Scalable adaptive PDE solvers in arbitrary domains

Kumar Saurabh§
Iowa State University
Ames, Iowa

Masado Ishii§
University of Utah
Salt Lake City, Utah

Milinda Fernando
University of Utah
Salt Lake City, Utah

Boshun Gao
Iowa State University
Ames, Iowa

Kendrick Tan
Iowa State University
Ames, Iowa

Ming-Chen Hsu
Iowa State University
Ames, Iowa

Adarsh Krishnamurthy
Iowa State University
Ames, Iowa

Hari Sundar
University of Utah
Salt Lake City, Utah

Baskar Ganapathysubramanian
Iowa State University
Ames, Iowa

Abstract

Efficiently and accurately simulating partial differential equations (PDEs) in and around arbitrarily defined geometries, especially with high levels of adaptivity, has significant implications for different application domains. A key bottleneck in the above process is the fast construction of a 'good' adaptively-refined mesh. In this work, we present an efficient novel octree-based adaptive discretization approach capable of carving out arbitrarily shaped void regions from the parent domain: an essential requirement for fluid simulations around complex objects. Carving out objects produces an *incomplete octree*. We develop efficient top-down and bottom-up traversal methods to perform finite element computations on *incomplete* octrees. We validate the framework by (a) showing appropriate convergence analysis and (b) computing the drag coefficient for flow past a sphere for a wide range of Reynolds numbers ($O(1-10^6)$) encompassing the *drag crisis* regime. Finally, we deploy the framework on a realistic geometry on a current project to evaluate COVID-19 transmission risk in classrooms.

1 Introduction

The discretization of the domain (i.e., mesh generation) is a critical aspect of numerically solving PDEs. The resolution and quality of the mesh are intimately related to the overall accuracy of PDE solvers. Even though mesh generation is a fundamental part of numerical approaches, creating quality meshes continues to be a significant bottleneck in the overall workflow. This bottleneck is exacerbated when considering adaptivity and parallel deployment and becomes exceptionally challenging in the presence of an arbitrarily shaped geometric object that has to be *carved* out from the computational domain. Such challenges are particularly common in simulating the flow over external objects. Streamlining this workflow is one of the components of the NASA 2030 computational fluid dynamics (CFD) milestone towards the goal of conducting overnight large-eddy simulations (LES) [55]: *"Mesh generation and adaptivity continue to be significant bottlenecks in the CFD workflow."*

Immersed boundary methods (IBMs) [42] are commonly used to simulate fluid flow around geometric objects *immersed* in a computational domain. A significant advantage of IBM approaches arises from performing the complete simulation on structured grids [42, 47], thus avoiding any requirement of the grid conforming to the immersed geometric object (Fig. 1a). Naively immersing the object can lead to large void regions. These regions do not participate in the solution but require the associated matrix and vector storage as they form a part of the mesh data structure. This problem is exacerbated in the presence of multiple objects. This paper presents a strategy wherein the void regions are first *carved* out from the main computational domain (Fig. 1c) —in the vein of the finite cell approach [60] with the object then immersed in the carved domain. This approach reduces the number of degrees of freedom and, hence, the memory footprint associated with the void regions.

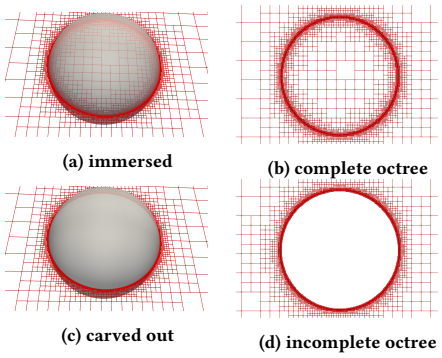


Fig. 1. Difference between the adaptive mesh for *immersed* and *carved* out for the sphere case. In immersed case, we retain the full octree and this gives to a significantly large number of elements and nodes compared to the carved out case. It must be noted the elements that are inside the object do not contribute to the accuracy of the solution. Eventually Dirichlet Boundary condition are imposed on all the In nodes.

Tree-based grid generation (quadtrees in 2D and octrees in 3D) is common in computational sciences [6, 11, 17, 24, 28, 30, 48, 56, 58] largely due to its simplicity and parallel scalability. The ability to efficiently refine (and coarsen) regions of interest using tree-based data structures have made it possible to deploy them on large-scale multi-physics simulations [2, 3, 14, 24, 32, 33, 37, 37, 49, 53]. Existing algorithms for tree-based grid generation are mainly focused on axis-aligned hierarchical splitting on isotropic domains (i.e., spheres, squares, and cubes). Such approaches cannot easily support anisotropic domains (for example, an elongated channel) or

§these authors contributed equally.

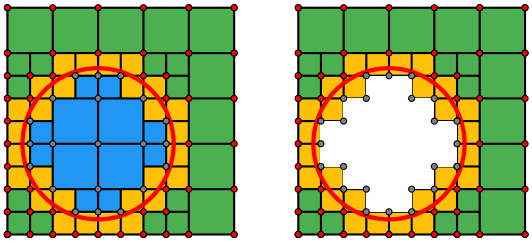
body-conforming mesh generation for a carved-out object. Standard workarounds include stretching or warping (using a coordinate transformation) of the computational domain, transforming a cubic domain into a rectangular channel. Unfortunately, these asymmetric transformations come at the cost of degradation in the overall quality of the domain discretization leading to large condition numbers in the resultant matrix (see Sec. 4.2).

Our contributions in this paper are as follows: (a) we develop an efficient tree-based adaptive mesh generation framework that relaxes the requirement of the mesh to conform to isotropic domains; (b) we compare the current approach with the state-of-the-art *immersed* method strategies [31, 59, 62, 65]; (c) we show the parallel scalability of our framework on the Frontera supercomputer up to 16K cores; (d) we deploy the framework in conjunction with a well established FEM formulation: variational multiscale (VMS) method [12] to model non-trivial applications of simulating the flow fields in classrooms to understand the risk of transmission of Coronavirus. The fast generation of quality meshes is pivotal to this application. Here, we present an octree-based mesh generation tool that provides an alternative to using two-tier meshes (HHG [13], p4est [17]) and is not dependent on having top-level hexahedral meshes–that can be hard to generate. In contrast, our approach works with any arbitrary user-supplied function that returns In or Out (of the object) for any queried point.

2 Related Work

There have been significant algorithmic advances for the fast generation of octrees on modern supercomputers. For instance, octrees have been used for voxelization of 3D objects to accelerate ray-tracing [64], signed distance calculation [67], compression [50], and fast rendering [34]. Building upon these successes, octrees have become one of the more common mesh generation tools for large-scale PDE simulations, with scalable and adaptive capabilities [21, 24, 29, 32, 49, 58]. But most work related to solving PDEs has been focused on the generation of *complete* octrees in the context of PDE solvers [18, 23, 24, 30, 38, 51, 56, 57, 63]. In a complete octree, every non-leaf subtree has all $2^d = 8$ children, and thus, the union of all leaf octants is a filled cube without holes. This makes simulating non-cuboid domains non-trivial, with most approaches either relying on stretching (coordinate transforms) or using a much larger bounding box. Secondly, complex objects have to be *immersed* into the octree mesh [21, 29, 53], rather than being *carved-out*. Naively immersing the object in the octree can leave many elements that fall into the void regions. This problem is further exacerbated in the presence of multiple objects. The elements that fall into the void regions are not solved during the simulations but have an associated memory footprint. The carving of an object leads to the construction of an incomplete octree. An octree is incomplete if there are non-leaf subtrees with one or more missing children. Finally, there has been limited work in developing octree-based mesh generation to efficiently solve PDEs over complex geometries. As stated earlier, carving out affords multiple advantages (explored in this paper) at the cost of a voxelated boundary of the object. We circumvent the voxelated boundary issue via an immersed boundary (IBM) formulation on this carved-out octree.

An alternate approach has been to use two-tier meshes [13, 17] that rely on having a top-level unstructured hexahedral mesh that conforms to the complex geometry and can independently refine (uniformly or adaptively) each hexahedral element of the top-level mesh. While this approach works well for simple shapes, like spheres [16], hex-meshing is non-trivial for more complex geometries [61]. An alternative is then to use affine transforms within each top-level element but limits the ability to have isoparametrically refined elements. In contrast, our approach can take an arbitrary function to carve out the domain and is capable of on-the-fly refinement and coarsening that matches the arbitrary function within the refinement tolerance.



(a) Complete quadtree **(b) Incomplete quadtree**

Fig. 2. A disk (enclosed region within the red circle) immersed in a *complete* (Fig. 2a) or *incomplete* (Fig. 2b) quadtree mesh. Every leaf in the tree represents an element occupying a region of space, which is either completely *inside* (■) the body; completely *outside* (■) the body; or *intercepted* (■) by the boundary (solid red circle). A complete tree (Fig. 2a) requires all 2^d children of each non-leaf subtree to be present, and thus gaps are not allowed in the middle of the mesh. However, useful information is only contributed by elements outside (■) or intercepted (■) by the body.

In this work, we address some of the key algorithmic challenges that are important to consider for carrying out efficient carving-out within the octree framework, yet missing from the existing literature:

- Careful mathematical abstraction that ensures the correctness of the generated mesh for any given arbitrary shape. Additionally, within the correctly carved out region, it is critical to correctly mark the boundary elements and nodes to solve PDEs correctly. (see Sec. 3.1)
- Handling of hanging nodes is critical during the carving out operations. Specifically, no hanging nodes should be present at the boundaries. If so, the parent of these nodes can lie in the inactive region, which is discarded during tree pruning. This would result in an incorrect PDE solution. (see Sec. 3.4)
- It is essential that the partitioning algorithm only looks at the active region of the octree. This ensures that FEM computations are evenly distributed and hence load-balanced. The data structure used in previous literature [66] first builds a complete octree distributed among processors before canceling out the inactive regions. This leads to the generation of complete trees with a substantial fraction of the trees in inactive regions. (see Sec. 3.2)
- Efficient pruning of trees at coarser levels is essential. Earlier approaches first generate the complete octrees before pruning. This can lead to substantial overheads for non-cube geometries, such as an elongated channel. (see Sec. 3.2)

3 Methodology

This section describes the methodology used to generate the tree-based grids (i.e., quadtrees in 2D, octrees in 3D) for arbitrary geometric domains. The key idea is to carve out regions from a d-dimensional cube that is *inside* the immersed geometric object

to generate the PDE solution domain (see Fig. 2). In this paper, we refer to the aforementioned domain that is left after carving out as the *subdomain*. As noted above, the subdomain may be a subrectangle of a regular box, or it may have carved regions excluded from an d-dimensional cubic domain. The algorithms presented here are dimension agnostic, but for simplicity, we mainly focus on 3D-based grids (octree) unless specified otherwise.

Previous work [17, 23, 30, 46, 56] have demonstrated efficient methods to construct 2:1-balanced complete octrees and additional data structures to perform efficient numerical computations at a large scale. These methods order the octants of the octree using a space-filling curve (SFC) (such as the Hilbert or Morton curve) to achieve better memory accesses locality and improved distributed-memory domain decomposition. This paper presents parallel algorithms to extend numerical computations on incomplete octrees.

3.1 Specification of the Subdomain

We describe an abstraction of the application-dependent arbitrary subdomains. The subsequent tree-based algorithms depend on a user-defined function to decide whether a given point or region in space should be retained or discarded ("carved"). In addition, the abstraction encodes enough detail for the octree algorithms to support efficient pruning during tree traversals.

Let $\Omega = C \cup C'$ be a cube comprised of two disjoint subsets: a closed *carved* set $C \subset \Omega$, and its open complement: the *retained* set $C' \equiv \Omega \setminus C$. Enforcing C as a closed set means that it contains the boundary, $\partial C \subset C$. (Referring back to Fig. 2, in 2D, C would be the disk, including ∂C, the red circle.)

Suppose that Ω is hierarchically partitioned according to an octree, $\mathcal{T}$, which captures ∂C well enough under some metric. Any octant e of $\mathcal{T}$ belongs to one of the following categories, depending on the closure of the region it bounds, $\bar{e}$:

(1) "carved," if $\bar{e} \subset C$.
(2) "retained," if $\bar{e} \not\subset C$.
   (a) "intercepted," if "retained" and $\bar{e} \cap C \neq \emptyset$.
   (b) "non-intercepted," if $\bar{e} \subset C'$

The retained octants form an incomplete octree; that is, we define $\mathcal{T}_I \equiv \mathcal{T} \setminus \{\text{carved leafs}\}$. The intercepted and non-intercepted sets specify the subdomain-boundary octants and the subdomain-internal octants, respectively.

3.1.1 Features of the Abstraction: Defining the subdomain abstraction in this way ensures that the octree pruning problem is well-defined. Notice the following:

- An application can specify a subdomain through a function $F(\bar{e})$, where $\bar{e}$ is any filled-in cube of zero or positive side length, such that

$$F(\bar{e}) := \begin{cases} \text{label("carved")} & \text{if } \bar{e} \subset \text{C} \\ \text{label("retain-internal")} & \text{if } \bar{e} \subset \text{C'} \\ \text{label("retain-boundary")} & \text{otherwise} \end{cases}$$

  - The function $F(\bar{e})$ applies to both octants and nodal points.
  - Points can not be classified "intercepted," as all points are contained in the union of C and C'.
  - The implementation of F must take care with nontrivial intersections between ∂C and an element. Even if all vertices lie in C, it is possible for the element to be intercepted, and in such a case, it should be labeled as "retain-boundary." In an application, the intersection test may be as simple or complex as needed by the geometry being captured.
- C is assumed closed; hence its complement, C', is open.
  - This convention permits the robust classification of a boundary element that sits flush with ∂C. The element is labeled "retain-boundary," while the boundary nodes are labeled "carved."
- This abstraction ensures the correctness of the generated mesh, for any arbitrary geometry, along with the correct tagging of boundary elements and nodes which is of utmost important for solving PDEs correctly.
- It is not necessary to generate a complete octree before filtering out the carved octants. If an octant is carved, all its children are carved. If an octant is non-intercepted, so are all its children. The next section describes how to construct incomplete octrees by proactively pruning subtrees during construction.

3.2 Octree Construction

The previous literatures have shown efficient octree algorithms to sort, construct, and traverse octrees in SFC order [23, 30, 56]. Most of the past literature has been limited to the construction of a complete octree in an isotropic domain [18, 23, 24, 30, 38, 51, 56, 57, 63]. This work builds upon them to efficiently construct the tree in presence of void regions. Algorithms 1 and 2 forms the central crux of the work, where an efficient approach for octree construction in the presence of void regions is introduced. Algorithm 3 describes the partitioning algorithm based on DistTreeSort to partition the trees.

The octree is constructed recursively in a top-down fashion (see Algorithms 1 and 2), with child subtrees being traversed in an order determined by a regional segment of the SFC. The top of the tree represents the entire isotropic domain, while subtrees represent cubical subregions. A given subtree is immediately pruned if it is classified as a carved region; otherwise, it is constructed. Constructing a subtree entails either appending the subtree as a leaf or refining it based on a refinement criterion. In Algorithm 1, the refinement criterion has a depth in the tree coarser than a target depth, whereas in Algorithm 2, the criterion has a depth that is coarser than a subset of seed octants. Other criteria are possible, e.g., intercepting the subdomain boundary or containing more than a maximal number of points from an initial point cloud distribution. If a subtree is to be refined, it is split into its eight child subtrees (in 3D). The children are permuted into the regional SFC ordering and constructed recursively.

Algorithm 3 uses the octree partitioning method based on DistTreeSort [23, 30] to distribute octrees in parallel. DistTreeSort uses TreeSort based comparison-free search algorithm for octree construction. Instead of performing comparison-based binary

Algorithm 1 ConstructUniform

Require: Region S, SFC oracle I, final level L, function $F()$.
Ensure: Set T of level-L leafs covering subdomain, sorted by SFC.

```
1: if F(S) ≠ Carved then                                   ▷ Else prune
2:     if level of S ≥ L then
3:         T.push(S)                                       ▷ Leaf.
4:     else
5:         for c_sfc ← 1 to 2^dim do                       ▷ Regional SFC order
6:             c_morton ← I.sfc2Morton(c_sfc)
7:             ConstructUniform(S.child(c_morton), I.child(c_sfc))
```

Algorithm 2 ConstructConstrained

Require: Region S, SFC oracle I, seed octants B, function $F()$.
Ensure: Set T of leafs, no coarser than B, covering the subdomain, sorted by SFC.

```
if F(S) ≠ Carved then                                   ▷ Else prune
    L ← finest level in B
    if |B| = 0 or level of S ≥ L then
        T.push(S)                                       ▷ Leaf.
    else
                                                        ▷ Bucket seeds to SFC-sorted children of S.
        l ← level(S) +1
        counts[2^dim] ← 0
        for b ∈ B do
            counts[child_num(b, l)]++
        counts[] ← permute(counts, I)
        offsets[] ← scan(counts)
                                                        ▷ Construct child subtrees in SFC order.
        for c_sfc ← 1 to 2^dim do
            c_morton ← I.sfc2Morton(c_sfc)
            S_c ← S.child(c_morton)
            I_c ← I.child(c_sfc)
            B_c ← B.slice(offsets[c_sfc], offsets[c_sfc + 1])
            ConstructConstrained(S_c I_c, B_c)
```

Algorithm 3 DistributedConstructConstrained

Require: Distributed set of seed octants B, function $F()$.
Ensure: Distributed set T of leafs, no coarser than B, covering the subdomain, sorted by SFC.

```
DistTreeSort(B, load_tol)                               ▷ [30]
T_tmp ← ConstructConstrained(TreeRoot, SFC_Root, B)
DistTreeSort(T_tmp, load_tol)
T_local ← DistributedUniqueLeafs(T_tmp)
return T_local
```

searches, TreeSort performs MSD radix sort, except that the ordering of buckets are permuted at each level according to the specified SFC. By performing a fixed number of passes over the input data in a highly localized manner, TreeSort avoids cache misses and random memory access leading to better memory performance. [23, 24] Since the constructed octrees from previous algorithms entail only the active regions of the isotropic domain, DistTreeSort distributes only these aforementioned active portion. This is the main difference from the past approaches, where the sorting algorithm looks at the complete tree. This step is pivotal to ensure load-balanced computation. Similar to Algorithm 2, a set of seed octants is used to control the output tree depth. In the distributed setting, the seed octants also inform the domain decomposition so that each rank will own approximately the same number of elements. Note that DistTreeSort accepts a tunable load-balance tolerance. A large tolerance will partition the tree at coarse levels. A small tolerance will balance the load more evenly at the expense of splitting coarse subtrees over multiple processes. Once the depth-constraining seed octants have been partitioned, each rank constructs a tree satisfying the local constraints. Then, overlaps between trees must be resolved. Duplicate octants are deleted. Finer octants are preferred to coarser overlapping octants in order to satisfy the depth constraints globally.

3.3 2:1 Balancing

In a *2:1-balanced* octree (Fig. 3), a pair of octants sharing any parts of their boundaries may differ in scale by at most a factor of 2:1. In other words, they may differ by at most one level in the tree. Numerical computations on the octree grid are simpler in terms of the neighborhood data structures if the octree obeys the 2:1-balancing constraint .

Algorithm 4 DistributedConstruct2to1Balanced

Require: Distributed set of seed octants B, function $F()$.
Ensure: Distributed set T of leafs, no coarser than B, covering the subdomain, obeying 2:1-balance constraint, sorted by SFC.

```
T_1 ← DistributedConstructConstrained(B, F)
T_2 ← BottomUpConstrainNeighbors(T_1)                   ▷ F not applied
T_3 ← DistributedConstructConstrained(T_2, F)
return T_3
```

Algorithm 5 BottomUpConstrainNeighbors

Require: Unbalanced leafs T_1.
Ensure: Balanced seeds T_2.

```
T_aux[] ← stratify T_1 by levels, from finest to coarsest
for level l from finest to coarsest do
    for t ∈ T_aux[l] do
        for n ∈ MakeNeighbors(MakeParent(t)) do
            T_aux[l − 1].add_unique(n)
T_2 ← concatenate(T_aux)
return T_2
```

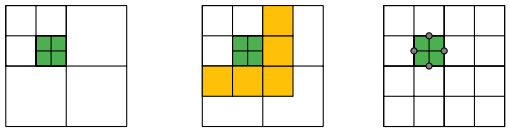


Fig. 3. Left most figure shows an octree which violates the 2:1 balanced constraint, where the octants that cause the violation is showed in (■). In the middle figure auxiliary balanced octants are showed in (■), in other words these are the octants needed to remove the balance constraint violation in (■). Right most figure shows the constructed octree with auxiliary balanced octants which satisfies the 2:1 balance constraint. The nodes marked by gray circles in the final 2:1 balanced mesh are hanging nodes.

We take a bottom-up approach to transform a given linear octree into a 2:1-balanced octree, based on the local block balancing method similar to the one by Sundar et al. [56]. In our method (Algorithms 4 and 5), the input octree comprises an initial set of seed octants. The seed set is iteratively updated from the finest to the coarsest level. For each seed octant, the neighbors of its parent octant are added to the next-coarser level of seeds. Duplicate octants are removed from the next level before proceeding. Finally, after all the levels have been processed, a new linear octree is constructed such that each seed octant becomes either a leaf or an ancestor subtree in the output octree. Thus the final seed set controls the resolution of the new octree, ensuring the result is 2:1-balanced. It is important not to preemptively discard the carved octants, which are generated as neighbors of parents of seed octants. Otherwise, two leaf octants of 4:1 or greater ratio could meet in a carved region.

3.4 Embedding Nodal Information

Each leaf octant in a linear 2:1-balanced octree represents an element in the FEM adaptive grid. For a given p-refinement, there are $(p+1)^3$ nodes per element (in 3D). Nodal points on the boundary of an element will be shared with same-level neighboring elements. Nodes incident on a coarser-level neighbor is considered *hanging nodes* (Fig. 3). The value of a hanging node is dependent on the values of the nodes on the coarser face or edge. Therefore, the set of independent degrees of freedom (DOFs) on the FEM grid (underpinning a grid vector) is defined by enumerating the unique, non-hanging nodes.

First, we loop over all elements and generate the node coordinates with a spacing of $\left(\text{length}_{\text{element}}\right)/p$ in each axis. (Nodes labeled as "carved" are marked as subdomain boundary nodes.) The

set of unique nodes is found by executing TreeSort on the nodal coordinates and removing duplicates.

An extra step is required to detect and discard hanging nodes. In an isotropic domain, a hanging node has fewer instances than the number expected for an ordinary node as a function of the coordinate and grid level. With user-specified geometry, however, the expected number of instances is nontrivial to compute. Our solution is to explicitly "cancel" possible hanging nodes using temporary *cancellation nodes*. The cancellation nodes are generated on the edges and faces of elements in between the ordinary nodes, anticipating the coordinates of hanging nodes from hypothetical finer neighbors. After sorting, every coordinate is occupied by a mix of ordinary and cancellation nodes. If a cancellation node is present, then the coordinate is incident on a coarser edge or face, and thus the node is hanging; the node is discarded. Otherwise, no cancellation node is present, and the coordinate is enumerated as an ordinary node. Thus we enumerate exactly the nodes which define a grid vector. Note that ensuring the absence of hanging nodes at the carved boundary is essential for accurate PDE solutions.

3.5 Matrix-free, Traversal-based matvec

We implement a traversal-based matrix-vector multiplication to perform matrix-free computations, which extends the methods by Ishii et al. [30] to incomplete trees.

Matrix free: The global matrix is defined as a summation of local elemental matrices, where the summation is due to common nodal points being shared by neighboring elements. We are able to apply the global operator to a grid vector without explicitly assembling the global matrix. Instead, we perform a series of elemental matrix-vector multiplications, and use the octree structure to compose the results.

Traversal-based: (Fig. 4) The elemental matrix couples elemental nodes in a global input grid vector with equivalent elemental nodes in a global output grid vector. Within a grid vector, the nodes pertaining to a particular element are generally not stored contiguously. If one were to read and write to the elemental nodes using an element-to-node map, the memory accesses would require indirection: $v_{glob}[map[e * npe + i]] += v_{loc}$. Not only do element-to-node maps cause indirect memory accesses; the maps become complicated to build if the octree is incomplete due to complex geometry. We take an alternative approach that obviates the need for element-to-node maps. Instead, through top-down and bottom-up traversals of the octree, we ensure that elemental nodes are stored contiguously in a leaf, and there apply the elemental matrix.

The idea of the top-down phase is to selectively copy nodes from coarser to finer levels until the leaf level, wherein the selected nodes are exactly the elemental nodes. Starting at the root of the tree, we have all the nodes in the grid vector. We create buckets for all child subtrees. Looping through the nodes, a node is copied into a bucket if the node is incident on the child subtree corresponding to that bucket. A node that is incident on multiple child subtrees will be duplicated. By recursing on each child subtree and its corresponding bucket of incident nodes, we eventually reach the leaf level.

Once the traversal reaches a leaf octant, the elemental nodes have been copied into a contiguous array. The elemental matrix-vector product is computed directly, without the use of an element-to-node map. The result is stored in a contiguous output buffer the same size as the local elemental input vector.

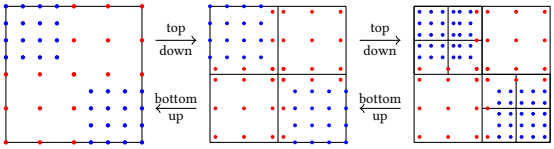


Fig. 4. Illustration of top-down & bottom-up tree traversals for a $2D$ tree with quadratic element order. The leftmost figure depicts the unique shared nodes (nodes are color-coded based on level), as we perform top-down traversal nodes shared across children of the parent get duplicated for each bucket recursively, once leaf node is reached it might be missing elemental local nodes, which can be interpolated from immediate parent (see the rightmost figure). After elemental local node computations, bottom-up traversal performed while merging the nodes duplicated in the top-down traversal.

After all child subtrees have been traversed, the bottom-up phase returns results from a finer to a coarser level. The parent subtree nodes are once again bucketed to child subtrees, but instead of the parent values being copied, the values of nodes from each child are accumulated into a parent output array. That is, for any node that is incident on multiple child subtrees, the values from all node instances are summed to a single value. The global matrix-vector product is completed after the bottom-up phase executes at the root of the octree.

Distributed memory is supported by two slight augmentations. Firstly, the top-down and bottom-up traversals operate on ghosted vectors. Therefore ghost exchanges are required before and after each local traversal. Secondly, the traversals are restricted to subtrees containing the owned octants. The list of owned octants is bucketed top-down, in conjunction with the bucketing of nodal points. A child subtree is traversed recursively only if one or more owned octants are bucketed to it. Note that because the traversal path is restricted by a list of existing octants, the traversal-based matvec gracefully handles incomplete octrees without special treatment.

Remark. *The traversal based matvec is designed to expose memory locality suited for deep memory hierarchies inherent in modern day clusters and accelerators like GPUs. In this work, we focus on distributed memory parallelism; the implementation on accelerators is deferred to future work.*

3.6 Traversal-based Matrix Assembly

In the previous section, we described matvec procedure that employs a tree traversal, requiring neither element-to-node maps nor global matrix assembly. In this section, we describe the matrix assembly procedure for computing the global sparse matrix. The efficient computation of matrix assembly becomes particularly important for the problems whose convergence heavily depends on the preconditioners.

To implement assembly, we have leveraged PETSc interface [8, 10], which only requires a sequence of entries ($\mathrm{id}_{\mathrm{row}}$, $\mathrm{id}_{\mathrm{col}}$, val), and can be configured to add entries with duplicate indices [9]. Note that any other distributed sparse-matrix library can be supported in a similar fashion.

The remaining task is to associate the correct global node indices with the rows and columns of every elemental matrix. We use an octree traversal to accomplish this task. Similar to the traversal-based matvec, nodes are selectively copied from coarser to finer

levels, recursively, until reaching the leaf, wherein the elemental nodes are contiguous. Note that integer node ids are copied instead of floating-point values from a grid vector. At the leaf, an entry of the matrix is emitted for every row and column of the elemental matrix, using the global row and column indices instead of the elemental ones. No bottom-up phase is required for assembly, as PETSc handles the merging of multi-instanced entries.

4 Results

Computing Environment: We performed experiments, including simulations and scaling studies, on the Cascade Lake Compute Nodes of the Frontera system. (Refer Sec. A.3 for compute configuration.)

Software and Libraries: We used PETSc [8] as the numerical algebra solver for solving system of equations. The DistTreeSort and TreeSort implementation is taken from Dendro [25]. All comparison with the immersed (IBM) method is performed using the open-source code [52] based on Saurabh et al. [53]. Additionally, Matlab [41] is used for analyzing the condition number of matrices, and trimesh [20] is used to compute the signed distance. The roofline plot was generated by using Intel Advisor.

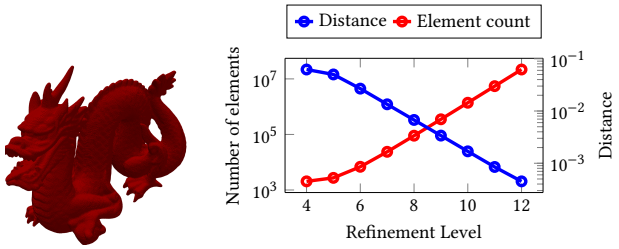
4.1 Approximation of Voxelized Geometry

The *carving-out* approach leads to a voxelized geometry, which is an approximation of the actual geometry. In this section, we compare how closely the voxelized geometry mimics the actual geometry by considering the example of the Stanford Dragon [36]. Fig. 5 compares the difference in the representation of actual boundary for the voxelized geometry by computing the signed distance[1]. Fig. 5a shows the voxel representation for the Stanford dragon. Fig. 5b compares the L_∞ error of computed signed distance between the boundary nodes of the voxelized geometry and the actual STL file. Similar to the previous case, we can see that with increase in the refinement, the voxelized geometry approaches the actual geometry.

4.2 Conditioning of Discrete Operators

As stated earlier, one approach to deploy traditional octrees on elongated channels is to stretch the mesh along the elongated channel [22, 40]. But this has a detrimental effect on condition number, which in turn will deteriorate the convergence of linear solvers. Table 1 compares the variation in the condition number [2] with the stretching of the elements for a Laplace operator in 2D. We can see that with the increase in the aspect ratio of the mesh, the condition number of the linear system increases. With the generation of incomplete octree, we can ensure the aspect ratio of each element in the mesh remains 1. Furthermore, since the error is dominated by the coarsest resolution, the incomplete octree permits decreasing the overall DOFs, at a given coarse resolution. This, in turn, decreases the condition number of the linear system.

[1] computed using `trimesh` library. A positive value denotes inside.
[2] evaluated with Matlab `condest` command



(a) Voxelized **(b) Error**

Fig. 5. Figure showing the voxelized geometry for the Stanford Dragon on octree mesh Fig. 5a. The error (Fig. 5b) is measured as the maximum of signed distance from boundary nodes of octree to the STL mesh. With increase in the refinement at the surface of geometry, the octree mesh coincides with the actual 3D mesh resulting in decrease in the signed distance error. Note the first order convergence in signed distance error with resolution.

Channel length	Complete octree		Incomplete octree	
	DOFs	Condition Number	DOFs	Condition Number
1	1089	402.6	1089	402.6
2	1089	466.7	561	155.6
4	1089	510.1	297	42.5
8	1089	512.0	165	13.3
16	1089	10580.5	99	5.0

Table 1: Comparison of condition number for the case with complete octree and incomplete octree. In the case of complete octree, each element of the mesh was stretched according to the channel aspect ratio (represented here by the length) to conform with the channel boundaries, whereas in the case of incomplete octree, the aspect ratio was fixed to be 1 and the elements outside the domain are removed.

4.3 Convergence Test for Discrete Operators

Here, we present the convergence analysis for the Poisson operator $-\Delta u = f$ over the domain Ω with $u = u_D$ on the domain boundary Γ. Inserting appropriate finite dimensional function spaces for trial and test function, the weak form of the Poisson operator can be written as: $(\nabla w^h, \nabla u^h)_\Omega = (w^h, f)_\Omega$

As mentioned previously, deploying incomplete octree based methods results in a voxelated geometry for a complicated geometrical shape. As discussed in Sec. 4.1, with the increase in the refinement of the element, the voxelated geometry approaches the true geometry. The rate of convergence of the distance follows only first order. Therefore, careful treatment is needed at the boundary elements to ensure an accurate order of convergence. In this context, several methods have been proposed in the literature [15, 35, 42]. In this work, we use the Shifted Boundary Method (SBM) [5, 39] to treat boundary elements.

The main idea behind SBM is to reformulate the original boundary value problem over a surrogate computational domain by modifying the original boundary conditions using Taylor series expansions. The weak form of Poisson operator after applying SBM treatment can be written as:

$$(\nabla w^h, \nabla u^h)_{\tilde{\Omega}} - (w^h, \nabla u^h \cdot \tilde{\mathbf{n}})_{\tilde{\Gamma}} - (\nabla w^h \cdot \tilde{\mathbf{n}}, u^h + \nabla u^h \cdot \mathbf{d} - u_D)_{\tilde{\Gamma}} + \frac{\alpha}{h}(w^h + \nabla w^h \cdot \mathbf{d}, u^h + \nabla u^h \cdot \mathbf{d} - u_D)_{\tilde{\Gamma}} = (w^h, f)_{\tilde{\Omega}}$$

where $\tilde{\Omega}$ is the voxelated domain, $\tilde{\Gamma}$ is the surface of the voxelated domain, $\tilde{n}$ is the unit normal of the voxelated surface, α is the penalty term, h is the element length, and $\mathbf{d}$ is the distance vector from the boundary surface of voxelated domain $\tilde{\Gamma}$ to the true surface Γ. The main idea of the method is to shift the boundary condition from Γ to $\tilde{\Gamma}$ by using second-order accurate Taylor series expansion. We omit the details here and refer to [5, 39] for detailed analysis.

To perform the convergence study, we consider Poisson problem on a two-dimensional disk of radius $R = 0.5$, centered at $(x_0 = 0.5, y_0 = 0.5)$ and $f = 1$. An exact solution exists and can be written as: $u(r) = 0.25(R^2 - r^2)$, where $r = \sqrt{(x - x_0)^2 + (y - y_0)^2}$. Fig. 6 shows the convergence behaviour for the linear basis function. If

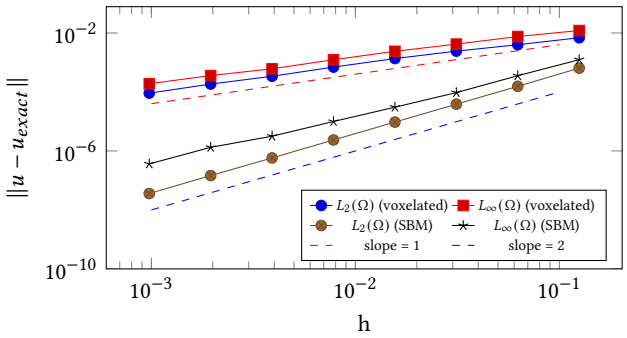


Fig. 6. *Convergence plot*: Figure showing the convergence behaviour for the Poisson operator on a two-dimensional circular disk.

we naively apply the boundary condition at the boundary nodes of the voxelated geometry, we only get a first-order convergence in both L_2 and L_∞ norm. This is because the right boundary condition is applied at the wrong place, which is shifted by a distance **d** from the true boundary. As seen from the signed distance plot (Fig. 5b), the voxelated geometry boundary approaches the true geometry according to the first order, and so is the convergence for the discrete Poisson operator. With the SBM method, we recover back the theoretical second-order convergence in both L_2 and L_∞ norm for the linear basis function.

4.4 Comparison with *Immersed* Case

Here, we present the comparison of the carved out approach with the immersed approach in terms of the number of DOF and the total number of elements. We note that this analysis is equation agnostic. In order to compare the overall mesh element size and DOF, we set the background mesh to a constant refinement level and refined it near the object. Tab. 2 compares the fraction of elements and DOF required for immersed and carved out approach. In the carved out case, all the elements and nodes that are inside the domain are discarded during the tree construction as mentioned in Sec. 3.2, whereas for the immersed case, we retain the complete octree mesh. 2:1 balancing of octrees leads to the ripple effect, because of which there is a significant number of elements that are inside the domain (Fig. 1). The nodes and elements that are marked In (i.e. inside the object, sphere/dragon in this case) do not contribute towards the accuracy of the solution. These are not solved for in the system of equations, and eventually, a Dirichlet boundary condition is applied to it, but they had the associated cost during tree traversal and memory footprint for matrix and vector storage. Overall, we see about an increase of 80–90% in element size and a 33–40% in the DOF count if we immerse an object. The excess DOF count is significantly smaller than the element count because of the fact that we are performing continuous Galerkin (CG) computations and several elements share a common DOF. Additionally, we must recall the fact that in CG computations, hanging nodes do not contribute to the additional degrees of freedom. However, if we were to perform discontinuous Galerkin (DG) computation, each element would have its own unique node id and associated DOF. In such computations, the excess DOF count would scale as excess element count. The actual fraction of DOF and element that is reduced as a result of carving out depends upon the surface area and volume of the object that is being carved out. A large surface area of the object would result in more elements at the finest resolution near the boundaries of the object. In contrast, a larger volume would result in more elements being discarded out from the interior of the object. Constructing an incomplete octree by cutting the elements inside the object results in processing fewer elements during a solve.

		Refine Level			
		11	12	13	14
Sphere	f_{elem}	1.75	1.79	1.81	1.82
	f_{DOF}	1.30	1.31	1.32	1.33
Stanford Dragon	f_{elem}	1.84	1.87	1.90	1.92
	f_{DOF}	1.36	1.39	1.41	1.43

Table 2: Comparison of the ratio of number of elements (f_{elem}) and degrees of freedom (f_{DOF}) with and without (*immersed*) carving out the sphere and the Stanford dragon from the domain. The base refinement was set to 4 and the refinement level near the object was varied from 11 to 14.

4.5 Scaling

We evaluated the strong and weak scaling performance of our traversal-based MATVEC using linear and quadratic elements on the Frontera supercomputer for two different cases: a) an elongated channel of dimension $16 \times 1 \times 1$, b) a spherical region carved out from the cube. We individually timed the execution of major components of MATVEC, namely top-down and bottom-up traversal, leaf MATVEC to compute the elemental operators, malloc and communication cost. It must be noted that for any PDE solver, MATVEC is the basic building block and determines the overall parallel performance and scalability. We highlight some important points regarding the experimental setup for performing the scaling studies and the interpretability of the scaling results:

- *Strong Scaling:* For each of the strong scaling cases, we generated a fixed mesh defined by different refinement levels in different regions of interest. When comparing the mesh with linear and quadratic elements, the total number and distribution of elements in a given mesh is the same not only at a global level but also locally at each processor level. Note that the partitioning algorithm DistTreeSort is agnostic to the underlying element order and distributes the element at the octant level before the nodal information is encoded [3].
- However, the total number of DOF and problem size grows as $O((p+1)^d)$ for an arbitrary order p and dimension d [4]. Hence, the mesh with linear and quadratic elements have different computation and communication complexity. For instance, both linear and quadratic mesh for channel strong scaling study have 13.5M elements. But the linear element mesh has 13.7M DOFs, whereas the quadratic has 109.1M DOFs.
- *Weak Scaling:* For the weak scaling runs, with an increase in the number of processors, we increase the refinement level in the regions of interest in such a way that the average number of elements per processor remains the same. Similar to the strong scaling, for a given number of processors, the total number and distribution of elements are the same for both linear and quadratic

[3] More formally, consider a mesh M with N global elements distributed over p processor. If we globally number the elements of mesh from $0 \cdots N-1$, then if a processor $k, k \leq p$, receives $m_i \cdots m_k$ (m_i's being the global element number) elements for linear, $0 \leq m_i \leq m_k \leq N-1$, then the processor k for quadratic mesh will also receive the same sequence of elements $m_i \cdots m_k$.

[4] Every element has $O((p+1)^d)$ nodes (Refer Sec. 3.4).

basis functions both globally and locally. Hence, the quadratic mesh has a greater number of DOF compared to the linear one.
- In all the scaling figures (Fig. 7 –Fig. 10), for a given number of processors, the left bar corresponds to the MATVEC execution profile for the linear elements, and the right bar corresponds to the execution profile of the quadratic elements. The total execution time for the linear elements is shown by solid blue lines and red dashed lines for the quadratic.

4.5.1 Scaling results for the channel: The incomplete octrees representing $16\times1\times1$ elongated channel, with greater refinement on the boundary and minimal refinement on the interior, are generated to carry out the scaling studies. This is representative of the common cases that arise in the boundary-dominated physical phenomena. Each scaling run was repeated for linear and quadratic hexahedral grids.

For the strong scaling runs, we generated octree mesh with 13M elements for linear and quadratic basis functions. Both linear and quadratic mesh is similar at the elemental level. Fig. 7 shows the strong scaling behavior in terms of parallel cost (Run time × number of cores) for both the linear and quadratic basis functions. A constant line would mean ideal strong scaling efficiency. For the linear mesh, MATVEC execution time decreased from 2.87 s on 224 processors to 0.027 s on 28K processors, resulting in 81% parallel efficiency for 128 fold increase in processor count. Similarly, for the quadratic mesh, we see a reduction in MATVEC execution time from 13.5 s on 224 processors to 0.1 s on 28K processors, resulting in 90% parallel efficiency. The overall theoretical complexity for MATVEC for a given element of order p has been shown to scale as $O(d(p+1)^{d+1})$. We see a factor of 4.2 × increase in MATVEC execution time for quadratic element ($p = 2$) over linear ($p = 1$), which is within the theoretical bounds.

For the weak scaling runs, we created grids with a fixed grain size of about 35K elements per core and timed MATVEC execution time. The coarsest mesh consists of 981K elements on 28 processors with 1.02M DOFs for linear and 8.01M DOFs for quadratic element, whereas the finest mesh consists of 502M elements on a 14K processors with 505M DOFs for linear and 4 billion DOFs for quadratic elements. Fig. 8 plots the mean execution of the MATVEC averaged over 100 iterations as a function of the number of cores. A constant execution time would imply ideal weak scaling efficiency. We observed a slowly growing weak-scaled execution time. Overall the time increased from about 1.58 s on 28 cores to 1.9 s on 14 K cores for linear elements (82% weak scaling efficiency) and 7.04 s to 8.04 s for quadratic elements (86% weak scaling efficiency).

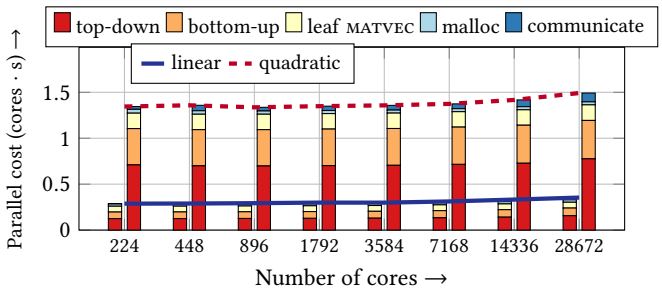


Fig. 7. *Strong scaling for channel case.* Parallel cost evaluated with the 3D Poisson MATVEC on Frontera supercomputer. Problem size was fixed at 13M elements (13.7M unknowns for linear and 109.1M unknowns for quadratic)

Case Type	Element Order	Strong scaling			Weak Scaling	
		Num elements	Num DOFs	Efficiency	Num elements/core	Efficiency
Channel (Sec. 4.5.1)	Linear	13.5 M	13.7 M	0.81	35K	0.82
	Quadratic	13.5 M	101.9 M	0.90	35K	0.86
Sphere (Sec. 4.5.2)	Linear	17.5 M	17.4 M	0.90	10K	0.74
	Quadratic	17.5 M	139.7 M	0.96	10K	0.83

Table 3: Summary of scaling efficiency for the channel and spherical carved out region.

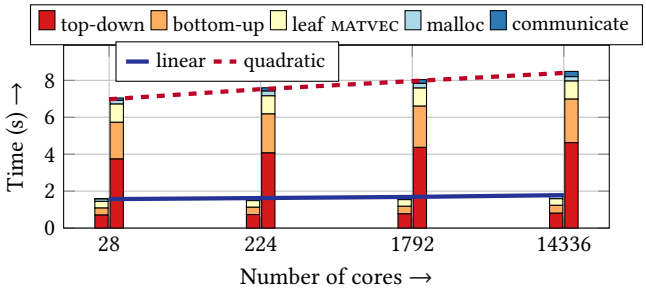


Fig. 8. *Weak scaling run time for channel case:* Execution time of 3D Poisson MATVEC on Frontera supercomputer, for a fixed grain size of about 35K elements per core.

4.5.2 Scaling results for a spherical carved out region: To study the scaling behavior for a complex carved out geometry, we carved out a spherical region from a cubical domain. A sphere of diameter $d = 1$ unit is carved out from a cubical domain of $10 \times 10 \times 10$. Overall, full mesh contains 5 levels of octree adaptivity with maximum refinement near the sphere. Such domain and mesh resolution are similar to the application problem used for validation of Navier–Stokes simulation.

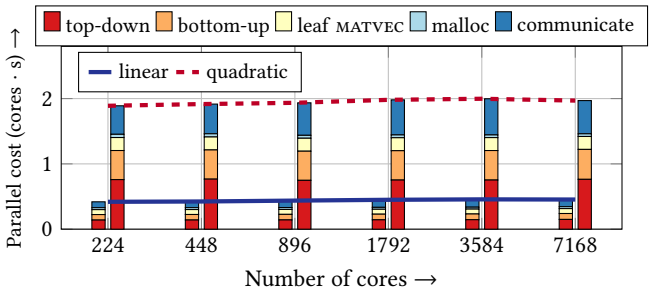


Fig. 9. *Strong scaling for sphere case:* Parallel cost evaluated on Frontera supercomputer. Problem size was fixed at 17.5 M elements(17.4M unknowns for linear and 139.7M unknowns for quadratic)

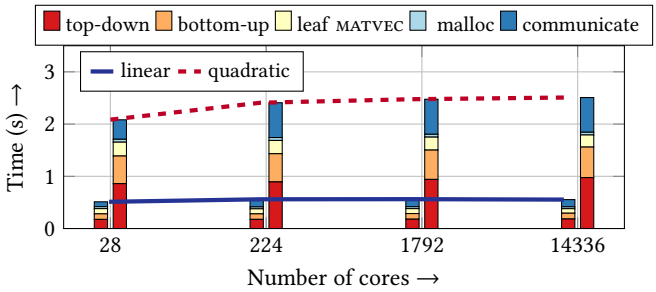


Fig. 10. *Weak scaling run time for sphere case:* Mean Execution time of 100 MATVEC on Frontera supercomputer, for a fixed grain size of about 10K elements per core.

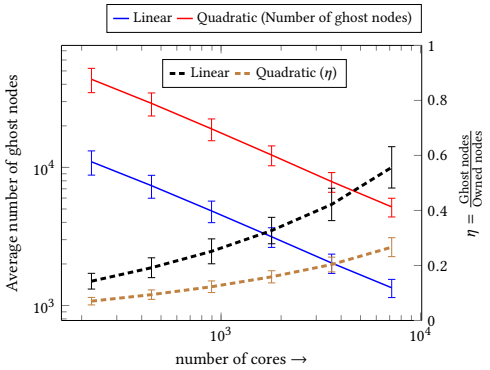


Fig. 11. Figure showing the mean and standard deviation of the distribution of ghost nodes (shown by solid lines) and ratio of ghost nodes by owned nodes per processor (shown by dashed lines)

We created grids of about 17.5M elements for the strong scaling, which correspond to 17.3M DOFs for the linear and 139.7M DOFs for the quadratic basis functions. Similar to the channel case, the mesh partition for both the linear and quadratic is similar at the elemental level but has a different number of DOFs. Fig. 9 shows the parallel efficiency averaged over 100 MATVEC iterations. Overall we observe a good overall parallel efficiency. In the case of linear elements, we observe a 29× reduction in MATVEC execution time for 32 fold increase in processor (90% strong scaling efficiency). In contrast, the quadratic element resulted in a 31× reduction in computation time (96% strong scaling efficiency).

For the weak scaling, we kept a constant grain size of around 10K elements per processor. The coarsest mesh consists of about 290K elements resulting in 280K DOFs for the linear basis function and 2.3 M for the quadratic. In contrast, the finest mesh consists of 138 M elements with about 138M DOFs for linear and 1.1 billion DOFs for quadratic basis function. Fig. 10 shows the overall weak scaling performance. MATVEC execution time grew from for 4.1 s on 28 processors to about 5.5 s on 14K processors for linear elements, resulting in a factor of about 1.34× increase for 512× in the number of processors (74% efficiency). In the case of quadratic elements, the execution time increased from 20 s on 28 processors to about 25 s on 14K processors, yielding about 83% weak scaling efficiency. Tab. 3 summarizes the scaling efficiency for both the channel and the sphere case.

Further, we analyzed the distribution of ghost nodes per processor for the above sphere case, which is indicative of bytes of data communicated. In our experiment, we kept a similar distribution of the elements across processors for both linear and quadratic basis function but has different degrees of freedom associated with them. The amount of data communicated across processors is a function of the total number of ghost elements that share partition boundaries. With the increase in the number of processors, the total number of ghost elements increases, but the average number of ghost elements decreases. For an arbitrary order element p, the number of nodes that share faces across the processor boundaries (and hence needs exchange of information) grows as $O((p+1)^{d-1})$. Since the partition is similar at the elemental level, the average number of ghost nodes that are needed for ghost exchange is higher for the quadratic compared to the linear elements. The solid lines in Fig. 11 show the comparison for linear and quadratic case.

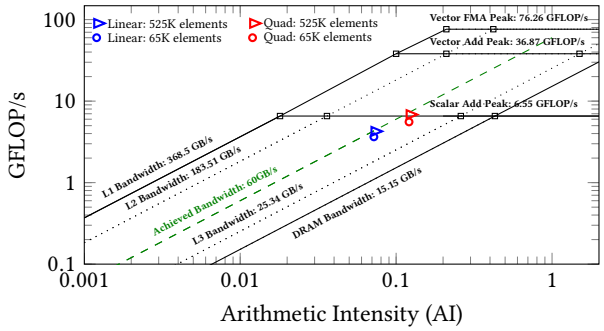


Fig. 12. Figure showing roofline plot for the Poisson MATVEC for linear and quadratic basis function for two different meshes on Frontera. The plot was generated using Intel Advisor. The green dashed line shows the achieved bandwidth from our code. All values reported in the plot corresponds to double precision floating point operations.

Additionally, we also analyzed the distribution of the ratio of ghost nodes to the number of owned nodes (denoted by η), which is indicative of the extent over which the communication can be overlapped with computation. Let N_L be the number of local nodes that are owned by processor (do not share processor boundaries with any other elements) and N_G be the number of ghost nodes, then:

$$\eta = \frac{N_G}{N_L} \propto \frac{(p+1)^{d-1}}{(p+1)^d} = \frac{1}{(p+1)}$$

From the above equation, we can see that this ratio grows inversely with respect to the degree of element. We observe similar behavior in our experiments, as shown by dashed lines in Fig. 11. This explains the better scaling efficiency for quadratic as compared to linear shown in Tab. 3. For a single processor run, $\eta = 0$. With an increase in the number of processors η increases, and in extreme limit of parallelization, when each processor has only one element, $\eta \to 1$. It is non-trivial to analyze the exact rate of increase in η for arbitrary shapes as a function of processor count and is beyond the scope of the current work.

4.5.3 Roofline: Fig. 12 shows the single core roofline plot for the elemental MATVEC computation of Poisson operator using linear and quadratic basis function on Frontera. Overall, we can see that the code is memory bound as is common for finite element codes. We observe higher arithmetic intensity [5] (AI) for quadratic (0.121) as compared to linear (0.072) elements. The amount of data needed for MATVEC computation grows as $O((p+1)^d)$ whereas the MATVEC computation complexity grows as $O(d(p+1)^{d+1})$. Therefore, AI tends to increase with polynomial order, which explains the observed behavior. We are able to achieve a performance of about 4 GFLOP/s using linear basis function and 7 GFLOP/s using quadratic basis functions for two different meshes, which corresponds to a bandwidth of approximately 60 GB/s as shown by the green lines. We note that we have not used any hand-coded explicit vectorization to ensure the portability of the code across various platforms and relied on compiler-directed vectorization. We would like to explore some future avenues from the code optimization point of view pertaining to more efficient cache blocking techniques and architecture-specific efficient vectorized implementation of tensor products.

[5] AI is measured as the amount of floating point operations performed per byte of data loaded into the memory.

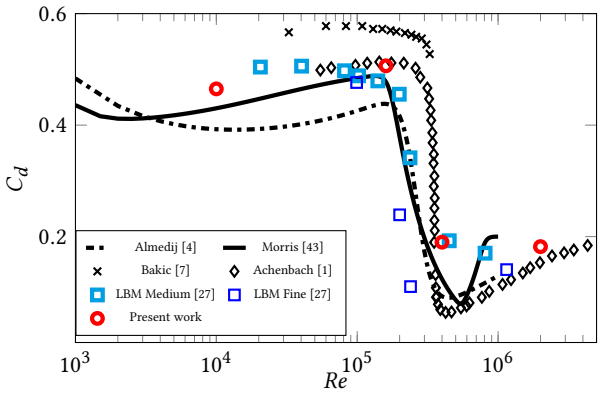


Fig. 13. Drag crisis: Variation in C_d close to the region of drag crisis. We see a good agreement with the experimental data and past numerical results.

Base Refinement	Boundary Refinement	Num Elements	Num Processors	Dendro (s)		Current Approach (s)	
				Mesh Creation	matvec	Mesh Creation	matvec
10	12	3,138,525	448	107.87	59.14	1.69	9.27
			896	55.41	38.03	1.00	4.53
			1792	38.21	25.47	0.87	2.34
10	14	49,096,209	448	280.88	447.05	21.51	142.89
			896	159.87	349.39	11.51	77.33
			1792	127.21	295.02	6.34	48.51
12	12	17,440,929	448	-	-	4.57	42.58
			896	-	-	2.29	19.65
			1792	-	-	1.75	10.76
12	14	63,398,613	448	-	-	33.56	182.23
			896	-	-	17.14	125.53
			1792	-	-	9.86	47.65

Table 4: Comparison of the time (in seconds) for mesh generation and Navier–Stokes matvec for the current approach with Dendro based octree framework. With level $\geq$ 12 of base refinement, Dendro framework gave memory error, and hence no time is reported.

4.6 Comparison with Existing Method

Here, we compare the performance of the proposed algorithm with the existing octree based framework, specifically Dendro [26, 51, 56]. Dendro is a well-validated software and has been widely used in various large scale scientific simulations [24, 32, 44–46, 53, 66] and has over 200 citations. Dendro has additional support for carrying out carving operations [66]. We choose Dendro as our benchmark for comparison. For comparison, we choose an elongated channel of dimensions $128 \times 4 \times 1$. The overall mesh is determined by two levels of refinement: base refinement and boundary refinement. Such a channel is commonly found in microfluidic devices, and simulating such devices is an active area of research [19, 54].

Table 4 compares the time for mesh generation and total matvec time[6] for Navier–Stokes equation. Unlike the 3D Poisson operator used for scaling case, the time to compute elemental operator (denoted by leaf matvec) is substantially more expensive. The extra overhead introduced by performing top-down and bottom-up traversal will be significantly smaller compared to performing elemental matvec. The overall time to solve is dominated by load balancing of FEM computation. Since Dendro looks at the complete octree, a significant portion of the elements lie inside the void regions. The partitioning algorithm distributes the elements of complete octree (almost) equally among processors. This leads to an imbalance in the overall FEM computation. This is clear from the presented matvec results. Additionally, in all our runs, we were not able to go beyond the level of 12 with the Dendro framework. This limits our ability to compare for a further elongated channel. In contrast, within the current framework, we achieve a significant improvement in terms of both scalability and time to solve. Overall, we observed a speedup of about 20× for mesh generation and 5× for matvec time. The actual speedup that we can achieve is application-specific and needs to be studied individually. The major factors that determine the overall speedup can be mainly categorized as a) the fraction of volume that can be excluded out; b) complexity of In –Out test; c) the resultant communication pattern.

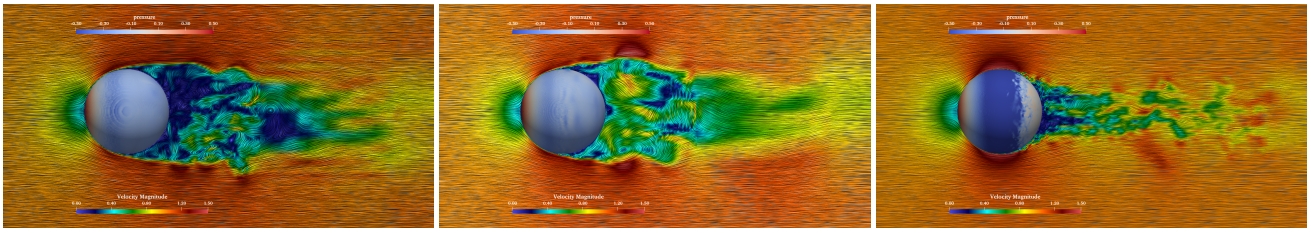
[6]This time include top-down, bottom up, leaf matvec and ghost exchange time

5 Application: Classroom Airflow Simulation

Validation: We first validate the solver by demonstrating the ability to capture the drag crisis by simulating flow past a sphere. A sphere of diameter $d = 1$ is placed at a distance $3d$ from the inlet at $(3d, 3d, 3d)$ in a computation domain of $(10d, 6d, 6d)$. The walls of the domain, except the outlet have constant non-dimensional freestream velocity of $(1, 0, 0)$ and zero pressure gradient. At the outlet, the pressure is set to 0 and zero gradient velocity boundary condition is applied at the wall. No-slip boundary condition (zero Dirichlet) for velocity is imposed on the surface of the sphere. We use a well-established Variational Multiscale (VMS) stabilized Finite Element method for solving the Navier–Stokes equation [12].

Fig. 13 plots the variation of C_d across a range of Reynolds numbers close to the drag crisis regime. We see that the results are in excellent agreement with experimental and other numerical results. We are able to accurately capture the drag crisis phenomena, where a sudden drop in drag from 0.5 [1] - 0.6 [7] at Re around 16,000 to 0.1 [1] - 0.2 [27] at Re of 2 million is observed. The finest resolution simulation consists of ∼40M elements which is significantly lower than for LBM simulation by Geier et al. [27]. We visualize the transition across the drag crises regime in Fig. 14. The drop in drag in Fig. 13 is due to the change in wake structure and pressure distribution in Fig. 14.

Application: We finally demonstrate the ability of our framework to simulate flow past complex geometries. We consider a realistic scenario of modeling airflow in a classroom with complex furniture, seated students with/without computers (and monitors), and a standing instructor. We are particularly interested in accessing if specific locations in the room are at significantly higher risk for transmission – for example, where there is local recirculation causing limited air exchange with the outside environment. In such cases, it becomes imperative to identify if such locations have a higher risk and rank among alternate seating arrangements to mitigate this risk. The current incomplete octree framework allows us to efficiently and rapidly evaluate various seating arrangements and scenarios. In order to *carve* out the geometry, we perform a series of In - Out tests. This gives an automated way to carve out complex geometries from the domain. Fig. 15 shows the computational domain of size $4.83 \times 3.34 \times 1$ that includes complex features such as tables, chairs and mannequins representing students and instructor. The velocity inlets and pressure outlets are located at the top of the domain. The non-dimensional inlet velocity of $(0, 0, -1)$ is imposed at velocity inlets and zero pressure at pressure outlets. $Re = 10^5$ was considered based on the inlet velocity and classroom



(a) Re = 10,000 (b) Re = 160,000 (c) Re = 2,000,000

Fig. 14. The wake structures and pressure distribution on sphere at different Reynolds number. The drag crisis is evident by noticing the wake structure as it changes from being divergent at $Re = 160,000$ (high drag state) to being convergent at $Re = 2 \times 10^6$ (low drag state). At the same time, we observe a high pressure region being developed behind the sphere. The development of this high pressure zone is attributed to the low drag state.

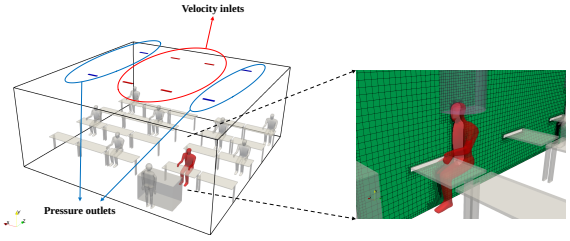


Fig. 15. Figure showing the classroom domain and different regions of boundary conditions such as velocity inlet and pressure outlet. The right side shows the zoomed image with mesh refinement near the object. In all our simulations, we consider that the person marked in red is COVID positive.

Base level	Exit refine level	Body refine level	Elements		Num procs	Immersed (s)		Carved out (s)	
			Active Elements	f_{excess}		Mesh construction	Solve time	Mesh construction	Solve time
6	8	10	924,549	1.53	224	92.36	178.74	38.57	61.56
					448	50.48	94.95	22.47	32.05
6	9	10	1,259,670	1.43	224	136.13	220.6	48	83.3
					448	73.24	95.06	28.88	45.10
7	9	11	3,461,548	1.64	448	210.8	309.60	89.04	131.52
					896	107.88	161.70	53.24	71.3

Table 5: Comparison of mesh generation and solve time for IBM with the current approach for the classroom case. The carved out approach leads to a significant reduction in number of elements. f_{excess} represents the excess fraction of elements obtained as a result of generating complete octree.

height. These flow rates, air exchange rates, and room geometry represent typical values in classrooms seen in US schools.

Fig. 16 illustrates preliminary results enabled by our framework to evaluate the transmission of COVID viral load in the classroom. We evaluate the impact of one infected individual (colored red) who periodically coughs, releasing an aerosolized load of viral particles. We model the time-dependent transmission of the viral load as a scalar transport equation that is advected by a statistically steady–state flow field obtained from the solution of Navier–Stokes solver. We considered a classroom under two different scenarios: with (Fig. 16b) and without (Fig. 16a) the presence of computer monitors. We observe a significant reduction in transmission risk in the case with monitors due to the monitors redirecting the flow field upwards away from the occupied zone.

Tab. 5 compares the IMGA based immersed implementation (based on the open-source code [52]) with the carved-out approach. The overall mesh consists of three refinement levels: base refinement, exit refinement (near the velocity inlet and pressure outlet), and object refinement (near the monitors, tables, and mannequin). To get the refined final mesh, we start the mesh at the base level and successively refine the mesh until the required refinement level is reached. At each iteration, we perform a series of ray-tracing to

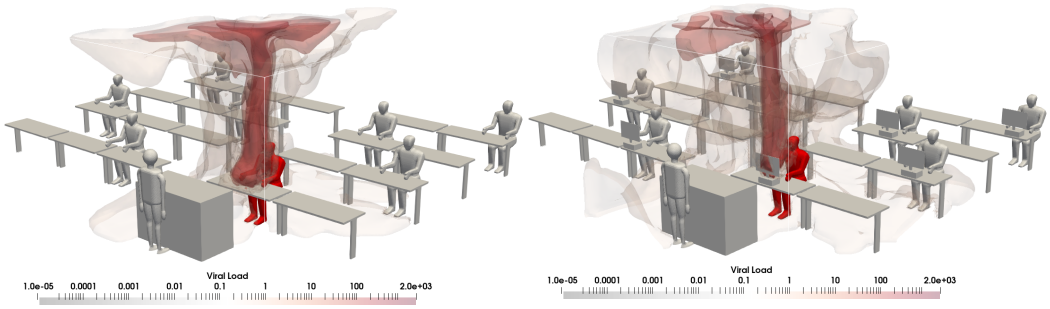
Base level	Exit level	Body Level	Num Elements		Number of Processors				
					224	448	896	1792	3584
7	8	11	5,555,871	Time(s)	344.87	176.15	92.16	46.44	23.95
				Efficiency	1.0	0.98	0.94	0.93	0.90
9	9	11	23,054,077	Time(s)	-	539.15	272.28	142.73	75.7
				Efficiency	-	1.0	0.99	0.94	0.89

Table 6: *Scaling result for classroom simulation:* Comparison of total solve time and strong scaling efficiency with increase in processor count for two different meshes with varying levels of refinement.

determine In or Out relative to the object. Overall, we see an approximately 50% increase in element size for the immersed case. We achieve a speedup of approximately 2.2× during the mesh creation stage and 2.8× for the complete solve time. The speedup obtained in this case is significantly smaller than the channel case described in Sec. 4.6 mainly because of the nature of objects being carved out. The mannequin or the table considered here has a large surface area to volume ratio. The small volume resulted in most of the octants percolating close to the finest level before they can be discarded. Additionally, ray-tracing based In -Out test needs to be performed at each iteration of refinement, which is quite expensive [53]. Once the mesh is generated, we see a significant speedup in the overall solve time. Tab. 6 compares the scaling efficiency for two different meshes. Overall we achieve a good scaling efficiency of about 90% over 16 fold increase in the number of processors.

6 Conclusion

We present a fast and scalable tree-based mesh generation that is not limited to isotropic domains, which serves as an alternative to using two-tier meshes that are not dependent on having top-level hexahedral meshes. The algorithms presented in the paper are dimension agnostic and provides a generic way to handle any arbitrary geometries. Our approach allows all elements to remain isotropic, which speeds up assembly and does not affect conditioning due to element stretching. The scaling behavior of the MATVEC, which is the most dominant part of any FEM solver, has been verified up to $O(16K)$ cores. We further showcase the applicability of these algorithms by solving Navier–Stokes for a large-scale 3D problem in the presence of complex geometries. These algorithmic features allows fast, well-balanced creation of complex meshes and efficient solvers that open the way for parametric exploration of very large-scale simulations (as our example simulation suggests). In future, we plan to extend the algorithms to incorporate DG based FEM along with Finite Difference and Finite Volume Methods.



(a) Without monitors **(b) With monitors**

Fig. 16. *Classroom scenario:* Evaluation of viral load (in quanta / m^3) in two classroom scenarios with and without monitors. The mannequin marked in the red is infected with COVID and transmits the virus. The isocontours represent the regions of different viral load concentrations in space.

References

[1] Elmar Achenbach. 1972. Experiments on the flow past spheres at very high Reynolds numbers. *Journal of Fluid Mechanics* 54, 3 (1972), 565–575.

[2] Gilou Agbaglah, Sébastien Delaux, Daniel Fuster, Jérôme Hoepffner, Christophe Josserand, Stéphane Popinet, Pascal Ray, Ruben Scardovelli, and Stéphane Zaleski. 2011. Parallel simulation of multiphase flows using octree adaptivity and the volume-of-fluid method. *Comptes Rendus Mecanique* 339, 2-3 (2011), 194–207.

[3] Mohammad W Akhtar and Stanley J Kleis. 2013. Boiling flow simulations on adaptive octree grids. *International journal of multiphase flow* 53 (2013), 88–99.

[4] Jaber Almedeij. 2008. Drag coefficient of flow around a sphere: Matching asymptotically the wide trend. *Powder Technology* 186, 3 (2008), 218–223.

[5] Nabil M Atallah, Claudio Canuto, and Guglielmo Scovazzi. 2020. The second-generation Shifted Boundary Method and its numerical analysis. *Computer Methods in Applied Mechanics and Engineering* 372 (2020), 113341.

[6] Michael Bader. 2012. *Space-filling curves: an introduction with applications in scientific computing*. Vol. 9. Springer Science & Business Media.

[7] Vukman Bakic. 2003. Experimental investigation of turbulent flows around a sphere. *TU Hamburg–Harburg, Schriftenreihe Schiffbau, Bericht Nr. 621, Juli 2003* (2003).

[8] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2019. PETSc Web page. https://www.mcs.anl.gov/petsc. https://www.mcs.anl.gov/petsc

[9] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2020. *PETSc Users Manual*. Technical Report ANL-95/11 - Revision 3.14. Argonne National Laboratory. https://www.mcs.anl.gov/petsc

[10] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 1997. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen (Eds.). Birkhäuser Press, 163–202.

[11] Peter Bastian, Markus Blatt, Andreas Dedner, Christian Engwer, Robert Klöfkorn, Mario Ohlberger, and Oliver Sander. 2008. A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework. *Computing* 82, 2-3 (2008), 103–119.

[12] Y Bazilevs, VM Calo, JA Cottrell, TJR Hughes, A Reali, and G Scovazzi. 2007. Variational multiscale residual-based turbulence modeling for large eddy simulation of incompressible flows. *Computer methods in applied mechanics and engineering* 197, 1-4 (2007), 173–201.

[13] Benjamin Karl Bergen and Frank Hülsemann. 2004. Hierarchical hybrid grids: data structures and core algorithms for multigrid. *Numerical Linear Algebra with Applications* 11, 2-3 (2004), 279–291. https://doi.org/10.1002/nla.382 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nla.382

[14] Jacobo Bielak, Omar Ghattas, and EJ Kim. 2005. Parallel octree-based finite element method for large-scale earthquake ground motion simulation. *Computer Modeling in Engineering and Sciences* 10, 2 (2005), 99.

[15] Erik Burman and Peter Hansbo. 2012. Fictitious domain finite element methods using cut elements: II. A stabilized Nitsche method. *Applied Numerical Mathematics* 62, 4 (2012), 328–341.

[16] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, Eh Tan, T. Tu, L. C. Wilcox, and S. Zhong. 2008. Scalable adaptive mantle convection simulation on petascale supercomputers. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–15. https://doi.org/10.1109/SC.2008.5214248

[17] 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.

[18] Jose J Camata and Alvaro LGA Coutinho. 2013. Parallel implementation and performance analysis of a linear octree finite element mesh generation scheme. *Concurrency and Computation: Practice and Experience* 25, 6 (2013), 826–842.

[19] Lei Chai, Guodong Xia, Mingzheng Zhou, and Jian Li. 2011. Numerical simulation of fluid flow and heat transfer in a microchannel heat sink with offset fan-shaped reentrant cavities in sidewall. *International Communications in Heat and Mass Transfer* 38, 5 (2011), 577–584.

[20] Dawson-Haggerty et al. [n.d.]. *trimesh*. https://trimsh.org/

[21] Raphael Egan, Arthur Guittet, Fernando Temprano-Coleto, Tobin Isaac, François J Peaudecerf, Julien R Landel, Paolo Luzzatto-Fegiz, Carsten Burstedde, and Frederic Gibou. [n.d.]. Direct numerical simulation of incompressible flows on parallel Octree grids. *J. Comput. Phys.* 428 ([n. d.]), 110084.

[22] M Esmaily, Lluís Jofre, Ali Mani, and Gianluca Iaccarino. 2018. A scalable geometric multigrid solver for nonsymmetric elliptic systems with application to variable-density flows. *J. Comput. Phys.* 357 (2018), 142–158.

[23] Milinda Fernando, Dmitry Duplyakin, and Hari Sundar. 2017. Machine and application aware partitioning for adaptive mesh refinement applications. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 231–242.

[24] Milinda Fernando, David Neilsen, Hyun Lim, Eric Hirschmann, and Hari Sundar. 2019. Massively Parallel Simulations of Binary Black Hole Intermediate-Mass-Ratio Inspirals. *SIAM Journal on Scientific Computing* 41, 2 (2019), C97–C138.

[25] Milinda Shayamal Fernando and Hari Sundar. 2020. paralab/Dendro-5.01: Local timestepping on octree grids. (Jun 2020). https://doi.org/10.5281/zenodo.3879315

[26] Milinda Shayamal Fernando and Hari Sundar. 2020. *paralab/Dendro-5.01: LTS work*. https://doi.org/10.5281/zenodo.3876881

[27] Martin Geier, Andrea Pasquali, and Martin Schönherr. 2017. Parametrization of the cumulant lattice Boltzmann method for fourth order accurate diffusion Part II: Application to flow around a sphere at drag crisis. *J. Comput. Phys.* 348 (2017), 889–898.

[28] Deborah M Greaves and AGL Borthwick. 1999. Hierarchical tree-based finite element mesh generation. *Internat. J. Numer. Methods Engrg.* 45, 4 (1999), 447–471.

[29] Boyce E Griffith, Richard D Hornung, David M McQueen, and Charles S Peskin. 2007. An adaptive, formally second order accurate version of the immersed boundary method. *Journal of computational physics* 223, 1 (2007), 10–49.

[30] Masado Ishii, Milinda Fernando, Kumar Saurabh, Biswajit Khara, Baskar Ganapathysubramanian, and Hari Sundar. 2019. Solving PDEs in space-time: 4D tree-based adaptivity, mesh-free and matrix-free approaches. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–61.

[31] Shin K Kang and Yassin A Hassan. 2011. A comparative study of direct-forcing immersed boundary-lattice Boltzmann methods for stationary complex boundaries. *International Journal for Numerical Methods in Fluids* 66, 9 (2011), 1132–1158.

[32] Makrand A Khanwale, Kumar Saurabh, Milinda Fernando, Victor M Calo, James A Rossmanith, Hari Sundar, and Baskar Ganapathysubramanian. 2020. A fully-coupled framework for solving Cahn-Hilliard Navier-Stokes equations: Second-order, energy-stable numerical methods on adaptive octree based meshes. *arXiv preprint arXiv:2009.06628* (2020).

[33] E Kim, Jacobo Bielak, and Omar Ghattas. 2003. Large-scale northridge earthquake simluation using octree-based multiresolution mesh method. In *Proceedings of the 16th ASCE Engineering Mechanics Conference*. Seattle, WA, USA;.

[34] Samuli Laine and Tero Karras. 2010. Efficient sparse voxel octrees–analysis, extensions, and implementation. *NVIDIA Corporation* 2 (2010).

[35] Jinmo Lee and Donghyun You. 2013. An implicit ghost-cell immersed boundary method for simulations of moving body problems with control of spurious force oscillations. *J. Comput. Phys.* 233 (2013), 295–314.

[36] Marc Levoy, J Gerth, B Curless, and K Pull. 2005. The Stanford 3D scanning repository. *URL http://www-graphics. stanford. edu/data/3dscanrep* 5 (2005).

[37] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. 2004. Simulating water and smoke with an octree data structure. In *ACM SIGGRAPH 2004 Papers*. 457–462.

[38] Michael Macri and Suvranu De. 2008. An octree partition of unity method (OctPUM) with enrichments for multiscale modeling of heterogeneous media. *Computers & structures* 86, 7-8 (2008), 780–795.

[39] Alex Main and Guglielmo Scovazzi. 2018. The shifted boundary method for embedded domain computations. Part I: Poisson and Stokes problems. *J. Comput. Phys.* 372 (2018), 972–995.

[40] Ali Mani. 2012. Analysis and optimization of numerical sponge layers as a nonreflective boundary treatment. *J. Comput. Phys.* 231, 2 (2012), 704–716.

[41] MATLAB. 2018. *9.7.0.1190202 (R2019b)*. The MathWorks Inc., Natick, Massachusetts.

[42] Rajat Mittal and Gianluca Iaccarino. 2005. Immersed boundary methods. *Annu. Rev. Fluid Mech.* 37 (2005), 239–261.

[43] Faith A Morrison. 2013. *An introduction to fluid mechanics*. Cambridge University Press.

[44] Jayanta Mukherjee and William D Gropp. 2010. *Performance evaluation and enhancement of Dendro*. Technical Report.

[45] David Neilsen, Milinda Fernando, Hari Sundar, and Eric Hirschmann. 2019. Dendro-GR: A scalable framework for Adaptive Computational General Relativity on Heterogeneous Clusters. In *APS April Meeting Abstracts*, Vol. 2019. G11–003.

[46] David Neilsen, Milinda Fernando, Hari Sundar, Eric Hirschmann, and Hyun Lim. 2018. Massively Parallel Simulations of Binary Black Hole Intermediate-Mass-Ratio Inspirals. *APS* 2018 (2018), D14–005.

[47] Charles S Peskin. 1977. Numerical analysis of blood flow in the heart. *Journal of computational physics* 25, 3 (1977), 220–252.

[48] 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.

[49] Johann Rudi, A Cristiano I Malossi, Tobin Isaac, Georg Stadler, Michael Gurnis, Peter WJ Staar, Yves Ineichen, Costas Bekas, Alessandro Curioni, and Omar Ghattas. 2015. An extreme-scale implicit solver for complex PDEs: highly heterogeneous flow in earth's mantle. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–12.

[50] Hanan Samet and Andrzej Kochut. 2002. Octree approximation and compression methods.. In *3DPVT*. Citeseer, 460–469.

[51] Rahul S Sampath, Santi S Adavani, Hari Sundar, Ilya Lashuk, and George Biros. 2008. Dendro: parallel algorithms for multigrid and AMR methods on 2: 1 balanced octrees. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE, 1–12.

[52] Kumar Saurabh, Boshun Gao, Milinda Fernando, Hari Sundar, and Baskar Ganapathysubramanian. 2020. Flow simulations integrating adaptive octree meshes with immersogeometric analysis. (Jun 2020). https://doi.org/10.5281/zenodo.3879392

[53] Kumar Saurabh, Boshun Gao, Milinda Fernando, Songzhe Xu, Makrand A Khanwale, Biswajit Khara, Ming-Chen Hsu, Adarsh Krishnamurthy, Hari Sundar, and Baskar Ganapathysubramanian. 2021. Industrial scale Large Eddy Simulations with adaptive octree meshes using immersogeometric analysis. *Computers & Mathematics with Applications* 97 (2021), 28–44.

[54] C Shen, DB Tian, C Xie, and J Fan. 2004. Examination of the LBM in simulation of microchannel flow in transitional regime. *Microscale Thermophysical Engineering* 8, 4 (2004), 423–432.

[55] Jeffrey Slotnick, Abdollah Khodadoust, Juan Alonso, David Darmofal, William Gropp, Elizabeth Lurie, and Dimitri Mavriplis. 2014. CFD vision 2030 study: a path to revolutionary computational aerosciences.

[56] 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.

[57] 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.

[58] Tiankai Tu, David R O'Hallaron, and Omar Ghattas. 2005. Scalable parallel octree meshing for terascale applications. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE, 4–4.

[59] Markus Uhlmann. 2005. An immersed boundary method with direct forcing for the simulation of particulate flows. *J. Comput. Phys.* 209, 2 (2005), 448–476.

[60] Vasco Varduhn, Ming-Chen Hsu, Martin Ruess, and Dominik Schillinger. 2016. The tetrahedral finite cell method: higher-order immersogeometric analysis on adaptive non-boundary-fitted meshes. *Internat. J. Numer. Methods Engrg.* 107, 12 (2016), 1054–1079.

[61] Ryan Viertel and Braxton Osting. 2019. An approach to quad meshing based on harmonic cross-valued maps and the Ginzburg–Landau theory. *SIAM Journal on Scientific Computing* 41, 1 (2019), A452–A479.

[62] Zeli Wang, Jianren Fan, and Kun Luo. 2008. Combined multi-direct forcing and immersed boundary method for simulating flows with moving particles. *International Journal of Multiphase Flow* 34, 3 (2008), 283–302.

[63] Tobias Weinzierl. 2019. The Peano software—parallel, automaton-based, dynamically adaptive grid traversals. *ACM Transactions on Mathematical Software (TOMS)* 45, 2 (2019), 1–41.

[64] Kyu-Young Whang, Ju-Won Song, Ji-Woong Chang, Ji-Yun Kim, Wan-Sup Cho, Chong-Mok Park, and Il-Yeol Song. 1995. Octree-R: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics* 1, 4 (1995), 343–349.

[65] Fei Xu, Dominik Schillinger, David Kamensky, Vasco Varduhn, Chenglong Wang, and Ming-Chen Hsu. 2016. The tetrahedral finite cell method for fluids: Immersogeometric analysis of turbulent flow around complex geometries. *Computers & Fluids* 141 (2016), 135–154.

[66] Songzhe Xu, Boshun Gao, Alec Lofquist, Milinda Fernando, Ming-Chen Hsu, Hari Sundar, and Baskar Ganapathysubramanian. 2021. An octree-based immersogeometric approach for modeling inertial migration of particles in channels. *Computers & Fluids* 214 (2021), 104764.

[67] Hongfeng Yu, Jinrong Xie, Kwan-Liu Ma, Hemanth Kolla, and Jacqueline H Chen. 2015. Scalable parallel distance field construction for large-scale applications. *IEEE transactions on visualization and computer graphics* 21, 10 (2015), 1187–1200.

A Artifact Description

A.1 Libraries dependencies

The following dependencies are required to compile the code:

- C/C++ compilers with C++11 standards and OpenMP support
- MPI implementation (e.g. openmpi, mvapich2)
- PETSc 3.8 or higher
- ZLib compression library (used to write `.vtu` files in binary format with compression enabled)
- MKL / LAPACK library
- CMake 2.8 or higher version
- libconfig for parameter reading from file.

A.2 Frontera environment

Experiments performed in Frontera are executed in the following module environment.

```
Currently Loaded Modulefiles:
  1) intel/19.0.5   4) python3/3.7.0
  2) impi/19.0.5    5) autotools/1.2
  3) petsc/3.12     6) cmake/3.16.1
```

A.3 Frontera compute node configuration

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                56
On-line CPU(s) list:   0-55
Thread(s) per core:    1
Core(s) per socket:    28
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon
                       Platinum 8280
                       CPU @ 2.70GHz
Stepping:              7
CPU MHz:               2700.000
BogoMIPS:              5400.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              39424K

MemTotal:       195920208 kB
MemFree:        168962328 kB
MemAvailable:   168337408 kB
```

B Artifact Evaluation

B.1 Signed distance computation

Let M denotes the closed orientable 2-manifold triangular mesh. The signed distance from a point $\mathbf{p}$ rto M is given by:

$$d(\mathbf{p}, M) = \inf_{\mathbf{x} \in M} \|\mathbf{p} - \mathbf{x}\| \, \text{sign}(\mathbf{n} \cdot (\mathbf{p} - \mathbf{c})) \tag{1}$$

where: $\mathbf{c}$ denotes the closest point to $\mathbf{p}$ and $\mathbf{n}$ denotes the outward normal. A positive value of d, means the point is inside the surface and vice-versa.

B.2 Details of solver selection

PETSc was used to solve all the linear algebra problems. In particular, bi-conjugate gradient descent (`-ksp_type bcgs`) solver was used in conjunction with Additive - Schwartz (`-pc_type asm`) preconditioner to solve the linear system of equations. The NEWTONLS class by PETSc, that implements a Newton Line Search method, was used for the nonlinear problems. Both the relative residual tolerance and the absolute residual tolerance for linear and non - linear solve are set to 10^{-6} in all numerical results.

B.3 Downloading and installing the code

This section presents how to run and reproduce the results presented in the paper. You can clone the repository using, `git clone git@github.com:abcd/sc21-kt.git`. We use `CMake` to configure and build. In Frontera node,

- `git clone git@github.com:abcd/sc21-kt.git`
- `mkdir build && cd build`
- Load the module environment
- `mkdir build && cd build`
- `cmake ../.`
- `make MVCChannel MVCSphere signedDistance`

B.4 Running experiments

B.4.1 `MVCChannel`*:* Scaling run for channel case. In order to run the scaling case, it requires the 3 parameters: a) baseLevel b) boundaryLevel and c) element order (1/2). For example, on Frontera it can be ran as:

```
ibrun MVCChannel 10 12 1 log10_12.out
```

where, 10 is the base refinement level, 12 is the boundary refinement level, 1 is the element order and `log10_12.out` is the output file containing the relevant timing information.

B.4.2 `MVCSphere`*:* Scaling run for sphere case. In order to run the scaling case, it requires the 3 parameters: a) baseLevel b) boundaryLevel and c) element order (1/2). For example, on Frontera it can be ran as:

```
ibrun MVCSphere 7 12 1 log7_12.out
```

where, 7 is the base refinement level, 12 is the boundary refinement level, 1 is the element order and `log7_12.out` is the output file containing the relevant timing information.

B.4.3 `signedDistance`*:* The computation of signedDistance. In order to find the signed distance, we successively refined near the boundaries and computed the signed distance. In order to run the code:

```
ibrun signedDistance stlFileName 4 14
```

where: `stlFileName` is the name of stl file, 4 is the minimum refinement level and 14 is the maximum level of refinement at the stl boundary. After each successive iteration, the code output the

information of boundary nodes. Then we compute the signed distance, using the python script provided under the `scripts` folder. In order the run the python scripts:

`python3 signedDistance stlFileName`.

Note that you might need to change the number of processor on your machine as the python script is parallel and make use of `multiprocessing` library.

B.4.4 Roofline plot: We computed the roofline using Intel Advisor. In order to run the roofline plot, first run the survey using:

```
ibrun -np 1 advixe-cl -collect survey -project-dir
outputDir -search-dir
src:=examples/BenchMark_channel/src – MVCChannel
baseLevel boundaryLevel eleOrder outputFile
```

where: outputDir is the directory to store output and `MVCChannel baseLevel boundaryLevel eleOrder outputFile` is the same as in previous description of Channel scaling.

Finally in order to collect FLOPS count:

```
ibrun -np 1 advixe-cl -collect=tripcounts –flop
–mark-up-list= src/benchmark.cpp -project-dir= outputDir
– MVCChannel baseLevel
boundaryLevel eleOrder outputFile
```

where: outputDir must be same directory as the survey directory and `MVCChannel` must be called with the same arguments.