Scalable adaptive PDE solvers in arbitrary domains

Kumar Saurabh§
Iowa State University
Ames, Iowa, U.S.A

Masado Ishii§
University of Utah
Salt Lake City, Utah, U.S.A

Milinda Fernando
University of Utah
Salt Lake City, Utah, U.S.A

Boshun Gao
Iowa State University
Ames, Iowa, U.S.A

Kendrick Tan
Iowa State University
Ames, Iowa, U.S.A

Ming-Chen Hsu
Iowa State University
Ames, Iowa, U.S.A

Adarsh Krishnamurthy
Iowa State University
Ames, Iowa, U.S.A

Hari Sundar
University of Utah
Salt Lake City, Utah, U.S.A

Baskar
Ganapathysubramanian
Iowa State University
Ames, Iowa, U.S.A

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.

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, 39, 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 Is 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, 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.

**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.

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.

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 sub-rectangle 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 (octrees) 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 \( C' = \Omega \setminus C \). 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 \( \partial C \), the red circle.)

Suppose that \( \Omega \) is hierarchically partitioned according to an octree, \( T \), which captures \( \partial C \) well enough under some metric. Any octant \( e \) of \( T \) 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 \( \bar{e} \notin C \).
   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 \( T_l \equiv T \setminus \{ \text{carved leaves} \} \). 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

\[
F(\bar{e}) := \begin{cases} 
\text{label(“carved”) if } \bar{e} \subset C \\
\text{label(“retain-internal”) if } \bar{e} \subset C' \\
\text{label(“retain-boundary”) otherwise }
\end{cases}
\]

- 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 \( \partial C \) 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 is flush with \( \partial C \). 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 importance 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 cubic 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.

---

**Algorithm 1 ConstructUniform**

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

```plaintext
1: if \( F(S) \neq \text{Carved} \) then \( \triangleright \) Else prune
2: \hspace{1em} if level of \( S \geq L \) then
3: \quad \hspace{1em} T.push(S) \hspace{1em} \( \triangleright \) Leaf.
4: \hspace{1em} else
5: \quad \hspace{1em} \hspace{1em} for \( c_{dL} \leftarrow 1 \text{ to } 2^{dL} \) do \( \triangleright \) Regional SFC order
6: \quad \hspace{1em} \hspace{1em} c_{dL} \leftarrow I.sfc2Morton(c_{dL}) \hspace{1em} \( \triangleright \) Morton to Morton
7: \quad \hspace{1em} \hspace{1em} \text{ConstructUniform}(S.child(c_{dL}), L, child(c_{dL}))
```

---
Algorithm 3 DistributedConstructConstrained

Require: Distributed set of seed octants \( B \), function \( F() \).
Ensure: Distributed set of seed octants \( B \), function \( F() \).

1. if \( F_S \neq \) Carved then \( \rightarrow \) Else prune
2. \( L \) ← finest level in \( B \)
3. if \( |B| = 0 \) or level of \( S \geq L \) then 
   4. \( T \) push(S) \( \rightarrow \) Leaf
   5. else 
      6. \( \rightarrow \) Bucket seeds to SFC-sorted children of \( S \).
      7. \( I \) ← level(S) + 1
      8. \( \text{counts} \left[ 2^{	ext{dim}} \right] \) ← 0
      9. for \( b \in B \) do
         10. \( \text{counts} \left[ \text{child}_n(b, l) \right] \) ++
         11. offsets() ← permute(counts, \( I \))
         12. \( \text{ offsets() } \) scan(counts)
      13. \( \text{for } \epsilon_l \left[ 1 \rightarrow 2^\text{dim} \right] \) do
         14. \( \text{for } \epsilon_{\text{dim}} \left[ 1 \rightarrow 2^\text{dim} \right] \) do 
            15. \( \text{for } \epsilon_{\text{dim}} \left[ 1 \rightarrow 2^\text{dim} \right] \) do 
               16. \( S_c \text{ } \left[ \text{sfc2Morton}(\epsilon_{\text{dim}}) \right] 
               17. \( L_c \left[ \text{child}((\epsilon_{\text{dim}}) \right] 
               18. \( B_c \left[ \text{slice} \left( \text{offsets}[c], \text{offsets}[c] \left[ \text{count}[l] \right] \right) 
               19. \text{ConstructConstrained}(S_c, L_c, B_c)

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 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.

Algorithm 4 DistributedConstruct2to1Balanced

Require: Distributed set of seed octants \( B \), function \( F() \).
Ensure: Distributed set of seed octants \( B \), function \( F() \).

1. \( T_{aux} \) ← DistributedConstructConstrained(\( B \), \( F() \)) \( \rightarrow \) \( F \) not applied
2. \( T_2 \) ← BottomUpConstrainNeighbors(\( T_1 \))
3. \( T_3 \) ← DistributedConstructConstrained(\( T_2 \), \( F() \))
4. return \( T_3 \)

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 (\(8\)). In the middle figure auxiliary balanced octants are showed in (\(9\)), in other words these are the octants needed to remove the balance constraint violation in (\(8\)). Right most figure shows the constructed octree with auxiliary balanced octants which satisfies the 2:1 balanced constraint. The nodes marked by gray circles in the final 2:1 balanced mesh are hanging nodes.

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. 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 subtre 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 \( \text{length}_{\text{element}} / p \) 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: \( \nu_{\text{glob}}[\text{map}[e \ast npe + i]] += \nu_{\text{loc}} \). 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.

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 sub-trees 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 (id_row, id_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 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. Fig. 5a shows the voxel representation for the Stanford dragon. Fig. 5b compares the $L_{\infty}$ error of computed signed distance between

![Voxelized](image)

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 of 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 DOFs | Incomplete octree DOFs |
|-----------------|----------------------|------------------------|
| 1               | 1089                 | 402.6                  |
| 2               | 1089                 | 466.7                  |
| 4               | 1089                 | 510.1                  |
| 8               | 1089                 | 512.0                  |
| 16              | 1089                 | 10580.5                |

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.

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 $\Delta$ 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.

4.3 Convergence Test for Discrete Operators

Here, we present the convergence analysis for the Poisson operator $\nabla u = f$ over the domain $\Omega$ with $u = u_D$ on the domain boundary $\Gamma$. Inserting appropriate finite dimensional function spaces for trial and test function, the weak form of the Poisson operator can be written as:

$$
(\nabla w_h, \nabla u)_{\Omega} = (w_h, f)_{\Omega}
$$

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

![Trimesh](image)

evaluated with Matlab cond test command
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, $\alpha$ is the penalty term, $h$ is the element length, and $d$ is the distance vector from the boundary surface of voxelated domain $\tilde{\Gamma}$ to the true surface $\Gamma$. The main idea of the method is to shift the boundary condition from $\Gamma$ 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 a 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:

$$u(r) = 0.25\sqrt{R^2 - r^2}$$

where $r = \sqrt{(x - x_0)^2 + (y - y_0)^2}$. Fig. 6 shows the convergence behaviour for the linear basis function. If we naïvely apply the boundary condition at the boundary nodes of the voxelated geometry, we only get a first-order convergence in both $L_2$ and $L_\infty$ 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 $L_\infty$ 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 $\text{Is}$ (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 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.

### 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.

| Refine Level | Sphere | Stanford | Dragon |
|--------------|--------|----------|--------|
| 11           | 1.75   | 1.84     | 1.36   |
| 12           | 1.79   | 1.87     | 1.39   |
| 13           | 1.81   | 1.90     | 1.41   |
| 14           | 1.81   | 1.92     | 1.43   |

Table 2: Comparison of the ratio of number of elements ($f_{\text{elem}}$) and degrees of freedom ($f_{\text{DOF}}$) 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.

Fig. 6. Convergence plot: Figure showing the convergence behaviour for the Poisson operator on a two-dimensional circular disk.
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 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 $16 \times 1 \times 1$ 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 $\times$ 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 $4.2 \times$ 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.

3More formally, consider a mesh $M$ with $N$ global elements distributed over $p$ processors. If we globally number the elements of mesh from $0 \cdots N − 1$, then if a processor $k$, $k \leq p$, receives $m_1 \cdots m_p$ ($m_i$’s being the global element number) elements for linear, $0 \leq m_i \leq m_p \leq N − 1$, then the processor $k$ for quadratic mesh will also receive the same sequence of elements $m_1 \cdots m_p$.

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

| Case Type | Element Order | Strong scaling | Weak Scaling |
|-----------|--------------|---------------|--------------|
|           |              | Num elements | Num DOFs | Efficiency | Num elements/core | Efficiency |
| Channel   | Linear       | 13.5 M       | 13.7 M    | 0.81       | 35K             | 0.82       |
| (Sec. 4.5.1) | Quadratic    | 13.5 M       | 101.9 M   | 0.90       | 35K             | 0.86       |
| Sphere    | Linear       | 17.5 M       | 17.4 M    | 0.99       | 10K             | 0.74       |
| (Sec. 4.5.2) | Quadratic    | 17.5 M       | 135.7 M   | 0.96       | 10K             | 0.81       |

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

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).

SC’21, November 14–19, 2021, St. Louis, MO Saurabh and Ishii, et al.

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.

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 partitioning for both the linear and quadratic is similar at the
With the increase in the number of processors, the total 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.

Additionally, we also analyzed the distribution of the ratio of ghost nodes to the number of owned nodes (denoted by \( \eta \)), 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 \( \eta \) increases, and in extreme limit of parallelization, when each processor has only one element, \( \eta \rightarrow 1 \). It is non-trivial to analyze the exact rate of increase in \( \eta \) 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 \(5^5\) (AI) for quadratic \((0.121)\) as compared to linear \((0.072)\) elements. The amount of data needed for matvec computation grows as \(O((p + 1)^2)\) 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 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 \([1]\) to 0.6 \([7]\) at \(Re\) around 16,000 to 0.1 \([1]\) to 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 crisis regime in Fig. 14. The 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 20x for mesh generation and 5x 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.

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

\(^5\)AI is measured as the amount of floating point operations performed per byte of data loaded into the memory.

\(^6\)This time include top-down, bottom up, leaf matvec and ghost exchange time.
We evaluate the impact of one infected individual (colored red) who periodically coughs, releasing an aerosolized load of viral particles. We consider a time-dependent transmission of the viral load to simulate flow past complex geometries. We consider a Reynolds number \( Re \) solver. We considered a classroom under two different scenarios: steady–state flow field obtained from the solution of Navier–Stokes equations and time-dependent transmission of the viral load to evaluate the transmission of COVID viral load in the classroom. Typical values in classrooms seen in US schools. These flow rates, air exchange rates, and room geometry are imposed at velocity inlets and zero pressure at pressure outlets. The non-dimensional inlet velocity of instructor. The velocity inlets and pressure outlets are located at the top of the domain. The non-dimensional inlet velocity of \( Re = 10,000 \) is considered based on the inlet velocity and classroom height. These flow rates, air exchange rates, and room geometry represent typical values in classrooms seen in US schools.

![Fig. 14](image1.png) \( Re = 10,000 \)

![Fig. 14](image2.png) \( Re = 160,000 \)

![Fig. 14](image3.png) \( Re = 2,000,000 \)

**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 \( Re = 2 \times 10^5 \) (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](image4.png)

**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.

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. \( Re = 10^3 \) was considered based on the inlet velocity and classroom height. These flow rates, air exchange rates, and room geometry represent typical values in classrooms seen in US schools.

![Fig. 16](image5.png) \( f_{\text{excess}} \)

**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:

- \( Re = 10,000 \)
- \( Re = 160,000 \)
- \( Re = 2,000,000 \)

![Fig. 17](image6.png)

**Fig. 17.** 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. \( f_{\text{excess}} \) represents the excess fraction of elements obtained as a result of generating complete octree.

**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. \( f_{\text{excess}} \) represents the excess fraction of elements obtained as a result of generating complete octree.

| Base level | Exit level | Body Level | Num Elements | Elements \( f_{\text{excess}} \) | Num proc | Mesh construction time | Mesh construction time | Mesh construction time | Mesh construction time |
|------------|------------|------------|--------------|-----------------------------|---------|----------------------|----------------------|----------------------|----------------------|
| 6          | 8          | 10         | 924,549,153  | 224                         | 92,36   | 178,74               | 38,57                | 44,55                | 44,55                |
| 7          | 9          | 10         | 3,359,670,183| 224                         | 73,24   | 95,86                | 28,88                | 45,10                | 45,10                |
| 7          | 9          | 11         | 2,961,548,164| 448                         | 210,8   | 309,68               | 89,04                | 131,12               | 131,12               |

**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.

| Base level | Exit level | Body Level | Num Elements | Number of Processors |
|------------|------------|------------|--------------|----------------------|
| 7          | 8          | 11         | 5,555,787    | 344,87               |
| 9          | 9          | 11         | 23,054,077   | 539,15               |

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 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 \( \times \) during the mesh creation stage and 2.8 \( \times \) 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

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.
We present a fast and scalable tree-based mesh generation that is 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 provide 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 allow 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.

6 Conclusion

We present a fast and scalable tree-based mesh generation that is 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 provide 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 allow 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.

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] Gouw Aghaugh, Sébastien Delaux, Daniel Puster, Jérôme Hoerdtner, Christophe Jossenard, 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 Mécanique 339, 2-3 (2011), 194–207.

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

[4] Jaber Almedjed. 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] Vukan Bakić. 2003. Experimental investigation of turbulent flows around a sphere. TU Hamburg–Harburg, Schriftenreihe Schiffbau, Bericht Nr. 121, Juli 2003 (2003).

[8] Satish Balay, Shirirang 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

[9] Satish Balay, Shirirang 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 Bazileva, VM Calo, J. Cottrell, T. 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

[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
[67] Hongfeng Yu, Jinxong 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
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: 168862328 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 $p$ to $M$ is given by:

$$d(p, M) = \inf_{x \in M} \|p - x\| \text{sign}(n \cdot (p - c))$$  \hspace{1cm} (3)

where $c$ denotes the closest point to $p$ and $n$ 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 (\texttt{-ksp_type bcgs}) solver was used in conjunction with Additive - Schwartz (\texttt{-pc_type asm}) preconditioner to solve the linear system of equations. The \texttt{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, \texttt{git clone bitbucket.org/baskargroup/sc21-kt.git}. We use CMake to configure and build. In Frontera node,

- \texttt{git clone bitbucket.org/baskargroup/sc21-kt.git}
- \texttt{mkdir build && cd build}
- \texttt{Load the module environment}
- \texttt{mkdir build && cd build}
- \texttt{cmake ../.}
- \texttt{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:

\texttt{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 \texttt{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:

\texttt{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 \texttt{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:

\texttt{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 to run the python script:

\texttt{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.

\textbf{B.4.4 Roofline plot:} We computed the roofline using Intel Advisor.

In order to run the roofline plot, first run the survey using:

\texttt{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:

\texttt{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.