Algorithm 995: An Efficient Parallel Anisotropic Delaunay Mesh Generator for Two-Dimensional Finite Element Analysis

JULIETTE PARDUE and ANDREY CHERNIKOV, Old Dominion University

A bottom-up approach to parallel anisotropic mesh generation is presented by building a mesh generator starting from the basic operations of vertex insertion and Delaunay triangles. Applications focusing on high-lift design or dynamic stall, or numerical methods and modeling test cases, still focus on two-dimensional domains. This automated parallel mesh generation approach can generate high-fidelity unstructured meshes with anisotropic boundary layers for use in the computational fluid dynamics field. The anisotropy requirement adds a level of complexity to a parallel meshing algorithm by making computation depend on the local alignment of elements, which in turn is dictated by geometric boundaries and the density functions—one-dimensional spacing functions generated from an exponential distribution. This approach yields computational savings in mesh generation and flow solution through well-shaped anisotropic triangles instead of isotropic triangles. The validity of the meshes is shown through solution characteristic comparisons to verified reference solutions. A 79% parallel weak scaling efficiency on 1,024 distributed memory nodes, and a 72% parallel efficiency over the fastest sequential isotropic mesh generator on 512 distributed memory nodes, is shown through numerical experiments.

CCS Concepts: • **Mathematics of computing** → **Mathematical software**; • **Applied computing** → **Physics**; *Aerospace*;

Additional Key Words and Phrases: Anisotropic mesh generation, parallel algorithms, finite element analysis, boundary layer, computational geometry

ACM Reference format:

Juliette Pardue and Andrey Chernikov. 2019. Algorithm 995: An Efficient Parallel Anisotropic Delaunay Mesh Generator for Two-Dimensional Finite Element Analysis. *ACM Trans. Math. Softw.* 45, 3, Article 33 (July 2019), 30 pages.

<https://doi.org/10.1145/3301321>

1 INTRODUCTION

Mesh generation is a step in the iterative pipeline for finite element analysis (FEA) as shown in Figure 1. After an analysis of the partial differential equations (PDE) solution on the mesh, the mesh is adapted to yield a more favorable error estimation, which is typically a faster operation

This work was supported in part by the National Institute of Aerospace (NIA; <http://www.nianet.org>) and National Science Foundation (NSF) grant CCF-1439079. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIA or the NSF.

Authors' address: J. Pardue and A. Chernikov, Department of Computer Science, Old Dominion University, 4700 Elkhorn Ave, Norfolk, VA 23529; emails: {jpardue, achernik}@cs.odu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

0098-3500/2019/07-ART33 \$15.00

<https://doi.org/10.1145/3301321>

Fig. 1. Development pipeline.

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 adaptation. The adaptation 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, the mesh may be adapted to provide a more accurate solution. Assuming proper care for the adaptation steps, this new mesh will be more accurate in the desired regions with respect to the PDE solution. This means that with each adaptation step, the next iteration of the mesh will have a smaller error estimate than the previous mesh. So at each iteration, the overall adaptation work required to reach a highly accurate solution decreases since the mesh's accuracy always improves.

Since this work is funded in pursuit of aerospace research, examples in this work will focus on computational fluid dynamics (CFD), which is just one example of what FEA is capable of. However, this approach is applicable for any physical phenomena that are anisotropic or semistructured in nature. When executing any pipeline of tasks, it is critical to consider Amdahl's law, which states that the speedup of a program is limited by the sequential fraction of the program. The NASA CFD 2030 Vision (Slotnick et al. 2014) has identified that mesh generation and adaptivity are significant bottlenecks in the CFD pipeline. CFD is just one example of what FEA is capable of. Hardware advances have allowed for higher-resolution simulations, and although flow solver software has matured at a faster rate, mesh generation has become increasingly difficult due to the demand for larger meshes over more complex geometries. The NASA CFD 2030 Vision states that mesh generation routinely requires human intervention due to a lack of robustness in treating complex geometries. One of the key findings is a lack of automation, and inherently including robustness, in the mesh generation process: "A single engineer/scientist must be able to conceive, create, analyze, and interpret a large ensemble of related simulations in a time-critical period, without individually managing each simulation, to a pre-specified level of accuracy." Clearly, a mesh generator is needed that can be parameterized, executed, and run (without any human interaction during runtime) via a script to submit a batch collection of related jobs. Chawner et al. (2015a) discuss in more detail the geometry and meshing concerns in the NASA CFD 2030 Vision along with a history of three decades of meshing evolution, which identifies the common unaddressed capabilities over the years. Human intervention has always been a problem with mesh generation, and automation has been a request since 1984. In addition to robustness still being requested in the mid-1990s, more efficient mesh generators were also desired because the time to generate a series of related grids was prohibitively too long. Chawner et al. also identify the desire for meshes over more complex geometries along with physics-coupled meshing, such as generating an anisotropic boundary layer for Reynolds-averaged Navier-Stokes flows. Different mesh generation codes use different quality metrics to determine unacceptable elements, and different solver codes are more sensitive to certain quality criteria. To summarize, mesh generators need to be able to produce large "high-quality" grids quickly around complex geometries without any human intervention, where "high-quality" is dependent on the flow solver's discretization. When executing a pipeline of task, it is critical to consider Amdahl's law, which states that the speedup of a program is limited

by the sequential fraction of the program. Mesh generation is the sequential portion of the FEA pipeline, so the entire simulation pipeline will not be scalable until the mesh generation task is scalable.

The summary of the panel discussion at AIAA's 2015 SciTech Conference by Chawner et al. (2015b) further discusses the aforementioned needs for mesh generators, including mesh generation's utilization of High Performance Computing (HPC) advances. The shortcomings in robustness and the inability for mesh generators to create a valid mesh around complex configurations has become a roadblock for parallel mesh generation. If a mesh generator fails to produce a valid mesh 10% of the time, then attempting to run the mesh generator in parallel with 10 processes will typically lead to a failure. Due to a lack of advancement in fault tolerance for HPC, if even just one meshing process fails, then all processes abort and a mesh is not produced. Underutilization of HPC resources is due to the scalability and robustness of certain mesh generation approaches, such as the advancing-front method, which requires a lot of synchronization to ensure that multiple processes do not create meshes that overlap. In addition to the poor parallel scalability, the advancing-front method is based on heuristics and is not guaranteed to terminate, so this approach is not a good candidate to parallelize. It is clear that the priority for mesh generators has become automation, which requires robustness.

Several applications that model complex physical phenomena such as shocks, boundary layers, and vortices require fast construction of very large two-dimensional meshes. There are also applications, such as the direct numerical simulation of turbulence in cylinder flows with very large Reynolds numbers (Dong and Karniadakis 2005), which use two-dimensional meshes for three-dimensional simulations. The mesh is generated in the xy -plane and then the planar elements are extruded in the z -direction for the cylinder flows to create volume elements. As the Reynolds number increases, the size of the mesh grows in the order of $Re^{9/4}$ (Karniadakis and Orszag 1993). For these simulations with very large Reynolds numbers, a two-dimensional parallel distributed-memory mesh generator is crucial to eliminate the sequential bottleneck of mesh generation and significantly reduce the overall time to solution.

Since the end goal is to generate a mesh that 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. One iteration through the pipeline is one call to a meshing application, one call to the flow solver with the mesh, and one analysis performed on the flow solution by an error estimator to determine if the solution is sufficiently resolved. 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 that require refinement, or the solution may be so poor that the error estimator cannot determine regions that need refinement. The algorithm in this article generates the initial mesh used to start the first iteration of the pipeline. The initial mesh is generated from a series of vertices and line segments defining the input geometry. The boundary layer growth function is specified, and the distance to the farfield is specified. An input mesh of the outer boundary is not needed for the application. The application described in this article does not perform mesh adaptation, but there are situations where a different, more accurate boundary layer growth function is desired. This application makes it easy for the boundary layer growth function to be modified to produce a new initial mesh with more resolution. Design optimization also requires initial meshes for many related geometries that have small differences.

This article is an extension of our previous work (Pardue and Chernikov 2016). In this work, we resolve the scalability issues associated with the load balancing paradigm. Previously, processes would experience long idle times while waiting for work, causing a drop in performance when

Fig. 2. The 30p30n airfoil consisting of three components.

many processes are low on work. We included a Laplacian normal direction vector smoothing operation to resolve some cases of boundary layer intersections without sacrificing resolution and without drastically disturbing the orthogonality and alignment of the anisotropic triangles. The Laplacian smoothing operation is also preferred where there is a region with multiple discontinuities in the slope of the geometry's surface. We evaluated our application on a larger machine and also performed a weak scaling study up to 1,024 processes. We obtained a new geometry configuration—the circulation control airfoil. Although the 30p30n airfoil (Figure 2) in the previous work had some complex regions, the circulation control airfoil features additional regions of complexity that were not seen in the 30p30n airfoil. We improved the performance of detecting intersections in the boundary layer and improved the quality of how these intersections are resolved. We extended the gradation control to remove potential triangles that are stretched normal to the surface instead of along the surface. The gradation control adds and removes vertices from the boundary layer so that the outer layer of the boundary layer contains isotropic triangles.

The remaining sections of the article review recent and related mesh generation work and further discuss the motivation for parallel mesh generators in Section 2. The majority of this article details the parallel mesh generation algorithm in Section 3 and its implementation in Section 4. Section 5 shows the mesh quality by comparing solution characteristics to a verified reference solution. The remainder of the article discusses the performance of the parallel mesh generation algorithm compared to the fastest sequential mesh generation algorithm and a commercial mesh generation application in Section 6 and the robustness of the algorithm for a complex input model in Section 7. The article concludes with a discussion on how this algorithm will be extended to handle three-dimensional domains in Section 8 and a summary of the impact of this work in Section 9.

2 MOTIVATION

Current parallel mesh generators, which handle the isotropic cases (Chernikov and Chrisochoides 2006, 2008a, 2008b; Chrisochoides and Nave 2003; Foteinos and Chrisochoides 2014; Globisch 1995; Ito et al. 2007; Kadow 2004; Khan and Topping 1991; Lämmer and Burghardt 2000; Linardakis and Chrisochoides 2008), do not perform well for parallel anisotropic mesh generation. Isotropic mesh generators, which focus on solution-based adaptation create many unnecessary elements where flow velocities change rapidly in one direction. These anisotropic gradations in the flow velocities require anisotropic elements, typically with a 10,000:1 aspect ratio of stretching normal to the flow versus stretching in the direction of flow. A quadrilateral element with this aspect ratio would have a width that is 10,000 times the height of the quadrilateral. Representing these regions with isotropic elements incurs a multiple orders of magnitude 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.

With the fast-developing field of CFD, a new mesh type has been introduced—graded anisotropic meshes. The most anisotropic elements are along the surface of the input. These stretched triangles have an aspect ratio near 10,000:1. Each additional layer away from the surface is less anisotropic. The triangles on the outer layer of the boundary layer should be roughly isotropic or have an aspect ratio close to 1:1. There is a smooth decrease of the aspect ratios from the surface layer to the outer layer. As stated before, these graded anisotropic meshes aim to decrease the computational efforts of the PDE solvers, as well as decrease the number of elements in the mesh. Others (Marcum and Alauzet 2015; Pirzadeh 1996; Zhang et al. 2011) have developed sequential anisotropic mesh generation paradigms to facilitate these CFD simulations with viscous flows. Serial two-dimensional tools specifically for aerospace application development exist, such as MSES (Drela 1994), which also caters toward airfoil development. Another approach is to use level-set techniques (Mavriplis 1990; Peraire and Morgan 1997). Level-set techniques inflate the surface of the input model to create a new outer layer. For all vertices on the outer layer, the distance to the closest vertex is constant. The space between the surface and the outer layer is triangulated. Then a new layer is inflated from the outer layer and the process repeats until the boundary layer is completely discretized. The drawback of using level-set techniques is that the approach is difficult to parallelize because all processes must synchronize at each layer. For the approach in this article, a single level set is calculated and used as the outer layer of the boundary layer. This has a similar effect to level-set methods, but the need to synchronize at each layer is eliminated. In this work, a parallel algorithm is presented that

- generates a high-fidelity, anisotropic boundary layer mesh from a user-defined growth function;
- generates a globally Delaunay, graded, isotropic mesh in parallel;
- resolves potential interpolation errors in the boundary layer caused by the local mesh density;
- resolves self-intersections and multi-element intersections in the anisotropic boundary layer;
- is a push-button mesh generator, of which no human interaction is required after startup; and
- is scalable and efficient.

3 ALGORITHM

Since the focus of this work is aerospace research, the figures and examples will be airfoils—two-dimensional slices of wings. Many applications require meshes that are composed of two different mesh types, a semi-structured anisotropic boundary layer and an unstructured isotropic mesh region; two separate paradigms are needed to generate high-fidelity initial meshes that are computationally efficient for PDE solvers. The semi-structured anisotropic boundary layer is generated through an extrusion-based advancing-front method, as presented by Aubry et al. (2015), whereas the unstructured isotropic mesh region is generated using a graded decoupled approach along with Delaunay refinement. For the anisotropic vertex insertion method, we can take advantage of the semi-structured pattern to exploit parallelism with little communication. The boundary layer is triangulated in parallel through a domain decomposition approach, whereas the isotropic mesh region is refined concurrently through a decoupling approach that creates fixed borders between subdomains. Figure 3 shows the high-level steps to the algorithm, along with relevant Message Passing Interface (MPI) and threading operations. This figure will be referenced throughout the following sections to provide context to the overall approach for each step. The algorithm is broken into four sections: input, refinement, meshing, and output. Each process needs to read

#	Operation	Parallelism	MPI / Pthread Commands
Input Stage			
1	Read input geometry & meshing parameters	Sequential with all	
2	Surface Analysis	Parallel	
3	Gather surface normals at root	Sequential at root / Parallel with others	MPI_Gather at root
Refinement Stage			
4	Smooth local self intersections and around regions of high curvature	Sequential with root	
5	Remove points that will create triangles stretched normal to the flow	Sequential with root	
6	Clip rays to resolved non-local intersections	Sequential with root	
7	Determine number of points needed to transition to isotropic triangles	Sequential with root	
8	Add fans of rays at sharp trailing edges	Sequential with root	
Meshing Stage			
9	Distribute finalized, valid rays to all processes	Sequential at root / Parallel with others	MPI_Scatter from root
10	Create boundary layer vertices	Parallel	
11	Gather boundary layer vertices at root	Sequential at root / Parallel with others	MPI_Gather at root
12	Root creates initial boundary layer subdomain and initial isotropic mesh region subdomains	Sequential with root	
13	Launch communicator thread	Sequential with all	pthread_create
14	All processes decompose/decouple subdomains and send to other processes that need work	Parallel	MPI_Send / MPI_Recv
15	All processes triangulate/refine subdomains while communicator/manager thread monitors progress	Parallel	
16	Join communicator thread	Sequential with all	pthread_join
Output Stage			
17	Send subdomain meshes back to root	Sequential at root / Parallel with others	MPI_Recv (root) / MPI_Send (others)
18	Output mesh	Sequential with root	

Fig. 3. Steps of the algorithm with the parallel and sequential components identified along with the MPI and Pthread commands.

the input geometry, boundary layer growth function parameters, and the isotropic mesh region parameters.

3.1 Boundary Layer

Physical phenomena such as boundary layers in fluid mechanics are anisotropic in nature. The flow velocities change rapidly along the normal of the surface, opposed to along the surface. It is beneficial to discretize the mesh in a way that efficiently captures these flow velocities. This yields substantial computational savings without compromising accuracy. This dictates that the mesh should be refined in the direction normal to the surface, as shown in Figure 4, where these strong gradients exist. No vertices are inserted on the surface, because it is assumed that the surface is sufficiently resolved. These characteristics allow for the extrusion-based vertex insertion along the normal of the surface at each vertex on the planar straight-line graph (PSLG). A PSLG is composed of vertices and linear segments connecting them. This information may have been exported from a computer-aided design software package. Each black vertex in Figure 4 is treated as an endpoint for a ray, whereas the normal at the vertex is treated as the direction of the ray. The rays have a finite length to start with, based on the boundary layer growth function. Vertices of a ray may be

Fig. 4. NACA 0012 airfoil with surface vertices and normals. The leading edge is at the far left side and the trailing edge is at the far right side, and both are shaded darker.

removed or additional vertices may be added based on criteria discussed in future sections. Each vertex is incident on two edges. The vertex normal is the average of the normals of these two edges. New vertices are then inserted along the ray according to a growth function. The tip of the normal vector is where the outer layer vertex will be placed. The total height of the boundary layer is calculated from the growth function. There are multiple functions, as presented by Garimella and Shephard (2000), which can be used to space the prospective vertices. 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 exponential. These growth functions insert vertices along the normal of the PSLG. Vertices are packed close to the surface and become more spaced out the farther they are away from the surface. Each ray has the same growth function and spacing along its normal. However, other more sophisticated, adaptive growth functions (Garimella and Shephard 2000) may be necessary for more complex geometries. For simplicity, we use an exponential growth function. The code can be modified to input a custom growth function. The growth function is a one-dimensional spacing function representing the distance each vertex should be placed along the normal vector.

After the input geometry is read, the root process analyzes the surface to detect where there are sharp corners in the geometry (see step 2 in Figure 3). The surface normal at each vertex is computed using the two neighboring vertices. This surface normal is used to create the initial ray at each normal. This is a quick operation because it only requires the input to be scanned once. If needed, each process can analyze a subset of the input since the normal vectors and corners are locally determined at a vertex using the two neighboring vertices.

Clearly, larger angles will naturally occur where the slope changes rapidly, such as the leading edge shown in Figure 4, and where the slope has discontinuities, such as the trailing edge in Figure 4, which causes these extremely large angles around cusps, affecting the density of the mesh in the corresponding area. The regions where these large angles occur are the areas of the mesh that need refinement to satisfy the resolution constraints of the boundary layer mesh region. These regions that have a singularity in the geometry also have a singularity in the solution. Additional triangles will be required where there are singularities to adequately resolve solution features. The solution changes rapidly in other regions where there is a high degree of curvature for the input model. Refinement is needed in these regions to ensure that there is a smooth change in area between neighboring triangles. These refinement procedures are discussed in detail in the following section.

3.2 Convex Corners

Since the boundary conditions affect the entire solution over the mesh, it is critical that the boundary layer be properly discretized. Figure 5 shows the poorly shaped triangles at a sharp corner, which causes discretization errors when the PDE solution is computed. Previously in Pardue and

Chernikov (2016), large angles that were not at a sharp trailing edge were treated by adding new vertices that are uniformly spaced between the two vertices that have a large angle between their normals. Linear interpolation between the two original normals was used to compute the direction of the new normals. However, due to the explicit representation of the PSLG, which uses linear edges, the program only has information about the location of vertices and the linear segments that connect the vertices, so the location of the new vertices cannot be accurately determined because the program does not have any higher-order information about the curvature of the input geometry. For this reason, and to keep the integrity of the user's input, no new vertices are added to the input model. The current version of the application maintains complete boundary conformity to the input model. If a user wants more resolution in a certain region of the input model, then the user should modify the input model, as this is not a responsibility of the mesh generator. Instead, Laplacian smoothing of the normal vectors is used in these regions where there are large angles (see step 3 in Figure 3). This is also beneficial because it provides more uniform coverage of the boundary layer domain without adding additional vertices. Laplacian smoothing is an iterative procedure where new directions are computed for normal vectors based on local information. The aim of Laplacian smoothing is to reduce deviation between neighboring normal vectors so that the outer layer of the boundary layer is uniformly discretized. At each iteration, a normal vector is calculated by averaging its two neighboring normal vectors with itself. This is done once for every normal vector in a set. At the end of each iteration, the absolute difference between the new normal vectors and their corresponding previous normal vectors is calculated as the residual. Once the residual is below a certain threshold, typically 10^{-8} , the set of normal vectors is at their optimal position. The length of the normal vectors is preserved during the Laplacian smoothing operation. Laplacian smoothing is done before the vertices are inserted because the vertices need to be placed along the normal vector. The smoothing operation is performed sequentially because it is quick and usually terminates after a couple dozen iterations. There are two options to parallelize this step, if needed. Independent, non-overlapping regions of the mesh can be smoothed concurrently because they will not interfere with each other. Additionally, multiple processes can smooth a large region by dividing it into contiguous subregions. Each subregion has its rays smoothed concurrently, and some synchronization is performed at the end of each iteration to update the overlapping ghost rays that exist on the shared sections between subregions.

Cusps at sharp trailing edges are handled by using multiple normals at the cusp vertex (see step 7 in Figure 3). A fan of rays is emitted from the cusp vertex so that the outer surface has isotropic triangles. The number of additional rays is determined by the total height of the boundary layer and also by the angle between surface normals at the cusp vertex. The direction of the rays is determined using linear interpolation. After the rays have been computed for each of the vertices of the PSLG, the angle between the current ray and the forward neighboring vertex's ray is computed. The vertices of the PSLG are stored in counter-clockwise order, so a vertex's forward neighbor is the vertex that comes after itself in the counter-clockwise ordering. If the angle is too large or there is a cusp at the location, then the aforementioned approach of Laplacian smoothing of the normal vectors of the rays is employed. Figure 5 shows two large angles easily visible at the trailing edge of the leading slat of the 30p30n airfoil after the ray-based vertex insertion, and Figure 6 includes the fan of augmented vertices determined by the large angle detection. For regions where there is a blunt trailing edge, Laplacian smoothing is used in the neighborhood of the blunt trailing edge to smooth the direction of the normal vectors. Figure 7 shows the region of the blunt trailing edge of the trailing slat of the 30p30n airfoil before Laplacian smoothing, and Figure 8 shows the result of the smoothing operation. The smoothing operation described in Garimella (1998) causes adjacent triangles on neighboring rays to have similar cell sizes and provides better coverage by increasing triangle density where there are singularities in the airfoil's surface. Two parameters

Fig. 5. Poorly sized triangles at the trailing edge of the leading slat of the 30p30n airfoil caused by a discontinuity in the slope of the surface.

Fig. 6. Mesh at the trailing edge of the leading slat of the 30p30n airfoil after a fan of rays is added.

Fig. 7. Poorly sized triangles at the blunt trailing edge of the trailing slat of the 30p30n airfoil caused by discontinuities in the surface's slope.

Fig. 8. Mesh at the blunt trailing edge on the trailing slat of the 30p30n airfoil after Laplacian normal direction vector smoothing is applied.

control whether a convex corner is classified as a sharp or blunt trailing edge. A vertex is on a sharp trailing edge if its normal vector has at least a 60-degree angle with its forward neighboring vertex's normal vector and at least a 60-degree angle with its previous neighboring vertex's normal vector. A vertex is on a blunt trailing edge for the same conditions but with 30-degree angles each. The operation of adding fans of normals at sharp trailing edges is the last step of the refinement stage, because it is not known if this fan will cause an intersection with another region of the mesh.

3.3 Mesh Validity

Once the new rays have been determined, each element of the airfoil needs to be checked for self-intersections. A ray may intersect a different ray, the input surface, or the outer layer of the boundary layer. Each ray is used to create a line segment where the endpoints of the line segment

Fig. 9. AABBs for a multi-element airfoil with three elements.

are the ray's surface vertex and outer layer vertex. A single element is a connected set of cyclical vertices defining a closed polygon. Figure 9 shows a multi-element airfoil with three elements. For multi-element airfoils, each element's rays need to be checked for intersections with the enclosing border of the boundary layer for all other elements and the surface of all other elements. A hierarchical approach to intersection checks is performed for multi-element intersections. The hierarchical approach reduces the search space incrementally by using the least computationally expensive pruning operations first instead of performing a brute-force, exhaustive search by comparing every ray and segment to every other ray and segment to check for intersections. For multi-element intersection checks, the axis-aligned bounding box (AABB) (Foley et al. 1994) of each element is first checked to see if it intersects with another element's AABB. An AABB is a rectangle with edges parallel to the x -axis or y -axis. If the AABBs do not overlap, such as the AABBs for elements one and three in Figure 9, then no intersections will occur between the rays and segments of these two elements. When the AABBs do intersect, candidate rays are pruned by whether or not they intersect the AABB of another element's boundary layer. A modified version of the Cohen-Sutherland algorithm for polygon clipping (Foley et al. 1994) is used to quickly and efficiently check for AABB intersections. These AABB intersection checks do not need to be performed for the self-intersection checks because an element's AABB contains all of the element's rays, so no rays would be removed from the search space. This step is not done in parallel because there is a small number of airfoil elements. For the 30p30n airfoil, which is a more complex airfoil, there are only three airfoil elements. There is not enough work to do these tasks in parallel, and the cost of performing these operations sequentially is negligible. At this point, we have significantly reduced the search space for the intersection checks using AABBs. For Figure 9, potential intersections have been identified in the shared region between element one and element two, and the shared region between element two and element three. Next, a new series of tests using a search tree will be performed for the identified candidate regions to further reduce the search space. The search tree identifies pairs of potentially intersecting line segments. These pairs are then checked to ensure that the intersection actually occurs. Intersections are resolved differently depending on the types of line segments that contribute to an intersection. A line segment may be part of the input surface, a segment created from a ray, or part of the outer layer border of the boundary layer. If intersections occur between ray line segments that are neighboring each other, then the rays' normal vectors have their directions smoothed to mitigate the intersection. All other intersections are resolved by reducing the height of the boundary layer where the intersections occurred until the intersection no longer exists. The search tree and more details about how these intersections are resolved are discussed in the remainder of this section.

The next set of tests use an alternating digital tree (ADT) (Bonet and Peraire 1991) and are performed for both the self-intersection checks and on the candidate rays from the AABB multi-element intersection checks. The ADT works by mapping a d -dimensional space to a $(d + d)$ -dimensional half-space. The one-dimensional analog is shown in Figure 10. In two dimensions, the ADT is used with a search algorithm that determines if a four-dimensional vertex lies within a particular four-dimensional space subregion. The ADT is used to efficiently determine if a line segment's AABB overlaps with any of the other line segments' AABBs. This is done by projecting

Fig. 10. One-dimensional analog for the ADT, showing the mapping of the one-dimensional extent box to two-dimensional space.

line segments' extent boxes to four-dimensional vertices; a line segment's extent box (as a four-dimensional vertex) can be tested to see if it intersects with any of the n line segments' extent boxes (also as four-dimensional vertices) in $O(\log n)$ time. The candidate rays that intersect the AABB then have their extent boxes projected to four-dimensional vertices. The extent boxes of the other element's enclosing border segments of its boundary layer are also projected to four-dimensional vertices and then stored in an ADT. Each candidate ray's extent box is queried against the ADT to determine if a ray's extent box intersects an extent box of the boundary layer's outer border. The approach is the same for the self-intersection checks where the candidate rays are all of the rays for an element, and the ADT contains the surface of the element and the enclosing border of the element's boundary layer. For the set of ray segments to be tested, the non-surface vertex is set to be a little farther than the vertex on the outer border of the boundary layer. The additional distance is equal to the outer layer thickness to account for the situation where two walls may be extruding toward each other and come close to meeting but do not cross over. This ensures that an isotropic triangle can be placed between two fronts that are near each other. Inserting a vertex too close to another front would cause sliver triangles, or triangles that are nearly flat, to be created for the isotropic mesh region inside the small gap (Figure 11). Figure 12 shows the resulting boundary layer mesh and isotropic mesh region after a layer was removed from the boundary layer to avoid sliver triangles in the isotropic mesh region.

The identified extent box intersections do not guarantee that a ray will intersect another element's boundary layer, but it significantly and efficiently reduces the search space. Intersection checks using computational geometry are then performed for each identified ray and border segment pair. If there is an intersection, then the intersection vertex between the segments is calculated. For self-intersections, if there are a set of intersections in a contiguous region, then Laplacian smoothing is used in these regions to smooth the normal direction of the rays to resolve the intersections. Figure 13 shows the self-intersections for a right-angle concave corner on the lower side of the main slat of the 30p30n airfoil, and Figure 14 shows the resulting mesh after the smoothing operation. The Laplacian smoothing operation is desirable for these important feature regions because it does not affect the overall triangle density of the region and preserves the semi-structured pattern instead of clipping the rays and introducing unstructured elements close to the surface. The smoothing operation also preserves the orthogonality, alignment, and stretching of the triangles. The rays that are in the regions where local self-intersections exist are added to the set of rays to smooth (see step 3 in Figure 3). Laplacian smoothing of the normal direction vectors will not work for multi-element intersections because the normal direction is not the cause of the intersection. For example, an intersection can occur between two circles that are placed close together. The normal direction vectors are optimal with respect to their neighboring normal direction vectors because the surface of the circle is smooth. The intersection occurs because the thickness of

Fig. 11. Shaded sliver triangles in the isotropic mesh region because two walls of the boundary layer are too close.

Fig. 12. Shaded well-shaped triangles in the isotropic mesh region after the two walls of the boundary layer had one fewer layers inserted.

Fig. 13. Intersections for a right-angle concavity of the main slat of the 30p30n airfoil.

Fig. 14. Resolved intersections by Laplacian normal direction vector smoothing for a right-angle concavity of the main slat of the 30p30n airfoil.

the boundary layer is too large in these regions, causing an overlap. The same principle applies for non-local self-intersections that occur between two different regions of an element's boundary layer. These intersections are treated by decreasing the number of vertices that will be placed along the normal vector so that no vertex is placed past the intersection vertex. Figure 15 shows this case where one region's rays intersect another region's outer border. Figure 16 shows the resulting anisotropic and isotropic meshes after the intersections have been resolved.

3.4 Boundary Layer Vertex Insertion

At this point, the first set of mesh intersection tests have been completed but no vertices have actually been created yet. Instead, the rays keep track of how many vertices it will contribute to the final boundary layer mesh. When the vertices are created, they will be created with respect to the user-defined boundary layer growth function. The growth function is composed of three parts: the thickness of the first layer, the growth rate, and the number of layers. Each layer is thicker than the previous layer by the percentage of the growth rate. The user's input for the number

Fig. 15. Identified multi-element intersections between the main slat's outer boundary and the rays for the trailing slat of the 30p30n airfoil.

Fig. 16. Resolved multi-element intersections in the 30p30n airfoil by clipping far enough away from the intersections to avoid creating sliver triangles.

of layers is used as the initial number of vertices that each ray will create. The original number of layers for a given ray may have been decreased to resolve an intersection. Now the rays will be checked to make sure the outer layer triangles are isotropic. This provides a smooth transition to the graded isotropic mesh region. The criteria for a triangle to be isotropic is having an aspect ratio between 0.55 and 1.5, but these parameters can be modified. It is to be noted that the triangulation is not constructed at this time, but the shape and size of the triangle can be determined because we know where the vertices will be located based on the ray and boundary layer growth function. If an outer layer triangle is not isotropic, then the number of layers, or vertices along the ray, will be increased or decreased, depending on the direction that the triangle is stretched. If a triangle is stretched more in the direction normal to the surface, then the number of layers is decreased. Figure 17 shows the resulting mesh if poorly stretched triangles in the black boundary layer were not removed. This causes a poor transition into the isotropic mesh region due to the rapid change in cell size and aspect ratio. Figure 18 shows the resulting mesh after poorly shaped triangles are removed with the gradation control operation. If the triangle is stretched more in the direction along the surface, then the number of layers is increased. If a ray previously had an intersection, which has been resolved at this stage, then the ray will be disallowed from increasing the number of layers because it is already known that this will cause an intersection. Additionally, if a ray is requesting to increase its number of layers, then the ADT that was previously constructed during the intersection testing stage is queried. Figure 19 shows varying heights for the boundary layer where additional layers have been added, or where some layers have been removed. All of the triangles on the outer layer of the boundary layer are roughly isotropic, which will provide a smooth transition to the isotropic mesh region. All of the vertices are created in one shot because all of the rays are valid and non-intersecting, so no additional checks need to be performed. Each process will compute the vertices along their set of rays with respect to the user-defined growth function. Vertices are inserted based on the number of layers requested by the given ray. This number of layers may be the original number of layers that a user requested. The number of layers may be fewer if an intersection is resolved, so vertices are only inserted up until the intersection vertex that was determined during the intersection tests. The number of vertices along a ray may also have been changed to make sure that the outer layer triangles will be isotropic. The vertices are then gathered at the root process to create the boundary layer subdomains.

Fig. 17. Black boundary layer triangles are stretched along the normal, creating small sliver triangles and irregular jumps in cell size when transitioning to the highlighted unstructured isotropic triangles.

Fig. 18. Black boundary layer triangles after the number of layers were locally decreased, which removed the poorly shaped triangles, providing a smooth transition to the highlighted unstructured isotropic triangles.

Fig. 19. Main slat of the 30p30n airfoil showing different heights for the boundary.

3.5 Parallel Triangulation of Boundary Layer

After the vertex insertion step is complete, the vertices in the boundary layer need to be triangulated. The algorithm presented by Blelloch et al. (1996) is used, which utilizes the duality between the two-dimensional Delaunay triangulation and the three-dimensional lower convex hull of a paraboloid (Figure 20(a)). The algorithm splits one subdomain into two independent subdomains that can be triangulated concurrently. This algorithm was chosen because it is recursive, so subdomains can be repeatedly split to generate enough work so that each process has a subdomain.

Delaunay was chosen as the triangulation algorithm because the Delaunay property is always maintained using this method. Delaunay is a useful property to maintain if the algorithm is later extended with early Delaunay decoupling for additional scalability. The implicit ordering and generation of triangles may not be Delaunay. For regions where there are different boundary layer growth functions, a different spacing of vertices along the rays will be used. An implicit ordering and generation of triangles where there are different growth functions may produce ill-suited and incorrectly skewed triangles. Additionally, Delaunay can better handle the irregularities in narrow concavities where some rays may require a fewer number of layers to avoid an intersection, causing a different number of layers between neighboring rays. The algorithm works by dividing

Fig. 20. (a) Paraboloid and lower convex hull of paraboloid with corresponding 2D Delaunay triangulation for a sample vertex set; (b) Flattened paraboloid projection of the vertex set and lower convex hull with corresponding 2D vertex set and dividing Delaunay path for a sample vertex set.

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. Kadow (2004) 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. This approach was chosen because the dividing path created between subdomains corresponds to constraining edges that would be present in the final triangulation if the domain were triangulated sequentially without being decomposed. Other algorithms (Chew et al. 1997) that use user-defined dividing paths to arbitrarily partition the domain are undesirable, because these user-defined dividing paths may not have been present in the final triangulation and will disturb the alignment and orthogonality of the anisotropic elements. The median line, for efficiency and simplicity of the algorithm, is parallel to the x -axis or y -axis, known as the cut axis. The lower convex hull is calculated by sweeping in the direction of the cut axis. The vertices do not lie directly on 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 (Shewchuk 1996).

Our algorithm starts by creating an initial subdomain that contains all of the boundary layer vertices. This initial subdomain will be split into two smaller subdomains. In the implementation, each disconnected region of the input is treated as a single subdomain initially. For Figure 9, before any decomposition happens, there would be three initial subdomains, one for each element of the airfoil. A subdomain stores vertices in x -sorted order and a copy in y -sorted order. Not much memory is saved if there is only one copy of the vertices for the x -sorted ordering and a container of pointers for the y -sorted ordering. The algorithm also performs slowly for iterating over pointers to vertices for the y -sorted ordering because of cache inefficiency. This allows for the AABB to be computed in constant time using the first and last vertex of the x -sorted and y -sorted vertices. If the length in the x -direction of the AABB is shorter than the length in the y -direction, then the cut axis is set to be the x -axis. This avoids 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 vertices along the cut axis

Fig. 21. Steps of the monotone chain algorithm. The vertical dashed line sweeps from the lowest x -coordinate vertex to the 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 vertex of the lower convex hull makes a right-hand turn, so the next to last vertex must be removed because it is not part of the lower convex hull. (d) The current lower convex hull after the non-hull vertex is removed.

Fig. 22. The boundary layer of the 30p30n airfoil decomposed into 128 Delaunay independent subdomains.

are projected onto a paraboloid centered at the median vertex and then flattened onto the vertical plane perpendicular to the cut axis (Figure 20(b)). The vertices that have been flattened onto the vertical plane are then used to compute the lower convex hull in worst-case linear time using the monotone chain algorithm (Andrew 1979) (Figure 21 shows the steps of the algorithm). The algorithm works by incrementally constructing the lower convex hull from a coordinate-sorted set of vertices by adding one vertex at a time and removing a vertex if it makes a right-hand turn. Since the vertices were in sorted order before the projection, the vertices will be in sorted order after the projection and flattening. The original vertices that correspond to the vertices on the lower convex hull are then used to create new edges.

Two new subdomains are then created, and the original subdomain's vertices are then partitioned into the two new subdomains' vertices. For simplicity and without loss of generality, assume the cut axis is the y -axis. All vertices in the original subdomain that have an x -coordinate less than the median vertex's x -coordinate are added to the left subdomain, whereas vertices with x -coordinates greater than the median vertex's x -coordinate are added to the right subdomain. Additionally, vertices comprising the lower convex hull are added to both the left and right subdomains' vertices. This partitioning step is done by iterating over the original subdomain's x -sorted and y -sorted vertices to maintain the sorted vertices in linear time. These new subdomains are then sent to another process for further decomposing until all processes have a subdomain. The total complexity for one decomposition step is $O(n)$ because the median is picked in constant time, and the remaining steps of projecting the vertices, computing the lower convex hull, and partitioning the vertices to two separate subdomains each require $O(n)$ time.

Once a subdomain has been sufficiently decomposed, the enclosing border of edges is determined and then triangulated with Triangle. The criteria for if a subdomain is sufficiently decomposed stays true to the original algorithm by Blueloch et al. (1996), 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 because 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 reaches a given tolerance, then decomposition ceases for this subdomain. The decomposition recursive level is dependent on the number of processes. Figure 22 shows the boundary layer decomposed into 128 Delaunay

Fig. 23. Initial four isotropic mesh region quadrants for decoupling that cover the remainder of the domain—the farfield. The near-body isotropic mesh subdomain is shaded black, and the corner vertices of the near-body isotropic mesh subdomain are shown.

subdomains, which can all be triangulated independently. To summarize, the three conditions for when a boundary layer subdomain is sufficiently decomposed are as follows: (1) the subdomain has no internal vertices, and all vertices are incident on an edge of the subdomain; (2) the subdomain has fewer vertices than a user-specified tolerance (this value is set to 250 in the code because it is quicker to triangulate these small subdomains than it is to decompose the subdomain further); and (3) all processes have a boundary layer subdomain, so no more subdomains need to be decomposed since there is enough work for each process.

3.6 Graded Isotropic Mesh Region

After the boundary layer has been determined, the remainder of the domain needs to be meshed using Delaunay refinement. There are no vertices in the isotropic mesh region yet. The large isotropic mesh region needs to be broken into smaller, independent subdomains so that multiple subdomains can be meshed concurrently. To facilitate the parallel refinement, the approach of generating graded Delaunay decoupling paths (Linardakis and Chrisochoides 2008) is followed. The decoupling aims to discretize the borders between subdomains in such a way that the Delaunay property is maintained when the subdomains are independently refined. The algorithm splits one subdomain into four independent subdomains that can be refined concurrently. This algorithm was chosen because it is recursive, so subdomains can be repeatedly split to generate enough work so that each process has a subdomain.

The size of the isotropic triangles is determined by a sizing function. The sizing function specifies smaller triangles near the boundary layer and larger triangles near the outer boundary of the domain. This sizing function is used with Triangle to provide a smooth gradation of triangle size based on distance from the initial geometry toward the farfield. The farfield typically has a size of a factor of 30 to 50 chord lengths, the length of the airfoil, from the initial geometry in the x and y directions. The borders between subdomains are a collection of vertices and edges that form a closed polygon. The edges are constrained, meaning that the edges of the border will not be split and will be present in the resulting triangular mesh. The edges between subdomains are discretized using the triangle sizing function. Each segment of the shared border needs to be bound by the circumradius-to-shortest-edge ratio of $\sqrt{2}$, dictated by Ruppert's algorithm for Delaunay refinement (Ruppert 1995), and bound by the sizing function for the location. The decoupling method is a conservative approach that refines the shared border to account for the shortest edges that may be created during Delaunay refinement. To generate subdomains that can be efficiently triangulated and refined by Triangle, it is beneficial to create subdomains that are convex and do not contain any holes, as Triangle first creates an initial triangulation and then removes elements inside concavities and holes from the initial triangulation. Although convexity does improve efficiency, it is not necessary for correctness of Linardakis' algorithm. The isotropic mesh region is

initially decoupled into four quadrants shown in Figure 23, where the solid region is the near-body subdomain that contains the airfoil.

The graded Delaunay decoupling path needs to be generated for the four initial subdomains. For each subdomain, a value k is computed at different vertices on the decoupling path. This value k determines the sizing function's specification of the local edge length of triangles at a given vertex. This edge length size is based on the termination conditions of Delaunay refinement that will be used for the decoupling procedure. This edge length is determined from (1), where A is the area of the desired triangle at the given vertex. The theorem and proof for this equation is presented by Linardakis and Chrisochoides (2008), and the $\sqrt{2}$ term in the denominator is intentional and used to bound triangles by the maximum circumradius-to-shortest-edge ratio that is guaranteed by Delaunay refinement. The sizing function takes an input of a vertex location and outputs the desired triangle area, A , which is used as an input for (1). For each of the red vertices that is shared with the near-body subdomain in Figure 23, we march along the shared borders between subdomains toward the farfield and then around the outer border to insert new vertices.

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

The decoupling process remains the same after the initial decoupling, only the decoupling path is altered for future subdomains. Further decoupling paths are in the form of a “+” shape where a new vertex is inserted at the center of a subdomain and a path is created from the center vertex to each of the four midpoints along the sides of the subdomain. This creates four new subdomains to replace the old subdomain. New vertices are not added to the outer border of the subdomain, as the outer border of the subdomain is already properly discretized, but the interior is empty. The decoupling path connects to an already existing vertex that is closest to the midpoint of the outer border. This eliminates the need for communication between processes since new vertices are only inserted within the interior of a subdomain, and thus the shared borders between other subdomains are undisturbed. Figure 24 shows the border for a subdomain, whereas Figure 25 shows the resulting mesh for the subdomain. There are 612 triangles in this subdomain. Now consider the procedure for splitting the subdomain in Figure 24 into four new subdomains. Figure 26 shows the center vertex inserted and the decoupling path that connects to existing vertices on the original subdomain. These four subdomains can now be meshed concurrently. Figure 27 shows the resulting mesh and contains 614 triangles for the four new subdomains, which is only two more triangles than the mesh for the original subdomain. The decoupling process is recursive and subdomains are repeatedly decoupled and sent to other processes until all processes have sufficient work to ensure that no process is idle and wastes computational resources. Once a subdomain is ready to be refined, the border is constructed. The vertices are stored in counter-clockwise order, so constructing the border is done by iterating over the vertices in order. The same sizing function used for the decoupling is also used for Triangle's Delaunay refinement area bound. Figure 28 shows the decoupled Delaunay subdomains. Each subdomain can be independently refined. Subdomains were decoupled based on the estimated number of triangles for the subdomain. Each subdomain has roughly the same number of triangles, so subdomains near the airfoil at the center have a smaller area because the triangles have a smaller area. The number of triangles for an isotropic mesh region subdomain is estimated by sampling the sizing function at several control points inside the subdomain. The values from the sizing function represent the desired area of triangles for the local region. These values are used with the area of the subdomain, calculated using the outer border vertices and the shoelace formula (Braden 1986) (also known as Gauss' area formula), to estimate the number of triangles that will be in the subdomain.

Fig. 24. Border for a subdomain.

Fig. 25. Resulting mesh for the subdomain.

Fig. 26. Borders for the new subdomains.

Fig. 27. Resulting mesh for the new subdomains.

Fig. 28. Decoupled Delaunay subdomains.

3.7 Load Balancing

Each process starts with a subset of the domain. A process may have some decomposed boundary layer subdomains and some decoupled isotropic region subdomains. To ensure that no process idles too long, when a process has a small amount of work left, it will request work from another process. A process' work load is based on the total cost of all of the subdomains it has left to triangulate or refine. A subdomain's cost is computed as an estimate of the number of triangles that will be in the subdomain. A global memory window, detailed in Section 4, is allocated on the root process as an array that will hold the work load estimates for each process. Each process will periodically update its work load estimate.

Whenever a process' work load falls below a threshold, it will fetch the global memory window and determine the processes that have work available. The process that is low on work will ask for more work from a random process from the set of suitable processes. The process that has been requested will send a response confirming or denying that it can give work to the process that is low on work. If the requested process denies the work request, then the process that is low on work will pick a different random process from the set of suitable processes until a process is found that can fulfill the request for more work. This helps to alleviate the burden of coordination by the root process. If a process runs out of work and cannot find another process to request more

work from, then this process sends a message to the root informing the root that this process is out of work and finished meshing.

4 IMPLEMENTATION

The software standard used for our implementation is MPI, implemented as MPICH v3.0. We also use the POSIX Threads model because each process has two threads: a meshing thread and a communicating thread. While the meshing thread is decomposing, decoupling, triangulating, or refining subdomains, the communicating thread takes care of updating the process' work units count on the global memory window, evaluating the process' current work load and requesting work from other processes, and fulfilling work requests from other processes.

The global memory window on the root is an MPI window object, which is used to facilitate remote memory access (RMA). RMA is advantageous because it is a direct memory transfer from the memory of one process to the memory of another process. These direct memory accesses bypass the operating system by using the network adapter to yield zero-copy, high-throughput, and low-latency transfers. There is only a 7.2-microsecond latency for MPICH—the MPI implementation used with this package (Liu et al. 2004). However, the hardware needs to support this type of communication to reap the benefits of RMA. The cluster used for the evaluations in this work supports RMA through InfiniBand. The specific interconnect is the 2011 4X FDR InfiniBand with a throughput of approximately 56 gigabits per second (Gb/sec) (InfiniBand Trade Association 2018). The communicator thread on each process will periodically update how much work the process has by using RMA via the MPI put operation, which puts the current work load estimate on the global memory window on the root process. When a process needs work, the communicator thread will fetch the values in the global memory window using RMA via the MPI get operation, which stores the values of the global memory window into a variable on the current process' memory. From there, the communicator thread determines who it should request work from. The actual transfer of work is done through MPI send and receive operations, not RMA operations.

Since all processes write and read data from the root, there is a potential for a bottleneck. RMA operations continue in parallel with other system operations and bypass the processors, caches, and context switches. A bottleneck may exist at the interconnect. The interconnect used by the cluster in this work's evaluation has a throughput of 56Gb/sec, but older interconnects may have as low of a throughput as 2Gb/sec. Each process in the application will update their workload estimate in the global memory window no more than 10 times per second. Each update is an 8 byte long integer, or 64 bits. So the maximum amount of bits per second that a process will write to the root process is 640 bits. If the older interconnects have a throughput of 2Gb/sec, the maximum number of simultaneous processes that can write to the root are $P = \frac{2 \text{ Gb/sec}}{640 \text{ bits/sec}} = \frac{2*1024*1024*1024 \text{ bits/sec}}{640 \text{ bits/sec}} = 3,355,443 \text{ processes}$.

Practical data structure and algorithm designs must be utilized to achieve a scalable and cache-friendly distributed memory, parallel application. The boundary layer vertices are locally stored contiguously, and the ordering is implicitly known by each process due to the structured configuration, so the root process does not need to reorder the vertex vector after the gather operation. A contiguous memory container is used for storing the vertex objects to take advantage of spatial locality in the cache during the boundary layer decomposition since the vertex objects are iterated over in order for projecting and flattening the vertices and for computing the lower convex hull. For the task of projecting the vertices onto the paraboloid and flattening them onto the vertical plane, we made the decision to include the projected coordinates in the class definition for vertex objects instead of creating a separate array to store the projected coordinates. This allows us to take advantage of spatial locality when we are storing or accessing the projected coordinates,

and it also avoids the repetitive allocation and deallocation of the projected coordinates since the data is allocated once at the creation of a vertex object. The vertex class stores vertex objects as contiguous memory and has a unique identifier, two doubles for the Cartesian coordinates, two coordinates for the projected coordinates, a flag representing if the vertex is on a boundary, and a flag if the vertex is on the lower convex hull during a decomposition step. The edge class also defines contiguous storage for edge objects and consists of a unique identifier, the vertex ids for the endpoints, and the type of edge. The type of edge is used to apply boundary conditions to the model surface and the farfield boundary. The final basic data structure is the ray class, which is used to create the boundary layer vertices. Ray objects are also stored contiguously, and the key variables are the coordinates and id for the vertex on the surface where the ray is emitted from, a vector representing the direction to insert vertices along, and the number of vertices to insert along the vector. The id of the outer layer vertex is stored with the ray object and is used to create the enclosing border of the boundary layer.

During the boundary layer decomposition, for the task of partitioning the vertices after the lower convex hull is computed, 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, and low-level memory copies can be used for the primary-axis-sorted vertices, or the x -sorted vertices in this case. Some of the vertices of the lower convex hull may appear before the median vertex (having an x -coordinate less than the median vertex's x -coordinate), whereas other vertices of the lower convex hull appear after the median vertex. All of the vertices of the lower convex hull need to be included in both subdomains. Vertices of the lower convex hull that are before the median vertex need to be added to the right subdomain, whereas vertices of the lower convex hull that are after the median vertex need to be added to the left subdomain. To include all of 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, whereas 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.

On examining the source code for Triangle, we noticed that the input vertices are sorted by their x -coordinate on invocation. This allowed us to remove the sorting step from Triangle since we maintain the x -sorted vertices on each decomposition step and use the x -sorted vertices to populate the input data used to call Triangle. Since we overdecompose the boundary layer and isotropic mesh region, which yields subdomains with a small number of vertices, we also specify that Triangle should only use vertical cuts for the divide-and-conquer algorithm, which improves the performance for small vertex sets.

Our algorithm is efficient in terms of communication and algorithmic performance. For the decoupled isotropic mesh region, we store the vertices in counter-clockwise order so the edges do not need to be created until the subdomain is ready to be refined with Triangle. This reduces the communication costs since there is less data to be transmitted. This also saves computational resources during successive decoupling steps for the same subdomain because constructing the edges after

each decoupling step would require us to determine which edges of the original subdomain belong to which newly decoupled subdomain. We would also waste computational resources due to data moving costs and the repetitive allocations and deallocations of the edges. When a boundary layer subdomain is sent to another process for decomposition, the projected coordinates are not communicated since they are dependent on the median vertex, which changes between each decomposition step. During the decomposition step, after the dividing path has been determined by the lower convex hull, but before the vertices are partitioned, we determine if either of the two new subdomains will be sufficiently decomposed. For each of the new subdomains that are sufficiently decomposed, we only maintain the x -sorted vertices since these are the vertices that are needed by Triangle. This also reduces communication costs for transferring a boundary layer subdomain that has already been sufficiently decomposed.

For the load balancing, subdomains are stored in a priority queue where the subdomain at the front of the queue is estimated to need the most time to mesh. Meshing the largest subdomains first, when all processes have work, allows us to save the smaller subdomains for more aggressive load balancing when there is little total work left and the application is close to terminating. This helps us minimize process idle time during the final moments of execution. If a process has a boundary layer subdomain, then these subdomains are at the front of the queue since they contain the most vertices, compared to the isotropic mesh region subdomains, which only have vertices on their border, and thus the boundary layer subdomains take longer to transfer to another process. Since we have a mesher thread and a communicator thread for each process, the communication times only cause a slowdown when the mesher thread runs out of work and has to wait on a large subdomain to be received by the communicator thread. The communicator thread requests additional work before the mesher thread runs out of work.

5 CONVERGENCE STUDY

The flow solver, FUN3D (Biedron et al. 2015), was used to run three-dimensional simulations. The algorithm described in this article outputs a two-dimensional mesh that is then converted to a three-dimensional mesh through a preprocessing tool used with FUN3D. The two-dimensional triangles are placed in the xz -plane with a y -coordinate of 0. The two-dimensional triangles are also duplicated in the xz -plane but with a y -coordinate of 1. The extrusion joins the pairs of triangles in the two planes to form triangular prisms.

The meshes generated with this approach have been validated against reference solutions from the NASA Langley Research Center's Turbulence Modeling Resource Website. The NACA 0012 airfoil is used for validation. The forces of airflow are simulated over the airfoil, and the reference simulation parameters are for an airfoil with a chord length of 1 at mach speed 0.15, Reynolds number of 6 million, and angle of attack at 10 degrees. The farfield is located 500 chord lengths away from the airfoil to avoid a farfield vertex vortex. The freestream flow conditions specify the reference temperature to be 540 Rankine. The governing equations are the Navier-Stokes equations using the Spalart-Allmaras turbulence model with compressible, calorically perfect gas. The boundary conditions for the NACA0012 surface are viscous, and the no-slip condition is explicitly set. The bounding box of the domain has its boundary condition set to be the farfield by using Riemann invariants. Boundary conditions are also set for the extruded surfaces to enforce the symmetry plane. The reference values for the coefficient of lift and coefficient of drag are given as 1.0983 and 0.01242, respectively, in the convergence study of Diskin et al. (2016).

The approach in the algorithm is used to generate a family of meshes around NACA 0012 airfoils. A scaling factor of $\frac{1}{\sqrt{2}}$ is used to vary the density of the boundary discretization for the airfoil, the thickness of the first layer off the wall, the growth rate of the boundary layer's exponential growth function, and the number of layers in the boundary layer. The first model and mesh are generated

Fig. 29. Family of meshes around the NACA 0012 airfoil. Each mesh has approximately twice as many vertices as the previous mesh.

Table 1. Table of Geometry and Simulation Data for the Convergence Study

Airfoil Vertices	Mesh Vertices	Boundary Layer Triangles	Isotropic Region Triangles	CL	CD
160	16,511	16,180	16,618	1.0066	0.02060
224	32,598	32,228	32,648	1.0521	0.01572
320	65,149	64,760	65,082	1.0745	0.01350
452	131,125	128,976	132,622	1.0831	0.01271
640	265,560	259,120	271,072	1.0867	0.01238

using a scaling factor of 1. Each subsequent model and mesh use a scaling factor of $\frac{1}{\sqrt{2}}^i$ for the i^{th} model (starting at zero). Using the scaling factor, the total height of the boundary layer remains invariant between the different meshes, but the number of vertices in the boundary layer doubles from each previous mesh. Figure 29 shows the family of five meshes at the leading edge used for this convergence study, and Table 1 shows the mesh statistics and the results of each simulation. All simulations were fully converged to machine zero. For the largest mesh, the coefficient of lift has approximately a 1% error with the reference value 1.0983, and the coefficient of drag has less than 0.5% error with the reference value 0.01242.

6 ALGORITHMIC PERFORMANCE AND EVALUATION

We executed our application on a cluster with 128 nodes with a 4X FDR InfiniBand interconnect with RMA support (approximately 56Gbit/sec) dual-socket eight-core 2.60GHz Intel E5-2670 Sandy Bridge (2 x 20 MB cache) with 32GB RAM per node. Each MPI rank was assigned to a core. This was the largest amount of computational resources that could be consumed for one job on the cluster at NASA Langley. We show the strong scaling speedup in Figure 30, 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 (see steps 2 through 16 in Figure 3); the weak scaling speedup in Figure 31, the rate of total work completed as the number of processes increase, while the amount of work for each process is constant; and the strong scaling and weak scaling efficiency in Figure 32 and Figure 33, respectively, the ratio of speedup to the number of processes used. We evaluated the strong scalability, the speedup when the amount of work, or mesh size in this case, is kept fixed. The execution times that we measured do not take input and output times into consideration (see steps 1, 17, and 18 in Figure 3). Figure 30 shows good strong scalability up to 512 processes with a speedup of approximately 368. Figure 32 shows the efficiency is roughly 72% for 512 processes. Since our application uses Triangle for the boundary layer triangulation and isotropic mesh refinement, our application's running time using one process is almost equivalent to the execution time of Triangle. The sequential meshing time of Triangle is 192 seconds, whereas the sequential meshing time of

our application is 196 seconds, yielding an efficiency of approximately 98% for the sequential case. The increase in meshing time for our application is due to the additional triangles created by the isotropic decoupling method. The time to read the input file is under 1 second for 1,500 surface vertices. The sequential time to write an ASCII file for the mesh with 172,768,355 triangles is 9 minutes. The requirements of the output file are dependent on the flow solver being used. If a flow solver can handle a distributed mesh or read from a binary file, the writing time will be less. A parallel file writer can be implemented to reduce the time it takes to output the mesh. This would eliminate step 15 and make step 16 parallel in Figure 3. The performance of our application was also compared to AFLR2 (Marcum 1998), a commercial two-dimensional mesh generator that can create semi-structured boundary layer meshes. AFLR2 is a sequential code, so we compared the sequential performance of the applications for a NACA0012 airfoil. The finest mesh for the convergence study, the last mesh in Figure 29, is used for the performance comparison. The same input geometry was used, and the same outer boundary discretization for the farfield was used for both applications. The same first layer thickness of the boundary layer and the same exponential growth rate for the boundary layer growth function were used for both applications. AFLR2 created 265,055 vertices in 4.887 seconds, and our application created 265,560 vertices in 0.976 seconds, a five times speedup. Both applications took a little more than 2 seconds to write the output file.

Only 512 processes are used for the strong scaling study, because the largest mesh that could fit in memory on one process for the sequential case contained 175 million triangles. When a large number of processes are used, the resulting mesh is generated in under 1 second, which is too short of a time to get a reliable measurement. To evaluate how well the application performs for generating larger meshes, the weak scaling performance is evaluated by increasing the amount of work roughly proportional to the number of processes. The mesh size is dependent on the input parameters, particularly the number of layers in the boundary layer, growth functions, and distance to farfield, where the cell size of the outer triangles of the boundary layer dictates the triangle area for the isotropic mesh region. To scale the work proportionally to the number of processes, we kept the input geometry, isotropic mesh region sizing function, and distance to the farfield fixed for each execution. The boundary layer and isotropic mesh region growth functions are made uniform, where the number of layers is proportional to the number of processes and the initial distance from the geometry is inversely proportional to the number of processes. With gradation control turned off, this doubles the number of vertices in the boundary layer when the number of processes is doubled but keeps the overall boundary layer thickness constant since the initial distance from the geometry is halved, and thus the area of the triangles on the outer border is halved, yielding an isotropic triangle area that is also halved. However, the actual number of triangles created for the isotropic mesh region size only roughly increases proportionally with the number of processes due to the Delaunay refinement operation. To alleviate this concern, the rate of triangles created per second (in millions) is shown in Figure 31. The parallel efficiency compared to the sequential execution is shown in Figure 33. A 79% weak scaling efficiency is achieved with 1,024 processes and 1.2 billion total triangles created per second. The sequential case took 64 seconds to complete and the resulting mesh contains 100,582,272 triangles, whereas the 1024 case took 81 seconds to complete and the resulting mesh contains 103,345,182,528 triangles. In Figure 31, the “ideal” curve represents linear speedup and shows a linearly proportional increase of the number of processes used to the total number of triangles created per second. The sequential case with one process has a rate of 1,571,598 triangles per second. Ideally, two processes should produce twice as many triangles per second.

The main contribution to the drop in efficiency is the time it takes to create the initial boundary layer subdomain due to having to sort the vertices. The two sorting algorithms we explored were mergesort and quicksort because they lend themselves to be parallelized. The parallel mergesort

Fig. 30. Strong scalability up to 512 processes with a fixed mesh size of 172,768,335 triangles.

Fig. 31. Weak scalability up to 1,024 processes.

Fig. 32. Efficiency of the strong scalability measurement up to 512 processes with a fixed mesh size of 172,768,335 triangles.

Fig. 33. Efficiency of the weak scalability measurement up to 1,024 processes.

algorithm starts by scattering the input across all processes. Each process sorts their range, then sends or receives data to or from a partner process. The sending process then exits while the receiving process merges its data and its partner's data. However, the final step is a sequential merge of all n elements of the input, which is $O(n)$ complexity. Mergesort exhibits poor performance due to the computational structure where the most parallelism is exploited in the beginning steps when each process has a portion of the data. After each iteration, the number of working processes is halved, whereas the amount of data for each working process is doubled, and thus mergesort does not scale. The other consideration was quicksort, because after each partitioning step the data is divided into two independent regions. The parallel quicksort works by scattering the input across all processes. Then the leader process broadcasts the pivot to its team. Each process in the team then divides their portion into a lower and upper subarray, based on the pivot. After the partitioning step is done, the team will split into two new teams where one team partitions the lower subarray and the other team partitions the upper subarray. The old team does an all-to-all communication to receive new elements and send old elements. For example, a process that will be part of the lower team will send all of its upper elements to members of the upper team. The parallel quicksort exploits the most parallelism after $\log p$ iterations, because each process has an independent portion of the data. However, these subarrays are not the same size, so some processes may finish sorting their own subarray and idle. The load imbalance is dependent on the pivot choice. In addition, there is a lot of communication and synchronization at each partitioning step due to the all-to-all operation. Mergesort would also not solve the memory bottleneck problem. However, the root process does not need all of the vertices; it only needs the boundary layer vertices, not the isotropic mesh region vertices.

The refinement stage is a sequential portion of the algorithm, but it is inexpensive compared to the task of meshing the subdomains (see step 15 in Figure 3). The root process performs all of the refinement work, but many of the refinement operations lend themselves to be parallelized. Using the ADT, all of the intersections can be determined in $O(n * \log n)$ time, where n is the number of vertices on the input surface. Step 4, Laplacian smoothing, may be parallelized in two ways. Non-overlapping subregions of the mesh can be concurrently smoothed. Thus, after all of the smoothing regions have been determined, each of the independent ranges can be distributed to other processes. Then each process can smooth its own subregion concurrently without communicating to other processes. The other option to parallelize the smoothing operation is to have all processes work together. After all of the smoothing regions have been determined, each process takes a subset of the domain to be smoothed. Each subset has two neighboring subsets: one before and after the subset. Neighboring subsets share one normal vector on the partition boundary, which will need to be communicated to the neighboring process after each iteration. Each process will perform one iteration of smoothing over its subset of the domain and then communicate with the neighbors to update the normal vectors on partition boundaries.

Steps 5 and 7 can be done locally with each process' subset of the domain. Communication will need to be done for partition boundaries between processes to ensure conformity. These two steps involve decreasing and increasing, respectively, the number of layers (or vertices) that will be inserted along each ray. This provides a smooth transition into the isotropic mesh region because the outer layer triangles in the boundary layer are isotropic. Step 6 requires constructing the ADT for geometric searching to efficiently determine where there are intersections in the boundary layer. Each process could query the ADT for its subset of the domain. However, each process would still have to construct the ADT with the entirety of the domain, because each process needs to check if its subset of the domain intersects with any region inside the entirety of the domain. Currently, the root already has to construct the ADT for the entirety of the domain while all other processes are idle. If this step were parallelized, then each process would need to repeat this operation, but it would not affect the runtime because the root already has to perform this substep. Computational savings will be observed for the querying substep because the root will no longer be testing every ray in the domain for an intersection. Each process will query the ADT for its subset of rays in the domain. However, synchronization will be required to resolve intersections between two subsets of the domain that are owned by different processes. The two processes will need to communicate to coordinate how many layers need to be removed from each intersecting ray.

Step 8 does not require parallelization, as there is not enough work for each process. Multiple normals are added at sharp trailing edges on the geometry. Typically, there are only a handful of sharp trailing edges on models. Figure 2 has three sharp trailing edges: one at the top and one at the bottom of the leftmost slat, and one on the center slat toward the right side of the slat. A fan of additional rays needs to be added at each of these three sharp trailing edges. Since this operation is a local operation, the three sharp trailing edges can be treated concurrently, so only three processes would have work. However, if a model were to have a few hundred sharp trailing edges, then a small increase in speedup may be observed. Note that the operation of adding a single fan of additional rays is inexpensive because it only requires rotating a vector between two fixed vectors.

Steps 11 and 12 are the only significant sequential bottleneck of the algorithm. The boundary layer vertices need to be gathered at the root so that the root can decompose the boundary layer domain into two subdomains. The root keeps one subdomain and sends the other subdomain to process one. The root and process one decompose each of their subdomains and then send a subdomain to process two and process three, respectively. These operations repeat in parallel until every

Fig. 34. Circulation control airfoil with regions of interest.

Fig. 35. Region (a) mesh.

Fig. 36. Region (b) mesh.

Fig. 37. Region (c) mesh.

process has a boundary layer subdomain. Step 11, gathering the vertices at the root, is necessary so the root can perform the initial decomposition—step 12. The computational complexity of the main step of the algorithm, the triangulation of subdomains performed by Triangle, is $O(n * \log n)$, where n is the number of vertices in the subdomain. However, the refinement is usually $O(n)$ in practice, where n is the size of the output.

7 ROBUSTNESS FOR A COMPLEX INPUT MODEL

The circulation control airfoil (Englar et al. 2009; Rumsey and Nishino 2011), developed and analyzed between NASA Langley and Georgia Tech, has many features shown in Figure 34. Figure 35 shows region (a) of the circulation control airfoil after the normal vectors were smoothed in the concavity using Laplacian smoothing to resolve intersections. Gradation control causes a couple of additional layers to be added. Figure 36 shows region (b), which contains smooth and discontinuous surface slopes inside the chamber. Normal vectors in concave regions and regions of high convex curvature are smoothed using Laplacian smoothing. Gradation control removes layers near concavities and adds layers near convex regions. Figure 37 shows a blunt trailing edge near a small gap. The normal vectors are smoothed around the the blunt trailing edge region, and layers are removed to treat the intersections in the small gap.

8 FUTURE WORK

Future work will consist of extending the approach in this article to handle three-dimensional domains. Recall that the main operations of the algorithm are the boundary layer vertex placement strategy, boundary layer subdomain decomposition, and isotropic subdomain decoupling. The boundary layer vertex placement strategy aims to insert vertices along the normal at each

input node. This method extends directly to three-dimensional domains where the normal at each input node is the average of the normals of all incident faces. Multiple normals will need to be present at surface discontinuities. Smoothing and pruning of the normal direction vectors will still need to be performed, and these operations are defined for three-dimensional vectors. The boundary layer subdomain decomposition step holds for the property that each element of the Delaunay tessellation of d -dimensional vertices corresponds to a facet of the lower convex hull of the projection of these vertices onto a $(d + 1)$ -dimensional paraboloid. For three-dimensional domains, the four-dimensional paraboloid will be projected onto one of the three-dimensional planes where the lower convex hull will be computed. The faces of this three-dimensional lower convex hull will correspond to the triangular faces of tetrahedra in the dividing plane between two independent three-dimensional subdomains, analogous to how the edges of the two-dimensional lower convex hull corresponds to the edges of the triangles in the dividing path between two independent two-dimensional subdomains. However, the algorithm requires that all of the boundary layer vertices be present on one process so that the initial decomposition can be performed. This will cause a bottleneck for large-scale three-dimensional geometries. A modified approach will need to be explored, such as by performing early decoupling (Linardakis and Chrisochoides 2008) of the subproblems. There are two options for generating the three-dimensional isotropic mesh region in parallel. One option is extending the decoupling method to three-dimensional domains by creating shared planes of triangular faces that will serve as the shared border. The triangular faces will be bound by the sizing function, but it is unclear if the bound on the circumradius-to-shortest-edge ratio will be sufficient for the 3D decoupling method since a sliver tetrahedron can have a low circumradius to shortest edge but still be of poor quality due to an unfavorable dihedral angle near 0 degrees or 180 degrees and a near zero volume. Additional quality criteria will need to be used to avoid sliver tetrahedra. An alternative to the decoupled approach is a so-called constrained parallel meshing approach, which relies on asynchronous messages for maintaining boundary conformity.

9 CONCLUSION

The framework and a practical implementation to generate high-quality, two-dimensional unstructured initial meshes for complex domains in parallel is developed for use in FEA simulations on distributed-memory machines. The size of a mesh is dependent on the application, so parallel mesh generation is motivated by very large two-dimensional meshes, which are needed to fully resolve the flow features that model complex physical phenomena. The extension to handle three-dimensional domains is discussed where the mathematical properties of the boundary layer generation process holds true, whereas options are available for further exploring how to generate the isotropic mesh region. The input geometry, boundary layer parameters, and distance to the farfield are the only inputs to this automated process. The application has three software dependencies—the software Triangle, POSIX Threads, and MPICH v3.0—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 FEA pipeline will yield a final, acceptable mesh in fewer iterations than an ill-suited initial mesh. 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. This approach provides an efficient and automated parallel application to facilitate aerospace development with rapid turnaround time.

ACKNOWLEDGMENTS

We thank Mike Park for his discussions and comments. We also thank Chris Rumsey and Joseph Derlaga for the circulation control airfoil. We thank the reviewers for their detailed comments that helped in improving the article.

REFERENCES

- Alex M. Andrew. 1979. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters* 9, 5 (1979), 216–219.
- R. Aubry, S. Dey, E. Mestreau, K. Karamete, and D. Gayman. 2015. An entropy satisfying boundary layer surface mesh generation. *SIAM Journal on Scientific Computing* 37, 4 (2015), A1957–A1974.
- Robert T. Biedron, Jan-René Carlson, Joseph M. Derlaga, Peter A. Gnoffo, Dana P. Hammond, William T. Jones, Bil Kleb, et al. 2015. FUN3D Manual: 12.8. NASA/TM-2015-218807. Retrieved March 28, 2019 from https://fun3d.larc.nasa.gov/papers/FUN3D_Manual-12.8.pdf.
- Guy E. Blelloch, Gary L. Miller, and Dafna Talmor. 1996. Developing a practical, projection-based parallel Delaunay algorithm. In *Proceedings of the 12th Annual Symposium on Computational Geometry*. ACM, New York, NY, 186–195.
- Javier Bonet and Jaime Peraire. 1991. An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problems. *International Journal for Numerical Methods in Engineering* 31, 1 (1991), 1–17.
- Bart Braden. 1986. The surveyor’s area formula. *College Mathematics Journal* 17, 4 (1986), 326–337. <http://www.jstor.org/stable/2686282>.
- John R. Chawner, John F. Dannenhoffer III, Saikat Dey, Hugh Thornburg, William T. Jones, Jeffrey Slotnick, and Nigel J. Taylor. 2015b. *The Path to and State of Geometry and Meshing in 2030: Panel Summary*. Technical Report. Department of Mechanical and Aerospace Engineering, Syracuse University.
- John R. Chawner, John F. Dannenhoffer III, and Nigel J. Taylor. 2015a. *Geometry, Mesh Generation, and the CFD 2030 Vision*. Technical Report. Department of Mechanical and Aerospace Engineering, Syracuse, NY.
- Andrey N. Chernikov and Nikos P. Chrisochoides. 2006. Parallel guaranteed quality Delaunay uniform mesh refinement. *SIAM Journal on Scientific Computing* 28, 5 (2006), 1907–1926.
- Andrey N. Chernikov and Nikos P. Chrisochoides. 2008a. Algorithm 872: Parallel 2D constrained Delaunay mesh generation. *ACM Transactions on Mathematical Software* 34, 1 (2008), 6.
- Andrey N. Chernikov and Nikos P. Chrisochoides. 2008b. Three-dimensional Delaunay refinement for multi-core processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing*. ACM, New York, NY, 214–224.
- L. Paul Chew, Nikos Chrisochoides, and Florian Sukup. 1997. Parallel constrained Delaunay meshing. *ASME Applied Mechanics Division* 220 (1997), 89–96.
- Nikos Chrisochoides and Dmian Nave. 2003. Parallel Delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering* 58, 2 (2003), 161–176.
- Boris Diskin, James L. Thomas, Christopher L. Rumsey, and Axel Schwppe. 2016. Grid-convergence of Reynolds-averaged Navier-Stokes solutions for benchmark flows in two dimensions. *AIAA Journal* 54, 9 (Sept. 2016), 2563–2588. DOI: <https://doi.org/10.2514/1.J054555>
- Suchuan Dong and George E. Karniadakis. 2005. DNS of flow past a stationary and oscillating cylinder at $Re = 10000$. *Journal of Fluids and Structures* 20, 4 (2005), 519–531.
- Mark Drela. 1994. MSES: Multielement airfoil design/analysis system. Massachusetts Institute of Technology, Cambridge. <http://web.mit.edu/drela/Public/web/mSES/mSES.pdf>.
- Robert J. Englar, Gregory S. Jones, Brian G. Allan, and John C. Lin. 2009. 2-D circulation control airfoil benchmark experiments intended for CFD code validation. *AIAA Paper* 2009-902. Retrieved March 28, 2019 from <https://arc.aiaa.org/doi/10.2514/6.2009-902>.
- James D. Foley, Andries Van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. 1994. *Introduction to Computer Graphics*. Vol. 55. Addison-Wesley Reading.
- Panagiotis A. Foteinos and Nikos P. Chrisochoides. 2014. High quality real-time image-to-mesh conversion for finite element simulations. *Journal of Parallel and Distributed Computing* 74, 2 (2014), 2123–2140.
- R. V. Garimella. 1998. *Anisotropic tetrahedral mesh generation*. Ph.D. Dissertation. Rensselaer Polytechnic Institute.
- Rao V. Garimella and Mark S. Shephard. 2000. Boundary layer mesh generation for viscous flow simulations. *International Journal for Numerical Methods in Engineering* 49, 1–2 (2000), 193–218.
- Gerhard Globisch. 1995. —A parallel mesh generator. *Parallel Computing* 21, 3 (1995), 509–524.
- InfiniBand Trade Association. 2018. InfiniBand Roadmap. Retrieved March 28, 2019 from <https://www.infinibandta.org/about-infiniband/>.
- Yasushi Ito, Alan M. Shih, Anil K. Erukala, Bharat K. Soni, Andrey Chernikov, Nikos P. Chrisochoides, and Kazuhiro Nakahashi. 2007. Parallel unstructured mesh generation by an advancing front method. *Mathematics and Computers in Simulation* 75, 5 (2007), 200–209.
- Clemens Martin Joachim Kadow. 2004. *Parallel Delaunay refinement mesh generation*. Ph.D. Dissertation. Carnegie Mellon University.
- George Em Karniadakis and Steven A. Orszag. 1993. Nodes, modes and flow codes. *Physics Today* 46, 3 (1993), 34–42.
- A. I. Khan and B. H. V. Topping. 1991. Parallel adaptive mesh generation. *Computing Systems in Engineering* 2, 1 (1991), 75–101.

- L. Lämmer and M. Burghardt. 2000. Parallel generation of triangular and quadrilateral meshes. *Advances in Engineering Software* 31, 12 (2000), 929–936.
- Leonidas Linardakis and Nikos Chrisochoides. 2008. Graded Delaunay decoupling method for parallel guaranteed quality planar mesh generation. *SIAM Journal on Scientific Computing* 30, 4 (2008), 1875–1891.
- Jiuxing Liu, Weihang Jiang, Pete Wyckoff, Dhableswar K. Panda, David Ashton, Darius Buntinas, William Gropp, et al. 2004. Design and implementation of MPICH2 over InfiniBand with RDMA support. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. IEEE, Los Alamitos, CA, 16.
- David Marcum and Frederic Alauzet. 2015. Enhanced viscous mesh generation with metric-based blending. In *New Challenges in Grid Generation and Adaptivity for Scientific Computing*. Vol. 5. Springer International Publishing, 181–201. DOI: https://doi.org/10.1007/978-3-319-06053-8_9
- David L. Marcum. 1998. Advancing-front/local-reconnection (AFLR) unstructured grid generation. In *Computational Fluid Dynamics Review 1998 (in Two Volumes)*. World Scientific, 140–157.
- Dimitri J. Mavriplis. 1990. Adaptive mesh generation for viscous flows using Delaunay triangulation. *Journal of Computational Physics* 90, 2 (Oct. 1990), 271–291.
- Juliette Pardue and Andrey Chernikov. 2016. Parallel two-dimensional unstructured anisotropic Delaunay mesh generation of complex domains for aerospace applications. In *Proceedings of the 45th International Conference on Parallel Processing*.
- J. Peraire and K. Morgan. 1997. Unstructured mesh generation including directional refinement for aerodynamic flow simulation. *Finite Elements in Analysis and Design* 25, 3–4 (1997), 343–356.
- S. Z. Pirzadeh. 1996. Three-dimensional unstructured viscous grids by the advancing-layers method. *AIAA Journal* 34, 1 (Jan. 1996), 43–49. DOI: <https://doi.org/10.2514/3.13019>
- Christopher L. Rumsey and Takafumi Nishino. 2011. Numerical study comparing RANS and LES approaches on a circulation control airfoil. *International Journal of Heat and Fluid Flow* 32, 5 (2011), 847–864.
- Jim Ruppert. 1995. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms* 18, 3 (1995), 548–585.
- Jonathan Richard Shewchuk. 1996. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry Towards Geometric Engineering*. Springer, 203–222.
- Jeffrey Slotnick, Abdollah Khodadoust, Juan Alonso, David Darmofal, William Gropp, Elizabeth Lurie, and Dimitri Mavriplis. 2014. *CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences*. NASA CR-2014-218178. Langley Research Center. DOI: <https://doi.org/2060/20140003093>
- Rui Zhang, Khée Poh Lam, and Yongjie Zhang. 2011. Conformal adaptive hexahedral-dominant mesh generation for CFD simulation in architectural design applications. In *Proceedings of the Winter Simulation Conference*. 928–942.

Received April 2017; revised June 2018; accepted November 2018