

Notation	**Description**
$\mathcal{P}(n)$	Parent of node n.
$\mathcal{C}(n)$	Children of node n.
$\mathcal{A}(n)$	Ancestors of node n.
$\mathcal{D}(n)$	Descendants of node n.
$\mathcal{I}(n)$	Insulation layer of node n.
$\mathcal{S}_{dir}(n)$	Searching directions of node n.
$\mathcal{T}_N(n)$	N-balanced Minimum Spanning Tree of node n.
$\mathcal{N}_o(n,l)$	Off branch neighbors of node n at level l.
$\mathcal{L}(n,l)$	Locational code of node n at level l.
$\mathcal{M}(n)$	Morton code of node n.
$\mathcal{B}$	32-bit integer register used as a bitmask.
$h(k)$	Hash function.
$l(n)$	Level of node n.
$num(\cdot)$	The number of elements.

Without loss of generality, let us take the *1-balanced* quadtree in 2D as an example. As a starting point, Figure 4 demonstrates a special case where only one quadrant is considered to construct the *1-balanced* quadtree. The tree is structured with the minimum number of quadrants necessary to maintain the 2:1 balance condition. Specifically, if any four neighboring leaf quadrants are replaced by a coarser parent quadrant, this results in a violation of the 2:1 balance condition. We denote $\mathcal{T}_N(n)$ as the ***N*-balanced Minimum Spanning Tree** (N-balanced MST) for a specified leaf node n, where $\mathcal{T}_N(n)$ represents the coarsest complete tree that adheres to the N-balanced condition. With above defined $\mathcal{T}_N(n)$, the union of any two N-balanced MSTs also adheres to the balance condition. Applying this principle, our method for constructing the N-balanced tree from a collection of seed points can be conceptualized in two stages. During the initial stage, we employ a stack-free parallel h-adaptation algorithm to build the N-balanced MST for each node at the finest level that contains at least one seed point. Subsequently, in the second stage, a hash table is utilized to collect all internal nodes of all $\mathcal{T}_N(n)$, followed by the construction of complete balanced tree and parent-child relationships based on a sorted array of internal nodes in descending order. At first glance, the concept of merging all $\mathcal{T}_N(n)$ into a single tree may appear to pose performance challenges, particularly due to potential conflicts arising from duplicate nodes. Nevertheless, this issue is effectively mitigated by our stack-free parallel h-adaptation algorithm, which implements node generation on-the-fly (without the necessity to store $\mathcal{T}_N(n)$ during construction) while employing a hash table to minimize conflicts.

To facilitate the following discussion, Table 1 summarizes the notations used in the subsequent context. Section 4.1 initially presents the construction of the N-balanced MST for a given leaf node at the finest level in a parallel manner. Section 4.2 details the construction of a complete balanced tree from distinct internal nodes collected via a hash table. Building on this, Section 5 introduces the refinement counters-based dynamic update algorithm to support high-frequency updates in fluid simulations.

4 STACK-FREE PARALLEL *h*-ADAPTATION

There are two common methods to represent a tree. The pointer-based representation breaks down the domain into a collection of unsorted nodes and uses pointers to establish parent-child relationships. In contrast, the linear representation assigns a unique key to each node and represents the tree as a collection of sorted leaf nodes. In our implementation, we propose using a combined representation, as demonstrated in Figure 4(c). It consists of two parts: the first part stores a collection of sorted nodes representing the uniform grid at the topmost level l_{top}, while the second part stores their children in descending order according to their locational keys, representing the adaptive grid below the topmost level. Our data structure implements h-adaptation, where the grid size (h) is dynamically adjusted in response to spatial variations [Koschier et al. 2016]. The data structure that stores the tree has the following properties: (1) All children of the same parent are stored consecutively, therefore only the index of the first child needs to be stored in the parent, which simplifies tree traversal and node access; (2) The tree

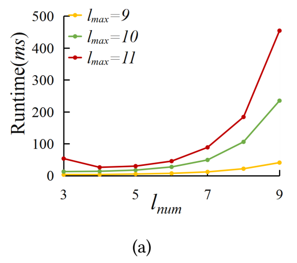
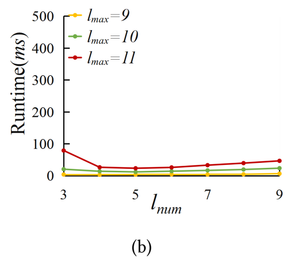



Fig. 6. A comparison of the computational cost of a 2-balanced octree representation for the armadillo model. Note as the level number l_{num} increase, (a) the performance of the stack-based approach rapidly deteriorates due to its frequent read and write operations on the stack; (b) In contrast, the performance of our method, which utilizes two 32-bit integer registers as buffers, remains relatively constant across different level numbers.

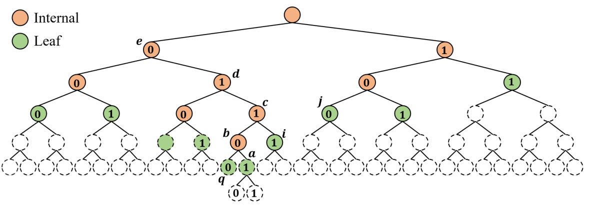


Fig. 7. A *0*-balanced MST of the node *a* used to demonstrate the ripple effect. Note if the leaf node *a* is to be refined, it triggers the simultaneous splitting of leaf nodes *i* and *j* to maintain the 2:1 balance condition.

maintains good data locality for nodes that are spatially close to each other.

To construct an adaptive tree fulfilling the above requirements, we first demonstrate how to calculate the locational code by using Morton encoding [Sundar et al. 2008; Wang et al. 2024] together with the level information. Assuming the maximum level of the tree is l_{max}, the locational code for a node with integer coordinates (x, y, z) at level l is defined as follows:

$$\mathcal{L} = \underbrace{\cdots [z]_i [y]_i [x]_i \cdots}_{dim*l} \quad \underbrace{0}_{dim*(l_{max}-l)} \quad \underbrace{[l]}_{l_0}, \tag{1}$$

where $[\cdot]$ represents the binary representation, and the subscript i indicates the i-th bit of the binary representation which is in the range of $i \in [1, l]$. The first $dim*l$ bits represent the standard Morton code $\mathcal{M}$, excluding the root node. The middle $dim * (l_{max} - l)$ bits are padding bits, and the last $l_0 = \lceil \log_2 l_{max} \rceil$ bits represent the level information. This encoding scheme allows the construction of an octree up to a maximum depth of 9 on a 32-bit system and a maximum depth of 19 on a 64-bit system. Given the locational code $\mathcal{L}$, the Morton code can conversely be derived as

$$\mathcal{M} = \mathcal{L} \gg dim * (l_{max} - l) + l_0, \tag{2}$$

where $\gg$ is the bitwise right shift operator.

4.1 Stack-Free Traversal for Internal Nodes

For a given quadrant, constructing the corresponding N-balanced MST is equivalent to finding all internal nodes. For example, to build the tree structure shown in Figure 4(b), it suffices to find all internal nodes (orange nodes) and then generate the four child nodes for each of them. This section explains how to identify all internal nodes in a stack-free parallel manner. The difficulty arises from a property known as the *ripple effect* [Sundar et al. 2008; Tu and O'Hallaron 2004]. This effect occurs when the refinement of a node triggers a sequence of splits, causing another node to split even if it is not in its immediate neighborhood. For simplicity, consider the binary tree demonstrated in Figure 7 as an example. If the leaf node a is further refined, it triggers the simultaneous splitting of leaf nodes i and j to maintain the 2:1 balance condition. Therefore, a straightforward implementation for finding nodes requiring splitting is to recursively identify neighbors that do not satisfy the balance condition. Unfortunately, a recursive implementation can trigger execution divergence in GPU implementation, which seriously degrades performance. A remedy could be using a stack to cache all neighbors, thus turning a recursive implementation into a loop-based one. Nevertheless, implementing a dynamic-sized stack on a GPU is still challenging. For instance, constructing a 2-balanced MST for a given octant may require a stack size of up to $O(2^{l_{max}})$ for each thread. To accelerate stack operations, the stack can be stored in shared memory. However, as l_{max} increases, the total memory required might exceed the shared memory capacity. Additionally, bank conflicts must be carefully avoided. Figure 6(a) demonstrates an implementation of constructing balanced trees using a stack. As the number of levels actually constructed, $l_{num} = l_{max} - l_{top} + 1$, increases, the performance of the stack-based approach rapidly deteriorates due to its frequent read and write operations on the stack.

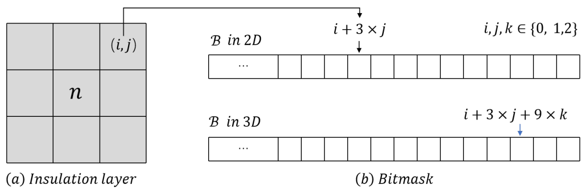


Fig. 8. (a) An illustration of the insulation layer around quadrant n; (b) The 32-bit *integer register* is used as a bitmask to identify nodes requiring further refinement.

To address the aforementioned issues, let us first present several key properties regarding the refinement of a leaf node in the N-balanced MST:

(1) If a leaf node is to be refined, only neighbors that do not belong to its sibling (hereafter referred to as **off-branch neighbors**) will be required to check the 2:1 balance condition along each direction (see Table 2 for more details);
(2) Refining a leaf node at level l will not trigger refinement for leaf nodes at or below this level ($\geq l$).
(3) As a leaf node undergoes refinement, the number of leaf nodes requiring further refinement at each higher level ($< l$) should not exceed 3^{dim}.

Taking the *0-balanced* binary tree in Figure 7 as an example, when the leaf node a is refined, the first principle indicates that we do not need to check the 2:1 balance condition between node a and q since it is automatically satisfied. Therefore, only the neighbor on the opposite side needs to be checked. In this scenario, leaf node i requires refinement due to a violation of the 2:1 balance condition. The second principle states that the refinement of node

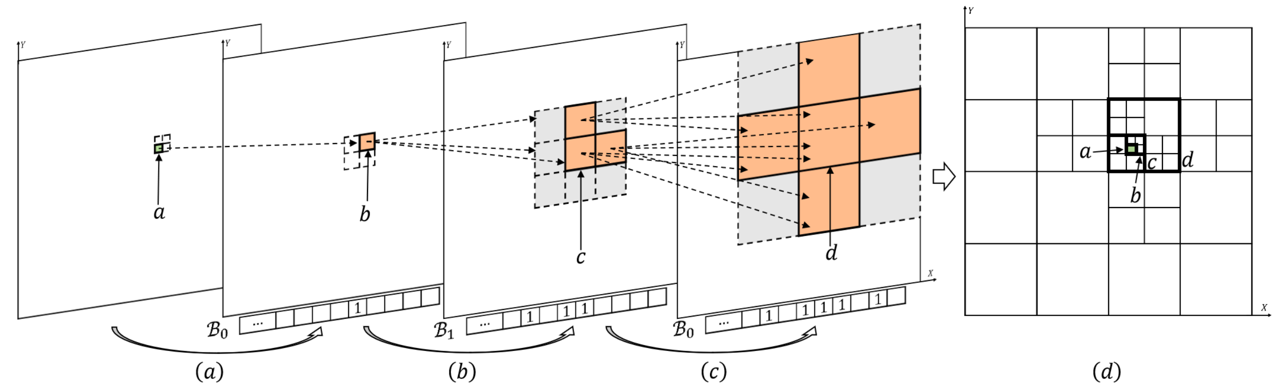


Fig. 9. An illustration of the stack-free parallel h-adaptation algorithm. (a) Node a, being a leaf with seed points, implies that its parent b requires refinement. (b) Node b then triggers its parent c as well as the parents of its off-branch neighbors $\mathcal{P}(\mathcal{N}_o(b))$, all of which lie within the insulation layer of node c. These nodes are identified, inserted into the hash table, and mapped into the integer register $\mathcal{B}_1$. (c) At the insulation layer of node d, the nodes requiring refinement are likewise identified, inserted into the hash table, and mapped into the integer register $\mathcal{B}_0$. (d) The final 1-balanced MST for quadrant a, obtained by generating four children for each internal node.

i cannot introduce violations for leaf nodes at the same level or lower levels, therefore, only node i and off-branch neighbors at a higher level (node j) require further refinement. To verify the third property, Figure 8(a) gives an illustration of the insulation layer around quadrant n, which is defined as the union of n and its potential neighbors at the same level. According to Sundar et al. [2008], no quadrant outside the insulation layer can compel n to split. In other words, the refinement of quadrant n or $\mathcal{D}(n)$ does not necessitate the splitting of quadrants outside the insulation layer. Therefore, the number of leaf nodes at each level that require splitting should not exceed 3^{dim}.

Inspired by the properties of an MST, our solution for identifying all internal nodes of an N-balanced MST involves assigning a thread to each leaf node, as demonstrated in Figure 9. For each thread, it is only necessary to check whether the refinement of the parent node triggers further refinement for nodes at higher levels. Taking the quadrant a in Figure 9 as an example, node a being a leaf with seed points implies that its parent b requires refinement, which can only trigger quadrants in the insulation layer of the nest parent node c. Therefore, the 32-bit *integer register* is sufficient, with the rightmost 3^{dim} bits indicating whether a node in the insulation layer requires further refinement, as demonstrated in Figure 8. After identifying the internal nodes within the insulation layer of node c, as shown in Figure 9(c), the traversal proceeds to the parents of the off-branch neighbors located within the insulation layer of the next parent node d. This procedure is repeated until either the 32-bit *integer register* is empty or the topmost level l_{top} is reached. All identified internal nodes are collected through a hash table similar to cuCollections [Juenger et al. 2023]. The insertion process is implemented using atomic operations to avoid conflicts between different threads. For more information about the hash table, refer to Section 6.1.1.

Algorithm 1 demonstrates our practical implementation on GPUs. Two 32-bit *integer registers*, $\mathcal{B}_0$ and $\mathcal{B}_1$, are used to track nodes requiring refinements in the insulation layer at each level. $\mathcal{S}_{dir}$ represents the set of all searching directions, where the elements contained in $\mathcal{S}_{dir}$ depend on the type of tree to be constructed. For example, when constructing the *1-balanced* quadtree in Figure 9, $\mathcal{S}_{dir}$ for the quadrant b in Figure 9(b) contains only two elements located in the positive x and y directions. Section 4.1.1 provides more details on how to calculate $\mathcal{S}_{dir}$. After identifying the off-branch neighbor n' (line 10), the locational code of its parent is attempted to be inserted into a hash table (line 12). If the location code is successfully inserted, the corresponding bit mask is set to one, indicating that this node could potentially trigger further refinements. If the location code insertion fails, it indicates that

Algorithm 1 Stack-Free Traversal of Internal Nodes

```
Define: $l_{top}, l_{max}$, and a leaf node $n$ at level $l_{max}$;
1:  int $\mathcal{B}_0 = 0, \mathcal{B}_1 = 0, l' = l_{max} - 1$;
2:  Initialize $\mathcal{B}_0$ from $\mathcal{M}(n)$;
3:  while $\mathcal{B}_0 \neq 0$ and $l' > l_{top}$ do
4:      for s = 0 to $3^{dim}$ do
5:          % See Figure 8
6:          if $((\mathcal{B}_0 \gg s)\&1) == 1$ then
7:              Calculate the node index $(n_x, n_y, n_z)$;
8:              % See section 4.1.1
9:              for each direction $d \in \mathcal{S}_{dir}$ do
10:                 Find the off branch neighbor $n'$;
11:                 Calculate the locational code $\mathcal{L}$ of $\mathcal{P}(n')$;
12:                 if Successfully insert $\mathcal{L}$ into hash table $\mathcal{H}$ then
13:                     Calculate the node index of $\mathcal{P}(n')$;
14:                     % See Figure 8
15:                     Map the index of $\mathcal{P}(n')$ in $\mathcal{I}(\mathcal{A}(n))$ to $\mathcal{B}_1$;
16:                 end if
17:             end for
18:         end if
19:     end for
20:     $\mathcal{B}_0 = \mathcal{B}_1, \mathcal{B}_1 = 0$;
21:     $l' \leftarrow l' - 1$
22: end while
```

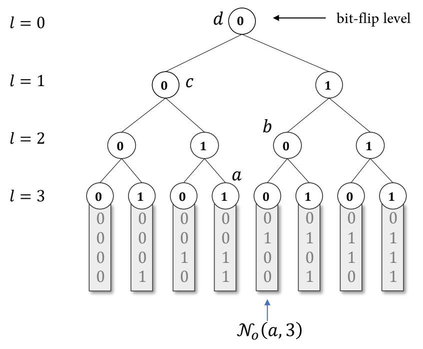


Fig. 10. A graphical illustration on the principle in finding off-branch neighbors. Note that detection of the bit-flip level is equivalent to finding the least common ancestor between two neighboring nodes.

the node has already been inserted by another thread. In this case, pruning is performed to avoid redundant traversal of the node. It is important to note that the while loop in lines 3~22 always iterates from level $l_{max} - 1$ to l_{top}, resulting in an overall time complexity of $O((l_{max} - l_{top}) * num(\mathcal{S}_{dir}) * 3^{dim})$. To achieve optimal performance, the processes for finding off-branch neighbors and removing duplicate nodes must be carefully optimized.

4.1.1 Identification of off-branch neighbors. To identify the off-branch neighbor in a specific direction, Hasbestan and Senocak [2018] first propose detecting the level at which the bit is flipped, followed by flipping all bits up to the identified bit-flip level. Detecting the bit-flip level thus involves a while loop with a time complexity of up to $O(l - l_{top})$. The challenge is whether it is possible to reduce the time complexity of finding an off-branch neighbor to $O(1)$?

Revisiting the problem of finding the off-branch neighbor with a binary tree demonstrated in Figure 10, we discovered that the detection of the bit-flip level in [Hasbestan and Senocak 2018] is equivalent to finding the least common ancestor (LCA) [Isaac et al. 2012] between the two neighboring nodes. Therefore, a more efficient way is by algebraic calculation. For example, consider the leaf node a. If the last bit of its Morton code is 1, its off-branch neighbor will be located on the positive side of the x-axis. Otherwise, it will be on the negative side. Given the node index as (n_x, n_y, n_z), the x-component of the node index for its off-branch neighbor at the same level can be easily calculated as

$$n'_x = (n_x \& 1 == 1) \; ? \; n_x + 1 : n_x - 1; \tag{3}$$

Since the above calculation can be performed independently for other axes, we can use a combination of n'_x, n'_y and n'_z to construct $\mathcal{S}_{dir}$ for all cases, as demonstrated in Table 2.

Table 2. An illustration on all possible searching directions.

	0-balanced	1-balanced	2-balanced
Binary Tree	n'_x		
Quadtree	$(n'_x,0)$ $(0,n'_y)$ (n'_x,n'_y)	$(n'_x,0)$ $(0,n'_y)$	
Octree	$(n'_x,0,0)$ $(0,n'_y,0)$ $(0,0,n'_z)$ $(n'_x,n'_y,0)$ $(n'_x,0,n'_z)$ $(0,n'_y,n'_z)$ (n'_x,n'_y,n'_z)	$(n'_x,0,0)$ $(0,n'_y,0)$ $(0,0,n'_z)$ $(n'_x,n'_y,0)$ $(n'_x,0,n'_z)$ $(0,n'_y,n'_z)$	$(n'_x,0,0)$ $(0,n'_y,0)$ $(0,0,n'_z)$

4.2 Construction of the Complete Tree

After the locational codes of all internal nodes are collected from the hash table, they are rearranged into a compact array $\mathcal{I}$ in descending order. As demonstrated in Figure 11, the final array $\mathcal{O}$, which represents N-balanced trees, consists of two components: a uniform section on the left, containing $t_0 = 2^{dim * l_{top}}$ nodes at the topmost level, and an adaptive section on the right, containing $t_1 * 2^{dim}$ nodes, where $t_1 = num(\mathcal{I})$. Each element in $\mathcal{O}$ contains the locational code of a node n along with the index j of its first child, expressed as $(\mathcal{L}, j)$. If $j = -1$, it indicates that the node has no children. The following procedure for constructing the complete tree consists of two stages. In the first stage, all missing leaf nodes are generated. In the second stage, parent-child relationships are established for all nodes, completing the tree.

4.2.1 Generate missing leaf nodes. By comparing the tree composed of internal nodes on the left side of Figure 11 with the balanced quadtree in Figure 4(b), it is evident that each internal node generates 2^{dim} child nodes, resulting in a complete and consistent tree structure. A total of $t_0 + t_1$ threads are assigned to generate the missing leaf nodes. If the thread ID t is less than t_0, a node is generated and stored at $\mathcal{O}[\mathcal{M}]$ (Algorithm 2 lines 6-7). Otherwise, 2^{dim} nodes, which are the children of $\mathcal{I}[t - t_0]$, are generated and stored consecutively starting from $\mathcal{O}[t_0 + (t - t_0) \cdot 2^{\text{dim}}]$ (Algorithm 2 lines 10-11). For example, node 2 in Figure 11(b) generates $C(2) = \{g, h, i, j\}$.

4.2.2 Set up parent-child relationships. Since the locational keys of internal nodes are sorted in descending order, the parent of an internal node is positioned after it, as illustrated in Figure 11(a). To identify the location of a parent node, Algorithm 2 first establishes parent-child relationships for internal nodes in the array $\mathcal{I}$, following the approach of Karras [2012] (lines 18-32). If the level of an internal node equals l_{top}, the location of its parent node is determined based on the index of the uniform grid (line 19). Otherwise, the locational code of the parent node is computed (line 21), and its position is determined through a binary search. The search process comprises two stages. In the first stage, a power-of-two upper bound is established by starting from 2 and doubling the value until the inequality is no longer satisfied (Lines 23-25). In the second stage, the range is narrowed by halving the value, starting from the upper bound, until the exact location is identified (Lines 27-31). Based on the parent-child relationships established for internal nodes in $\mathcal{I}$, the parent-child relationships for nodes in the new array $\mathcal{O}$ can be determined. Given a node at index t in $\mathcal{I}$, with its parent located at $\hat{t}$ (line 32), the children of node $\mathcal{I}[t]$ stored in $\mathcal{O}$ must be children to one of node $\mathcal{I}[\hat{t}]$'s children stored in $\mathcal{O}$ (lines 33-34). For example, in Figure 11, the parent of node 2, node 5, generates four children, $C(5) = \{e, f, 2, 4\}$, one of which must have a locational code equal to that of node 2. This completes the establishment of parent-child

Algorithm 2 Construction of the Complete Tree

```
Define: $\mathcal{I}$: An array of internal nodes in descending order;
Define: $\mathcal{O}$: An array of all nodes representing an $N$-balanced tree;
1:  $t_0 = 2^{dim*l_{top}}$;
2:  $t_1 = num(\mathcal{I})$;
3:  % Generate missing leaf nodes.
4:  for each thread ID $t \in [0, t_0 + t_1)$ do
5:      if $t < t_0$ then
6:          calculate the Morton code $\mathcal{M}$ and locational code $\mathcal{L}$;
7:          $\mathcal{O}[\mathcal{M}] \leftarrow (\mathcal{L}, -1)$;
8:      else
9:          for each $c \in \mathcal{C}(\mathcal{I}[t - t_0])$ do
10:             $i = \mathcal{M}(c)\&(2^{dim} - 1)$;
11:             $\mathcal{O}[t_0 + (t - t_0) * 2^{dim} + i] \leftarrow (\mathcal{L}(c), -1)$;
12:         end for
13:     end if
14: end for
15: % Set up parent-child relationships.
16: for each thread ID $t \in [0, t_1)$ do
17:     $n \leftarrow \mathcal{I}[t]$;
18:     if $l(n) = l_{top}$ then
19:         $\mathcal{O}[\mathcal{M}(n)].j = t_0 + t * 2^{dim}$;
20:     else
21:         $\bar{\mathcal{L}} \leftarrow \mathcal{L}(\mathcal{P}(n))$;
22:         $\gamma_{max} = 2$;
23:         while $\mathcal{L}(\mathcal{I}[t + \gamma_{max}]) \geq \bar{\mathcal{L}}$ do
24:             $\gamma_{max} = \gamma_{max} * 2$
25:         end while
26:         $\gamma = 0$;
27:         for $\beta \leftarrow \{\gamma_{max}/2, \gamma_{max}/4, \cdots, 1\}$ do
28:             if $\mathcal{L}(\mathcal{I}[t + \gamma + \beta]) \geq \bar{\mathcal{L}}$ then
29:                 $\gamma = \gamma + \beta$;
30:             end if
31:         end for
32:         $\hat{t} \leftarrow t + \gamma$;
33:         $i = \mathcal{M}(n)\&(2^{dim} - 1)$;
34:         $\mathcal{O}[t_0 + \hat{t} * 2^{dim} + i].j = t_0 + t * 2^{dim}$;
35:     end if
36: end for
```

relationships for nodes in $\mathcal{O}$. In the practical implementation, it is not necessary to explicitly store the parent-child relationships for internal nodes in the array $\mathcal{I}$. Therefore, a dotted line is used in Figure 11 to represent these relationships. In contrast, the solid line indicates the parent-child connections in the final tree.

5 DYNAMIC UPDATE BASED ON REFINEMENT COUNTERS

The dynamic update process involves locally inserting or removing a subset of MSTs from an existing balanced tree structure, thereby updating the tree while maintaining the balance constraint. This operation is extensively employed in practical simulation scenarios, such as updating the computational grid of the next time frame based on the current frame. While the insertion procedure has been thoroughly examined in Section 4.1, the remaining challenge is how to efficiently delete MSTs from the current balanced tree. A straightforward implementation is to use a counter to record how many MSTs share each node, and a node is only removed when all associated MSTs have been deleted. This requires each MST to be fully traversed during insertion. However, traversing all MSTs during insertion without early pruning can severely degrade performance as thread conflicts happen more frequently at higher levels. To address this, a pruning strategy is introduced during parallel MSTs insertion (see Algorithm 1, Line 12). The problem is: without traversing the whole MST, how can we get a counter to facilitate the deletion of MSTs? To address this, **refinement counters** are

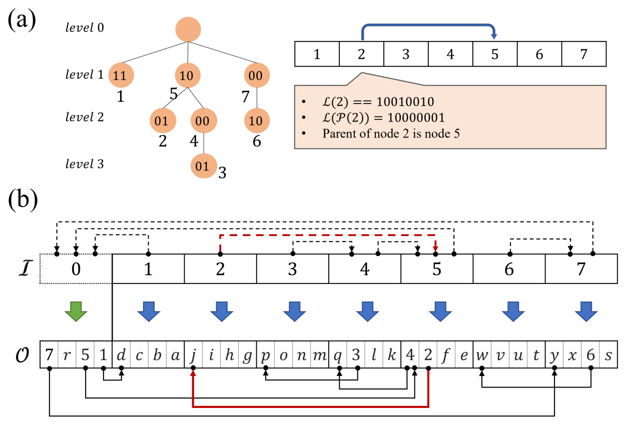

(a)
level 0
level 1
level 2
level 3
$\mathcal{L}(2)$ == 10010010
$\mathcal{L}(\mathcal{P}(2))$ = 10000001
Parent of node 2 is node 5
(b)
$\mathcal{I}$
$\mathcal{O}$

Fig. 11. An illustration on how to complete the construction of the N-balanced tree based on the sorted internal nodes. (a) The left side illustrates the tree composed of internal nodes in Figure 4(b). The right side presents the internal nodes arranged in descending order based on their locational codes. (b) The array $\mathcal{I}$ stores the locational codes of all internal nodes in descending order. The array $\mathcal{O}$ contains the final tree that contains a uniform section on the left and an adaptive section on the right. The algorithm for constructing the complete tree from internal nodes is demonstrated in Algorithm 2.

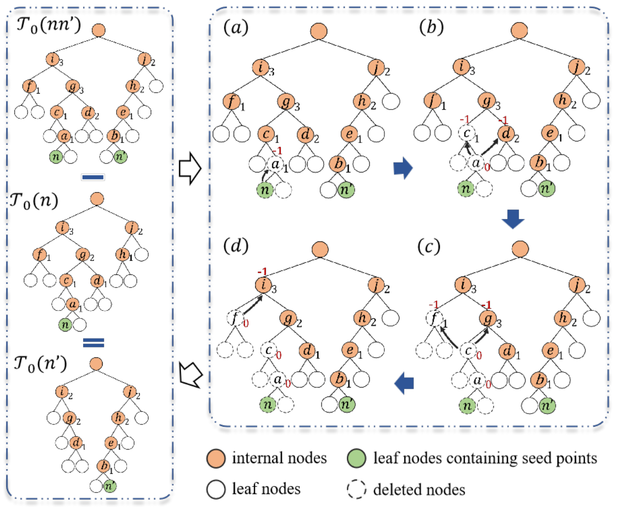


Fig. 12. Illustration of the deletion of $\mathcal{T}_0(n)$ using refinement counters in a 0-balanced binary tree.

Table 3. Node counts for octree representation of common models.

Models	$num(I_{seed})$	l_{max}	Non-balanced octree		2-balanced octree		1-balanced octree		0-balanced octree	
			Internal	All	Internal	All	Internal	All	Internal	All
Sphere	39,736	7	10,594 (27%)	84,760	11,934 (30%)	95,480	12,656 (32%)	101,256	12,958 (33%)	103,672
Bunny	126,757	8	33,190 (26%)	265,528	37,862 (30%)	302,904	40,047 (32%)	320,384	40,796 (32%)	326,376
Kitten	568,443	9	149,625 (26%)	1,197,008	169,446 (30%)	1,355,576	179,400 (32%)	1,435,208	183,425 (32%)	1,467,408
Armadillo	811,092	9	211,238 (26%)	1,689,912	238,198 (29%)	1,905,592	251,927 (31%)	2,015,424	256,709 (32%)	2,053,680
David head	1,074,792	9	284,613 (26%)	2,276,912	321,517 (30%)	2,572,144	340,080 (32%)	2,720,648	347,824 (32%)	2,782,600
Dragon	1,545,107	10	395,448 (26%)	3,163,592	444,216 (29%)	3,553,736	467,664 (30%)	3,741,320	476,879 (31%)	3,815,040
Budda	2,170,392	10	549,502 (25%)	4,396,024	620,742 (29%)	4,965,944	649,227 (30%)	5,193,824	658,909 (30%)	5,271,280
Ramesses	3,498,751	11	893,128 (26%)	7,145,032	1,005,744 (29%)	8,045,960	1,065,633 (30%)	8,525,072	1,081,131 (31%)	8,649,056
Asia dragon	17,111,461	12	4,429,642 (26%)	35,437,144	4,983,878 (29%)	39,871,032	5,272,086 (31%)	42,176,696	5,388,129 (31%)	43,105,040

utilized, which record the number of internal nodes that can trigger the refinement of the current node. During MST deletion, when a node is visited, its refinement counter is decremented by one. If the counter still remains greater than zero, it indicates that other non-deleted MSTs can still reach this node, indicating the current node should not be deleted, and the current traversal is stopped. Otherwise, the node is removed, and the traversal continues. Finally, all internal nodes with refinement counters greater than zero are collected, and the complete balanced tree is constructed according to the method described in Section 4.2.

The refinement counters are created only immediately prior to the deletion of MSTs. For any internal node n, its refinement counter is defined as follows:

(1) If n has a child that is a leaf node containing seed points, then its refinement counter is 1;
(2) Otherwise, its refinement counter equals the number of internal nodes among its children and the off-branch neighbors of its children.

As illustrated in the 0-balanced binary tree $\mathcal{T}_0(nn')$ in Figure 12, internal node a has a child node o that is a leaf containing seed points, the refinement of a is triggered, and its refinement counter is 1. For other internal nodes, such as g, its children c and d, along with d's off-branch neighbor e, are all internal nodes. The refinement of g is triggered by traversals from c, d, and e, resulting in a refinement counter of 3.

MSTs deletion traverses all internal nodes following the same procedure as insertion. Figure 12 illustrates the deletion of $\mathcal{T}_0(n)$ in a 0-balanced binary tree. As shown in Figure 12(b), when node d is visited and its refinement counter is decremented by 1, the counter remains greater than zero, indicating that other MSTs (e.g., $\mathcal{T}_0(n')$) will still access this node; thus, the node is not deleted, and the traversal stops. In contrast, when node c is visited and its refinement counter reaches zero, the node can be safely removed, and the traversal continues (Figure 12(c)). Refinement counters ensure that an internal node is deleted only after all internal nodes that trigger its refinement have been removed—that is, after all MSTs that access it have been deleted—thereby guaranteeing the maintenance of the balance constraint during dynamic updates. The MST deletion algorithm only needs to replace Line 12 in Algorithm 1 with the operation of decrementing the node's refinement counter and checking whether it has reached zero. All other steps remain the same as in the insertion process. Consequently, the time complexity of the deletion process is also $O((l_{max} - l_{top}) * num(\mathcal{S}_{dir}) * 3^{dim})$.

To summarize, the dynamic update process based on the refinement counter can be described as follows:

(1) Insert all internal nodes of the N-balanced tree into a new hash table and calculate their refinement counters;
(2) Traverse the MSTs of seed points to be deleted and update the counters of relevant nodes;
(3) Traverse the MSTs of seed points to be inserted, setting the counters of newly inserted internal nodes to one;
(4) Collect internal nodes with counters greater than zero from the hash table and construct the complete tree structure based on Algorithm 2.

6 RESULTS AND COMPARISONS

Unless explicitly stated, the experiments and comparisons are performed on a PC equipped with an Intel Xeon W 2245 CPU, 64 GB of RAM, and an NVIDIA GeForce RTX 3090 GPU with 24 GB of memory. Our algorithm is implemented in C++17 with all computationally intensive parts parallelized in CUDA 12.4.

6.1 Performance Evaluation

6.1.1 Parameter selection for the hash table. The hash tables used in Algorithm 1 have the following configurations:

(1) The capacity of the hash table is fixed at m;
(2) The hashing-by-division scheme [Mehta and Sahni 2004] is applied to map an input key into an array index for the hash table:
$$h(k) = (k \times p)\%m, \tag{4}$$
where k is the input key, p is a prime number, and % is the modulo operation;
(3) As a collision occurs, the open addressing with linear probing is applied [Pagh and Rodler 2004];
(4) The key value zero is designated as a sentinel value to signify an empty bucket.

To determine an appropriate value for the hash table size m and prime number p, a series of experiments is conducted. The size of the hash table is defined as $m = \alpha \cdot num(I_{seed})$, where I_{seed} represents the array of seed points generated by sampling the given model, with duplicates removed, and α is a scaling factor. The suitable range of α will be determined through experiments to ensure that

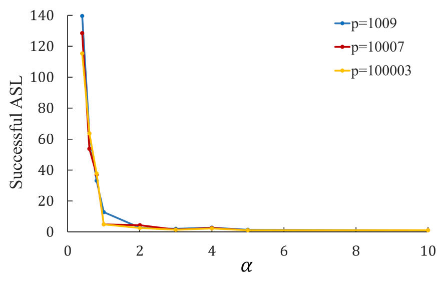


Fig. 13. The performance of the hash table varies with changes in the parameter α and the prime number p. The successful average search length (ASL) is defined as the average number of comparisons needed when inserting an element into the hash table.

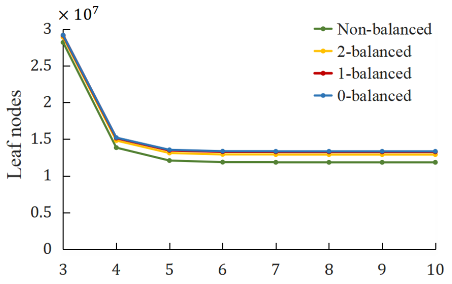


Fig. 14. The variation trends of the number of leaf nodes with respect to the level number l_{num} and types of balance constraints.

the hash table is pre-allocated with sufficient space. Table 3 provides statistics on octree representations for common models. Varying numbers of seed points are sampled from these diverse models and used to construct different types of octrees, including 2-balanced, 1-balanced, 0-balanced, and non-balanced octrees. The *Internal* column represents the number of internal nodes, while the *All* column indicates the number of all nodes, including internal and leaf nodes. Our hash table is specifically designed to store internal nodes, whose count is significantly smaller than the count of all nodes, regardless of whether the octree is balanced or non-balanced. The values in parentheses next to the *Internal* column indicate the percentage of internal nodes relative to the number of seed points. This percentage serves as the minimum threshold that the constant α must exceed to ensure that the allocated hash table space is adequate to meet the storage requirements. Based on the empirical data from the diverse models in Table 3, the value of α is recommended to take values in the range$[0.4, +\infty)$.

With the range of values for the constant α established, additional experiments are conducted to select an optimal value. Figure 13 shows the performance of the hash table across various α values, using the 0-balanced octree representation of the armadillo model as a case study. Considering that the elements in the hash table are collected after insertion rather than being searched within, the key metric used to evaluate the hash table is the successful average search length (ASL), defined as the average number of comparisons needed when inserting an element into the hash table. The experimental results indicate that as α increases, the growth in hash table size leads to a gradual reduction in the successful ASL, which aligns with the theoretical expectation of an asymptotic approach to $O(1)$ behavior for hash table operations as the table becomes sufficiently large. Based on the empirical data in Figure 13, the recommended range of values for α is $[1, 3]$. Combined with Table 3, the corresponding load factor ranges of approximately 10% to 30%. Additionally, the impact of the prime number p for the hash function is also tested through experiments, revealing that the choice of p has only a minor effect on overall hash table performance. In subsequent experiments, the parameters for the hash table implementation are set to $\alpha = 3$ and $p = 100003$.

In dynamic update mode, the size of the hash table is defined as $m = \beta \cdot num(I_{internal})$, where $I_{internal}$ represents the array of internal nodes from the existing N-balanced tree. Considering that nodes in $I_{internal}$ will be inserted into the hash table along with updating part seed points, the recommended range for β is $[2, 5]$, corresponding to a load factor range of approximately 20% to 50%. In the experiments adopting dynamic update mode described in this paper, β is consistently set to 3.

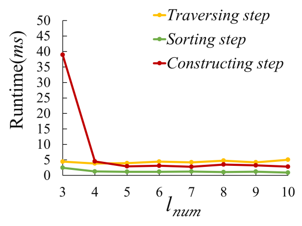
6.1.2 *Parameter selection for tree construction.*

This section presents an experimental analysis of the effects of different parameter settings, including the types of balance constraints and the level number l_{num}, on the algorithm's performance. All experiments are conducted using balanced trees constructed from the Stanford Bunny model with $l_{max} = 10$. Figure 2 illustrates octrees of the bunny model under different balance constraints. Figure 14 shows the trends in the number of leaf nodes with respect to the level number l_{num} and the types of balance constraints. The results indicate that as l_{num} increases, the adaptability of the balanced tree improves, leading to a gradual decrease in the number of leaf nodes. For any fixed l_{num}, stronger balance constraints produce a larger number of leaf nodes. These trends are consistent with intuitive expectations.

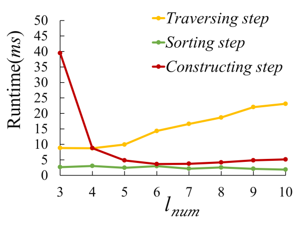
As described in Section 4, the balanced tree construction process consists of three main steps: (1) **traversing** all internal nodes using Algorithm 1; (2) **sorting** the internal nodes; and (3) **constructing** the complete balanced tree via Algorithm 2. Figure 15 presents the time consumption trends for each component under different types of balance constraints and level numbers l_{num}. Notably, the sorting step operates solely on the internal node array and exhibits stable time consumption, posing no bottleneck to further efficiency improvements. The time consumption of Algorithm 2 follows a trend similar to the variation in the number of leaf nodes (Figure 14), decreasing gradually as l_{num} increases and eventually stabilizing without limiting the overall efficiency. In contrast, the computational cost of Algorithm 1 increases with both the level number l_{num} and the strength of the balance constraints, eventually becoming the dominant cost in the construction process.

6.1.3 *Parameter selection for dynamic tree updates.*

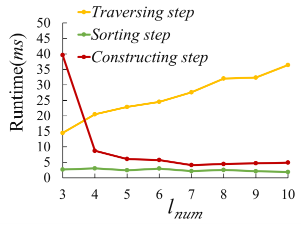
This section presents an experimental analysis of the dynamic update performance under different parameter choices, including the type of balance constraint, the level number l_{num}, and the update ratio of seed points. Since the main difference between dynamic updates mode (DUM) and the reconstruction-from-scratch mode (RFSM) lies



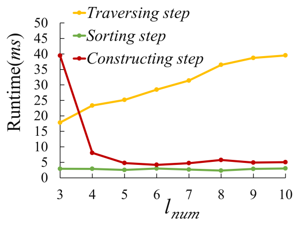
(a) Non-balanced octree



(b) 2-balanced octree



(c) 1-balanced octree



(d) 0-balanced octree

Fig. 15. The variation trends in time consumption for the three main steps of the construction process under different balance constraints and level numbers l_{num}.

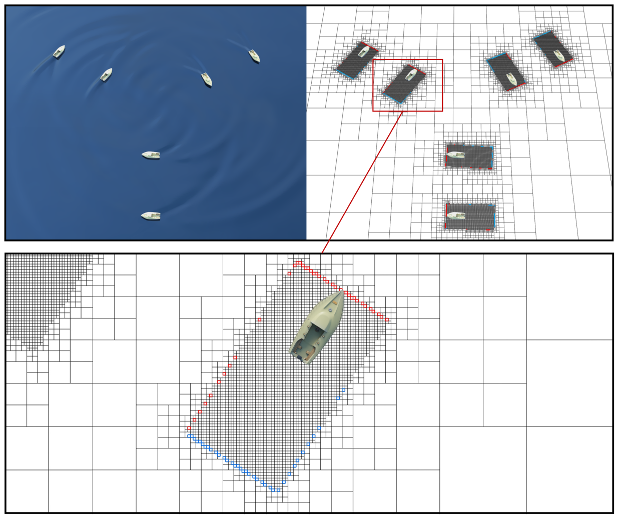


Fig. 16. The upper left figure depicts the demo: the yachts on the lake, while the upper right shows the lake represented by a 1-balanced quadtree. The lower figure features a close-up of the nodes surrounding a yacht, where the red nodes represent the increased seed nodes, and the blue nodes indicate the decreased seed nodes.

in the traversing step, which is also the performance bottleneck of the RFSM, this section focuses on the comparison of this step. The experiments in this section include two sets: quadtrees in 2D and octrees in 3D.

Figure 16 illustrates a shallow water wave simulation scenario, where the lake surface is represented using a 1-balanced quadtree with l_{max} = 14. Figure 17 illustrates the variation trend of the time ratio between the dynamic update and reconstruction-from-scratch modes with respect to level number (l_{num}) and the seed

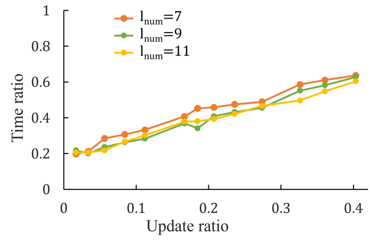
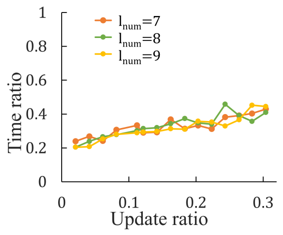
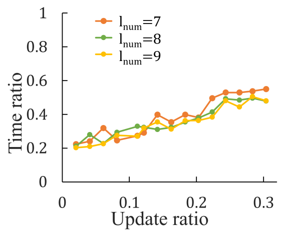


Fig. 17. For the 1-balanced quadtree, the variation trend of the time ratio between the dynamic update and construction-from-scratch modes with respect to level number (l_{num}) and the seed nodes update ratio.



(a) 2-balanced octree



(b) 0-balanced octree

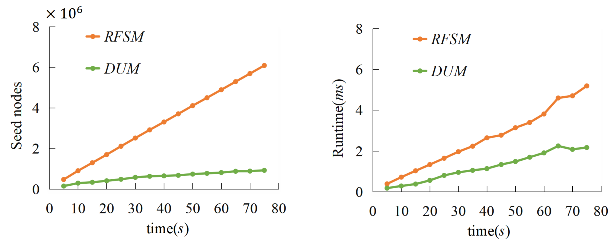
Fig. 18. For the octree, the variation trend of the time ratio between the dynamic update and the construction-from-scratch modes with respect to the type of balance constraint, the level number l_{num}, and the seed nodes update ratio.

nodes update ratio. The seed nodes are defined as the nodes within the bounding box surrounding the yacht, and the update ratio of seed nodes is controlled by adjusting the yacht's speed. Figure 18 illustrates the variation trend corresponding to octree constructed from the Stanford Bunny model with l_{max} = 12. The dynamic update process requires first inserting the internal nodes into a hash table and calculating their refinement counters. Consequently, when the update ratio of seed nodes approaches zero, the dynamic update mode incurs a certain proportion of base time overhead, accounting for approximately 20% of the time consumption of construction. Aside from the base time overhead, the time ratio increases almost proportionally with the seed point update ratio, which aligns with the theoretical expectation: the time complexity of deleting MSTs during dynamic updates is equivalent to that of the insertion process. Considering the presence of base time overhead, we recommend reconstructing the balanced tree structure from scratch when the seed point update ratio exceeds 70%.

Figure 19 illustrates the variation of the number of seed nodes and the runtime over time under the realistic waterfall scenarios shown in Figure 24. In this scenario, particles are continuously emitted from the source at a fixed rate, causing the number of particles to gradually increase and the flow domain to expand. As shown in Figure 19(a), the number of seed nodes—defined as leaf nodes containing particles—increases steadily, but at a slower rate in the DUM than in the RFSM. Figure 19(b) further demonstrates that as the number of seed nodes grows and the flow domain expands, the runtime of both modes increases, while the computational cost of DUM is significantly lower than that of RFSM.

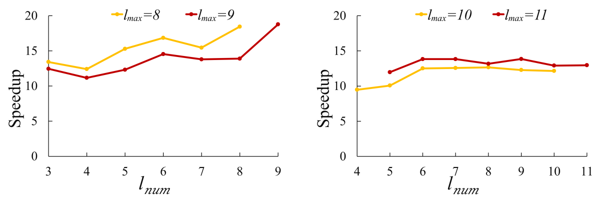
Table 4. The comparison of computational costs between our method and the bottom-up approach in constructing balanced and non-balanced octrees.

Models	l_{max}	l_{num}	Non-balanced octree			2-balanced octree			1-balanced octree			0-balanced octree		
			Leaf nodes	Bottom-up(*ms*)	Ours(*ms*)	Leaf nodes	Bottom-up(*ms*)	Ours(*ms*)	Leaf nodes	Bottom-up(*ms*)	Ours(*ms*)	Leaf nodes	Bottom-up(*ms*)	Ours(*ms*)
Sphere	7	4	77,078	5.31	**0.50 (10.69×)**	85,730	6.00	**0.52 (11.54×)**	90,056	11.16	**0.74 (15.01×)**	92,114	16.52	**0.75 (22.03×)**
Bunny	8	5	235,705	12.12	**1.17 (10.36×)**	267,324	15.90	**1.38 (11.52×)**	282,087	15.55	**1.25 (12.44×)**	287,204	16.05	**1.27 (12.67×)**
Kitten	9	5	1,077,056	30.63	**2.46 (12.47×)**	1,212,128	41.13	**2.47 (16.67×)**	1,280,105	41.57	**2.99 (13.92×)**	1,307,839	41.47	**3.04 (13.63×)**
Armadillo	9	5	1,507,080	33.23	**2.76 (12.04×)**	1,690,669	46.93	**3.25 (14.43×)**	1,784,308	46.04	**3.86 (10.77×)**	1,817,271	50.01	**3.65 (13.71×)**
David head	9	5	2,019,081	35.90	**3.42 (10.51×)**	2,271,900	54.09	**3.69 (14.67×)**	2,399,216	56.76	**3.86 (14.69×)**	2,452,570	67.18	**4.40 (15.26×)**
Dragon	10	5	3,022,839	40.18	**4.16 (9.66×)**	3,356,361	52.00	**4.48 (11.61×)**	3,517,011	62.59	**5.77 (10.84×)**	3,580,557	60.23	**6.14 (9.81×)**
Budda	10	6	3,877,021	69.73	**4.99 (13.97×)**	4,372,859	94.25	**6.52 (14.46×)**	4,570,819	107.50	**7.62 (14.11×)**	4,638,341	118.24	**7.83 (15.11×)**
Ramesses	11	6	6,509,665	88.73	**7.54 (11.77×)**	7,292,937	115.58	**9.46 (12.21×)**	7,708,898	196.59	**12.57 (15.64×)**	7,816,747	188.58	**12.11 (15.57×)**
Asia dragon	12	7	31,264,437	831.21	**38.90 (21.37×)**	35,136,438	1237.43	**63.89 (19.37×)**	37,150,331	1328.11	**68.71 (19.33×)**	37,961,834	1343.19	**74.15 (18.11×)**



(a) The number of seed nodes (b) Runtime

Fig. 19. Number of seed nodes and runtime over time in the realistic waterfall scenarios of Figure 24.



(a) Bunny (b) Armadillo

Fig. 20. The speedup of our method over the bottom-up algorithm varies with l_{num} under different models and constraints. (a) Experiments on the 1-balanced octree representation of the Stanford Bunny model with l_{max} = 8, 9. (b) Experiments on the 0-balanced octree representation of the armadillo model with l_{max} = 10, 11.

6.2 Performance Comparison

To validate the efficiency of our method, a series of comparative experiments is conducted with state-of-the-art approaches. These include comparisons with the bottom-up method for both balanced and non-balanced tree constructions, as well as comparisons with the non-balanced tree construction proposed by Karras [2012] and the balanced tree construction presented by Wang et al. [2024].

6.2.1 Comparison with the bottom-up algorithm. The GPU-based bottom-up algorithm is implemented following the approach outlined in [Zhou et al. 2010], where the original work does not enforce the balance constraint. For the balancing refinement process, we adopt a strategy similar to that in [Teunissen and Ebert 2018]. This approach parallelizes the refinement of all leaf nodes, checking their neighbors to verify if the constraint is satisfied. If the constraint is not met, the leaf nodes are further subdivided, with each node allowed to change its level by at most one level at a time. Table 4 compares nine commonly used models in computer graphics, including both non-balanced and balanced octrees. Regardless of octree type, our algorithm achieves significant speedups, ranging from 9.66× to 22.03×. Figure 20 compares our method with the bottom-up approach with respect to l_{num}. Specifically, Figure 20(a) shows the construction of 1-balanced octrees for the Bunny model with maximum depths of l_{max} = 8 and l_{max} = 9, while Figure 20(b) illustrates the construction of 0-balanced octrees for the Armadillo model with l_{max} = 10 and l_{max} = 11. The results indicate that across various models, balance constraints, and parameter settings, our method consistently achieves speedups exceeding an order of magnitude compared to the bottom-up approach.

6.2.2 Comparison with Karras [2012]. Table 5 compares our method with the approach proposed by Karras [2012]. Notably, the method described in [Karras 2012] is limited to constructing non-balanced tree structures across all levels. Therefore, the comparative experiments are conducted under the condition that l_{num} is set to l_{max}. The octree constructed using the method of Karras [2012] is not complete, meaning that some internal and leaf nodes are missing, so that each internal node does not necessarily have exactly eight children. As a result, additional operations are required to generate all missing internal and leaf nodes, which introduces significant time overhead. In contrast, our method naturally generates a complete tree covering the entire domain. As shown in Table 5, our method achieves a speedup ranging from 2.05× to 3.53× in constructing non-balanced octrees. Under identical model conditions and parameters, our method can even construct balanced octrees more efficiently than the approach in [Karras 2012] for non-balanced octree construction.

6.2.3 Comparison with Wang et al. [2024]. Figure 21 presents a comparison between our method and the approach proposed by Wang et al. [2024], using examples from their paper, as their method's open-source code is unavailable. To ensure a fair evaluation, we measured the computational costs of our approach on a computer with specifications matching those used by the authors. The system is equipped with an Intel i7-8700 CPU (6 cores) and an NVIDIA GeForce RTX 2060 GPU with 1920 CUDA cores and 6 GB of memory. The comparative experiment focuses on constructing a 1-balanced octree representation of the bunny model. For maximum tree depths of l_{max} = 9 and l_{max} = 10, the topmost level is set to l_{top} = 4, resulting in approximately 0.4 million and 0.9 million leaf nodes, respectively, which aligns with the parameters used in their experiments. Their approach involves completing tree construction

Table 5. The comparison of computational costs between our method and the approach proposed by Karras [2012] in constructing non-balanced octrees.

Models	l_{max}	Non-balanced octree			2-balanced octree		1-balanced octree		0-balanced octree	
		Leaf nodes	[Karras 2012](ms)	Ours(ms)	Leaf nodes	Ours(ms)	Leaf nodes	Ours(ms)	Leaf nodes	Ours(ms)
Sphere	7	74,166	1.73	**0.49 (3.53×)**	83,546	0.52	88,600	0.56	90,714	0.58
Bunny	8	232,338	2.79	**1.01 (2.77×)**	265,042	1.62	280,337	1.82	285,580	2.12
Kitten	9	1,047,383	6.69	**2.48 (2.70×)**	1,186,130	3.32	1,255,808	3.64	1,283,983	3.81
Armadillo	9	1,478,674	9.70	**3.42 (2.84×)**	1,667,394	3.70	1,763,497	4.39	1,796,971	4.56
David head	9	1,992,299	11.56	**3.79 (3.05×)**	2,250,627	4.65	2,380,568	5.70	2,434,776	5.58
Dragon	10	2,768,144	12.28	**5.24 (2.35×)**	3,109,520	6.98	3,273,656	7.99	3,338,161	8.40
Budda	10	3,846,522	19.17	**5.85 (3.28×)**	4,345,202	8.68	4,544,597	10.06	4,612,371	10.52
Ramesses	11	6,251,904	22.00	**10.74 (2.05×)**	7,040,216	15.65	7,459,439	17.68	7,567,925	19.65
Aisa dragon	12	31,007,502	162.93	**48.05 (3.27×)**	34,887,154	83.56	36,904,610	94.91	37,716,911	102.82

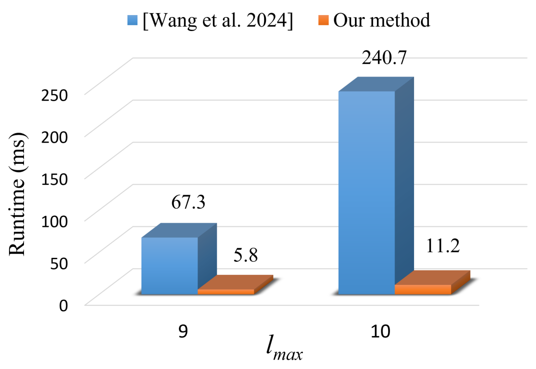


Fig. 21. The comparison of computational cost between our method and approach presented by Wang et al. [2024] in constructing a 1-balanced octree representation of the Stanford Bunny model.

before performing the balancing refinement, leading to the evaluation of the computational costs for these two stages separately. In contrast, our method performs both tree construction and balancing refinement concurrently. As a result, the time cost of our approach is compared with the cumulative cost of their two-stage process. The method of Wang et al. [2024] requires a sorting algorithm at every level, and each level is processed sequentially. In contrast, in our method, sorting is applied only once to the internal node array, and each thread can independently construct the MST for its corresponding seed point. This high degree of parallelism makes our method significantly more efficient. The results in Figure 21 show that when $l_{max} = 9$ and $l_{max} = 10$, our method achieves speeds 11.60 times and 21.44 times faster, respectively, than the method proposed by Wang et al. [2024].

7 APPLICATIONS

7.1 Setup Neighbor Lists for Balanced Trees

Neighbor lists record the indices of each node's neighbors, enabling efficient access to neighborhood information during applications. The definition of a neighbor may vary depending on the specific use case. For example, in the Eulerian fluid simulation described in Section 7.2, only face-adjacent neighbors are considered. In contrast, the particle neighborhood search in Section 7.3 requires neighbors that are adjacent by face, edge, or vertex. Despite these differences, the construction of neighbor lists follows a unified procedure: identifying neighboring nodes at the same level for a given node n through a top-down traversal. During this process, three cases may occur:

(1) A leaf node is reached at a level higher than $l + 1$.
(2) A leaf node is reached at level l.
(3) An internal node is reached at level l.

The construction of neighbor lists begins by assigning one thread to each leaf node. For a leaf node at level l, the algorithm searches for its neighboring nodes at the same level. If the search results in the first case, where the neighboring node in the given direction is at a higher level than l, the neighbor list of node n is updated, and the index of node n is also written into the neighbor list of its neighbor. In the second case, only the neighbor list of node n is updated. In the third case, the update has already been handled by the neighboring node, so no further action is required. Following this procedure, the neighbor lists for all leaf nodes are correctly constructed.

7.2 Adaptive Grid for Eulerian Fluids

Balanced tree structures are commonly used in fluid simulations because of their beneficial properties. Figure 23 depicts an hourglass scenario where the motion of water is represented by adaptive grids composed of the leaf nodes of a 2-balanced octree, while the movement of sand is modeled using particles. The simulation process is divided into four steps:

(1) Solve the dynamics of sand particles based on the work of Macklin et al. [2014].
(2) Transfer the velocities of sand particles onto the adaptive grid through interpolation.
(3) Solve the pressure Poisson equation on the adaptive grid, following the methodology described by Losasso et al. [2004], to ensure incompressibility.
(4) Transfer the velocity field back to sand particles by interpolating the velocity changes on the adaptive grid.

The interpolation between the adaptive grid and the particles is implemented based on the method proposed in [Setaluri et al. 2014], utilizing several auxiliary structures, including the vertex array of the adaptive grid, an index structure that maps grid cells to vertices, and another that maps vertices to grid cells. In Step (3), solving the pressure Poisson equation is supported by the neighbor lists of leaf nodes. Figure 22 presents a statistical analysis of the computational

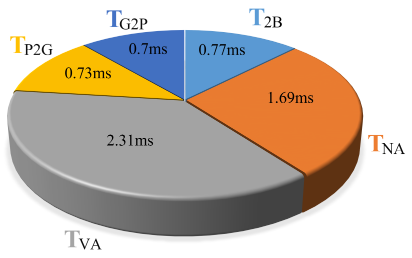


Fig. 22. Statistics on computational cost: T_{2B} is the construction of 2-balanced octree. T_{NA} is the construction of neighborhood auxiliary structures. T_{VA} is the construction of vertices-related auxiliary structures. T_{P2G} is the interpolation process from particles to adaptive grids. T_{G2P} is the interpolation process from adaptive grids to particles.

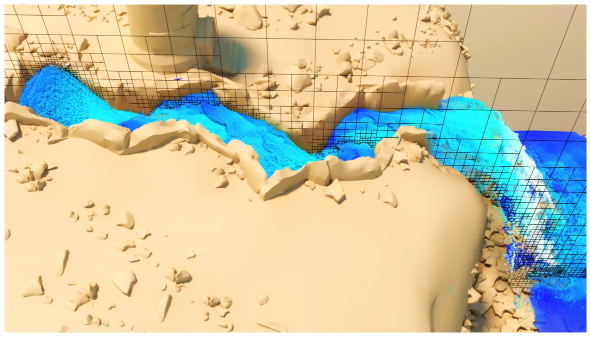


Fig. 24. Two-dimensional cross-sectional representations of octrees in the waterfall scenario.

costs associated with these processes, where the 2-balanced octree is constructed with $l_{max} = l_{num} = 9$. Figure 23 shows the evolution of water and sand over time, with the grid structure adapting accordingly. The efficiency of our algorithm provides robust support for this simulation process.

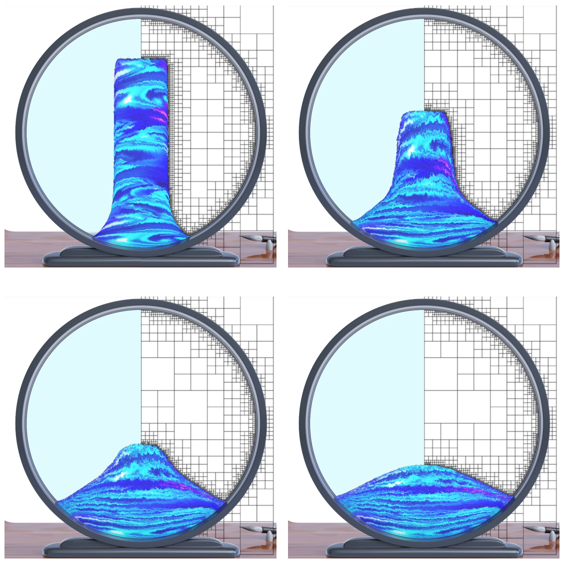


Fig. 23. Simulation of hourglass at different moments.

7.3 Neighborhood Search for Particle Fluids

As described in [Fernández-Fernández et al. 2022], octrees are effective tools for neighborhood search. In this application, because particle positions change continuously, the octree structure must also evolve over time, requiring efficient construction and updating. Figures 3, 5, and 24 illustrate three fluid scenarios and show two-dimensional cross-sections of the 2-balanced octrees used for neighborhood search in each case. By establishing an index mapping between fluid particles and the leaf nodes of the octree, the octree together with the structure storing the neighbors of leaf nodes can be leveraged to accelerate the neighborhood search for particles located within the support domain. In the current implementation, we set the smoothing length as a constant. Due to the adaptivity of the octree, the structure is also suitable for particle fluids with varying support radii.

8 CONCLUSIONS AND LIMITATIONS

We propose a novel framework that enables the simultaneous construction of trees and solving of balance constraints. This framework allows for the construction of both balanced and non-balanced trees and exhibits generality in the construction of octrees, quadtrees, and binary trees. We achieve not only stack-free N-balanced MSTs construction but also eliminate the iterative process caused by ripple effect. Furthermore, we implement an efficient solution based on refinement counters to dynamically update balanced trees, supporting both parallel insertion and deletion operations. Our method is highly suitable for GPU architectures, resulting in significant efficiency improvements, as demonstrated by comprehensive comparative experiments. Our efficient algorithm provides strong support for fluid simulations with dynamic boundaries.

The primary limitation of our method arises in extreme cases where the computational domain is large but the seed distribution is sparse (e.g., with only one seed). In such cases, the estimated hash table size may be insufficient to store all internal nodes. In other words, there is no universal rule for selecting optimal hash table parameters across all scenarios. Second, the depth of the tree can affect performance. Achieving optimal performance may depend on the user's experience in selecting appropriate parameters. Finally, the current implementation confines seed nodes to the same level. Given that the stack-free traversal strategy for MST construction is inherently agnostic to the levels of the seed nodes, this observation points to a promising avenue for future work: extending the algorithm to support balanced tree construction from seed nodes distributed across multiple levels.

ACKNOWLEDGMENT

We would like to thank anonymous reviewers for their valuable comments on improving the paper. The project was supported by the National Natural Science Foundation of China (No.62332015, 62302490), and the Basic Research Project of ISCAS (No.ISCAS-JCMS-202403).

REFERENCES

Mridul Aanjaneya, Ming Gao, Haixiang Liu, Christopher Batty, and Eftychios Sifakis. 2017. Power Diagrams and Sparse Paged Grids for High Resolution Adaptive Liquids. *ACM Trans. Graph.* 36, 4, Article 140 (jul 2017), 12 pages. https://doi.org/10.1145/3072959.3073625

Prekshu Ajmera, Rhushabh Goradia, Sharat Chandran, and Srinivas Aluru. 2008. Fast, parallel, GPU-based construction of space filling curves and octrees. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games* (Redwood City, California) *(I3D '08)*. Association for Computing Machinery, New York, NY, USA, Article 10, 1 pages. https://doi.org/10.1145/1342250.1357022

Ryoichi Ando and Christopher Batty. 2020. A Practical Octree Liquid Simulator with Adaptive Surface Resolution. *ACM Trans. Graph.* 39, 4, Article 32 (aug 2020), 17 pages. https://doi.org/10.1145/3386569.3392460

Christopher Batty. 2017. A cell-centred finite volume method for the Poisson problem on non-graded quadtrees with second order accurate gradients. *J. Comput. Phys.* 331 (2017), 49–72. https://doi.org/10.1016/j.jcp.2016.11.035

Carsten Burstedde, Lucas C Wilcox, and Omar Ghattas. 2011. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing* 33, 3 (2011), 1103–1133.

Xiao-Rui Chen, Min Tang, Cheng Li, Dinesh Manocha, and Ruo-Feng Tong. 2022. BADF: Bounding Volume Hierarchies Centric Adaptive Distance Field Computation for Deformable Objects on GPUs. *Journal of Computer Science and Technology* 37 (06 2022), 731–740. https://doi.org/10.1007/s11390-022-0331-x

José Antonio Fernández-Fernández, Lukas Westhofen, Fabian Löschner, Stefan Rhys Jeske, Andreas Longva, and Jan Bender. 2022. Fast Octree Neighborhood Search for SPH Simulations. *ACM Transactions on Graphics (TOG)* 41, 6 (2022), 1–13.

Florian Ferstl, Rüdiger Westermann, and Christian Dick. 2014. Large-scale liquid simulation on adaptive hexahedral grids. *IEEE transactions on visualization and computer graphics* 20, 10 (2014), 1405–1417.

Ryan Goldade, Yipeng Wang, Mridul Aanjaneya, and Christopher Batty. 2019. An Adaptive Variational Finite Difference Framework for Efficient Symmetric Octree Viscosity. *ACM Trans. Graph.* 38, 4, Article 94 (jul 2019), 14 pages. https://doi.org/10.1145/3306346.3322939

Arthur Guittet, Maxime Theillard, and Frédéric Gibou. 2015. A stable projection method for the incompressible Navier–Stokes equations on arbitrary geometries and adaptive Quad/Octrees. *J. Comput. Phys.* 292 (2015), 215–238. https://doi.org/10.1016/j.jcp.2015.03.024

Jaber J. Hasbestan and Inanc Senocak. 2018. Binarized-octree generation for Cartesian adaptive mesh refinement around immersed geometries. *J. Comput. Phys.* 368 (2018), 179–195. https://doi.org/10.1016/j.jcp.2018.04.039

Tobin Isaac, Carsten Burstedde, and Omar Ghattas. 2012. Low-Cost Parallel Algorithms for 2:1 Octree Balance. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 426–437. https://doi.org/10.1109/IPDPS.2012.47

Daniel Juenger, Nico Iskos, Yunsong Wang, Jake Hemstad, Christian Hundt, and Nikolay Sakharnykh. 2023. Maximizing Performance with Massively Parallel Hash Maps on GPUs. https://developer.nvidia.com/blog/maximizing-performance-with-massively-parallel-hash-maps-on-gpus/#entry-content-comments

Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Paris, France) *(EGGH-HPG'12)*. Eurographics Association, Goslar, DEU, 33–37.

Byungmoon Kim, Panagiotis Tsiotras, Jeong-Mo Hong, and Oh-young Song. 2015. Interpolation and parallel adjustment of center-sampled trees with new balancing constraints. *The Visual Computer* 31 (2015), 1351–1363.

Yeojin Kim, Byungmoon Kim, and Young J Kim. 2018. Dynamic deep octree for high-resolution volumetric painting in virtual reality. In *Computer Graphics Forum*, Vol. 37. Wiley Online Library, 179–190.

Dan Koschier, Crispin Deul, and Jan Bender. 2016. Hierarchical hp-adaptive signed distance fields. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Zurich, Switzerland) *(SCA '16)*. Eurographics Association, Goslar, DEU, 189–198.

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH construction on GPUs. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 375–384.

Xiaosheng Li, Xiaowei He, Xuehui Liu, Jian J. Zhang, Baoquan Liu, and Enhua Wu. 2016. Multiphase Interface Tracking with Fast Semi-Lagrangian Contouring. *IEEE Transactions on Visualization and Computer Graphics* 22, 8 (2016), 1973–1986. https://doi.org/10.1109/TVCG.2015.2476788

Fuchang Liu and Young J Kim. 2013. Exact and adaptive signed distance fieldscomputation for rigid and deformablemodels on gpus. *IEEE transactions on visualization and computer graphics* 20, 5 (2013), 714–725.

Frank Losasso, Ronald Fedkiw, and Stanley Osher. 2006. Spatially adaptive techniques for level set methods and incompressible flow. *Computers & Fluids* 35, 10 (2006), 995–1010. https://doi.org/10.1016/j.compfluid.2005.01.006

Frank Losasso, Frédéric Gibou, and Ron Fedkiw. 2004. Simulating water and smoke with an octree data structure. In *Acm siggraph 2004 papers*. 457–462.

Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. 2014. Unified particle physics for real-time applications. *ACM Trans. Graph.* 33, 4, Article 153 (jul 2014), 12 pages. https://doi.org/10.1145/2601097.2601152

Dinesh P Mehta and Sartaj Sahni. 2004. *Handbook of data structures and applications*. Chapman and Hall/CRC.

Nathan Morrical and John Edwards. 2017. Parallel quadtree construction on collections of objects. *Computers & Graphics* 66 (2017), 162–168. https://doi.org/10.1016/j.cag.2017.05.024 Shape Modeling International 2017.

Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

Stéphane Popinet. 2003. Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries. *J. Comput. Phys.* 190, 2 (2003), 572–600. https://doi.org/10.1016/S0021-9991(03)00298-5

Michael Schwarz and Hans-Peter Seidel. 2010. Fast parallel surface and solid voxelization on GPUs. *ACM Trans. Graph.* 29, 6, Article 179 (dec 2010), 10 pages. https://doi.org/10.1145/1882261.1866201

Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics (TOG)* 33, 6 (2014), 1–12.

F.S. Sousa, C.F. Lages, J.L. Ansoni, A. Castelo, and A. Simao. 2019. A finite difference method with meshless interpolation for incompressible flows in non-graded tree-based grids. *J. Comput. Phys.* 396 (2019), 848–866. https://doi.org/10.1016/j.jcp.2019.07.011

Hari Sundar, Rahul S. Sampath, and George Biros. 2008. Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel. *SIAM Journal on Scientific Computing* 30, 5 (2008), 2675–2708. https://doi.org/10.1137/070681727 arXiv:https://doi.org/10.1137/070681727

Jannis Teunissen and Ute Ebert. 2018. Afivo: A framework for quadtree/octree AMR with shared-memory parallelization and geometric multigrid methods. *Computer Physics Communications* 233 (2018), 156–166. https://doi.org/10.1016/j.cpc.2018.06.018

Tiankai Tu and David R. O'Hallaron. 2004. Balance Refinement of Massive Linear Octree Datasets. https://api.semanticscholar.org/CorpusID:978691

Lai Wang, Freddie Witherden, and Antony Jameson. 2024. An efficient GPU-based h-adaptation framework via linear trees for the flux reconstruction method. *J. Comput. Phys.* 502 (2024), 112823. https://doi.org/10.1016/j.jcp.2024.112823

Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. 2010. Data-parallel octrees for surface reconstruction. *IEEE transactions on visualization and computer graphics* 17, 5 (2010), 669–681.

Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.* 27, 5, Article 126 (dec 2008), 11 pages. https://doi.org/10.1145/1409060.1409079