Summer 2015

Parallel Two-Dimensional Unstructured Anisotropic Delaunay Mesh Generation for Aerospace Applications

Juliette Kelly Pardue
Old Dominion University, spardue@odu.edu

PARALLEL TWO-DIMENSIONAL UNSTRUCTURED ANISOTROPIC DELAUNAY MESH GENERATION FOR AEROSPACE APPLICATIONS

by

Juliette Kelly Pardue
B.S. May 2014, 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
August 2015

Approved by:

_____
Andrey Chernikov (Director)

_____
Nikos Chrisochoides
(Member)

_____
Boris Diskin (Member)

_____
Michele Weigle (Member)

ABSTRACT

PARALLEL TWO-DIMENSIONAL UNSTRUCTURED ANISOTROPIC DELAUNAY MESH GENERATION FOR AEROSPACE APPLICATIONS

Juliette Kelly Pardue
Old Dominion University, 2015
Director: Dr. Andrey Chernikov

A bottom-up approach to parallel anisotropic mesh generation is presented by building a mesh generator from the principles of point-insertion, triangulation, and Delaunay refinement. Applications focusing on high-lift design or dynamic stall, or numerical methods and modeling test cases focus on two-dimensional domains. This push-button parallel mesh generation approach can generate high-fidelity unstructured meshes with anisotropic boundary layers for use in the computational fluid dynamics field.

This thesis is dedicated to those trying to find a reason to get up in the morning

ACKNOWLEDGMENTS

I thank my advisor, Dr. Andrey Chernikov, for his guidance, support, and unlimited patience throughout my studies, as well as the insights provided by Dr. Nikos Chrisochoides.

I am grateful to Dr. Boris Diskin for the opportunity to work under the National Institute of Aerospace and for being able to complete my research at NASA Langley.

I also thank Michael Park for his invaluable comments and discussions.

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Mesh generation is the scientific art of representing or discretizing a domain with polygonal elements. Mesh generation is used with the finite element method (FEM), an approach in the field of numerical analysis used to estimate solutions to boundary value problems. A boundary value problem is given as a set of partial differential equations (PDE) and the boundary conditions of the domain defined by the PDE. Software tools, known as solvers, are developed to solve a set of PDE by providing an approximate solution based on specified boundary conditions. The input to these solvers is a mesh. The quality of the mesh is critical to the performance of the solver, the accuracy of the approximated solution, and the existence of a solution. Small angles, large angles, poor gradation, and low resolution all have an effect on the solver's performance and solution [29]. Small angles affect the stiffness matrix of the mesh which may lead to a divergent solution while large angles affect the interpolation accuracy and discretization errors. A poor gradation of element sizes causes unstable extrapolation when computing high-order schemes. If the domain is not discretized to provide enough resolution at critical areas, then the solution will lose accuracy.

Mesh generation is also a step in the iterative pipeline for designing aerospace structures as shown in Fig. 1. After an analysis of the partial differential equations (PDE) solution on the mesh, the mesh is refined to yield a more favorable error estimation, which is typically a faster operation than generating an entirely new mesh. This is because the PDE solution identifies, for a mesh that possesses at least some degree of accuracy with

respect to the PDE, a subset of mesh regions for refinement. The refinement process aims to gradually and incrementally add more resolution to the identified areas, as opposed to over-refining the mesh, which causes the PDE solver to waste computation time due to the excessive computations. After a PDE solution for a mesh is computed and analyzed, then the mesh may be refined to provide a more accurate solution. Assuming proper care for the refinement steps, this new mesh will be more accurate in the desired regions with respect to the PDE solution. This means that with each refinement step, the next iteration of the mesh will have a smaller error estimate than the previous mesh. So at each iteration, the overall refinement work required to reach a highly accurate solution decreases since the mesh's accuracy always improves.

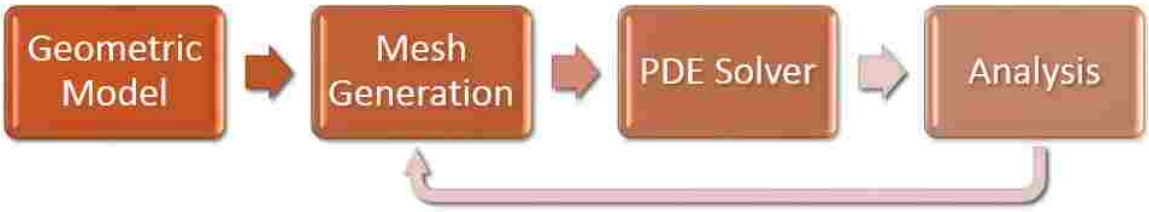


Fig. 1. Development pipeline

When executing any pipeline of tasks, it is critical to consider Amdahl's law [30] which states that the speedup of a program is limited by the sequential fraction of the program, see Fig 2. Consider a program where only 25% of the program needs to be performed sequentially. Applying Amdahl's law, even if we were able to parallelize the remaining 75% of the program in such a way that the execution of this parallel portion terminates instantaneously, the largest speedup we would be able to achieve is four. This is why it is crucial that all tasks in the pipeline be parallelized, because the effect of

Amdahl's law multiplies when a repetitive pipeline is followed, such as the one depicted in Fig. 1.

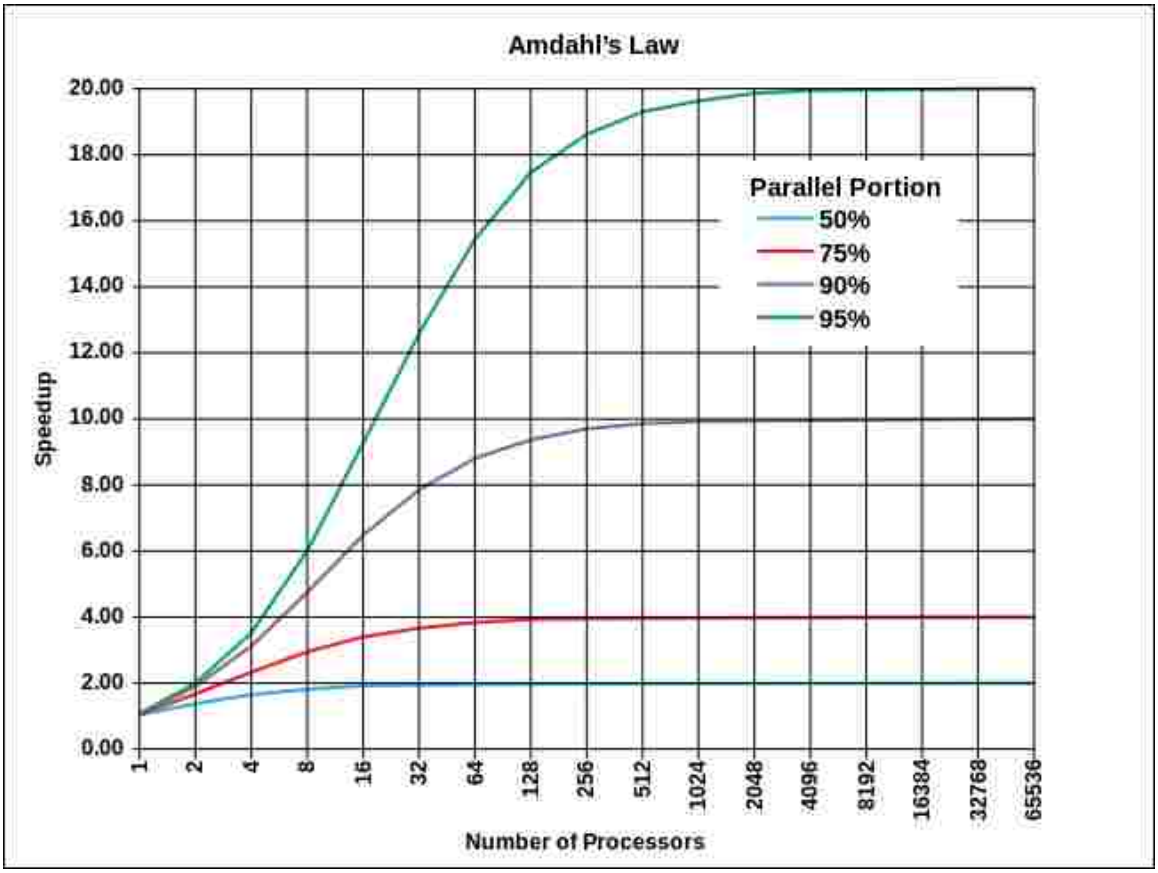


Fig. 2. Max speedup according to Amdahl's Law

Since the end goal is to generate a mesh which accurately and efficiently fits the PDE, the time to achieve this goal is dependent on the number of iterations through the pipeline of mesh generation to PDE solver to analysis. Clearly, this iterative process needs an initial mesh to begin the process. If the initial mesh closely represents the PDE, then fewer iterations through the pipeline are required to achieve a suitable solution. However, if the initial mesh is highly inaccurate with respect to the PDE, then the first iterations

through the pipeline will present numerous areas which require refinement. Eventually, after so many iterations through the pipeline that began with an unsuitable initial mesh, the current mesh will have similar error estimates as an initial mesh which was well-suited and closely represented the PDE. So the initial mesh sets the pace for the remainder of the iterations through the pipeline as well as the amount of refinement work. Clearly we need an initial mesh that has a high degree of accuracy with respect to the PDE while simultaneously being an efficient discretization of the domain in order to provide the most CPU savings to the PDE solver and to also generate the final mesh with the fewest number of iterations through the pipeline, thus yielding the fastest overall execution time.

Chapter 2

BACKGROUND

There are currently a myriad of mesh generation techniques and mesh types, which have variable uses and meet particular demands. Isotropic mesh generation and anisotropic mesh generation along with the advancing front and Delaunay refinement methods are covered in this section. Also, some current parallel mesh generation techniques are reviewed.

2.1 ISOTROPIC MESH GENERATION

Properties of Delaunay triangulations hold for isotropic meshes and are mathematically provable. Properties of Delaunay triangulations include the empty-circumcircle property, which states that for each triangle, the circumscribed circle does not contain any other vertices of the triangulation. Delaunay triangulations also maximize the minimal angle of the overall mesh, which improves the condition number of the stiffness matrix which has an effect on the rate of convergence and in some cases, the existence of a solution when solving the PDE. The proof of termination and uniqueness is also guaranteed by the Delaunay property. With the robust mathematics involving element quality and proof of termination, isotropic Delaunay mesh generators have become common place. The best sequential isotropic Delaunay triangulator and mesh generator is Triangle [3], which is robust and has the fastest evaluated execution time while also providing mechanisms for prescribing element quality and size as well.

2.2 ANISOTROPIC MESH GENERATION

With the fast developing field of computational fluid dynamics (CFD), a new mesh type has been introduced, graded anisotropic meshes. Graded anisotropic meshes aim to decrease the computational efforts of the PDE solvers as well as to decrease the number of elements in the mesh. Isotropic mesh generators which focus on solution-based adaptation create many unnecessary elements where there is a high degree of gradation in the flow velocities in a given direction. These anisotropic gradations in the flow velocities require anisotropic elements, typically with a 10,000:1 aspect ratio, so representing these regions with isotropic elements incurs a 10,000 fold increase in the number of elements. Using isotropic mesh generators to model anisotropic PDE has a negative effect for two reasons: the mesh generation time is increased significantly because more elements need to be created and refined, and the time for the flow solver is increased due to the increase in the number of elements in the mesh. Isotropic mesh generators are faced with the choice to prescribe either a high-density region to capture the anisotropic gradations in flow velocities while introducing wasted computations, or to settle for a low-resolution region to save computations while sacrificing the ability to capture the anisotropic gradients. Fig. 3 shows an isotropic discretization of a sample anisotropic domain, while Fig. 4 shows the same sample domain discretized appropriately with anisotropic elements.

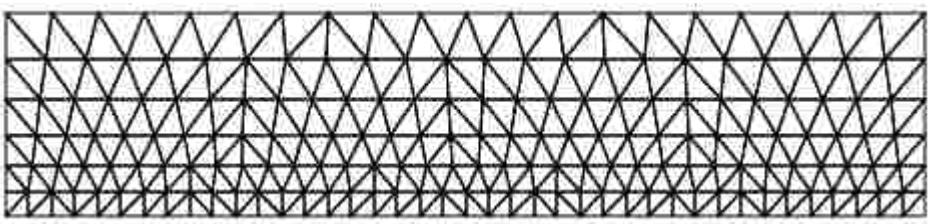


Fig. 3. Isotropic representation of an anisotropic domain.

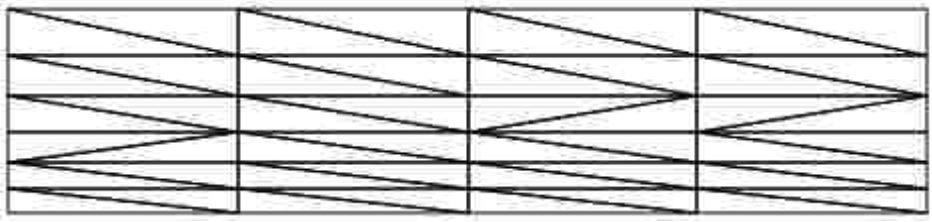


Fig. 4. Anisotropic representation of an anisotropic domain.

Sequential anisotropic mesh generators [8, 12] have been developed to help take advantage of these anisotropic characteristics. Sequential two-dimensional tools for aerospace application development exist, such as XFOIL [17] and MSES [18], which also cater towards airfoil development through geometry discretization and mesh generation. Other general-purpose anisotropic mesh generation approaches by Li et al. and Bossen and Heckbert [27, 28] do not offer any guarantee of termination or uniqueness due to the inability to mathematically prove the algorithm, making them unreliable choices.

2.3 ADVANCING FRONT METHOD

The advancing front method presented by Marcum and Schoberl [22, 23] works by first discretizing the geometry into segments which become the initial fronts, see Fig. 5a, and then choosing a front to advance with by creating a triangle using the selected front as the base of the new triangle. A new point may need to be created if there is no suitable point currently available, see Fig. 5b. The front that is the base of the triangle is then removed from the set of fronts because it has become obscured by the new triangle. The other two edges of the triangle are added to the set of fronts on the condition that they are not obscured by the new triangle. The new point is chosen as the point that will produce the optimal triangle for the desired mesh. The optimal triangle is dependent on the PDE that is being solved.

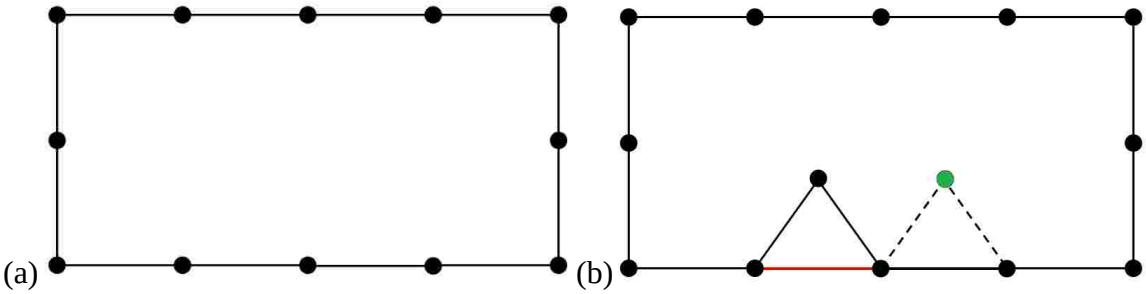


Fig. 5. (a) Initial discretization of the boundary of the geometry. All edges are part of the front; (b) First triangle created and front (red) removed. Point for second triangle (green) being considered.

The advancing front method terminates once there are no remaining fronts, meaning the domain has been fully discretized. However, the advancing front method has difficulties terminating when two fronts of drastically different edge lengths attempt to merge. An inefficiency occurs with this approach when attempting to create a new triangle since the new triangle must not intersect with any existing triangles or fronts, so intersection checks need to be performed for the local neighborhood of the prospective triangle.

2.4 DELAUNAY REFINEMENT METHOD

Delaunay refinement algorithms [24, 25] begin with an initial Delaunay triangulation, Fig. 6b, of the vertices of the input geometry, Fig. 6a, and then aims to improve the overall quality of the mesh by inserting new points, known as Steiner points. The invariant of these algorithms is that the mesh is always a Delaunay mesh and each new point may require that current triangles be removed and replaced with new triangles or a constraining edge may be split, causing the triangles that share the edge to be removed and replaced with new triangles. The criteria for refining the mesh is typically based on element

size and angles. The algorithm terminates once there are no more ill-suited elements according to the refinement policy, see Fig. 6c and Fig. 6d.

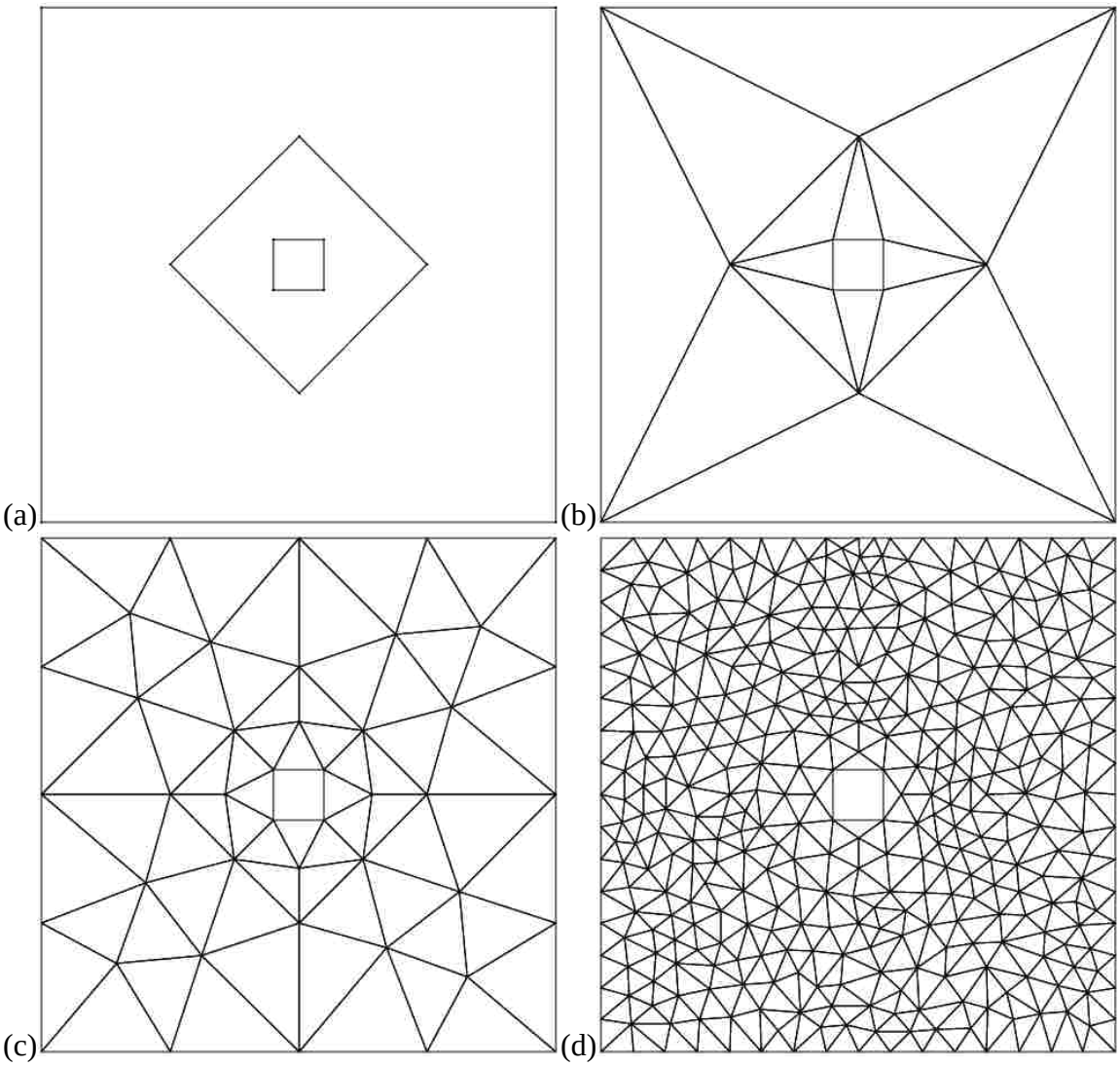


Fig. 6. (a) Input geometry; (b) Initial triangulation; (c) Initial triangulation refined using Delaunay refinement with angle constraint of 33 degrees; (d) Initial triangulation refined using Delaunay refinement with angle constraint of 33 degrees and area constraint of 1 unit.

2.5 PARALLEL MESH GENERATION

Current parallel mesh generators by Globisch, Kadow, Lammera and Burghardt, and Khan and Topping exist which handle the isotropic cases [14, 16, 19, 20] do not perform well for parallel anisotropic mesh generation. Extensive efforts have been applied by Ito et al. and Chrisochoides and Nave [13, 15] to generating meshes in parallel for the uniform isotropic and graded isotropic case. In parallel, Zagaris et al. [10, 11] and sequentially Loseille et al. and Zhang et al. [8, 12], researchers have begun developing anisotropic mesh generation paradigms to facilitate these CFD simulations.

Since efficient unstructured meshes for aerospace applications are comprised of two different mesh types, a pseudo-structured anisotropic boundary layer and an unstructured isotropic inviscid region, two separate paradigms are needed to generate high-fidelity initial meshes that are computationally efficient for PDE solvers. The pseudo-structured anisotropic boundary layer is generated through an extrusion-based advancing-front method, as presented by Aubry et al. [9], while the unstructured isotropic inviscid region is generated using a graded decoupled approach presented by Linardakis and Chrisochoides [5] along with Delaunay refinement presented by Shewchuk [26].

CHAPTER 3

PARALLEL ANISOTROPIC BOUNDARY LAYER POINT INSERTION

Physical phenomena such as boundary layers in fluid mechanics are anisotropic in nature. There is a high degree of gradation in the flow velocities normal to the surface, thus it is beneficial to discretize the mesh in a way that efficiently captures these anisotropic flow velocities in order to yield substantial CPU savings without compromising accuracy. This dictates that the mesh should be refined in the direction normal to the surface, as shown in Fig. 7, where these strong gradients exist. These characteristics allow for the extrusion-based point insertion along the normal of the surface at each vertex on the planar straight-line graph (PSLG). Essentially, each vertex is treated as an endpoint for a ray while the normal at the vertex is treated as the direction of the ray. New points are then inserted along the ray, as in Fig. 8, according to a growth function. There are multiple functions, as presented by Garimella and Shephard [1], which can be used to space the prospective points. Certain growth functions may yield a more accurate discretization of the domain depending on the PDE that is being solved. Two common growth functions are polynomial and geometric, which offer a uniform growth along the normal of the PSLG. However, other more sophisticated, adaptive growth functions [1], may be necessary for more complex geometries.

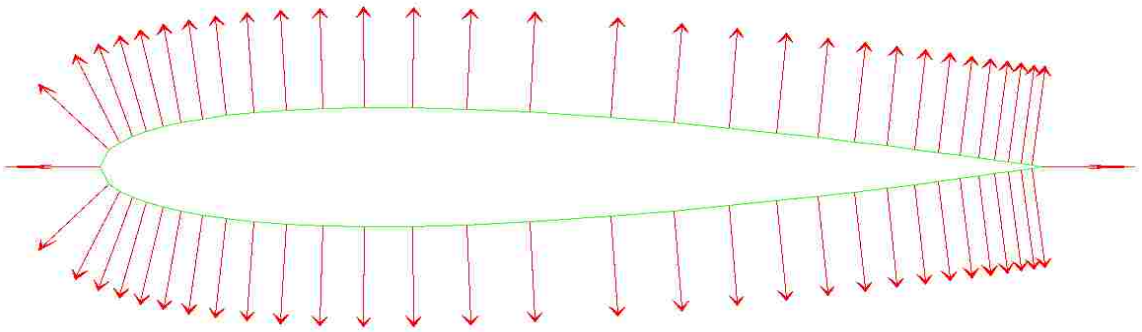


Fig. 7. NACA 0012 Airfoil with surface normals

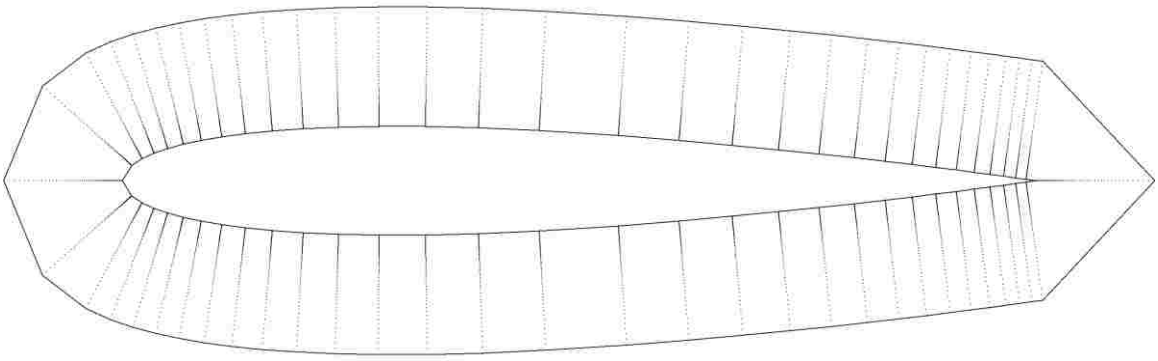


Fig. 8. NACA 0012 Airfoil with vertices inserted along surface normals

Clearly, larger angles will naturally occur where the slope changes rapidly, such as the leading edge shown in Fig. 8, and extremely large angles at cusps, such as the trailing edge in Fig. 8. The regions where these large angles occur are the areas of the mesh that need refinement to satisfy the resolution constraints of the mesh's boundary layer region. For the flow solver, since the boundary conditions are calculated first and are propagated through and affect the entire solution over the mesh, it is critical that the boundary layer be properly discretized. This means avoiding the case of intersecting rays. Additionally, if the

angle between two rays is too large, then the distance between vertices of neighboring rays will grow at excessively rapid rates, affecting the density of the mesh in the corresponding area, causing interpolation errors when the PDE solution is computed.

3.1 RAY INSERTION

To treat cases where there is a large angle between two rays, a new point is created between the two points that have a large angle between their normals. Due to the nature of the PSLG being an explicit representation of the geometry, we lose the ability to determine the exact location of the new point, and therefore the normal at this point. Due to a lack of implicit information about the geometry, the midpoint of the two points and the average of the two normals at the two points is used to create a new ray. This is for the case where only one refining ray is needed to satisfy the angle constraint, a user-input constant that sets an upper bound on the angle between neighboring rays. The approach is analogous for cases where more refining rays are needed where the new points for the new rays are uniformly spaced along the straight line connecting the original two points. This process is done in parallel where each thread accesses a shared queue of vertices and computes the normal at the vertex which becomes the ray. After the rays have been determined for each of the vertices of the PSLG, the angle between the current ray and the forward neighboring vertex's ray is computed. If the angle is too large, then the aforementioned approach of creating refining rays is implored. Fig. 9a shows two large angles easily visible at the trailing edge of the NACA 0012 Airfoil after the ray-based point insertion, while Fig. 9b includes the augmented points determined by the large angle detection. An angle constraint of seven degrees was used for Fig. 9b. A smaller angle constraint can be applied to yield a trailing edge region with high resolution.

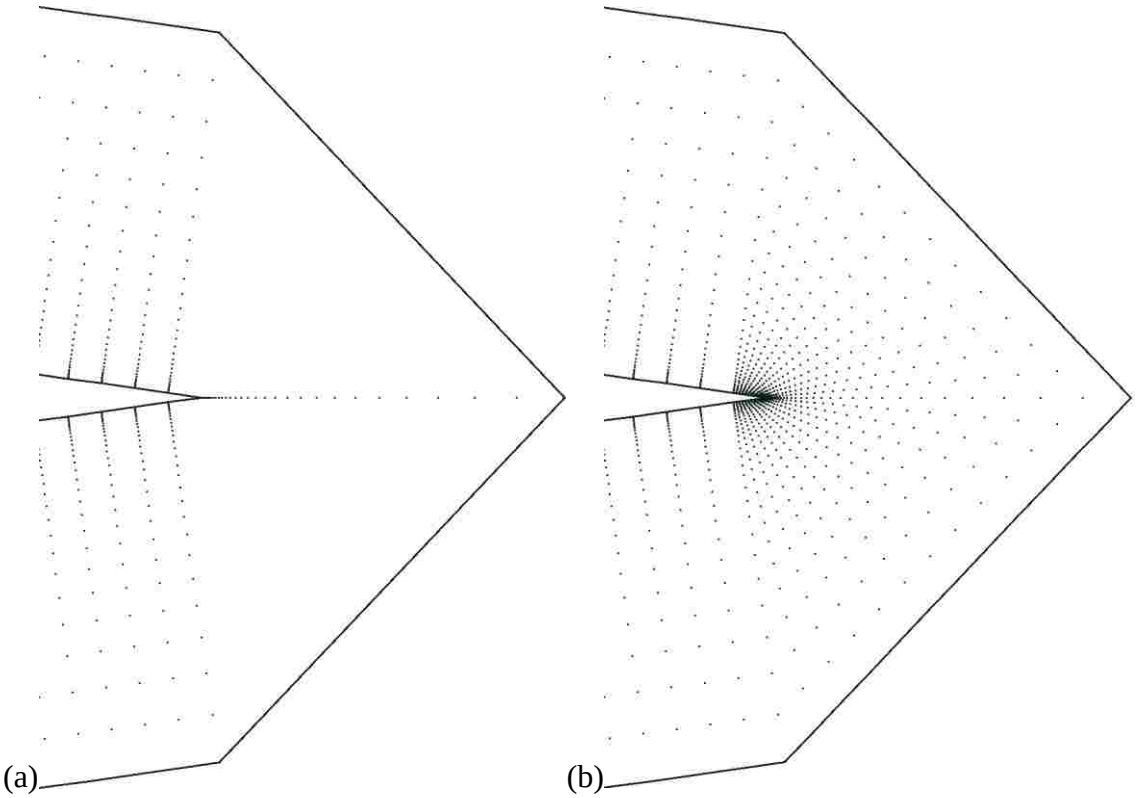


Fig. 9. (a) Trailing edge point insertion for original surface normals; (b) Trailing edge point insertion with refining rays added

3.2 RAY INTERSECTION

Once the rays have been determined, a quality check is performed to determine if any of the rays intersect. Since the rays extrude outward from the surface in the direction of the normal, convex geometries will not have intersecting rays, because all interior angles of a convex polygon are not greater than 180 degrees. This allows for the search space to be greatly reduced by only checking concavities, which can be determined using the two-dimensional orientation test to determine if a point lies to the left, right, or on a directed line. Walking along the PSLG in a counter-clockwise order, the orientation test is performed to see if point p_i lies to the left of the directed line (p_{i-2}, p_{i-1}). If this holds true, then we have detected the beginning of a concavity. To determine the end of the concave

region, the edge (p_{i-2}, p_{i-1}) is kept constant while the walking continues along the geometry until the first point that is on or to the right of the directed line (p_{i-2}, p_{i-1}) is found. Since we assume a simple polygon as the input geometry, and we keep track of the edges that each vertex is incident upon, we can perform the counter-clockwise walk in linear time with respect to the number of vertices since each vertex is incident upon exactly two edges.

3.3 ANISOTROPIC VERTEX CREATION

After the intersection check terminates, the main thread allocates space for all of the needed Vertex objects. After the data is allocated, each thread accesses a shared queue of Ray objects and calculates the new points along the ray direction with respect to the growth function and uses these points to set the coordinates of the associated vertices. New vertices are inserted until the number of vertices for the current ray equals the number of layers requested, or in the case of an intersection with another ray, until the intersection point. The process is performed in parallel since no communication between threads is needed, because the data has already been allocated by the main thread, so each worker thread only needs to compute the location in memory for its vertices that it will be initializing, but does not need to modify the global container storing the Vertex objects. The final point inserted along each ray direction is then used to create constraining edges to enclose the boundary layer.

CHAPTER 4

PARALLEL TRIANGULATION OF BOUNDARY LAYER

After the point insertion step is complete, the vertices in the boundary layer need to be triangulated. The algorithm presented by Blelloch et al. [2] is used, which utilizes the duality between the two-dimensional Delaunay Triangulation and the three-dimensional lower convex hull of a paraboloid, see Fig. 10.

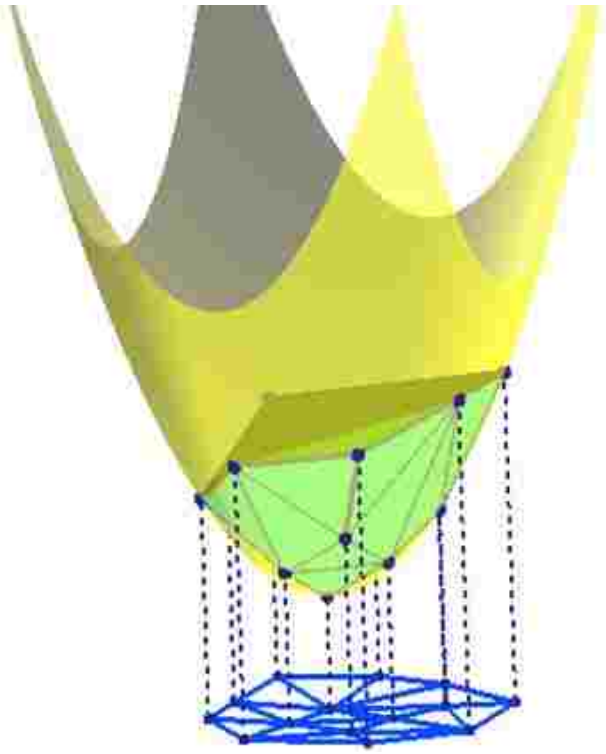


Fig. 10. Paraboloid (yellow), lower convex hull (green) with corresponding Delaunay triangulation (blue), points on the Cartesian plane, and projected points on the paraboloid for a sample point set.

4.1 2D DELAUNAY TRIANGULATION FROM 3D CONVEX HULL

The connection is between the two-dimensional Delaunay Triangulation's two-dimensional in-circle test and the three-dimensional lower convex hull of the paraboloid's three-dimensional orientation test. The two-dimensional in-circle test, see Fig. 11a, answers the question, "does point t lie inside, on, or outside the circle defined by points a, b, and c?" by evaluating the determinant of the matrix in Fig. 11b. If the determinant is negative, then point t lies within the circle. If the determinant is zero, then point t lies on the circle. If the determinant is positive, then point t lies outside of the circle. Similarly, the three-dimensional orientation test, see Fig. 12a, answers the question, "does point t lie below, on, or above the plane defined by points a, b, and c?" by evaluating the determinant of the matrix in Fig. 12b. If the determinant is negative, then point t lies below the plane. If the determinant is zero, then point t lies on the plane. If the determinant is positive, then point t lies above the plane. When the point set is projected onto the paraboloid $z = x^2 + y^2$, the matrix for computing the three-dimensional orientation test is identical to the matrix used to compute the two-dimensional in-circle test because the equation of the paraboloid is the third term for each row in both the two-dimensional in-circle test and the three-dimensional orientation test. This means that each facet of the lower convex hull of the paraboloid corresponds to a triangle in the Delaunay Triangulation.

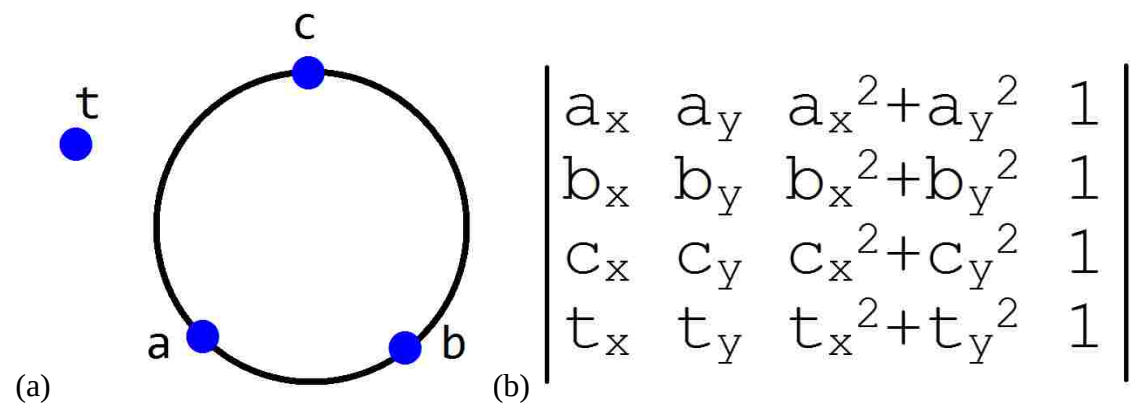


Fig. 11. (a) Two-dimensional in-circle test; (b) Matrix used to evaluate the two-dimensional in-circle test

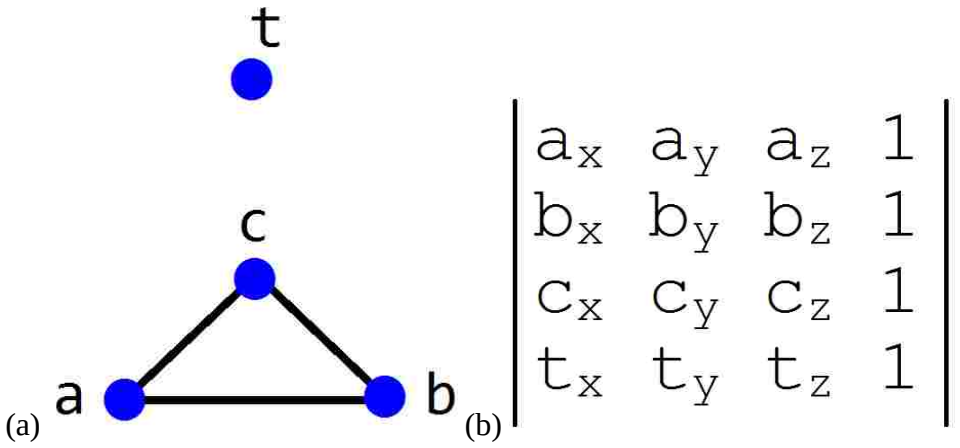


Fig. 12. (a) Three-dimensional orientation test; (b) Matrix used to evaluate the three-dimensional orientation test

Lemma: *Consider* Δpqr *and the plane defined by the projection of these points p', q', and r'. The circumcircle, C, of* Δpqr *is empty iff the plane defined by p', q', and r' is a face of the lower convex hull of the projected point set onto the paraboloid.*

Proof: *If the plane is not a face of the lower convex hull, then there must be a point in the domain, s', which lies below the plane, and s which is inside C. However, C is empty, so*

the plane defined by p', q', and r' is a face of the lower convex hull. Conversely, if C is not empty, then there must be a point in the domain, s, which is inside C, and s' which lies below the plane defined by p', q', and r'. However, this plane is a face of the lower convex hull, so C is empty.

4.2 ALGORITHM OVERVIEW

The algorithm works by dividing a set of vertices into two subdomains with a median line and dividing path of Delaunay edges. The path of Delaunay edges divides Delaunay triangles based on if their circumcenter is to one side of the median line or to the opposite side of the median line. This approach was chosen because the dividing path created between subdomains corresponds to constraining edges which would be present in the final triangulation if the domain were triangulated sequentially without being decomposed, unlike other algorithms [7] which use user-defined dividing paths to arbitrarily partition the domain. However, this is undesirable as these user-defined dividing paths are artificial and would not have been present in the original triangulation. These artificial dividing paths have a negative effect on the discretization of the boundary layer since these dividing paths disturb the spacing pattern and the alignment and orthogonality. The median line, for efficiency and simplicity of the algorithm, is parallel to the x-axis or y-axis, known as the cut axis. These Delaunay edges are edges of Delaunay triangles in the final triangulation, which allows for each subdomain to be triangulated independently by a state-of-the-art Delaunay triangulator, Triangle [3]. This approach is used as a coarse-partitioner which aims to decompose the domain into coarse regions which can be meshed independently. Each subdomain only needs to be recursively divided until there are enough

subdomains to yield an acceptable degree of load balancing for the concurrent triangulation of the subdomains.

4.3 COMPUTING THE DIVIDING DELAUNAY PATH

Our algorithm starts by creating an initial Subdomain object which stores vertices in x-sorted order and a copy in y-sorted order. The sorting is performed using a parallel version of quicksort, where half of the threads sort by the x-coordinate vertices and the other half of the threads sort by the y-coordinate vertices. The parallel version of quicksort uses a global pivot to partition the vertices into two groups: vertices less than the global pivot and vertices greater than or equal to the global pivot. This approach is applied recursively until there are enough partitioned ranges for each thread to independently sort. This allows for the bounding box to be computed in constant time using the first and last vertex of the x-sorted and y-sorted vertices. The cut axis is set to be the axis parallel to the shortest edge of the bounding box, thus to avoid the creation of long, skinny subdomains which are more expensive to triangulate with Triangle due to the merge step of Triangle's divide-and-conquer approach. Maintaining the sorted vertices also allows for the median vertex along the cut axis to be located in constant time. Using this median vertex, the cut-axis-sorted vertices are projected onto a paraboloid centered at the median vertex and then flattened onto the vertical plane perpendicular to the cut axis.

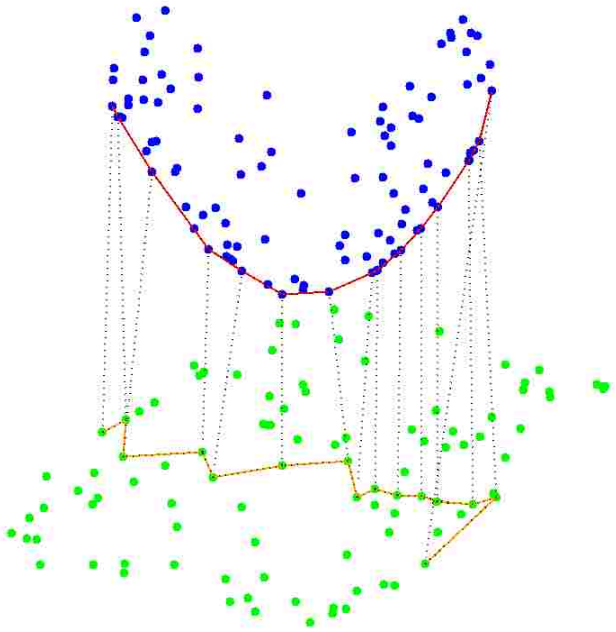


Fig. 13. Cartesian point set (green) and Delaunay path (brown) with corresponding flattened projection of the point set (blue) and lower convex hull (red).

Kadow [16] provides a more in-depth proof regarding the mathematics concerning the relationship between the two-dimensional Delaunay triangulation, three-dimensional lower convex hull of a paraboloid, and two-dimensional lower convex hull of a paraboloid flattened onto a vertical plane. The vertices that have been flattened onto the vertical plane are then used to compute the lower convex hull, see Fig. 13, in worst case linear time using the Monotone Chain algorithm [4]. See Fig. 14 for the steps of the Monotone Chain algorithm for a sample Cartesian point set. The Monotone Chain algorithm works by incrementally constructing the lower convex hull from a coordinate-sorted set of points by adding one point at a time and removing a point if it makes a right-hand turn. Since the vertices were in sorted order before the projection, then the vertices will be in sorted order after the projection and flattening. The original vertices that correspond to the points on the lower convex hull are then used to create new edges.

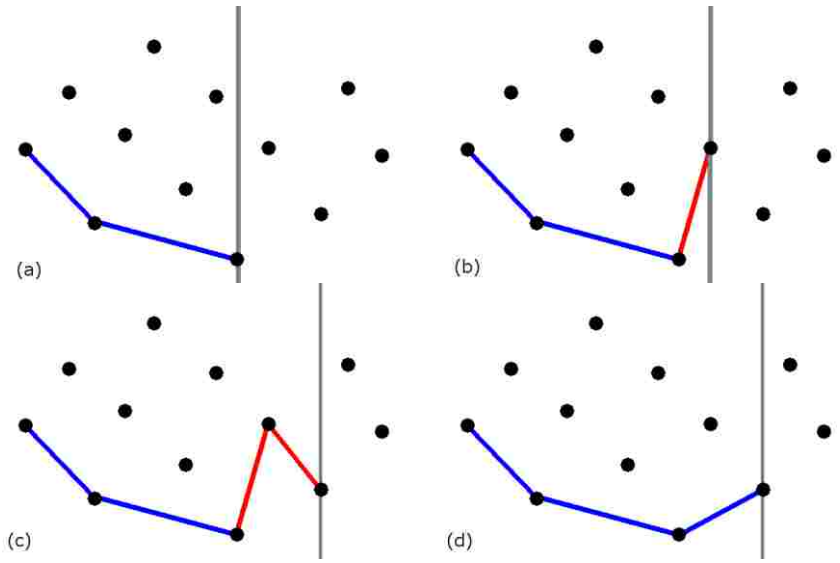


Fig. 14. Steps of the Monotone Chain Algorithm. The vertical grey line sweeps from lowest x-coordinate vertex to highest x-coordinate vertex. (a) The current lower convex hull; (b) The previous lower convex hull with the next vertex added; (c) The next to last point of the lower convex hull makes a right-hand turn, so the next to last point must be removed as it is not part of the lower convex hull; (d) The current lower convex hull after the non-hull point is removed.

Fig. 15 shows the boundary layer vertices in the Cartesian plane for a coarse-grained airfoil. These vertices are then projected onto the paraboloid centered at the median vertex, see Fig. 16. The paraboloid is then flattened onto the vertical plane, Fig. 17, and the lower convex hull is computed, which becomes the dividing path, see Fig. 18. Since the projection and flattening operations are involutory, these operations can be reversed. Fig. 19 shows the paraboloid with the dividing path computed from the vertical plane, and Fig. 20 shows the original vertices on the Cartesian plane with the dividing path. This dividing path is a Delaunay path since all edges in the path are side of triangles in the final Delaunay triangulation.

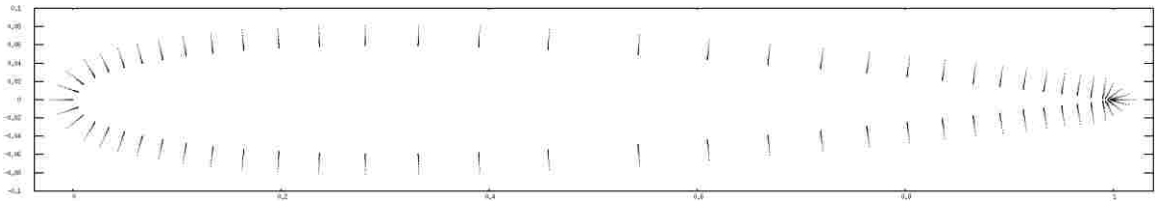


Fig. 15. NACA 0012 Airfoil with boundary layer vertices in the Cartesian plane

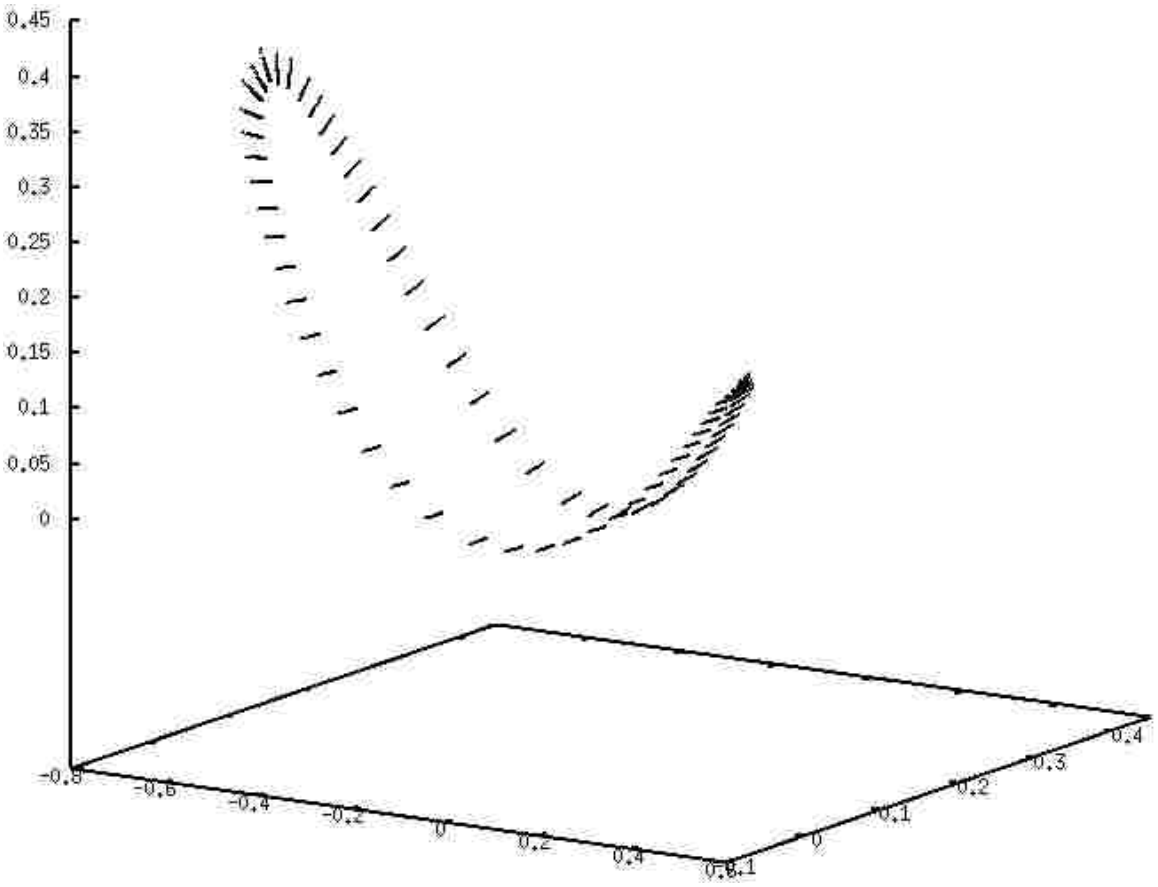


Fig. 16. NACA 0012 Airfoil with boundary layer vertices projected onto a paraboloid centered at the median vertex

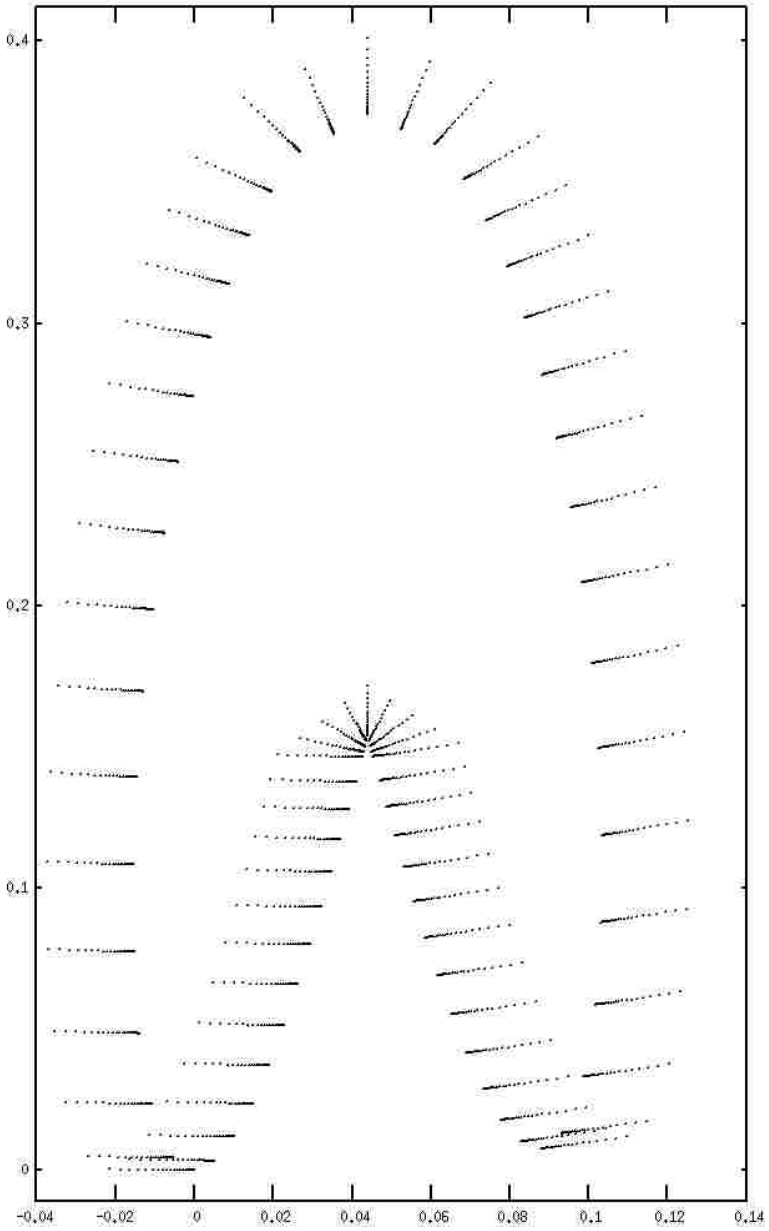


Fig. 17. NACA 0012 Airfoil with boundary layer vertices flattened onto the vertical plane

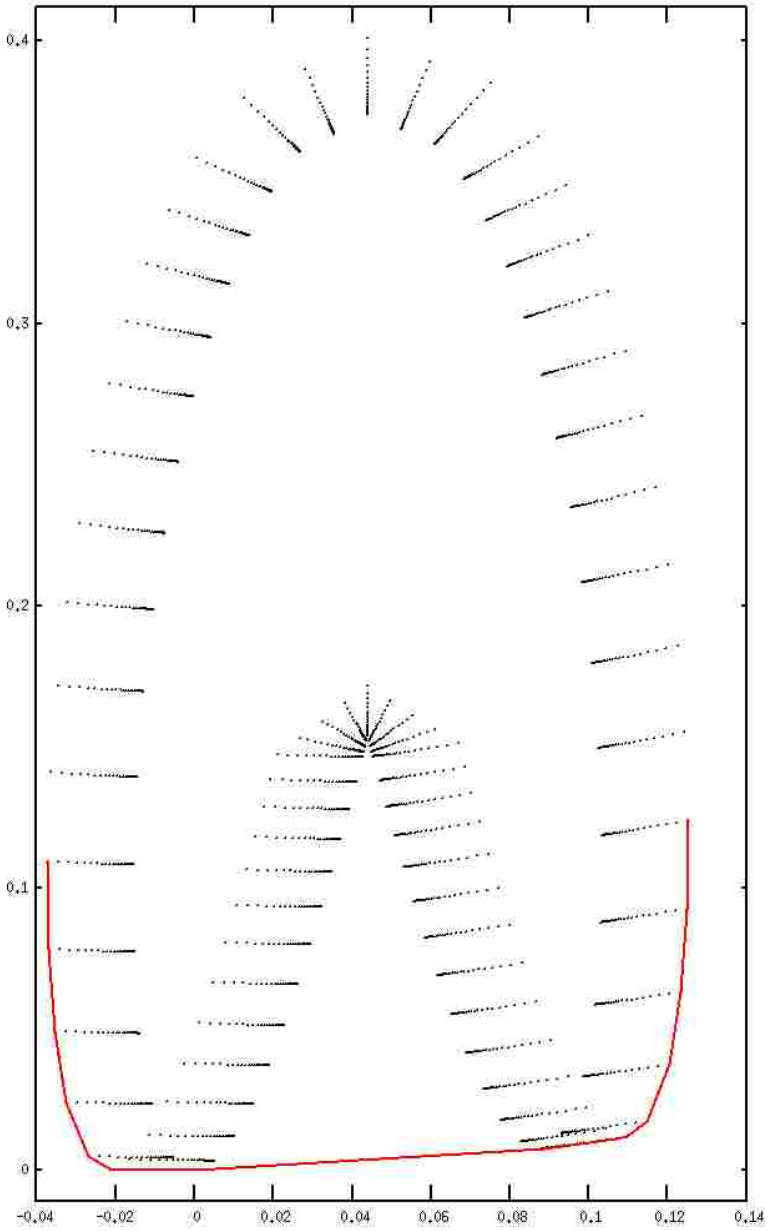


Fig. 18. NACA 0012 Airfoil with boundary layer vertices flattened onto the vertical plane with dividing path (red)

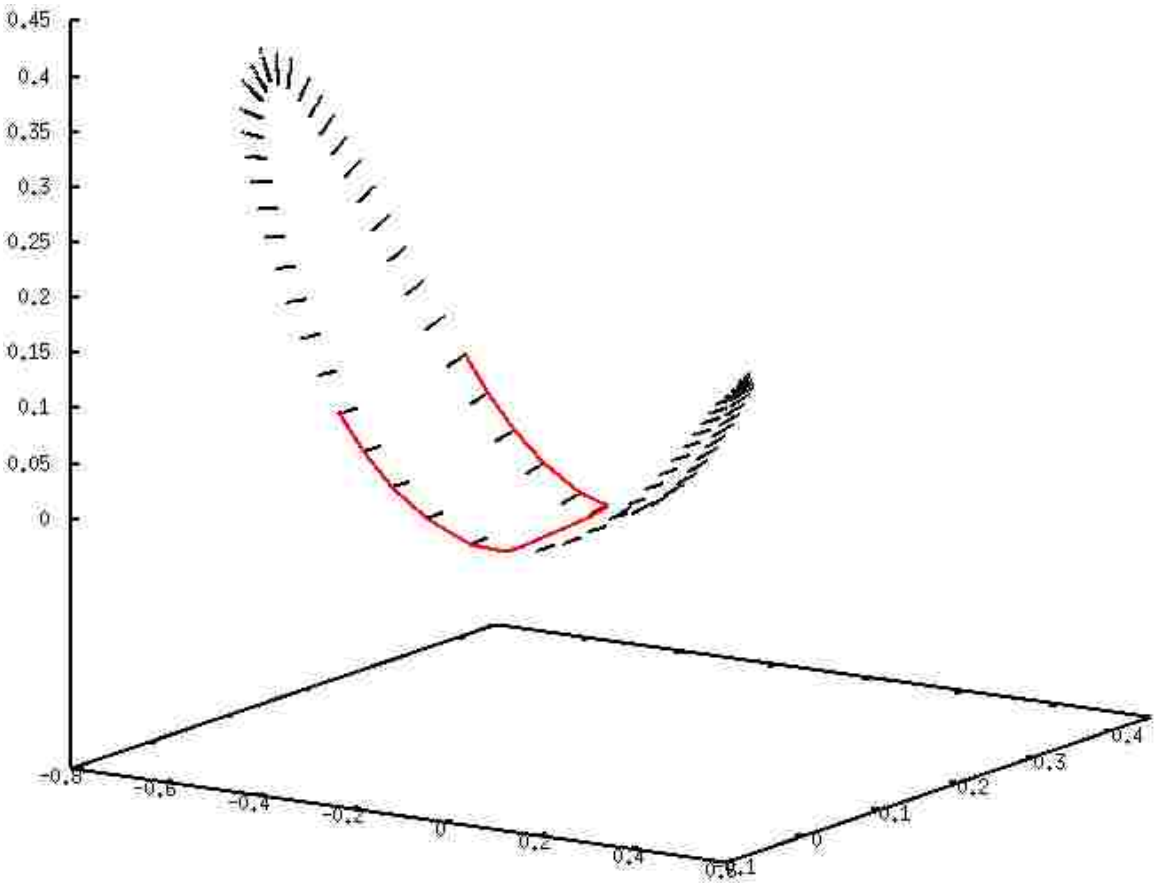


Fig. 19. NACA 0012 Airfoil with boundary layer vertices projected onto a paraboloid centered at the median vertex with dividing path (red)

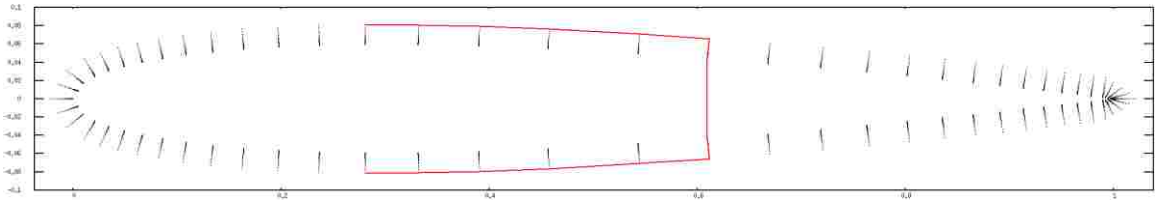


Fig. 20. NACA 0012 Airfoil with boundary layer vertices in the Cartesian plane with dividing path (red)

4.4 SUBDOMAIN PARTITIONING

Two new Subdomain objects are then created and the original Subdomain object's vertices are then partitioned into the two new Subdomain object's vertices. For simplicity and without loss of generality, assume the cut axis is the y-axis. All vertices in the original Subdomain object which have an x-coordinate less than the median vertex's x-coordinate are added to the left Subdomain object's vertices while vertices with x-coordinates greater than the median vertex's x-coordinate are added to the right Subdomain object's vertices. Additionally, vertices that comprise the lower convex hull are added to both the left and right Subdomain object's vertices. This partitioning step is done by iterating over the original Subdomain object's x-sorted and y-sorted vertices in order to maintain the sorted vertices in linear time. These new Subdomain objects are then added to a shared stack for further decomposing.

4.5 SUBDOMAIN TRIANGULATION

Once a subdomain has been sufficiently decomposed, the enclosing border of edges is determined and then triangulated with Triangle. The task of determining the enclosing border for the subdomain is visited in the implementation section. The criteria for if a subdomain is sufficiently decomposed stays true to the original algorithm, whereas if there are no internal vertices (vertices not marked as being on the subdomain boundary), then the subdomain's decomposition is halted. We have added two more variable constraints as we are utilizing this approach as a coarse-partitioner: if the number of vertices is less than a given tolerance or if the decomposition's recursive level of a particular Subdomain object reaches a given tolerance, then decomposition ceases for this subdomain.

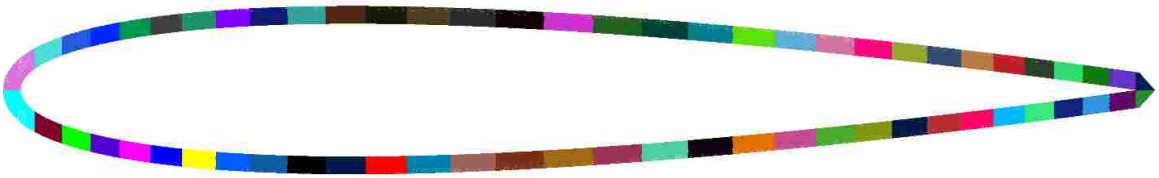
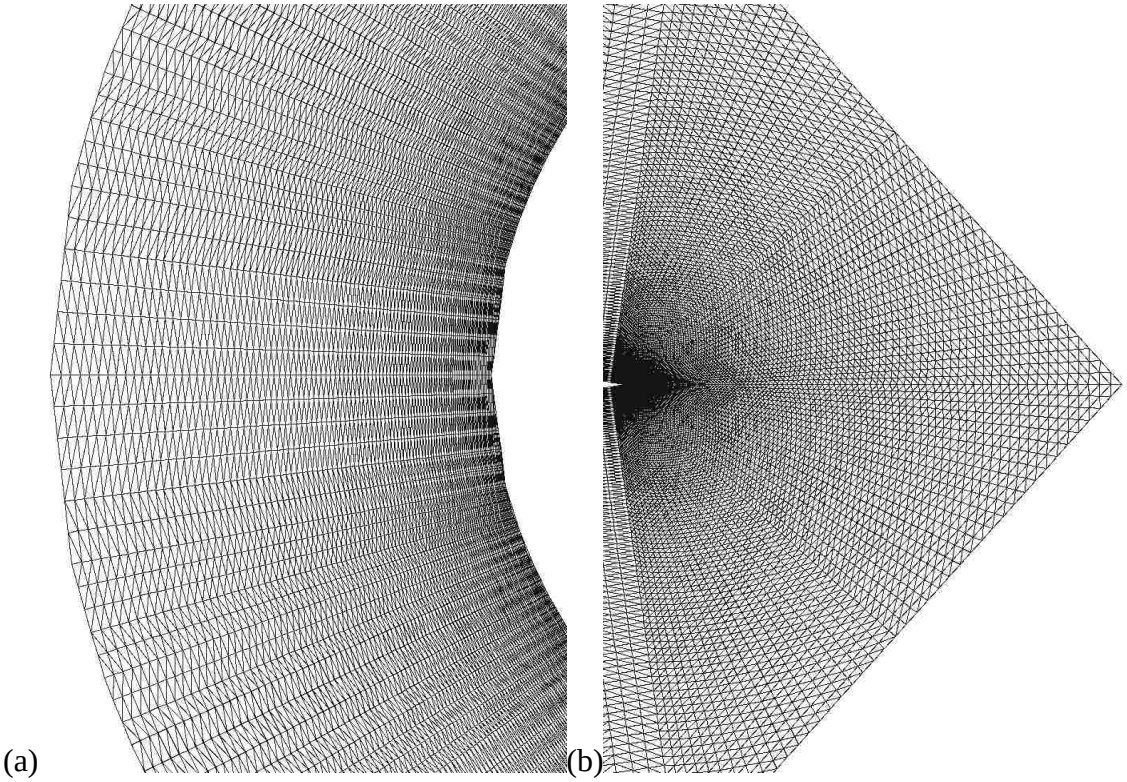


Fig. 21. Decomposed Delaunay subdomains for NACA 0012 Airfoil

Fig. 21 shows the boundary layer decomposed into 64 Delaunay subdomains, which can all be triangulated independently. Fig. 22 shows a fine-grain, high-density boundary layer mesh of the leading edge and trailing edge of the NACA 0012 Airfoil with refining rays added after the subdomains have been triangulated. The ray angle tolerance used is four degrees, yielding a dense distribution of elements in these critical areas.



(a) (b)

Fig. 22. (a) Triangulation of leading edge region with refining rays; (b) Triangulation of trailing edge region with refining rays

Typically for airfoils, the leading edge and trailing edge require more refinement. This is due to two factors: the implicit geometrical representation has larger angles between surface normals in these regions, and the governing PDE. The trailing edge is a highly discussed matter in the CFD field due to the Kutta condition and the presence of the stagnation points near the trailing edge and the trailing edge wake. Without any knowledge of the angle of attack through the fluid, we are only able to prescribe a high degree of refining rays uniformly to the trailing edge region, creating high-resolution, graded isotropic elements since the location of the wake is variable due to the angle of attack. The leading edge region is a high-gradient region with a stagnation point, so quality is critical as this is the first part of the airfoil that contacts the fluid, so if the leading edge region is not accurately discretized, this inaccuracy has an effect on the PDE solution.

4.6 DECOMPOSITION PERFORMANCE

The decomposition algorithm is used as a coarse-partitioner, where the dividing step requires O(n) time with our current implementation. The median vertex can be determined in constant time, projecting the vertices requires $\Theta(n)$ time, computing the lower convex hull is performed in O(n) time, maintaining the primary-axis-sorted vertices requires O(n) time for the move and copy operations, and maintaining the cut-axis-sorted vertices also requires O(n) time due to the comparisons and move and copy operations. Fig. 23 shows the execution times for triangulating the boundary layer based on different depths of decomposition. The depths of decomposition that were measured are 3, 4, 5, 6, and 7; yielding 8, 16, 32, 64, and 128 subdomains, respectively. The time to triangulate the domain with Triangle is also shown.

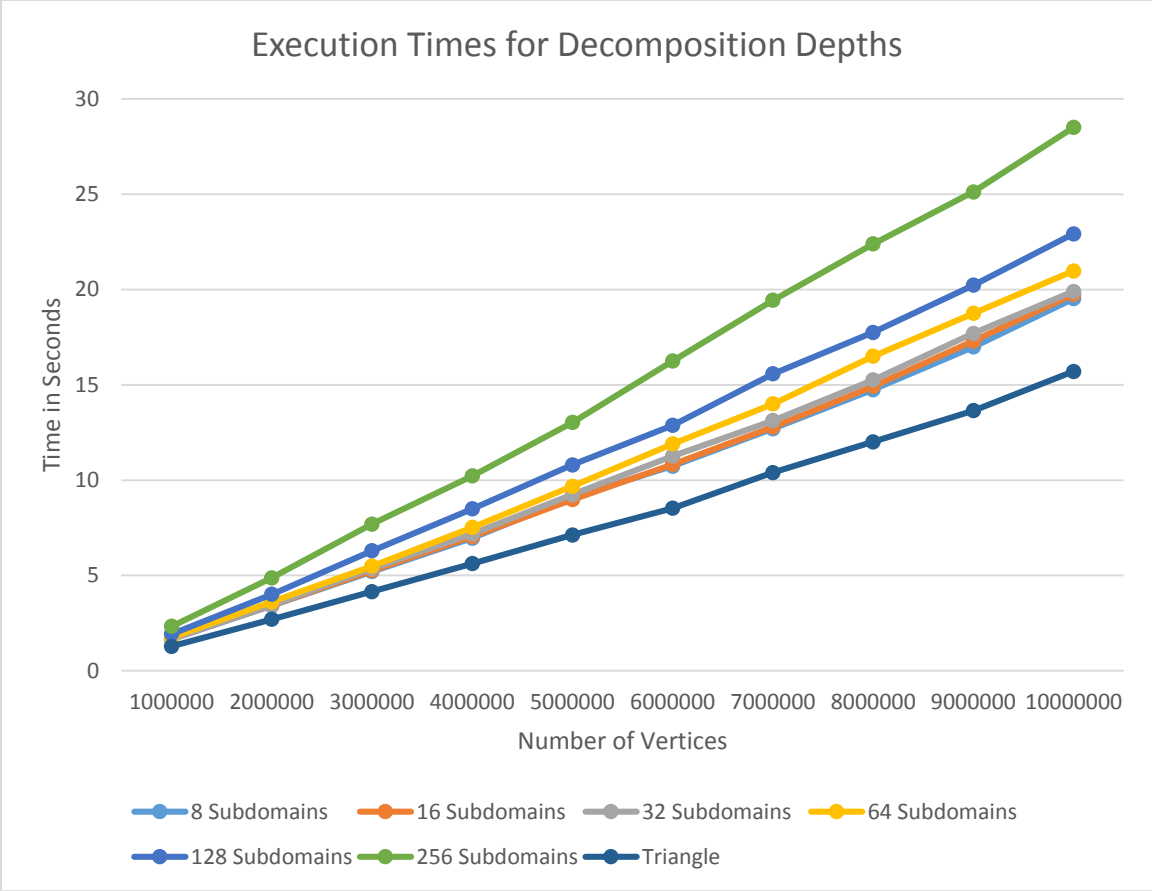


Fig. 23. Execution times for different decomposition depths

Thus, the total execution times for coarse-partitioning has a loglinear complexity solution, so using the decomposition algorithm as a coarse-partitioner for the triangulation of the boundary layer is favorable to the overall performance of the application.

CHAPTER 5

ISOTROPIC INVISCID REGION

For generating the isotropic inviscid region, we use a sizing function along with Triangle's ability to use a user-defined area constraint, typically defined as a sizing function based on distance from the airfoil, for Delaunay refinement to provide a smooth gradation of triangle size based on distance from the initial geometry towards the far-field. Since the size of the inviscid region is typically a factor of 30 to 50 chord lengths from the initial geometry, the time to refine the inviscid region is extremely high, due to the large area of the domain compared to the boundary layer. Thus, we need to generate the inviscid region in parallel. To facilitate the parallel refinement, we follow the approach of generating graded Delaunay decoupling paths as presented by Linardakis and Chrisochoides in [5], in order to distribute the refinement work among the worker threads by creating subdomains which can be concurrently refined. In order to generate subdomains that can be efficiently triangulated and refined by Triangle, it is essential to create subdomains that are convex and do not contain any holes since Triangle first creates an initial triangulation and then removes elements inside concavities and holes from the initial triangulation. Using rectangles for the near-body inviscid region and layers of geometrically similar trapezoids moving towards the far-field, we can eliminate the cost associated with removing triangles from concavities. Rectangles and trapezoids are simple shapes that are used to partition the inviscid region. In order to generate the graded Delaunay decoupling path of a prospective subdomain, we compute a value k from (1),

where A is the area of the desired element at the given vertex, for each of the four vertices on the corners of the trapezoidal subdomain.

$$k = \frac{1}{2}\sqrt{\frac{A}{\sqrt{2}}} \quad (1)$$

Using the k value for each of the four vertices, for each edge, E, comprised of vertex u and vertex v, a graded Delaunay decoupling path is created by discretizing the path from vertex u to vertex v. Assume $k_u \leq k_v$ for the k values at u and v respectively. We choose an integer in the range of $|E|/2k_u$ and $\sqrt{3}|E|/2k_v$ to be the number of segments that edge E will be split into. If there is no integer in the range, then vertex v is too far from vertex u, so we choose a new vertex v until we have an integer in the range. After the four edges of the trapezoid have been discretized, the border is constructed and the subdomain is triangulated and refined using the sizing function with Triangle. This sizing function is the same sizing function used to compute the k values at the corner vertices. The size of each trapezoidal subdomain needs to be small enough to simultaneously yield enough subdomains to be refined while also providing a sufficient amount of work to support the load balancing scheme and minimize idle time.

Fig. 24 shows the decoupled subdomains after Delaunay refinement. The resulting mesh is globally Delaunay. Depending on the number of threads available, the inviscid region may need further decoupling. To maintain a good load balancing scheme, the area of the decoupled subdomains grows as the subdomains are further away from the initial geometry, since the sizing function dictates that a larger element area is sufficient further away from the geometry. The decoupled subdomains near the geometry have a smaller area because the size of elements in these regions near the boundary layer require smaller elements. Thus the work to be done by Delaunay refinement is greater for a fixed

subdomain area if the subdomain is closer to the boundary layer. The mesh is a fine-grain mesh and the inviscid region is 50 chord lengths in each coordinate direction from the airfoil. Assuming that the chord length of the airfoil is one, then each side of the bounding box containing the inviscid region has a length of one hundred. The total mesh size is 3,335,075 elements.

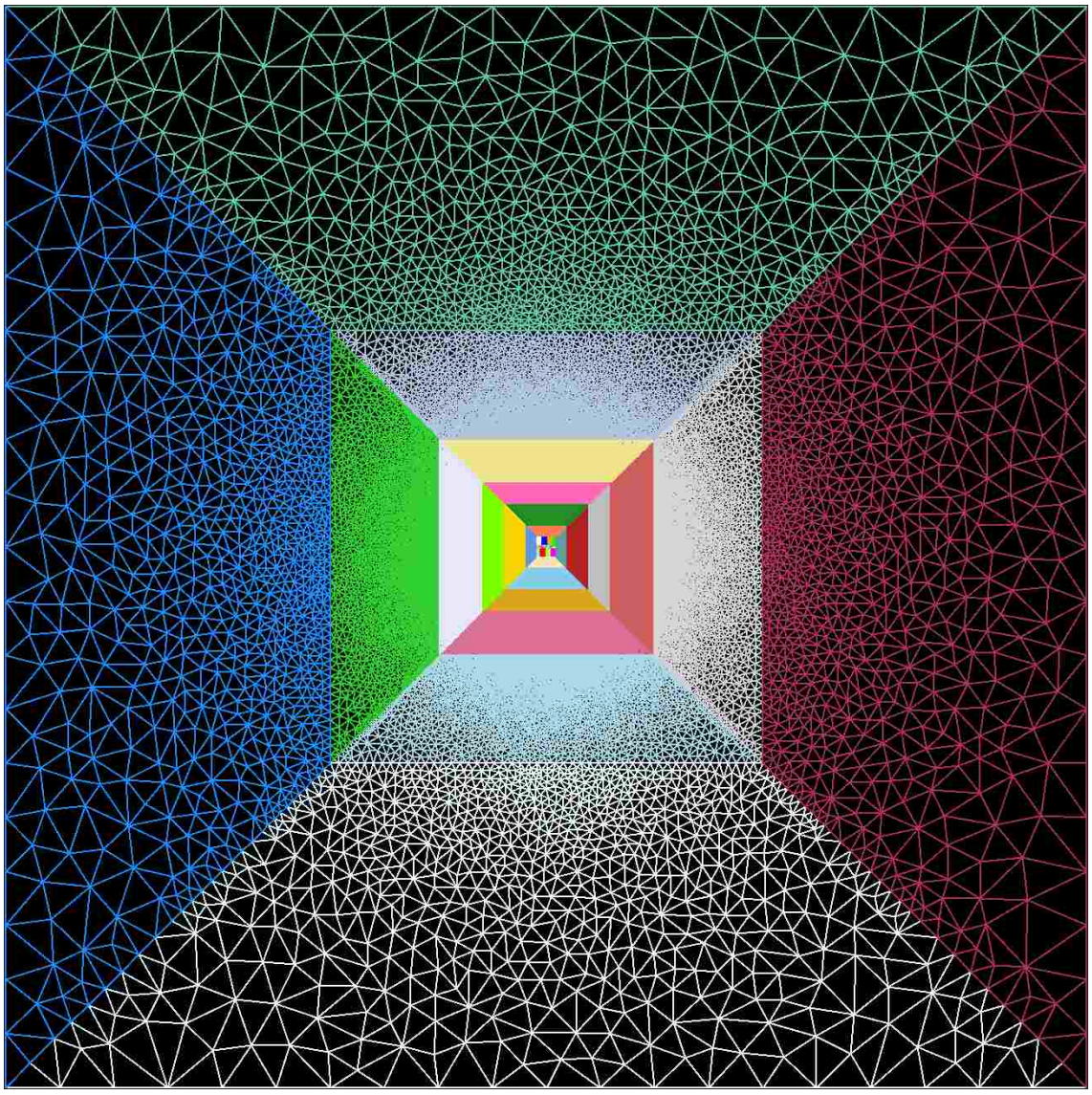


Fig. 24. Fine-grain mesh for NACA 0012 Airfoil with inviscid region size of 50 chord lengths

Fig. 25 shows the stitching region with the smooth gradation of elements from the outer layer of the boundary layer near the trailing edge to the near-body inviscid region. The growth function used to generate the boundary layer has a growth rate of 15% and an initial layer height of 2.5×10^{-5}. Fig. 26 shows a larger portion of the near-body inviscid region for the same mesh towards the leading edge.

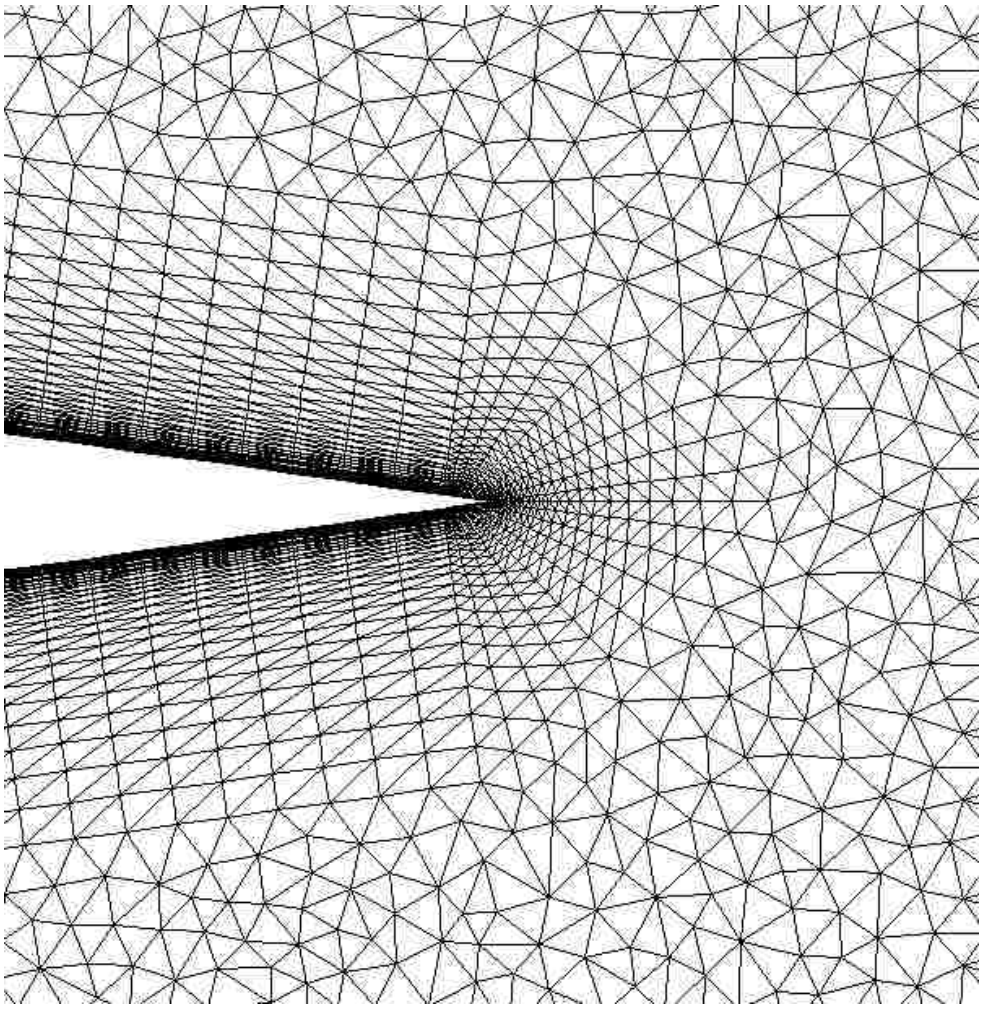


Fig. 25. Boundary layer to near-body inviscid region transition for NACA 0012 Airfoil with 30 viscous layers

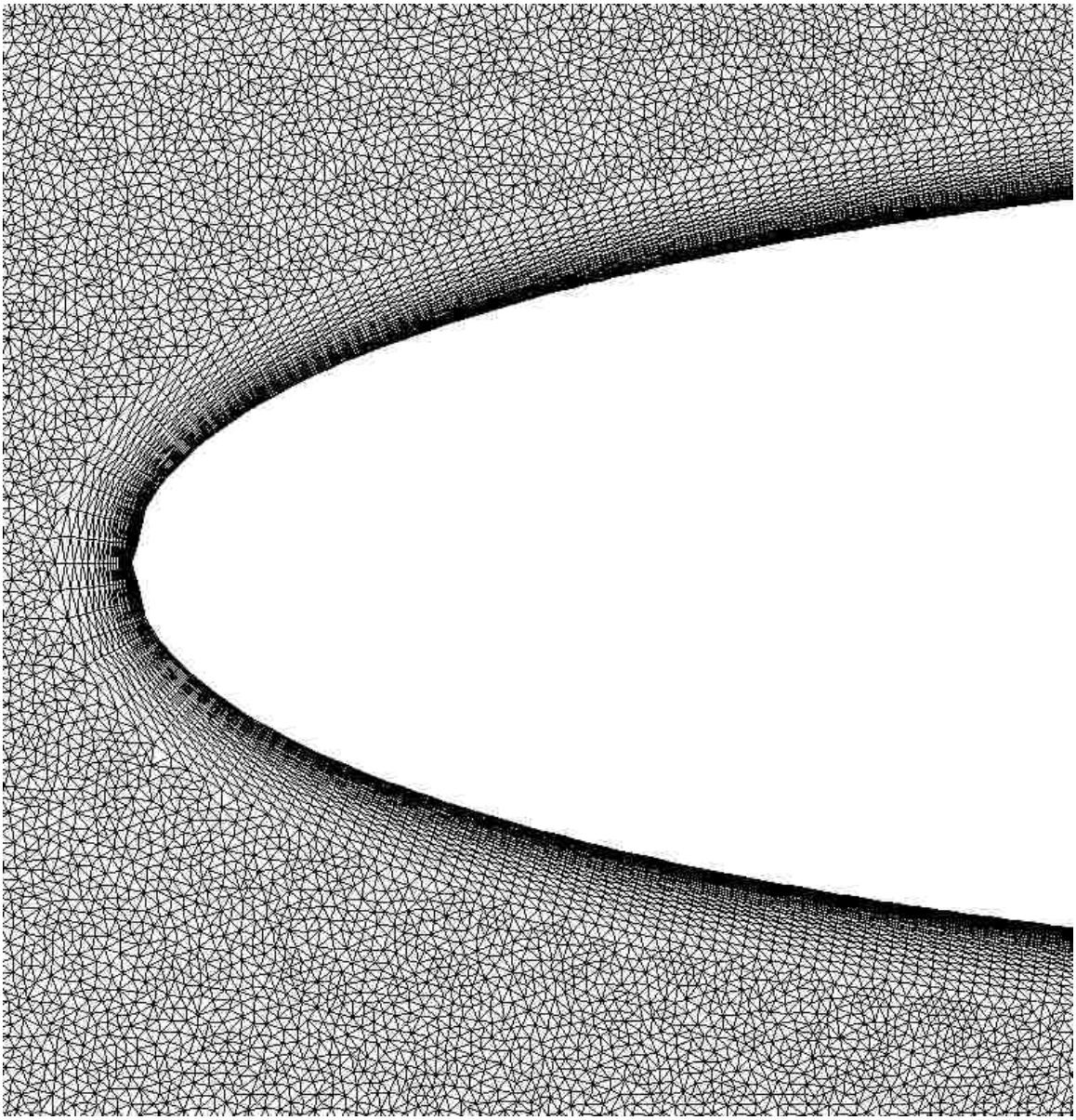


Fig. 26. Leading edge of NACA 0012 Airfoil and near-body inviscid region

Fig. 27 shows the corresponding coarse-grain mesh with an inviscid region size of 50 chord lengths and is comprised of 451,091 elements and Fig. 28 is a zoomed in view of the same mesh.

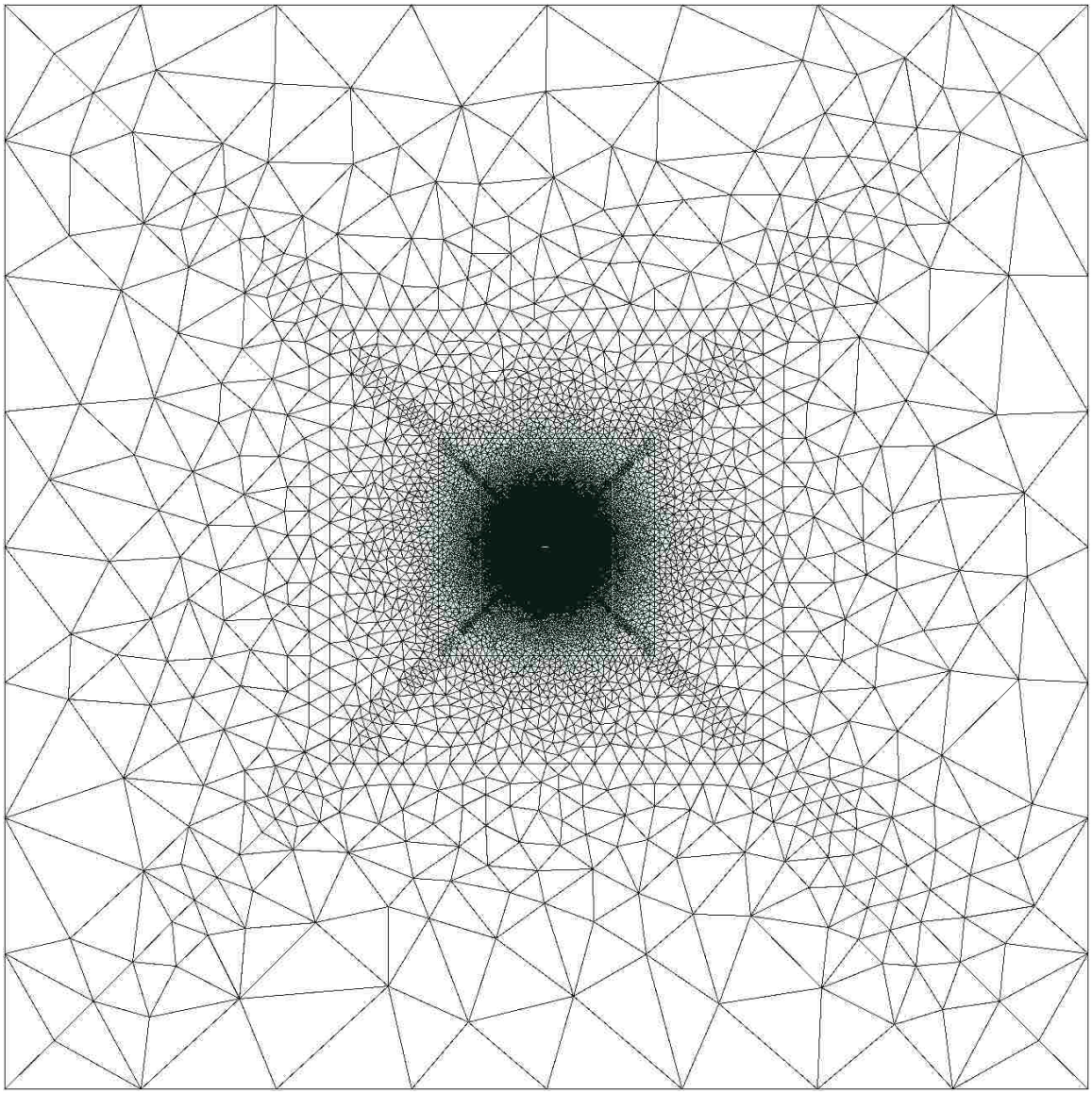


Fig. 27. Coarse-grain mesh for NACA 0012 Airfoil with inviscid region size of 50 chord lengths

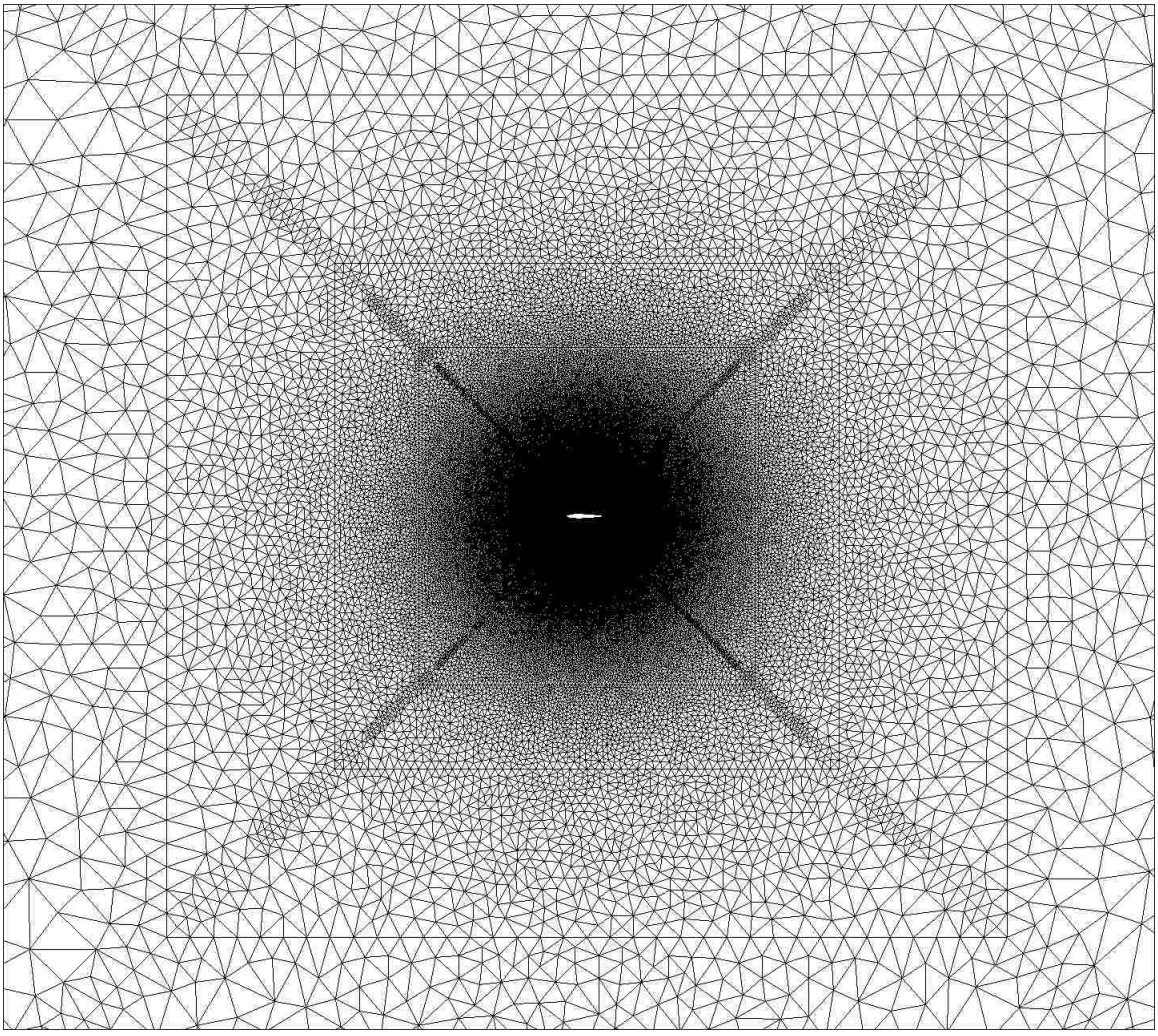


Fig. 28. Zoomed-in region of Fig. 27

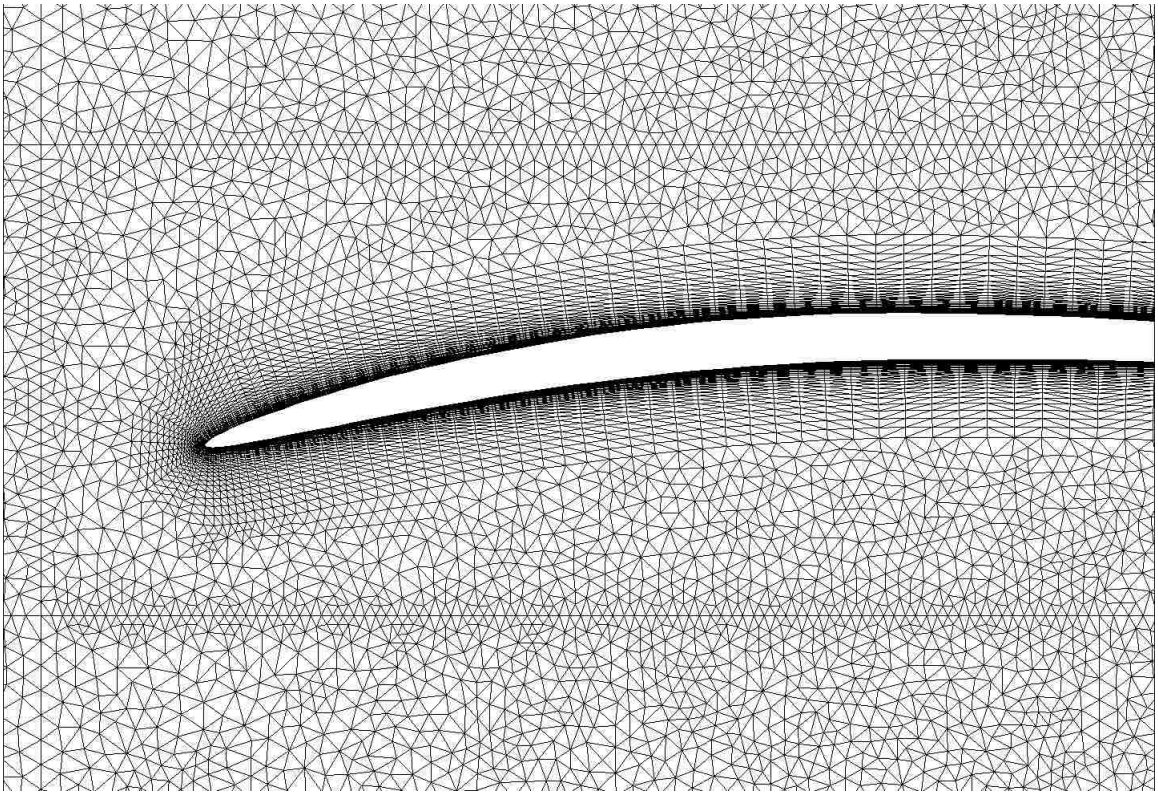


Fig. 29. Leading edge half for Wright 1903 Airfoil and near-body inviscid region

Fig. 29 shows the front half of the Wright 1903 Airfoil from the leading edge while Fig. 30 shows the corresponding rear half from the trailing edge. This mesh was generated from a PSLG using cosine spacing and is comprised of 30 anisotropic layers in the boundary layer. The resulting mesh contains 196,291 triangles.

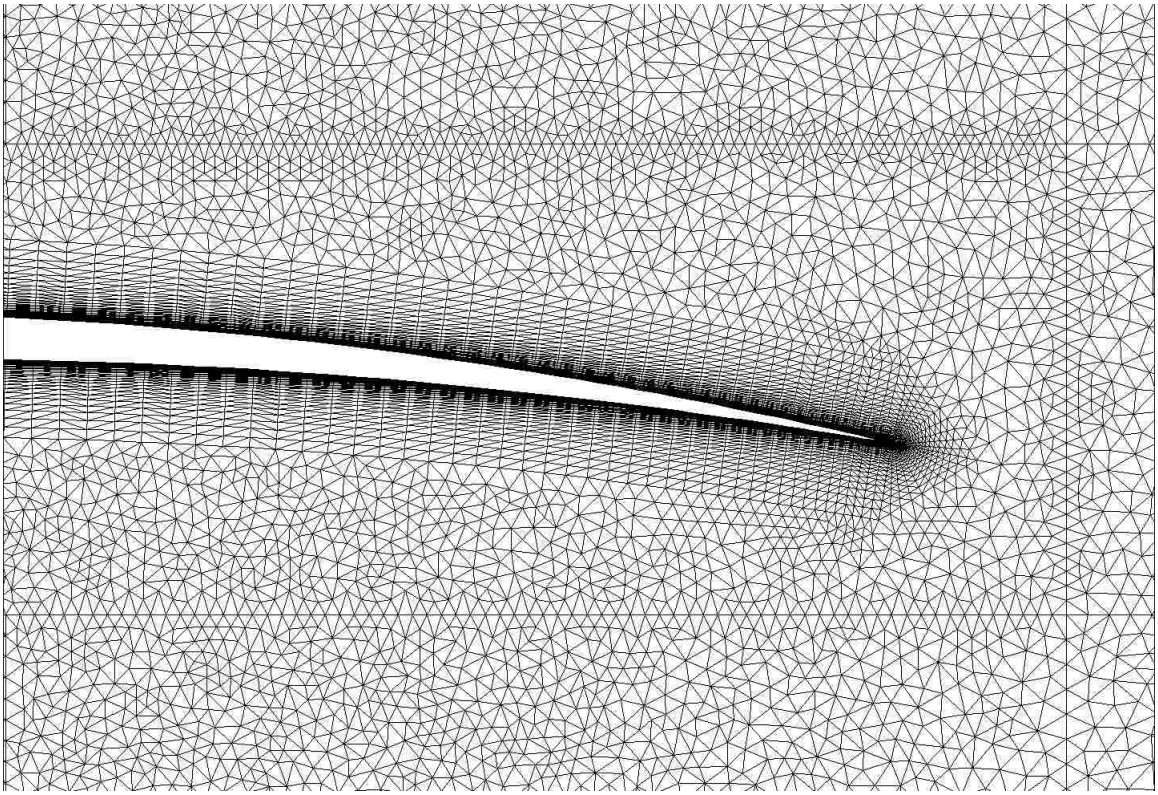


Fig. 30. Trailing edge half for Wright 1903 Airfoil and near-body inviscid region

CHAPTER 6

IMPLEMENTATION

After the first implementation, benchmarking times, see Fig. 31, were taken to examine the scalability of the algorithm. The vertical axis shows eight groupings of average executions, run with one to eight threads for the decomposition and triangulation of the boundary layer. The breakdown of execution percentage is organized by thread id, starting at zero. The performance of the first implementation was deemed unsatisfactory, which prompted implementation changes to achieve better performance results. The upcoming subsections discuss the current implementation decisions.

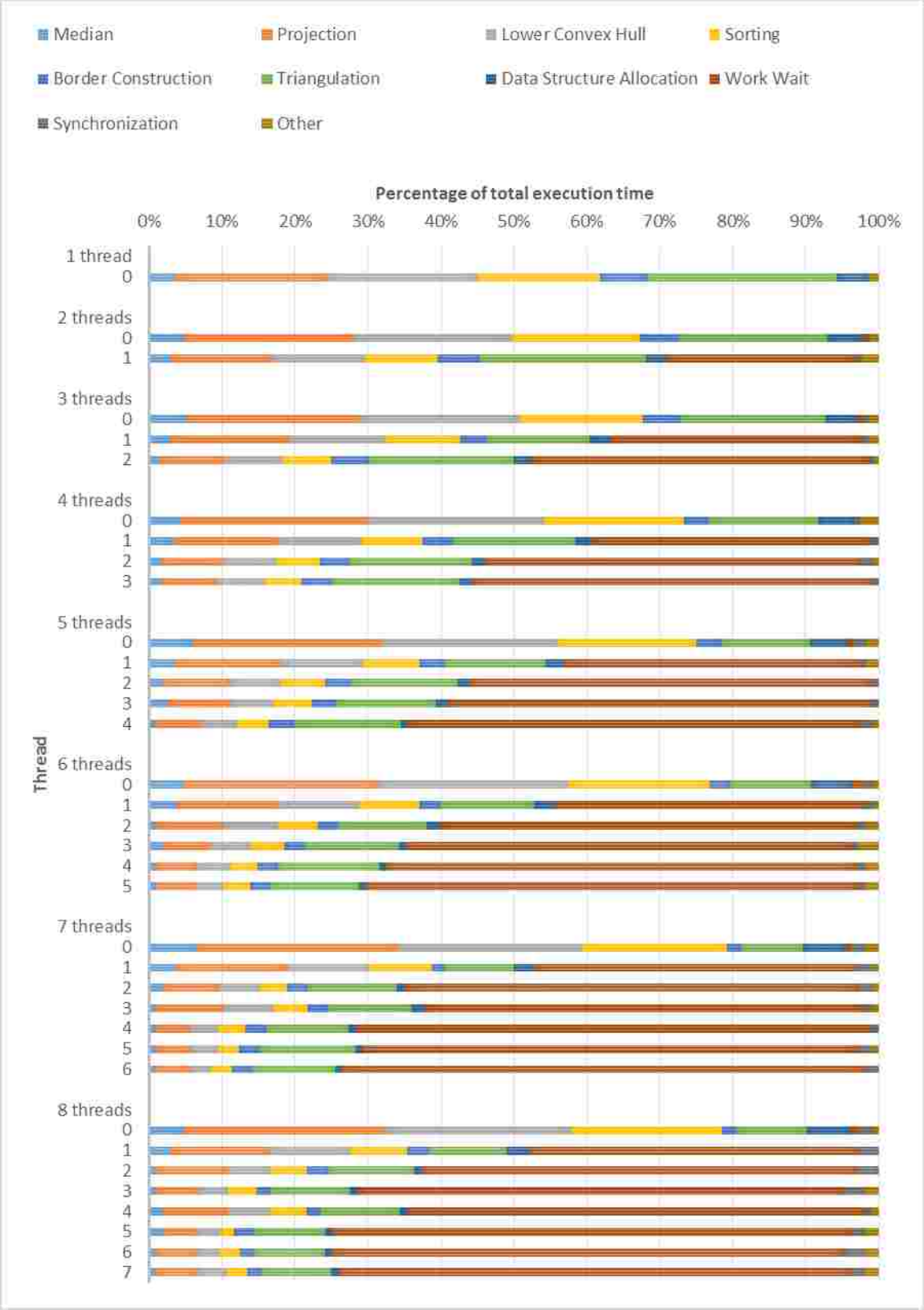


Fig. 31. First implementation execution time percentages

Practical data structure and algorithm designs must be utilized in order to achieve a scalable and cache-friendly parallel application. A contiguous memory container is used for storing the Vertex objects in order to take advantage of spatial locality in the cache since the Vertex objects are iterated over in order for projecting and flattening the vertices and for computing the lower convex hull during the boundary layer triangulation process. As previously referenced, the original algorithm calls for using the median vertex of all internal points, but this requires iterating over the collection of vertices. This was replaced by the constant time selection of the median vertex of all of the vertices.

6.1 POINTSET PROJECTION

For the task of projecting the points onto the paraboloid and flattening them onto the vertical plane, we made the decision to include the data for the projected coordinates with the Vertex objects, as opposed to creating the necessary data to store the projected coordinates when they are needed. This avoids the repetitive allocation and deallocation of the projected coordinates since the data is allocated once at the creation of a Vertex object. For large collections of Vertex objects, storing the projected coordinates detached from the associated Vertex objects causes some Vertex objects to be replaced by the collection of projected coordinates in the cache. This causes additional overhead to the algorithm due to the page faults for bringing in the collection of projected coordinates to the cache before the lower convex hull is computed, and the page faults for bringing the Vertex objects back in the cache after the lower convex hull is computed. The computations for projecting a point onto the paraboloid and then flattening the point onto the vertical plane is performed in a single step to map the Cartesian point to the vertical plane directly without projecting the point on the paraboloid first.

6.2 DATA ALLOCATION

Since a portable and modern C++ object-oriented implementation is desired, nuisances in compiler optimization techniques must be utilized. GNU's state-of-the-art compiler, GCC [6] and other compilers offer similar optimization techniques for reordering and eliminating instructions. One disadvantage of modern C++ and the use of the Standard Template Library (STL) with objects is the strong connection between data allocation and object construction. It is time consuming for one thread to allocate all of the data and default construct, or create, these objects because construction cannot be performed as an aggregate operation on a set of objects. However, data allocation can be performed as an aggregate operation. Object creation is done in two parts: first the data is allocated where the object will be stored and then the object's data members are initialized. A default constructor, which is responsible for initializing the object's data members, is overridden. This overridden constructor performs no default initializations of the object. Compiler optimization techniques replace these default constructions with a No Operation (NOP), which is an instruction that does no operation and is consequently extremely fast. This is especially beneficial for the parallel point-insertion routines because the data for the new Vertex Objects can be allocated extremely fast, which is important as this is one of the sequential tasks of the program.

6.3 VERTEX PARTITIONING

For the triangulation of the boundary layer, the task of partitioning the vertices after the lower convex hull is computed, a naïve approach was used which involved determining the cut-axis-sorted vertices by sorting the primary-axis-sorted vertices based on the non-sorted coordinate. This was replaced with the current approach of iterating over the cut-

axis-sorted vertices from beginning to end and determining, based on the median line, which subdomain the vertex belongs in. This replaces the $O(n\log_2 n)$ sorting step with the iterative $\Theta(n)$ approach. Additionally, the primary-axis-sorted vertices can be split at the median vertex. For simplicity and without loss of generality, assume that the cut axis is the y-axis, so the primary axis is then the x-axis. This allows for all of the vertices before the median vertex to be placed in the left subdomain, and all of the vertices after and including the median vertex to be placed in the right subdomain. The benefit of this is that no comparisons, and thus no branch operations need to be performed for the primary-axis-sorted vertices, or the x-sorted vertices in this case. However, the vertices of the lower convex hull are not represented completely in the left and right subdomains. In order to include the vertices in the lower convex hull, the vertices with an x-coordinate less than the median vertex's x-coordinate are placed at the front of the right subdomain's x-sorted vertices while the vertices with an x-coordinate greater than or equal to the median vertex's x-coordinate are placed at the end of the left subdomain's x-sorted vertices. These lower convex hull vertices are actually added to the right subdomain's x-sorted vertices before the original subdomain's x-sorted vertices to avoid the cost of reshuffling the vertices since we are using contiguous storage. Additionally, the data for the original subdomain is reused for the left subdomain. Not only does this eliminate the cost of deallocation for the original subdomain and allocation for the left subdomain, but for the task of partitioning the primary-axis-sorted vertices, the vertices before the median vertex are already in the proper location, which eliminates half of the data moving costs which is half of the work required for partitioning the primary-axis-sorted vertices.

6.4 BORDER CONSTRUCTION

Recall that once a subdomain is sufficiently decomposed, that its border is constructed in preparation for triangulation. To avoid the cost of determining the new border at each decomposition step, the border is constructed only after the subdomain is sufficiently decomposed. This is facilitated by a dynamic bitset representing which vertices are contained within the subdomain. A dynamic bitset is a contiguous container of indexed bits which can be modified and is a specialization defined by the C++ Standard [31]. Each Vertex object also contains a set of all edges that its vertex is incident upon. For the pseudo-structured design of this region of the mesh, the upper bound of the number of edges that a vertex may be incident upon is six. To construct the border, the set of incident edges where the current vertex is listed as the first endpoint of the edge are iterated over and checked to determine if the second endpoint exists in the subdomain. If this holds true, then this edge is added to the border. The check to determine if the current vertex is the first endpoint is to ensure that there are no duplicate edges added. Using the dynamic bitset, the check for existence can be performed in constant time. This method is also favorable in regards to cache performance since the dynamic bitset only uses one bit per vertex which allows for a large portions of the bitset to fit on a cache line and fill the cache line completely.

6.5 TRIANGE

Upon examining the source code for Triangle, we noticed that the input vertices are sorted by their x-coordinate upon invocation. This allowed us to remove the sorting step from Triangle since we maintain the x-sorted vertices upon each decomposition step and use the x-sorted vertices to populate the input data used to call Triangle. This speeds up the

execution by eliminating the allocation and deallocation costs of pushing and popping stack frames from the call stack during the sorting step. Additionally, if a subdomain is short and wide or determined to have a small number of vertices, then Triangle is instructed to use only vertical cuts in the divide-and-conquer algorithm in order to speed up the triangulation process.

6.6 THREAD IDLE TIME

See Fig. 31 for the series, "Work Wait" which is thread idle time while a thread waits for a subdomain to decompose. Thread idle time drastically increases since a single thread is responsible for a single decomposition procedure of a particular subdomain. This is because there is initially only one subdomain, the entire boundary layer. So all but one thread must wait until this first subdomain is decomposed, but this only creates two new subdomains to replace the old one. The number of subdomains double at each level of decomposition, so at the k^{th} level of decomposition, the number of subdomains is equal to 2^k. Thus, there are $p\text{-}2^k$ threads waiting at the k^{th} level of decomposition where p is the total number of threads running for the application. In order for each thread to have a subdomain to decompose, $\log_2 p$ levels of decomposition must be achieved. The decomposition of the initial subdomain is the sequential portion of the algorithm, and all other decompositions before $\log_2 p$ levels of decomposition are reached are not fully parallel, which affect the overall scalability and performance of the algorithm according to Amdahl's Law. The waiting threads need to be given meaningful work to eliminate thread idle time. Since the creation time of the decoupled inviscid subdomains is negligible, the waiting threads begin by meshing the decoupled inviscid subdomains while a single thread works on the decomposition of the boundary layer subdomains.

6.7 DIVIDING PATH TRIMMING

Recall Fig. 20, the dividing path in the Cartesian plane. This dividing path has two regions which must be removed: the segments along the outer borders and the segments intersecting the interior, see Fig. 32.

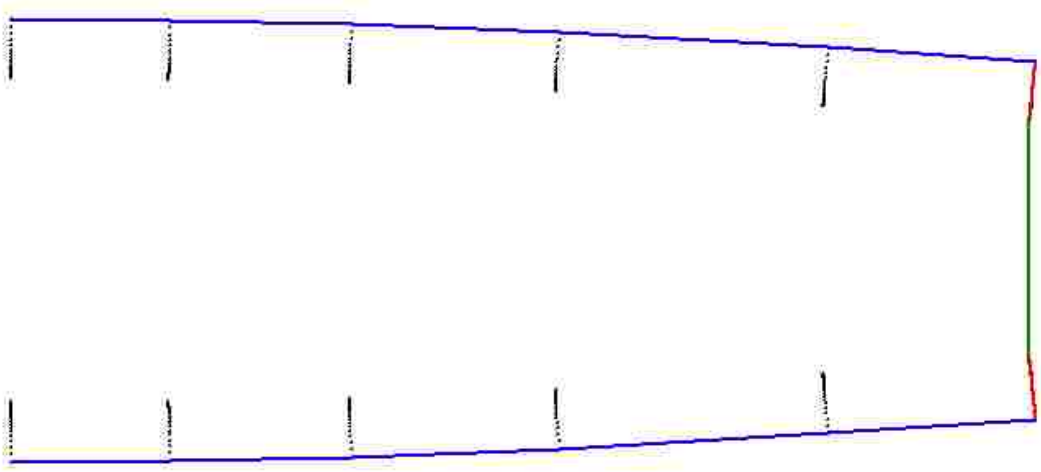
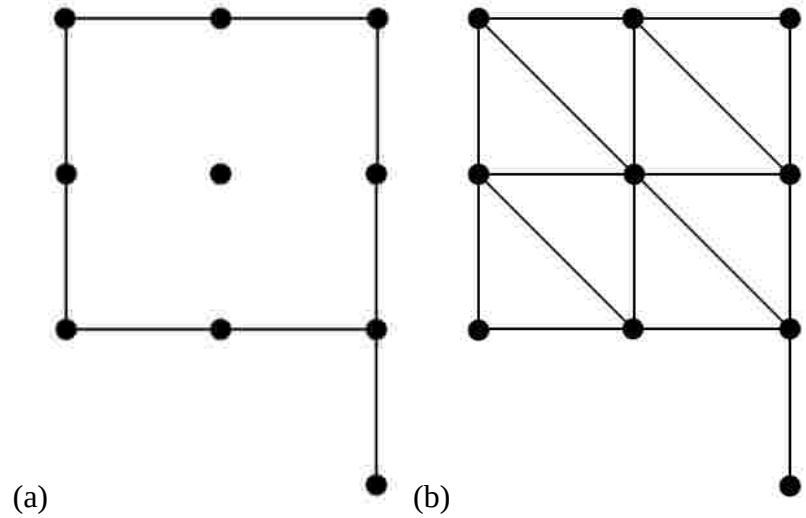


Fig. 32. Dividing path from Fig. 20 with outer borders (blue), interior path intersection (green), and final dividing path segments (red)

If the segments along the outer borders are part of the dividing path, then the resulting subdomains may have pinch vertices, see Fig. 33. These pinch vertices are not part of the functional final triangulation since they are not a vertex of any triangle. Pinch vertices increase the execution time of the triangulation process because Triangle creates an initial triangulation from the input vertices, then conforms the triangulation according to the input edges through edge swapping, and finally removes all triangles that lie outside the input domain. The pinch points cause wasted computations at three steps of the triangulation process. These pinch points will become incident upon initial triangles

generated from the input point set. Then the edges that the pinch points are incident upon are enforced. The last step is to remove the triangles that the pinch points are incident upon.



(a) (b)

Fig. 33. (a) Sample PSLG with pinch point; (b) Triangulation of the interior of the sample PSLG

Segments which intersect the interior pose a problem during the hole-enforcement step of Triangle. The interior of the initial input geometry is used to define a hole by using a pair of xy-coordinates of any point in the interior. The hole-enforcement step executes after the edge-swapping step of the initial triangulation and is initiated by locating the triangle which contains the point used to define the hole. This triangle is removed from the triangulation and all triangles that share an edge with this triangle are removed unless the shared edge is one of the initial edges of the PSLG. This process continues until all triangles in the hole are removed. If there is an edge that divides this hole into two independent regions, then the hole-enforcement step will only remove the triangles on one side of this dividing edge, depending on the location of the point defining the hole.

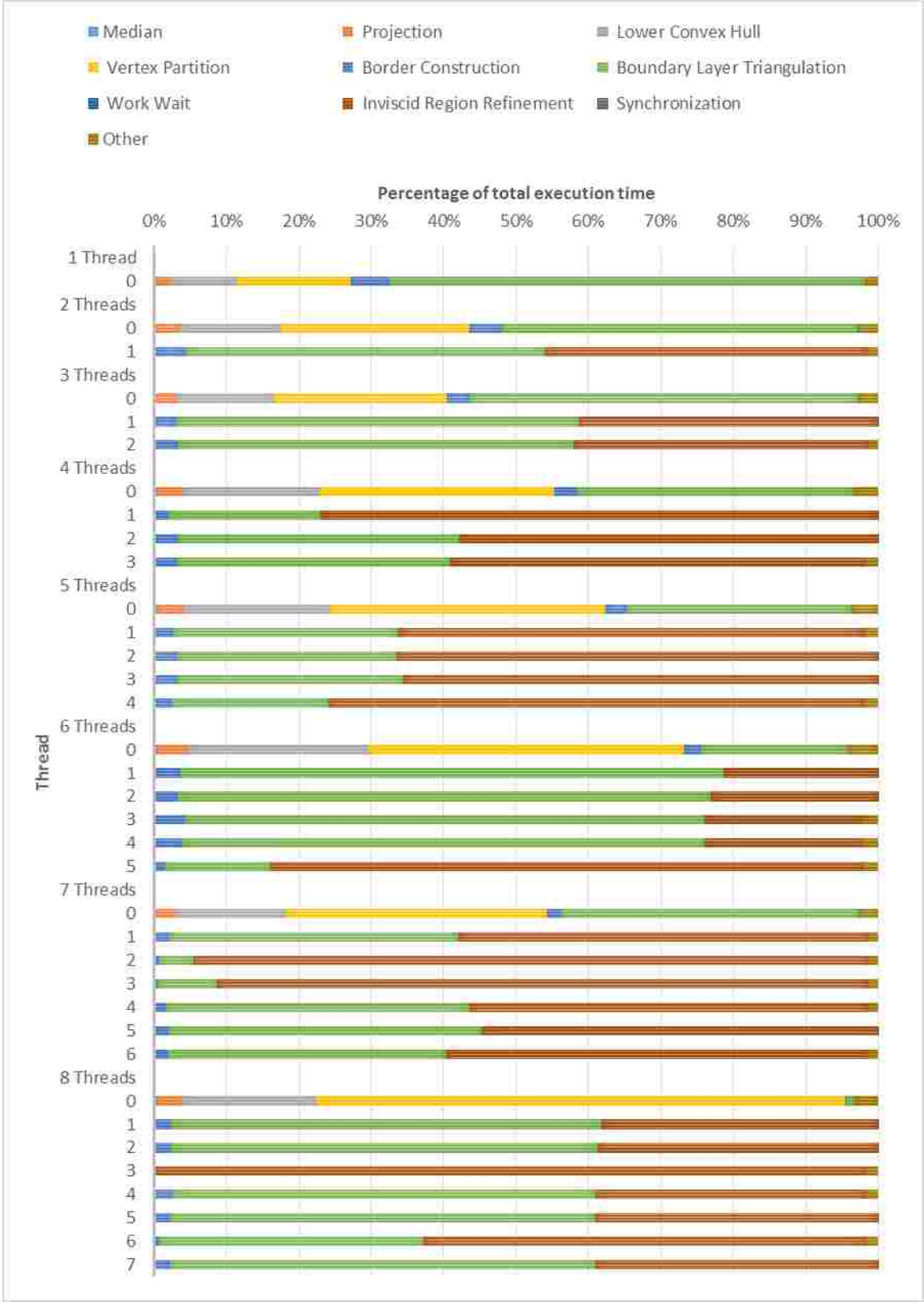


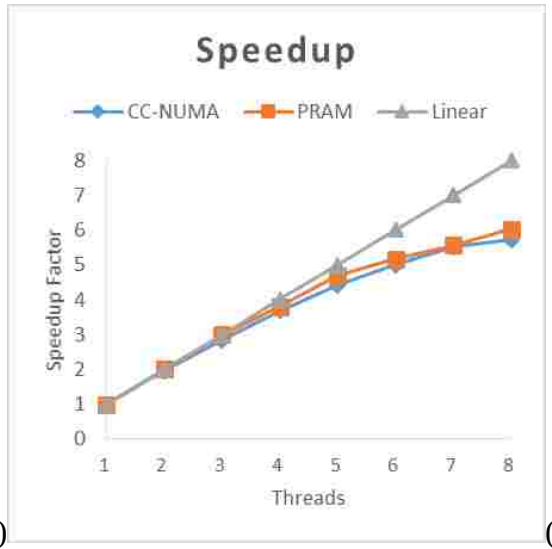
Fig. 34. Current implementation execution percentages

After making the aforementioned implementation decisions, a new performance evaluation was measured, see Fig. 34. The measured execution times are until all boundary layer subdomains have been triangulated, in order to provide comparable results with the first implementation's benchmarking in Fig. 31. There is still more refinement work to be done in the inviscid region. The time to fetch the median vertex is negligible and is not visually apparent in the new performance chart. Additionally, the time for data structure allocation has been absorbed into the vertex partitioning step because move operations are performed for vertices to the right (or above in the case of an x-axis cut) of the median line while only the vertices on the lower convex hull are copied. Since only vertices on the lower convex hull need to be copied and the number of vertices on the lower convex hull is few compared to the current subdomain, the time to allocate the data for these new vertices is negligible. Clearly there is no thread idle time since no thread has to wait for work. Previous implementation's waiting threads now refine the inviscid region when there are no boundary layer subdomains to be triangulated.

CHAPTER 7

EVALUATION

The application has been executed on two separate machines. One machine is a Concurrent-Read Exclusive-Write Parallel Random Access Machine (CREW PRAM) and consists of eight Intel Xeon CPU E7-4830 at 2.13 GHz processors and 32 GB of memory. The other machine is a Cache-Coherent Non-Uniform Memory Access (CC-NUMA) machine and consists of eight Intel Xeon CPU X5560 at 2.80 GHz processors and 16 GB of memory. The coarse-grained NACA 0012 Airfoil was used for the performance evaluations. The final mesh is comprised of 451,091 triangles. The speed up, defined as the ratio of the execution time of the fastest sequential algorithm (Triangle in the two-dimensional case) to the execution time of the parallel algorithm; and efficiency, defined as the ratio of speedup to the number of threads used, were measured for the application. The execution times that were measured do not take input and output times into consideration. Since the application uses Triangle for the boundary layer triangulation and inviscid region refinement, the application's running time using one thread is almost equivalent to the sequential execution time of Triangle, which has a loglinear complexity with respect to the number of triangles.

(a) 

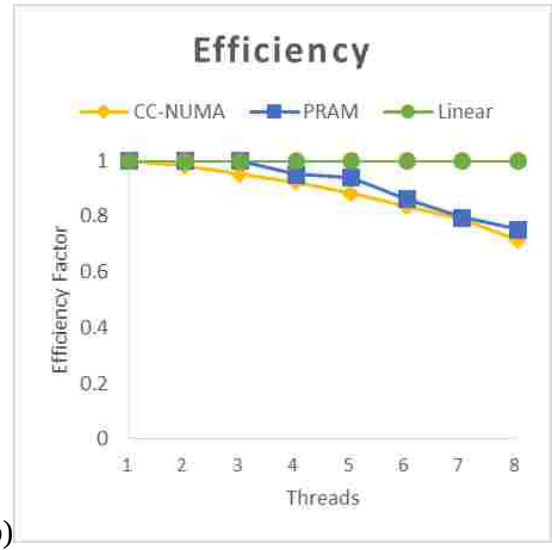
(b) 

Fig. 35. (a) Speedup data for our application compared to Triangle (b) Efficiency data for our application compared to Triangle

Our application performed favorable with an efficiency of approximately 90% for four threads, see Fig. 35b. The max speedup we achieved is a factor of six for eight threads on the PRAM architecture, see Fig. 35a. Typically for CC-NUMA architectures, parallel programs are slower than their equivalent PRAM architectures due to the overhead of maintaining cache coherence for shared memory [32].

CHAPTER 8

CONCLUSION

The framework and a practical implementation using modern C++ programming styles to generate high-quality, two-dimensional unstructured initial meshes comprised of the anisotropic, high-fidelity boundary layer and decoupled inviscid region in parallel for use in CFD simulations on shared-memory machines was presented and evaluated. The application is a push-button application, meaning that the user only needs to start the program by specifying the initial geometry, anisotropic gradation, and ray angle constraint, then momentarily waiting for the resulting mesh. Additionally, the application has two dependencies, the software Triangle and the POSIX Threads library, making the application as a whole, a lightweight and portable parallel mesh generator for aerospace applications with viscous flows. Using a high-fidelity mesh to begin the iterative CFD pipeline will yield a final, acceptable mesh in fewer iterations than an ill-suited initial mesh due to the anisotropic boundary layer and graded inviscid region. Constructing this initial mesh in parallel also eliminates an expensive and sequential bottleneck in the development process, yielding a pipeline better suited to consider Amdahl's law. Since observing that this approach is feasible for two-dimensional meshes, a plan to extend the approach to generate three-dimensional meshes in parallel is next. However, this approach is beneficial even in two-dimensional cases by providing a scalable, shared-memory parallel, push-button application to facilitate aerospace development with rapid turnaround time.

REFERENCES

[1] R.V. Garimella, M.S. Shephard, Boundary Layer Mesh Generation for Viscous Flow Simulations, International Journal for Numerical Methods in Engineering. 49 (2000) 193-218.
[2] G.E. Blelloch, G.L. Miller, D. Talmor, Developing a Practical Projection-Based Parallel Delaunay Algorithm, 12th Annual Symposium on Computational Geometry. (1996) 186-195.
[3] J.R. Shewchuk, Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator, Applied Computational Geometry: Towards Geometric Engineering. 1148 (1996) 203-222.
[4] A.M. Andrew, Another Efficient Algorithm for Convex Hulls in Two Dimensions, Information Processing Letters 9. (1979) 216-219.
[5] L. Linardakis, N. Chrisochoides, Graded Delaunay Decoupling Method for Parallel Guaranteed Quality Planar Mesh Generation, SIAM Journal on Scientific Computing. 30 (2008) 1875-1891.
[6] GCC, the GNU Compiler Collection, https://gcc.gnu.org.
[7] L.P. Chew, N. Chrisochoides, F. Sukup, Parallel Constrained Delaunay Meshing, Symposium on Trends in Unstructured Mesh Generation. (1997) 89-96.
[8] A. Loseille, D. Marcum, F. Alauzet, Alignement and Orthogonality in Anisotropic Metric-Based Mesh Adaption, 53rd AIAA Computational Fluid Dynamics Conference. (2015).
[9] R. Aubry, K. Karamete, E. Mestreau, D. Gayman, S. Dey, Ensuring a Smooth Transition from Semi-Structured Surface Boundary Layer Mesh to Fully Unstructured Anisotropic Surface Mesh, 53rd AIAA Computational Fluid Dynamics Conference. (2015).
[10] G. Zagaris, S. Pirzadeh, A. Chernikov, N. Chrisochoides, Parallel Mesh Generation For CFD Simulations of Complex Real-World Aerodynamic Problems, US National Congress on Computational Mechanics. (2007).
[11] G. Zagaris, S. Pirzadeh, N. Chrisochoides, A Framework for Parallel Unstructured Grid Generation for Practical Aerodynamic Simulations, 47th AIAA Aerospace Sciences Meeting. (2009).
[12] R. Zhang, K.P. Lam, Y. Zhang, Conformal and Adaptive Hexahedral-Dominant Mesh Generation for CFD Simulation of Architecture Applications, Winter Simulation Conference. (2011).
[13] Y. Ito, A. Shih, A. Erukala, B. Soni, A. Chernikov, N. Chrisochoides, K. Nakahashi, Parallel Unstructured Mesh Generation Using an Advancing Front Method, Mathematics and Computers in Simulation. 75 (2007) 200-209.
[14] G. Globisch, PARMESH – A Parallel Mesh Generator, Parallel Computing. 21 (1995) 509-524.
[15] N. Chrisochoides, D. Nave, Parallel Delaunay Mesh Generation Kernel, International Journal for Numerical Methods in Engineering. 58 (2003) 161-176.
[16] C. Kadow, Parallel Delaunay Refinement Mesh Generation, Ph.D. thesis, Carnegie Mellon University. (2004).

[17] XFOIL, Subsonic Airfoil Development System, http://web.mit.edu/drela/Public/web/xfoil.
[18] MSES, Multielement Airfoil Design/Analysis System, http://raphael.mit.edu/drela/msessum.ps.
[19] L. Lammera and M. Burghardt, Parallel Generation of Triangular and Quadrilateral Meshes, Advances in Engineering Software. 31 (2000) 929-936.
[20] A.I. Khan and B.H.V. Topping, Parallel Adaptive Mesh Generation, Computing Systems in Engineering. 2 (1991) 75-101.
[21] Daniels220. "AmdahlsLaw", English Wikipedia, Wikimedia Commons, http://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg
[22] D.L. Marcum, Advancing-Front/Local-Reconnection (AFLR) Unstructured Grid Generation, Computational Fluid Dynamics Review, World Scientific-Singapore. (1998) 140.
[23] J. Schoberl, NETGEN - An Advancing Front 2D/3D-Mesh Generator Based On Abstract Rules, Computing and Visualization in Science. 1 (1997) 41-52.
[24] L.P. Chew, Guaranteed-Quality Mesh Generation for Curved Surfaces, Proceedings of the Ninth Annual Symposium on Computational Geometry. (1993) 274–280.
[25] J. Ruppert, A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation, Journal of Algorithms. 18 (1995) 548–585.
[26] J.R. Shewchuk, Delaunay Refinement Mesh Generation, Ph.D. thesis, Carnegie Mellon University. (1997).
[27] X.Y. Li, S.H. Teng, and A. Ungor, Biting Ellipses to Generate Anisotropic Mesh, Eighth International Meshing Roundtable. (1999) 97–108.
[28] F.J. Bossen and P.S. Heckbert, A Pliant Method for Anisotropic Mesh Generation, Fifth International Meshing Roundtable. (1996) 63–74.
[29] J. Shewchuk, What is a Good Linear Finite Element? Interpolation, Conditioning, Anisotropy, and Quality Measures (preprint), University of California at Berkeley. (2002).
[30] "Amdahl's Law", English Wikipedia, Wikimedia Commons, https://en.wikipedia.org/wiki/Amdahl%27s_law
[31] "The Standard : Standard C++", International Organization for Standardization, https://isocpp.org/std/the-standard
[32] Z. Majo and T. R. Gross, Memory System Performance in a NUMA Multicore Multiprocessor, Proceedings of the 4th Annual International Conference on Systems and Storage. (2011).

VITA

Juliette Kelly Pardue

Department of Computer Science

Old Dominion University

Norfolk, VA 23529

EDUCATION

- B.S., 2014, Computer Science, Old Dominion University

CONFERENCES

- Parallel Two-Dimensional Unstructured Anisotropic Delaunay Mesh Generation for Aerospace Applications, Submitted to the 24th International Meshing Roundtable. (2015).

- A Scalable Parallel Arbitrary-Dimensional Image Distance Transform, Modeling, Simulation, and Visualization Student Capstone Conference. (2015).

- Scalability of a Parallel Arbitrary-Dimensional Image Distance Transform, Modeling, Simulation, and Visualization Student Capstone Conference. (2014).

WORKING EXPERIENCE

- Research Assistant, NIA/NASA (June 2014 – Present)

- Research Assistant, Old Dominion University (February 2012 – June 2014)

Typeset using Word.