Journal of Computational Physics 502 (2024) 112823

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

An efficient GPU-based *h*-adaptation framework via linear trees for the flux reconstruction method

Lai Wang [a,*,1], Freddie Witherden [b], Antony Jameson [b,c]

[a] Gaithersburg, 20878, MD, USA
[b] Department of Ocean Engineering, Texas A&M University, College Station, 77843, TX, USA
[c] Department of Aerospace Engineering, Texas A&M University, College Station, 77843, TX, USA

ARTICLE INFO

Keywords:
Linear octree
Linear quadtree
Flux reconstruction
AMR
GPU

ABSTRACT

In this paper, we develop the first entirely graphics processing unit (GPU) based *h*-adaptive flux reconstruction (FR) method with linear trees. The adaptive solver fully operates on the GPU hardware, using a linear quadtree for two dimensional (2D) problems and a linear octree for three dimensional (3D) problems. We articulate how to efficiently perform tree construction, 2:1 balancing, connectivity query, and how to perform adaptation for the flux reconstruction method on the GPU hardware. As a proof of concept, we apply the adaptive flux reconstruction method to solve the inviscid isentropic vortex propagation problem on 2D and 3D meshes and transitional flow over a 3D square cylinder at $\mathrm{Re} = 400$ to demonstrate the efficiency of the developed adaptive FR method on a single GPU card. Depending on the computational domain size, acceleration of one or two orders of magnitude can be achieved compared to uniform meshing for long distance transportation. The total computational cost of adaption, including tree manipulations, connectivity query and data transfer, compared to that of the numerical solver, is insignificant. It can be less than 2% of the total wall clock time for 3D problems when we perform adaptation every 10-40 time steps with an explicit 3-stage Runge–Kutta time integrator.

1. Introduction

The computational fluid dynamics (CFD) 2030 vision study stated that "An engineer/scientist must be able to generate, analyze, and interpret a large ensemble of related simulations in a time-critical period (e.g., 24 hours), without individually managing each simulation, to a pre-specified level of accuracy and confidence" [1]. To this end, we would like to reduce the human intervention as much as possible in all stages of CFD simulations, including mesh generation, simulation, and data analyzing.

A Cartesian-grid-based solver can be a powerful tool for automated CFD simulations since it can almost eliminate human intervention in meshing and it has more flexibility to deal with moving and deforming objects. There are two major challenges of utilizing a Cartesian grid for wall-bounded CFD simulations, namely, (a) how to resolve wall boundaries accurately with/without turbulence modeling and (b) how to perform automated adaptive mesh refinement (AMR), including initial meshing and mesh adaptation for CFD solvers. To address the first issue, the immersed boundary method [2] has been arguably the most popular method for low-to-medium Reynolds numbers. The immersed boundary method can be generally divided into two broad categories, (a) the continuous

* Corresponding author.
E-mail addresses: bx58858@umbc.edu (L. Wang), fdw@tamu.edu (F. Witherden), antony.jameson@tamu.edu (A. Jameson).
[1] Independent researcher.

https://doi.org/10.1016/j.jcp.2024.112823
Received 11 June 2023; Received in revised form 23 November 2023; Accepted 30 January 2024

Available online 5 February 2024

forcing approach [3] and (b) the discrete forcing approach [4]. The continuous forcing IBM would only be first order accurate in near wall region. In contrast, the discrete forcing IBM, often with ghost-cell techniques, were reported to achieve high-order accuracy for finite difference methods [5]. The cut-cell approach [6,7] can accurately resolve the wall as the voxel meshes will be cut into in-solid and in-fluid parts by the wall boundaries. In the context of discontinuous Galerkin methods, one can divide those cut-cell polygons into elements, such as tetrahedral elements, whose quadrature rules are easy to find. To overcome the CFL limit of explicit time stepping, extrapolating the polynomial basis functions from intact elements to small corners [8] can be used. Alternatively, the discontinuous Galerkin difference discretization [9] can be formulated such that the stiffness of the system will not severely deteriorate at the cost of compactness. The extended finite element method [10] was originally developed to handle crack propagation in solid mechanics by Moës et al. Recently, Kummer et al. extended the idea to discontinuous Galerking methods to deal with wall boundaries [11] with enriched basis functions for the cut cells and agglomeration was needed to circumvent the CFL restriction for small cells. A Cartesian grid can also serve as the background grid in the overset method [12,13] to handle high Reynolds number flows with near-wall body-fitted meshes and wall models [14]. However, more efforts will then be needed to automate the mesh generation for the boundary layer, such as extruding the surface mesh in the wall-normal direction, and it will be more challenging when the body starts to deform. Regardless of which type of methods is used to handle wall geometries, without AMR one would have to manually refine the mesh resolution in the near-body region and wake region. A nonconforming Cartesian mesh is normally required to reduce the element count due to topological requirements. Moreover, if one is to study the maneuver of any vehicle, the enlarged refined region to account for the motion would lead to an overwhelming total number of elements and hence a waste of computational resources.

The last decade has witnessed the revolution of CFD embarked by the development of high-order methods, such as discontinuous Galerkin methods [15] and flux reconstruction methods [16], which are better suited for parallel computing [17] and the development of modern accelerators, such as the graphics processing unit (GPU). Discontinuous finite element type high-order methods are compact in nature and they have been found to be more efficient in scale-resolving simulations than their low-order counterparts [18]. The compactness of these methods makes them great candidates to work with AMR methodologies for Cartesian grids in the sense that fewer number of elements will be needed to achieve the same resolution compared to other non-compact high-order methods such as the finite volume methods and the cost of adaptation will hence be lower. In addition, the complexity of building up the high-order stencil on a non-uniform mesh for high-order finite difference methods will require more efforts on searching and bookkeeping of neighboring elements. GPU-based solvers have become more and more popular due to the tremendous speedup they offer compared to traditional CPU-based solvers. In foreseeable future, utilizing the computational power of GPU cards would be more affordable compared to CPUs for massive problem sizes. There are developments done regarding GPU-based adaptive solvers using octrees. Many of them relied on the CPU hardware to perform the mesh adaptation and information would be sent to GPU from CPU [19]. This streamline will magnify the bottleneck of data transfer between CPU and GPU. Hence, we would strive to develop efficient high-order CFD solvers with AMR that entirely operate on the GPU cards. We are interested in discontinuous finite element methods in particular due to the fact that elements will only need to communication through interface fluxes instead of relying on continuous values on element vertices as continuous Galerkin method does.

Tree structures are widely used in AMR [20]. Specifically, one can use a quadtree for two dimensional problems and an octree for three dimensional problems. There are two prevalent approaches, namely, top-bottom and bottom-up trees. The top-bottom approach is intuitively straightforward in the sense that one starts from the root node and gradually performs refinement until satisfaction and its implementation is commonly based on pointers. However, a plain pointer-based implementation does not have great data locality. A linear tree (quadtree/octree) is ubiquitous in that only a linear array is used to store the leaf nodes of the tree. And each leaf node is usually represented by a unique ID, such as the Morton ID, which encodes all the information, namely, coordinates and depth, of the represented leaf node. The bottom-up tree construction normally starts with spatial distribution of points that we are interested in, such as a human skull surface mesh we would like to render or a surface mesh of the airplane we would like to perform CFD studies. These points are immediately encoded as scattered octants. One needs to perform tree completion such that the resulting octants will continuously fill the domain with no holes and overlaps and all initial points are encapsulated in the finest octants or leaf nodes. In computational graphics, researchers have so far developed GPU-based octree data structures which can perform the bottom-up tree completion. However, when used in CFD, a tree needs to be 2:1 balanced. 2:1 balance refers to the fact that the depth difference of two adjacent octants cannot exceed 2 and it is needed for an easy enforcement of conservation. A meshless method could be an exception. One example is that in [21], Jambunathan and Levin developed a meshless Monte-Carlo method using an unbalanced octree on GPUs, where the initial tree was built on CPU first. In the work of Pavlukhin and Menshov [22], an entirely GPU-based tree was implemented. And information of three levels of tree nodes was stored, two of which are interior nodes. Not enough information was provided on 2:1 balancing of the tree. The approach is overall different from the classic linear tree. The challenge of 2:1 balancing is to efficiently find the neighbors of any given octant in the tree. Sundar et al. [23] proposed to use searching keys to perform binary search in the sorted tree for efficient balancing and the local balancing on CPU hardware will rely on hash tables to reduce the duplicates of octants for optimal performance. On GPU hardware with massive parallelism, a thread-safe hash table with no locks is not readily available. In this paper, we are going to articulate how one can performance efficient linear tree completion and tree balancing for GPU hardware without any hash tables.

We argue that a tree of octants is not functionally equivalent to a mesh of elements for CFD simulations. Finding connectivity among different entities in the mesh, such as elements, faces, points, is as important as the tree manipulation algorithms for finite element methods. On massively parallel hardware, the absence of efficient thread-safe hash tables would make the mesh generation challenging, which normally requires hash tables to find unique points, edges, faces, and the connectivity of them. We will explain

in this paper what makes discontinuous finite element methods great candidates to work with linear trees for efficient AMR on GPU hardware.

Mesh adaptation (h-adaptation) and polynomial adaptation (p-adaptation) methods are widely used for high-order discontinuous methods. Polynomial adaptation is commonly praised for its ease of implementation compared to mesh adaptation and the great speedup that it can offer [24]. However, things will be quite different for GPU computations. For massive parallelism with graphics cards, minimization of thread divergence is warranted for performance. Generally, there are two major types of calculation for any high-order discontinuous finite element methods, namely, element-related and interface-related calculations. For mesh adaptation with linear trees, regardless of the depth of any octant in the tree, element-related calculations will be identical for all voxel elements (quadrilateral elements in 2D and hexahedral elements in 3D) including interpolations to the interfaces. Only conforming and nonconforming interior interfaces and boundary faces will need different procedures of common flux calculations and they are resolution-indifferent. For polynomial adaptation, the numerical discretization operators will be completely polynomial-degree dependent. Moreover, optimization of the data layout and shared memory utilization, could be very different and special treatment will be needed for high polynomial degrees since the shared memory would not be big enough to fit the entire numerical operator, such as the projection matrices for data transfer in adaptation. The more polynomial degrees are used, the more significant the divergence would be. Therefore, mesh adaptation appears to be more attractive for GPU hardware for optimal performance.

Contributions. In this paper, we report a GPU-based h-adaptive flux reconstruction method with linear trees. This is the first paper that elaborates the novel algorithms for tree completion and tree balancing and how to efficiently use linear tree for discontinuous finite element methods on GPU hardware. We provide enough algorithm/coding details for the readers. The developed method has shown significant speedup for long distance vortex propagation. For 3D simulations, the adaptation cost including tree manipulations, face query and data transfer, can be less than 2% of the overall computational cost when we perform adaptation every 10-40 times with explicit time integrators for both inviscid flows and transitional flows. This work paves the way for further development of automated Cartesian-grid-based CFD solver which can utilizing massively parallel hardware.

Organization. The remainder of this paper is organized as follows. In Section 2, we introduce the flux reconstruction method and linear trees briefly. Section 3 documents in detail how to construct and balance a linear tree entirely on GPU hardware. And we also illuminate how to build the face connectivity for the flux reconstruction method. In Section 4, we articulate how to perform adaptation and how data should be transferred in the adaptation procedure. We will present the numerical experiments in Section 5. Section 6 summarizes this work.

2. Background

2.1. The flux reconstruction method

The flux reconstruction method was originally proposed by Huynh [16] for tensor product elements. Researchers have extensively studied and enriched this family of high-order methods [25–28] since then. We remark that for high-order collocation methods, the nested inner resolution has a great potential of making the right hand side evaluation dominate the total computational cost such that the cost of mesh adaptation can be significantly smaller compared to low order methods. This has previously been observed with a p-adaptive flux reconstruction method on unstructured grids [24] where as the polynomial degree increases, the adaptation overhead gets smaller.

Considering the following 3D conservation law (2D problems are treated similar)

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} + \frac{\partial g}{\partial y} + \frac{\partial h}{\partial z} = 0 \tag{1}$$

on a Cartesian grid, we can calculate the spatial derivatives dimension by dimension as illuminated by Huynh [16]. In this work, local coordinate ξ, η, ζ are aligned with x, y, z, respectively. Therefore, taking f as an example, we first construct the corrected flux polynomial $\hat{f}$ as

$$\hat{f} = f(\xi) + g_L(\xi)(\hat{f}(-1) - f(-1)) + g_R(\xi)(\hat{f}(1) - f(1)), \tag{2}$$

in the local coordinate system, where the local flux polynomial $f(\xi)$ is represented using a Lagrangian polynomial as

$$f(\xi) = \sum_{i=1}^{k+1} l_i(\xi) f_i, \tag{3}$$

and k is the polynomial degree of $l_i(\xi)$. $\hat{f}(\pm 1)$ are numerical fluxes at the interfaces, which are referred to as the common fluxes. And a Riemann solver is normally used for their calculations. g_L and g_R are left and right correction functions, which are symmetric with respect to $\xi = 0$. The derivative of the corrected flux polynomial can be expressed as

$$\hat{f}' = \sum_{i=1}^{k+1} l_i'(\xi) f_i + g_L'(\xi)(\hat{f}(-1) - f(-1)) + g_R'(\xi)(\hat{f}(1) - f(1)). \tag{4}$$

In terms of implementation, these differentiation operators are pre-computed. In our GPU implementation, we store the local differentiation operator of f as well as $g_L'(\xi)$ in the constant memory. Radau polynomials are used in this work [16] to recover the discontinuous Galerkin method. The derivative with respect to x can be computed as

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial f}{\partial \eta}\frac{\partial \eta}{\partial x} + \frac{\partial f}{\partial \zeta}\frac{\partial \zeta}{\partial x}. \tag{5}$$

Since ξ, η, ζ are aligned with x, y, and z respectively, $\frac{\partial f}{\partial \eta}\frac{\partial \eta}{\partial x} = \frac{\partial f}{\partial \zeta}\frac{\partial \zeta}{\partial x} = 0$ and the above equation is simplified as

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial \xi}\frac{\partial \xi}{\partial x}. \tag{6}$$

The local discontinuous Galerkin method [29] is utilized for viscous problems and we refer readers to [17] for implementation details for unstructured meshes whilst a 1D approach is also used for viscous terms in this work. Even though the mesh in AMR is non-uniform, Δx of every element is not explicitly stored since it can be inferred from the encoded information of each element, which will be explained in the following content. In our current study, a three stage and third order explicit Runge–Kutta method is used for time integration [30]. From time step t^n to time step t^{n+1}, the solution update is organized as

$$\begin{cases} u^1 = u^n + \Delta t R(u^n), \\ u^2 = \frac{3}{4}u^n + \frac{1}{4}u^1 + \frac{1}{4}\Delta t R(u^1), \\ u^{n+1} = \frac{1}{3}u^n + \frac{2}{3}u^2 + \frac{2}{3}\Delta t R(u^2), \end{cases} \tag{7}$$

where $R = -\frac{\partial f}{\partial x} - \frac{\partial g}{\partial y} - \frac{\partial h}{\partial z}$, which is referred to as the right hand side. Efficient octree algorithms would make building up a h-multigrid framework for implicit time integration straightforward on the GPU hardware. We would like to explore whether previously-developed implicit methods [31,32] can be accelerated with a h-multigrid solver or preconditioner for compressible flows, given that the p-multigrid methods in [32] do not work well on GPU for compressible flows when Ma $>= 0.1$. Throughout this paper, we are going to use "dim" to indicate the dimension of the problem.

2.2. Linear trees and Morton IDs

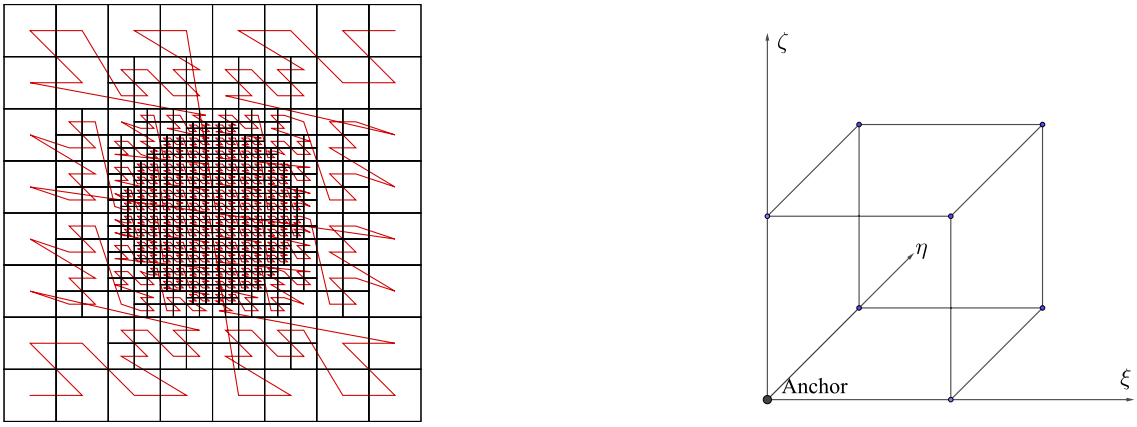
In tree structures, such as the binary search tree, a leaf node is a node in the tree that does not have any child nodes. A linear tree (quadtree for 2D and octree for 3D) refers to the fact that only the leaf nodes of the tree are stored as a sorted array of integers where each integer represents a leaf node in the tree. A node is usually referred to as an octant in an octree. And we use an octant to represent a node in a quadtree for consistency. In this paper, we are concerned with using Morton encoding to get the Morton ID of each node of the tree. The sorted array of Morton IDs is one type of space filling curves (SFC) and when it is visualized, we can observe curves of zigzag shapes filling the entire domain, as shown in Fig. 1a, and they are usually called Z-shape SFC.

Morton encoding can encrypt both the coordinates and the depth of an octant in the tree. With predefined spatial resolution, one would take the coordinates of one corner of the octant and convert them into an integer. The encoded corner is referred to as the anchor of the octant [23]. As shown in Fig. 1b, in this paper we use the left-front-bottom corner as the anchor. In Morton encoding, the inclusive region of the octant O_j at depth D_j is defined as $[\mathbf{x}_j, \mathbf{x}_j + \Delta x_j)$, where $\mathbf{x}_j$ is the coordinate of the anchor and Δx_j is the width of the octant. In this paper, the widths of any octant in the tree in all directions are the same if not specifically mentioned. As the size of the octants are sorted out at different depths, one can always convert the coordinates of the mesh from floats to integers. To distinguish, we use $[\mathbf{x}_j^f, \mathbf{x}_j^f + \Delta^f x_j)$ to denote the region of an octant in the actual physical domain, and we use $[\mathbf{x}_j^I, \mathbf{x}_j^I + \Delta^I x_j)$ for the normalized integer representation of the octant domain. For any octant O_j at depth D_j, we can perform the Morton encoding by interleaving the bits of different components of $\mathbf{x}_j^I$ first and then append the level information. If we define the maximum depth allowed in the octree as D_{max}, the overall algorithm is described in Algorithm 1, following the work of Sundar el al. [23]. Note the difference between raw Morton ID $\overline{O_j}$ and Morton ID O_j is that the former only has the coordinate information.

Algorithm 1 Morton Encoding.

1: Determine the maximum tree depth D_{max} and minimum element size Δx_{min}^f for the application and calculate the maximum number of bits, m_b, that is needed to represent D_{max}.
2: Convert the float coordinates $\mathbf{x}_j^f$ of any octant into integer coordinates $\mathbf{x}_j^I$ based on Δx_{min}^f and determine the depth of D_j.
3: Interleave the bits of three components of the coordinate as $\boldsymbol{x}^{D_j-1} \dots \boldsymbol{x}^1 \boldsymbol{x}^0$ to get the raw Morton ID $\overline{O_j}$, where $\boldsymbol{x}^k = x_3^k x_2^k x_1^k$ for 3D and $\boldsymbol{x}^k = x_2^k x_1^k$ for 2D in this algorithm, which means retrieving the k-th bit of each component of $\mathbf{x}_j^I$ and putting them together to get a binary number of 2/3 bits for a 2/3-dimensional problem.
4: Add padding bits to the raw Morton ID $\overline{O_j}$ as $\boldsymbol{x}^{D_j-1} \dots \boldsymbol{x}^1 \boldsymbol{x}^0 \underbrace{0}_{\text{repeat } \dim\cdot(D_{max}-D_j)}$.
5: Append the binary representation of the depth $(D_j)_2$ to the result obtained in previous step as $\boldsymbol{x}^{D_j-1} \dots \boldsymbol{x}^1 \boldsymbol{x}^0 \underbrace{0}_{\text{repeat } \dim\cdot(D_{max}-D_j)} \underbrace{(D_j)_2}_{m_b\text{bits}}$ to get the Morton ID O_j which includes all the information that we need.

Once we obtain the Morton IDs of all octants in the tree, we sort the linear tree in ascending order to get $\mathcal{T}$, which is the post-order traversal of the tree. We refer readers to the work of Sundar el al. [23] for other interesting properties. Given the Morton ID of an octant, one can always decode it to retrieve the normalized integer coordinates as well as the width of the octant. Therefore, there is no need to save the non-uniform element size information explicitly. There are various ways of efficient implementation of interleaving the integer bits (Step 3 in Algorithm 1). We recommend the open-source software `libmorton` [33] for a collection of different approaches. In our GPU implementation, we utilize the magic number approach in [33].



(a) Z-shape filling curve (b) Anchor of an octant

Fig. 1. Illustration of Z-curve and anchor.

3. GPU-based tree algorithms for discontinuous methods

Without some efficient thread-safe data structures, especially hash tables, on GPU, we have to redesign the algorithms for tree manipulations. When we combine linear trees with adaptive CFD methods, efficient algorithms of building up the mesh connectivity will be equally important as tree completion and balancing algorithms. We are going to articulate the algorithms that we design for efficient tree manipulations and connectivity query. Some common algorithms such as sorting, binary search, etc., are from the open source library THRUST from NVIDIA [34]. Detailed coding techniques will be explained if it is important for readers to understand how the algorithms work.

3.1. Tree construction

A set of octants is said to be complete if the union of these octants covers the entire domain. Alternatively, one can also define a complete linear tree as a linear tree in which every interior node has exactly eight child nodes in 3D and four child nodes in 2D [23]. The algorithm that we use for tree completion follows the idea developed by Sundar et al. [23] with novel optimizations for the GPU hardware. An efficient GPU implementation will need to make use of the shared memory within one block of threads

Algorithm 2 Tree construction on GPU hardware using shared memory and atomic counters.

1: Encoding the scattered points $\mathbf{x}_j$ into Morton ID O_j. And sort the array in ascending order to get the initial incomplete tree $\mathcal{T}_{init}$.
2: For two consecutive octants O_j and O_{j+1} of $\mathcal{T}_{init}$, find the largest common ancestor (LCA).
- Split the LCA to get 2^{dim} child octants C_i, where $i = 1, \ldots, 2^{\text{dim}}$.
- Update the parent octants of O_j and O_{j+1} from C_i;
- For the rest octants in C_i, store any $O_j < C_i < O_{j+1}$ in the shared memory for new block-local array of octants $\mathcal{T}_{loc}$.
- Repeat the procedure separately for parents of O_j and O_{j+1}, until O_j and O_{j+1} can be found in the new array of C_i.

3: Sort the block-local array $\mathcal{T}_{loc}$ and remove the duplicates within the kernel function. In the meantime, use block-local atomic counter to record the number of new octants in each block of threads.
4: Update global atomic counter with the local one and write the block-local array to tail of the global array $\mathcal{T}_{init}$.
5: Sort the tree in ascending order using `thrust::sort`.
6: Linearize the tree to get the complete tree $\mathcal{T}$.

to store the newly generated octants and then write the results into global memory of the GPU card. Two atomic counters are used in our implementation. The first one is a block-local counter to record locally how many new octants are created and the second one is a global counter to record and synchronize among blocks for all the new octants. One caveat is that since the shared memory is used to store the newly generated Morton IDs, the total amount could exceed the full capacity of the shared memory under rare circumstances. Out of shared memory in tree algorithms is not as intimidating as it sounds. It just means the sparsity of the initial octants cause excessive octants generation within one GPU block. We can always repeat the procedure when out of shared memory happens and note that for next round, the problem will be greatly alleviated since we have already injected a significant amount new octants into the tree. We need to linearize the tree, which refers to removing the duplicates and overlaps in the tree.

Linearized tree. A linearized tree is a non-overlapping and complete tree. We propose the following simple algorithm for tree linearization as documented in Algorithm 3. The designed algorithm fits data parallel paradigm very well by reversing the tree and utilizing the `thrust::unique` algorithm in THRUST.

Algorithm 3 Tree linearization using THRUST.

1: Reverse the tree (change the tree from ascending order to descending order).
2: Perform the algorithm `thrust::unique` in the thrust library with a binary predicate which tells if two consecutive Morton IDs are overlapping.
3: Reverse the remaining tree back.

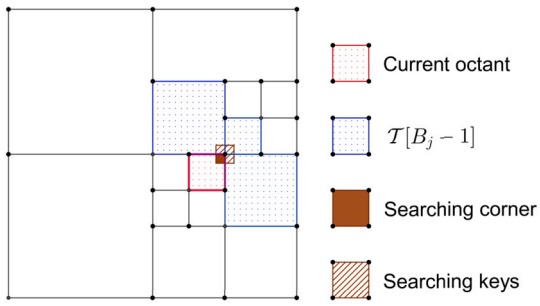


Fig. 2. Illustration of the searching keys for neighbors of an octant.

3.2. 2:1 balance of the linear tree

The linearized complete tree from Algorithm 2 is not a 2:1 balanced tree. 2:1 balance refers to the depth difference of any two adjacent octants must not exceed 2. In this paper, we pursue strong 2:1 balance, namely, any two adjacent octants that share either an edge or a face must not have tree-depth difference larger than 1. Similar to polynomial balancing in polynomial adaptation [24], the canonical method of balancing the tree is to start from the finest resolution to ensure the 2:1 rule is enforced and then repeat the same process for coarser resolution one by one. The most challenge part of balancing a linear tree is to find the neighbors of any given octant in the tree. With a linear sorted tree represented by Morton IDs, finding a neighbor of an octant is equivalent to finding the lower bound of the corresponding Morton ID in the sorted array.

Lower bound. For any given Morton ID O_j, we define the lower bound location of O_j in the linear tree $\mathcal{T}$ as the location $\mathcal{B}_j$ where $\mathcal{T}[\mathcal{B}_j] \geq O_j$. We use notation $a[i]$ to indicate i-th entry in array a in this manuscript. The algorithm `thrust::lower_bound` in THRUST is used since it can efficiently find lower bounds of an input array in another array. Based on the properties of Morton IDs, we have three significant scenarios, which will be used later, listed as follows:

$$\begin{cases} \mathcal{T}[\mathcal{B}_j] = O_j, \text{ i.e., lower bound of } O_j \text{ in the tree is } O_j \text{ itself;} \\ \mathcal{T}[\mathcal{B}_j - 1] \ni O_j, \text{ i.e., the octant on the left of the lower bound of } O_j \text{ is the parent of } O_j; \\ \mathcal{T}[\mathcal{B}_j] - O_j = 1 \text{ and } \mathcal{T}[\mathcal{B}_j - 1] \in O_j, \text{i.e., the lower bound is the first child of octant } O_j. \end{cases} \tag{8}$$

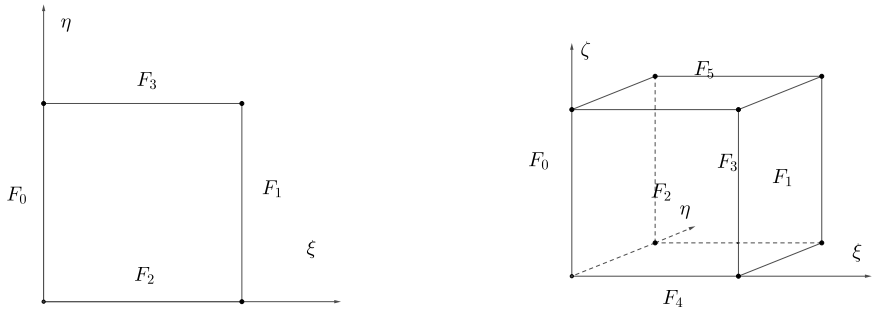
Note that in this paper, we use $A \in B$ to denote that A is a child of B and $B \ni A$ to denote B is a parent of A. We adopt the concept of searching corners used by Sundar et al. [23]. A searching corner is the octant on the finest resolution that shares a vertex with both the current octant and the parent octant. The searching keys are octants that make the shared vertex as the center of a cube/square formed by the searching corner and keys. We remark that a searching key does not have to be in the tree and one can always find either the searching key itself or an ancestor of the searching key as long as the tree is complete and linearized. The schematic diagram of a searching corner and corresponding searching keys are presented in Fig. 2 and we also visually show what is a lower bound for a given searching key.

The algorithm that we develop to balance a tree at depth D is presented in Algorithm 4. The optimization here for multi-threading, compared to the work of Sundar et al. [23], is that we collect all the searching keys together and perform data parallel computing. In the course of balancing a tree, the searching keys and local refinement of target octants will end up with lots of duplicated child

Algorithm 4 Balancing tree at depth D on GPU.

1: Collect all the Morton IDs that are on depth D in the tree $\mathcal{T}$ and store them as $\mathcal{R}_D$.
2: Find all the searching keys of all the Morton IDs in $\mathcal{R}_D$ and store the unique values as $\mathcal{S}_D$.
3: For every searching key in $\mathcal{S}_D$, find the location of its lower bound in the tree $\mathcal{T}$ and store the results in $\mathcal{B}_D$.
4: For each searching corner $\mathcal{S}_j$ in $\mathcal{S}_D$, if $\mathcal{T}[\mathcal{B}_{D,j} - 1]$ is a parent of $\mathcal{S}_j$, refine $\mathcal{T}[\mathcal{B}_{D,j-1}]$ and store any octant which is not a parent of $\mathcal{S}_j$ in the shared memory. Repeat the process for the new parent octant of $\mathcal{S}_j$ until the depth of these new octants reaches $D + 1$.
5: Perform a block-lock removal of duplicates and use a block-local atomic counter to record the total unique Morton IDs per block.
6: Use the block-local counter to update the global atomic counter of the total number of new octants and in the meantime write new octants to the tail of tree $\mathcal{T}$.
7: Sort and linearize the updated tree $\mathcal{T}$.

octants. One could utilize a hash table to quickly remove duplicates on the fly as Step 4 of Algorithm 4 generates new octants if the algorithm were implemented for CPU hardware. On GPU hardware, we use the same approach as in Algorithm 2 for duplicates removal. Note that as long as the tree is completed, we do not have to perform tree completion anymore in the 2:1 balancing algorithm. One additional note is that in numerical experiments, out of shared memory is more likely to happen in Algorithm 4



(a) Faces in a 2D element (b) Faces in a 3D element

Fig. 3. Face numbering for 2D and 3D elements.

instead of Algorithm 2 in the phase of building an initial mesh. In CFD simulations, since the mesh before adaptation is already completed and balanced, it would not be an issue.

3.3. Interface data structure and connectivity

A linear tree of octants is not equivalent to a mesh of elements. For methods like continuous Galerkin methods, nodal information such as how a node is connected to elements is needed. Finding such information is trivial on CPU hardware with hash tables but challenging if done entirely on GPU. Fortunately, for discontinuous method, such as the flux reconstruction method used in the work, we only need to know how elements communicate through faces.

Data structures. For a tree of length M, we define an array $\mathcal{F}$ of length $n_f \cdot M$, where $n_f = 2 \cdot \dim$ is the number of faces that an octant has and dim is the dimension of the problem. Following C++ convection of array indexing, the entries in $[n_f \cdot \iota_e, n_f \cdot (\iota_e + 1))$ store the indices of the neighboring elements of the current element ι_e. For any octant, we define the face indices as illustrated in Fig. 3.

Face matching. With the linear data structure defined for face matching, the remaining question is to find the neighbors. Similarly, we need to identify the searching keys first. The nuance here is that we define the searching keys on the same depth as the current octant. There are three types of faces in the nonuniform mesh represented by a linear tree, namely, conforming faces, nonconforming faces and single-sided faces. Both conforming and nonconforming faces are interior faces and single-sided faces will only appear on the boundary. The algorithm to find the matching pair for the faces that an octant have is organized in Algorithm 5.

Algorithm 5 Build face connectivity.

1: Find the search keys of any octant O_j in tree $\mathcal{T}$ by calculating the Morton IDs of all its neighbors which share faces with it as shown in Fig. 4 and store in S.
2: Find lower bounds in the current tree $\mathcal{T}$ for all keys and store in B.
- If $\mathcal{T}[B_j] = S_j$, then the neighbor of O_j, namely, S_j is in the tree and the corresponding face of S_j is the matching conforming face.
- If $\mathcal{T}[B_j - 1] \ni S_j$, then the neighbor of O_j is the parent of S_j. The corresponding face of $\mathcal{T}[B_j - 1]$ is the coarse face of a nonconforming pair.

When we build up the searching keys S in Algorithm 5, these keys do not have to be in the tree, similar to the searching keys for 2:1 balance. In Step 2 of Algorithm 5, the spatial relationship of two sides of one conforming pair is straightforward. For the i_f-th face of O_j, its matching conforming face will be the $(i_f + 1)$-th face of the neighboring element if i_f is an even number or $i_f - 1$ if i_f is an odd number, following the numbering convention shown in Fig. 3. Note that raw Morton ID $\overline{O_j}$ is obtained by removing the padding zero bits and depth bits of O_j. Taking Face 0 of $\overline{O}_j$ and $\overline{O}_j + 2$ in Fig. 5 as an example, they both have Face 1 of $O_{c,0}$ as the matching face. When we need to project the working variables from the fine faces (Face 0 of $\overline{O}_j$ and $\overline{O}_j + 2$) to the coarse face (Face 1 of $O_{c,1}$), we can retrieve the last two bits of $\overline{O}_j$ and $\overline{O}_j + 2$ to get 0 and 2 in decimal representation. As we define the first contributor along one axis will be the one that has smaller coordinates, we will know $\overline{O}_j$ is the first contributor and $\overline{O}_j + 2$ is the second in the projection procedure of common flux calculations. Similarly, for Face 2 of $\overline{O}_j$ and $\overline{O}_j + 1$, the last two bits of these two numbers will be 0 and 1 and $\overline{O}_j$ is first contributor for the nonconforming face formed with coarser octant $O_{c,1}$. We remark that for any $2^{\dim - 1}$ siblings that form a nonconforming face with another coarser element, we do not care if the other $2^{\dim - 1} - 1$ siblings are interior nodes or leaf nodes in the tree.

We explain the reason why we do not try to find the neighbors of any element that shares a face with $2^{\dim - 1}$ finer elements. Considering the example presented in Fig. 6, for $O_{c,0}$, one needs to find both $\overline{O}_k$ or $\overline{O_j}$ because their Morton IDs cannot be inferred from each other due to the presence of refined sibling octant on the right side of $\overline{O_j}$. Hence, more searching and bookkeeping will be needed. More information regarding how the face connectivity information will be used in discontinuous Galerkin methods will be illuminated in next section. It is noteworthy that what is needed for discontinuous Galekin methods can simply be obtained by

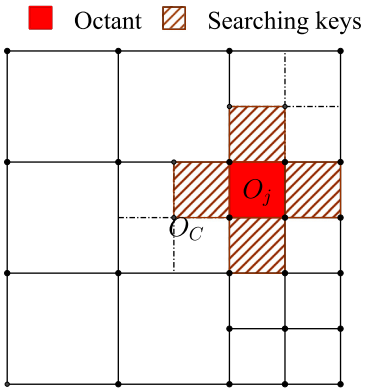


Fig. 4. Illustration of the searching keys of an octant for interface matching.

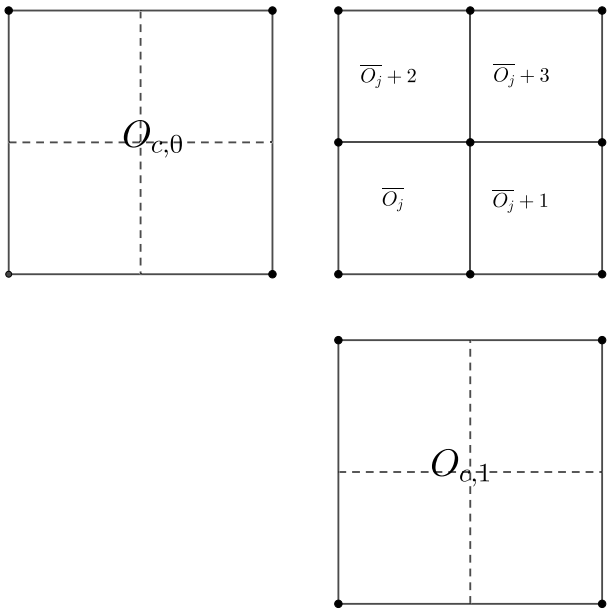


Fig. 5. Illustration of face matching in quadtree, where all siblings of $\overline{O}_j$ are in the tree.

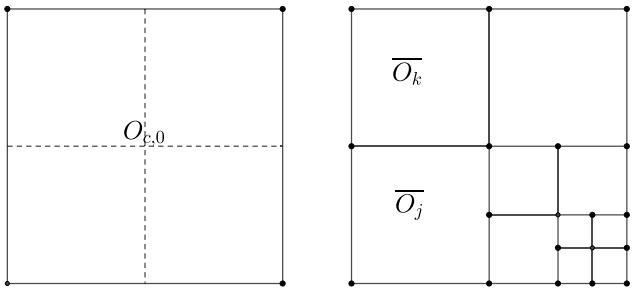


Fig. 6. Illustration of face matching in quadtree, where $\overline{O}_k$ and $\overline{O}_j$ cannot be inferred from each other.

performing series of binary searches for the searching keys. The methods are indifferent to nodal connectivity, which would normally require data structures like hash tables to efficiently obtain. The attributes that discontinuous methods have make them better suited for AMR on GPU than continuous Galerkin methods.

4. The *h*-adaptive flux reconstruction method

4.1. Adaptation procedure

The procedure that we use to carry out adaptation is organized as in Algorithm 6. For coarsening, one has to first rectify the marking in Step 3 of Algorithm 6 simply due to the fact that an interior node of the tree could have child nodes and grandchild nodes at the same time. Moreover, we would have to identify those interior node in Step 4 and then remove the child nodes in Step 5 while no extra action is needed for refinement. The tree linearization will automatically remove any octant which is refined in the array. Our algorithm of adaptation procedure also implies that we can at most refine or coarsening an octant once in one adaptation procedure. Step 4 relies on the properties of the siblings illustrated in Fig. 5, where in the sorted array 2^{dim} siblings are consecutively stored with increment of the raw Morton ID equals to 1.

Adaptation criteria for CFD simulations of unsteady problems have been widely studied in literature. There are featured-based [24], discretization-error based, truncation-error based, and adjoint-based based [35] methodologies, etc. Discussion or evaluation of these different methodologies will be beyond the scope of this work and we recommend the work of Naddei et al. [36] and Ims and Wang [37]. When AMR is coupled with the flux reconstruction method, we would like to, in this work, focus on the performance of tree balancing, connectivity query, data transfer, compared to that of the flux reconstruction solver. Therefore, we are going to use feature-based approaches for the problems studied in the numerical experiment studies.

Algorithm 6 Adaptation procedure of the tree.

1: Evaluate the adaptation flags for each element in the mesh. Elements will be marked with flag -1 and 1 for coarsening and refinement, respectively. Flag 0 means the element will remain the same.
2: Refinement of any octant will be honored via directly splitting the octant and append the child octants to the tail of tree $\mathcal{T}$. Similar techniques of using block-local and global atomic counters are used as those used in 2:1 balance.
3: Rectify coarsening flags by reverting flag -1 to 0 if any octant does not have all its siblings existing in $\mathcal{T}$.
4: If all child octants of an interior node are marked for coarsening, collect the parent Morton ID in $\mathcal{P}$.
5: Remove those Morton IDs in $\mathcal{T}$ which can find its parent in $\mathcal{P}$ and append $\mathcal{P}$ to $\mathcal{T}$ afterwards.
6: Sort and linearize $\mathcal{T}$.

4.2. Solution data transfer

Data transfer between the new tree $\mathcal{T}_n$ and old tree $\mathcal{T}_o$ on GPU relies on the mapping between the two trees. There are three types of data transfer in an adaptation procedure.

Direct copy. Octants in $\mathcal{T}_n$ and $\mathcal{T}_o$ that have the same Morton IDs are elements which are kept as the same in one adaptation procedure. A direct copy of the solution variables is sufficient.

Interpolation. When an octant is split into 2^{dim} children, the working variables in these children can be interpolated from the parent octant. Note that the interpolation coefficients can be directly evaluated within the kernel function through tensor products of 1D Lagrange interpolations. We opt to use interpolation instead of projection for refinement to reduce the copy of a dense projection matrix into the shared memory here.

Projection. When 2^{dim} siblings are merged into one parent octant, a L_2 projection is used to transfer the data from child octants to the parent octant. Given K_ι, where $\iota = 1, 2, \ldots, 2^{\text{dim}}$, are the child octants which are fused into the parent octant P. The discrete formulation to project the state variables $\boldsymbol{u}_K^\iota$ of children octants to $\boldsymbol{u}_P$ of the parent octant can be formulated as

$$\boldsymbol{u}_P = \boldsymbol{M}^{-1} \sum_{\iota=1}^{2^{\text{dim}}} \boldsymbol{S}^\iota \boldsymbol{u}_K^\iota. \tag{9}$$

$\boldsymbol{M}^{-1}\boldsymbol{S}^\iota$ is the projection matrix accounting for the contribution of the ι-th child. We refer readers to [38] for detailed derivations. These matrices are precomputed and stored in the global memory of the GPU hardware. In the kernel function that deals with volumetric projection, each thread will process one solution point of the parent octant. And the projection is done in order through $\iota = 1$ to $\iota = 2^{\text{dim}}$ with corresponding projection matrix $\boldsymbol{M}^{-1}\boldsymbol{S}^\iota$ and state variables $\boldsymbol{u}_K^\iota$ loaded into the shared memory each time. Note that raw Morton IDs of these child octants are consecutive such that for the mapping between the old and new trees, one only need to find the first contributor and the rest can be inferred.

At this point, readers should be familiar with the usage of lower bounds and would not be surprised that we are going to use a similar approach to find the mapping between old tree $\mathcal{T}_o$ and new tree $\mathcal{T}_n$. In Algorithm 7, we present the algorithm to identify the mapping. The Step 5 of Algorithm 7 is needed because when we use atomic counters to write information into the data arrays on the global memory, there is no rule that regulates which thread writes first. By sorting the IDs, we can coalesce the memory access as much as possible in the data transfer procedure. As stated in Step 4 of Algorithm 7, the Morton ID of these child octants are consecutive.

Algorithm 7 Mapping between old and new tree.

1: Find the lower bounds for all Morton IDs O_j of the new tree $\mathcal{T}_n$ in the old tree $\mathcal{T}_o$ and store in B.
2: If $\mathcal{T}_o[B_j] = O_j$, then this is a direct-copy octant.
3: If $\mathcal{T}_o[B_j - 1]$ is a parent of O_j, then the $\mathcal{T}_o[B_j - 1]$ is refined to get O_j.
4: If $\mathcal{T}_o[B_j] - O_j = 1$, then this octant $\mathcal{T}_o[B_j]$ together with following $2^{\dim} - 1$ octants in the old tree are fused to get O_j.
5: Store corresponding B_j and O_j for all different scenarios. After done, sort the B_j and O_j arrays based on the value of O_j.

4.3. Common flux calculations on interfaces

In Section 3.3, we present the data structures and algorithms to identify the face connectivity information that is needed for the flux reconstruction method. In this section, we are going to explain how to perform the common flux calculations for inter-element communication in the flux reconstruction method. To avoid thread divergence, we will group the common flux calculations into three different types, namely, (1) conforming interior face pairs, (b) nonconforming interior face pairs, and (c) exterior boundary faces. For periodic boundary faces, they eventually become interior faces. The special treatment is that when we try to identify the searching keys in the 2:1 balancing algorithm or the face connectivity query algorithm we need to take into account the periodicity of integer coordinates. At a pair of conforming faces, one would only need to use the working variable from both sides $\boldsymbol{u}_l$ and $\boldsymbol{u}_r$ to calculate the common flux. At nonconforming interfaces, we use the approach proposed by Kopriva [38]. The procedure to perform common flux calculations for mortar faces is described in Algorithm 8.

Algorithm 8 Procedure of common flux calculations.

1: Interpolate the solution from interior solution points to local flux points.
2: Project the local solution on the flux points to the mortar face.
3: Compute the common fluxes using a Riemann solver on the mortar face.
4: Project the common fluxes back to each side of the local elements.

L_2 projections are used in the process to ensure conservation of the numerical methods. Taking the illustration in Fig. 7 as an example, the projection between the local faces $\Omega_f^{1,2}$ and mortar faces $\Xi_m^{1,2}$ is straightforward. Since they are conforming and the polynomial degrees are the same, a direct copy is all we need. The projection from $\boldsymbol{u}_{\Omega_c}$ on the local face Ω_C to $\hat{\boldsymbol{u}}_{\Xi_m^\iota}$ on mortar faces Ξ_m^ι, $\iota = 1, 2, \ldots, 2^{\dim-1}$ can be achieved as

$$\hat{\boldsymbol{u}}_{\Xi_m^\iota} = \boldsymbol{M}_{\Xi_m^\iota}^{-1} \boldsymbol{S}_{\Xi_m^\iota} \boldsymbol{u}_{\Omega_c}, \tag{10}$$

where $\boldsymbol{M}_{\Xi_m^\iota}^{-1} \boldsymbol{S}_{\Xi_m^\iota}$ is the projection matrix [38]. After the conserved variables are ready on the mortar faces, we use a Rusanov solver for the common inviscid flux calculations in this work. The projection of common fluxes from the mortar faces to the coarse face is a bit more complicated since $2^{\dim-1}$ mortar faces will contribute to the same coarse face. The projection reads

$$\boldsymbol{F}_{\Omega_c} = \boldsymbol{M}_{\Omega_c}^{-1} \sum_{\iota=1}^{2^{\dim}-1} \boldsymbol{S}_{\Omega_c}^{\iota} \hat{\boldsymbol{F}}_{\Xi_m^\iota}, \tag{11}$$

where $\boldsymbol{M}_{\Omega_c}^{-1} \boldsymbol{S}_{\Omega_c}^{\iota}$ is the corresponding projection matrix for the ι-th face that contributes to the projection [38]. We follow the similar procedure for the LDG method to deal with viscous problems. These projection operators in Eqs. (10) and (11) are pre-computed and stored in the global memory.

We remark that there is one major difference for the projections at nonconforming interfaces, compared to the projection for volumetric data transfer in Eq. (9). As mentioned in Section 3.3, we find the neighbors of finer elements at nonconforming interfaces. And the kernel function would be organized to calculate the common fluxes on the finer sides of the nonconforming faces, which are aligned with the mortar faces. Hence, the projection in Eq. (10) can be locally done within one thread block and each thread deals with one flux points after loading the corresponding operator and state variables into shared memory. However, mortar faces that contribute to the projection summation in Eq. (11) would possibly not be in the same block of threads such that different threads ought to write to the same destination in global memory. To protect the calculations from thread racing, we need to use the `atomicAdd` algorithm. Additionally, only one projection matrix will be loaded into the shared memory for the chunk of threads in a block that deal with the current mortar face.

5. Numerical results

In this section, a brief comparison of the GPU and CPU version of the tree completion and balancing algorithms is conducted first. We will then focus on the performance of the adaptive solver in terms of both accuracy and computational cost for inviscid and viscous flows. All simulations were run on a Windows Subsystem for Linux 2 (WSL2) with the Ubuntu 22.04.2 LTS Linux distribution while the host laptop has a NVIDIA GeForce RTX 2060 GPU card. There are 1920 CUDA cores, 6 GB memory for this GPU card and the bandwidth is 264.05 GB/s. The host laptop has an Intel i-10750H CPU with 6 cores. Note that the CUDA runtime and libraries would occupy around 1.2 GB GPU memory such that only 4.8 GB were available for the simulations.

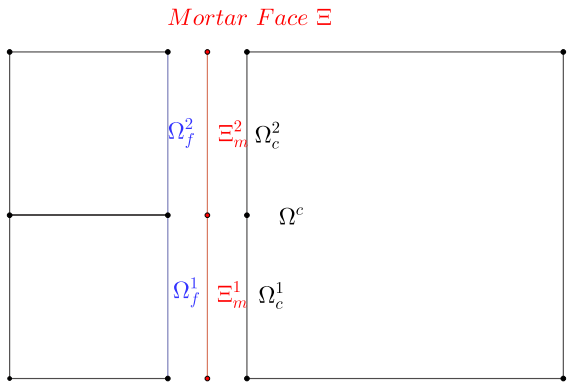
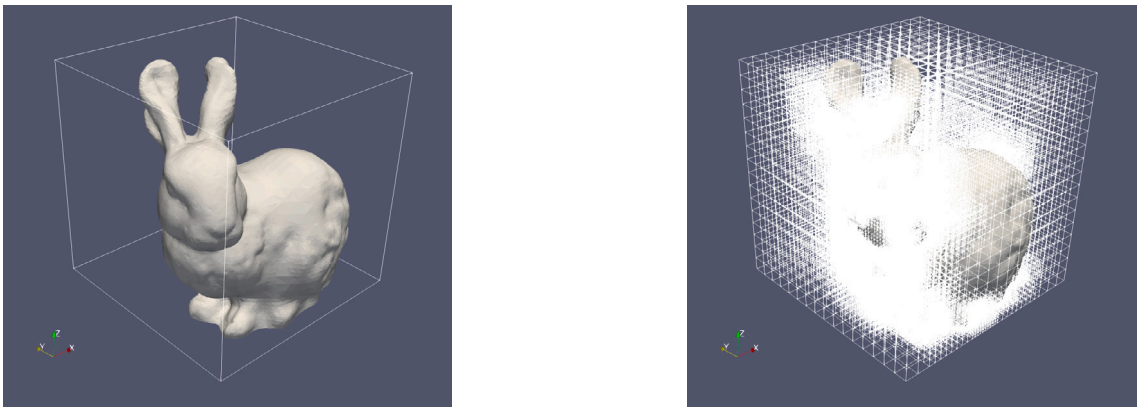


Fig. 7. Illustration of the mortar interface.



(a) (b)

Fig. 8. (a) Stanford bunny and (b) balanced tree.

5.1. Tree completion and tree balancing

In this section, we aim to test the efficiency of the tree completion and balancing algorithms for the algorithms that we developed. As a reference, we also implemented the CPU version of the code, which was made easy by the multi-platform support of THRUST and C++ template programming. The major difference is that in the CPU version we used `std::unordered_map` from the standard template library (STL) in C++, which essentially is a hash table data structure and a `std::mutex` is used to protect it from data racing. In all initial mesh generations of our numerical examples, we are going to use D_{min} to generate a coarse background mesh and D_{max} will be where the initial mesh points are encoded.

The Stanford bunny was used to test the robustness and performance of the developed tree completion and balancing algorithms. In Fig. 8, we present the geometry and the balanced tree. For this testing case, the minimum tree depth D_{min} was set as $D_{min} = 4$ for this case and the maximum tree depth $D_{max} = 9, 10$ were tested. In Fig. 9, the computational cost of tree completion and tree balancing are illustrated for both CPU version and GPU version of the code. When $D_{max} = 9$, the final element count is around 0.4 million. Overall, the computational cost of tree completion is trivial compared to that of 2:1 balancing for both CPU version and GPU version of the algorithms. Compared to parallel CPU version of the code (the CPU has 6 physical cores with hyper-threading enabled), we can get a speedup of 5.11 for 2:1 balancing using the newly developed GPU code. In average we allow each thread to create 16 new octants and the rest of the shared memory was used for sorting and removing duplicates within the block of threads. When $D_{max} = 10$, the newly generated octants in tree balancing Algorithm 4 would exceed the allocated cap. To resolve this issues, we performed the tree balance algorithm again and a speedup of 3.72 was achieved while the final element count is around 0.9 million. Even though, we have to repeat Algorithm 4, the speedup of the GPU code compared to the CPU version is still significant. Note that as long as we start CFD simulations with a complete and balanced tree, out of shared memory for refinement would not happen because we only allow any octant to be refined once per adaptation.

The speedup of GPU-based tree construction and balancing through this test is promising. Admittedly, until this point, we are still steps away from automated initial mesh generation for CFD simulations. One would need to refine the octants near the surface. Another data structure, namely, the GPU-based linear bounded volume hierarchy [39], can be used to accelerate the ray casting algorithms for this purpose. We will further discuss its application to an automated CFD solver in the future. In next section, we are going to focus on the accuracy and efficiency of p-adaptive FR using linear trees.

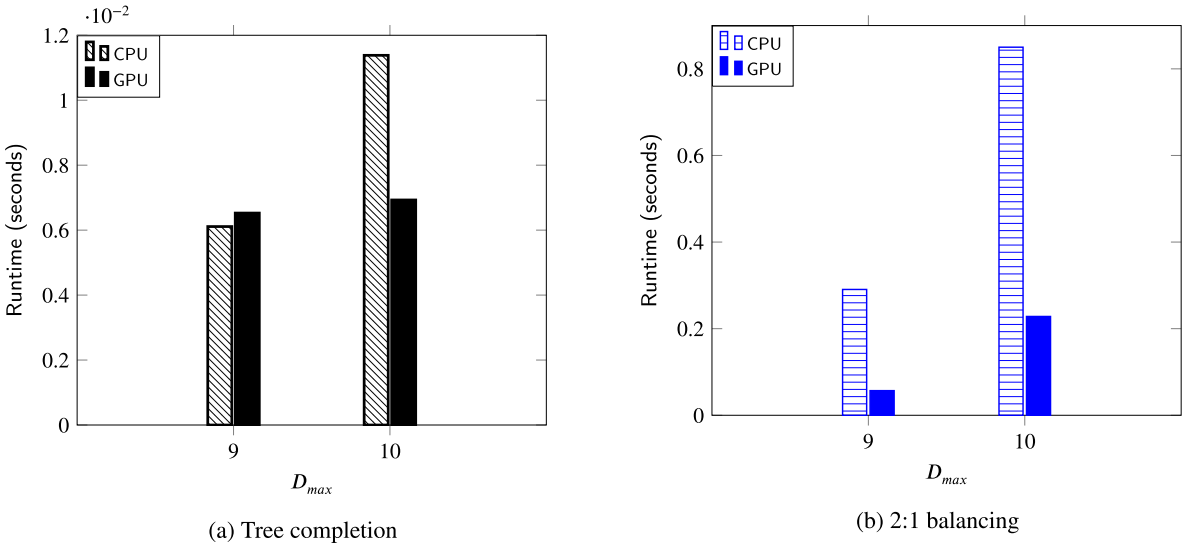


Fig. 9. Computational cost of tree completion and 2:1 balancing for Stanford bunny.

5.2. Performance study of the adaptive solver

As a proof of concept, we opt to use the canonical inviscid isentropic vortex propagation problem to study the performance of the adaptive solver with linear trees. The mathematical description of the vortex is formulated as

$$\begin{cases} \delta u = -\frac{\alpha}{2\pi}(y - y_0)e^{\phi(1-r^2)}, \\ \delta v = \frac{\alpha}{2\pi}(x - x_0)e^{\phi(1-r^2)}, \\ \delta w = 0, \\ \delta T = -\frac{\alpha^2(\gamma-1)}{16\phi\gamma\pi^2}e^{2\phi(1-r^2)}, \end{cases} \tag{12}$$

where $\phi = \frac{1}{2}$, $\alpha = 5$, and $r = (x - x_c(t))^2 + (y - y_c(t))^2$. $(x_c(t), y_c(t))^\top$ is the axis of the vortex center. Initially, the vortex is centered at axis $(0,0)^\top$. The free stream is defined as $(\rho, u, v, w, \mathrm{Ma}) = (1, 1, 1, 0, 0.5)$. A fixed time step size $\Delta t = 0.001$ is used for all simulations. The minimum element sizes in the mesh Δx_{min} were chosen as 25/64, 25/128, 25/256 for all simulations.

The first set of simulations were run with a 2D computational domain of size $[-12.5, 12.5]^2$. Grid refinement study was performed on both uniform and non-uniform meshes. For AMR with octree trees, the minimum tree depth was set as $D_{min} = 3$ for all 2D simulations and the maximum tree depth would vary from 6,7,8 for different Δx_{min}. We used a solution-based adaptation criterion to focus on the computational cost of tree manipulations, data transfer, and solver time. The adaptation criterion was defined as the following list.

- For an element, if there are more solution points within $\sqrt{(x - x_c(t))^2 + (y - y_c(t))^2} < 5$ than those are $\sqrt{(x - x_c(t))^2 + (y - y_c(t))^2} > 6$, then this element is marked for refinement.
- For an element, if there are fewer solution points within $\sqrt{(x - x_c(t))^2 + (y - y_c(t))^2} < 5$ than those are $\sqrt{(x - x_c(t))^2 + (y - y_c(t))^2} > 6$, then this element is marked for coarsening.
- Otherwise, the element remains the same.

Adaptation was performed every 10 time steps. Herein, L_2 norm of the density error, which is defined as

$$L_2(e_\rho) = \sqrt{\int (\rho_{exact} - \rho_{num})^2 dV}, \tag{13}$$

is used for the grid refinement study and the order of accuracy is defined as

$$\text{Order} = log(e_{\rho,1}/e_{\rho,2})/log(\Delta x_{min,1}/\Delta x_{min,2}) \tag{14}$$

in this work for any Mesh 1 and Mesh 2.

The statistics of simulations run on uniform meshes are presented in Table 1. Nominal orders of accuracy were preserved through the grid refinement study. In Fig. 10, we showcase the final nonuniform mesh of the simulation run using p^3 FR. The naive refinement methodology ended up with a very conservative/big refined region. Statistics of simulations performed on non-uniform meshes can

Table 1
Grid refinement study on domain $[-12.5, 12.5]^2$ with 2D uniform meshes.

Non-adaptive p^1 FR				
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Element count
25/64	$1.750e-02$		32.02	4096
25/128	$2.68e-03$	2.71	79.66	16384
25/256	$4.01e-04$	2.74	215.87	65536
Non-adaptive p^2 FR				
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Element count
25/64	$4.30e-04$		40.45	4096
25/128	$5.86e-05$	2.87	125.45	16384
25/256	$8.59e-06$	2.77	423.95	65536
Non-adaptive p^3 FR				
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Element count
25/64	$1.22e-05$		70.03	4096
25/128	$5.21e-07$	4.55	194.93	16384
25/256	$3.24e-08$	4.01	685.51	65536

Table 2
Grid refinement study on domain $[-12.5, 12.5]^2$ with 2D nonuniform meshes.

Adaptive p^1 FR						
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Adaptation (s)	Element count	D_{max}
25/64	$1.75e-02$		40.00	10.22	865	6
25/128	$2.68e-03$	2.71	47.24	14.74	2803	7
25/256	$4.01e-04$	2.74	66.18	17.62	10519	8
Adaptive p^2 FR						
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Adaptation (s)	Element count	D_{max}
25/64	$4.30e-04$		40.45	9.91	856	6
25/128	$5.86e-05$	2.87	53.82	14.78	2809	7
25/256	$8.59e-06$	2.77	114.05	22.87	10513	8
Adaptive p^3 FR						
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Adaptation (s)	Element count	D_{max}
25/64	$1.22e-05$		39.94	9.46	856	6
25/128	$5.21e-07$	4.55	60.54	14.60	2809	7
25/256	$3.25e-08$	4.00	162.58	25.22	10627	8

be found in Table 2. The adaptation cost in this table includes all aspects related to adaptation, such as tree manipulations, data transfer and face query. And we only observed trivial $L_2(e_\rho)$ differences between simulations running on uniform and non-uniform meshes. As expected, the computational cost can be significantly reduced through mesh adaptation. In Fig. 11a, we present the plots of total cost vs. $L_2(e_\rho)$ for these simulations. As shown in Fig. 11b, when the polynomial degree increases, the achieved speedup increases as well. And a speedup of 4.22 was achieved for p^3 FR when $D_{max} = 8$. This is a good testimony of the advantage of using high-order methods with octree-based adaptation techniques and is consistency with what we observed in our previous work of polynomial adaptation [24]. For the same polynomial degree, when the mesh was refined, speedup became more significant simply due to the fact the reduction in total element count became more profound. Note that for small problems, the overhead of adaptation was overwhelming such that no or trivial speedup was achieved. In Fig. 12a, 12b, and 12c, we illustrate runtime of different parts in the adaptive solver including total cost, total solver cost, and total adaptation cost, where three different components of adaptation cost, namely, tree manipulations, data transfer and face connectivity query are also shown. Note that tree manipulations includes tree adaptation, 2:1 balancing, and finding the mapping between old and new trees. For 2D problems, tree manipulations took the most part of the adaptation time. As the problem size increases, the cost of data transfer, compared to that of the building the face connectivity gradually increases. In Fig. 12d, the percentiles of the adaptation cost in the total cost are illustrated. Overall, as the polynomial degree increases, the ratio of $\frac{\text{Adaptation}}{\text{Total cost}}$ will decrease. For p^3 polynomials on the finest mesh, adaptation only took around 15% of the total cost. For p^1 polynomials, it could be as much as 31.2% of the total cost.

The second test we performed was on a larger computational domain, such that the vortex could propagate a longer distance within one period and larger maximum tree depths could be used to test the efficiency of our adaptive solver. The computational domain was within $[-50, 50]^2$ and the final status of the mesh and vortex are shown in Fig. 13. Overall statistics of the simulations on uniform meshes and nonuniform meshes are documented in Table 3 and Table 4. We could not run the simulation on the finest uniform mesh with p^3 FR because of not enough GPU memory. The speedup that AMR offers for this particular experiment, shown in Fig. 14, is around a factor of 49 for p^2 polynomials on the finest mesh. In Fig. 15, bar charts of different components of the entire solver are plotted. The trends of different components are similar to what are observed for simulations run on the smaller domain. The overhead of adaptation compared to the total cost slightly increased. In Fig. 16, we present the effect of tree depth on the

Table 3
Grid refinement study on domain $[-50, 50]^2$ with 2D uniform meshes.

p^1 FR				
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Element count
25/64	$4.65e-02$		870.31	65536
25/128	$9.19e-03$	2.34	3108.43	262144
25/256	$1.28e-03$	2.84	11554.10	1048576
p^2 FR				
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Element count
25/64	$1.17e-03$		1747.67	65536
25/128	$9.15e-05$	3.68	6299.62	262144
25/256	$9.49e-06$	3.27	23706.10	1048576
p^3 FR				
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Element count
25/64	$2.68e-05$		2720.30	65536
25/128	$5.34e-07$	5.64	9993.02	262144

Table 4
Grid refinement study on domain $[-50, 50]^2$ with 2D nonuniform meshes.

Adaptive p^1 FR						
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Adaptation (s)	Element count	D_{max}
25/64	$4.64e-02$		182.39	56.75	961	8
25/128	$9.19e-03$	2.33	211.71	77.14	2899	9
25/256	$1.28e-03$	2.84	294.26	92.44	10615	10
Adaptive p^2 FR						
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Adaptation (s)	Element count	D_{max}
25/64	$1.18e-03$		168.80	49.13	952	8
25/128	$9.19e-05$	3.68	230.35	76.90	2905	9
25/256	$9.54e-06$	3.27	488.48	115.14	10609	10
Adaptive p^3 FR						
Δx_{min}	$L_2(e_\rho)$	Order	Total cost (s)	Adaptation (s)	Element count	D_{max}
25/64	$2.68e-05$		176.38	49.69	952	8
25/128	$5.34e-07$	5.64	273.20	75.77	2905	9
25/256	$3.46e-08$	3.95	710.11	135.08	10723	10

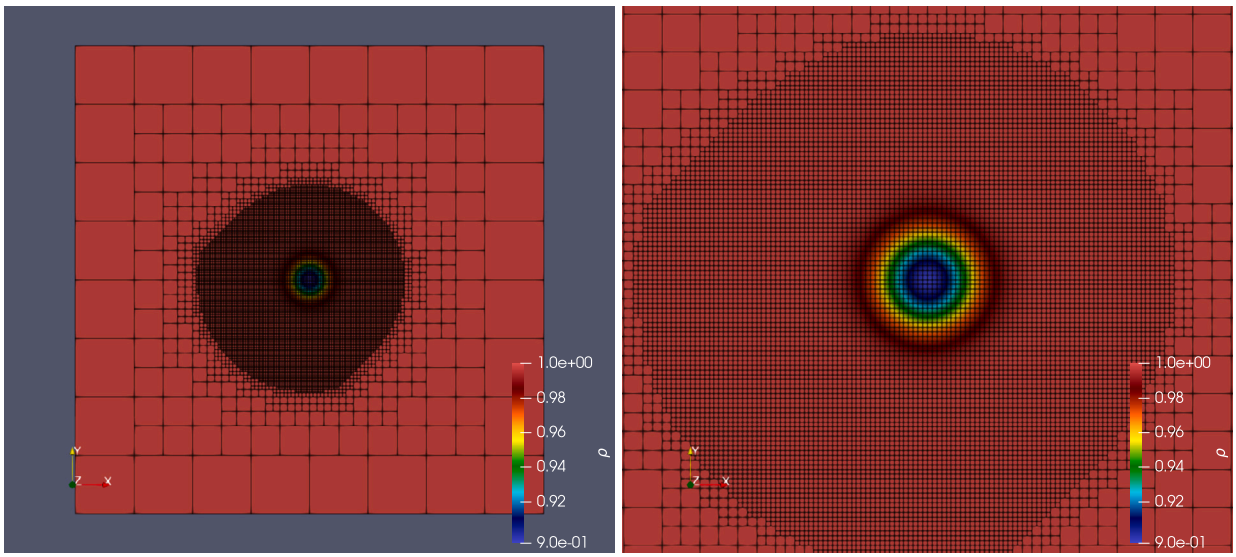
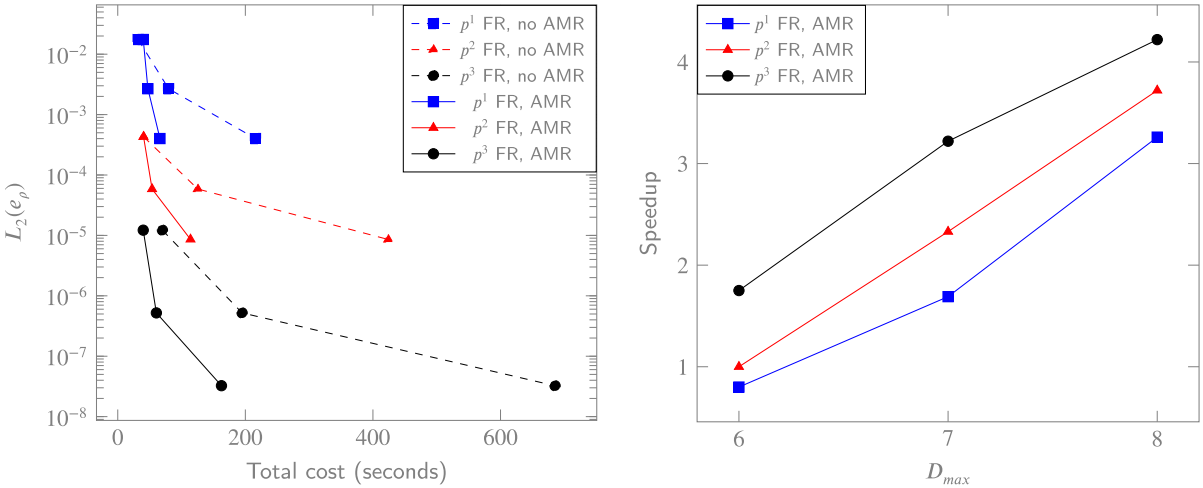


Fig. 10. Final status of the vortex on 2D mesh at $t = 25$ for small domain size. $D_{min} = 3$ and $D_{max} = 8$.



(a) Total cost vs. error

(b) D_{max} vs. speedup

Fig. 11. Runtime vs. error for FR with/without AMR for simulations run on 2D domain $[-12.5, 12.5]^2$.

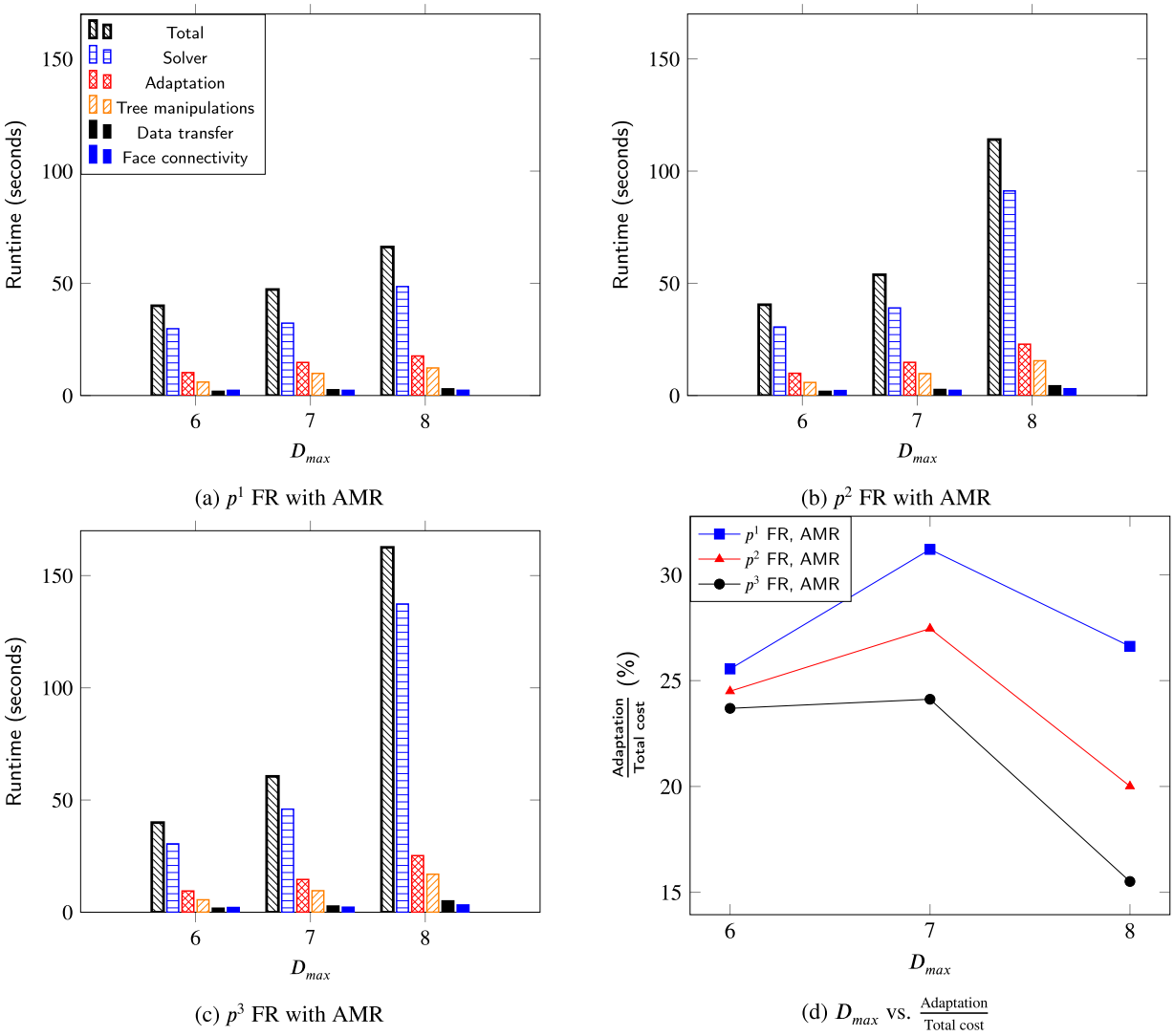
adaptation cost. When larger tree depth is allowed, in Algorithm 4, one need to perform balancing for more levels. Therefore, it is reasonable to see that the normalized adaptation cost of simulations run using $D_{max} = 10$ was larger than that using $D_{max} = 8$ given that the element count were close.

The third tests were conducted on 3D meshes. Simulations were all run with adaptation turned on. The domain on the xy plane was $[-12.5, 12, 5]^2$ and the thickness of the domain in z direction was 1.5625. We showcase a quarter of the vortex and domain after one period with mesh turned on in Fig. 17. The 3D effect would lead to a significant element count in the vortex region as we refine the mesh. The minimum tree depth was set as $D_{min} = 4$ such that in the coarsest region, there was only one element in z direction. The maximum tree depth D_{max} varied from 6 to 8 for this test. We used the same adaptation criteria as that in 2D simulations. In Table 5, we present the overall statistics. In Fig. 18a, 18b, and 18c, the computational cost of different components of the entire solver is illustrated. For 3D simulations, the overhead of adaptation compared to the total cost is significantly smaller. Similar to 2D results, tree manipulations took most of the time in adaptation. In Fig. 18d, we observe that as the total element count increases or the polynomial degree increases, the ratio $\frac{\text{Adaptation}}{\text{Total cost}}$ will decrease. For p^2 FR with $D_{max} = 8$, it is slightly less than 2%; and for p^3 FR with $D_{max} = 8$, it is slightly smaller than 1.5%. Compared to low-order methods or 2D problems, more inner points are nested within one element such that the element count becomes relatively smaller compared to the total number of solution points. This further demonstrates that with high-order methods, we can minimize the overhead of adaptation on the GPU hardware.

For dynamic memory management, `thrust::device_vector`, a container from THRUST, that we use for data storage will automatically allocate a new chunk of memory, then copy the data from the old memory space to the new space, and finally release the old memory, when its resize functionality is invoked and the demanded memory exceeds its capacity. This background procedure would cause the code to crash when the hardware is heavily loaded. Therefore, in the framework of the developed adaptive solver, pre-allocating is employed to minimize allocation overhead and memory fragmentation for the adaptive solver as well as preventing crash. For the adaptive solver, we use element number as 17,000 as the value to pre-allocate memory. In Fig. 19, the memory consumption details are presented for the adaptive solver on simulating the problem on the 3D domain. The tree used around 25 megabytes of memory only. The memory usage of the tree was 0.60%, 1.34%, and 3.57% of the total usage for p^3, p^2 and p^1 FR methods, respectively. One would expect the ratio to further increase for 2D problems. But overall, the fluid solver would consume most of the memory.

5.3. Viscous flow over a square cylinder

Flows over square cylinders are of engineering significance for structure design, flow-induced vibration analysis, etc. In this work, we present simulation results of the flow over a square cylinder of aspect ratio $A = 6$, defined by $A = L/W$, where L is the spanwise length of the cylinder and W is the width of the cylinder. The studied Reynolds number was $\mathrm{Re}_\infty = 400$ in this work and $\mathrm{Ma}_\infty = 0.1$. In the work of Sohankar et al. [40], simulations of Re_∞ ranging from 150 to 500 can be found for incompressible flows. Minimum tree depth for this simulation was fixed as $D_{min} = 6$. We truncated the tree in z-direction, such that there was only one element at far-field in z-direction. $D_{max} = 12$ and $D_{max} = 13$ were used such that there were 64 elements and 128 elements in the spanwise direction in the vicinity of the wall, respectively. The smallest elements had a dimension of $(0.0625W, 0.0625W, 0.09375W)$ for $D_{max} = 12$ and $(0.03125W, 0.03125W, 0.046875W)$ for $D_{max} = 13$. The domain size in this work was $[-128W, 128W]^2 \times [0, 6W]$, which was significantly larger in xy-plane than the one studied in [40] because truncation of the tree was only supported in z-direction in the developed code. A simple feature-based adaptation methodology was used for this study. We computed the smoothness indicator of an element e based on variable ϕ as



(a) p^1 FR with AMR (b) p^2 FR with AMR

(c) p^3 FR with AMR (d) D_{max} vs. $\frac{\text{Adaptation}}{\text{Total cost}}$

Fig. 12. Computational cost of different components of the adaptive p^1, p^2, and p^3 on 2D domain $[-12.5, 12.5]^2$.

Table 5
Grid refinement study on domain $[-12.5, 12.5]^2 \times [0, 1.5625]$ with 3D nonuniform meshes.

Adaptive p^1 FR						
Δx_{min}	$L_2(\rho)$	Order	Total cost (s)	Adaptation (s)	Element count	D_{max}
25/64	$2.19e-02$		59.05	11.36	3049	6
25/128	$3.35e-03$	2.71	244.39	26.43	20703	7
25/256	$5.01e-04$	2.74	1473.02	61.72	160220	8
Adaptive p^2 FR						
Δx_{min}	$L_2(\rho)$	Order	Total cost (s)	Adaptation (s)	Element count	D_{max}
25/64	$5.37e-04$		122.56	14.22	3028	6
25/128	$7.32e-05$	2.88	607.23	27.23	20759	7
25/256	$1.07e-05$	2.77	4193.54	81.39	160346	8
Adaptive p^3 FR						
Δx_{min}	$L_2(\rho)$	Order	Total cost (s)	Adaptation (s)	Element count	D_{max}
25/64	$1.53e-05$		226.83	17.32	3028	6
25/128	$6.52e-07$	4.55	1301.09	36.72	20787	7
25/256	4.05e-08	4.00	9051.31	123.37	162082	8

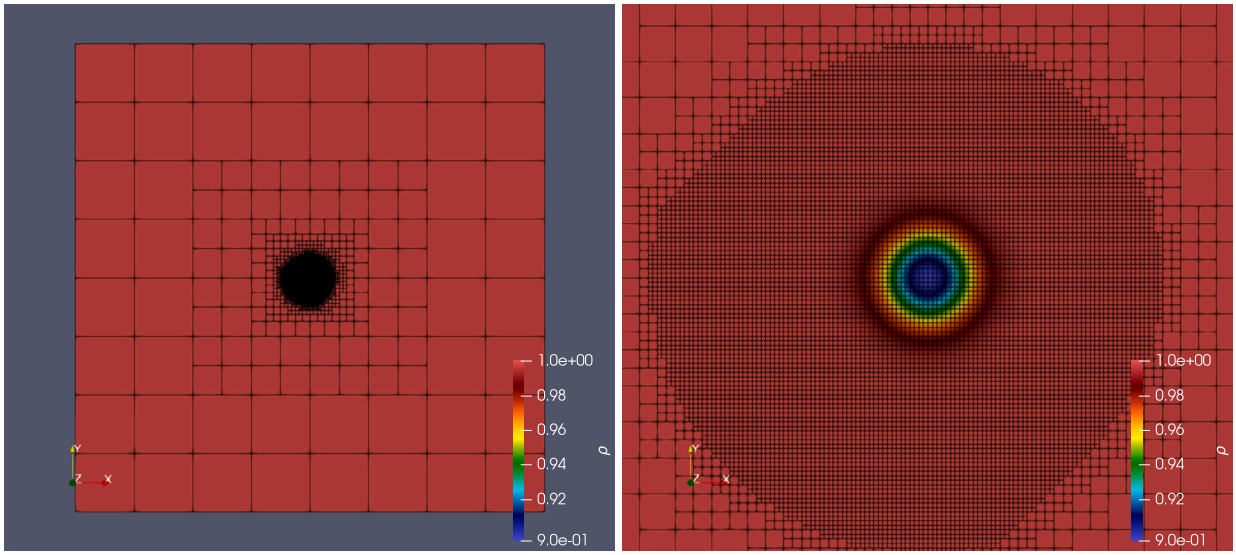
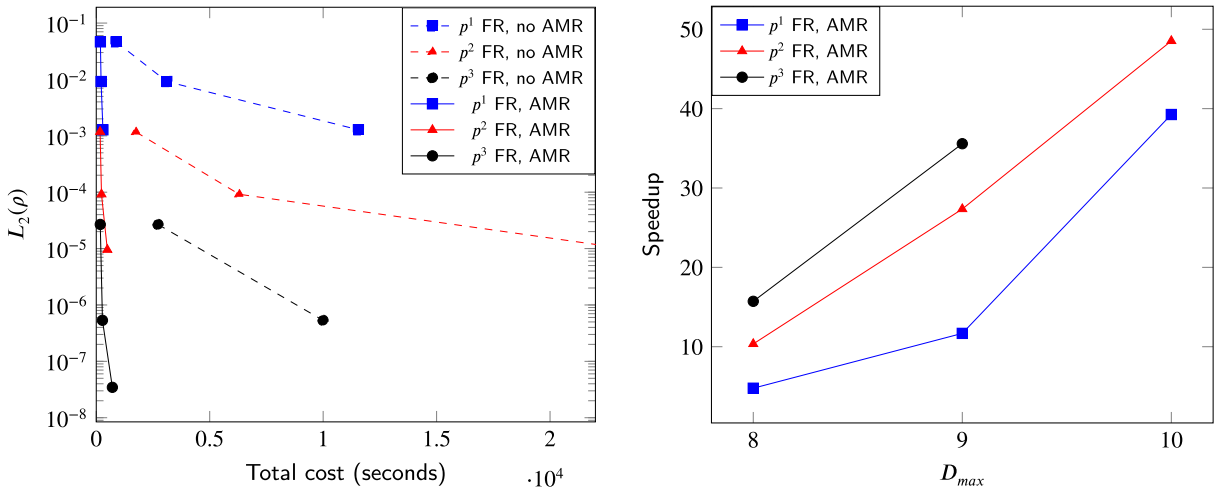


Fig. 13. Final status of the vortex on 2D mesh at $t = 100$ for larger domain size using p^3 FR. $D_{min} = 3$ and $D_{max} = 10$.



(a) Total cost vs. error

(b) D_{max} vs. Speedup

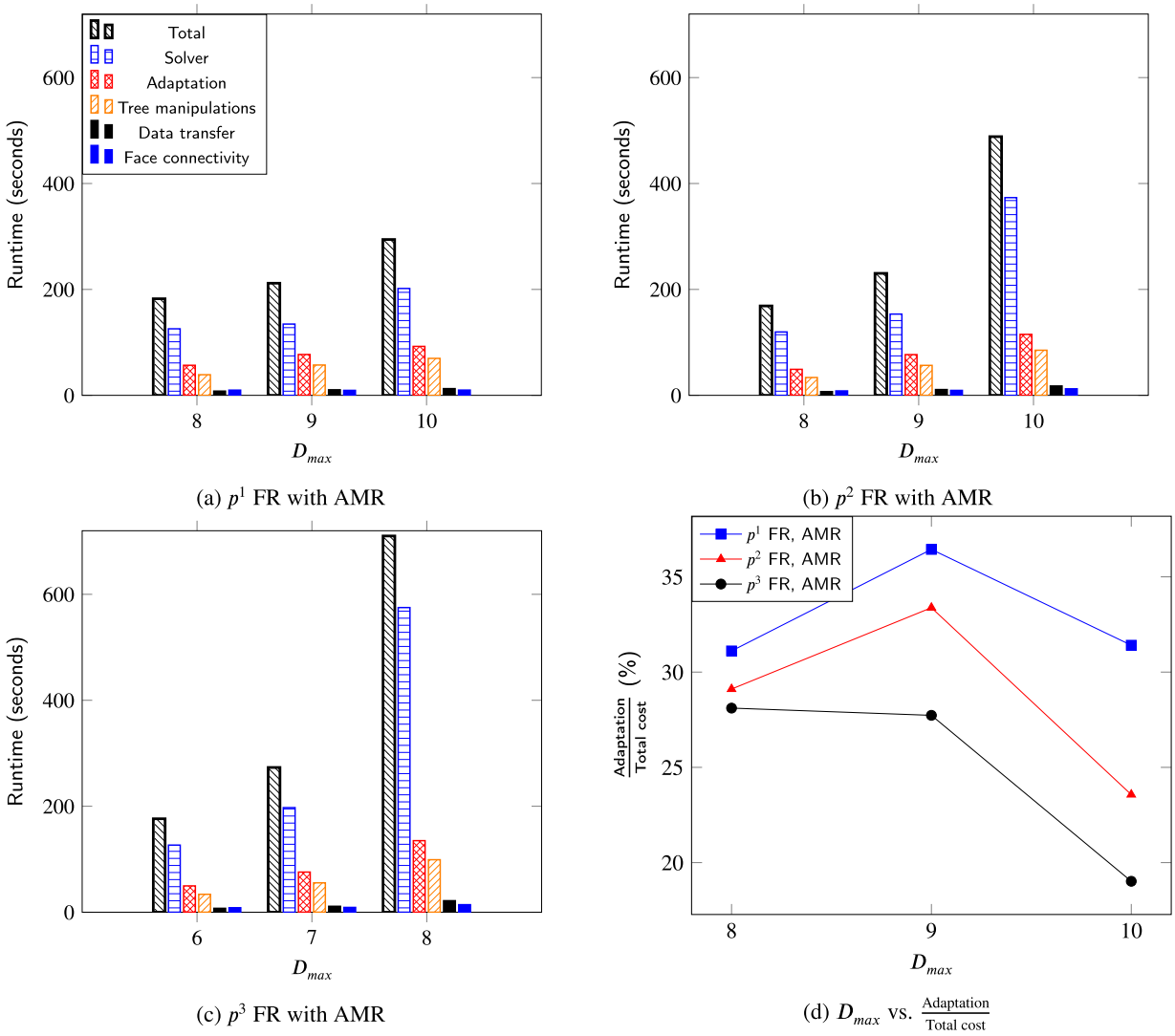
Fig. 14. Runtime vs. error for FR with/without AMR for simulations run on 2D domain $[-50, 50]^2$.

$$\eta_e = \frac{||\phi^p - \phi^{p-1}||_{L_2}}{||\phi^p||_{L_2}}, \tag{15}$$

which was modified from the work of Persson and Peraire [41] by Naddei et al. [36] for feature-based adaptation. In this work, we calculated smoothness indicators for any element $\eta_e^{v_i}$ based on modified momentum $\rho v_i + \rho_\infty U_\infty$, $i = 1, \ldots, \text{dim}$. The extra term $\rho_\infty U_\infty$ was used to avoid zero denominators since the flow field was initialized using $(\rho_\infty, U_\infty, 0, 0, \text{Re}_\infty)$. The coarsening and refinement criteria were defined as follows.

- For any element, if either one of the smoothness indicator $\eta_e^{v_i} > \alpha\ \eta_{e,max}^{v_i}$, then the element will be marked for refinement, where $\eta_{e,max}^{v_i}$ is the maximum value of $\eta_e^{v_i}$ of all elements.
- For any element, if all of the smoothness indicators $\eta_e^{v_i} < \beta\ \eta_{e,max}^{v_i}$, then the element will be marked for coarsening.
- Otherwise, the element remains the same.

$\alpha = 0.1$ and $\beta = 0.001$ were used for p^3 FR and $\alpha = 0.1$ and $\beta = 0.01$ were used for p^2 FR with adaptation in this study. Since transition happens in the wake, the above featured-based refinement procedure would lead to elements being refined to D_{max} in the wake region such that the program would crash due to insufficient memory on the GPU card. To get some meaningful results with



(a) p^1 FR with AMR (b) p^2 FR with AMR

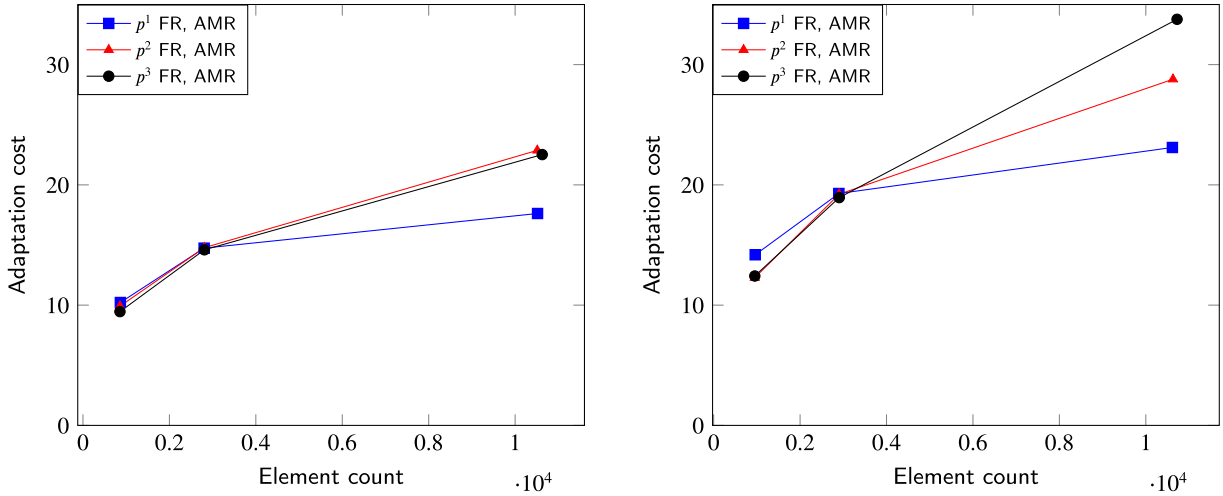
(c) p^3 FR with AMR (d) D_{max} vs. $\frac{\text{Adaptation}}{\text{Total cost}}$

Fig. 15. Computational cost of different components of adaptive p^1, p^2, and p^3 FR on 2D domain $[-50, 50]^2$.

limited hardware, for adaptive p^3 FR, elements were not allowed to be refined to $D = 12$ when $x > 0.875W$; for p^2 FR, elements were not allowed to be refined to $D = 13$ when $0.625W < x < 2W$ and were not allowed to be refined to $D = 12$ when $x > 2W$. Octants were not allowed to be refined when $x > 20W$. Note that the center of the cylinder was coincident with z-axis.

In this work, due to the problem size, single precision calculations were used such that all simulations could fit in the available memory. Witherden and Jameson reported that for unsteady simulations of turbulent flows, the predictions of single precision calculations were favorably close to those of double precision calculations [42]. $\Delta t = 0.0002W/U_\infty$ was used for all simulations. p^3 FR without adaptation was used to run the simulation to $t = 200W/U_\infty$ and there were 178,424 elements and $D_{max} = 12$. The solutions at $t = 100W/U_\infty$ were used as the initial conditions for p^2 FR without adaptation where the mesh was refined in the vicinity of the wall as well as the near-wall wake region such that $D_{max} = 13$ and the total number of elements was increased to 349,840. Illustrations of these two different meshes are shown in Fig. 20.

Simulations using p^2 and p^3 FR with adaptation were also started from the initial state at $t = 100W/U_\infty$ and the initial meshes were the ones shown in Fig. 20a and Fig. 20b, respectively. Mesh adaptations were performed every 40 time steps for both simulations. The overall statistics of 4 simulations are documented in Table 6 and they were collected for $t \in (100W/U_\infty, 200W/U_\infty]$. Force histories in this time period are depicted in Fig. 21a. The results of current study using different polynomial degrees and mesh resolutions agree well with each other. Slight oscillations were observed in the C_d histories of the adaptive solver, which is believed to be caused by the *ad hoc* restrictions of the feature-based adaptation algorithms to avoid crash since such behavior was not observed in our experiments with 2D square cylinder when those restrictions were not used. The differences of averaged drag coefficients $\overline{C_d}$ are within 1.5% and that of root mean squares of lift coefficients C_L' are within 2%. The Strauhal number S_t predictions are the same. Isosurface of Q-criterion where $Q = 0.5$ of these 4 simulations are shown in Fig. 22. Fig. 21b demonstrates that



(a) $D_{max} = 6, 7, 8$ and $D_{min} = 3$. (b) $D_{max} = 8, 9, 10$ and $D_{min} = 3$. Cost normalized by 4.

Fig. 16. Effect of tree depth on the cost of adaptation for 2D simulations.

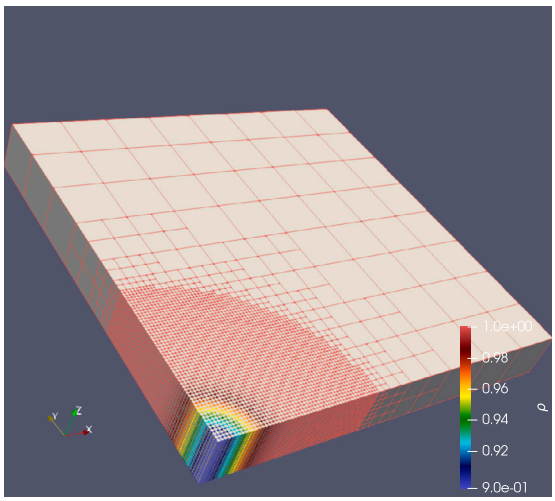
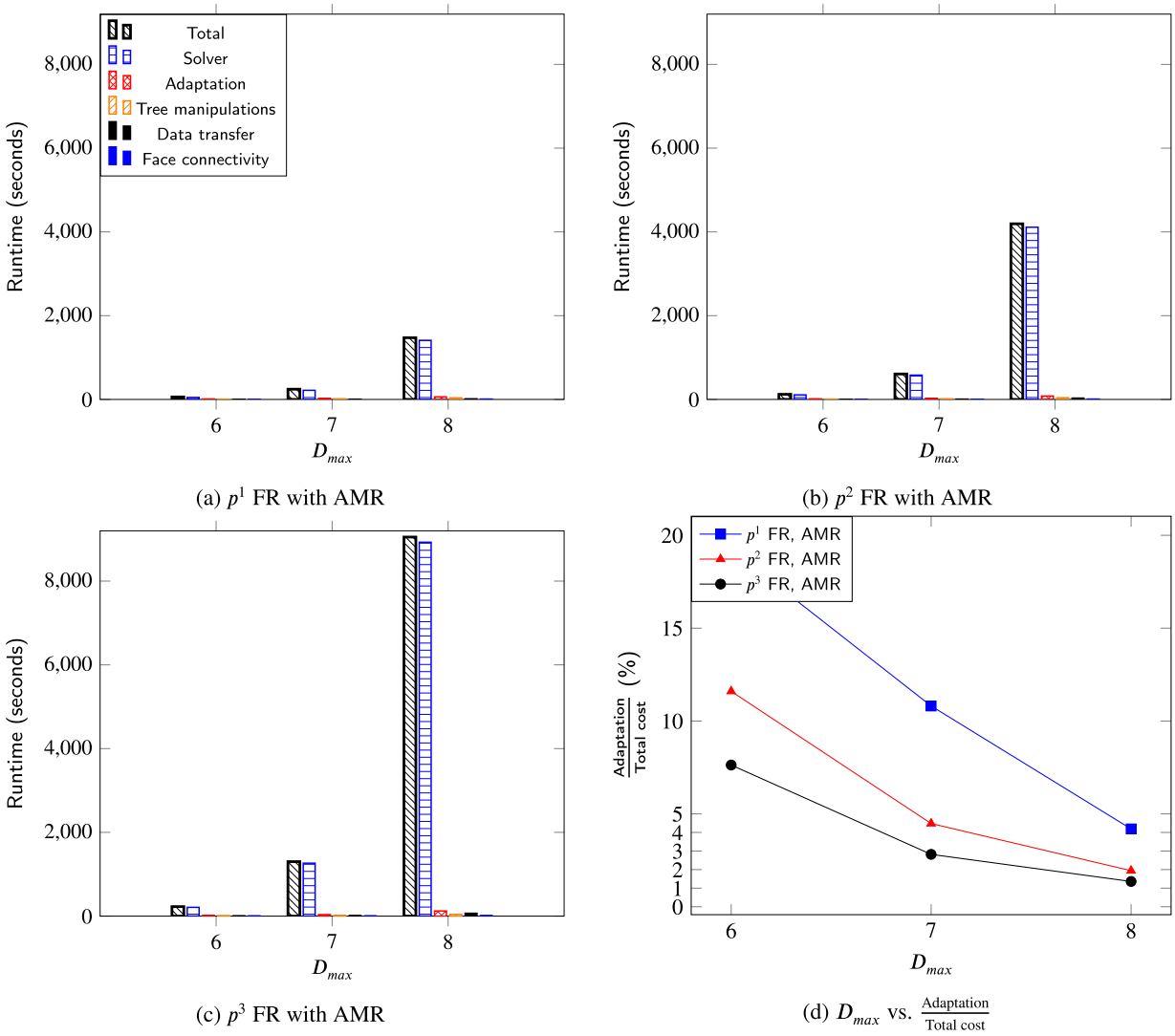


Fig. 17. Top-right quarter final mesh using p^2 FR with adaptation for vortex propagation on the 3D domain.

the unsteadiness of the transitional flow will lead to non-trivial variations in the total number of elements when the feature-based adaptation was used. Qualitatively, the vortex structures predicted with adaptation are close to those predicted without adaptation. The tips of the hair-pin structures were slightly better resolved with adaptation off. In Fig. 23, the mesh distributions at different slices of the adaptive solvers are shown. Mesh were more dense to resolve high vorticity region and mesh distribution at different slices along the spanwise direction were observed to be quite different. For simulating the stationary square cylinder, the speedup factor that adaptation can offer is around 1.17 for p^2 FR and 1.25 for p^3 FR while the corresponding reductions in the element count are 16% and 29%. The speedup is consistent with the reduction of element count. This is encouraging as we would expect when applied to a problem of large object motions, the adaptive solver that can save more significant computational cost, as indicated by the vortex propagation studied in the previous section.

We are more interested in the efficiency of the adaptation algorithms via linear trees for this study. In Fig. 24, we illustrate the computational cost of different components of the adaptive solver. The overall ratio $\frac{\text{Adaptation}}{\text{Total cost}}$ is 0.6% for p^2 and 1% for p^3 FR with adaptation. The observed low overhead of adaptation is consistent with results using double precision calculations for the inviscid vortex propagation. It is noteworthy that the ratio is higher for p^3 FR when its element count is only 43.3% of that of p^2 FR. The difference in the adaptation cost mainly comes from tree manipulation while the costs of data transfer and face connectivity are close. As shown in Fig. 23, for the simulation using p^2 FR, elements at depth $D = 11$ are densely distributed to resolve the vortices while with p^3 FR, the elements at depth $D = 11$ are more sparse in the wake. This sparsity would possibly lead to inefficiency in tree manipulations since in a thread block, there would be more thread divergence. Nonetheless, the adaptation cost is considered to be insignificant compared to that of the fluid solver when the fluid physics leads to nonuniform mesh distributions and we restrain



(a) p^1 FR with AMR (b) p^2 FR with AMR

(c) p^3 FR with AMR (d) D_{max} vs. $\frac{\text{Adaptation}}{\text{Total cost}}$

Fig. 18. Computational cost of different components of adaptive p^1, p^2, and p^3 FR on domain $[-12.5, 12.5]^2 \times [0, 1.5625]$.

Table 6
Simulation results of the flow over a square cylinder at Re = 400.

Numerical methods	S_t	$\overline{C_d}$	C'_L	Adaptation	Element count	Time (hours)	D_{max}
p^2 FR	0.131	1.685	0.76	No	349,840	44.56	13
p^2 FR	0.131	1.681	0.75	Yes	293,040	38.15	13
p^3 FR	0.131	1.707	0.775	No	178,424	51.03	12
p^3 FR	0.131	1.703	0.758	Yes	127,000	40.71	12
2nd order FV [40]	0.136	1.67	0.64	N/A	$168 \times 120 \times 24$	N/A	N/A

ourselves from further investigations. Note that single precision were used for the fluid solver which would be around 100% faster than double precision calculations while the number representations of the tree algorithms remained the same, compared to what has been discussed in previous section.

As mentioned earlier, we would preallocate memory for the tree as well as the fluid solver. The overall feature-based adaptation algorithms with the additional *ad hoc* restrictions on refinement made sure the total number of element would under control as shown in Fig. 21b. The number of elements that we used for preallocation was 390,000 for p^2 FR and 170,000 for p^3 FR. The memory consumption of the adaptive solver is illustrated in Fig. 25. The tree used 25 megabytes and 57 megabytes of memory for p^3 and p^2 FR methods. In the meantime, the fluid solver took around 4 gigabytes of memory for both methods.

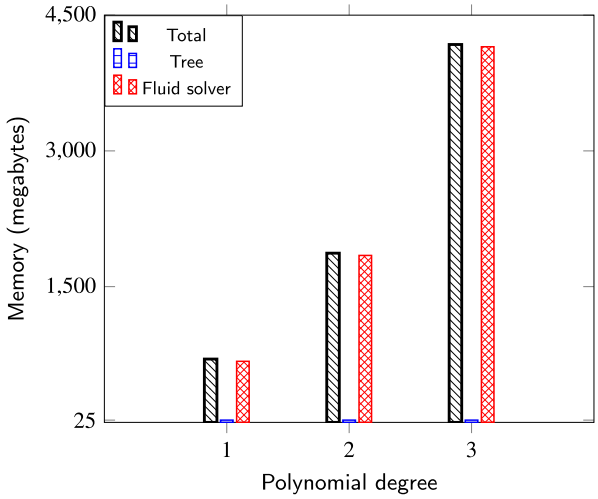
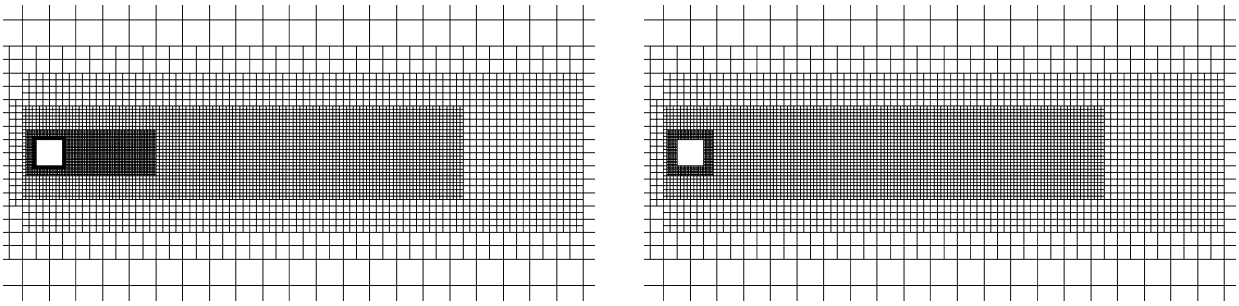
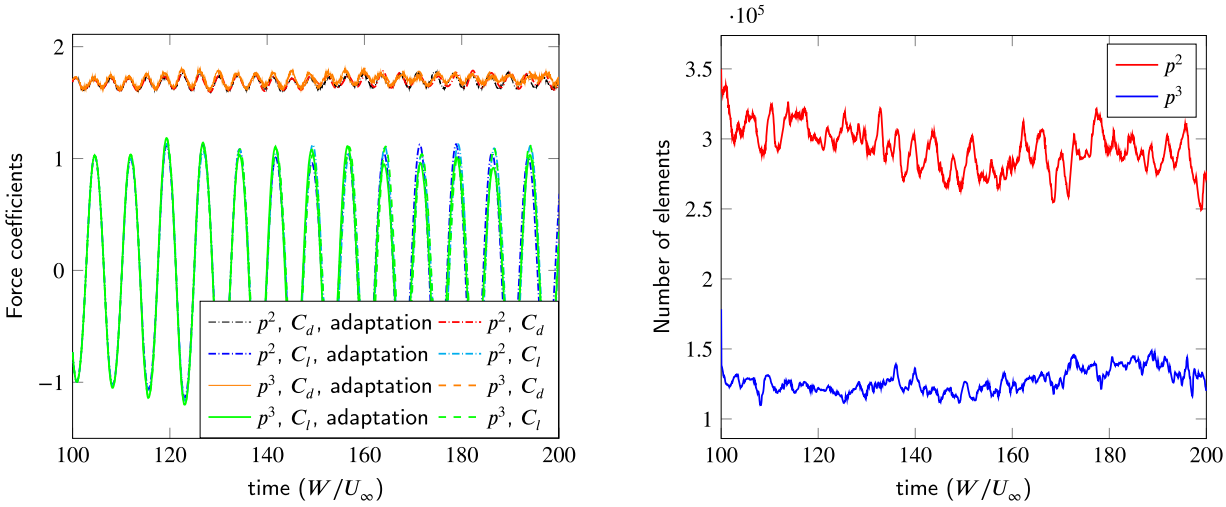


Fig. 19. Memory consumption of the adaptive solver on the 3D mesh of vortex propagation.



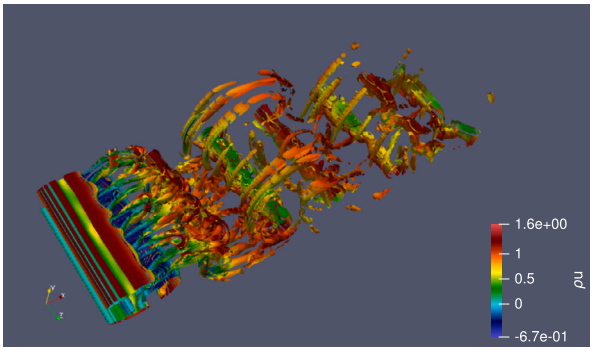
(a) 349,840 elements, smallest element width $0.03125W$ (b) 178,424 elements, smallest element width $0.0625W$

Fig. 20. Mesh (a) for p^2 FR and (b) for p^3 FR simulations of the flow over a square cylinder without adaptation.

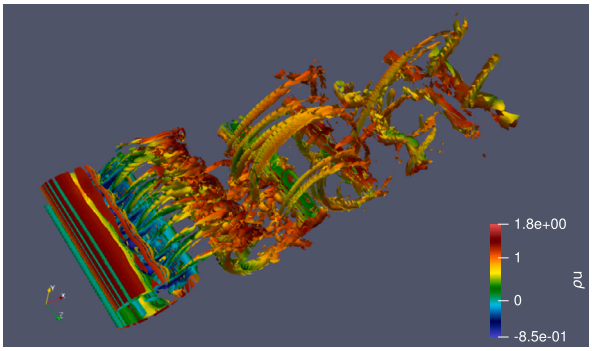


(a) History of force coefficients (b) History of number of elements with adaptation

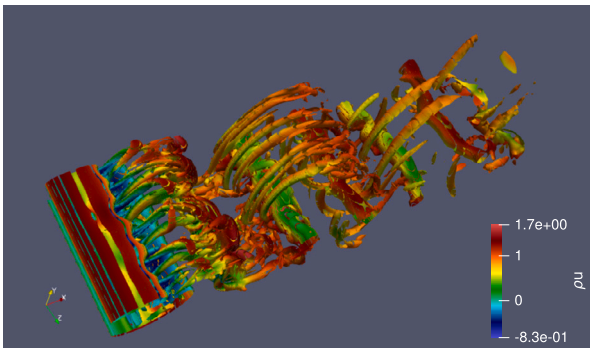
Fig. 21. Time history of different statistics of the flow over the square cylinder. (The reader is referred to the web version of this article for colored plots.)



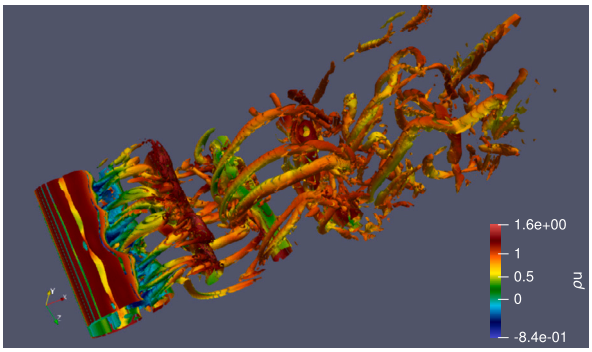
(a) p^2 FR without adaptation



(b) p^2 FR with adaptation

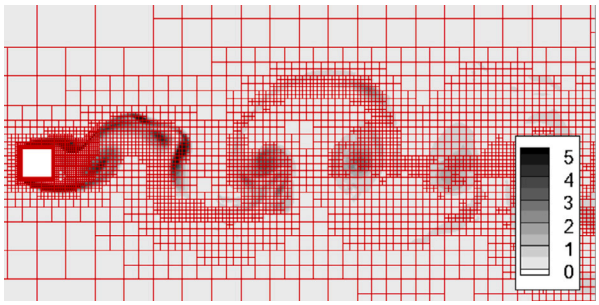


(c) p^3 FR without adaptation

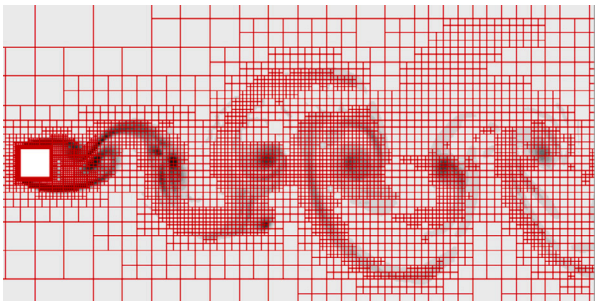


(d) p^3 FR with adaptation

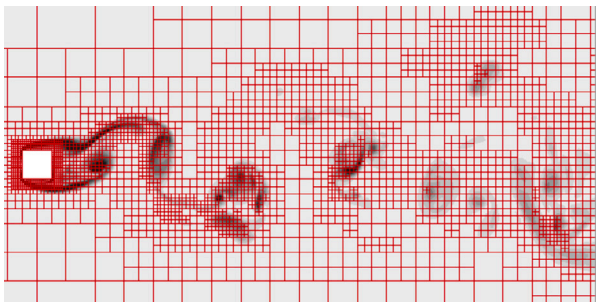
Fig. 22. Instantaneous isosurface of Q-criterion where $Q = 0.5$ colored by momentum in x direction. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)



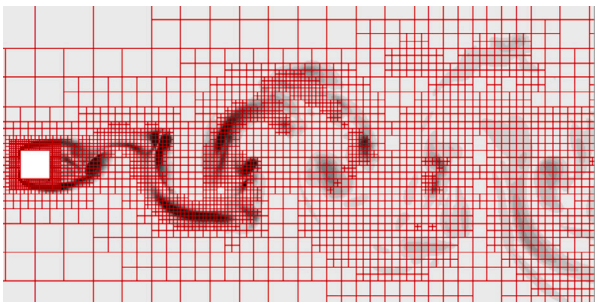
(a) p^2 FR, slice $z = 1$



(b) p^2 FR, slice $z = 4$

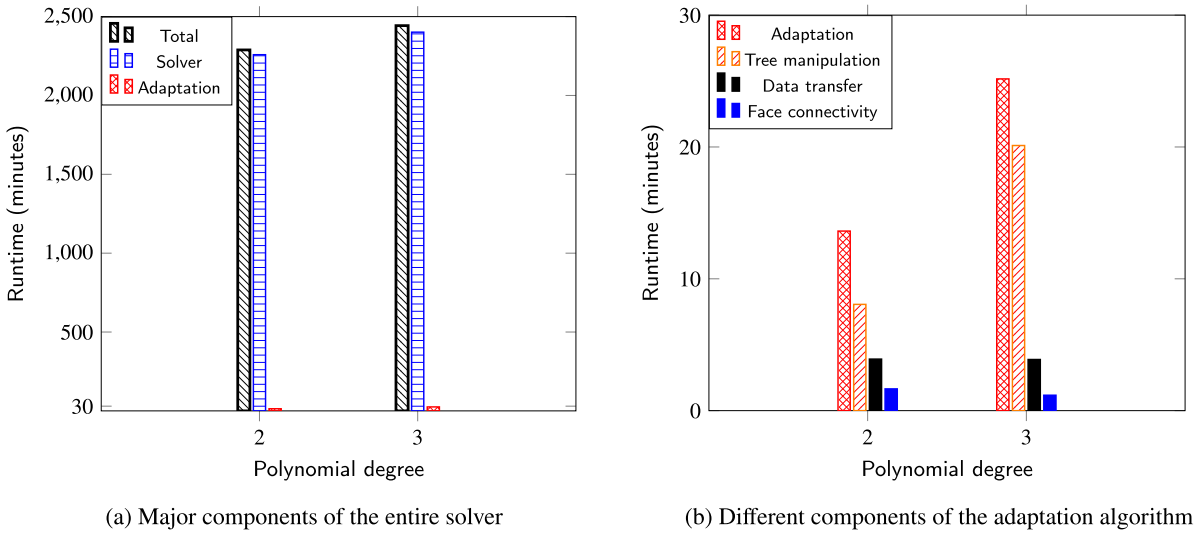


(c) p^3 FR, slice $z = 1$



(d) p^3 FR, slice $z = 4$

Fig. 23. Mesh distributions at different slices for the adaptive solver. The contour is the vorticity magnitude in gray scale.



(a) Major components of the entire solver (b) Different components of the adaptation algorithm

Fig. 24. Computational cost of different components of the adaptive solver for flow over square cylinder.

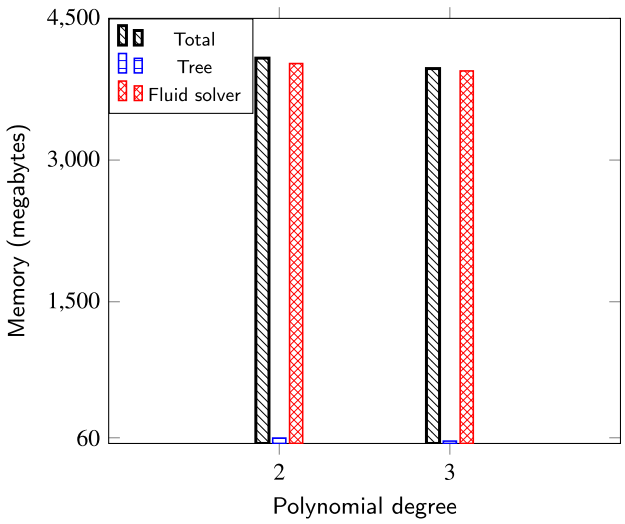


Fig. 25. Memory consumption of the adaptive solver for flow over the square cylinder.

6. Conclusion and future work

We develop the first entirely GPU-based adaptive flux reconstruction method enabled by linear trees for 2D and 3D problems. Novel algorithms are proposed for tree construction, tree balancing as well as connectivity query. The entire solver runs on GPU except that CPU codes are used to launch the kernel functions. We first show that the GPU version of tree balancing can be 4 to 5 times faster than the parallel CPU version. We demonstrate that significant accelerations for long distance vortex propagation problems can be achieved, compared to uniform meshing. We systematically analyze the computational cost of different components of the entire solver. It is found that the overhead of adaptation, including tree adaptation, tree balancing, mapping, data transferring, face connectivity query, is insignificant compared to the total cost, for both inviscid and viscous transitional flows. In 3D simulations, $\frac{\text{Adaptation}}{\text{Total cost}}$ can be smaller than 2% when adaptation is performed every 10-40 time steps with explicit time stepping. The memory footprint of the linear tree is trivial compared to that of the fluid solver. Overall, high-order methods with efficient AMR can be achieved entirely operating on GPU. Additionally, high-order methods tend to benefit more from adaptation compared to low-order ones. We are to address the initial meshing and wall boundary treatment in our future work as well as extending the developed methods to support multiple GPUs.

Admittedly, only bluff bodies can be tackled by the solver developed in this work for wall bounded flows. However, the developed AMR framework can still be useful for a wide range of CFD applications, such as atmosphere modeling [43], jet flow studies [44], etc. Recently, Kou et al. implemented the volume penalty method with the flux reconstruction formulation [45] for viscous flows and Funada and Imamura [46] developed a discrete forcing IBM method with ghost/image points for flux reconstruction solution of

inviscid flows. We have tested both methods for viscous flows. The continuous forcing approach by Kou et al. [45] is only first order in near wall region and often requires significantly smaller elements in near the wall compared to body-fitted mesh with high-order methods, which would limit its applications with explicit time stepping on GPU as Re gets higher. The approach of Funada and Imamura can preserve the nominal order of accuracy for inviscid flows over a 2D cylinder and 3D sphere. However, ghost-cell-type IBM methods normally would have accuracy inconsistency and stability issues up to how the image points are configured [5]. These methods are fairly simple to implement and are suitable for problems with large deformations and we will leave detailed discussion for future work. Moreover, we would also look into coupling the developed method in this work with the overset method developed by Crabill et al. [13] for high Reynolds number flows with wall modeling.

CRediT authorship contribution statement

Lai Wang: Conceptualization, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Freddie Witherden:** Conceptualization. **Antony Jameson:** Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

The first author would like to thank Dr. Qimei Gu for her emotional support during the preparation of this manuscript.

Funding

The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

References

[1] Jeffrey P. Slotnick, Abdollah Khodadoust, Juan Alonso, David Darmofal, William Gropp, Elizabeth Lurie, Dimitri J. Mavriplis, CFD vision 2030 study: a path to revolutionary computational aerosciences, Technical report, 2014.
[2] Charles S. Peskin, The immersed boundary method, Acta Numer. 11 (2002) 479–517.
[3] Markus Uhlmann, An immersed boundary method with direct forcing for the simulation of particulate flows, J. Comput. Phys. 209 (2) (2005) 448–476.
[4] Reza Ghias, Rajat Mittal, Haibo Dong, A sharp interface immersed boundary method for compressible viscous flows, J. Comput. Phys. 225 (1) (2007) 528–553.
[5] Cheng Chi, Abouelmagd Abdelsamie, Dominique Thévenin, A directional ghost-cell immersed boundary method for incompressible flows, J. Comput. Phys. 404 (2020) 109122.
[6] Z.J. Wang, A Quadtree-based adaptive Cartesian/Quad grid flow solver for Navier-Stokes equations, Comput. Fluids 27 (4) (1998) 529–549.
[7] David M. Ingram, Derek M. Causon, Clive G. Mingham, Developments in Cartesian cut cell methods, Math. Comput. Simul. 61 (3–6) (2003) 561–572.
[8] Hong Xiao, Eky Febrianto, Qiaoling Zhang, Fehmi Cirak, An immersed discontinuous Galerkin method for compressible Navier-Stokes equations on unstructured meshes, Int. J. Numer. Methods Fluids 91 (10) (2019) 487–508.
[9] Sharanjeet Kaur, Ge Yan, Jason E. Hicken, High-order cut-cell discontinuous Galerkin difference discretization, AIAA J. (2023) 1–10.
[10] Nicolas Moës, John Dolbow, Ted Belytschko, A finite element method for crack growth without remeshing, Int. J. Numer. Methods Eng. 46 (1) (1999) 131–150.
[11] Markus Geisenhofer, Florian Kummer, Björn Müller, A discontinuous Galerkin immersed boundary solver for compressible flows: adaptive local time stepping for artificial viscosity–based shock-capturing on cut cells, Int. J. Numer. Methods Fluids 91 (9) (2019) 448–472.
[12] Joseph L. Steger, F. Carroll Dougherty, John A. Benek, A Chimera Grid Scheme, 1983.
[13] Jacob Crabill, Freddie D. Witherden, Antony Jameson, A parallel direct cut algorithm for high-order overset methods with application to a spinning golf ball, J. Comput. Phys. 374 (2018) 692–723.
[14] Marshall C. Galbraith, Steve L. Karman, HLPW-4/GMGW-3: high-order discretization technology focus group workshop summary, J. Aircr. (2023) 1–13.
[15] Bernardo Cockburn, George E. Karniadakis, Chi-Wang Shu, Discontinuous Galerkin Methods: Theory, Computation and Applications, vol. 11, Springer Science & Business Media, 2012.
[16] Hung T. Huynh, A flux reconstruction approach to high-order schemes including discontinuous Galerkin methods, in: 18th AIAA Computational Fluid Dynamics Conference, 2007, p. 4079.
[17] Freddie D. Witherden, Antony M. Farrington, Peter E. Vincent PyFR, An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach, Comput. Phys. Commun. 185 (11) (2014) 3028–3040.
[18] B.C. Vermeire, F.D. Witherden, P.E. Vincent, On the utility of GPU accelerated high-order methods for unsteady flow simulations: a comparison with industry-standard tools, J. Comput. Phys. 334 (2017) 497–521.
[19] Mohamed Wahib, Naoya Maruayama, Data-centric GPU-based adaptive mesh refinement, in: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, 2015, pp. 1–7.
[20] Carsten Burstedde, Lucas C. Wilcox, Omar Ghattas, p4est: scalable algorithms for parallel adaptive mesh refinement on forests of octrees, SIAM J. Sci. Comput. 33 (3) (2011) 1103–1133, https://doi.org/10.1137/100791634.
[21] Revathi Jambunathan, Deborah A. Levin, Advanced parallelization strategies using hybrid MPI-CUDA octree DSMC method for modeling flow through porous media, Comput. Fluids 149 (2017) 70–87.
[22] Pavel Pavlukhin, Igor Menshov, GPU-aware AMR on octree-based grids, in: Parallel Computing Technologies: 15th International Conference, PaCT 2019, Almaty, Kazakhstan, August 19–23, 2019, Proceedings 15, Springer, 2019, pp. 214–220.

[23] Hari Sundar, Rahul S. Sampath, George Biros, Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel, SIAM J. Sci. Comput. 30 (5) (2008) 2675–2708.
[24] Lai Wang, Matthias K. Gobbert, Meilin Yu, A dynamically load-balanced parallel p-adaptive implicit high-order flux reconstruction method for under-resolved turbulence simulation, J. Comput. Phys. 417 (2020) 109581.
[25] Haiyang Gao, A unifying lifting collocation penalty formulation including the discontinuous Galerkin, spectral volume/difference methods for conservation laws on mixed grids, J. Comput. Phys. 228 (21) (2009) 8161–8186.
[26] Peter E. Vincent, Patrice Castonguay, Antony Jameson, A new class of high-order energy stable flux reconstruction schemes, J. Sci. Comput. 47 (2011) 50–72.
[27] Joshua Romero, Kartikey Asthana, Antony Jameson, A simplified formulation of the flux reconstruction method, J. Sci. Comput. 67 (2016) 351–374.
[28] Wang Lai, Meilin Yu, Compact direct flux reconstruction for conservation laws, J. Sci. Comput. 75 (2018) 253–275.
[29] Bernardo Cockburn, Chi-Wang Shu, The local discontinuous Galerkin method for time-dependent convection-diffusion systems, SIAM J. Numer. Anal. 35 (6) (1998) 2440–2463.
[30] Guang-Shan Jiang, Chi-Wang Shu, Efficient implementation of weighted ENO schemes, J. Comput. Phys. 126 (1) (1996) 202–228.
[31] Wang Lai, Meilin Yu, Comparison of ROW, ESDIRK, and BDF2 for unsteady flows with the high-order flux reconstruction formulation, J. Sci. Comput. 83 (2) (2020) 39.
[32] Lai Wang, Will Trojak, Freddie Witherden, Antony Jameson, Nonlinear p-multigrid preconditioner for implicit time integration of compressible Navier–Stokes equations with p-adaptive flux reconstruction, J. Sci. Comput. 93 (3) (2022) 81.
[33] Jeroen Baert, Libmorton: C++ morton encoding/decoding library, https://github.com/Forceflow/libmorton, 2018.
[34] Nathan Bell, Jared Hoberock, Thrust: a productivity-oriented library for CUDA, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 359–371.
[35] Krzysztof J. Fidkowski, Philip L. Roe, An entropy adjoint approach to mesh refinement, SIAM J. Sci. Comput. 32 (3) (2010) 1261–1287.
[36] Fabio Naddei, Marta de la Llave Plata, Vincent Couaillier, Frédéric Coquel, A comparison of refinement indicators for p-adaptive simulations of steady and unsteady flows using discontinuous Galerkin methods, J. Comput. Phys. (ISSN 0021-9991) 376 (2019) 508–533, https://doi.org/10.1016/j.jcp.2018.09.045.
[37] Jeremy Ims, Zhi J. Wang, A comparison of three error indicators for adaptive high-order large eddy simulation, in: AIAA SCITECH 2022 Forum, 2022, p. 1201.
[38] David A. Kopriva, A conservative staggered-grid Chebyshev multidomain method for compressible flows. II. A semi-structured method, J. Comput. Phys. (ISSN 0021-9991) 128 (2) (1996) 475–488.
[39] Tero Karras, Maximizing parallelism in the construction of BVHs, octrees, and k-d trees, in: Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics, 2012, pp. 33–37.
[40] Ahmad Sohankar, Christoffer Norberg, Lars Davidson, Simulation of three-dimensional flow around a square cylinder at moderate Reynolds numbers, Phys. Fluids 11 (2) (1999) 288–306.
[41] Per-Olof Persson, Jaime Peraire, Sub-cell shock capturing for discontinuous Galerkin methods, in: 44th AIAA Aerospace Sciences Meeting and Exhibit, 2006, p. 112.
[42] F.D. Witherden, A. Jameson, Impact of number representation for high-order implicit large-eddy simulations, AIAA J. 58 (1) (2020) 184–197.
[43] Michal A. Kopera, Francis X. Giraldo, Analysis of adaptive mesh refinement for imex discontinuous Galerkin solutions of the compressible Euler equations with application to atmospheric simulations, J. Comput. Phys. 275 (2014) 92–117.
[44] Xinmeng Tang, Edyta Dzieminska, Makoto Asahara, A. Koichi Hayashi, Nobuyuki Tsuboi, Numerical investigation of a high pressure hydrogen jet of 82 mpa with adaptive mesh refinement: concentration and velocity distributions, Int. J. Hydrog. Energy 43 (18) (2018) 9094–9109.
[45] Jiaqing Kou, Saumitra Joshi, Aurelio Hurtado-de Mendoza, Kunal Puri, Charles Hirsch, Esteban Ferrer, Immersed boundary method for high-order flux reconstruction based on volume penalization, J. Comput. Phys. 448 (2022) 110721.
[46] Masaya Funada, Taro Imamura, High-order immersed boundary method for inviscid flows applied to flux reconstruction method on a hierarchical Cartesian grid, Comput. Fluids 265 (2023) 105986.