Fall 12-2020

# Parallelization of the Advancing Front Local Reconnection Mesh Generation Software Using a Pseudo-Constrained Parallel Data Refinement Method

Kevin Mark Garner Jr.
*Old Dominion University*, yokevink@hotmail.com

# PARALLELIZATION OF THE ADVANCING FRONT LOCAL RECONNECTION MESH GENERATION SOFTWARE USING A PSEUDO-CONSTRAINED PARALLEL DATA REFINEMENT METHOD

by

Kevin Mark Garner Jr.
B.S. May 2016, Old Dominion University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December 2020

Approved by:

Nicolaos Chrisochoides (Director)

Andrey Chernikov (Member)

Michael Park (Member)

Ravi Mukkamala (Member)

# ABSTRACT

## PARALLELIZATION OF THE ADVANCING FRONT LOCAL RECONNECTION MESH GENERATION SOFTWARE USING A PSEUDO-CONSTRAINED PARALLEL DATA REFINEMENT METHOD

Kevin Mark Garner Jr.
Old Dominion University, 2020
Director: Dr. Nicolaos Chrisochoides

Preliminary results of a long-term project entailing the parallelization of an industrial strength sequential mesh generator, called Advancing Front Local Reconnection (AFLR), are presented. AFLR has been under development for the last 25 years at the NSF/ERC center at Mississippi State University. The parallel procedure that is presented is called Pseudo-constrained (PsC) Parallel Data Refinement (PDR) and consists of the following steps: (i) use an octree data-decomposition scheme to divide the original geometry into subdomains (octree leaves), (ii) refine each subdomain with the proper adjustments of its neighbors using the given refinement code, and (iii) combine all subdomain data into a single, conforming mesh. Parallelism was achieved by implementing Pseudo-constrained Parallel Data Refinement AFLR (PsC.AFLR) on top of a runtime system called Parallel Runtime Environment for Multi-computer Applications (PREMA). During run time, the PsC.AFLR method exposes data decomposition information (number of subdomains waiting to be refined) to the underlying runtime system. In turn, this system facilitates work-load balancing and guides the program's execution towards the most efficient utilization of hardware resources. Preliminary results, on the mesh refinement operation, show that the end-user productivity (measured in terms of elements refined per second) increases as the number of cores in use are increased. When using approximately 16 cores, PsC.AFLR outperforms the serial AFLR code by about 11 times. PsC.AFLR also maintains its stability by generating meshes of comparable quality. Although it offers good end-user productivity, PsC.AFLR suffers in its capability to generate meshes with the same level of density or quality as that of the serial AFLR software due to the constraints set by subdomain boundaries that are required to successfully execute AFLR. These constraints demonstrate that it is not ideal to use AFLR in a black box manner when parallelizing the software. Its source code must be modified to a non-trivial extent if one wishes to remove these constraints and maximize the end-user productivity and potential scalability.

This thesis is dedicated to the belief that no matter how hard life gets, you must always keep pushing forward because only you have the power to truly improve your circumstances, whether that be mentally or physically, and to succeed in your endeavors.

# ACKNOWLEDGEMENTS

There are many people who have contributed to the successful completion of this thesis. I would like to thank all my committee members for their input and guidance as I worked on this project. I would especially like to thank my advisor Dr. Nikos Chrisochoides for not only being a great advisor but for also being a great mentor and friend, and for imparting his own personal wisdom and knowledge onto me as I have matured both personally and professionally. I extend my thanks to those at the CRTC lab, who taught me much and helped me with some crucial aspects of this project – Thomas Kennedy, Polykarpos Thomadakis, Christos Tsolakis, and Fotis Drakopoulos. I would like to thank Dr. David Marcum, of the Center of Advanced Vehicular Systems at Mississippi State University, for acting as a consultant for any questions I had regarding the modifications of AFLR. I also extend my thanks to Dr. Michael Park for imparting knowledge onto me regarding mesh quality assurance for CFD solvers, which greatly aided me in my analysis. I would also like to thank NASA for this opportunity, given that this research was sponsored by NASA's Transformational Tools and Technologies Project (grant no. NNX15AU39A) of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate. I would also like to thank the Southern Regional Education Board (SREB Doctoral Fellowship), Virginia Space Grant Consortium (VSGC Graduate Research Fellowship), and the National Science Foundation (NSF grant no. CCF-1439079) for funding my research.

I want to finally thank my family and friends for their support through these years of my college career thus far – my mother Carolyn Vaughan, my grandmother Geraldine Riddick, my uncle Andre Riddick, and my friends Juan Rodriques and Hussam Hallak. Your unyielding support got me through some tough times and inspired me to work to the best of my ability.

# TABLE OF CONTENTS

Page

# LIST OF TABLES

Table Page

# LIST OF FIGURES

Figure Page

# 1. INTRODUCTION

Mesh generation software is used in many industries where high-fidelity simulations are required, such as in healthcare, defense, and aerospace. For the last 25 years, legacy Finite Element (FE) mesh generation methods and software were typically developed with a focus on high performance for single core architectures and without any thought towards scalability. NASA's Computational Fluid Dynamics (CFD) 2030 Vision will require these highly functional codes to run on large-scale parallel architectures [1]. The CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences (NASA/CR-2014-218178) states that "mesh generation and adaptivity continue to be significant bottlenecks in the CFD workflow, and very little government investment has been targeted in these areas. As more capable HPC hardware enables higher resolution simulations, fast, reliable mesh generation and adaptivity will become more problematic." Additionally, adaptive mesh techniques offer great potential, but have not seen widespread use due to issues related to software complexity, inadequate error estimation capabilities, and complex geometries. These issues make the parallelization of highly optimized, sequential versions of existing state-of-the-art mesh generation codes a critical near-term requirement for high-fidelity simulation.

# 2. LITERATURE REVIEW

Parallel mesh generation codes are typically developed using either of the following approaches – functionality-first or scalability-first. The functionality-first approach (which is the focus of this project) attempts to parallelize existing state-of-the-art serial software that are fully functional (robust in its features and capabilities). The scalability-first approach focuses on designing the software from the ground up to maintain good scalability with the caveat of incomplete functionality. New features and capabilities are implemented on an as-needed basis. The functionality-first approach becomes preferable if one is able to achieve an ideal speedup when parallelizing a code that is already fully functional.

There are two such projects developed by the Boeing Company and INRIA (French National Research Institute for Digital Science and Technology) named EPIC [2] and Feflo.a [3] [4], respectively. The runtime and scaling performance of these programs are studied in [5]. EPIC exploits coarse-grain parallelism by partitioning an input mesh into subdomains and then performing operations on these partitions, including point insertion, coarsening, reconnection, and smoothing while temporarily freezing subdomain boundaries. Once these operations are completed for the relevant subdomains, boundary elements are shifted between subdomains so that they are relocated in the interior of a partition and may also undergo refinement. An optimization technique is used to maintain a work-load balance between subdomains as multithreading is utilized for the parallelization of a subset of the partition refinement operations. The difference between EPIC and this project is the limited parallelism of refinement operations while this project attempts to parallelize almost every aspect of the mesh generation process at the subdomain level (point insertion, local reconnection, quality improvement, sliver removal, etc.). Furthermore, EPIC focuses on anisotropic mesh generation by adapting a mesh through edge breaks and collapse operations such that the individual element edge lengths match a given anisotropic metric tensor field. While EPIC focuses on anisotropy, this project is more robust in that it allows both isotropic and anisotropic mesh generation given the serial code it utilizes.

Feflo.a also exploits coarse-grain parallelism by decomposing a mesh into subdomains. Instead of data decomposition (methodology used in this project, detailed in Chapter 3), it uses domain decomposition by splitting the mesh at different levels of partitions [6]. Similarly to EPIC, subdomains are refined in parallel while the subdomain boundary elements are kept frozen. Then new subdomains are formed with these elements now in the interior so that they may also undergo refinement. Feflo.a processes manifold and non-manifold geometries composed of simplicial elements (contrary to the wide range of input elements that can be processed by this project's serial software such as quadrilateral, pentagonal, and hexahedral meshes). First, the input grid is adapted by improving the edge length distribution, using insertion and collapse operators, with respect to the input metric field. Then, the grid is optimized with coordinate smoothing and element edge/face swaps. Another key difference between Feflo.a and the code utilized in this project is that Feflo.a also focuses on anisotropic grid adaptation rather than isotropy. This project's end goal is to offer the full functionality of its serial code while also maintaining good end-user productivity and scalability.

# 3. METHODOLOGY

AFLR is the serial code that is utilized in this parallelization effort and is one of the top, industrial strength, mesh generators that is currently used by NASA, the DoD, DoE, and several aerospace industry research groups [7]. The Center for Real-time Computing (CRTC) at Old Dominion University (ODU) has proposed the telescopic approach (see Fig. 1) [8] [9], a framework that will leverage concurrency at multiple levels in parallel grid generation. At the chip and node levels, the telescopic approach deploys a Parallel Optimistic (PO) layer and Parallel Data Refinement (PDR) layer, respectively. The long-term goal of this effort is to integrate AFLR with the PDR layer, which in turn will be implemented on top of the PO layer in future efforts.

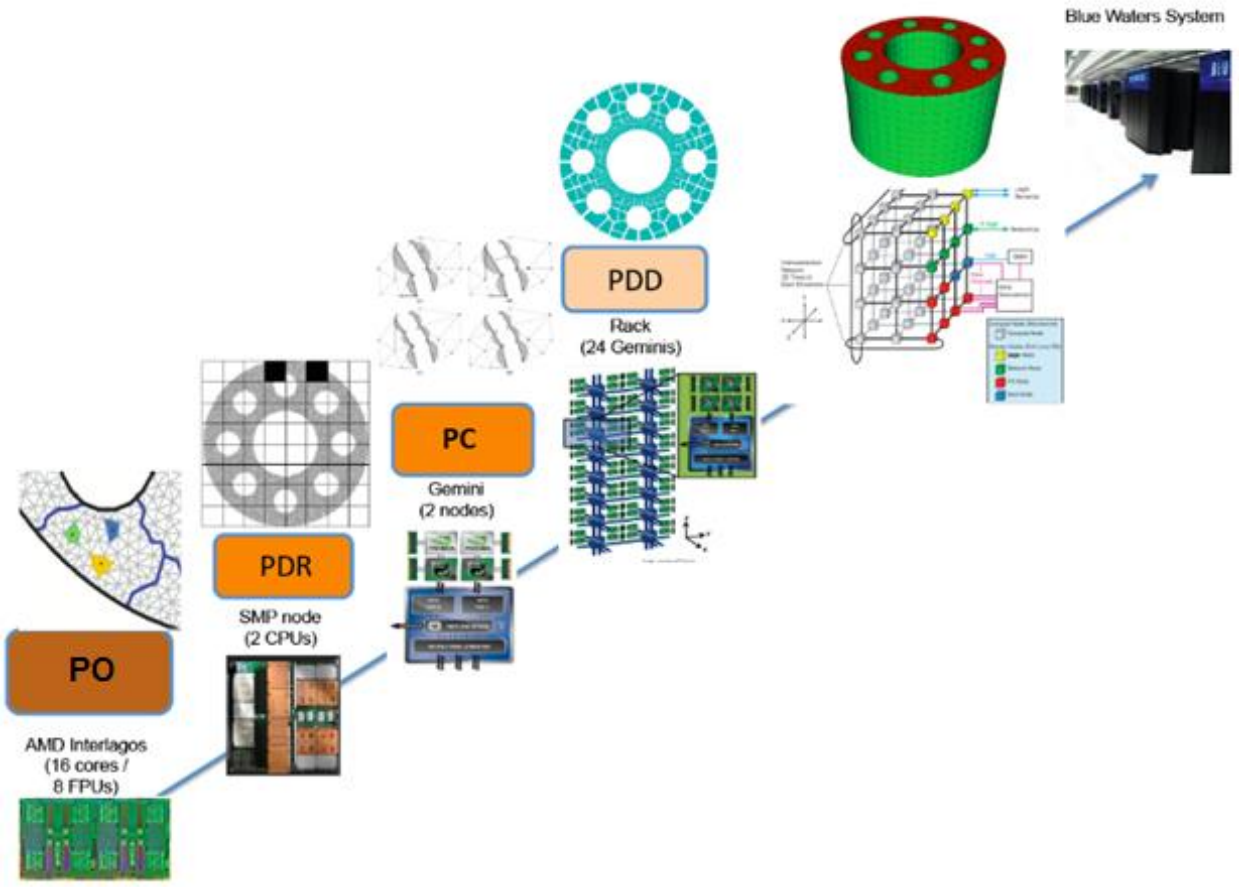


**Fig. 1. Telescopic Approach to Parallel Mesh Generation. It covers the complete spectrum of hardware that spans from the Chip (bottom left) to a complete machine like Blue Waters (top right). The PDR approach is second from the left and targets hardware at the node level.**

PDR maintains a fixed level of concurrency while parallelizing the refinement process. Its methodology is theoretically proven to maintain stability and robustness for parallel isotropic Delaunay-based mesh generation and has been experimentally verified for isotropic meshes [10] [11] [12]. It is also designed to allow for the utilization of any sequential mesh generator while offering guarantees for the following five requirements for parallel mesh generation [13]:

1) **Stability** ensures that a mesh generated in parallel maintains a level of quality comparable to that of a sequentially generated mesh. This quality is defined in terms of the density and shape of the elements evaluated in the metric field, and the number of the elements (fewer is better for the same level of metric conformity).
2) **Robustness** guarantees that the parallel software is able to correctly and efficiently process any input data. Operator intervention into a massively parallel computation is not only highly expensive, but most likely infeasible due to the large number of concurrently processed sub-problems.
3) **Scalability** compares the runtime of the best sequential implementation to the runtime of the parallel implementation, which should achieve a speedup. Non-trivial stages of the computation must be parallelized if one is to leverage current architectures that contain millions of cores.
4) **Code re-use** essentially means that the parallel algorithm should be designed in such a way that it can be replaced and/or updated with minimal effort, regardless of the sequential meshing code it uses. This is a practical approach due to the fact that sequential codes are constantly evolving to accommodate the functionality requirements from the wide ranges of applications and input geometries. Rewriting new parallel algorithms for every sequential meshing code can be highly expensive in time investment. The code re-use approach is only feasible if the sequential mesh generator satisfies the reproducibility criterion.
5) **Reproducibility** requires that the sequential mesh generator, when executed with the same input, produces 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. Elements within a mesh may undergo refinement more than once when in parallel, so it is imperative that the sequential mesh generator satisfy this requirement.

Previous work involving the integration of the mesh generator TetGen [14] with PDR shows that if the mesh generator fails to meet the reproducibility criterion in distributed memory, then the complexity of such state-of-the-art codes inhibits their modifications to a degree that their integration with parallel frameworks like PDR becomes impractical [13]. The original, sequential AFLR code was determined to be a suitable mesh generator to integrate with PDR, as it was tested and shown to maintain weak reproducibility, presented in Fig. 2. Comparisons of the dihedral angle quality statistics are given by the output mesh (after the initial refinement) and the subsequent output meshes (one from the refinement of using the surface of the initial output mesh and another using the volume of the initial output mesh). While results show that the quality is comparable between all of the output meshes, it is not identical. Therefore, one can conclude that AFLR has weak reproducibility, which satisfies PDR's requirement.

PDR decomposes a meshing problem by using an octree consisting of numerous leaves, or subdomains, that each hold a part of the mesh. The general idea of PDR is to concurrently refine the octree leaves while maintaining

mesh conformity. The main concern when parallelizing a refinement algorithm are the data dependencies between leaves caused by concurrent point insertions and the creation/deletion of elements in different octree leaves by multiple threads concurrently. PDR addresses this issue by introducing a buffer zone around each octree leaf. If a part of the mesh associated with a leaf is scheduled for refinement by a thread, no other thread can refine the parts of the mesh associated with the buffer zone of this leaf. This eliminates any data dependency risks and allows PDR to avoid fine-grain synchronization overheads associated with concurrent point insertions. A thread refines a leaf by running a sequential refinement code (AFLR in the implementation presented here) on the subdomain within that leaf. Fig. 3 shows a 2-D example of PDR's data decomposition and the assignment of data generated from the upper portion of a rocket geometry. Mathematical formulas are given for the different levels of neighboring leaves around the primary leaf under refinement (in red). The level 1 neighbors of a leaf are considered to be the buffer zone of that leaf (again, no leaf in the buffer zone may undergo refinement while the primary leaf undergoes refinement).

AFLR accepts an input geometry with an established boundary triangulation. A Delaunay-based method is used to construct an initial boundary-conforming tetrahedral mesh. Each initial boundary point is assigned a value, by a point distribution function, representative of the local point spacing on the boundary surface. This function is used to control the final field point spacing. All elements are initially made active, meaning that they need to be refined. If the points of an element satisfy the point distribution function, the element is made inactive and does not need to be refined. The advancing front method is used on active elements. A face of the element that is adjacent to another active element is selected. A new point is created by advancing in a direction, normal to the selected face, a distance that would produce an equilateral element based on an appropriate length scale (using the average point distribution). If a new point is too close to an existing point or another new point, it is rejected and removed. Accepted points are inserted into the existing grid by subdividing their containing elements. For example, if an edge point is inserted, then all elements sharing that edge are split. If a face point is inserted, then both elements sharing that face are split into three elements. All elements modified by point insertion, or any that undergo reconnection, are classified as active. A local reconnection scheme is used to optimize the connections between points (or edges). Edges are repeatedly reconnected, or swapped, to satisfy a desired quality criterion. A min-max (minimize the maximum angle) criterion is primarily used which maximizes the minimum element edge weight, thereby producing high overall grid quality and eliminating most field sliver elements. All active elements undergo a final optimization phase, which consists of three quality improvement passes (sliver removal, grid coordinate smoothing, and further reconnection) [7].

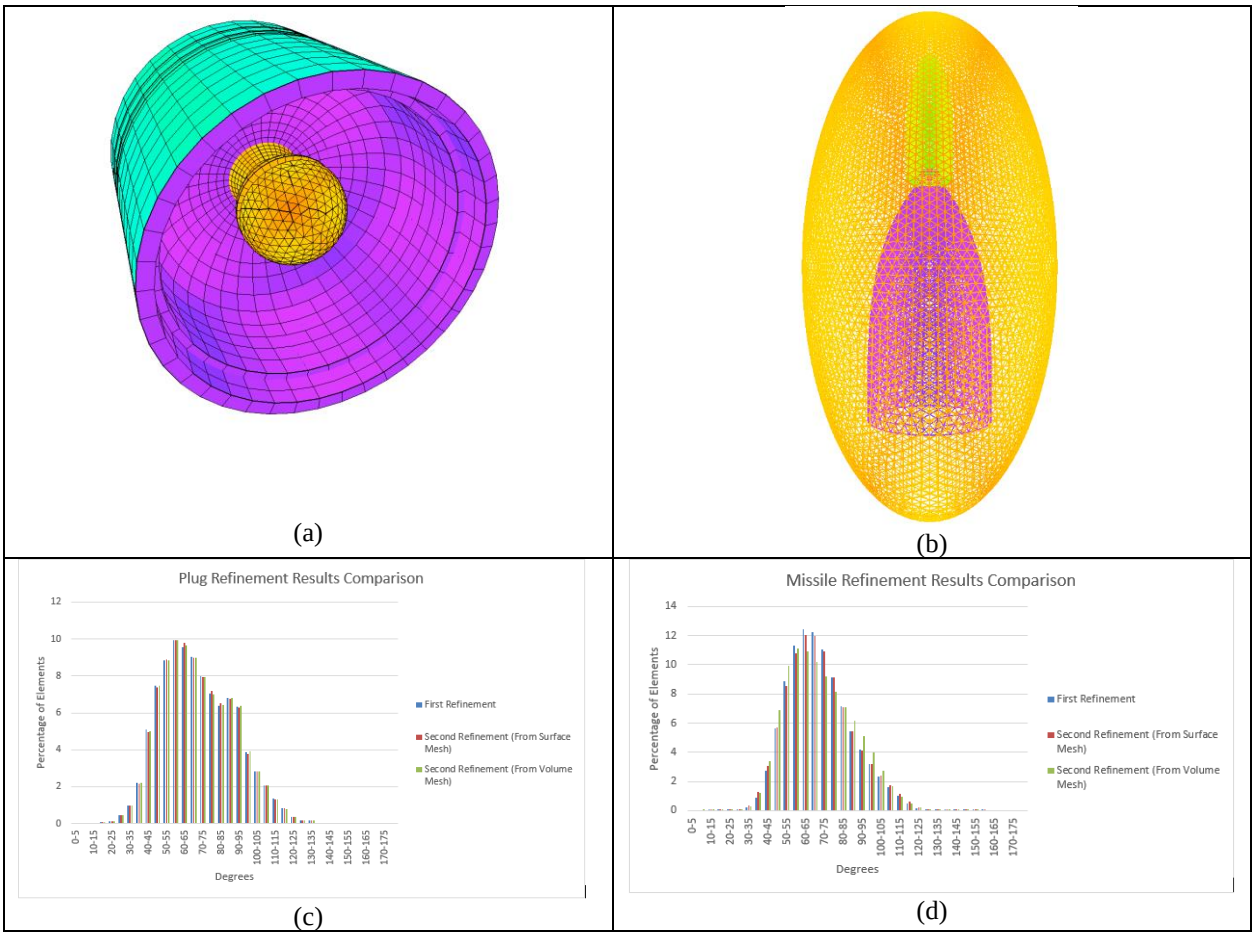


**Fig. 2. AFLR Reproducibility Results. Comparisons of the dihedral angle quality statistics are given between the output mesh (after the first refinement) and the subsequent output meshes (one from the refinement of using the surface of the first output mesh and another using the volume of the first output mesh). (a) and (b) show the plug and missile geometries, respectively, that were refined. (c) and (d) show the dihedral angle statistics generated from refinement of the plug and missile geometries, respectively.**

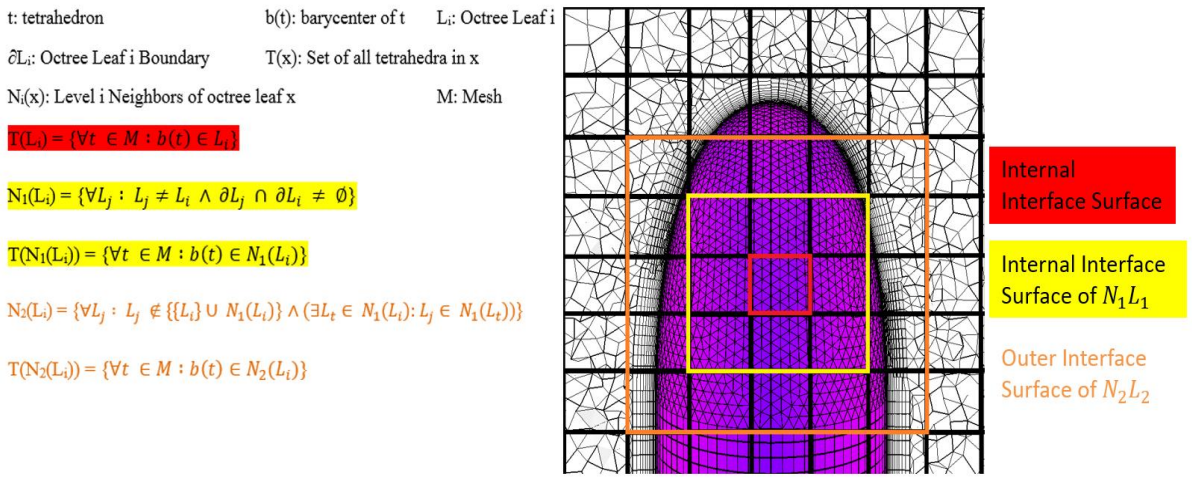


**Fig. 3. 2-D Example of PDR's Data Decomposition. Shown is a 2-D example of the upper portion of a data-decomposed rocket mesh where the red-boxed leaf is the primary leaf under refinement. The level 1 neighbors are those inside the yellow box (excluding the red leaf) and the level 2 neighbors are those inside the orange box (excluding all leaves inside the yellow box). The formulas give mathematical representations that denote the tetrahedra within leaves and the sets of neighboring leaves (with matching colors showing what is contained within each surface).**

## 3.1 INTEGRATION OF AFLR WITH PDR TO INCLUDE SUPER-SUBDOMAIN LOCAL RECONNECTION

Ideally, the sequential mesh generator should be considered a black box when integrating it with PDR; however, AFLR has a number of requirements which not only impose constraints on PDR, but also require that modifications be made within the AFLR code itself. Due to the constraints set on PDR (which will be further explained later), this implementation will henceforth be referred to as Pseudo-constrained Parallel Data Refinement AFLR (or PsC.AFLR). In order to make the necessary modifications and properly integrate AFLR into the parallel framework, an intricate understanding of the underlying data structures and methodologies used for the initial grid generation, point insertion, element edge swapping, and optimization was required. The following steps outline the general process of PsC.AFLR:

1. Accept an input geometry.
2. Generate an initial volume mesh.
3. Construct an octree.
4. Assign subdomains to octree leaves and insert leaves into refinement queue.
5. Remove a leaf from the queue. Extract a boundary for the leaf based on original element connectivity with neighboring subdomains.
6. Call AFLR to refine the leaf.
7. Merge all neighboring leaves, located within the buffer zone, with the newly refined leaf.
8. Call AFLR to perform local reconnection on the merged data.

9. Assign updated data to the necessary leaves.
10. Repeat steps 5-9 until there are no remaining leaves in the refinement queue (executed in parallel).
11. Merge all data and call AFLR to perform final optimization.
12. Output the final mesh.

The initial volume mesh is currently generated by TetGen (steps 1 and 2) for PsC.AFLR. PsC.AFLR requires an initial volume mesh that is dense enough to satisfy boundary requirements of individual octree leaves (given that leaf boundaries are extracted from the faces of initial volume elements, discussed in more detail later in this Section). AFLR requires a smooth, simply-connected boundary when refining a domain. If the initial mesh is too coarse, there may be a tetrahedron that spans multiple leaves. In this scenario, no boundary can be extracted for each leaf if a face is spanning across them all. One solution would be to reduce the octree level (increasing the size of the leaves/subdomains), but this would reduce the overall number of leaves/subdomains, thereby reducing the amount of achievable concurrency. In order to maintain parallelism, the initial mesh must be dense enough so that faces can be extracted for every leaf containing tetrahedra. A problem observed with AFLR is that it does not always generate tetrahedra within a certain volume constraint unless the mesh undergoes a significant amount of refinement. This is counterintuitive for the purpose of generating an initial mesh. If the initial mesh undergoes too much refinement (in

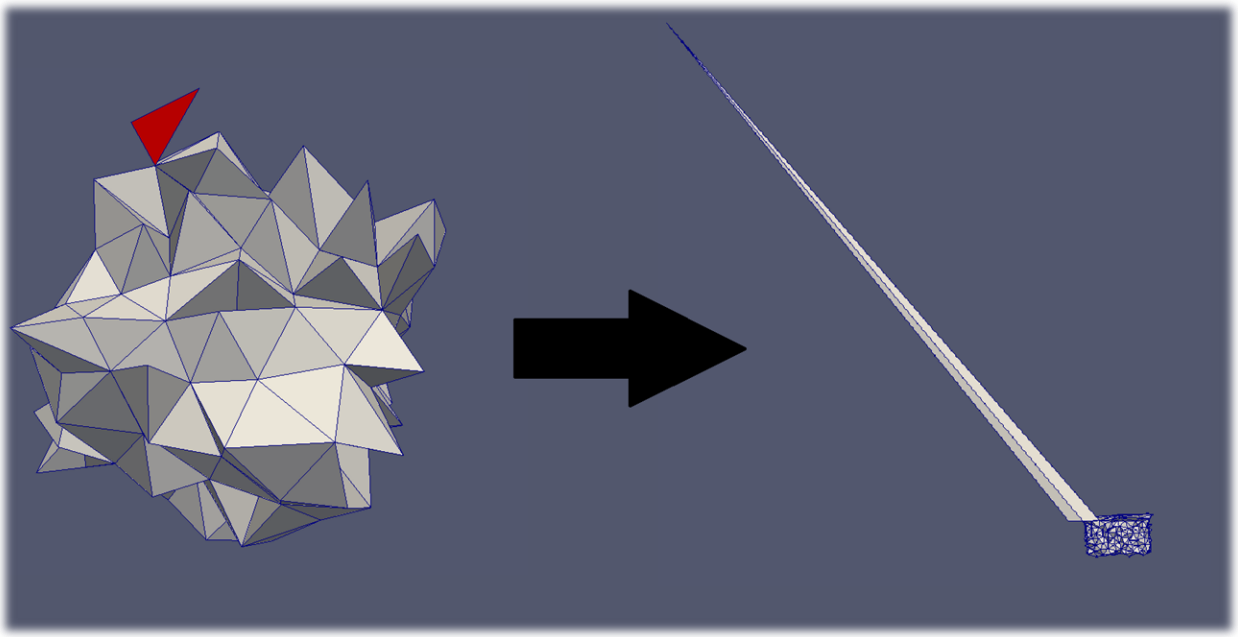

**Fig. 4. Disconnected Partition. The left side shows a subdomain of tetrahedra with one tetrahedron (in red) vertex-connected to the others (therefore considered to actually be "disconnected"). The right side shows a result generated after refining this subdomain (including the vertex-connected tetrahedron) in AFLR.**

order to satisfy the density requirement), then more runtime will be spent at this stage rather than in parallel refinement (sometimes taking hours for larger geometries) and rendering the parallel refinement futile. TetGen is currently being used to generate an initial mesh due to its low runtime and because a volume constraint can be set easily when refining a geometry.

The TetGen-generated mesh is given to PsC.AFLR and after an octree is constructed, data are assigned to octree leaves based on element barycenter. If the barycenter for an element falls within a leaf, that element is assigned to that leaf. Once data assignment is complete, it is possible that some leaves will have disconnected partitions (such as in Fig. 4), that is, elements (or groups of elements) connected to another element (or group) by only a point or edge. This is not acceptable for AFLR. Every tetrahedron must be face-connected to another (assuming that there is more than one tetrahedron of which AFLR is attempting to refine); otherwise, the results generated can vary greatly and will most definitely be incorrect (such as that on the right in Fig. 4). A method was created which checks for this problem before the refinement of a subdomain. If any disconnected tetrahedron is found, neighboring leaves are examined to find a tetrahedron with a matching face to the disconnected tetrahedron. Once found, the tetrahedron in question is reassigned to that particular leaf and removed from the leaf that is about to undergo refinement. This process is repeated until all connectivity issues are resolved.

In PsC.AFLR, a boundary must be created for a subdomain before any refinement of its elements can begin. This introduces overhead to the overall process due to the requirement that a smooth, simply-connected surface must be created for AFLR. The set of tetrahedra within a leaf is examined in isolation (as if the subdomain is the entire domain). Any face that is not shared between tetrahedra is considered to be a boundary face. It is possible to extract a boundary that contains an edge which is shared by more than two triangles. This is not acceptable for AFLR. This scenario occurs when there is a tetrahedron that has a barycenter just over the leaf boundary, causing it to be assigned to a neighboring leaf, while its neighboring tetrahedra were assigned to the primary leaf. When such an edge is found in the extracted boundary, the corresponding tetrahedron is located within the neighboring leaf, removed from that leaf, and added to the primary leaf. The boundary is extracted and examined again. This process repeats until a manifold boundary, acceptable for AFLR, is extracted.

Another implementation challenge, caused by data decomposition, is allowing subdomain boundary elements to undergo refinement. If a boundary face, shared by two leaves, undergoes refinement, then the corresponding elements within both leaves must be updated (which adds dependencies and increases overall runtime due to the required communication between the corresponding threads). Otherwise, the connectivity between the subdomains will be incorrect and the final mesh will be non-conforming. Subdomain boundary refinement is preferred so that boundary elements do not retain poor quality by the end of refinement. To solve this issue, AFLR was modified to not only accept a single set of data (points, triangles, and tetrahedra) for one leaf, but to also accept a second set of data – the set of all of its level 1 neighboring leaves. The internal interface surface is kept frozen in step 6, meaning that point insertion is not allowed on the leaf boundary. AFLR refines the individual leaf (advancing front point placement/insertion and local reconnection) but does not make any optimizations/quality improvement as the serial AFLR would. Instead, the newly refined leaf is merged with its level 1 neighbors into a super-subdomain and local reconnection is performed over the super-subdomain (thereby allowing the optimization of the primary leaf's

boundary elements). The internal interface between level 1 and level 2 neighbors remains frozen, so as to eliminate the need of updating level 2 neighbors during refinement (maintaining PDR's original method of concurrency). It is possible to have duplicate points when merging the two sets of data (leaf and its level 1 neighbors) because a neighboring leaf may contain a tetrahedron that has a point located in the primary leaf, or vice-versa. If each set of data were examined in isolation, both sets would contain the same internal interface points (due to the fact that they are subdomains which conform to one another). When these sets of data are merged, the duplicate points are removed and any tetrahedron or triangle that references these points are updated to use the same indices (all tetrahedra and faces use integer-based indices to denote which points they contain, so if two tetrahedra contain the same point, they will use the same index to reference that point). The removal of duplicate points is necessary as they are not permitted by AFLR.

Once local reconnection over the super-subdomain has completed, this refined data is returned to PDR and is assigned to octree leaves. No points are deleted during refinement, so only new points are added to leaves. All previous tetrahedra data within the leaf and its level 1 neighbors are deleted. The new tetrahedra are assigned to leaves. Having undergone swapping, a tetrahedron will have a different barycenter. It is possible for the barycenter to move just enough to be assigned to a level 2 neighbor. If a level 2 neighbor must be updated during refinement, then this limits parallelism and conflicts with PDR's method of concurrency. A thread should only refine a leaf and its level 1 neighbors without allowing any changes to propagate beyond the level 1 region. If this situation occurs, the tetrahedron is assigned to a leaf, that contains a tetrahedron with a matching face, and that is a level 1 neighbor of both the level 2 leaf and of the primary leaf.

A function was also added to AFLR which accepts a set of data and performs quality improvement/optimization on it (sliver removal and local reconnection). After all leaves have undergone refinement, their data are combined into a single set and passed into this function to be optimized sequentially (so that PsC.AFLR performs a final optimization step over the entire domain just as serial AFLR does).

## 3.2 INTEGRATION OF AFLR WITH PDR WITHOUT SUPER-SUBDOMAIN LOCAL RECONNECTION

The results of the previous implementation can be seen in Section 4.1. An alternate implementation was created in order to reduce the overall runtime induced by the previous implementation and to remove several of its constraints (to be discussed later). Previously, level 1 neighbor leaves were merged with the primary leaf and local reconnection was performed over the super-subdomain in order to improve the quality of the primary leaf's subdomain boundary elements. While this does offer some improvement of the end stability given by PsC.AFLR, it is not essential. If internal interface elements remain frozen throughout refinement and local reconnection is not performed on these elements, the quality of the final mesh generated by PsC.AFLR is still comparable to that generated by serial AFLR and falls within the operational limits of CFD solvers such as FUN3D and SU2 [15] [16]. The process outline of PsC.AFLR was modified to be the following:

1. Accept an input geometry.

2. Generate an initial volume mesh.
3. Construct an octree.
4. Assign subdomains to octree leaves and insert leaves into refinement queue.
5. Remove a leaf from the queue. Extract a boundary for the leaf based on original element connectivity with neighboring subdomains.
6. Call AFLR to refine the leaf.
7. Assign updated data to the leaf.
8. Repeat steps 5-7 until there are no remaining leaves in the refinement queue (executed in parallel).
9. Merge all data and call AFLR to perform final optimization.
10. Output the final mesh.

This modification allowed for the removal of several PsC operations that held significant contributions to the overall runtime of the previous implementation. These include the: removal and addition of level 2 tetrahedra before and after refinement, extraction of level 1 neighbor leaves to pass into AFLR, merging of level 1 neighbors with the primary leaf, and data assignment for level 1 neighbor leaves. The removal and addition of level 2 tetrahedra involve the temporary removal of tetrahedra from a subdomain that contain level 2 points, and the later re-assignment of these tetrahedra back to their respective leaves after refinement. This was necessary as level 1 neighbor boundary elements may contain points that are assigned to level 2 neighbors, in which case these points were not packed and migrated to the node where the refinement process was about to begin. Only a leaf and its level 1 neighbor data (that is, data within the level 1 neighbor leaves) are packed for migration, which follows PDR's method of concurrency. The temporary removal of these tetrahedra was acceptable because these elements would have either already undergone local reconnection (since they are boundary elements) or would have done so in a later refinement process for another leaf. These modifications also removed the need to search for duplicate points within the single set of merged data (since there is no longer a set of merged data) and the need to find an appropriate assignment for a tetrahedron that had a shifting barycenter due to the super-subdomain local reconnection (assigning what was once a level 1 tetrahedron, which now has a level 2 barycenter, to a level 1 neighbor leaf). The removal of these processes introduced significant improvements on the overall end-user productivity of PsC.AFLR, which will be seen in Chapter 4.

## 3.3 INTEGRATION OF PsC.AFLR onto PREMA

Parallelism was achieved by fully integrating PsC.AFLR onto a runtime system called the Parallel Runtime Environment for Multi-computer Applications (PREMA) 2.0 [17]. PREMA 2.0 is a parallel runtime system developed to support adaptive and irregular applications. It is capable of running in both shared and distributed memory. PREMA provides a globally addressable namespace, message forwarding, and data migration capabilities by using constructs called mobile objects and mobile pointers. Mobile objects are user-defined data objects that may encapsulate data not residing in contiguous memory (a leaf and its level 1 neighbors). A mobile pointer is a unique identifier created for each mobile object that can be used by the system even if the object has migrated to different ranks. This enables a rank to send a message to a specific mobile object and execute a user-defined function on it, regardless of its location. In PsC.AFLR, a master-worker model is used, where steps 1-4 of both outlines are executed by the master thread and

steps 5-9 and 5-7 are executed by worker threads in parallel for the first and second implementations, respectively. During run time, the PsC.AFLR method exposes data decomposition information (number of leaves/subdomains waiting to be refined in the queue) to the underlying run-time system. PDR essentially informs PREMA that a leaf may undergo refinement if it and its level 1 neighbors are not currently under use in another leaf's refinement process. The master thread will send a message to the corresponding mobile pointer (representative of the leaf and its set of neighbors that are ready for refinement), essentially informing PREMA to execute a refinement function given the mobile object's data. PREMA 2.0 monitors the load of the system and performs migration (of the leaf and neighbor data) to an available worker 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.

# 4. RESULTS

PsC.AFLR was executed and tested on the Turing cluster at ODU [18]. Each node runs Red Hat Enterprise Linux Server release 6.10 with 32 cores per node. Each CPU is an Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz with 128 GB of memory. GCC version 6.3.0 and MPICH version 3.1.3 are used for compilation and execution.

## 4.1 RESULTS OF IMPLEMENTATION WITH SUPER-SUBDOMAIN LOCAL RECONNECTION

All data presented are from the refinement time of both applications (PsC.AFLR and serial AFLR) and does not include initial volume mesh generation time or end optimization time. These two processes have introduced challenges in the parallel implementation that will be addressed in future work. Fig. 5 shows a geometry of a horn bulb that was used for testing with both PsC.AFLR and serial AFLR. The horn bulb geometry contains 1,062,042 surface elements. The number of tetrahedra within the initial volume mesh used were 3,773,233 and 1,655,568 for PsC.AFLR and serial AFLR, respectively. The number of elements within the initial mesh for PsC.AFLR are greater due to preprocessing requirements for data decomposition. The nature of these requirements and their effect on the final mesh is explained later (and will be further explored in the future). The octree used in PsC.AFLR is at level 4 (containing 4,096 leaves, or subdomains). The final number of tetrahedra generated for PsC.AFLR is approximately 13 million and 116,130,365 for serial AFLR. Serial AFLR's refinement time is 16,101 seconds and its refinement speed is 7,212 elements/sec. Table 1 shows the performance achieved by PsC.AFLR, in addition to its refinement speed for each number of cores used.

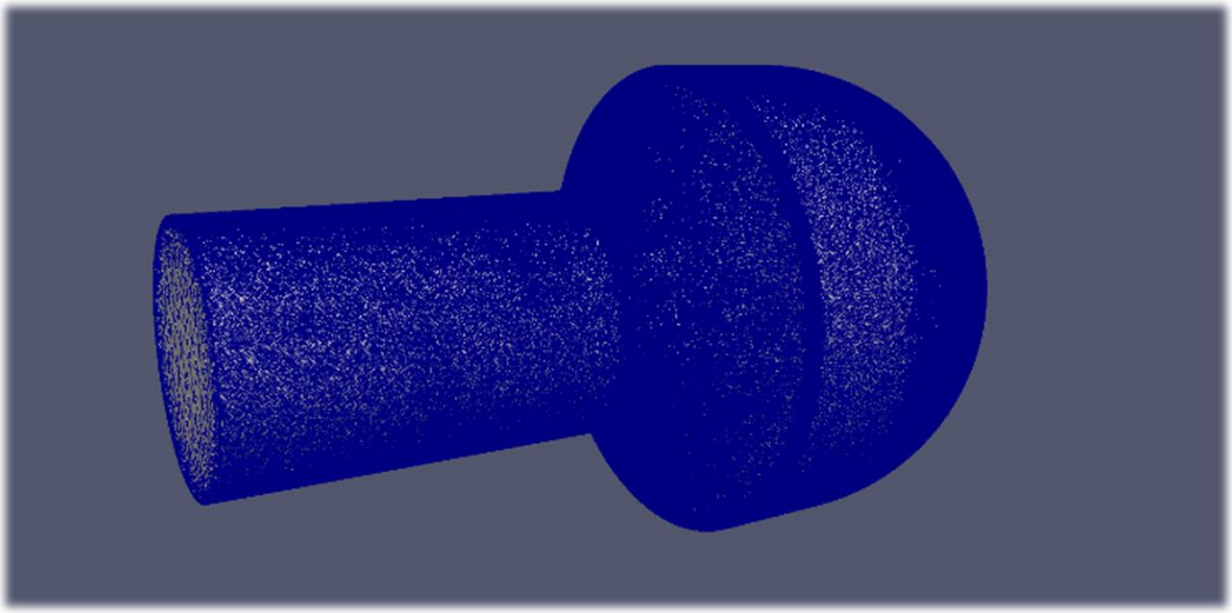

**Fig. 5. Horn Bulb Geometry with 1,062,042 Surface Elements.**

**Table 1 PsC.AFLR Horn Bulb Performance Results from First Implementation. The performance data are based on the horn bulb geometry defined by 1,062,042 surface elements where PsC.AFLR generates about 13 million tetrahedra and serial AFLR (in red in the table) generates about 116 million tetrahedra.**

| # of Cores | 1 | | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| # of generated elements | 116,130,365 | 13,006,043 | 13,425,162 | 13,144,072 | 12,984,778 | 13,005,792 | 12,872,925 | 13,154,378 |
| Runtime (sec) | 16,102 | 6,186 | 3,144 | 1,626 | 1,040 | 750 | 682 | 654 |
| Refinement Speed (Elements per sec) | 7,212 | 2,103 | 4,137 | 7,996 | 12,500 | 17,344 | 19,078 | 19,886 |

Based on the results observed in Table 1, the end-user productivity increases as the number of cores are increased and PsC.AFLR outperforms serial AFLR (in both total runtime and refinement speed). PsC.AFLR is capable of generating a larger number of elements per second than the serial code on just 4 cores. It also outperforms serial AFLR by about 2.5 times on 16 cores. Although PsC.AFLR produces a final mesh with fewer elements than the serial software, its final mesh maintains satisfactory quality in comparison as shown in Fig. 6. The minimum dihedral angle of an element in the final mesh is 3.47 degrees while the maximum is 172.58 degrees with PsC.AFLR (as opposed to serial AFLR having a minimum of 7.3 degrees and a maximum of 164.57 degrees). A percentage breakdown of the average time spent executing PsC operations and AFLR operations (within the context of PsC.AFLR) is shown in Fig.

7. The average time for each PsC operation was gathered, compared to the total time spent executing PsC-specific operations, in Fig. 8, and the same is shown for AFLR (within the context of PsC.AFLR) in Fig. 9. This data was gathered by taking the average times among the master and worker processes for each of 100 runs, executed for each number of cores (100 runs for 1 core, 100 runs for 2 cores, etc.). Fig. 7 shows that approximately 70% of total runtime is spent executing PsC PDR operations on average across all numbers of core runs. In Fig. 8, L2 Tet Removal and Addition reference the temporary removal of tetrahedra from a subdomain that contain level 2 points, and the later re-assignment of these tetrahedra back to their respective leaves after refinement, as described in Section 3.2. Partition Swap represents the time spent detecting disconnected partitions and re-assigning them to neighboring leaves. This is the most time-consuming PsC PDR operation. Neighbor Extraction is the time spent merging all level 1 neighbor data into a single set of data to be passed as a parameter to AFLR and Data Assignment represents the time spent assigning data to octree leaves after refinement. Fig. 9 shows that of all AFLR operations, the most time is spent merging the leaf and its level 1 neighbors.

A breakdown of the average time spent making data dissemination decisions by PREMA and time spent executing load balancing operations is shown in Fig. 10. Load balancing operations include the packing, unpacking, and migration of data between parallel processes. Data dissemination decisions made by PREMA account for the sending of messages between the master and worker processes, requests for work made by the workers, and the master determining which workers to assign work based on their current workloads. Between load balancing and PREMA, the majority of PsC.PDR's total runtime is spent making data dissemination decisions. This happens simultaneously while PsC PDR and AFLR operations are being executed, due to PREMA's asynchronous message reception and computation execution being handled in separate threads. When executed on a smaller number of cores, load balancing and PREMA do not make up the total runtime but as more cores are used, they make up the entire execution time of PsC.AFLR, showing a gain in the amount of parallelism leveraged.

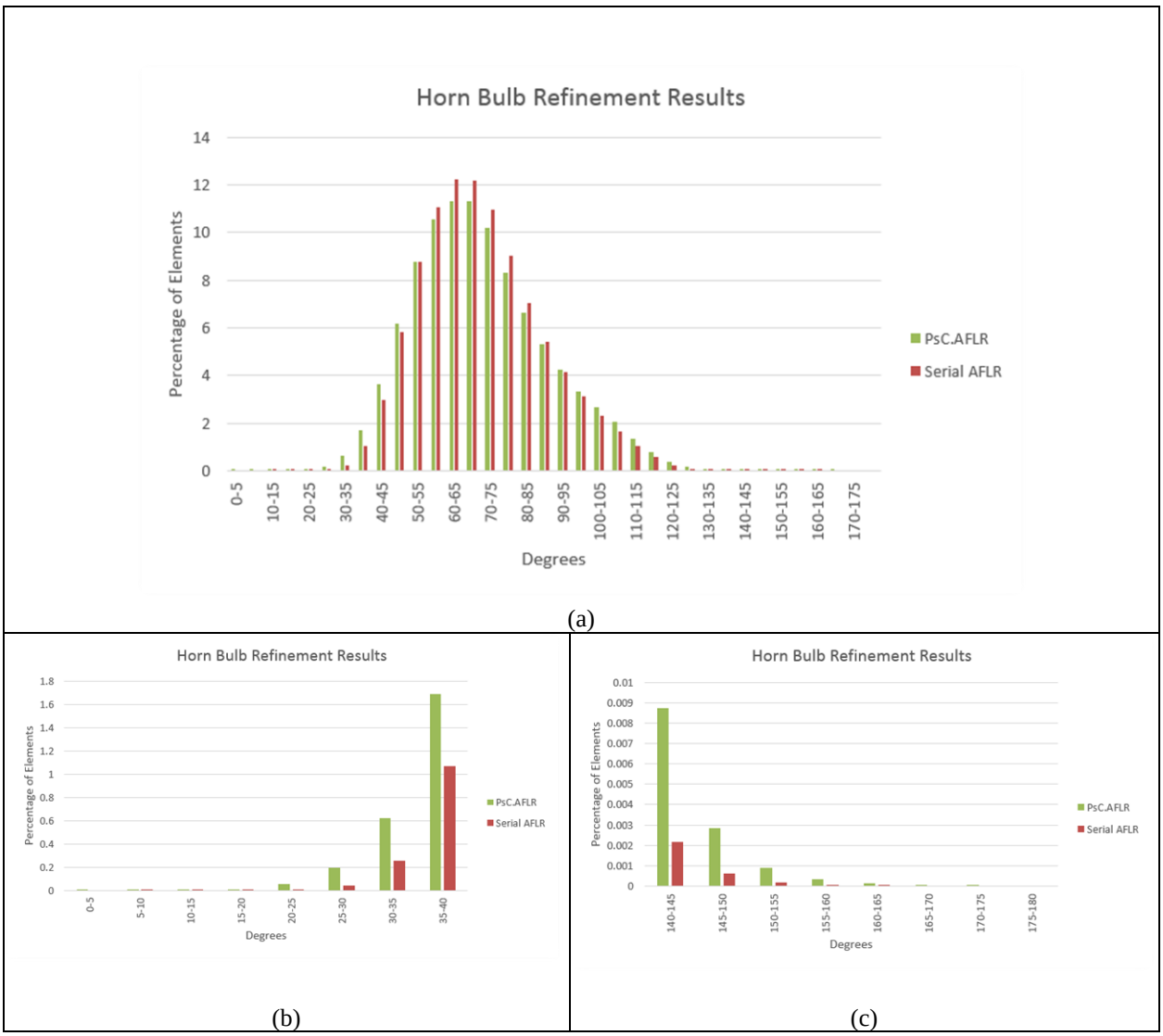


**Fig. 6. First Implementation Horn Bulb Dihedral Angle Results. The dihedral angle statistics of the horn bulb final meshes generated by PsC.AFLR and serial AFLR are shown and compared in (a). (b) and (c) show the lower (0 to 40 degrees) and upper (140 to 180 degrees) dihedral angle statistics, respectively.**

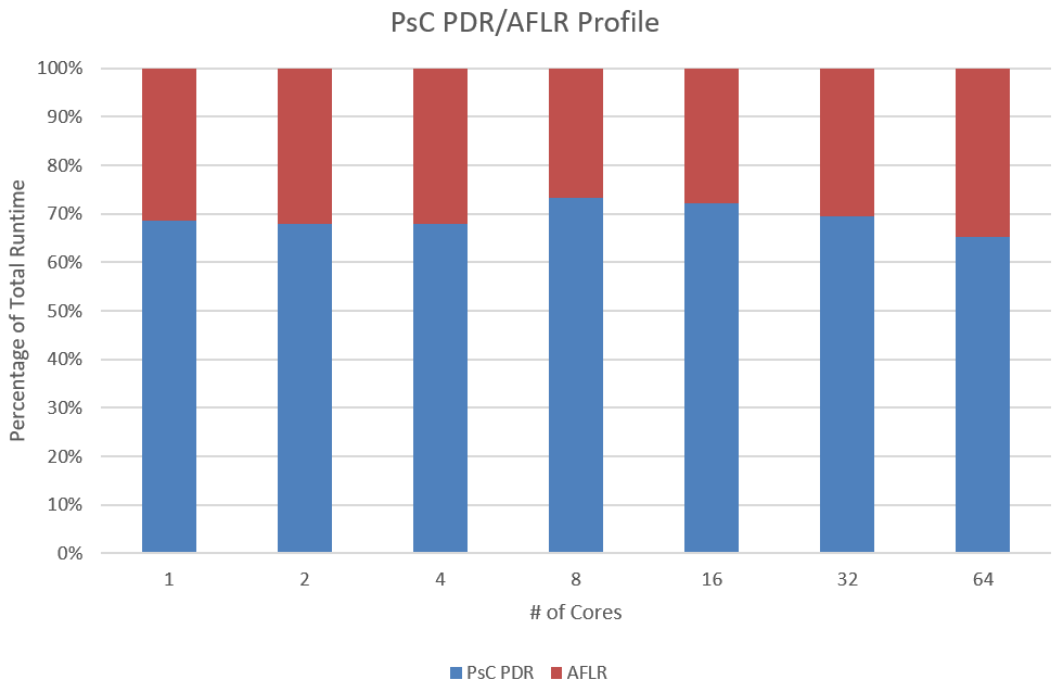


**Fig. 7. First Implementation PsC PDR/AFLR Profile. A percentage breakdown of the average time spent executing PsC PDR and AFLR operations (within the context of PsC.AFLR) in reference to the total runtime is shown.**

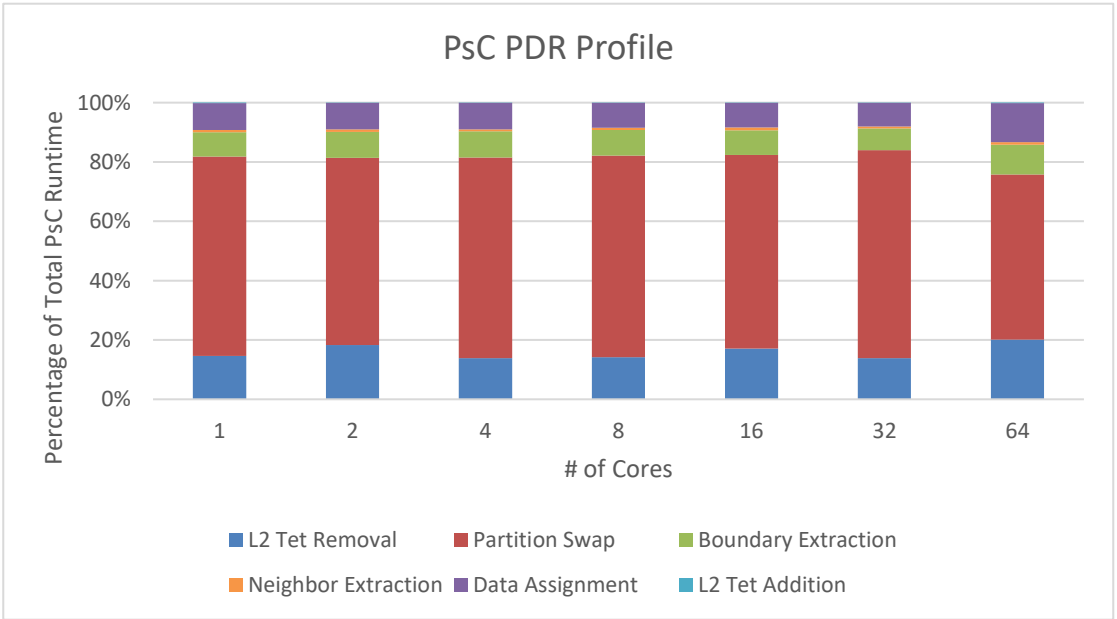


**Fig. 8. First Implementation PsC PDR Profile. A breakdown of PsC operations is presented. The average percentage of time spent in each operation in reference to the total amount of time spent performing PsC operations is shown.**

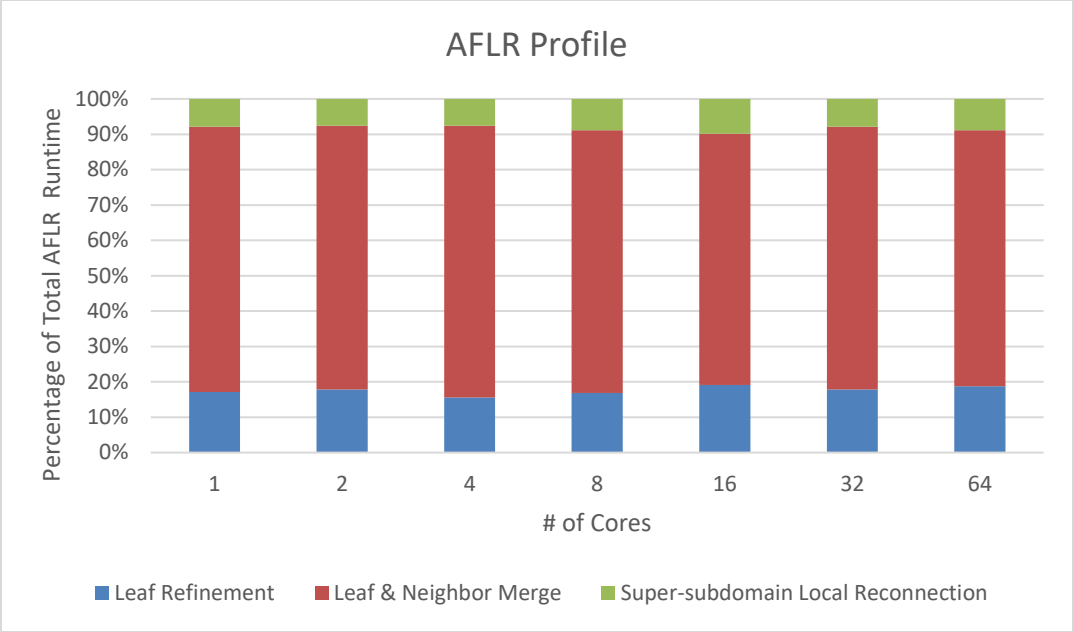


**Fig. 9. First Implementation AFLR Profile. A breakdown of AFLR operations (within the context of PsC.AFLR) is presented. The average percentage of time spent in each operation in reference to the total amount of time spent performing AFLR operations is shown.**

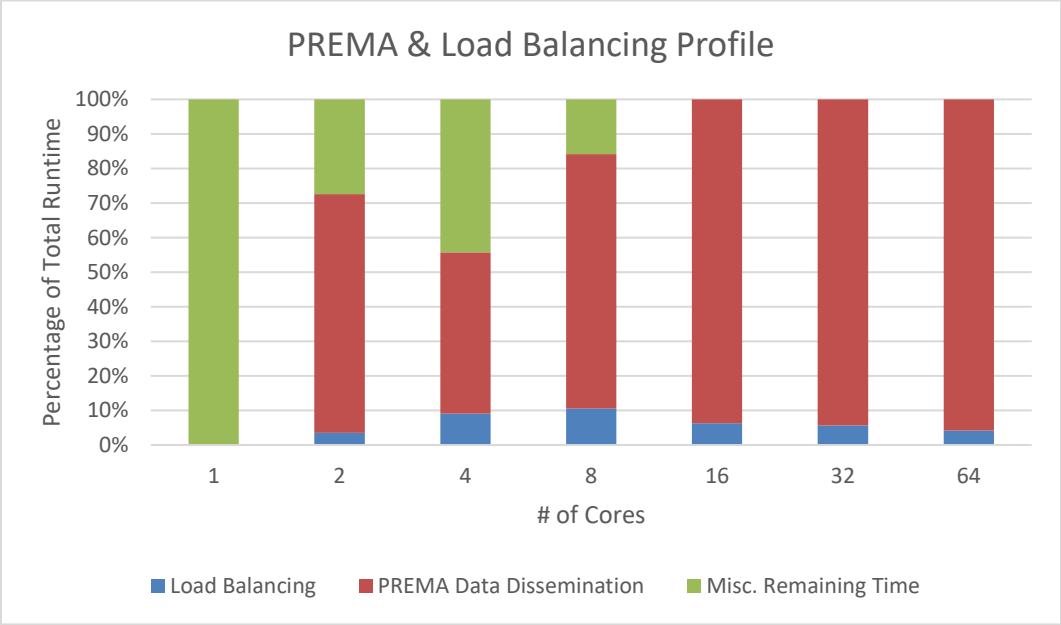


**Fig. 10. First Implementation PREMA & Load Balancing Profile. For the first implementation of PsC.AFLR, a breakdown of the average time spent making data dissemination decisions by PREMA and time spent executing load balancing operations is shown.**

## 4.2 RESULTS OF IMPLEMENTATION WITHOUT SUPER-SUBDOMAIN LOCAL RECONNECTION

In the first implementation during refinement, level 1 neighbor leaves were merged with their corresponding primary leaf and local reconnection was performed over the super-subdomain in order to improve the quality of the primary leaf's subdomain boundary elements. As stated previously, if internal interface elements remain frozen throughout refinement and local reconnection is not performed on these elements, the quality of the final mesh generated by PsC.AFLR is still comparable to that generated by serial AFLR and falls within the operational limits of CFD solvers. Fig. 11 compares the dihedral angle statistics of the meshes output when performing local reconnection over super-subdomains and without performing this operation (simply skipping the super-subdomain local reconnection step within AFLR). The geometry under refinement in this test is the aforementioned horn bulb. The minimum dihedral angles of the meshes generated are 7.3, 3.47, and 2.25 degrees for serial AFLR, PsC.AFLR (with super-subdomain local reconnection), and PsC.AFLR (without super-subdomain local reconnection), respectively. The maximum dihedral angles are 164.57, 172.58, and 168.48 degrees for serial AFLR, PsC.AFLR (with super-subdomain local reconnection), and PsC.AFLR (without super-subdomain local reconnection), respectively. These results prompted the creation of the second implementation (and the subsequent modification/removal of the aforementioned PsC PDR processes).

The aforementioned horn bulb was also used for performance testing of the second implementation. Table 2 shows the improvement in performance achieved by the updated PsC.AFLR with the aforementioned modifications. Based on these results, the end-user productivity improved by about 5 times on average across all numbers of cores. PsC.AFLR is capable of generating a larger number of elements per second than the serial code on each number of cores (even on one core, which will be explained in the analysis in Chapter 5) and outperforms serial AFLR. Although PsC.AFLR produces a final mesh with fewer elements than the serial software, its final mesh maintains satisfactory quality in comparison as shown in Fig. 12. The minimum dihedral angle of an element in the final mesh is 3.52 degrees while the maximum is 165.71 degrees with PsC.AFLR (as opposed to serial AFLR having a minimum of 7.3 degrees and a maximum of 164.57 degrees).

A rocket geometry, pictured in Fig. 13, was also tested with both PsC.AFLR and serial AFLR. The rocket geometry originally contained transparent/embedded surfaces. These surfaces were specifically the plume, engine exhaust, and nozzle exhaust. Due to the limited capabilities of this implementation and the fact that it currently only refines manifold, genus zero geometries, these surfaces were removed from the rocket geometry before testing. The rocket geometry contains 1,030,692 surface elements. The number of tetrahedra within the initial volume mesh used were 2,776,378 and 1,533,775 for PsC.AFLR and serial AFLR, respectively. The octree used in PsC.AFLR is again at level 4 (containing 4,096 leaves, or subdomains). The final number of tetrahedra generated for the rocket geometry by PsC.AFLR is approximately 38 million and 146,225,745 for serial AFLR. Serial AFLR's refinement time is 16,646 seconds and its refinement speed is 8,785 elements/sec. Table 3 shows the performance achieved by PsC.AFLR, in addition to its refinement speed for each number of cores used, when refining the rocket geometry.

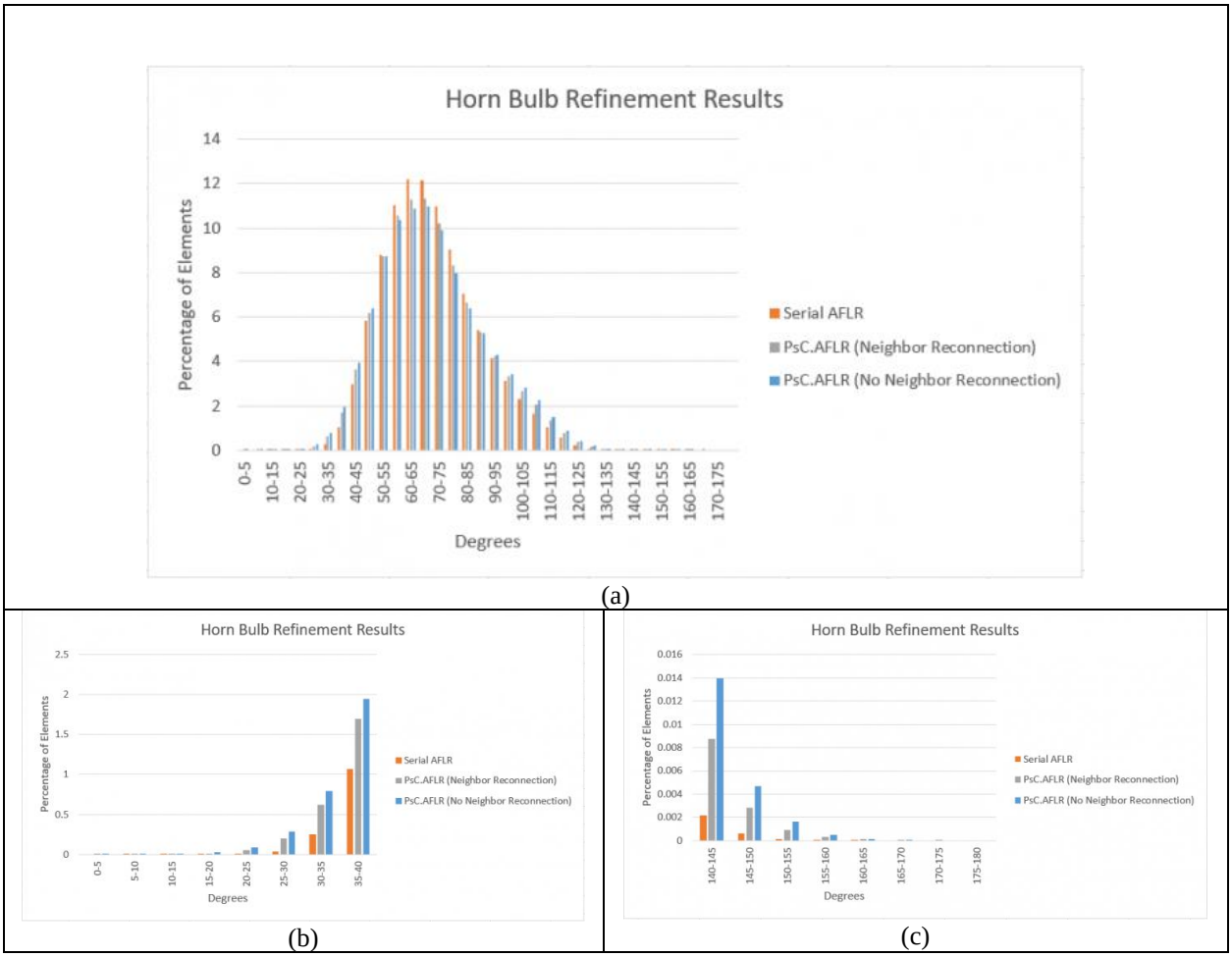


**Fig. 11. Dihedral Angle Results Comparison between First and Second Implementations. The dihedral angle statistics of the horn bulb meshes generated when performing local reconnection over super-subdomains (merged leaf and level 1 neighbor tetrahedra), and without performing this operation, are shown and compared in (a). (b) and (c) show the lower (0 to 40 degrees) and upper (140 to 180 degrees) dihedral angle statistics, respectively.**

**Table 2 PsC.AFLR Horn Bulb Performance Results from Second Implementation. The performance data are based on the horn bulb geometry defined by 1,062,042 surface elements where PsC.AFLR generates about 14 million tetrahedra and serial AFLR (in red in the table) generates about 116 million tetrahedra**

| # of Cores | 1 | | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| # of generated elements | 116,130,365 | 14,666,874 | 14,698,453 | 14,478,622 | 14,730,358 | 14,455,376 | 14,453,916 | 14,549,829 |
| Runtime (sec) | 16,102 | 856 | 573 | 401 | 224 | 175 | 175 | 278 |
| Refinement Speed (Elements per sec) | 7,212 | 17,130 | 25,652 | 36,073 | 65,902 | 82,418 | 82,504 | 52,415 |

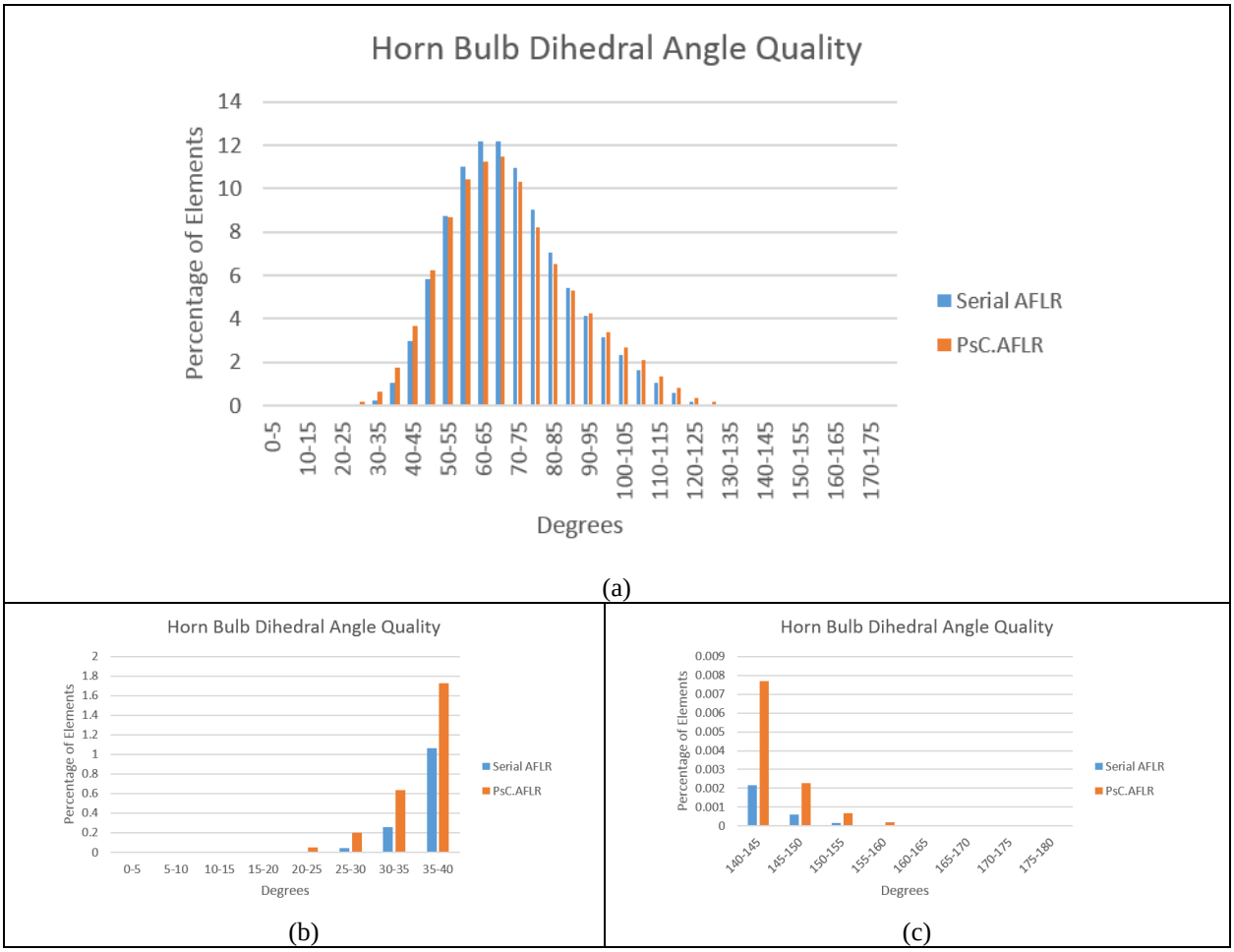


**Fig. 12. Second Implementation Horn Bulb Dihedral Angle Results. The dihedral angle statistics of the horn bulb final meshes generated by the second implementation of PsC.AFLR and serial AFLR are shown and compared in (a). (b) and (c) show the lower (0 to 40 degrees) and upper (140 to 180 degrees) dihedral angle statistics, respectively.**

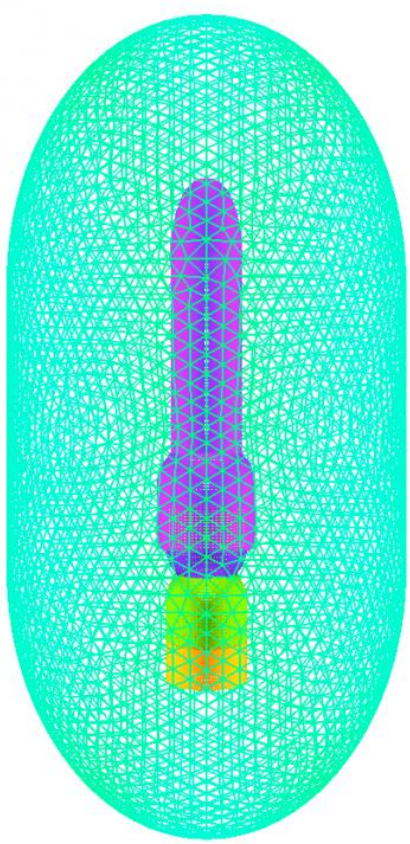

**Fig. 13. Rocket Geometry with 1,030,692 Surface Elements.**

**Table 3 PsC.AFLR Rocket Performance Results from Second Implementation. The performance data are based on the rocket geometry defined by 1,030,692 surface elements where PsC.AFLR generates about 38 million tetrahedra and serial AFLR (in red in the table) generates about 146 million tetrahedra**

| # of Cores | 1 | | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| # of generated elements | 146,225,745 | 38,049,568 | 39,275,974 | 37,782,405 | 37,787,646 | 38,051,434 | 37,912,838 | 37,742,444 |
| Runtime (sec) | 16,646 | 1,643 | 1,290 | 1,420 | 1,152 | 1,135 | 1,007 | 1,339 |
| Refinement Speed (Elements per sec) | 8,785 | 23,156 | 30,437 | 26,608 | 32,739 | 33,524 | 37,641 | 28,312 |

Based on the results observed in Table 3, PsC.AFLR outperforms serial AFLR (in both total runtime and refinement speed) and is capable of generating a larger number of elements per second than the serial code (even on one core, which will also be explained in the analysis in Chapter 5). Although PsC.AFLR produces a final mesh with fewer elements than the serial software, its final mesh maintains satisfactory quality in comparison as shown in Fig. 14. The minimum dihedral angle of an element in the final mesh is 2.27 degrees while the maximum is 164.97 degrees with PsC.AFLR (as opposed to serial AFLR having a minimum of 0.01 degrees and a maximum of 179.98 degrees). A percentage breakdown of the average time spent executing PsC operations and AFLR operations (within the context of PsC.AFLR) is shown in Fig. 15. The average time for each PsC operation was gathered, compared to the total time spent executing PsC-specific operations, in Fig. 16. This data was gathered by also taking the average times among the master and worker processes for each of 100 runs, executed for each number of cores (100 runs for 1 core, 100 runs for 2 cores, etc.). Fig. 15 shows that the time spent executing PsC PDR operations varies on different numbers of cores but lessens as the numbers of cores are increased. Overall, PsC PDR operations still dominate a percentage of the runtime but is an improvement on the percentages seen in Fig. 7, from the previous implementation. In Fig. 16, Boundary Extraction is now the most time-consuming PsC PDR operation, in contrast to Partition Swap being the most time-consuming operation in the previous implementation. The reduction in the runtime of the Partition Swap process is attributed to the fact that super-subdomains no longer undergo local reconnection. Once disconnected partitions are reassigned, the tetrahedra within those corresponding leaves will never need to be reassigned again (due to disconnected partitions). Tetrahedra that were normally affected by local reconnection within the level 1 neighbor leaves in the first implementation are no longer altered during the refinement process of a primary leaf. The barycenters of these tetrahedra remain unaltered, thereby removing the need to assign the tetrahedra to new leaves (which would possibly create new disconnected partitions). Additionally, AFLR will not generate a domain that contains any disconnected partitions, so all generated elements are simply assigned to the primary leaf and are guaranteed to be face-connected (conforming to one another and to the elements in neighboring subdomains), assuring that there are no disconnected partitions that would need to be reassigned later during another leaf's refinement process.

A breakdown of the average time spent making data dissemination decisions by PREMA and time spent executing load balancing operations is shown in Fig. 17. Between load balancing and PREMA, more time is spent in load balancing when PsC.AFLR is executed on a smaller number of cores while more time is spent making data dissemination decisions when the program is executed on a larger number of cores. Although load balancing and PREMA do not make up the entire execution time of PsC.AFLR, one can see that the time spent executing these operations increases as more cores are used and is executed simultaneously with PsC PDR and AFLR operations in about 90% of the total runtime when run on 32 cores (again showing a gain in the amount of parallelism leveraged as more cores are used).

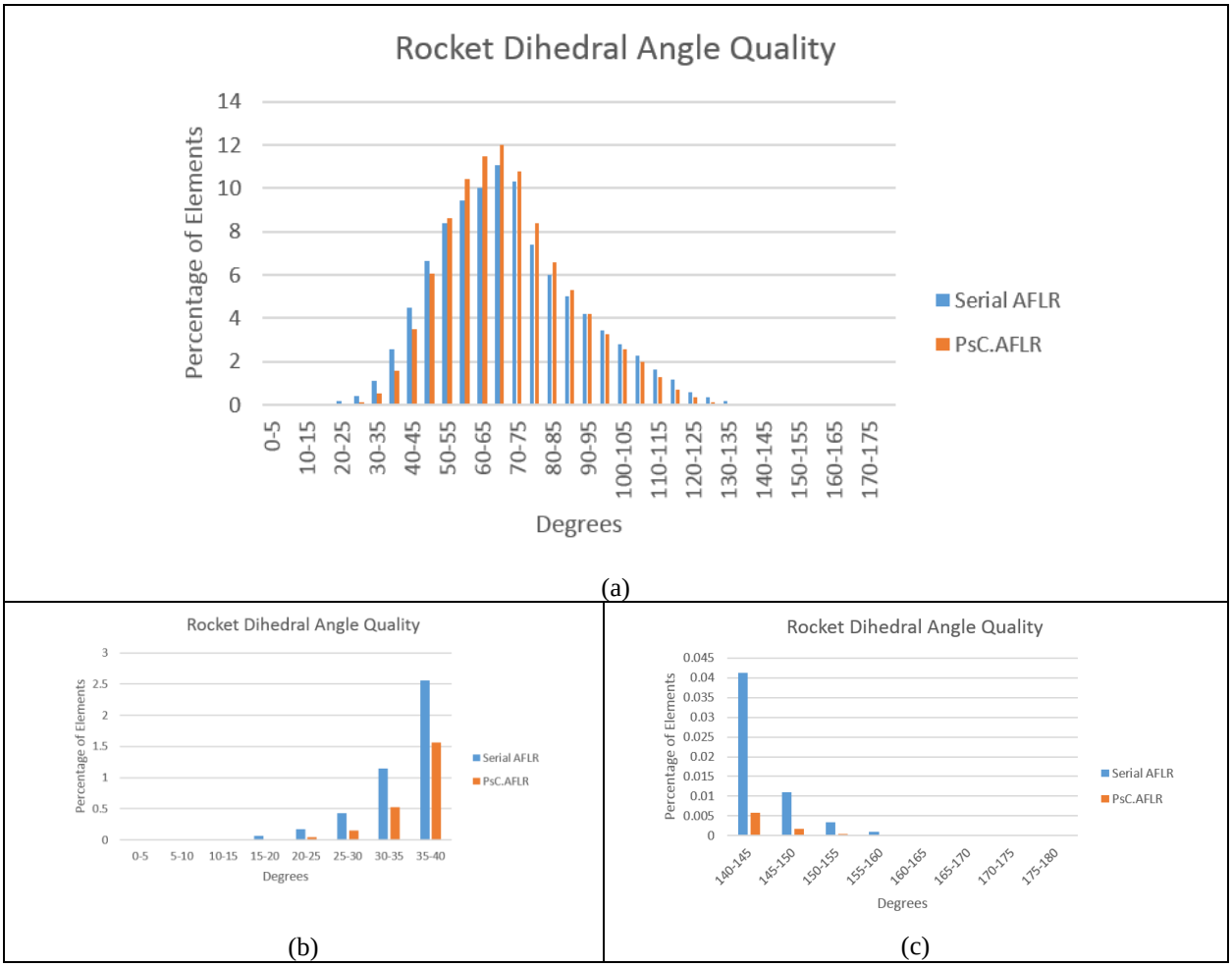


**Fig. 14. Second Implementation Rocket Dihedral Angle Results. The dihedral angle statistics of the rocket final meshes generated by the second implementation of PsC.AFLR and serial AFLR are shown and compared in (a). (b) and (c) show the lower (0 to 40 degrees) and upper (140 to 180 degrees) dihedral angle statistics, respectively.**

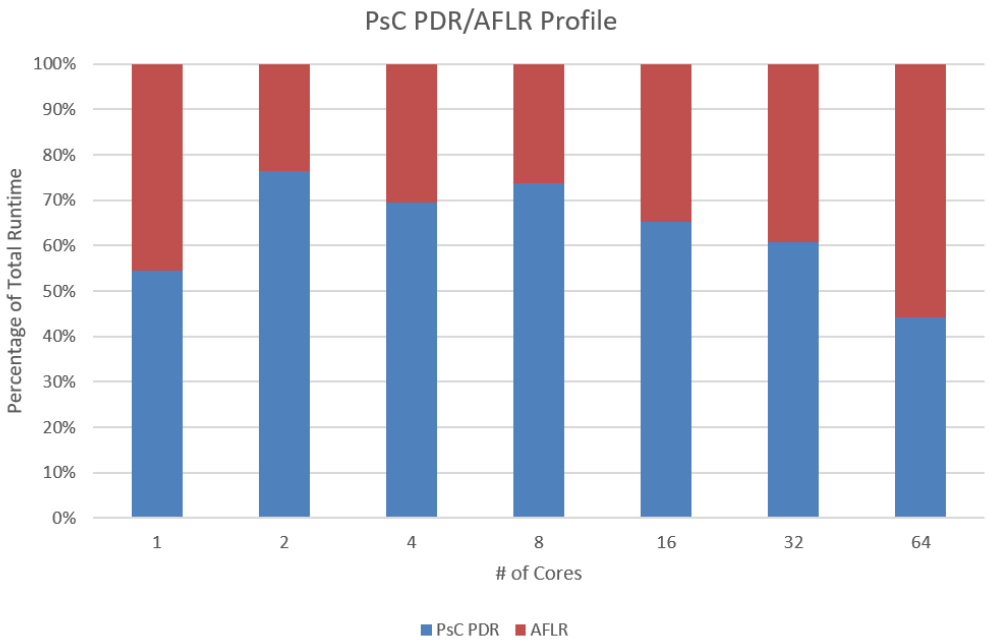


**Fig. 16. Second Implementation PsC PDR/AFLR Profile. A percentage breakdown of the average time spent executing PsC PDR and AFLR operations (within the context of PsC.AFLR) in reference to the total runtime is shown for the second PsC.AFLR implementation.**

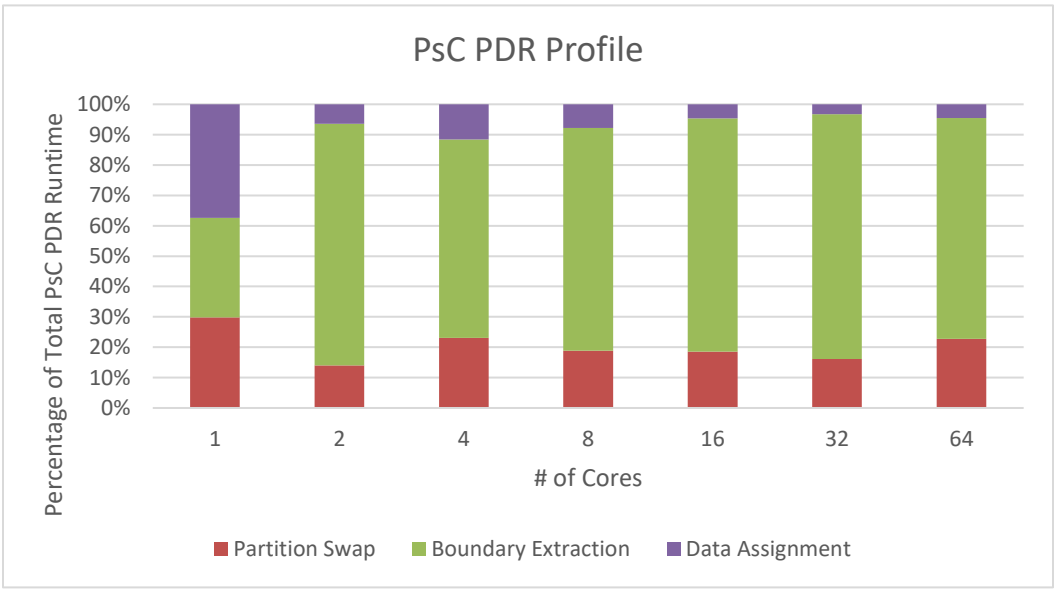


**Fig. 15. Second Implementation PsC PDR Profile. A breakdown of PsC PDR operations within the second implementation of PsC.AFLR is presented. The average percentage of time spent in each operation in reference to the total amount of time spent performing PsC PDR operations is shown.**

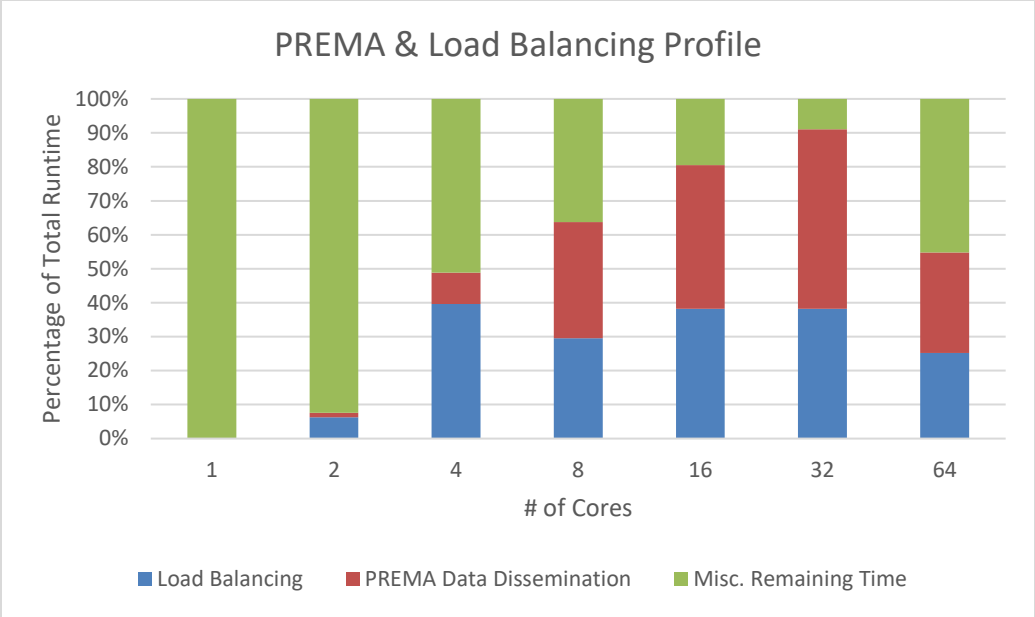


**Fig. 17. Second Implementation PREMA & Load Balancing Profile. For the second implementation of PsC.AFLR, a breakdown of the average time spent making data dissemination decisions by PREMA and time spent executing load balancing operations is shown.**

# 5. ANALYSIS

Although it offers good end-user productivity, PsC.AFLR suffers in its capability (in both implementations) to generate meshes with the same level of density or quality as that of the serial AFLR software due to the constraints set by subdomain boundaries that are required to not only successfully execute AFLR but also determine its execution behavior given the nature of the advancing front method. PsC.AFLR and serial AFLR essentially produce two different final volume meshes when refining the same geometry. The final number of elements between the two applications is different for two reasons:

1) PsC.AFLR extracts a surface triangulation for a subdomain based on the initial volume elements and these surface elements remain frozen throughout the refinement of that particular leaf. In the first implementation, the surface elements undergo local reconnection once the newly refined leaf data is merged with the elements within the surrounding level 1 neighbor leaves, but they do not undergo point insertion (which would create more elements), in this implementation or the second, in order to maintain conformity between the subdomains.
2) Serial AFLR was run in its default state (no parameter adjustments). This is not satisfactory for PsC.AFLR when refining subdomains because it uses an initial mesh generated by TetGen (due to initial volume mesh generation challenges with AFLR) and therefore utilizes the point distribution of the subdomain surfaces (from the coarse TetGen mesh) rather than the external dense surface of the geometry (which serial AFLR uses). This point distribution directly affects the number of elements created. AFLR's point distribution function was abstracted to be used before refinement within PsC.AFLR so that it can utilize this value based on the external surface. The point distribution is explicitly set to this value as a parameter for all subdomains. Henceforth, PsC.AFLR is able to generate a denser mesh of higher quality although it extracts coarse surfaces based on its initial TetGen-generated mesh. Even with this adjustment though, these constraints do not allow PsC.AFLR to generate a mesh as dense as that generated by serial AFLR; however, PsC.AFLR still maintains satisfactory quality.

The final meshes generated by PsC.AFLR and serial AFLR cannot easily be compared because the serial AFLR-generated meshes are much larger than those generated by PsC.AFLR. AFLR does not perform as well within the context of PsC.AFLR due to the constraints set by its own requirements. AFLR's boundary requirement reduces the number of elements that AFLR creates in PsC.AFLR due to the coarse boundary extracted from the TetGen-generated mesh. Adjusting the point distribution improves the subdomain density, but ultimately does not allow PsC.AFLR to generate a mesh as dense as serial AFLR, which also affects the final dihedral angle quality of its elements.

Further evidence of how the coarse boundary elements affect subdomain density can be seen in Fig. 18, which shows shape and size quality metrics, as defined in [19], between the volumes generated for the horn bulb geometry by PsC.AFLR and serial AFLR. The minimum values for the shape and size metric are $2.8*10^{-5}$ in the mesh generated by PsC.AFLR and $1.4*10^{-7}$ in the mesh generated by serial AFLR. Serial AFLR is expected to have elements with a shape and size metric that is smaller due to its much larger density/volume of elements (8 times larger) and its better dihedral angle quality. One can observe that the density of the subdomains, within the volume generated by PsC.AFLR, is affected by the poor quality of the shape and relative size of the subdomain boundary elements. These

surface elements constrain the interior elements and prevent the subdomains from becoming as dense as that seen in the volume generated by serial AFLR. This is also why PsC.AFLR seems to have better performance than serial AFLR even when executed on one core, because PsC.AFLR essentially generates a different volume. This issue stems from the fact that PsC.AFLR is dependent on the coarse surfaces extracted from the initial volume mesh. PsC.AFLR redefines the problem by focusing on subdomains (surrounded by these coarse surfaces) individually rather than on the entire domain. The elements within these subdomains are therefore refined differently (to a coarser level of density than that of the serial-output mesh in order to keep subdomain boundary elements frozen and to maintain their shape and relative size) than they would be if the domain was refined as a whole.

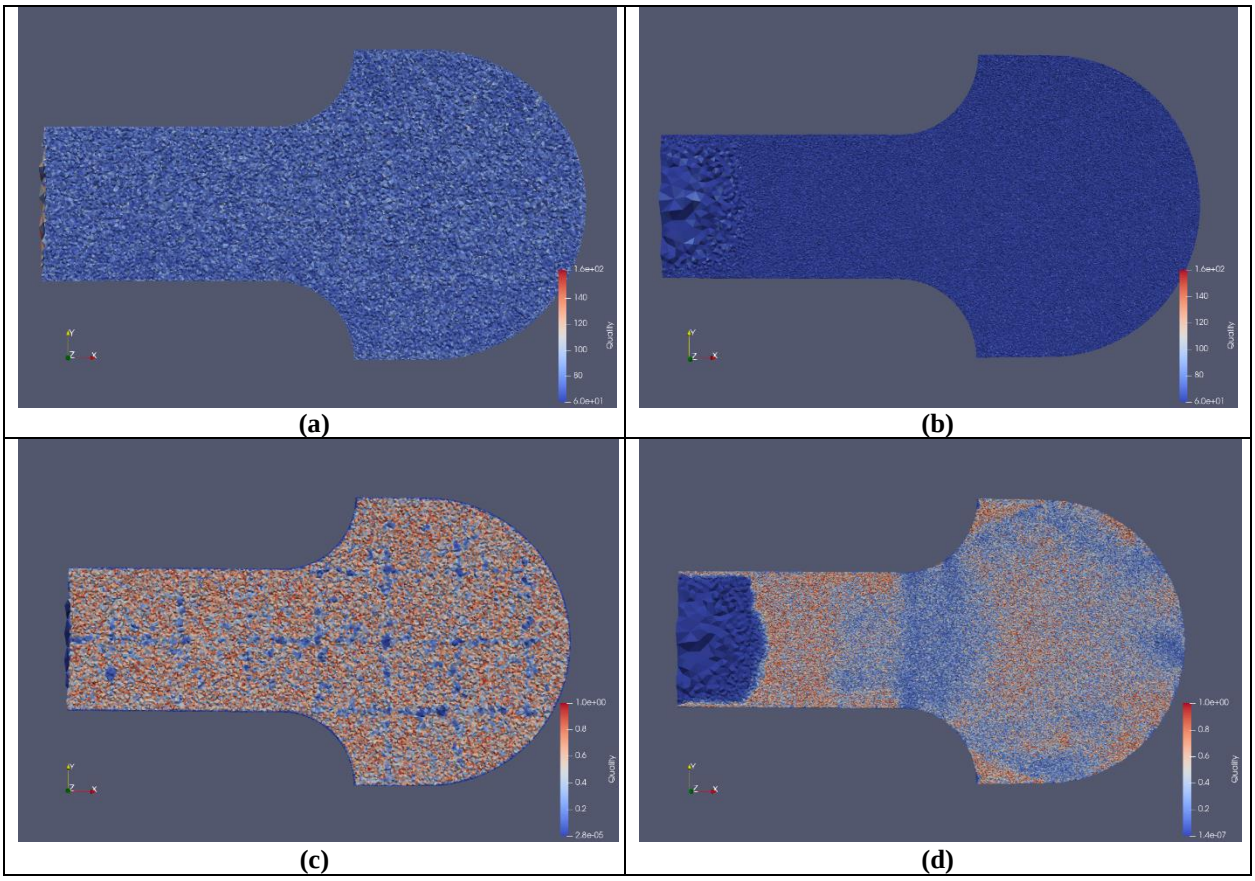


**Fig. 18. Dihedral Angle & Shape and Size Quality Metrics Visualized. Slices of the final meshes genereated by PsC.AFLR (a)(c) and serial AFLR (b)(d) are shown. (a) and (b) are colored by dihedral angle quality. (c) and (d) are colored by the shape and size quality metric defined in [19]. PsC.AFLR maintains good dihedral angle quality, as shown in (a). (c) shows the subdomain boundary elements, identified by their dark blue color, constraining the interior subdomain elements, thus preventing the subdomains from achieving the same density as the volume generated by serial AFLR in (d).**

Another observation of how the problem domain is affected by the coarse surfaces, which in turn influences the behavior of AFLR, can be gathered from Fig. 9 (time percentages of AFLR operations in the super-subdomain local reconnection implementation). The percent contribution of each AFLR operation is stable across all numbers of core runs. This is because the problem input size (the number of elements in each leaf and its level 1 neighbors) is approximately the same when refined by AFLR, especially since the geometry (horn bulb) does not have varying degrees of element density throughout the mesh. Its distribution function is defined mostly by the coarse leaf surfaces (even the leaves that contain the external surface elements are still greatly impacted by the coarse elements given that the coarse elements will surround a leaf on almost all sides). The external input size (1,062,042 surface elements) of the horn bulb geometry remains constant for each run and the average size of the generated meshes in each run are approximately equal (about 13 million volume elements). This shows that AFLR remains stable (in both the amount of runtime and the number of generated elements) when given problem sizes and distribution functions that are approximately equal.

# 6. CONCLUSIONS

Due to the requirements and constraints set by AFLR, it is not possible to simply use it in a black box manner when attempting to parallelize the software. One must consider the overhead introduced simply from preparing a subdomain of data for refinement. Partition reassignment, boundary extraction, and data assignment all play a significant role in both the success of refinement (generating correct results, i.e. a conforming mesh with good quality) and in the runtime. Although they produce overhead on the runtime, these operations are essential when parallelizing AFLR if one wishes to minimize the modifications needed for AFLR. PsC.AFLR has good end-user productivity given its refinement speed and is able to outperform serial AFLR by about 11 times on 16 cores. Still, a pressing issue is that meshes generated by PsC.AFLR are much less dense than those generated by serial AFLR. The two programs essentially produce two different final volume meshes, regardless of attempting to run AFLR within PsC.AFLR using the same settings as serial AFLR (including when using identical point distribution values). The aforementioned operations, although necessary, constrain the full capabilities of AFLR to generate dense meshes of high quality.

## 6.1 FUTURE WORK

There is still much to be accomplished in order to create a PDR.AFLR implementation. Several additions will need to be made in order to maintain the same level of functionality as the serial code. Based on past work and results [13], an unconstrained parallel data refinement implementation with AFLR is expected to be much faster and efficient than the pseudo-constrained PDR.AFLR. This will require a significant time investment, as many fundamental changes will need to be made to the source code. Successfully implementing these changes would eliminate several pseudo-constrained PDR processes, which would likely significantly increase end-user productivity and improve scalability, making PDR.AFLR the first fully functional unstructured mesh generation/refinement application that will be capable of maintaining good parallel efficiency at $10^6$ concurrency levels.

As mentioned previously, TetGen is currently being used to generate an initial mesh due to its low runtime and because a volume constraint can be set easily when refining a geometry. The caveat, as described in Chapter 5, is that the final mesh generated by PsC.AFLR will not be as dense as that generated by serial AFLR due to the coarse elements of the TetGen-generated mesh and while the quality of the final mesh is satisfactory, it is not as good as that generated by the serial code. More research and collaboration with Dr. David Marcum (original developer of AFLR) is required to establish a more reliable method of generating the initial mesh. This problem may be eliminated completely if the aforementioned modifications are made to AFLR for an unconstrained PDR.AFLR implementation (eliminating the need for a dense initial mesh if boundaries do not need to be extracted for each leaf).

Another methodology must be developed to further refine subdomain boundary elements (actual point insertion) after they are frozen during leaf refinement (since these elements are extracted from the initial mesh and play a significant role in the density of a subdomain). There are similar methods in other mesh generation programs, such as *EPIC* from Boeing (discussed in Chapter 2) and *refine* from NASA, which freeze boundary elements during subdomain refinement to maintain mesh conformity [5]. These elements are later shifted between subdomains for refinement. The methods used to shift and refine these elements will be examined so that a similar approach may be applied to PDR.AFLR.

The runtime of the partition swapping process can be further reduced by removing the process entirely. The method of assigning data based on element barycenter will be modified to take advantage of a mathematically proven optimization-based geometry partitioning algorithm, which will eliminate the creation of disconnected partitions [20].

PsC.AFLR currently accepts only manifold genus zero computational geometries. Robustness will be addressed by identifying leaves that contain disconnected volumes of a mesh (caused by hole(s) in the geometry) and these individual pieces will be refined independently of each other. A new methodology will also be developed to allow PDR.AFLR to process geometries with transparent/embedded surfaces. An embedded surface must remain frozen during refinement, and because there may a volume on both sides of the surface, both sides can be considered as two separate subdomains. This is similar to the disconnected volume problem of a mesh with a genus greater than zero. Leaves with embedded surfaces will need to be partitioned further into additional subdomains, and these subdomains will be refined independently. Many geometries, such as meshes with turbulent flow around a rocket or missile, typically contain transparent/embedded surfaces. Once this functionality has been implemented, PDR.AFLR will be capable of refining these types of meshes. Code Re-use will be addressed by further expanding the design of PDR and developing a universal API, one that is capable of handling different data types/structures of different mesh generators. Scalability will also be improved by parallelizing the final optimization process.

# REFERENCES


[1] J. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie and D. Mavriplis, "CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences," NASA, Hampton, VA, USA, Rep. NF1676L-18332, CR Rep. NASA/CR-2014-218178, 2014. Accessed: Nov. 17, 2020. [Online]. Available: https://ntrs.nasa.gov/citations/20140003093

[2] T. Michal and J. Krakos, "Anisotropic Mesh Adaptation Through Edge Primitive Operations," in *50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, Nashville, TN, USA, 2012, Paper 159.

[3] A. Loseille and R. Löhner, "Anisotropic Adaptive Simulations in Aerodynamics," in *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Orlando, FL, USA, 2010, Paper 169.

[4] A. Loseille, "Unstructured Mesh Generation and Adaptation," *Handbook of Numerical Analysis*, vol. 18, pp. 263-302, 2017.

[5] C. Tsolakis, N. Chrisochoides, M. Park, A. Loseille and T. Michal, "Parallel Anisotropic Unstructured Grid Adaptation," in *AIAA SciTech Forum 2019*, San Diego, CA, USA, 2019, Paper 1995.

[6] A. Loseille, V. Menier and F. Alauzet, "Parallel Generation of Large-size Adapted Meshes," *Procedia Engineering*, vol. 124, pp. 57-69, 2015.

[7] D. Marcum and N. Weatherill, "Unstructured Grid Generation Using Iterative Point Insertion and Local Reconnection," *AIAA Journal*, vol. 33, no. 9, pp. 1619-1625, September, 1995.

[8] N. Chrisochoides, "Telescopic Approach for Extreme-scale Parallel Mesh Generation for CFD Applications," in *AIAA Aviation*, Washington, D.C., USA, 2016, Paper 3181.

[9] N. Chrisochoides, A. Chernikov, A. Fedorov, A. Kot, L. Linardakis and P. Foteinos, "Towards Exascale Parallel Delaunay Mesh Generation," in *18th International Meshing Roundtable*, Salt Lake City, UT, USA, 2009.

[10] A. Chernikov and N. Chrisochoides, "Parallel Guaranteed-quality Delaunay Uniform Mesh Refinement," *SIAM Journal on Scientific Computing*, vol. 28, no. 5, pp. 1907-1926, 2006.

[11] A. Chernikov and N. Chrisochoides, "Practical and Efficient Point-insertion Scheduling Method for Parallel Guaranteed-quality Delaunay Refinement," in *18th ACM International Conference on Supercomputing*, Malo, France, 2004.

[12] A. Chernikov and N. Chrisochoides, "Three-dimensional Delaunay Refinement for Multi-Core Processors," in *22nd ACM International Conference on Supercomputing*, Island of Kos, Greece, 2008.

[13] N. Chrisochoides, A. Chernikov, T. Kennedy, C. Tsolakis and K. Garner, "Parallel Data Refinement Layer of a Telescopic Approach for Extreme-scale Parallel Mesh Generation for CFD Applications," in *AIAA Aviation 2018*, Atlanta, Georgia, USA, 2018, Paper 2887.

[14] H. Si, "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator," *ACM Transactions on Mathematical Software*, vol. 41, no. 2, pp. 11:1-11:36, 2015, doi: 10.1145/2629697.

[15] FUN3D Manual: 13.6, by R. Biedron, J.-R. Carlson, J. Derlaga, P. Gnoffo, D. Hammond, W. Jones, B. Kleb, E. Lee-Rausch, E. Nielsen, M. Park, C. Rumsey, J. Thomas, K. Thompson and W. Wood. (2019, October). Patent NASA/TM-2019-220416. [Online]. Available: https://ntrs.nasa.gov/citations/20190033239.

[16] T. Economon, F. Palacios, S. Copeland, T. Lukaczyk and J. Alonso, "SU2: An Open-Source Suite for Multiphysics Simulation and Design," *AIAA Journal*, vol. 54, no. 3, pp. 828-846, 2015, doi: 10.2514/1.J053813.

[17] P. Thomadakis, C. Tsolakis, K. Vogiatzis, A. Kot and N. Chrisochoides, "Parallel Software Framework for Large-Scale Parallel Mesh Generation and Adaptation for CFD Solvers," in *AIAA Aviation Forum 2018*, Atlanta, 2018, Paper 2888.

[18] Old Dominion University, "High Performance Computing," 15 September 2018. [Online]. Available: https://www.odu.edu/facultystaff/research/resources/computing/high-performance-computing. [Accessed 13 May 2019].

[19] C. Stimpson, C. Ernst, P. Knupp, P. Pebay and D. Thompson, "The Verdict Library Reference Manual," Sandia National Laboratories, USA, Rep. 9, 2007.

[20] N. Chrisochoides, C. Houstis, E. Houstis, S. Kortesis and J. Rice, "Automatic Load Balanced Partitioning Strategies," Purdue University, West Lafayette, IN, USA, Rep. 89-862, 1989. [Online]. Available: https://docs.lib.purdue.edu/cstech/733

# VITA

Kevin Mark Garner Jr.

Computer Science Department
Old Dominion University
3321 Engineering & Computational Sciences Building
Norfolk, VA 23529
Email: kgarn006@odu.edu
Linkedin: https://www.linkedin.com/in/kevin-garner-aba04b138

## EDUCATION

1. M.Sc., Department of Computer Science, Old Dominion University, December 2020
2. B.Sc., Department of Computer Science, Old Dominion University, May 2016

## HONORS AND AWARDS

1. Southern Regional Education Board (SREB) State Doctoral Scholars Fellowship, 2018-present
2. Virginia Space Grant Consortium Graduate Fellowship, 2016-2018
3. Computer Science Outstanding Graduate Researcher Award, 2018
4. ODU Presidential Scholarship, 2012-2016
5. Member of the National Society of Collegiate Scholars, 2013 – present

## WORK EXPERIENCE

1. Research Assistant at Old Dominion University, Norfolk, VA during Fall 2016 – present
2. Teaching Assistant at Old Dominion University for CS 381 Discrete Structures and CS 126G in the Computer Science Department, Norfolk, VA during Fall 2016

## PUBLICATIONS

**First Author Papers in Proceedings of Refereed Conferences**

1. K. Garner, P. Thomadakis, C. Tsolakis, T. Kennedy, and N. Chrisochoides, “On the End-user Productivity of a Pseudo-constrained Parallel Data Refinement method for the Advancing Front Local Reconnection Mesh Generation Software,” in *AIAA Aviation Forum 2019*, Dallas, TX, USA, 2019, Paper 2844.
2. K. Garner, T. Kennedy, C. Tsolakis and N. Chrisochoides, “Stability of Advancing Front Local Reconnection for Parallel Data Refinement,” in *Virginia Space Grant Consortium 2018 Student Research Conference*, Norfolk, VA, USA, 2018.
3. K. Garner, T. Kennedy and N. Chrisochoides, "Integration of Parallel Data Refinement Method with Advancing Front Local Reconnection Mesh Refinement Software," in *Virginia Space Grant Consortium 2017 Student Research Conference*, Williamsburg, VA, USA, 2017.

**Poster Publications**

4. K. Garner, D. Feng, F. Drakopoulos, Y. Liu and N. Chrisochoides, "Image-to-Mesh Conversion Tool," in *24th International Meshing Roundtable*, Austin, TX, USA, 2015.

## INVITED SCHOLARLY TALKS

1. Unstructured Adaptive Mesh Generation, Virginia Space Grant Consortium Board of Directors Meeting, Old Dominion University, Norfolk, VA, USA, April 2018.