Towards Distributed Semi-speculative Adaptive Anisotropic Parallel Mesh Generation

Kevin Garner*, Christos Tsolakis†, Polykarpos Thomadakis‡ and Nikos Chrisochoides§
Center for Real-time Computing, Old Dominion University, Norfolk, VA 23529, USA

High-performance computing (HPC) mesh generation codes are often optimized while intertwining functionality with performance aspects of code, making these methods highly complex and difficult to maintain amid the evolving intricacies of HPC hardware. Additionally, current state-of-the-art HPC methods utilize collective communication and global synchronization techniques that have been shown to hinder potential scalability. This paper presents the foundational elements of a distributed memory method for adaptive anisotropic mesh generation that is designed to avoid the use of collective communication techniques while leveraging concurrency offered by large-scale computing. In the presented method, meshing functionality is separated from performance aspects by utilizing a separate entity for each - a shared memory mesh generation code called CDT3D and PREMA for parallel runtime support. Lessons are presented regarding some re-designs of CDT3D that were required to enable its integration into the distributed memory method. In the presented method, an initial mesh is data decomposed and subdomains are distributed amongst the nodes of an HPC cluster. The interior elements of subdomains are initially adapted while interface elements (subdomain boundaries) remain frozen. Interface elements then undergo several iterations of shifting so that they are adapted when their data dependencies are resolved. Preliminary results show that the presented method is able to produce meshes of comparable quality to those generated by the original shared memory CDT3D software. However, the distributed method's scalability is hindered by CDT3D's performance with regards to efficiently re-processing data it previously generated during other iterations of interface shifting. Local communication performance regarding non-meshing operations suggests that if CDT3D can be further re-designed, the interface shift operation presents a potentially viable solution in achieving scalability for mesh adaptation when targeting configurations with large numbers of cores.

I. Nomenclature

S	=	subdomain of a partitioned mesh
$I_{1,2}$	=	interface boundary between subdomains 1 and 2
$\mathcal{M}$	=	continuous metric field
M	=	discrete metric field defined at the vertices of a mesh
$C(\cdot)$	=	complexity of a metric field
L_a	=	Euclidean edge length evaluated in the metric of vertex a
M_{mean}	=	metric tensor interpolated at the centroid of a tetrahedron
$\|k\|$	=	volume of a tetrahedron in evaluated metric M_{mean}
Q_k	=	mean ratio shape measure

II. Introduction

Anisotropic mesh adaptation utilizes an anisotropic metric field to modify an existing mesh so that the estimated errors on this mesh, with respect to some simulation, are reduced. This metric field is used to control the orientation and

*Research Assistant
†Research Assistant
‡Research Assistant
§Richard T. Cheng Chair Professor of Computer Science, email: nikos@cs.odu.edu

size of elements individually for each direction so that the generated anisotropic mesh accurately captures the features of the underlying simulation. Mesh adaptation is a critical component in the study of Computational Fluid Dynamics (CFD), as CFD simulations in turn propel the design and analysis of aerospace vehicles. NASA's 2030 Vision identifies mesh generation as a bottleneck in the CFD workflow, underlining the need for leveraging emerging high-performance computing (HPC) architectures to meet the requirements of large-scale, time-dependent CFD problems [1]. This paper presents the foundational elements of a distributed memory method for adaptive anisotropic mesh generation that is designed to leverage concurrency offered by large-scale computing. One of the challenges related to the development of HPC mesh generation codes is how meshing operations are implemented with regards to performance (such as thread management, load balancing, etc.). If meshing functionality and performance are intertwined within a method's code, the respective method becomes highly complex and difficult to maintain amid the evolving intricacies of HPC hardware. It is essential that HPC mesh generation methods be capable of leveraging the concurrency offered by emerging hardware while requiring minimal updates to their source code. Consequently, the presented method separates meshing functionality from performance (scalability) aspects by utilizing a separate entity for each - CDT3D [2, 3] for mesh generation and PREMA [4–6] for parallel runtime support. CDT3D is a shared memory code. It contains meshing operations that are intended to be used within the broader scope of scalable data-parallel and partially coupled methods within a framework we term the Telescopic Approach [7], seen in Figure 1. The Telescopic Approach provides a layout of multiple memory hierarchies within an exascale architecture and describes how different meshing kernels might be utilized at each level to achieve maximum concurrency. CDT3D is specifically built to operate within the lowest level of the hierarchy, exploiting fine-grain parallelism at the core and node levels. PREMA is a parallel runtime system designed to support interoperability between the memory hierarchies of the telescopic approach. It provides constructs that enable asynchronous message passing between encapsulations of data, work load balancing, and migration capabilities all within a globally addressable namespace.

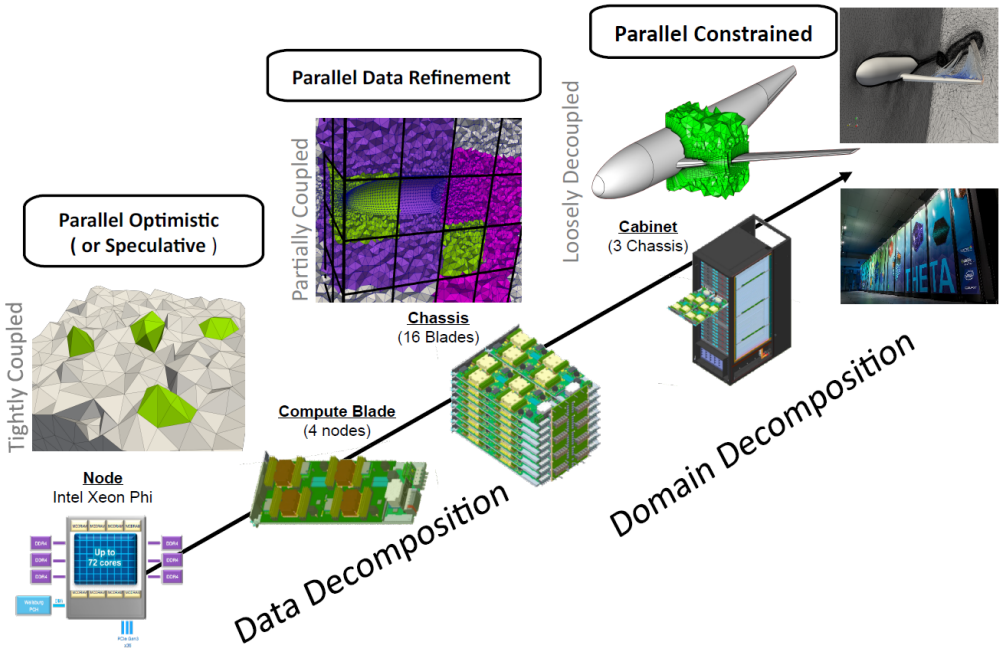


Fig. 1 The Telescopic Approach (adapted from [3, 7])

To maximize performance in such a framework, there are several attributes that one must consider when designing a parallel mesh generation software. The following attributes are critical for enhancing a software's adaptability to emerging high-performance computing architectures, thereby ensuring its longevity and reliability when exploiting new methods for concurrency:

1) **Stability** represents the level of quality of a mesh generated in parallel in comparison to the quality of a sequentially generated mesh. This quality is defined by the shape of individual elements (evaluated in an anisotropic metric field) and the number of elements within the mesh (fewer is better for the same level of metric

conformity).

2) **Robustness** is the ability of the parallel software to correctly and efficiently process any input data. The processing of data should not require operator intervention, as this is not only highly expensive for massively parallel computations, but most likely infeasible due to the large number of concurrently processed sub-problems.
3) **Scalability** is the ratio of the runtime of the best sequential implementation to the runtime of the parallel implementation, which should show an increase in speedup that is ideally proportional to the number of processing elements utilized. However, this speedup is limited by the inverse of the sequential fraction of the software, according to Amdhal's law [8]. Therefore, non-trivial stages of the computation must be parallelized if one is to leverage as much concurrency as possible in the current and emerging architectures that are designed to deliver million-to billion-way concurrency.
4) **Code re-use** suggests a modular design that utilizes mesh operations (such as point creation/insertion, edge swapping, point smoothing, etc.) from any sequential (or previously-built parallel) mesh generators without requiring significant updates to accommodate these new operations. Rewriting new parallel algorithms for every new meshing operation can be highly expensive in time investment due to the complexity and evolving functionality of such codes. The code re-use approach is only feasible if the software satisfies the reproducibility criterion.
5) **Reproducibility** [9] is a requirement that the mesh generation code, when executed with the same input, produce either identical results (termed Strong Reproducibility) or those of the same quality (Weak Reproducibility) under the following modes of execution: (i) continuous without restarts, and (ii) with restarts and reconstructions of the internal data structures. This is essential if generated mesh elements are re-processed by the software's meshing algorithm after data movement occurs between parallel processes.

In [3, 10], CDT3D addressed robustness and was shown to maintain stability and excellent scalability (with various test cases, including its performance within the context of a simulation pipeline on a single multicore node), while addressing code re-use in [11]. While presenting the foundational elements of this new distributed memory method, several challenges (and their resulting lessons) are also presented given that CDT3D is optimized for a shared memory architecture, and what additional measures were taken to enable the code's integration into the distributed memory method. Whereas CDT3D targets the chip level, the presented distributed memory method serves to exploit coarse-grain parallelism at the node level.

In the presented method, an initial mesh is data decomposed and subdomains are distributed amongst the nodes of an HPC cluster. Meshing operations within the shared memory code are designed to adopt a speculative execution model, enabling the strict adaptation of interior subdomain elements so that interface elements can be adapted in a separate step to maintain mesh conformity. Interface elements undergo several iterations of shifting (i.e., movement between parallel processes) so that they are adapted when their data dependencies are resolved (i.e., cavities of elements required to permit their adaptation are local). To aid in this endeavor, PREMA is utilized. As mentioned previously, this runtime system alleviates the burden of work scheduling and load balancing for distributed memory applications.

Current state-of-the-art HPC mesh generation methods utilize collective communication and global synchronization techniques, limited by traditional message passing between parallel processes [12–20]. These techniques have been shown to hinder potential scalability [10, 21, 22]. PREMA not only offers asynchronous message passing between processes but also enables asynchronous message passing between encapsulations of data (i.e., subdomains). This functionality is described in section III.B. Additionally, it offers a particularly useful "event" feature that aids in establishing data dependencies between subdomains, thus enabling "neighborhoods" of subdomains to work independently of each other in performing interface shifts and adaptation. This early implementation of the distributed memory method utilizes a master/worker model, where the master process is responsible for the creation of these PREMA events (designating which subdomains will send/receive data within their respective neighborhoods during each interface shift iteration). We follow the practice of first implementing our method with a centralized model (such as the method in [23] before its subsequent decentralization in [24]) to gauge its stability and ensure correctness with regards to the final mesh it generates. Although this centralized model contains implicit global synchronization, PREMA and CDT3D act as building blocks enabling the distributed memory CDT3D method to avoid the use of collective communication and to eventually avoid any global synchronization techniques. The completion of its decentralized version (where PREMA events will be created asynchronously amongst neighborhoods of subdomains) can potentially grant the distributed memory method better performance than that seen thus far with existing state-of-the-art HPC software. This is a future step and is outside the scope of this paper.

Preliminary results show that after several passes of interface shifts and adaptation, the presented method is able to produce meshes of comparable quality to those generated by the original shared memory CDT3D software. However, the

shared memory CDT3D method presents a challenge regarding its ability to re-process data that it generated in previous adaptations (a necessity given that interface elements are merged with elements already adapted during shifting). It does so successfully but not efficiently with regards to time. This phenomenon is investigated in section VI.D. These results echo the recurrent conclusion (also deduced from other distributed memory parallelization efforts of sequential and shared memory mesh generation methods [9, 25–27]) that a code cannot be simply integrated into a parallel framework as a black box if it was designed and developed independently of that framework. This suggests that the shared memory CDT3D method must undergo further re-design if it is to be integrated successfully into the Telescopic Approach, enabling the distributed memory method to achieve scalability. Local communication performance regarding non-meshing operations, however, suggests that if CDT3D can be successfully re-designed, the interface shift operation presents a potentially viable solution in achieving scalability for mesh adaptation when targeting configurations with large numbers of cores (building upon the shared memory CDT3D method which only utilized up to 40 cores in an earlier study [10]).

III. Background

A. Shared Memory Mesh Generation

The shared-memory CDT3D implements a tightly-coupled method and exploits fine-grain parallelism at the cavity level using data decomposition, targeting shared memory multicore nodes using multithreaded execution at the chip level. Multiple mesh operations (e.g., point insertion, edge/face swapping, etc.) are performed concurrently on different data by using fast atomic lock instructions to guarantee correctness. These locks are used to acquire the necessary dependencies for the corresponding operation. Failure to do so will result in unlocking any acquired resources (rollback) and attempting to apply an operator on a different set of data. This is the essence of the speculative execution model, which is to exploit parallelism "everywhere possible" from the beginning of refinement when there is no, or very coarse, tessellation (contrary to existing methods that require sequential preprocessing and are in some cases just as expensive as the parallel mesh refinement itself). The speculative execution model is implemented using the *separation of concerns* ideology [28] [11]. Functionality is separated from performance components with CDT3D as well. Meshing operations are abstracted as tasks, and these tasks are only performed when their corresponding dependencies are satisfied (i.e., successfully locked). Such abstractions provide easy interoperability with a low-level runtime system such as PREMA (discussed in more detail in section III.B). The distributed memory method currently utilizes the Pthread version of CDT3D as opposed to other backends offered by its tasking framework (details of additional execution backends can be found in [11]).

CDT3D offers metric-based anisotropic mesh adaptation, accepting an analytic or discrete metric field as input, and can be combined with Computer-Aided Design (CAD)-based information to accomplish adaptation [3]. The pipeline of operations for CDT3D's anisotropic mesh adaptation can be seen in Figure 2. While the mesh adaptation phase can improve the overall mesh quality, it should be noted that this phase focuses on satisfying spacing requirements (i.e., edge lengths of elements in the metric space). The quality improvement phase focuses on satisfying element shape requirements (i.e., improving element mean ratio). See section VI.A for details regarding how qualitative criteria are measured in the metric space for our evaluation.

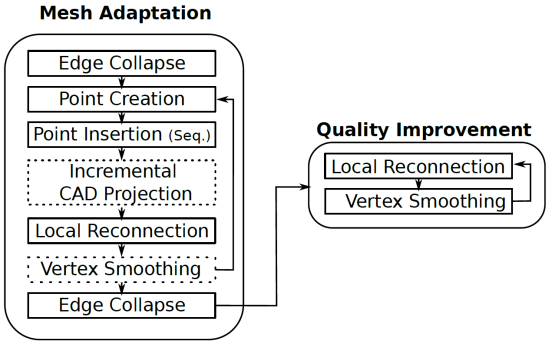


Fig. 2 CDT3D pipeline of anisotropic adaptive mesh generation [3]

CDT3D was compared with three other parallel anisotropic mesh adaptation methods used extensively within the industry [10]. The quantitative and qualitative results of each method were compared from testing on a benchmark created by the Unstructured Grid Adaptation Working Group (UGAWG) [29]. This benchmark served to evaluate adaptive mesh mechanics for analytic metric fields on planar and simple curved domains. In each case, CDT3D was shown to maintain stability of metric conformity (with both element shape size and edge length). It also showcased good performance when utilizing up to 40 cores on a single multicore node, exhibiting good weak scaling speedup and almost linear speedup amongst its strong scaling cases.

B. Parallel Runtime System

Message passing and data migration within the presented distributed memory method is handled by utilizing the Parallel Runtime Environment for Multicore Applications (PREMA) system [4–6]. This system provides work scheduling and load balancing on both shared and distributed memory architectures, alleviating the application developer of these responsibilities. PREMA introduces constructs called mobile objects, which are encapsulations of data (not necessarily residing in contiguous memory), and mobile pointers which are used to identify mobile objects within a global namespace. Interactions between data can be expressed as remote method invocations (handlers) between mobile objects rather than only between processes or threads. While non-conflicting handlers are executed concurrently across processing elements, PREMA offers the ability to utilize multiple hardware threads to share work within the context of individual handlers [6]. Due to the nature of adaptive applications, especially in the context of mesh generation, workload disparity is often witnessed amongst the mobile objects processed by handlers. This fine-grain parallelism allows for more efficient utilization of shared memory resources to help bridge this disparity in workload processing time.

PREMA offers distributed memory load balancing by monitoring work loads between ranks and performing migrations of mobile objects to available workers without interrupting execution. Communication and execution are separated into different threads to provide asynchronous message reception and instant computation execution at the arrival of new work requests. As stated previously, the mobile pointer construct allows method invocations to be made to mobile objects regardless of their location (potential migration to another rank). PREMA also provides the ability to establish dependencies between mobile objects and to execute user-defined events once all dependencies have been satisfied. This functionality becomes particularly useful when migrating cavity data needed for the adaptation of interface boundary elements within subdomains (discussed in more detail in section V).

IV. Related Work

The presented distributed memory method is motivated by past investigations of “black-box approaches," where parallelization was attempted for mesh generation programs (most of them were originally sequential) while making the least amount of modifications possible to their source code. One such approach is termed "functionality-first," which involves the parallelization of state-of-the-art mesh generation software that are fully functional and optimized for single-core architectures. Several studies addressing the viability of functionality-first black box approaches included VGRID [25], TetGen [9], and AFLR [26]. The effort regarding VGRID utilized a binary of the sequential software within a distributed memory method [25]. Consequently, its performance was hindered by VGRID process creation, file I/O, and data structure initialization. Additionally, its scalability was limited by the sequential design of VGRID and insufficient exploitable concurrency given by the domain decomposition method utilized. With regards to the parallelization of TetGen, TetGen itself failed the reproducibility criterion (it could not initialize its data structures based on a mesh that it itself had already generated) [9]. Although AFLR’s source code was integrated successfully into a distributed method and it satisfied the reproducibility criterion, this parallelization effort was hindered by the mesh quality within each subdomain being constrained by interfaces (subdomain boundaries, as an input boundary was required by AFLR to refine any domain) [26]. Another black-box approach focused on the integration of a shared memory method, called PODM [24], into a distributed memory framework [27]. PODM is a Delaunay-based mesh generation method (as opposed to CDT3D’s local reconnection-based method) that was not originally designed for execution within a distributed setting (or integration into the Telescopic Approach). This approach strictly utilized PODM as a black box and involved the migration of large amounts of data (entire subdomains) throughout execution in order to resolve data dependencies that were required to satisfy the Delaunay property. This overhead accounted for more than 50% of execution time, making this distributed approach 7x slower than the shared memory PODM when utilizing the same numbers of cores [27]. Evaluations of these black-box approaches exhibited the recurrent conclusion that if a code is not originally designed for scalability, it cannot be simply integrated into a parallel framework as a

black box. Rather than devoting significant amounts of time to redesigning such codes, these studies encourage the development of "scalability-first" approaches (those designed with scalability as the focus and functionality added as needed), if one wishes to leverage the maximum potential speedup offered by large-scale architectures.

A goal of the presented distributed memory method is to avoid collective communication, as this is not ideal for large-scale computing. The overhead of collective communication was reported in an extensive study involving numerous proxy applications within DoE's Exascale Computing Project (ECP) [21, 22]. The purpose of the ECP study is to understand communication patterns utilized by these applications and to identify where optimization efforts may be focused. It was observed that most of the applications spent more than 50% of their runtime in communication (as opposed to computation). While the majority of communication calls were primarily point-to-point (messages between individual processes), the amount of runtime spent in communication was dominated by collective calls (e.g. MPI_Allreduce, MPI_Alltoall, etc.).

Another observation of note is that none of the applications in the study utilized neighborhood collectives, a feature introduced in MPI 3.0 that permits collective communication calls within subgroups of processing elements [30]. This feature is designed to operate based on a process topology. Neighborhood collectives, in addition to several other MPI functions, are suggested as potential resources in helping drive forth optimizations for these exascale applications represented by their proxy counterparts.

The parallel meshing strategies utilized by the aforementioned state-of-the-art codes (that were compared to the shared memory CDT3D) all exhibit good speedup in the strong scaling and weak scaling cases presented in [10]. However, there are implicit global synchronization points in these codes that induce increasingly noticeable overhead when utilizing up to several hundred cores. Given the observations presented in the study of DoE's ECP, this overhead may be exacerbated when meshing billions of elements on much higher configurations of cores. For example, *Feflo.a*, a functionality-first, partially-coupled coarse-grained approach developed by INRIA [12], solves the interface problem by freezing interfaces during adaptation and then re-partitioning the domain to focus on adapting interface elements. This re-partitioning occurs only after all subdomains have completed adaptation and is repeated over several passes. This domain decomposition was reported to be one of the main parallel overheads. Other methods use different graph-based techniques for domain decomposition, also freezing interface elements during adaptation and then performing global re-partitioning steps over several iterations to adapt interface elements [13, 14, 31].

A scalability-first, partially-coupled coarse-grained approach developed by NASA, called *refine* [15], also re-partitions the domain at the end of each adaptation pass. Interior elements are adapted while interfaces remain frozen. The method then focuses on adapting interface elements while simultaneously performing communication to update neighboring subdomains of the changes to shared interface elements. Global identifiers are utilized to denote duplicate points between subdomains. Moreover, all-to-all communication occurs at the end of each adaptation pass. Each subdomain communicates with all other subdomains to ensure that each newly inserted grid point has a unique global identifier.

Two partially-coupled, coarse-grained methods, known as *Pragmatic* [16, 32] and *Omega_h* [17, 33], utilize a data decomposition technique where they color a dependency graph of the cavities targeted by mesh operations. Each method utilizes different techniques in handling interface elements however. Initially, interfaces remain frozen during adaptivity and a global re-partitioning step is performed at the end of each adaptation pass to then adapt those interfaces in *Pragmatic* [16]. In *Omega_h*, interface elements are replicated across subdomains so that for every cavity there exists a subdomain where mesh operations may be applied to the interface point's/element's full cavity only within that subdomain. A global synchronization step then removes duplicate data before obtaining a new dependency graph of cavities for the next adaptation pass.

Another partially-coupled, coarse-grained method called *FMDB* [18, 19] bases its data decomposition techniques on distributed databases. Duplicate data for interface points are stored on neighboring subdomains; moreover, particular subdomains are selected where mesh operations are applied to those interface elements. In a global synchronization step, the processors containing these subdomains communicate with those containing neighbors at the end of each adaptation pass regarding updates (while these interface elements remain frozen in the neighbors during their own adaptation). A global re-partitioning step is also used at the end of each adaptation pass to maintain load balancing.

The distributed memory method presented aims to avoid collective communication, and does not perform global re-partitioning. Additionally, it does not require global synchronization to update global identifiers for duplicate data between subdomains. It instead takes a similar approach to *EPIC* [20], a functionality-first, partially-coupled coarse-grained approach developed by Boeing. *EPIC* freezes interfaces during each adaptation pass and then all processes synchronize to shift the interface elements into the interiors of subdomains. A subset of mesh operations within *EPIC* also utilize multi-threading. With regards to distributed adaptive anisotropic 3D mesh generation (and to

the best of our knowledge), no other method applies a tightly-coupled speculative fine-grained approach, such as the shared memory CDT3D software, for the adaptation of individual subdomains. The presented distributed memory method performs an initial adaptation pass with frozen interfaces (fully utilizing the multithreaded speculative execution model for adapting interior elements within each subdomain). Neighborhoods of subdomains (enabled by PREMA's event functionality) then shift cavities of data needed to adapt those interface elements over several passes. Message passing is performed within these neighborhoods of subdomains, avoiding any all-to-all communication.

V. Distributed Memory Method

A. High-level Algorithm

Given its design and performance in stability on a single shared-memory node, the shared memory CDT3D (SMCDT3D) was abstracted as a library to be used in the adaptation of individual subdomains in the distributed memory method. A high-level overview of the distributed memory method (DMCDT3D) is shown in Algorithm 1. It essentially includes six steps, three of which are executed in a loop until all elements have undergone the mesh adaptation phase of the shared memory CDT3D (which includes operations that optimize both the spacing and shape measure of elements but primarily focuses on satisfying spacing requirements). The final step focuses on satisfying shape measure quality. The steps include: decomposition, interior adaptation of all subdomains, interface shift, interior adaptation of colored subdomains, a topology update of subdomain adjacency, and quality improvement.

Algorithm 1 High-level algorithm of early Distributed Memory CDT3D implementation

DMCDT3D(M_i, m, N)
Input: M_i is the initial mesh
m is the target metric
N is the number of subdomains
Output: Adapted mesh that conforms to the anisotropic metric field m

1: Perform data decomposition to create subdomains $S_1...S_N$
2: Distribute subdomains $S_1...S_N$ to processes
3: ADAPT_SUBDOMAIN($S_1...S_N$, $m_1...m_N$) ▷ // Adapt interior elements amongst subdomains
4: **while** ($\exists$ elements $\in S_1...S_N$ that have NOT undergone mesh adaptation) **do** ▷ // Adapt interface elements
5: $S_{S_1}...S_{S_n}$, $S_{R_1}...S_{R_k}$ = COLOR_SUBDOMAINS($S_1...S_N$) ▷ // Color subdomains to designate which will send or receive interface data
6: INTERFACE_SHIFT($S_1...S_N$)
7: ADAPT_SUBDOMAIN($S_{R_1}...S_{R_k}$, $m_{R_1}...m_{R_k}$)
8: UPDATE_TOPOLOGY($S_1...S_N$)
9: **end while**
10: QUALITY_IMPROVEMENT($S_1...S_N$, $m_1...m_N$) ▷ // Commence quality improvement over all subdomains' interior elements
11: TERMINATE()

B. Data Decomposition

Different methods of decomposition may be applied to the grid. The method of decomposition applied in the current implementation is called PQR, which uses a sorting-based method to partition elements into subdomains based on a boundary-conforming curvilinear coordinate system [34]. A connectivity graph partitioning heuristic is utilized, where the mesh itself is considered to be a Euclidean graph (the elements are vertices and their face-connected neighbors establish edges). This heuristic uses Euclidean metrics and minimizes the diameter of subdomains, delivering quasi-uniform partitions. Once all subdomains have been created, they are packed and distributed among processes using PREMA.

C. Interior Adaptation

After data decomposition, the interior elements within each subdomain are adapted. Whenever a subdomain undergoes adaptation, all of its interface elements are frozen. Let a mesh be partitioned into N subdomains $S_1 ... S_N$. If $S_1 \cap S_2 \neq \emptyset$ then S_1 and S_2 are neighbors. Let $I_{1,2} = S_1 \cap S_2$. All points in $I_{1,2}$ are interface points. Any edge, face, or tetrahedron defined by an interface point in $I_{1,2}$ is an interface edge, interface face, or interface tetrahedron, respectively. Any other subdomain which contains elements that are defined by a point in $I_{1,2}$ is also considered a neighbor of both S_1 and S_2 and will contain a copy of that same interface point.

Let $neighs$ be the set of all neighbor subdomains for S_1. The set of interior points in S_1 is defined as: $\forall points \in S_1 and \notin I_{1,neighs}$. Elements that are solely defined by interior points are permitted to undergo adaptation. Interface elements remain frozen to maintain conformity between subdomains. As mentioned in section III.A, some operations in CDT3D are designed to use a speculative execution model. Consequently, interface elements are frozen by locking interface vertices before any operation commences. An overview of each operation and how they are affected by interface vertices is described below.

1. Edge Collapse and Vertex Smoothing

The edge collapse operation is used within CDT3D as a pre-refinement and post-refinement operation. It serves to remove edges smaller than a target value. As a pre-refinement operation, coarsening a mesh before executing subsequent operations can lead to better quality in the final output mesh (demonstrated in [10]). As a post-refinement operation, it is used to remove short edges created during refinement/adaptation. The vertex smoothing operation moves a vertex incrementally along multiple directions of a search space to determine a position that optimizes the quality of all connected tetrahedra. Both the edge collapse and vertex smoothing operations follow the speculative execution model, described in section III.A. Each operation iterates through all vertices, attempting to lock each vertex and all its adjacent vertices (implicitly granting exclusive access to all adjacent tetrahedra). If any locks fail, all acquired resources are released and the method proceeds to the next vertex. See [3] for more details regarding the implementation of these operations. Figure 3 shows a 2D example of an edge and points that would not be allowed to undergo removal or smoothing due to interface vertices already being locked. If an edge contains an interface vertex, it cannot be removed. If a vertex is adjacent to an interface vertex, it cannot be smoothed.

It should be noted that the order in which operations are designed to be executed in the shared memory method should also be reflected in their order of execution in the distributed method. As stated previously, the edge collapse operation serves as a pre-refinement and post-refinement operation. If this operation is used by default during the mixed interior/interface adaptation of subdomains, elements that were already generated to satisfy spacing criteria will be repeatedly removed and re-generated over each interface shift iteration, wasting time and computational resources. When attempting to use a method like the shared memory CDT3D as a black box, one must be mindful of each operation's design/purpose and how they can be used appropriately in a distributed setting to maximize potential scalability. Edge collapse is only used as a pre-refinement operation during the interior adaptation phase (line 3 of Algorithm 1). It is used as a post-refinement operation only after the interface shifting phase has been completed and before the quality improvement phase commences (line 10 of Algorithm 1). Vertex smoothing can be utilized in all meshing phases of the distributed memory method; however, it is only used during the quality improvement phase in the current implementation (line 10 of Algorithm 1).

2. Local Reconnection

Local Reconnection includes four types of topological transformations, or flips, that include metric-based information to improve element quality. This operation also uses the speculative execution model. After assigning lists of tetrahedra to threads, each thread iterates through its assigned tetrahedra. The thread attempts to lock the vertices of a tetrahedron. For each of the tetrahedron's faces that have a neighboring tetrahedron, the thread attempts to lock the opposite vertex of that neighbor tetrahedron and attempts flips. If any locking fails, the thread either moves on to the next neighbor or releases any acquired resources and moves on to the next tetrahedron. For more details regarding this operation's implementation, see [3]. Figure 4 shows a 2D example of elements that would be permitted to undergo flips. Any combination of elements are permitted to undergo local reconnection iff none of those elements are interface elements (i.e., defined by an interface vertex). Local Reconnection is utilized in all meshing phases of the distributed memory method - interior adaptation, interface shift, and quality improvement (lines 3, 7, and 10 of Algorithm 1, respectively).

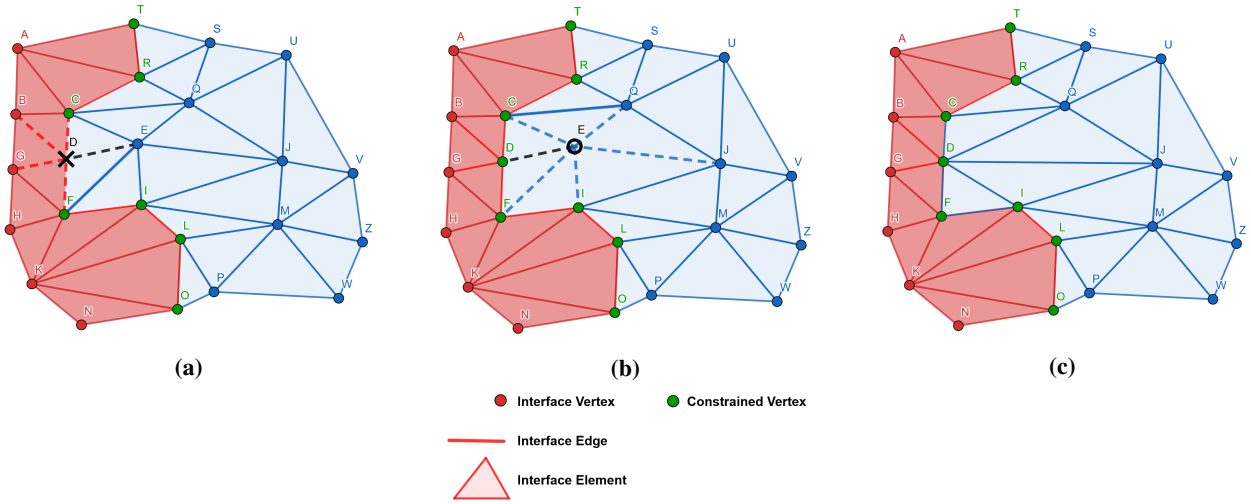


Fig. 3 Any edge that does not define an interface element, or contain an interface vertex, may be removed (i.e., any edge that is blue). However, the primary vertex during locking of the edge collapse operation must be considered. The dotted lines are edge connections to points that are dependencies of the primary vertex (those that the method will attempt to lock). In (a), the black dotted edge DE cannot be removed because the primary vertex is D, which is adjacent to interface vertices B and G, failing to lock them. In (b), the primary vertex is E, and all adjacent vertices are successfully locked. Edge DE can therefore be removed, resulting in (c). Because vertex smoothing uses the same locking mechanisms, a similar principle applies. No interface vertices may undergo smoothing because they are already locked. No constrained vertex (those in green) may undergo smoothing because the smoothing operation would fail to lock all their adjacent vertices (i.e., those that are interface vertices).

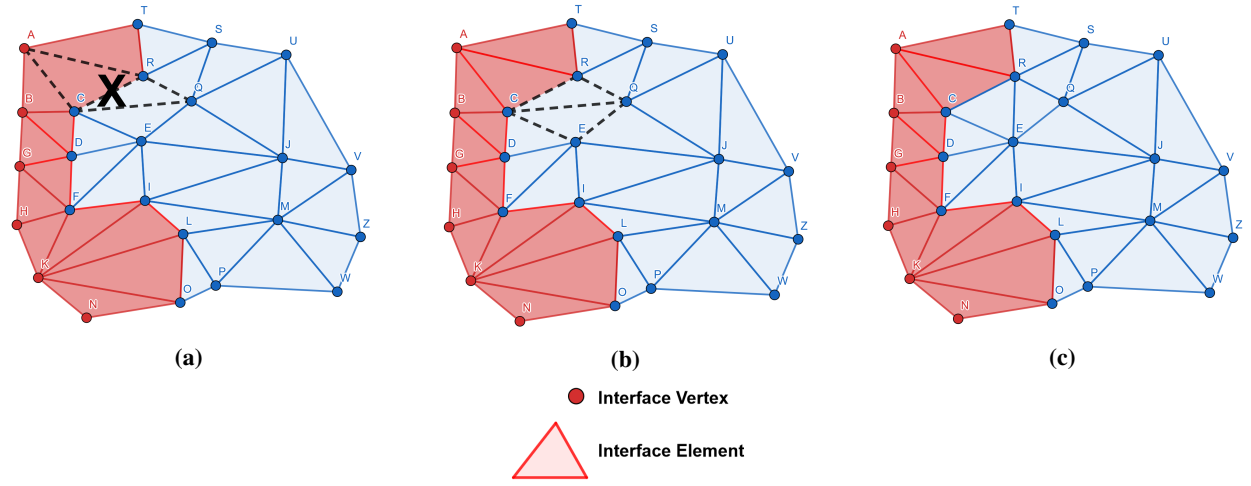


Fig. 4 Any combination of elements are permitted to undergo local reconnection iff none of those elements are interface elements (i.e., those in red that are defined by an interface vertex). As an example, the black dotted lines represent elements that the method attempts to lock in order to perform a flip. In (a), elements ARC and RQC cannot undergo swapping. Element ARC cannot be locked by the local reconnection operation due to point A already being locked (since it is an interface vertex). In (b), elements RQC and CQE are permitted to undergo swapping since neither are defined by interface points. (c) is the result of the flip, creating elements CRE and RQE.

3. Point Creation/Insertion

Similar to the local reconnection operation, lists of tetrahedra are assigned to threads. Each thread iterates its list of tetrahedra that do not satisfy edge length criteria in the metric space. A centroid-based point creation technique is utilized to create candidate points. If a point satisfies all proximity checks and encroachment rules (e.g., not too close to an existing point or boundary face, etc.), it is accepted and inserted into the mesh. For more details regarding the implementation of these operations, see [3].

It should be noted that point creation/insertion on elements adjacent to interface elements inhibits grid generation convergence. We have found that CDT3D will continuously attempt to improve elements adjacent to the interface by inserting points over numerous iterations until crashing. The method will struggle to truly satisfy spacing requirements within the metric space and will inadvertently create slivers (i.e., elements with a volume near zero). Therefore, a buffer zone must be established around locked (interface) elements to reach convergence. Point creation/insertion is prohibited for any interface element or element within this buffer zone. Elements within the buffer zone include those defined by a vertex that is also shared by an interface element. Figure 5 shows a 2D example of constrained elements that cannot undergo point creation/insertion because they are adjacent to interface elements. Constraining elements that surround interfaces is a necessity also seen in the state-of-the-art method *Feflo.a* [12], where surrounding cavity elements must be included in the adaptation of interface elements to produce good quality elements. Point creation/insertion are utilized in the interior adaptation and interface shift phases of the distributed memory method (lines 3 and 7 of Algorithm 1, respectively).

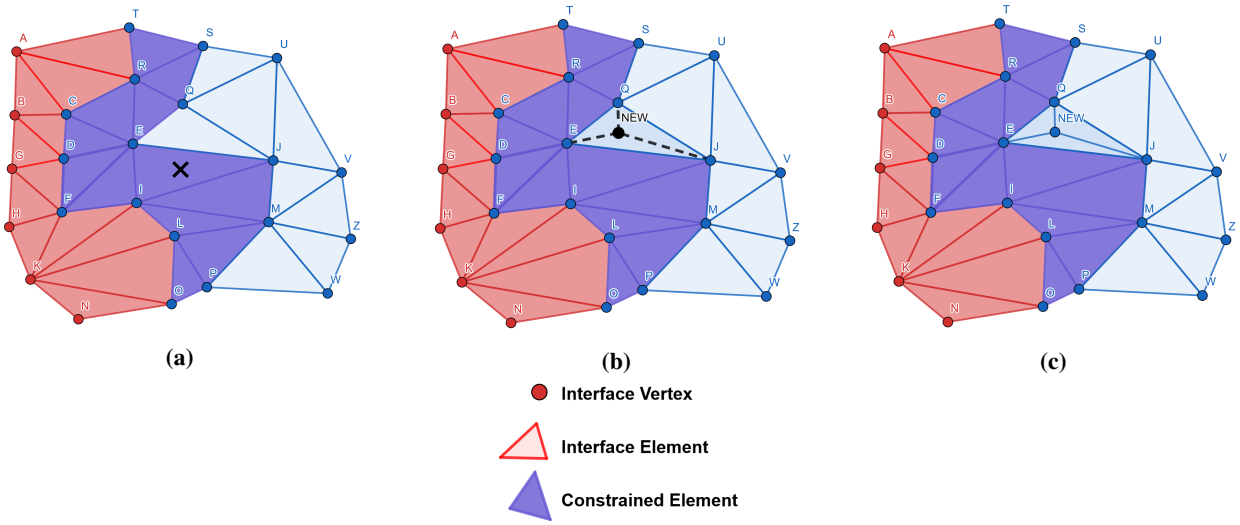


Fig. 5 Interface elements (those in red) cannot undergo point creation/insertion. Constrained elements (those in purple) share a point with an interface element and therefore cannot undergo point creation/insertion. (a) shows the point creation operation failing to generate any candidate points for element EJI because it is constrained due to point I being shared with elements IKF and ILK. In (b), point creation is permitted for element QJE, resulting in the creation of three new elements in (c).

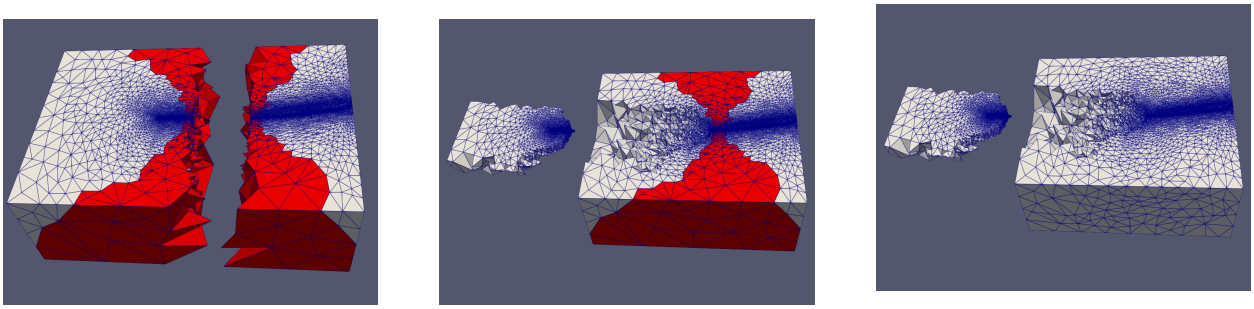
D. Interface Shift & Mixed Interior/Interface Adaptation

Due to how interface elements can affect the overall quality of the final mesh (as seen in the study regarding the parallelization of AFLR) [26], they are shifted to the interiors of subdomains over several iterations so that they may undergo adaptation. This mixed interior/interface (MII) adaptation phase begins after all subdomains have completed the stage of interior adaptation (line 3 in Algorithm 1). Subdomain adjacency can be considered as an undirected graph, where a subdomain is a vertex and an edge is a neighbor connection between two subdomains that share an interface point. Many subdomains within this graph are selected, or "colored," to receive data while particular neighbors are selected to send their corresponding interface data to those that are colored. Subdomains colored to receive data are prioritized by the number of elements that have not undergone mesh adaptation (i.e., interface elements and those constrained by them). All surrounding neighbors of a colored subdomain are designated to send data only to that

subdomain. Due to this method of coloring, it is possible for some subdomains to go uncolored. Once an initial pass of coloring is complete, remaining subdomains are colored even if they only have one neighbor from which they may receive interface data (that has not already been designated to send to another colored subdomain). This allows the distributed method to exploit as much parallelism as possible at the coarse-grained level. The interface data sent consists not only of the interface elements themselves but also their corresponding cavities plus several layers of elements that are required to successfully permit their adaptation in each meshing operation. Consider a single tetrahedron as a single layer. The set containing each tetrahedron connected to each of its faces are considered to be a second layer, and all of the tetrahedra connected to each of their faces are considered to be the third layer, and so on.

PREMA's aforementioned event feature is utilized, where colored subdomains designate their neighbors as dependencies. Recall that PREMA allows subdomains (i.e., mobile objects) to communicate amongst themselves (rather than limiting communication specifically to processes). Once an event activates (i.e., all neighbor subdomains have completed their own interior adaptation and/or adjacency update communication), all neighbor subdomains gather interface data, send it to their colored counterpart, and the receiver scatters the interface data. Once all data has been received and scattered, the colored subdomain initiates interior adaptation. Those elements that were considered interfaces in the previous iteration are now considered to be interior elements due to this "padding" of cavities and layers of additional elements. These elements combined with the original interior elements will all now be processed for adaptation. PREMA allows for multiple events to be created and executed simultaneously, where subdomains communicate amongst themselves within neighborhoods to commence interface shifts and adaptation. To verify the stability and correctness of the distributed memory method, the current implementation utilizes a master/worker model, where the master process colors subdomains and sets up PREMA events so that the corresponding neighborhoods can commence their interface shifts. In the future, the master/worker model will be removed and PREMA events will be created amongst subdomains as they color themselves within neighborhoods, enabling the method to become fully asynchronous.

Figure 6 shows an example of the interface shift process with a data decomposition of a delta wing geometry being adapted at a target complexity of 500,000. For simplicity and ease of understanding, there are two subdomains which each contain approximately 2.3 million elements by the end of interior adaptation. About 200,000 elements are shifted (i.e., sent) from the left subdomain to the right subdomain. The interface elements and those constrained by the interface elements (red) are then adapted, generating about 400,000 additional elements.



(a) Data Decomposition **(b) Interface Shift** **(c) Mixed Interior/Interface Adaptation**

Fig. 6 The interface shift and adaptation process is shown on a data decomposition of a delta wing geometry. Red elements contain edges that do not conform to the target metric, as they are affected by the frozen interface. (a) shows a data decomposition where two subdomains have each undergone interior adaptation. (b) shows the transfer of interface elements (and corresponding cavities/layers) from the left subdomain to the right subdomain. (c) shows the adaptation of those elements that were previously interface elements in both subdomains.

As in the shared memory method (which requires multiple grid generation passes of adaptation to obtain satisfactory grid quality, i.e. metric conformity) [2, 3], several iterations of interface shift and adaptation passes are required to attain sufficient grid quality in the distributed memory method. Before an iteration of MII adaptation can commence for a set of subdomains, the graph of subdomain adjacency must be up-to-date to ensure correct communication between subdomains. After a gather/scatter operation, subdomain adjacency may change (i.e., loss of adjacency or new adjacency). Each subdomain must communicate with its prospective neighbors to verify which of them are still neighbors, which are new neighbors (if any), and which are no longer neighbors. The distributed memory method

performs this communication between subdomains asynchronously while other subdomains may still be undergoing adaptation. This subdomain adjacency update algorithm can be seen in Algorithm 2, providing a detailed look into line 8 of Algorithm 1. All subdomains must complete all adjacency update communication before the master process commences a subsequent interface shift iteration (this is the only point of implicit global synchronization for this early implementation of the distributed memory method). Once the point creation operation of the mesh adaptation phase has been applied to all elements that were considered to be interface elements during the interior adaptation phase, interface shifting ends. Basing the completion of the interface shift phase specifically on the point creation operation is explained in section V.D.2 (with regards to a modification that was made to the shared memory method to enhance the distributed method's potential scalability). Quality Improvement finally commences over each subdomain (which initially calls the post-refinement edge collapse operation), the final mesh is output (if specified), and the distributed memory method terminates.

To exploit parallelism during the gather, scatter, and subdomain adjacency update operations, OpenMP constructs are utilized. For example, a `#pragma omp parallel for` construct is used when iterating and comparing sets of interface points during the topology update. Additionally, when data structures are converted from complex types in the shared memory method into simpler types for packing/sending (or vice-versa), similar constructs are used. The necessity of this conversion is described in section V.E.

1. Maintaining Simple Connectivity Amongst Subdomains

An additional challenge regarding the interface shift and MII adaptation involves maintaining simple connectivity within each subdomain (i.e. ensuring that no tetrahedron, or partition of tetrahedra, are loosely-connected, sharing only a point or an edge, to other tetrahedra or partitions within that subdomain). Every tetrahedron must share a face with at least one other tetrahedron. All tetrahedra within a subdomain can be considered as an undirected graph, where a tetrahedron is a vertex and an edge denotes that two tetrahedra share a face. Through edge traversal, if every vertex of the undirected graph can reach every other vertex, then the subdomain is simply connected. Each subdomain must be simply connected given that the shared memory software is designed to process only manifold geometries.

The notion of gathering layers of elements based on face-connectivity vs. point-connectivity must be considered. With face connectivity, layer one (from which more layers are gathered) is defined as the set of elements that each share a face with an element in the receiving subdomain. Layer one for point connectivity would include those that may only share a point with an element in the receiving subdomain. Gathering based on face-connectivity helps to ensure simple connectivity (although it doesn't guarantee it); however, gathering based on point-connectivity ensures that all mesh operations will have the needed cavities for interface elements (for example, recall that vertex smoothing operates on points and edge connections rather than tetrahedra and face connections). Gathering based on point-connectivity collects more elements, giving the shared memory method more flexibility to achieve adaptation, allowing the distributed memory method to satisfy spacing requirements quicker (as opposed to performing more interface shifts and iterations of MII adaptation). Consequently, gathering based on edge connectivity is not considered, as this would alienate those elements that are only point-connected to the receiving subdomain (again requiring the method to later perform additional interface shifts). To ensure simply connected volumes amongst subdomains, the distributed memory method adheres to the following:

1) After data decomposition and before subdomains are distributed, each subdomain is checked and possibly modified to ensure simple connectivity. If any loosely-connected partitions are found, they are re-assigned to another subdomain that contains an element with a matching face.
2) Subdomains are colored during the interface shift phase based on face connectivity. If a neighbor subdomain is connected to another by only point(s), the neighbor will not be selected to send those point-connected data, as this can invalidate simple connectivity for the resulting volume once scattering is complete.
3) While subdomains are colored based on face-connectivity, the initial layer of elements (from which more are gathered) are those that are point-connected to the receiving subdomain. As stated previously, this allows more elements to be collected and sent, reducing the number of shift iterations required. Subsequent layers of elements are gathered only based on face-connectivity, as described in section V.D. Due to the fact that the gather operation begins with a simply connected subdomain, and because there is at least one element which is face-connected to the receiving subdomain (guaranteed by the coloring), we can ensure that the data sent to the receiving subdomain is simply connected.
4) After ensuring that the set of elements gathered are simply connected, the set of elements remaining in the subdomain must be checked for simple connectivity. If any loosely-connected partitions are found, the partition

Algorithm 2 Algorithm for updating the topology of subdomain adjacency through asynchronous communication

SUBDOMAIN_TOPOLOGY_COMMUNICATION(S) - executed for mobile object S
Input: S is a subdomain

```
1: S->busy = true
2: Commence gather/scatter operation for interface data depending on color of S
3: neighbors = neighbor subdomains of S        ▷ // immediate neighbors if S is sender; neighbors of neighbors if S is receiver
4: S->busy = false
5: S->numNeighborsResponded = number of subdomains in neighbors list
6: neighbors_already_checked = empty list
7: if S->adjacency_check_queue is NOT empty then
8:     foreach subdomain ∈ adjacency_check_queue do
9:         Pop data from adjacency_check_queue
10:        Update adjacency status between subdomain and S
11:        RESPONSE(subdomain, S)
12:        S->numNeighborsResponded = S->numNeighborsResponded – 1
13:        neighbors_already_checked = neighbors_already_checked + subdomain
14:    end for
15: end if
16: foreach neigh ∈ (neighbors – neighbors_already_checked) do
17:    UPDATE_ADJACENCY(S, neigh)
18: end for
19: TERMINATE()
```

UPDATE_ADJACENCY(subdomain1, subdomain2) - executed for mobile object subdomain2

```
1: if subdomain2->busy == true then
2:     Insert interface data into subdomain2->adjacency_check_queue
3: else
4:     Update adjacency status between subdomain1 and subdomain2
5:     if there is new adjacency between subdomain1 and subdomain2 then
6:         if subdomain2->numNeighborsResponded == 0 then
7:             Send updated adjacency status to master process
8:         end if
9:     end if
10:    RESPONSE(subdomain1, subdomain2)
11: end if
```

RESPONSE(subdomain1, subdomain2) - executed for mobile object subdomain1

```
1: Update adjacency status between subdomain1 and subdomain2
2: subdomain1->numNeighborsResponded = subdomain1->numNeighborsResponded – 1
3: if subdomain1->numNeighborsResponded == 0 then
4:     Send updated adjacency and completion status to master process
5: end if
```

with the largest number of elements are designated to remain in the subdomain while the other partitions are designated to be sent with those already gathered. If a partition is loosely-connected to those elements remaining, it must be simply-connected to those gathered because the gather operation began with a simply connected subdomain.

5) Given that the receiving subdomain is only receiving simply connected sets of elements from those neighbors that are face-connected to it, this guarantees that it will remain simply connected after the scatter operation is complete.

2. Pseudo-active Element Modification

The shared memory CDT3D method denotes elements as "active" if they do not satisfy qualitative criteria. These elements become "inactive" when they do. CDT3D targets active elements when applying mesh operations. We have observed that among all the geometry cases with which CDT3D has been tested [3, 10], approximately 20-30% of elements remain active by the end of adaptation. During adaptation, the method attempts to improve the quality of these elements to reach certain qualitative thresholds. If little improvement is observed after several grid generation passes, the method terminates. Despite this phenomenon, CDT3D was shown to produce meshes with qualitative results that are comparable to those produced by state-of-the-art methods [10]. CDT3D was also tested and shown to function well when coupled with a solver as part of a CFD simulation pipeline, accurately capturing features of underlying simulations and occupying a small fraction of the pipeline's runtime [3].

CDT3D was designed with the assumption that the input geometry does not satisfy qualitative criteria. This design assumption is a problem for the distributed method due to the mixed interior/interface adaptation. Elements that may have still been active by the end of adaptation during the interior adaptation phase (or an interface shift iteration) will cause CDT3D to re-apply mesh operations and further attempt to improve these elements during a subsequent interface shift iteration. To maximize potential scalability, only elements to which mesh operations have not been applied (i.e., interface and constrained elements) should undergo adaptation.

To remedy this problem for the distributed method, the shared memory method was modified to acknowledge if elements are classified as "pseudo-active" or "pseudo-inactive." If an element is deactivated (made inactive) by the method, it is also made pseudo-inactive. If the point creation operation is applied to an element, it is made pseudo-inactive (even if the operation failed to create any candidate points that satisfy spacing criteria and the element remains active). If point creation skips an element (because it is locked, i.e., it is an interface element or is constrained by its proximity to an interface element), then that element is pseudo-active. We use the point creation operation to designate elements as pseudo-active because (1) its goal is to satisfy spacing requirements (which is the goal of the interface shift phase), (2) it is an operation that is always used in mixed interior/interface adaptation, and (3) because it is only applied to active elements (therefore, elements that become inactive will never be made pseudo-active again). Additional information that designates elements as either pseudo-active or pseudo-inactive is included when gathering and scattering data during an interface shift. When jumpstarting the shared memory method for a subdomain, if an element is pseudo-inactive, it is deactivated (made inactive) before any mesh operations are executed. With this modification, we attempt to direct CDT3D to focus on applying mesh operations only to elements that have not yet undergone adaptation (i.e., those that were previously interface/constrained elements and are pseudo-active). This pseudo-active modification allows the interface shift phase (loop in Algorithm V.A) to eventually exit. While this modification improved the scalability of the distributed method, it has not fully remedied the problem of CDT3D spending unnecessary time processing elements that have already undergone adaptation. This "redundancy" phenomenon is explored in more detail in section VI.D.

E. Distributed Data Structures

One should identify and encapsulate only data that is essential for jumpstarting the shared memory method in a distributed setting. There is a clean separation between the shared memory code and distributed code. The shared memory code itself is not distributed-aware. It is rather utilized as a library, where a number of adaptation operations can be executed on the interior of each subdomain while keeping interface elements fixed. Therefore, information pertaining to mesh elements within each subdomain must be extracted as input for the shared memory method. Simultaneously, the distributed memory method must maintain global identifiers specifying duplicate data between subdomains. Duplicate data are specified using global identifiers similar to the approach in [35]. These global identifiers are essential when receiving and scattering interface data, as they are used to identify both face and point connections between subdomains. Every vertex in the grid is assigned a subdomain id (during decomposition) and an integer id local to that particular subdomain. Decomposition and this assignment of data occurs on a single node, and subdomains are not packed/sent to

other nodes of the HPC cluster until all data assignment (and possible modification for simple connectivity) is completed. During assignment, whichever subdomain encounters the point first is considered its owner, while others store this point as a duplicate and its global identifier (subdomain id and local id) information. Interface points remain untouched throughout adaptation. Any newly created points are assigned new global identifiers (based on the number of current local ids for that subdomain) at the end of adaptation. If these points are gathered and sent to a neighbor during an interface shift, their global identifiers are sent with them so that the neighbor will also have this information.

The distributed memory CDT3D method is implemented in the C++ programming language and as such, the standard library **memcpy** function is utilized for the packing/unpacking of data during migration. It is preferable to organize subdomain data into the simplest data structures possible (arrays, plain old data types, etc.) to make the utilization of this function seamless; otherwise, careful attention must be given to more complex data structures, as they are more likely to induce memory errors if not handled correctly (e.g. container types, dynamically allocated pointers, etc.). These simple distributed memory method data structures are converted into the complex data structures utilized by the shared memory code (linked lists of dynamically allocated pointers to class objects representative of mesh elements, for example) when jump-starting a subdomain for adaptation, and vice-versa after a subdomain has completed adaptation (for an upcoming interface shift).

F. Requirements for the Distributed Method's Design

The following requirements summarize the lessons learned in needing to re-design the shared memory CDT3D code and understanding how to enable the integration of such a code into the distributed memory method:

1) Operations should be designed to execute successfully based only on subdomain input data provided (without assuming access to the entire input domain). Some data structures within the shared memory method were originally initialized based on this assumption. Their initialization has been modified to only account for what is needed for that particular subdomain.
2) If a subdomain becomes larger or smaller throughout program execution, any accompanying data structure(s) utilized within the shared memory method must also be updated before any subsequent processing of that subdomain, e.g., gather operation, scatter operation, or MII adaptation.
3) All operations should check if elements are locked, adhering to the mechanisms used by the speculative execution model, thus allowing the precursory freezing of specific data without needing to further modify each operation to process a new input type (interface data). For example, CDT3D includes a routine that iterates over all tetrahedra and attempts to remove slivers from the external boundary of the domain. Originally, this method assumed that if a tetrahedron contained a face with no neighbor tetrahedron attached, that that face is a boundary face. When applied to a subdomain, this face could be an interface face (of which the shared memory method is not aware). The routine has been modified to simply check if any of the face's points are locked. If so, it moves on to the next tetrahedron.
4) Point creation/insertion on elements adjacent to interface elements inhibits grid generation convergence; therefore, a buffer zone must be established around locked (interface) elements to reach convergence.
5) The order in which operations are designed to be executed in the shared memory method should also be reflected in their order of execution in the distributed method. The impact of the order of operations is observed in our evaluation, in combination with the pseudo-active modification that attempts to apply operations only to elements that need it (i.e., interface) so that the method does not waste time processing interior elements that have already undergone adaptation.

VI. Preliminary Results

A. Grid Adaptation Method

The same grid adaptation method utilized in [10], for the shared memory CDT3D's evaluation and comparison to other parallel meshing strategies, is utilized here for a progress and viability evaluation of the distributed memory method. The goal is to adapt a given grid so that it conforms to an anisotropic metric field M. A comprehensive introduction to the definition and properties of the metric tensor field is provided in [36]. The complexity C of a continuous metric field $\mathcal{M}$ is defined as:

$$C(\mathcal{M}) = \int_{\Omega} \sqrt{det(\mathcal{M}(x))} dx. \tag{1}$$

Complexity on the discrete grid is computed by sampling $\mathcal{M}$ at each vertex i as the discrete metric field M,

$$C(M) = \sum_{i=1}^{N} \sqrt{det(M_i)} V_i, \tag{2}$$

where V_i is the volume of the Voronoi dual surrounding each node. The complexity of a grid is known to have a linear dependency with respect to the number of points and tetrahedra, shown theoretically in [36] and experimentally verified in [37][38]. The number of vertices are approximately $2C$ while the number of tetrahedra are approximately $12C$. As shown in [36][10], scaling the complexity of a metric can generate the same relative distribution of element density and shape over a uniformly refined grid compared to the original complexity. The metric tensor M_{C_r} that corresponds to the target complexity C_r is evaluated by [36]:

$$M_{C_r} = \left(\frac{C_r}{C(M)} \right)^{\frac{2}{3}} M, \tag{3}$$

where M is the metric tensor before scaling and C(M) is the complexity of the discrete metric before scaling.

In order to evaluate the progress made in the distributed memory method's implementation, quantitative results are examined with respect to overhead incurred by the interface shift operation, as opposed to alternative methods which use global synchronization and re-partitioning to process interface elements during mesh generation. Qualitative results are examined with respect to metric conformity of the adapted mesh. These qualitative measures described below are the same as those used by the Unstructured Grid Adaptation Working Group*. The adapted meshes of the early distributed memory method are compared to those of the shared memory method in order to verify the viability of performing interface shifts in the distributed memory method and its impact on the method's stability.

The aim of metric conformity is the creation of a unit grid, where edges are unit-length and elements are unit-volume with respect to the target metric. For calculating edge length, we adopted the same definition that appears in [39]. For two vertices a and b, an edge length in the metric L_e can be evaluated using:

$$\begin{aligned} L_e &= \begin{cases} \frac{L_a - L_b}{log(L_a/L_b)} & |L_a - L_b| > 0.001 \\ \frac{L_a + L_b}{2} & otherwise \end{cases} \\ L_a &= (v_e^T M_a v_e)^{\frac{1}{2}}, L_b = (v_e^T M_b v_e)^{\frac{1}{2}} \end{aligned} \tag{4}$$

and an element mean ratio shape measure can be approximated in the discrete metric as:

$$Q_k = \frac{36}{3^{1/3}} \frac{\left(|k| \sqrt{det(M_{mean})} \right)^{\frac{2}{3}}}{\sum_{e \epsilon L} v_e^T M_{mean} v_e}, \tag{5}$$

where v is a vertex of element k and M_{mean} is the interpolated metric tensor evaluated at the centroid of element k. Since the goal is to create edges that are unit-length, edges with length above or below one are considered to be sub-optimal. The measure for mean ratio is bounded between zero and one since it is normalized by the volume of an equilateral element. One is the optimal quality for an element's mean ratio shape.

B. Experimental Setup

PREMA, shared memory CDT3D, and the early distributed memory CDT3D codes were all compiled using the GNU GCC 7.5.0 and Open MPI 3.1.4 compilers. Data were collected on Old Dominion University's Wahab cluster using dual socket nodes that each featured two Intel® Xeon® Gold 6148 CPUs @ 2.40 GHz (20 slots) and 384GB of memory. Each run in the following experiments was executed five times and the results were averaged. With regards to data reported for the execution of the distributed memory method, when using configurations of cores between 1 and 32, a single node is allocated and the corresponding number of cores are allocated on that node. When utilizing more than 32 cores, nodes are allocated with 32 cores each. With each configuration (except that of 1 core), 1 core is utilized for PREMA's communication per PREMA process (to allow for asynchronous communication and computation) while the remainder are utilized by the distributed memory method's operations (CDT3D mesh generation, gather/scatter, etc.). For example, a configuration of 256 cores means that 8 nodes are allocated with 32 cores each. 1 core is utilized for PREMA's communication and the remaining 31 are utilized by the respective operations on each node.

*https://ugawg.github.io/

C. Delta Wing Geometry

Figure 7 shows a delta wing geometry made of planar facets. Its multiscale metric [40] is constructed based on the Mach field of a subsonic laminar flow. The input grid is adapted from a complexity of 50,000 and is scaled to a complexity of 10 million (generating approximately 100 million elements) for both a qualitative and quantitative evaluation. In our evaluation, we consider elements with a mean ratio shape measure below 0.1 to be of poor quality. It is common practice in literature to apply mesh operations that target elements below this threshold [10, 29]. Elements of optimal quality have a mean ratio shape measure of 1 and contain edges that are unit-length, i.e. of length 1. Because both the shared memory and distributed memory methods generate final meshes with a small number of poor quality elements (with mean ratio shape measures below 0.1) when adapting the delta wing geometry at 10 million complexity, we also provide a qualitative evaluation when adapting the grid to a complexity of 500,000 (generating approximately 5 million elements). This case was also utilized in the aforementioned shared memory CDT3D evaluation study [10]. Details of the verification of the delta wing/grid adaptation process is provided in [41].

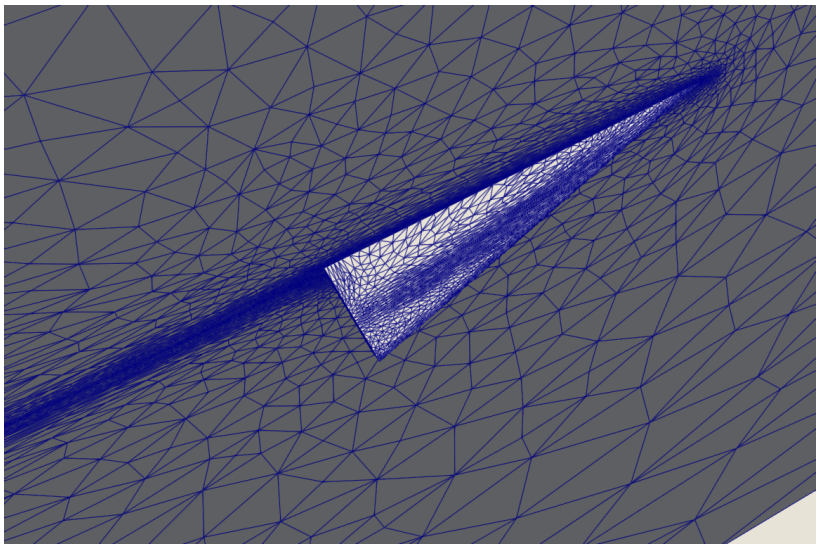


Fig. 7 Delta wing with multiscale metric derived from a laminar flow and a complexity of 50,000

1. Qualitative Results

There are factors that affect the distributed memory method's output grid's metric conformity and scalability, such as the method of decomposition and the number of interface shift iterations. The optimal settings of these heuristics vary between different geometries and require careful consideration when meshing larger geometries. After extensive testing, the optimal settings for adapting the delta wing geometry at 10 million complexity with the distributed memory CDT3D method was determined to be a PQR decomposition of 16 subdomains across the x-axis (strip decomposition) and 12 interface shift iterations. This decomposition is shown in Figure 8, where each color represents a subdomain to which an element belongs. Metric conformity, characterized by element shape measure and edge length histograms, for the output meshes generated by the shared memory CDT3D and distributed memory method are compared in Figures 9 and 10. Figures 11 and 12 show the mean ratio and edge length histograms for the delta wing geometry adapted at 500,000 complexity (using the same data decomposition configuration and number of interface shift iterations). It can be seen in both the linear and log scales that the meshes generated by DMCDT3D exhibit good overall quality similar to that generated by the original shared memory method. While both methods generate a final mesh with a few low quality elements (with mean ratio less than 0.1) in the case adapted at 10 million complexity, both of their meshes have a lower bound of 0.1 in the mean ratio measure for the case adapted at 500,000 complexity. In both cases, the methods exhibit similar distributions of edge length measures.

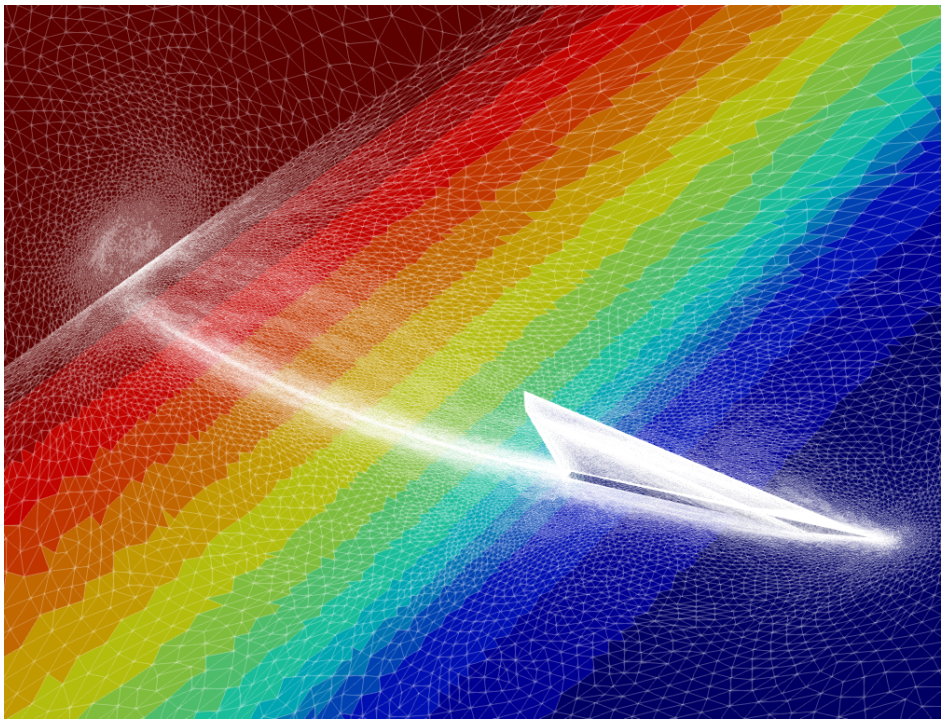


Fig. 8 Delta wing geometry decomposed across the x-axis with 16 subdomains using PQR - each color represents a different subdomain to which elements have been assigned

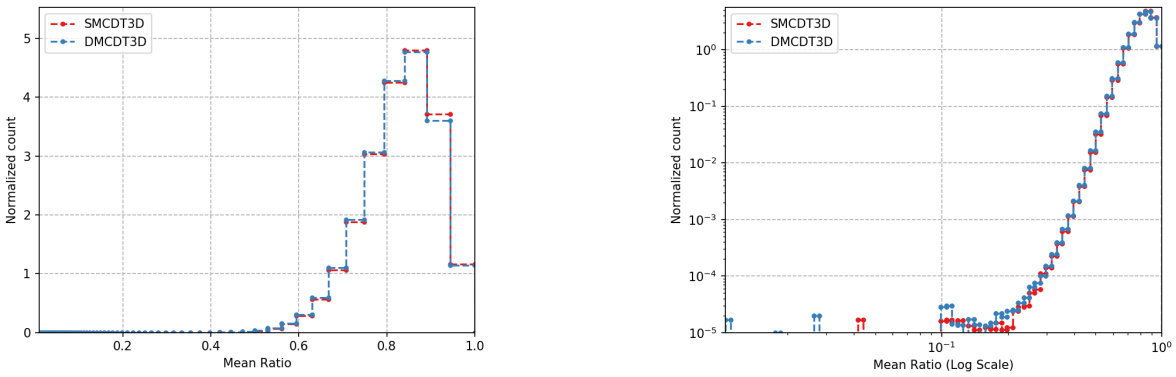


Fig. 9 Comparison of the mean ratio of elements within the meshes generated by the shared memory CDT3D method (SMCDT3D) and the distributed memory CDT3D method (DMCDT3D) for the delta wing geometry at 10 million complexity in linear and logarithmic scales

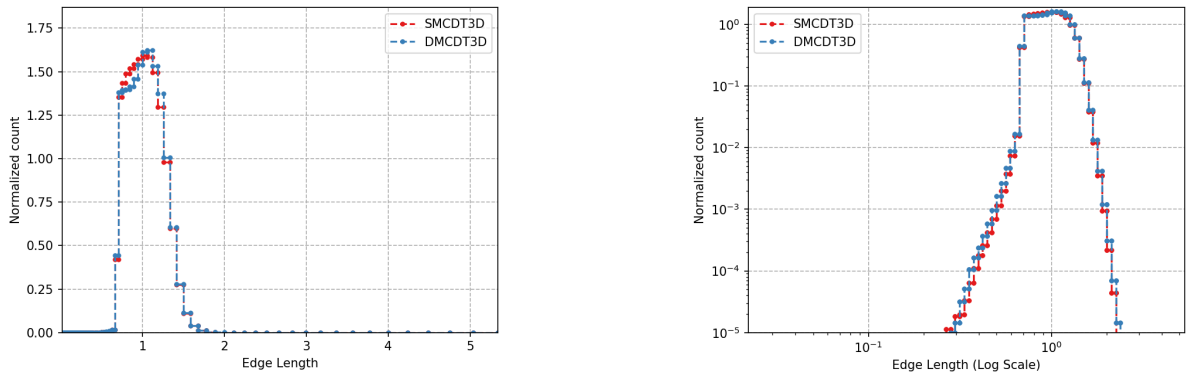


Fig. 10 Comparison of the edge lengths of elements within the meshes generated by the shared memory CDT3D method (SMCDT3D) and the distributed memory CDT3D method (DMCDT3D) for the delta wing geometry at 10 million complexity in linear and logarithmic scales

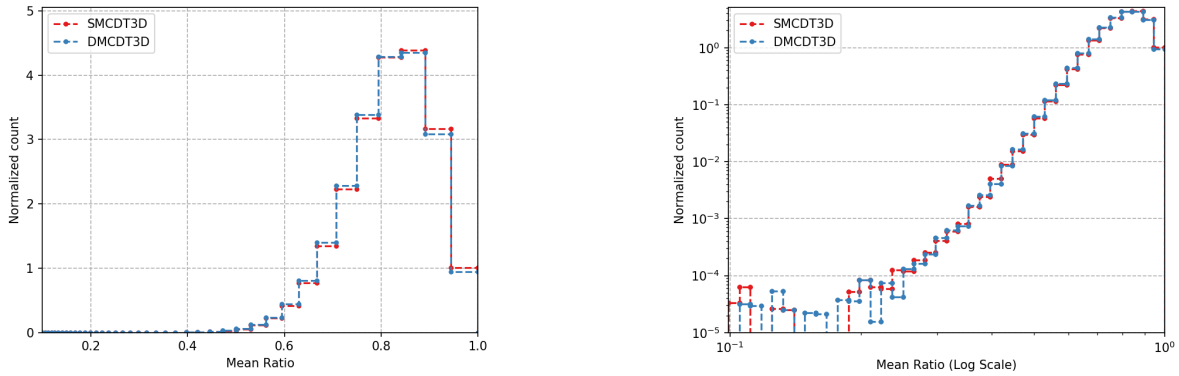


Fig. 11 Comparison of the mean ratio of elements within the meshes generated by the shared memory CDT3D method (SMCDT3D) and the distributed memory CDT3D method (DMCDT3D) for the delta wing geometry at 500,000 complexity in linear and logarithmic scales

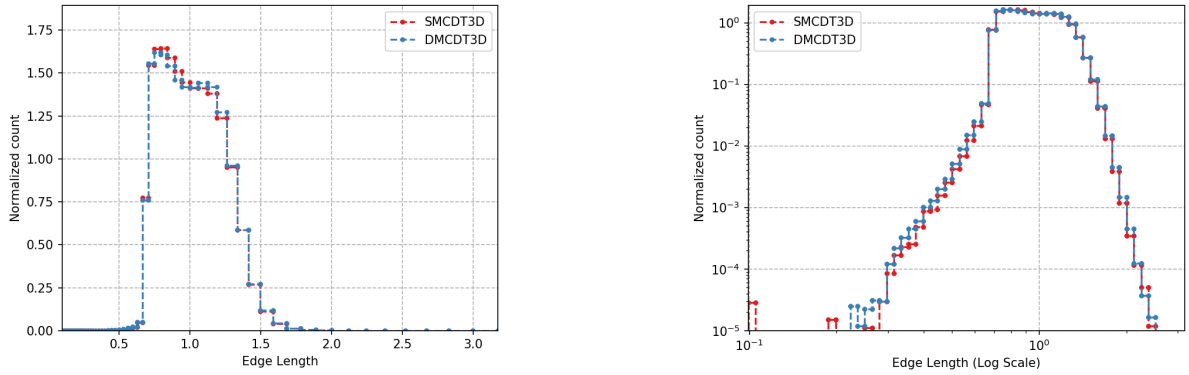


Fig. 12 Comparison of the edge lengths of elements within the meshes generated by the shared memory CDT3D method (SMCDT3D) and the distributed memory CDT3D method (DMCDT3D) for the delta wing geometry at 500,000 complexity in linear and logarithmic scales

2. Local Performance

As stated previously, the distributed memory method aims to avoid collective communication. PREMA gives the method the capability to avoid any explicit global synchronization. Establishing dependencies between mobile objects (subdomains) helps facilitate the execution of interface shifting using PREMA's event system. Subdomains (within neighborhoods) are colored to distinguish which will send or receive interface data, the shift is performed between corresponding neighbors, colored subdomains are adapted, and the topology of neighboring subdomains is updated. While the current implementation of the distributed memory method utilizes a centralized model in testing the correctness and stability of the method (the master process colors subdomains after they have communicated amongst themselves and finally the master about the topology update), its final implementation will utilize a decentralized model. Neighborhoods of subdomains will operate independently of each other (while working in tandem with PREMA's event functionality) without needing to communicate with the master process for coloring. Due to the partially synchronous nature of the current implementation (and its centralized model), PREMA's load balancing is not utilized. It is not expected to offer much improvement until the decentralized (fully asynchronous version) of the distributed memory method has been implemented.

Figure 13 shows the local performance costs of communication within the distributed memory method when adapting the delta wing geometry at 10 million complexity under the same heuristics as reported for the qualitative results (PQR decomposition of 16 subdomains across the x-axis). The communication surrounding the adaptation of mixed interior/interface data includes: coloring subdomains, gathering interface data (for those subdomains designated to send), scattering interface data (on those subdomains designated to receive), updating the topology of subdomain adjacency, converting between the distributed memory data structures and the shared memory method's data structures, and miscellaneous operations (declaring variables, resizing data structures, cleanup, profiling, etc.). The percentages represent the sum runtime spent performing each particular operation. On average, non-meshing operations occupy approximately 30-35% of the sum runtime across different configurations of core utilization. Compared to traditional communication methods that induce overhead of more than 50% (as reported in the ECP study [21]), the cost of communication in permitting the adaptation of interface elements in the early distributed memory method saves approximately 20% of communication overhead thus far.

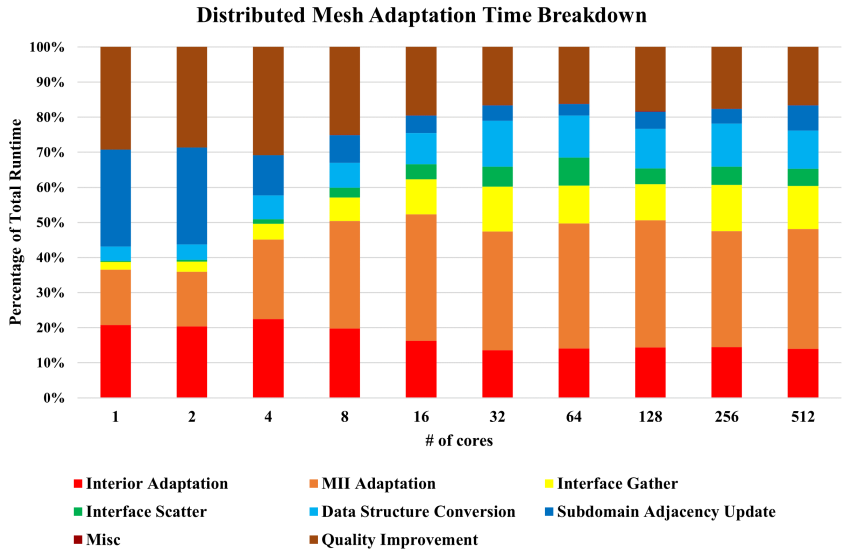


Fig. 13 Percentage breakdown of adaptation time for the Distributed Memory Method in adapting the delta wing geometry at 10 million complexity

Table 1 shows a runtime comparison of the shared memory CDT3D method and the early distributed memory CDT3D method when adapting the delta wing geometry at 10 million complexity. Again, the heuristics used by the distributed method are the same as previously reported, making this a strong scaling problem. Sequential pre-processing time of the distributed method is not included since it was a consistently small fraction across all runs, as data decomposition maintained a runtime of approximately 3 minutes (including re-assigning data to maintain simple connectivity). Time

spent coloring subdomains was also a consistently small fraction of total runtime (less than 1 second across all runs).

Table 1 Runtime (approximately in minutes) of SMCDT3D vs. DMCDT3D when adapting the delta wing geometry at 10 million complexity

Method	# Cores									
	1	2	4	8	16	32	64	128	256	512
SMCDT3D	954	522	301	158	82	42	-	-	-	-
DMCDT3D	3068	1519	528	283	193	124	89	54	46	42

D. CDT3D Redundancy Problem

While the non-meshing operations of the distributed method should be optimized, scalability is hindered primarily by CDT3D's "redundancy" problem. CDT3D is able to produce output of similar quality when processing elements that have already undergone adaptation (in the interior adaptation phase or a previous interface shift iteration); therefore, it satisfies the weak reproducibility requirement. However, it spends unnecessary time processing these elements (that it already generated and attempted to improve in one of those past iterations). To examine this phenomenon in more detail, we tested the distributed method with a two-subdomain data decomposition (across the x-axis) of the delta wing geometry and one interface shift iteration (similar to that seen in Figure 6). The delta wing is again adapted at a complexity of 10 million. Two nodes were allocated with 32 cores each (1 for PREMA's communication and 31 for mesh adaptation and all other operations). We tested this phenomenon across two experiments - one with the pseudo-active modification and order of operations (described in sections V.D.2 and V.C.1, respectively), and one with a naive approach without these modifications (using the black box and its default operations, including edge collapse, to mimic the original shared memory method's execution for each subdomain's adaptation). **During the interface shift of these two experiments, the subdomain undergoes two mixed interior/interface adaptation calls instead of just one for verification purposes**. This means that after the MII adaptation completes, CDT3D is immediately called again to adapt the data that was generated. In the first experiment (with the pseudo-active modification), this second call to commence MII adaptation ignores the pseudo-active modification (while still honoring the order of operations). This allows us to verify that the phenomenon occurs in general, regardless of any new data being merged with the subdomain. This second MII adaptation call in both redundancy experiments is henceforth referred to as the "verification" adaptation. Table 2 shows quantitative and qualitative results from both redundancy experiments regarding the subdomain that undergoes MII adaptation. The experiment with the pseudo-active modification and order of operations is referred to as "Modified" while the other (without these) is referred to as "Naive." Table 3 shows a further breakdown of time spent in each CDT3D operation for each adaptation in the experiments. Smaller operations (such as merging the lists of tetrahedra that were partitioned for threads) are omitted as they only make up a small fraction of CDT3D's runtime. Figure 14 provides a visual aid in understanding how the element edge lengths change throughout the distributed method's adaptation in the "Modified" experiment.

VII. Discussion and Future Work

While CDT3D satisfies the weak reproducibility requirement within the context of the distributed method (verified by Figure 14 and shown in the qualitative results in Table 2), the adaptation times for both the MII and verification adaptations in each experiment suggest that CDT3D is not designed to satisfy this requirement efficiently with regards to time. In the "Modified" experiment, both the MII and verification adaptation times are about 2/3 of the interior adaptation time. In the "Naive" experiment, both the MII and verification adaptation times are more than the interior adaptation time. It should be noted that only 1.4% of elements are activated at the beginning of the MII adaptation in the pseudo-active experiment, generating about 4% more elements in 416 seconds. About 33% of elements are activated at the beginning of the MII adaptation in the naive experiment, which also generates about 4% more elements. However, the naive experiment's MII adaptation takes almost double the time of the pseudo-active experiment's MII adaptation. This shows that while the pseudo-active modification alleviates the redundancy problem, it does not fully remedy the issue.

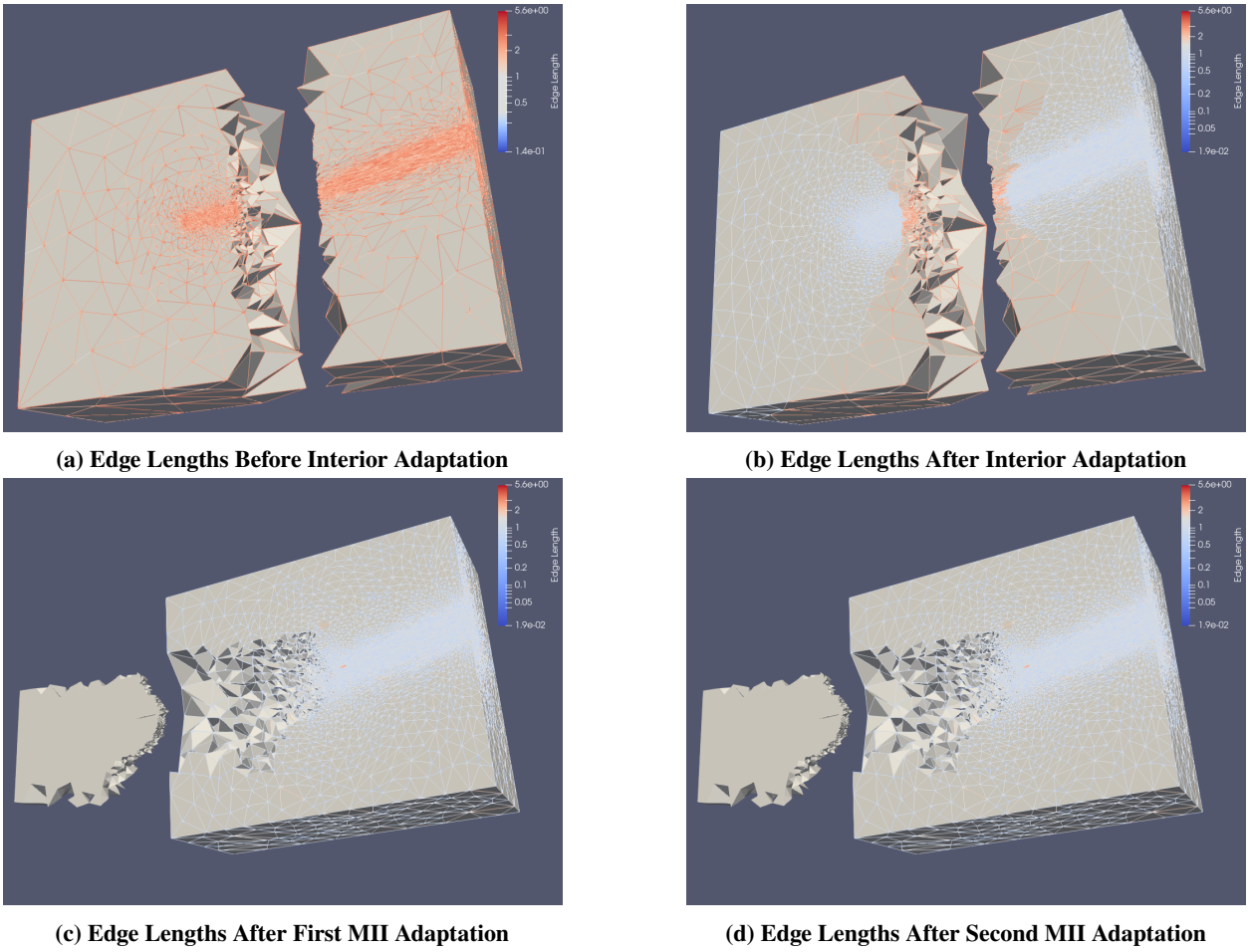
Despite past parallelization efforts that improved the performance of the local reconnection operation in shared memory (exhibiting 88% parallel efficiency when utilizing up to 24 cores for CDT3D's advancing front isotropic

Table 2 Quantitative and qualitative statistics for the subdomain that undergoes MII adaptation in each of the Redundancy experiments of CDT3D within DMCDT3D - The Modified column is the experiment that includes the order of operations and pseudo-active modification for the MII adaptation, and only the order of operations for the subsequent verification adaptation. The Naive column is the experiment that does not include the order of operations or pseudo-active modification (default settings) for the MII adaptation and subsequent verification adaptation ('M' is million and 'K' is thousand).

Adaptation Phase	Statistics	Naive	**Modified**
Interior	Adaptation Time (sec)	609.5	**593.79**
	Start # Tets	9.6M	9.6M
	End # Tets	46.5M	46.6M
	Minimum Mean Ratio	1.0984e-09	0.00869
	# Edges Longer than 2	113K	113K
	# Edges Shorter than 0.5	2K	3K
MII	Adaptation Time (sec)	774.77	**416.05**
	Start # Tets	48M	48M
	Start # Active Tets	16.1M	676K
	End # Tets	50.2M	50.2M
	Minimum Mean Ratio	0.00651	0.00869
	# Edges Longer than 2	11K	11K
	# Edges Shorter than 0.5	2K	3K
Verification	Adaptation Time (sec)	619.62	**404.3**
	Start # Tets	50.22M	50.22M
	Start # Active Tets	14.21M	12.8M
	End # Tets	50.16M	50.28M
	Minimum Mean Ratio	0.00716	0.0566
	# Edges Longer than 2	10K	6K
	# Edges Shorter than 0.5	1K	3K

Table 3 Breakdown of time spent (seconds) in CDT3D operations for the subdomain that undergoes MII adaptation in each Redundancy experiment of CDT3D within DMCDT3D - The Modified experiment includes the order of operations and pseudo-active modification while the naive does not.

Redundancy Experiment	Operations	Interior	MII	Total
Modified	Deactivation	27.44	25.07	52.51
	Create Vertices	103.67	65.2	168.87
	Insert Vertices	13.91	0.86	14.77
	Local Reconnection	**441.13**	**308.56**	**749.69**
	Partitioning Tets	1.54	7.67	9.21
Naive	Edge Collapse	20.27	18.55	38.82
	Deactivation	26.88	34.95	61.83
	Create Vertices	102.33	277.11	379.44
	Insert Vertices	14.19	1.95	16.14
	Local Reconnection	**438.19**	**390.14**	**828.33**
	Partitioning Tets	1.71	7.6	9.31



(a) Edge Lengths Before Interior Adaptation

(b) Edge Lengths After Interior Adaptation

(c) Edge Lengths After First MII Adaptation

(d) Edge Lengths After Second MII Adaptation

Fig. 14 The element edge lengths are shown throughout the distributed method's adaptation with the Modified experiment (that with the pseudo-active modification and order of operations). Orange/red edges are too long and do not satisfy spacing requirements in the metric space while white and light blue edges do. Dark blue edges are too short.

mesh generation method [2] and 98% efficiency for its local-reconnection-based adaptive anisotropic mesh generation method on 40 cores [3]), this operation still remains a bottleneck in CDT3D's runtime (seen in Table 3). These results suggest that the local reconnection operation requires further optimization and/or re-design for efficient execution in the distributed memory method. A potential cause of its domination in the mixed interior/interface adaptation time may be related to how reconnection affects inactive elements that are neighbors to active elements. If a combination of active and inactive elements undergo flips, the new tetrahedra are made active. Although the pseudo-active modification limits the number of active elements before adaptation commences, this number may grow over multiple grid generation passes due to local reconnection (generating active elements over those that were manually deactivated before adaptation). This will require further investigation, particularly to see which elements are modified by local reconnection during each grid generation pass. A potential solution would be to lock those vertices that define the outer faces of the pseudo-active region of tetrahedra, constraining CDT3D's operations to only modify elements inside this region (i.e., only modifying elements in each grid generation pass that were active before adaptation commenced). Another potential solution would be to only pass as input to CDT3D the region of pseudo-active elements and necessary layers to act as a buffer zone (for the aforementioned point creation requirement described in section V.C.3).

Additionally, the inspiration for the pseudo-active modification must be considered. 20-30% of elements remain active by the end of adaptation (even when executing the original shared memory method [3], as described in section V.D.2). It is likely that CDT3D spends time attempting to improve these elements in addition to the (previously constrained) interface elements during mixed interior/interface adaptation. Further investigation will be required to

see where these non-interface elements of lower quality are located within a subdomain to verify this hypothesis. If true, different qualitative thresholds may need to be introduced into CDT3D so that the method considers elements which have already undergone adaptation to be of sufficient quality. Therefore, it would only spend time modifying elements that are of much lower quality (i.e., those that have not undergone adaptation at all - interface elements and those constrained by them).

While the shared memory CDT3D software was developed to be a building block for scalable parallel mesh generation and adaptivity, more work is required if it is to function efficiently as the parallel optimistic layer of the Telescopic Approach (Figure 1). The results of the redundancy experiment echo the recurrent conclusion (similar to the final deductions of the aforementioned black box parallelization efforts in section IV) that a code cannot be simply integrated into a parallel framework as a black box if it was designed and developed independently of that framework.

This early implementation of the distributed memory method exhibits poor scalability due to CDT3D's redundancy problem, gather/scatter operations that must be optimized, and the temporary centralized model under use (for coloring during the interface shift phase, which creates an implicit global synchronization point in each iteration). As stated previously, OpenMP constructs are utilized in the gather/scatter operations. Although this helps to reduce the percentage of runtime occupied by these operations (to 30-35%) when utilizing multiple cores, these operations become much more expensive when utilizing a low number of cores. Additionally, these operations will eventually become a bottleneck when CDT3D's redundancy problem is fully remedied. Consequently, they must be re-designed and optimized to provide efficient use of CDT3D's data structures. While the centralized model is used to test the correctness and stability of the distributed method (the master process sets up PREMA events to denote which subdomains will send or receive data during interface shifting), its final implementation will utilize a decentralized model. Luby's algorithm [42] can be used for such an implementation to create maximal independent sets of subdomains. This will aid in allowing neighborhoods of subdomains to operate independently of each other (while working in tandem with PREMA's event functionality), avoiding the need to communicate with the master process for coloring. As stated previously, the method of decomposition can also affect how quickly the distributed memory method generates a mesh that conforms to the target metric. With regards to PQR's partitioning heuristics, metric complexity can serve as a weight when determining how large or small subdomains should be. The impact of this weight on the overall method will be investigated in the future. More comprehensive statistics (such as the min/max time of adaptation amongst subdomains, the overhead of PREMA, etc.) will also be investigated in the future.

It should also be noted that there is no interface shifting during the quality improvement phase. As seen in our evaluation, the distributed memory method is able to produce meshes of comparable quality to those of the shared memory method when adapting the delta wing geometry. The presented method will be tested with all the geometries that were utilized for the evaluation of the original shared memory CDT3D [3, 10]. Given the constraints set by interface elements, interface shifting will also be implemented during the quality improvement phase. However, CDT3D's redundancy problem must first be addressed. Otherwise, performance of the quality improvement phase will suffer from the same issue.

VIII. Conclusion

The foundational elements of a distributed memory method for adaptive anisotropic mesh generation are presented. Meshing functionality is separated from performance aspects in order to fit a scalable framework that is designed to leverage maximum concurrency offered by large-scale architectures. Mesh adaptation is handled by a shared memory code called CDT3D. Several requirements regarding the distributed method's design are given based on lessons learned from re-designing some components of the shared memory code, enabling its integration into the distributed memory method. All major operations within the shared memory code are designed to adopt a speculative execution model, enabling the strict adaptation of interior subdomain elements so that interface elements can be adapted in a separate step to maintain mesh conformity. Interface elements undergo several iterations of shifting so that they are adapted when their data dependencies are resolved. Communication and migration of data are handled by a parallel runtime system called PREMA. PREMA offers a particularly useful "event" feature which aids in establishing data dependencies between subdomains, thus enabling "neighborhoods" of subdomains to work independently of each other in performing interface shifts and adaptation.

Preliminary results show that after several iterations of interface shifts and adaptation, the distributed memory CDT3D method is able to produce meshes of comparable quality to those generated by the original shared memory CDT3D software. However, the shared memory CDT3D method presents a challenge regarding its ability to efficiently re-process data that it generated in previous adaptations (a necessity given that interface elements are merged with

elements already adapted during shifting). It does so successfully but not efficiently with regards to time. These results suggest that a code cannot be simply integrated into a parallel framework as a black box if it was designed and developed independently of that framework. The shared memory method must undergo further re-design if the distributed memory method is to achieve scalability.

Local communication performance regarding non-meshing operations, however, suggests that if CDT3D can be successfully re-designed, the interface shift operation presents a potentially viable solution in achieving scalability for mesh adaptation when targeting configurations with large numbers of cores (building upon the shared memory CDT3D method which only utilized up to 40 cores in an earlier study [10]). Given the costly overhead of collective communication identified within the study of DoE's Exascale Computing Project [21, 22] and seen in existing state-of-the-art HPC software [10], the distributed memory method's emphasis on avoiding global synchronization (using PREMA's event functionality to perform local communication within neighborhoods of subdomains) will likely prove beneficial upon completion of its fully asynchronous implementation.

Acknowledgments

This research was sponsored by the Richard T. Cheng Endowment, the Southern Regional Education Board (SREB) State Doctoral Scholar Fellowship, and the National Institute of General Medical Sciences of the National Institutes of Health under Award Number 1T32GM140911-03. The content is solely the authors' responsibility and does not necessarily represent the official views of the National Institutes of Health.

References

[1] Slotnick, J. P., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E., and Mavriplis, D. J., "CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences," NASA, 2014. https://doi.org/2060/20140003093, URL https://ntrs.nasa.gov/citations/20140003093, nASA CR-2014-218178.

[2] Drakopoulos, F., Tsolakis, C., and Chrisochoides, N., "Fine-grained Speculative Topological Transformation Scheme for Local Reconnection Methods," *AIAA Journal*, Vol. 57, 2019, pp. 4007–4018. https://doi.org/10.2514/1.J057657.

[3] Tsolakis, C., and Chrisochoides, N., "Speculative anisotropic mesh adaptation on shared memory for CFD applications," *Engineering with Computers*, 2024. https://doi.org/10.1007/s00366-024-01994-0.

[4] Barker, K., Chernikov, A., Chrisochoides, N., and Pingali, K., "A load balancing framework for adaptive and asynchronous applications," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 2, 2004, pp. 183–192. https://doi.org/10.1109/TPDS.2004.1264800.

[5] Thomadakis, P., Tsolakis, C., and Chrisochoides, N., "Multithreaded Runtime Framework for Parallel and Adaptive Applications," *Engineering with Computers*, Vol. 38, 2022, p. 4675–4695. https://doi.org/10.1007/s00366-022-01713-7.

[6] Thomadakis, P., and Chrisochoides, N., "Toward runtime support for unstructured and dynamic exascale-era applications," *The Journal of Supercomputing*, Vol. 79, 2023, p. 9245–9272. https://doi.org/10.1007/s11227-022-05023-z.

[7] Chrisochoides, N., "Telescopic Approach for Extreme-Scale Parallel Mesh Generation for CFD Applications," *46th AIAA Fluid Dynamics Conference*, Washington D.C., USA, 2016. https://doi.org/10.2514/6.2016-3181, aIAA 2016-3181.

[8] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, Association for Computing Machinery, New York, NY, USA, 1967, p. 483–485. https://doi.org/10.1145/1465482.1465560.

[9] Chrisochoides, N., Chernikov, A., Kennedy, T., Tsolakis, C., and Garner, K., "Parallel Data Refinement Layer of a Telescopic Approach for Extreme-scale Parallel Mesh Generation for CFD Applications," *2018 Aviation Technology, Integration, and Operations Conference*, Atlanta, Georgia, 2018. https://doi.org/10.2514/6.2018-2887, aIAA 2018-2887.

[10] Tsolakis, C., Chrisochoides, N., Park, M. A., Loseille, A., and Michal, T., "Parallel Anisotropic Unstructured Grid Adaptation," *AIAA Journal*, Vol. 59, 2021, pp. 4764–4776. https://doi.org/10.2514/1.J060270.

[11] Tsolakis, C., Thomadakis, P., and Chrisochoides, N., "Tasking Framework for Adaptive Speculative Parallel Mesh Generation," *The Journal of Supercomputing*, Vol. 78, 2022, pp. 1–32. https://doi.org/10.1007/s11227-021-04158-9.

[12] Loseille, A., Menier, V., and Alauzet, F., "Parallel Generation of Large-size Adapted Meshes," *Procedia Engineering*, Vol. 124, 2015, pp. 57–69. https://doi.org/10.1016/j.proeng.2015.10.122, URL https://www.sciencedirect.com/science/article/pii/S1877705815032233, 24th International Meshing Roundtable.

[13] Caplan, P. C., "Parallel four-dimensional anisotropic mesh adaptation," *International Meshing Roundtable 2022*, 2022. URL https://internationalmeshingroundtable.com/assets/papers/2022/09-Caplan-compressed.pdf.

[14] Digonnet, H., Coupez, T., Laure, P., and Silva, L., "Massively parallel anisotropic mesh adaptation," *The International Journal of High Performance Computing Applications*, Vol. 33, No. 1, 2019, pp. 3–24. https://doi.org/10.1177/1094342017693906.

[15] Park, M., and Darmofal, D., "Parallel Anisotropic Tetrahedral Adaption," *46th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, Nevada, USA, 2008. https://doi.org/10.2514/6.2008-917, aIAA 2008-917.

[16] Gorman, G., Southern, J., Farrell, P., Piggott, M., Rokos, G., and Kelly, P., "Hybrid OpenMP/MPI Anisotropic Mesh Smoothing," *Procedia Computer Science*, Vol. 9, 2012, pp. 1513–1522. https://doi.org/10.1016/j.procs.2012.04.166, URL https://www.sciencedirect.com/science/article/pii/S1877050912002876, proceedings of the International Conference on Computational Science, ICCS 2012.

[17] Ibanez, D. A., Seol, E. S., Smith, C. W., and Shephard, M. S., "PUMI: Parallel Unstructured Mesh Infrastructure," *ACM Trans. Math. Softw.*, Vol. 42, No. 3, 2016. https://doi.org/10.1145/2814935.

[18] Seol, E. S., and Shephard, M. S., "Efficient distributed mesh data structure for parallel automated adaptive analysis," *Engineering with Computers*, Vol. 22, 2006, pp. 197–213. https://doi.org/10.1007/s00366-006-0048-4.

[19] Sahni, O., Ovcharenko, A., Chitale, K. C., Jansen, K. E., and Shephard, M. S., "Parallel anisotropic mesh adaptation with boundary layers for automated viscous flow simulations," *Engineering with Computers*, Vol. 33, 2017, pp. 767–795. https://doi.org/10.1007/s00366-016-0437-2.

[20] Michal, T., and Krakos, J., "Anisotropic Mesh Adaptation Through Edge Primitive Operations," *50th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2012. https://doi.org/10.2514/6.2012-159, aIAA 2012-0159.

[21] Sultana, N., Ruefenacht, M., Skjellum, A., Bangalore, P., Laguna, I., and Mohror, K., "Understanding the use of message passing interface in exascale proxy applications," *Concurrency and Computation: Practice and Experience*, Vol. 33, 2020. https://doi.org/10.1002/cpe.5901.

[22] Klenk, B., and Fröning, H., "An Overview of MPI Characteristics of Exascale Proxy Applications," *International Conference on High Performance Computing*, 2017, pp. 217–236. https://doi.org/10.1007/978-3-319-58667-0_12.

[23] Foteinos, P., and Chrisochoides, N., "Dynamic Parallel 3D Delaunay Triangulation," *Proceedings of the 20th International Meshing Roundtable*, edited by W. R. Quadros, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 3–20. https://doi.org/10.1007/978-3-642-24734-7_1.

[24] Foteinos, P. A., and Chrisochoides, N. P., "High quality real-time Image-to-Mesh conversion for finite element simulations," *Journal of Parallel and Distributed Computing*, Vol. 74, No. 2, 2014, pp. 2123–2140. https://doi.org/10.1016/j.jpdc.2013.11.002, URL https://www.sciencedirect.com/science/article/pii/S0743731513002232.

[25] Zagaris, G., Pirzadeh, S., and Chrisochoides, N., "A Framework for Parallel Unstructured Grid Generation for Practical Aerodynamic Simulations," *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, Orlando, Florida, USA, 2013. https://doi.org/10.2514/6.2009-980.

[26] Garner, K., "Parallelization of the Advancing Front Local Reconnection Mesh Generation Software Using a Pseudo-Constrained Parallel Data Refinement Method," 2020. https://doi.org/10.25777/appr-3169, master's thesis.

[27] Thomadakis, P., and Chrisochoides, N., "Experience with Distributed Memory Delaunay-based Image-to-Mesh Conversion Implementation," *arXiv*, 2023. URL https://arxiv.org/abs/2308.12525.

[28] Dijkstra, E., *On the Role of Scientific Thought*, Springer-Verlag, Berlin, Heidelberg, 1982.

[29] Ibanez, D., Barral, N., Krakos, J., Loseille, A., Michal, T., and Park, M., "First benchmark of the Unstructured Grid Adaptation Working Group," *Procedia Engineering*, Vol. 203, 2017, pp. 154–166. https://doi.org/10.1016/j.proeng.2017.09.800, URL https://www.sciencedirect.com/science/article/pii/S1877705817343618, 26th International Meshing Roundtable, IMR26, 18-21 September 2017, Barcelona, Spain.

[30] Hoefler, T., and Träff, J., "Sparse collective operations for MPI," *23rd IEEE International Symposium on Parallel and Distributed Processing*, 2009, pp. 1–8. https://doi.org/10.1109/IPDPS.2009.5160935.

[31] Cirrottola, L., and Froehly, A., "Parallel unstructured mesh adaptation using iterative remeshing and repartitioning," Research Report RR-9307, INRIA Bordeaux, équipe CARDAMOM, Nov. 2019. URL https://inria.hal.science/hal-02386837.

[32] Rokos, G., Gorman, G. J., Southern, J., and Kelly, P. H. J., "A thread-parallel algorithm for anisotropic mesh adaptation," *arXiv*, 2013. URL https://arxiv.org/abs/1308.2480.

[33] Ibanez, D., and Shephard, M. S., "Mesh adaptation for moving objects on shared memory hardware," *25th International Meshing Roundtable*, Washington, D.C., USA, 2016. URL https://api.semanticscholar.org/CorpusID:18461041.

[34] Chrisochoides, N., Houstis, E., and Rice, J., "Mapping Algorithms and Software Environment for Data Parallel PDE Iterative Solvers," *Journal of Parallel and Distributed Computing*, Vol. 21, No. 1, 1994, pp. 75–95. https://doi.org/10.1006/jpdc.1994.1043, URL https://www.sciencedirect.com/science/article/pii/S0743731584710434.

[35] Nave, D., Chrisochoides, N., and Chew, L., "Guaranteed-quality parallel Delaunay refinement for restricted polyhedral domains," *Computational Geometry*, Vol. 28, No. 2, 2004, pp. 191–215. https://doi.org/10.1016/j.comgeo.2004.03.009, URL https://www.sciencedirect.com/science/article/pii/S0925772104000240, special Issue on the 18th Annual Symposium on Computational Geometry - SoCG2002.

[36] Loseille, A., and Alauzet, F., "Continuous Mesh Framework Part I: Well-Posed Continuous Interpolation Error," *SIAM Journal on Numerical Analysis*, Vol. 49, No. 1, 2011, pp. 38–60. https://doi.org/10.1137/090754078.

[37] Loseille, A., and Alauzet, F., "Continuous Mesh Framework Part II: Validations and Applications," *SIAM Journal on Numerical Analysis*, Vol. 49, No. 1, 2011, pp. 61–86. https://doi.org/10.1137/10078654X.

[38] Park, M. A., Loseille, A., Krakos, J. A., and Michal, T. R., "Comparing Anisotropic Output-Based Grid Adaptation Methods by Decomposition," *22nd AIAA Computational Fluid Dynamics Conference*, Dallas, TX, 2015. https://doi.org/10.2514/6.2015-2292, aIAA 2015-2292.

[39] Alauzet, F., "Size gradation control of anisotropic meshes," *Finite Elements in Analysis and Design*, Vol. 46, No. 1, 2010, pp. 181–202. https://doi.org/10.1016/j.finel.2009.06.028, URL https://www.sciencedirect.com/science/article/pii/S0168874X09000912, mesh Generation - Applications and Adaptation.

[40] Alauzet, F., and Loseille, A., "High-order sonic boom modeling based on adaptive methods," *Journal of Computational Physics*, Vol. 229, No. 3, 2010, pp. 561–593. https://doi.org/10.1016/j.jcp.2009.09.020, URL https://www.sciencedirect.com/science/article/pii/S0021999109005129.

[41] Park, M., Balan, A., Anderson, K., Galbraith, M., Caplan, P., Carson, H., Michal, T., Krakos, J., Kamenetskiy, D., Loseille, A., Alauzet, F., Frazza, L., and Barral, N., "Verification of Unstructured Grid Adaptation Components," *AIAA Scitech 2019 Forum*, 2019. https://doi.org/10.2514/6.2019-1723, aIAA-1723.

[42] Luby, M., "A Simple Parallel Algorithm for the Maximal Independent Set Problem," *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, Association for Computing Machinery, New York, NY, USA, 1985, p. 1–10. https://doi.org/10.1145/22145.22146.