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

Juliette Pardue and Andrey Chernikov
Department of Computer Science
Old Dominion University
Norfolk, VA, USA
{jpardue, achernik}@cs.odu.edu

***Abstract*—In this paper, we present a bottom-up approach to parallel anisotropic mesh generation by building a mesh generator from principles. Applications focusing on high-lift design or dynamic stall, or numerical methods and modeling test cases still focus on the two-dimensions. Our push-button 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. Our experimental results show 70% parallel efficiency over the fastest sequential isotropic mesh generator on 256 distributed memory nodes.**

Keywords—mesh generation; parallel algorithms; anisotropic; Delaunay;

I. INTRODUCTION

Mesh generation is a step in the iterative pipeline for designing aerospace structures as shown in Figure 1. After an analysis of the partial differential equations (PDE) solution on the mesh, the mesh is refined to yield a more favorable error estimation, which is typically a faster operation than generating an entirely new mesh. This is because the PDE solution identifies, for a mesh that possesses at least some degree of accuracy with respect to the PDE, a subset of mesh regions for refinement. The refinement process aims to gradually and incrementally add more resolution to the identified areas, as opposed to over-refining the mesh, which causes the PDE solver to waste computation time due to the excessive computations. After a PDE solution for a mesh is computed and analyzed, then the mesh may be refined to provide a more accurate solution. Assuming proper care for the refinement steps, this new mesh will be more accurate in the desired regions with respect to the PDE solution. This means that with each refinement step, the next iteration of the mesh will have a smaller error estimate than the previous mesh. So at each iteration, the overall refinement work required to reach a highly accurate solution decreases since the mesh's accuracy always improves. 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.

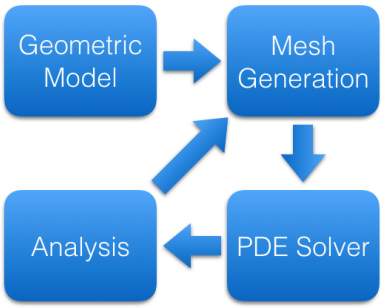

Figure 1. Development Pipeline

Since the end goal is to generate a mesh which accurately and efficiently fits the PDE, the time to achieve this goal is dependent on the number of iterations through the pipeline of mesh generation to PDE solver to analysis. Clearly, this iterative process needs an initial mesh to begin the process. If the initial mesh closely represents the PDE, then fewer iterations through the pipeline are required to achieve a suitable solution. However, if the initial mesh is highly inaccurate with respect to the PDE, then the first iterations through the pipeline will present numerous areas which require refinement. Eventually, after so many iterations through the pipeline that began with an unsuitable initial mesh, the current mesh will have similar error estimates as an initial mesh which was well-suited and closely represented the PDE. So the initial mesh sets the pace for the remainder of the iterations through the pipeline as well as the amount of refinement work. Clearly we need an initial mesh that has a high degree of accuracy with respect to the PDE while simultaneously being an efficient discretization of the domain in order to provide the most CPU savings to the PDE solver and to also generate the final mesh with the fewest number of iterations through the pipeline, thus yielding the fastest overall execution time.

A. Related Work

Current parallel mesh generators exist which handle the isotropic cases [5, 10, 11, 13, 14, 15, 16, 19, 20, 23, 24] do not perform well for parallel anisotropic mesh generation. Isotropic mesh generators which focus on solution-based adaptation create many unnecessary elements where there is a high degree

of gradation in the flow velocities in a direction. These anisotropic gradations in the flow velocities require anisotropic elements, typically with a 10,000:1 aspect ratio, so representing these regions with isotropic elements incurs a 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 computational fluid dynamics (CFD), a new mesh type has been introduced, graded anisotropic meshes. As stated before, these graded anisotropic meshes aim to decrease the computational efforts of the PDE solvers as well as to decrease the number of elements in the mesh. Others, sequentially, [8, 12] have begun developing anisotropic mesh generation paradigms to facilitate these CFD simulations. Serial two-dimensional tools for aerospace application development exist, such as XFOIL [17] and MSES [18], which also cater towards airfoil development. In this paper, we present an algorithm which:

- generates a high-fidelity, anisotropic boundary layer mesh in parallel based on a user-defined growth function
- generates a globally Delaunay, graded, isotropic inviscid region mesh in parallel
- resolves potential interpolation errors in the boundary layer caused by the local density of the mesh
- resolves self intersections and multi-element intersections in the anisotropic boundary layer
- is a push-button mesh generator, meaning the user only needs to provide the input configuration and wait for the output without any human intervention
- is scalable and efficient

II. ALGORITHM

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

A. Anisotropic Boundary Layer

Physical phenomena such as boundary layers in fluid mechanics are anisotropic in nature. There is a high degree of gradation in the flow velocities normal to the surface, thus it is beneficial to discretize the mesh in a way that efficiently captures these anisotropic flow velocities in order to yield substantial CPU savings without compromising accuracy. This dictates that the mesh should be refined in the direction normal to the surface, as shown in Figure 2, where these strong gradients exist. These characteristics allow for the extrusion-based point insertion along the normal of the surface at each vertex on the planar straight-line graph (PSLG). Essentially, each vertex is treated as an endpoint for a ray while the normal at the vertex is treated as the direction of the ray. New points are then inserted along the ray according to a growth function. There are multiple functions, as presented by Garimella and Shephard [1], which can be used to space the prospective points. Certain growth functions may yield a more accurate discretization of the domain depending on the PDE that is being solved. Two common growth functions are polynomial and geometric, which offer a uniform growth along the normal of the PSLG. However, other more sophisticated, adaptive growth functions [1], may be necessary for more complex geometries.

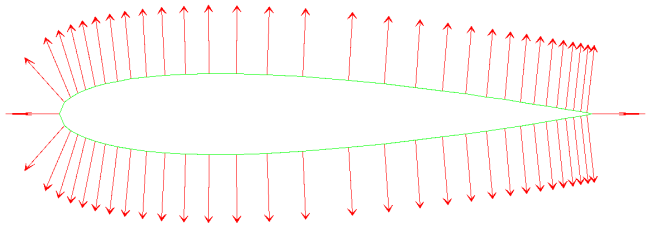

Figure 2. NACA 0012 airfoil with surface normals

Clearly, larger angles will naturally occur where the slope changes rapidly, such as the leading edge shown in Figure 2, and where the slope has discontinuities, such as the trailing edge in Figure 2 which causes these extremely large angles around cusps. The regions where these large angles occur are the areas of the mesh that need refinement to satisfy the resolution constraints of the mesh's boundary layer region. This is due to two factors: the governing PDE and the implicit geometrical representation which has larger angles between surface normals in these regions. The trailing edge is a highly discussed matter in the CFD field due to the Kutta condition and the presence of the stagnation points near the trailing edge and the trailing edge wake. The leading edge region is a high-gradient region with a stagnation point, so quality is critical as this is the first part of the airfoil that contacts the fluid, so if the leading edge region is not accurately discretized then this inaccuracy has a negative effect on the accuracy of the PDE solution.

B. Anisotropic Boundary Layer Refinement

Since the boundary conditions are calculated first and are propagated through and affect the entire solution over the mesh, it is critical that the boundary layer be properly discretized. This means avoiding the case of intersecting rays. Additionally, if the angle between two rays is too large, then the distance between vertices of neighboring rays will grow at excessively rapid rates, affecting the density of the mesh in the corresponding area, causing interpolation errors when the PDE solution is computed. To treat these cases, new points are created and uniformly spaced between the two points that have a large angle between their normals. Linear interpolation between the two original normals is used to compute the direction of the new normals. Cusps that typically exist at the

trailing edge are handled similarly. Instead of creating new vertices near the cusp point, a fan of rays is emitted at the cusp point where all of the rays of the fan have the cusp point as the origin. The direction of the rays is determined using linear interpolation. The fan of rays will curve inward towards the cusp point, as the physics dictate for these regions. This process is done in parallel where each process has a portion of the surface vertices (with the first and last vertex of a process' subset of the surface duplicated) and computes the normal at the vertex to create the corresponding ray. 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. If the angle is too large or there a cusp at the location, then the aforementioned approach of creating refining rays is implored. Figure 3 shows two large angles easily visible at the trailing edge of the leading slat of the 30p30n airfoil after the ray-based point insertion, while Figure 4 includes the fan of augmented points determined by the large angle detection. The rays of the fan curve towards the wake along with the rays before and after the fan.

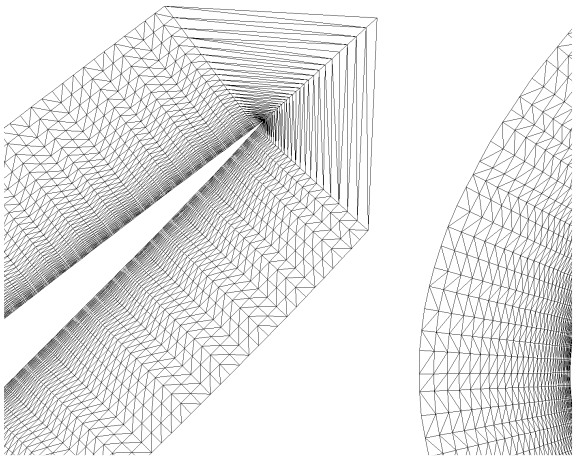

Figure 3. Poorly sized triangles at the trailing edge of the leading slat of the 30p30n airfoil caused by a discontinuity in the slope of the surface

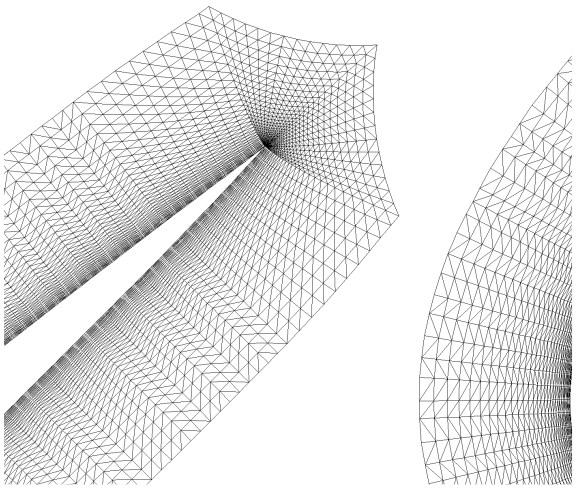

Figure 4. Properly sized elements at the trailing edge of the leading slat of the 30p30n airfoil after a fan of curved rays is added

Once the new rays have been determined, each element of the airfoil needs to be checked for self intersections of the rays; and in the case of a multi-element airfoil, each element's rays need to be checked for intersections with the outer border of the boundary layer for all other elements. A hierarchical approach to intersection checks is performed for multi-element intersections where candidate rays are pruned by whether or not they intersect the axis-aligned bounding box (AABB) of another element's boundary layer. A modified version of the Cohen-Sutherland algorithm for polygon clipping [6] is used to check for AABB intersections. These candidate rays are further pruned using an alternating digital tree (ADT) [21]. In two-dimensions, the ADT is used with a searching algorithm that determines if a four-dimensional point lies within a particular four-dimensional space subregion. By projecting line segments' extent boxes to four-dimensional points, a line segment's extent box (as a four-dimensional point) can be tested to see if it intersects with any of the n line segments' extent boxes (as four-dimensional points) in log(n) time. The candidate rays that intersect the AABB then have their extent boxes projected to four-dimensional points. The extent boxes of the enclosing border segments of the airfoil component's boundary layer are also projected to four-dimensional points 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 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 ray will only have points inserted up to the intersection point.

Self-intersections for each element are handled similarly. Instead of checking for intersections between rays and the outer border, intersections are checked between rays of the same element. For each airfoil element, each ray's extent box is projected to a four-dimensional point and added to an ADT. The ADT is queried against each ray's extent box to see which other rays have a potential intersection. Computational geometry is used to determine if the rays actually intersect, and in the case of an intersection, points are inserted up to the insertion point. Checking for intersections between n rays' extent boxes using the ADT can be performed in n*log(n) time.

C. Anisotropic Boundary Layer Point Insertion

Once the intersections have been resolved, each process will compute the points along their set of rays with respect to the user-defined growth function. Points are inserted until the intersection point, if it exists, or until the resulting triangles will be isotropic, to provide a smooth transition to the graded inviscid region. Figure 5 shows the smooth transition to the isotropic inviscid region. The points are then gathered at the root process in order to create the boundary layer subdomains. Since the points are locally stored contiguously and the ordering is implicitly known by each process due to the structured configuration, only the coordinates need to be communicated to the root, which significantly reduces the communication costs and alleviates a potential bottleneck.

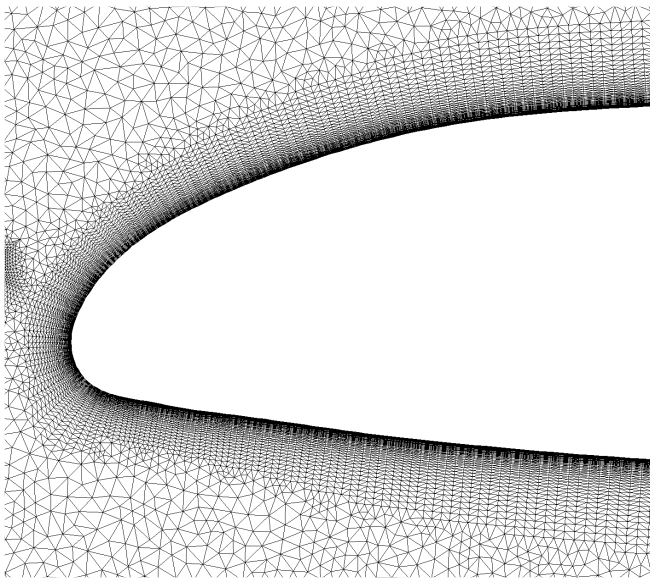

Figure 5. Main slat of the 30p30n airfoil showing different heights for the boundary layer in order to provide a smooth transition to the unstructured isotropic triangles

D. Parallel Triangulation of Anisotropic Boundary Layer

After the point insertion step is complete, the vertices in the boundary layer need to be triangulated. The algorithm presented by Blelloch et al. [2] is used, which utilizes the duality between the two-dimensional Delaunay Triangulation and the three-dimensional lower convex hull of a paraboloid, see Figure 6(a). The algorithm works by dividing a set of vertices into two subdomains with a median line and dividing path of Delaunay edges. The path of Delaunay edges divides Delaunay triangles based on if their circumcenter is to one side of the median line or to the opposite side of the median line. Kadow [16] provides a more in-depth proof regarding the mathematics concerning the relationship between the two-dimensional Delaunay triangulation, three-dimensional lower convex hull of a paraboloid, and two-dimensional lower convex hull of a paraboloid flattened onto a vertical plane. This approach was chosen because the dividing path created between subdomains corresponds to constraining edges which would be present in the final triangulation if the domain were triangulated sequentially without being decomposed. Other algorithms [7] which use user-defined dividing paths to arbitrarily partition the domain are undesirable as 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. These Delaunay edges are edges of Delaunay triangles in the final triangulation, which allows for each subdomain to be triangulated independently by a state-of-the-art Delaunay triangulator, Triangle [3]. Using this approach as a coarse-partitioner, each subdomain only needs to be recursively divided until there are enough subdomains to yield an acceptable degree of load balancing for the concurrent triangulation of the subdomains.

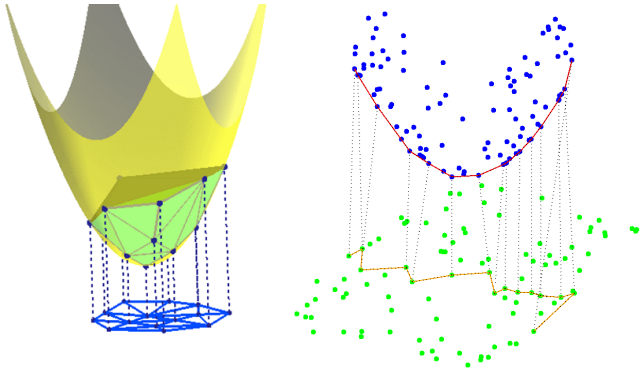

Figure 6. (a) Paraboloid and lower convex hull of paraboloid with corresponding 2D Delaunay triangulation for a sample point set; (b) Flattened paraboloid projection of the point set and lower convex hull with corresponding 2D point set and dividing Delaunay path for a sample point set

Our algorithm starts by creating an initial subdomain which stores vertices in x-sorted order and a copy in y-sorted order. This allows for the bounding box to be computed in constant time using the first and last vertex of the x-sorted and y-sorted vertices. The cut axis is set to be the axis parallel to the shortest edge of the bounding box to avoid the creation of long, skinny subdomains which are more expensive to triangulate with Triangle due to the merge step of Triangle's divide-and-conquer approach. Maintaining the sorted vertices also allows for the median vertex along the cut axis to be located in constant time. Using this median vertex, the vertices along the cut axis are projected onto a paraboloid centered at the median vertex and then flattened onto the vertical plane perpendicular to the cut axis, see Figure 6(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 [4], see Figure 7 showing the steps of the algorithm. The algorithm works by incrementally constructing the lower convex hull from a coordinate-sorted set of points by adding one point at a time and removing a point if it makes a right-hand turn. Since the vertices were in sorted order before the projection, then the vertices will be in sorted order after the projection and flattening. The original vertices that correspond to the points on the lower convex hull are then used to create new edges.

Two new subdomains are then created and the original subdomain's vertices are then partitioned into the two new subdomains's vertices. For simplicity and without loss of generality, assume the cut axis is the y-axis. All vertices in the original subdomain which have an x-coordinate less than the median vertex's x-coordinate are added to the left subdomain while vertices with x-coordinates greater than the median vertex's x-coordinate are added to the right subdomain. Additionally, vertices that comprise the lower convex hull are added to both the left and right Subdomain object's vertices. This partitioning step is done by iterating over the original subdomain'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 sufficient work to facilitate good load balancing.

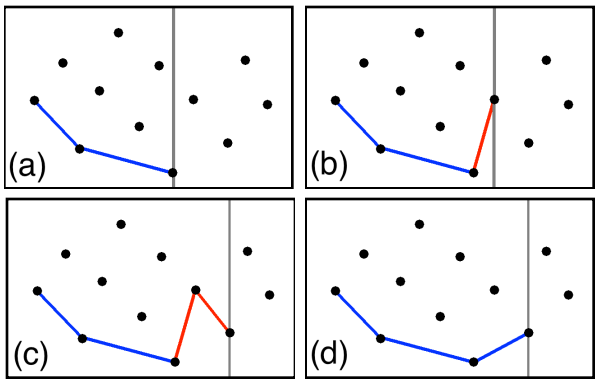

Figure 7. Steps of the Monotone Chain Algorithm. The vertical grey line sweeps from lowest x-coordinate vertex to highest x-coordinate vertex. (a) The current lower convex hull; (b) The previous lower convex hull with the next vertex added; (c) The next to last point of the lower convex hull makes a right-hand turn, so the next to last point must be removed as it is not part of the lower convex hull; (d) The current lower convex hull after the non-hull point is removed

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, whereas if there are no internal vertices (vertices not marked as being on the subdomain boundary), then the subdomain's decomposition is halted. We have added two more variable constraints as we are utilizing this approach as a coarse-partitioner: if the number of vertices is less than a given tolerance or if the decomposition's recursive level of a particular subdomain reaches a given tolerance, then decomposition ceases for this subdomain. The decomposition recursive level is dependent on the number of processes. Figure 8 shows the boundary layer decomposed into 128 Delaunay subdomains, which can all be triangulated independently.

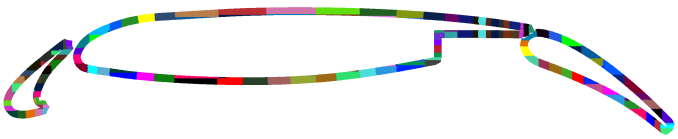

Figure 8. The boundary layer of the 30p30n airfoil decomposed into 128 Delaunay independent subdomains

E. Graded Isotropic Inviscid Region

For generating the isotropic inviscid region, we use a sizing function along with Triangle's ability to use a user-defined area constraint for Delaunay refinement to provide a smooth gradation of triangle size based on distance from the initial geometry towards the far-field. Since the size of the inviscid region is typically a factor of 30 to 50 chord lengths, the length of the airfoil, from the initial geometry in the x and y directions, the time to refine the inviscid region is extremely high, due to the exponential growing area. Thus, we need to generate the inviscid region in parallel. To facilitate the parallel refinement, we follow the approach of generating graded Delaunay decoupling paths as presented in [5], in order to distribute the refinement work among processes by creating subdomains which can be concurrently refined. 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. 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 [22], 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 worst-shaped elements that may be created. In order to generate subdomains that can be efficiently triangulated and refined by Triangle, it is essential to create subdomains that are convex and do not contain any holes since Triangle first creates an initial triangulation and then removes elements inside concavities and holes from the initial triangulation. The inviscid region is initially decoupled into four quadrants shown in Figure 9, where the solid region is the near-body subdomain which contains the airfoil.

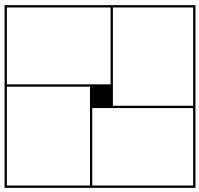

Figure 9. Initial four quadrants for decoupling

In order to generate the graded Delaunay decoupling path of the four initial subdomains, for each subdomain we compute a value k, an edge length size based on the termination conditions of Delaunay refinement that will be used for the decoupling procedure, from (1), where A is the area of the desired element at the given vertex, found by evaluating the sizing function, for the corner vertex that is shared with the near-body subdomain. We march along the shared borders between subdomains towards the farfield and then around the outer border to insert new vertices.

$$k = \frac{1}{2}\sqrt{\frac{A}{\sqrt{2}}} \tag{1}$$

Using the initial k value, $k_{current}$, at the corner vertex, $v_{current}$, a new point, v_{next}, can be created $2k_{current}/\sqrt{3} \leq D < 2k_{current}$ units from the current vertex. Assume $k_{current} \leq k_{next}$ for the k values at $v_{current}$ and v_{next} respectively. The following must hold true, $D < 2k_{next}$, to ensure that the Delaunay property be maintained. If this inequality is not true, then v_{next} needs to moved closer to satisfy the inequality. Then $v_{current}$ becomes v_{next} and the iteration continues. Four processes receive one of the initial quadrants for further decoupling.

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 point is inserted at the center of a subdomain and a path is created from the center point to each of the four midpoints along the sides of the subdomain. This creates four new subdomains to replace the old subdomain. New points are not added to the outer border of the subdomain; instead, the decoupling path connects to an already existing point that is closest to the midpoint of the outer border. This eliminates the need for communication between processes since new points are only inserted within the interior of a

subdomain, thus the shared borders between other subdomains are undisturbed. The decoupling process is recursive and subdomains are repeatedly decoupled and sent to other processes until all processes have sufficient work to facilitate good load balancing. 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 10 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.

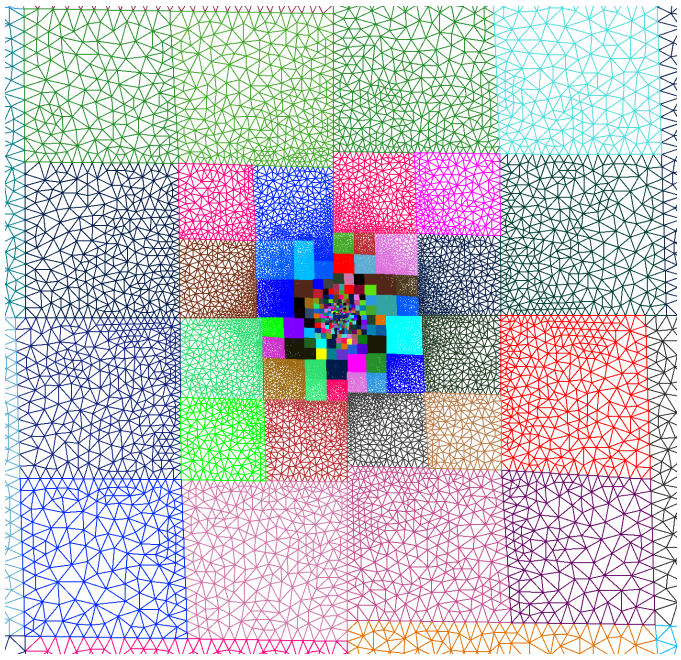

Figure 10. Decoupled Delaunay subdomains

F. *Load Balancing*

Each process starts with a subset of the domain, decomposed or decoupled into subdomains. The borders of each subdomain are fixed and consistent between neighboring 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 the Implementation section, is allocated on the root process as an array that will hold the work load estimates for each process. Each process will periodically update it's work load estimate. Whenever a process' work load falls below a threshold, it will fetch the global memory window and compute which process has the most work so that it can request work from this process.

III. IMPLEMENTATION

The software standard used for our implementation is the Message Passing Interface (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 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 are used to facilitate remote memory accesses (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. However, the hardware needs to support this type of communication to reap the benefits of RMA. The cluster used for our evaluations supports RMA through Infiniband. 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.

Practical data structure and algorithm designs must be utilized in order to achieve a scalable and cache-friendly distributed memory, parallel application. A contiguous memory container is used for storing the Vertex objects in order 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 points 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.

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. However, the

vertices of the lower convex hull are not represented completely in the left and right subdomains. In order to include the vertices in the lower convex hull, the vertices with an x-coordinate less than the median vertex's x-coordinate are placed at the front of the right subdomain's x-sorted vertices while the vertices with an x-coordinate greater than or equal to the median vertex's x-coordinate are placed at the end of the left subdomain's x-sorted vertices. These lower convex hull vertices are actually added to the right subdomain's x-sorted vertices before the original subdomain's x-sorted vertices to avoid the cost of reshuffling the vertices since we are using contiguous storage. Additionally, the data for the original subdomain is reused for the left subdomain. Not only does this eliminate the cost of deallocation for the original subdomain and allocation for the left subdomain, but for the task of partitioning the primary-axis-sorted vertices, the vertices before the median vertex are already in the proper location, which eliminates half of the data moving costs.

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

IV. Results & Evaluation

We executed our application on a cluster with 32 nodes with a 4X FDR Infiniband interconnect with RMA support (approximately 56 Gbit/sec) dual socket 8 core 2.60 GHz Intel E5-2670 Sandybridge (2 x 20MB cache) with 32 GB RAM per node. Each MPI rank was assigned to a core. We measured the speed up, the ratio of the execution time of the fastest sequential algorithm (Triangle in the two-dimensional case) to the execution time of the parallel algorithm; and efficiency, 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. Figure 11 shows good strong scalability up to 256 processes with a speedup of approximately 180 while 128 processes can achieve a speedup of approximately 102. Figure 12 shows the efficiency is roughly 70% for 256 processes and 80% for 128 processes. Since our application uses Triangle for the boundary layer triangulation and inviscid region refinement, our application's running time using one process is almost equivalent to the execution time of Triangle. The sequential meshing time of Triangle was 192 seconds while the sequential meshing time of our application was 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 inviscid 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 upon 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.

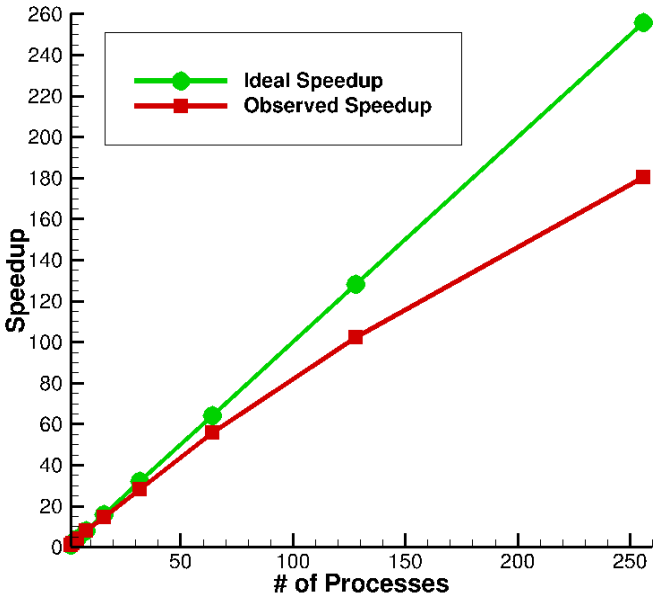

Figure 11. Strong scalability up to 256 processes with a fixed mesh size of 172,768,335 triangles

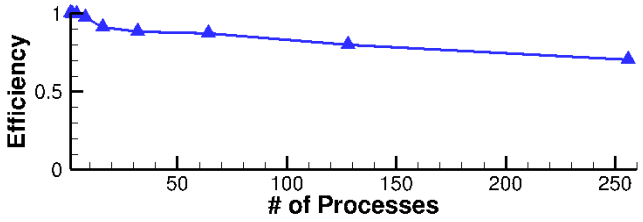

Figure 12. Efficiency of the strong scalability measurement up to 256 processes with a fixed mesh size of 172,768,335 triangles

Our algorithm is efficient in terms of communication and algorithmic performance. For the decoupled inviscid region, we store the points 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 CPU cycles 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 CPU cycles 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 points that are needed by Triangle. This also reduces communication costs for transferring a boundary layer subdomain that has already been sufficiently decomposed.

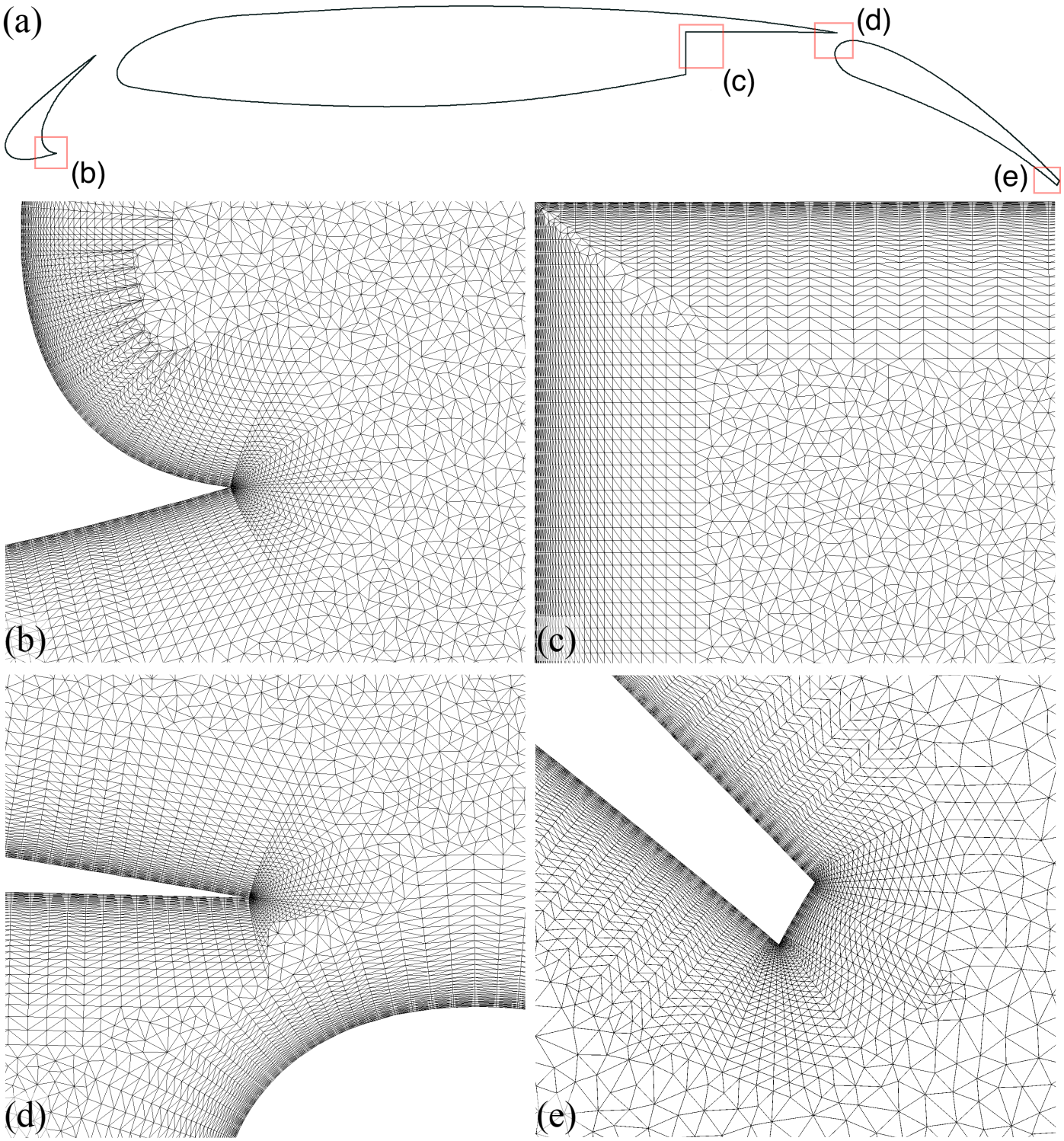

Figure 13. (a) 30p30n airfoil with highlighted regions; (b) Resolved self intersection at a concavity and curved fan of rays at the trailing edge on the leading slat; (c) Resolved self intersection at a 90 degree concave corner on the main slat; (d) Resolved multi-element intersection at the curved fans of rays at the trailing edge of the main slat and leading edge of the trailing slat; (e) Blunt trailing edge with new rays added around the two discontinuities of the trailing slat

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 points, compared to the inviscid subdomains which only have points on their border, 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.

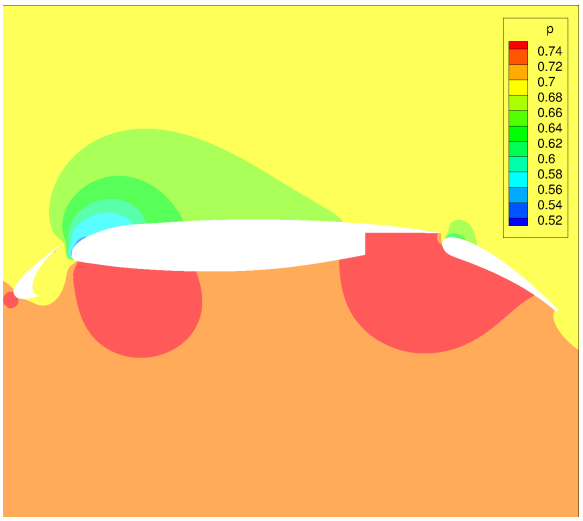

Figure 14. Flow solution for pressure for the 30p30n airfoil

The 30p30n multi-element airfoil in Figure 13 shows the different regions of interest and the resulting meshes for: self intersections, multi-element intersections, sharp trailing edge, and blunt trailing edge. Figure 14 shows the output of FUN3D [25], the flow solver used for our simulations, for pressure while Figure 15 shows the output for mach speed for the same simulation. The simulation was run with a mach number of 0.3, Reynolds number of 1 million, and angle of attack of 5 degrees. For Figure 14 we have a high pressure underneath and low pressure on top resulting in high lift. For Figure 15 the streamtraces show vortices at the leading slat cavity, the main slat cavity, and where the fluid separates at the trailing slat. The streamtraces also show the stagnation points on the underside of each slat of the airfoil at areas with low mach speeds which causes the fluid to split due to the angle of attack. There is also a region with a high mach number towards the upper side of the leading edge of the main slat due to the stagnation points on the leading and main slat which cause the fluid to accelerate through the small gap between the leading and main slat.

Figure 16 shows the convergence of the solution using FUN3D for the conservation of mass equation. We ran one simulation using a mesh for the 30p30n airfoil generated with our parallel anisotropic algorithm, denoted with the red square markers and "Anisotropic" label. The mesh with anisotropic boundary layers contained 360,241 triangles. The other simulation was run using a mesh for the 30p30n airfoil generated with Triangle using isotropic triangles, denoted with the green circle markers and "Isotropic" label. The isotropic mesh contains 5,314,372 triangles and was generated with Triangle using the quality switch so that all angles were above 20.7 degrees. The isotropic mesh contains over 14 times more elements and still took longer for its solution to converge. Both simulations were run using the same parameters. Both meshes were generated using the same sizing function for gradation and the same initial surface distribution of points. The anisotropic mesh solution converges by reaching a stopping tolerance of 10^{-12} around 5,000 iterations while the isotropic mesh solution takes around 10,000 iterations to converge.

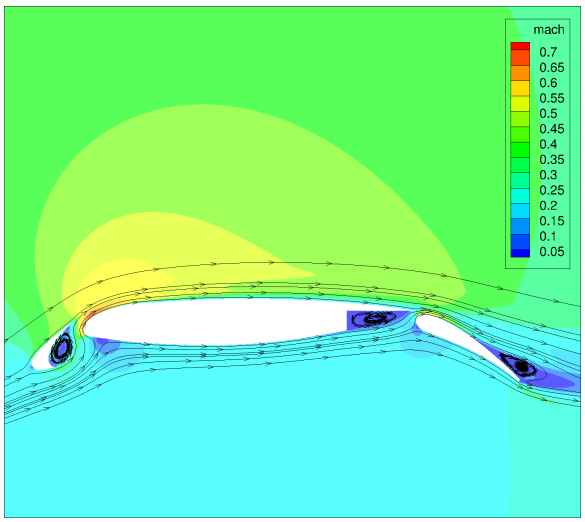

Figure 15. Flow solution for mach speed for the 30p30n airfoil

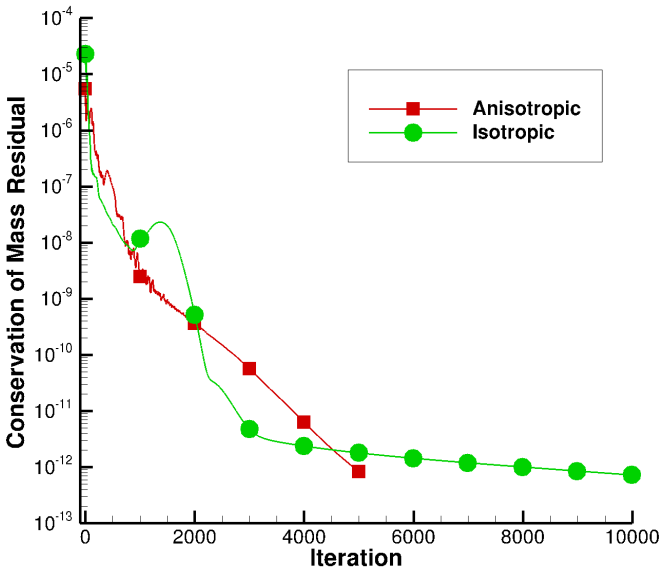

Figure 16. Convergence of the solution for the conservation of mass equation for the 30p30n airfoil using anisotropic triangles and isotropic triangles

V. Conclusion

We have developed the framework and a practical implementation to generate high-quality, two-dimensional unstructured initial meshes for complex domains in parallel for use in CFD simulations on distributed-memory machines. We showed that our anisotropic meshes contained fewer elements and reached solution convergence in fewer iterations than their corresponding isotropic meshes. The application is a push-button application, meaning that the user only needs to specify the input geometry and boundary layer parameters to start the program, then momentarily wait for the resulting mesh without having to further interact with the application. Additionally, the application has three 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 CFD 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. The evaluation of our approach on larger clusters is still a work in progress. Since observing that our approach is feasible for two-dimensional meshes, we plan to extend our approach to generate high-quality three-dimensional meshes in parallel. However, our approach is beneficial even in two-dimensional cases by providing a fast parallel, push-button application to facilitate aerospace development with rapid turnaround time.

Acknowledgment

This work was supported (in part) by the National Institute of Aerospace (http://www.nianet.org) and 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. We thank Mike Park for his discussions and comments.

References

[1] R.V. Garimella and M.S. Shephard, "Boundary Layer Mesh Generation for Viscous Flow Simulations," International Journal for Numerical Methods in Engineering, vol. 49, 2000, pp. 193-218.

[2] G.E. Blelloch, G.L. Miller, and D. Talmor, "Developing a Practical Projection-Based Parallel Delaunay Algorithm," Proc. 12th Annual Symposium on Computational Geometry, 1996, pp. 186-195.

[3] J.R. Shewchuk, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," Applied Computational Geometry: Towards Geometric Engineering vol. 1148, 1996, pp. 203-222.

[4] A.M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions," Information Processing Letters, vol. 9, 1979, pp. 216-219.

[5] L. Linardakis and N. Chrisochoides, "Graded Delaunay Decoupling Method for Parallel Guaranteed Quality Planar Mesh Generation," SIAM Journal on Scientific Computing, vol. 30, 2008, pp. 1875-1891.

[6] A. Van Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips, Introduction to Computer Graphics, vol. 55, 1994, pp 113.

[7] L.P. Chew, N. Chrisochoides, and F. Sukup, "Parallel Constrained Delaunay Meshing," Proc. Symposium on Trends in Unstructured Mesh Generation, 1997, pp. 89-96.

[8] A. Loseille, D. Marcum, and F. Alauzet, "Alignement and Orthogonality in Anisotropic Metric-Based Mesh Adaption," Proc. 53rd AIAA Computational Fluid Dynamics Conference, 2015.

[9] R. Aubry, K. Karamete, E. Mestreau, D. Gayman, and S. Dey, "Ensuring a Smooth Transition from Semi-Structured Surface Boundary Layer Mesh to Fully Unstructured Anisotropic Surface Mesh," Proc. 53rd AIAA Computational Fluid Dynamics Conference, 2015.

[10] A. Chernikov and N. Chrisochoides, "Algorithm 872: Parallel 2D Constrained Delaunay Mesh Generation," ACM Transactions on Mathematical Software, vol. 34, 2008, pp. 6-25.

[11] A. Chernikov and N. Chrisochoides, "Parallel Guaranteed Quality Delaunay Uniform Mesh Refinement," SIAM Journal on Scientific Computing, vol. 28, 2006, pp. 1907-1926.

[12] R. Zhang, K.P. Lam, and Y. Zhang, "Conformal and Adaptive Hexahedral-Dominant Mesh Generation for CFD Simulation of Architecture Applications," Proc. Winter Simulation Conference, 2011.

[13] Y. Ito, A. Shih, A. Erukala, B. Soni, A. Chernikov, N. Chrisochoides, and K. Nakahashi, "Parallel Unstructured Mesh Generation Using an Advancing Front Method," Mathematics and Computers in Simulation, vol. 75, 2007, pp. 200-209.

[14] G. Globisch, "PARMESH – A Parallel Mesh Generator," Parallel Computing, vol. 21, 1995, pp. 509-524.

[15] N. Chrisochoides and D. Nave, "Parallel Delaunay Mesh Generation Kernel," International Journal for Numerical Methods in Engineering, vol. 58, 2003, pp. 161-176.

[16] C. Kadow, "Parallel Delaunay Refinement Mesh Generation," Ph.D. thesis, Carnegie Mellon University, 2004.

[17] XFOIL, Subsonic Airfoil Development System, http://web.mit.edu/drela/Public/web/xfoil.

[18] MSES, Multielement Airfoil Design/Analysis System, http://raphael.mit.edu/drela/msessum.ps.

[19] L. Lammera and M. Burghardt, "Parallel Generation of Triangular and Quadrilateral Meshes," Advances in Engineering Software, vol. 31, 2000, pp. 929-936.

[20] A.I. Khan and B.H.V. Topping, "Parallel Adaptive Mesh Generation," Computing Systems in Engineering, vol. 2, 1991, pp. 75-101.

[21] J. Bonet and J. Peraire, "An Alternating Digital Tree (ADT) Algorithm for 3D Geometric Searching and Intersection Problems," International Journal for Numerical Methods in Engineering, vol. 31, 1991, pp. 1–17.

[22] J. Ruppert, "A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation," Journal of Algorithms, vol. 18, 1995, pp. 548-585.

[23] A. Chernikov and N. Chrisochoides, "Three-Dimensional Delaunay Refinement for Multi-Core Processors," Proc. ACM International Conference on Supercomputing, 2008, pp. 214-224.

[24] P. Foteinos, "High Quality Real-Time Image-to-Mesh Conversion for Finite Element Simulations," Journal on Parallel and Distributed Computing, vol. 74, 2014, pp. 2123-2140.

[25] R. Biedron, J. Carlson, J. Derlaga, P. Gnoffo, D. Hammond, W. Jones, W. Kleb, E. Lee-Rausch, E. Nielsen, M. Park, C. Rumsey, J. Thomas, and W. Wood, "FUN3D Manual: 12.8," 2015, NASA/TM-2015-218807.