SAFT: Shotgun advancing front technique for massively parallel mesh generation on graphics processing unit

Qingyi Zhou1 | Qiqi Wang2 | Zongfu Yu1

1Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, Wisconsin, USA
2Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA

Correspondence
Qingyi Zhou, Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, WI 53706, USA.
Email: qzhou75@wisc.edu

Funding information
College of Engineering, University of Wisconsin-Madison

Abstract
Large-scale numerical simulations need efficient parallel mesh generation schemes. Several parallel advancing front algorithms were proposed in the past decades, most of which require domain decomposition. In this article, we present a shotgun algorithm for parallel advancing front mesh generation. Our algorithm is front-based, therefore does not require domain decomposition. We’ve implemented the algorithm on GPU, which has thousands of CUDA cores. Different from traditional volume-based parallelization, each CUDA thread handles one face at a time. We deal with conflicts by discarding illegal new elements which intersect with each other. We name this proposed method “SAFT”, which stands for “shotgun advancing front technique”. Its performance, as well as scalability, has been evaluated on a laptop equipped with one NVIDIA GeForce RTX2060 graphics card. We have been able to generate high-quality 2D meshes efficiently (∼233k elements per second).

KEYWORDS
advancing front, mesh generation, parallel computing

1 | INTRODUCTION

Meshtes are the geometrical support for finite element/finite volume solvers. During the past decades, to ensure reliable numerical simulations in industrial applications, the size of meshes has been growing steadily. Nowadays, with the increased availability of parallel numerical solvers, the number of elements used in meshes can often reach billions. Such large meshes are commonly used for computational fluid dynamics (CFD) and computational electromagnetics (CEM). It often takes hours to generate such meshes even with state-of-the-art computing devices, which has increasingly become the bottleneck of simulation workflow.

While many solvers have been ported to distributed parallel machines, parallel mesh generators have lagged behind. However, for applications where remeshing is conducted constantly, for example, problems with moving bodies, the time required for mesh regeneration can consume a significant percentage of the total time. Faced with such difficulties, some efforts have been reported on parallel mesh generation. The two most common ways of generating unstructured triangular/tetrahedral mesh are advancing front technique (AFT) and Delaunay triangulation (DT). Both techniques involve an iterative process: AFT introduces one new element at a time, while DT introduces one new vertex at a time. In this article, we will mainly focus on parallel AFT.
While the original version of AFT is scalar in nature, parallelizing AFT can be quite challenging. Previous works\(^1\text{-}^4\) utilize computer clusters and involve hundreds of CPU cores working together. The limited number of threads makes it difficult to realize massively parallel mesh generation. More importantly, previous works require domain decomposition: the original domain must first be decomposed into sub-domains (often using graph-partitioning software such as METIS\(^5\)), then each sub-domain is handled by scalar AFT. Inside each domain, the calculations are carried out sequentially. The quality of domain decomposition depends on the complexity of the geometry. High-quality domain decomposition often requires human intervention, making it difficult to develop a fully automated workflow.

The development of graphics processing units (GPUs) with thousands of CUDA cores has led to a re-evaluation of parallel mesh generation/processing options.\(^6\) Although CPUs have also been developing fast, it remains difficult to gain access to high-performance computing clusters. GPUs, on the other hand, are already widely available on most personal computers, thus serve as ideal sites for developing and testing parallel algorithms. Provided that a parallel AFT based on GPU is developed, large-scale mesh generation can be realized easily on a single personal computer, rather than occupying a large number of machines in a computer cluster. Instead of volume-based parallelization, front-based parallelization can be realized using GPU, where each thread handles one face at a time. After collecting all the new elements generated, the intersection between all these elements can be calculated. Only a portion of new elements will be kept to ensure they do not intersect with each other, therefore conflict can be avoided.

In this article, based on the above ideas, we have proposed and implemented a parallel mesh generation algorithm by modifying scalar AFT. Our algorithm is named “SAFT,” which stands for “shotgun advancing front technique.” This name has been inspired by the shotgun sequencing method, in which long DNA sequences are divided into smaller fragments and can be sequenced in parallel. The algorithm is implemented using CUDA C++ (code available at https://github.com/zhouqingyi616/SAFT). The performance, as well as scalability of our proposed SAFT, is demonstrated on an NVIDIA Geforce RTX2060 graphics card, together with an Intel Core i7-9750 CPU (at 2.60 GHz). We have been able to generate 72.7M triangular elements in 7 min (≈176k elements per second). If tested with a smaller case containing 9.0M elements, an even higher generation speed of 233k elements per second can be reached.

2 | Algorithm

2.1 | Overview

For the sake of clarity, we first recall the main process of scalar AFT as follows:\(^7\) one starts from a topologically consistent triangulation of the domain boundary, which serves as the initial front.

While there are active faces left in the front, the following four steps are carried out sequentially to generate one new element:

1. Choose one face to grow from (often the smallest one).
2. Select a “best point” location based on the normal direction of that face.
3. Determine several possible locations where a new vertex can be placed. All these vertices are candidates at this moment.
4. Only the candidate which leads to the best element quality will be kept as the new vertex, while others are simply discarded.
5. Update active front: record new element, discard archived faces.

Researchers have been focusing on parallel AFT for a long time. Löhner et al.\(^8\) introduced parallel AFT over two decades ago. Some more recent works are prompted by the rise of shared-memory machines with hundreds of processors.\(^2\text{-}^4\) All the works mentioned above are based on some sort of domain decomposition, meaning that one needs to define the regions to be meshed by each processor in advance. However, domain decomposition leads to extra overhead and makes parallel AFT quite complicated.

Here we would like to develop a parallel AFT algorithm that does not require domain decomposition. Without domain decomposition, the parallelization must be front-based, meaning that each thread handles a small portion of faces in
the current front. If two faces are far from each other, growing new elements from these two faces can be considered as two completely independent tasks, which can be carried out concurrently. This important feature of AFT can lead to fine-grained parallelism, which facilitates load balancing. The whole mesh generation process can be divided into many independent tasks, whose number is equivalent to the number of faces contained in the current front. Considering that each task (finding a new element for a single face) is very small, it is more reasonable to use CUDA cores instead of CPU cores, because of the large number of CUDA cores available in GPUs. Modern GPUs often have thousands of CUDA cores, and can therefore support many threads working concurrently.

In order to utilize the power of GPU, we have modified the process of scalar AFT, as shown in the box below. Denote the number of threads as \( P \) (we would expect \( P \) new elements to be generated during a single iteration). The process of SAFT is as follows: one starts from a topologically consistent triangulation of the domain boundary, which serves as the initial front.

While there are active faces left in the front, the following eight steps are carried out during a single iteration:

1. Randomly choose \( P \) faces to grow from, and each thread handles exactly one face.
2. Each thread collects geometric information of neighboring faces.
3. Each thread selects a “best point” location based on the normal direction of the corresponding face.
4. Based on the information collected in step 2, each thread determines whether a point on neighboring faces exists, that should be used as a new vertex.
5. All \( P \) new elements (at most) are saved inside an array, which is copied and sent back to the CPU. These new elements are referred to as “candidates.”
6. Calculate intersections between \( P \) candidates, using \( P \) threads.
7. Based on step 6, only a portion of \( P \) candidates, which do not intersect each other, are chosen. These are all new elements we have created successfully in this iteration.
8. Update active front: record new elements, discard archived faces.

### 2.2 Detailed process

In this part, we discuss the major steps of the proposed SAFT algorithm in detail. The algorithm works in an iterative manner, and we shall focus on one single iteration (as shown in Figure 1), during which \( P \) new elements (at most) are generated. Note that although our final implementation involves only 2D cases, the proposed algorithm can be readily applied to 3D cases, and we use “face” instead of “edge” throughout the article to avoid confusion.

Denote the \( i \)th thread as \( t_i \), where \( i = 1, 2, \ldots, P \). Denote the “front” (or “active front”), which is the set containing all active faces, as \( F \). The number of faces contained in the current active front is therefore denoted as \( N_F \).

#### 2.2.1 Distribute faces

All front faces are saved inside a quadtree, which is denoted as \( QT(F) \). Each node of the quadtree contains one face at most, and only those nodes that contain front faces are marked as “active”. As the first step, \( P \) faces are chosen randomly from the current front: \( s_1, \ldots, s_i, \ldots, s_P \in F \). Each thread \( t_i \) handles exactly one face \( s_i \).

In order to prevent generated new elements from intersecting with the current front \( F \), information of nearby faces must be collected. In order to achieve this, each thread \( t_i \) needs to search over the quadtree, and find those faces that are close to face \( s_i \). More specifically, \( t_i \) searches over quadtree, checking all active nodes, which contain front faces. The set of faces which need to be considered by \( t_i \) is denoted as \( F^{(i)} \). The criterion for judging whether an arbitrary face \( s \) should belong to \( F^{(i)} \) is as follows: face \( s \) needs to be considered by \( t_i \) if and only if the distance between its middle point \( m_s \) and \( s_i \)'s middle point \( m_{s_i} \) is smaller than radius \( r_{add} \):

\[
\forall s \in F, s \in F^{(i)}, \text{ iff dist}(m_s, m_{s_i}) \leq r_{add}.
\]
The general process of SAFT. Each thread $t_i$ handles a single face $s_i$. After calculating the intersections between generated candidates, only a portion of them will be kept. Finally, the front needs to be updated.

The element size at $s_i$'s middle point $m_{s_i}$ is denoted as $h_i$ (which can be calculated using some user-defined function), while the length of $s_i$ is denoted as $l_{s_i}$. In our experiments, we choose the radius $r_{\text{add}}$ to be

$$r_{\text{add}} = 1.5 \cdot \max\{h_i, l_{s_i}\}.$$  \hspace{1cm} (2)

Once $F^{(i)}$ has been determined, thread $t_i$ can continue to try to find new vertex $v_i$, which will form a new element $e_i$ together with face $s_i$.

2.2.2 Find new elements

Each thread $t_i$ needs to find an optimal position for placing the new vertex $v_i$. New vertex can be an entirely new vertex (on the perpendicular bisector of face $s_i$), or it can be a vertex lying on existing front $F$. Thus finding an optimal position for a new vertex requires three steps:

1. Finding possible locations for new vertex on $s_i$'s perpendicular bisector.
2. Finding possible locations on neighboring faces.
3. Calculate the quality of the newly generated element, and choose the vertex which leads to the best element quality.

Before looking for $v_i$'s possible locations, we first have to determine the ideal position of the new vertex to form a new triangle. This ideal position is denoted as $c_i$. Suppose the element size $h_i$ is already known. Since the ideal position lies on $s_i$'s perpendicular bisector, we simply need to calculate the waist $w_i$:

$$w_i = \begin{cases} 
0.55 \cdot l_{s_i}, & \text{if } 0.55 \cdot l_{s_i} > h_i, \\
\frac{l_{s_i}}{0.55}, & \text{if } l_{s_i}/0.55 < h_i, \\
h_i, & \text{otherwise}. 
\end{cases} \hspace{1cm} (3)$$

Thus the position of $c_i$ can be easily determined, since its distance toward both vertices of face $s_i$ are $w_i$.

Now based on the ideal position $c_i$, the above three steps will be discussed in detail.
FIGURE 2 Find possible locations for new vertex on $s_i$’s perpendicular bisector. New faces must not intersect with existing faces.

FIGURE 3 Two cases that have to be prevented, even though new faces $s_1^i$ and $s_2^i$ do not intersect with current front directly. (A) The current front, drawn in black. We would like to grow a new element from orange face. (B) An illegal way to place the new vertex on the perpendicular bisector. (C) An illegal way to place the new vertex on the current front

New vertices on perpendicular bisector

As shown in Figure 2, a collection of $L_{\text{mid}}$ points are generated along the straight line between ideal position $c_i$ and middle point $m_i$. The set of these points are denoted by $B = \{b_1, \ldots, b_{L_{\text{mid}}}\}$. The exact position of $b_j (j = 1, 2, \ldots, L_{\text{mid}})$ is a linear combination of point $c_i$ and $m_i$:

$$b_j = \beta_{\text{mid}}^{j-1} \cdot c_i + (1 - \beta_{\text{mid}}^{j-1}) \cdot m_i,$$

(4)

where $\beta_{\text{mid}}$ is just a parameter controlling the coefficient of this linear combination. In this article, we choose $L_{\text{mid}} = 4$ and $\beta_{\text{mid}} = 0.85$.

These candidates $b_1, \ldots, b_{L_{\text{mid}}}$ must be checked carefully to ensure that the generated element is legal. Consider the element formed by face $s_i$ and some certain vertex $b \in B$. The two newly formed faces is denoted as $s_1^i$ and $s_2^i$. The angle between $s_i$ and new faces $s_1^i$, $s_2^i$ is denoted as $\theta_1^i$, $\theta_2^i$. The vertex $b \in B$ is considered “illegal” if one of the following four conditions is met:

1. $s_1^i$ or $s_2^i$ has intersection with some existing front face $s \in F^{(i)}$.
2. The angle between $s_i$ and its neighboring face $s_i'$ is smaller than $\theta_1^i$ or $\theta_2^i$. This criteria is used to prevent the case shown in Figure 3B from happening, where the two new faces $s_1^i$, $s_2^i$ have no direct intersection with current front $F$.
3. The new vertex $b$ is too close to some existing front face $s \in F^{(i)}$. Here the distance threshold is chosen to be $0.4 \cdot h_i$.
4. Some existing vertex $v_e$ is too close to $s_1^i$ or $s_2^i$. The distance threshold is chosen to be $0.35 \cdot l_{i1}$ or $0.35 \cdot l_{i2}$.
Find candidate positions for new vertex

\[ \text{Indirect intersection} \quad \text{Direct intersection} \quad \text{Form small angle} \]

Legal new vertex
Legal new vertex
Keep the element with best quality

**FIGURE 4** Find possible locations for new vertex on existing faces, which are not far from \( s_i \). New faces must not intersect with existing faces.

**New vertices on existing faces**

New vertex can also fall on existing faces. All existing vertices contained in set \( F_i \) are checked, as shown in Figure 4. The set of these points are denoted by \( N = \{ n_1, n_2, \ldots, n_{n_n} \} \). The criterion for judging whether an existing vertex \( n \) should belong to \( N \) is as follows: vertex \( n \) needs to be considered as candidate if and only if the distance between \( n \) and ideal position \( c_i \) is smaller than radius \( r_n \):

\[ \forall n, n \in N, \text{ iff } \text{dist}(n, c_i) \leq r_n. \quad \text{(5)} \]

In our experiments, we choose the radius \( r_n \) to be \( r_n = 1.5 \cdot w_i \), where \( w_i \) is the length of waist, which has already been derived using Equation (3).

Similar to the previous part, these candidates \( n_1, n_2, \ldots, n_{n_n} \) must be checked. Consider the element formed by face \( s_i \) and some certain vertex \( n \in N \). The two newly formed faces are \( s_{i1}^j \) and \( s_{i1}^j \). The angle between \( s_i \) and new faces \( s_{i1}^j \), \( s_{i1}^j \) are denoted as \( \theta_{1i} \), \( \theta_{2i} \). The vertex \( n \) is considered “illegal” if one of the following four conditions is met:

1. \( s_{i1}^j \) or \( s_{i1}^j \) has intersection with some existing front face \( s \in F^{(i)} \).
2. The angle between \( s_i \) and its neighboring face \( s_{i1}^j \) is smaller than \( \theta_{1i} \) or \( \theta_{2i} \). This criteria is used to prevent the case shown in Figure 3C from happening, where \( s_{i1}^j \), \( s_{i1}^j \) have no direct intersection with current front \( F \).
3. \( s_{i1}^j \) or \( s_{i1}^j \), together with some existing face \( s \in F^{(i)} \), forms small angle, which is smaller than threshold \( \theta_{\min} \). Here the angle threshold is chosen to be \( \theta_{\min} = 6^\circ \).
4. Some existing vertex \( v_i \) is too close to \( s_{i1}^j \) or \( s_{i1}^j \). The distance threshold is chosen to be \( 0.09 \cdot h_i \).

**Quality measure**

All legal vertices contained in set \( B = \{ b_1, \ldots, b_{b_{\text{mid}}} \} \) and \( N = \{ n_1, n_2, \ldots, n_{n_n} \} \) can now be collected into a new set \( V \). Now we need to go over all candidates \( v \in V \), and find the best candidate \( v^* \), which leads to a new element with best quality. Denote the two newly formed faces as \( s_{i1}^j \) and \( s_{i1}^j \). We now describe the derivation of newly formed element’s quality. First define \( \delta_1 \), \( \delta_2 \) as

\[ \delta_1 = \min\{ l_{i1}^j / h_i, h_i / l_{i1}^j \}, \quad \text{(6)} \]

\[ \delta_2 = \min\{ l_{i2}^j / h_i, h_i / l_{i2}^j \}. \quad \text{(7)} \]

Then define \( \alpha \) as

\[ \alpha = \frac{S^\Delta}{l_{i1}^j + l_{i2}^j + l_{i2}^j} \quad \text{(8)} \]
where \( S_\Delta \) denotes the area of this new element. Finally, the quality \( Q(v, s_i) \) can be derived as
\[
Q(v, s_i) = \alpha \cdot \delta_1 \cdot \delta_2.
\]  
(9)

Only the candidate \( v_i^* \) with best quality will be kept:
\[
v_i^* = \arg \max_{v \in V} Q(v, s_i),
\]  
(10)

and this \( v_i^* \), together with face \( s_i \), forms the new element \( e_i \).

2.2.3 Calculate intersection

While \( P \) new elements (at most) have already been obtained, some of these elements have to be discarded since they intersect each other and lead to conflicts. The first step is to calculate intersections between all \( P \) elements, which leads to a \( P \times P \) matrix \( I \). Here \( I_{ij} \) records whether candidate \( e_i \) intersects with candidate \( e_j \):
\[
I_{ij} = \begin{cases} 
1, & \text{if } i \text{ intersects with } j, \\
0, & \text{otherwise}. 
\end{cases}
\]  
(11)

Note that in our implementation, in order to reduce computation time, we do not directly calculate intersections between different elements, which can be quite expensive. Instead, a bounding box is created for each candidate. If the distance between two bounding boxes in both \( x \) and \( y \) directions is smaller than a certain threshold (which is \( 0.25(h_i + h_j) \), where \( h_i \) and \( h_j \) denotes element size), these two candidates are considered to "intersect" with each other. Also, if the distance between two elements \( e_i \) and \( e_j \) is larger than \( 2.5(h_i + h_j) \) (which is five times the average element size), the entire judgment is skipped. This criterion is inspired by Chernikov and Chrisochoides: they showed that in 2D DT, if the distance between two Steiner vertices is larger than \( 4\overline{r} \), then their cavities do not share any edge. Here \( \overline{r} \) represents an upper bound on the maximum circumradius in the mesh.

2.2.4 Choose legal elements

Based on the intersection matrix \( I \) obtained in the previous step, a portion of \( P \) candidates can be chosen. Specifically, \( i \)th and \( j \)th candidate cannot be chosen simultaneously if they intersect with each other \( (I_{ij} = 1) \). We have to determine a sub-set \( E_{\text{sub}} \) of \( E \), so that
\[
\forall e_i, e_j \in E_{\text{sub}}, \ I_{ij} = 0.
\]  
(12)

Here we choose legal candidates in a simple manner: we need to go over all candidate elements inside \( E \). If this element \( e_i \) has no intersection with all elements \( e_j \) satisfying \( j < i \), then this element can be added into \( E_{\text{sub}} \). Otherwise, \( e_i \) is not legal and should be discarded. Considering that all required information has been collected during calculating intersections, the time consumption of this "choose legal elements" step itself is almost negligible. In order to record the final results, we fill in a boolean array \( D = \{ d_1, \ldots, d_i, \ldots, d_P \} \). Here "\( d_i = \text{true} \)" means that new element \( e_i \) is legal and has been accepted, while "\( d_i = \text{false} \)" means \( e_i \) has been discarded.

2.2.5 Update active front

After all new elements have been determined, the active front needs to be updated. Faces that no longer belong to active front should be deleted from the quadtree, while newly generated front faces are inserted into the tree. This step closely resembles the "update active front" step of scalar AFT. More specifically, consider a new element \( e_i \in E_{\text{sub}} \), meaning that \( e_i \) is a legal element and should be kept. Thus face \( s_i \) should be archived and excluded from current front \( F \). Now consider the other two faces of \( e_i \), denoted as \( s_{1i} \) and \( s_{2i} \). They should be added into \( F \) if they do not already exist in \( F \). Otherwise, if \( s_{1j} \ (j = 1, 2) \) already exists in \( F \), which means new element \( e_i \) connects with current front, \( s_{1j} \) should be archived.
Note that in our implementation, to reduce computation time, we simply archive a face by marking its corresponding node as “inactive.” However, in this way the number of nodes contained in quadtree QT(F) will only increase because we do not delete any node. To solve this problem, we reconstruct the entire quadtree after every 20 iterations, thus the number of tree nodes cannot be significantly larger than the number of front faces NF.

2.3 Time complexity analysis

In this section, in order to gain a clear view of the algorithm’s scalability, we analyze the time complexity of each step in our proposed SAFT. Again we consider a single iteration. Denote the total number of faces contained in the initial front F as N. For a 2D problem, the total number of triangular elements that need to be generated is proportional to the area of the domain, which is proportional to $N^2$. Thus the total number of iterations required is proportional to $N^2/P$. Similarly, for a 3D problem, the number of tetrahedrons is proportional to the volume of the domain, which is proportional to $N^{1.5}$. Therefore the number of iterations required is proportional to $N^{1.5}/P$.

**Complexity of “Distribute faces”**
This step is fully parallelized: each thread $t_i$ searches over the quadtree and looks for faces that are close to $s_i$. When searching over the quadtree, if the bounding box of a certain tree node is far from $s_i$, then it’s not necessary to search over this node’s children. By making use of the quadtree data structure, the time complexity of finding a single face should be $O(\log N)$. Considering that the number of faces in $F(i)$ can be bounded by some constant, the time complexity of constructing $F(i)$ should still be $O(\log N)$.

**Complexity of “Find new elements”**
This step is fully parallelized, and again we look at a single thread $t_i$. Since $L_{mid}$ is a constant, the time complexity of looking for candidate vertices on bisector should be $O(1)$. On the other hand, considering that the number of faces in $F(i)$ can be bounded by some constant, the time complexity of looking for candidate vertices on the current front should also be $O(1)$. Therefore, the time complexity of “find new elements” is just $O(1)$. Note that this is often the most time-consuming step: verifying whether a new vertex is legal (check for intersections or small angles) requires many floating-point operations.

**Complexity of “Calculate intersection”**
The intersection matrix $I$ contains $P \times P$ elements. The derivation of these $P^2$ elements are completely independent, and thus can be fully parallelized theoretically. However, current GPUs have less than $10^8$ CUDA cores, while in our implementation a typical value of thread number is $P = 2048$. It is impossible to have $P^2$ threads running concurrently. In our implementation, considering the number of CUDA cores in our GPU, we choose to use $8P$ threads for this step. Each thread $t_i$ calculates $P/8$ matrix elements in $I$. Thus, the time complexity of calculating the $i$th row should be proportional to $P/8$, and the time complexity of this step should be $O(P)$. Fortunately, $P$ is not very large in our case: the typical value of $P$ is a few thousands. Thus this $O(P)$ does not lead to any problem in scalability, and this “calculate intersection” step is not the bottleneck of our algorithm.

**Complexity of “Choose legal elements”**
This step is not parallelized and is completed on the CPU, since a for-loop must be carried out over all these legal new elements. Inside the for-loop, we have to check whether new element $e_i$ intersects with some new element $e_j$ with $j < i$. Although theoretically, the time complexity should be $O(P)$, we should notice the fact that all the required information has already been obtained during the “calculate intersection” step. Therefore, in practice the time consumption of the “choose legal elements” step is negligible compared with that of other steps.

**Complexity of “Update active front”**
This step is not parallelized. That’s because modifying quadtrees, normally implemented in a recursive manner, can become difficult to handle in parallel. Therefore in the current implementation, we only modify quadtree on CPU. All legal new elements are inserted into quadtree QT(F). Each insertion requires $O(\log N)$ operations on average, and in total there are $P$ new elements (at most). Thus the time complexity of the “update active front” step should be $O(P \log N)$. 
3 | MESH GENERATION: PERFORMANCE

In this section, by giving several 2D examples, we demonstrate that SAFT is able to accomplish massively parallel mesh generation. In order to generate high-quality meshes, an optimization step has been added to the algorithm to improve the general quality of mesh elements. This is done by Laplacian smoothing, where each vertex (except for vertices lying on the boundary) is moved toward its neighbors’ center of mass.\(^\text{11}\)

3.1 | Small size test cases

In this part, small size 2D examples are used for demonstration. Two benchmarks are considered, namely:

1. A pulley together with a gear.
2. A violin.

We first mesh all boundaries with COMSOL Multiphysics 5.5. Mesh size on the boundaries is controlled by surface curvatures, while mesh size inside the domain is simply interpolated from the surface mesh. All meshes are generated on the same personal laptop, which is equipped with an NVIDIA GeForce RTX2060 graphics card and an Intel Core i7-9750 CPU (at 2.60 GHz), with threads number \(P = 128\).

In order to visualize the quality of generated triangular elements, we have calculated the equiangles skewness value of all elements. For an arbitrary triangular element, its skewness value \(\epsilon\) is defined as

\[
\epsilon = \max \left\{ \frac{\theta_{\text{max}} - \theta_e}{180^\circ - \theta_e}, \frac{\theta_e - \theta_{\text{min}}}{\theta_e} \right\},
\]

where \(\theta_{\text{max}}\) and \(\theta_{\text{min}}\) denote the maximum/minimum angle of the element, and \(\theta_e = 60^\circ\) since we are dealing with 2D triangular element.

Pulley and gear

The generated mesh, which contains 33,770 elements, is shown in Figure 5. The color of each triangular element represents the skewness value of that element. While Figure 5A shows the original mesh without smoothing, Figure 5B shows the final mesh after post-processed using Laplacian smoothing. It can be concluded that smoothing helps improve the mesh quality significantly.

Violin

For the 2D violin, the generated mesh, which contains 18,157 elements, is shown in Figure 6. While Figure 6A shows the original mesh without smoothing, Figure 6B shows the final mesh after post-processed using Laplacian smoothing. Again, Laplacian smoothing helps improve the mesh quality significantly.

3.2 | Large size test cases

In this part, we would like to test the scalability of our algorithm using 2D examples of larger size. Two benchmarks are considered, namely:

1. A larger violin.
2. A ring.

All meshes are generated on the same personal laptop, which is equipped with an NVIDIA GeForce RTX2060 graphics card and an Intel Core i7-9750 CPU (at 2.60 GHz). We have not only listed the total time required to finish the mesh generation, but also listed time consumptions of five different steps, namely:

1. “Find new”: Each thread collects information of nearby faces, and finds one optimal new element. This includes the “distribute faces” step described in Section 2.2.1, and the “find new elements” step described in Section 2.2.2.
2. “Intersect”: Each thread \( t_i \) calculates whether there are intersections between element \( e_i \) and other new elements. This is exactly the “calculate intersection” step described in Section 2.2.3.

3. “Choose”: The CPU chooses a legal subset from all the new elements returned by GPU. This is exactly the “choose legal elements” step described in Section 2.2.4.

4. “Update”: CPU records new faces and add them into front \( F \). The quadtree is also reconstructed, if necessary. This is exactly the “update active front” step described in Section 2.2.5.

5. “Move”: This represents moving data between CPU and GPU. This can be divided into two different parts: copying quadtree (which is saved using an array of tree nodes) from CPU to GPU, and copying new elements/faces from GPU back to CPU.

**FIGURE 5** Small test case: Pulley and gear. (A) The original mesh without Laplacian smoothing. (B) The final mesh, processed with Laplacian smoothing. The colorbars show the range of skewness value; smaller skewness indicates better element quality.
All timing results are measured in seconds. The results are averaged over 5 runs with different random seeds. Note that the “total time” does not include the final mesh improvement step, although this improvement is crucial for all mesh generation algorithms.

Besides the time consumption, for each single mesh generation process, we also check the “utility”, denoted as $u$. “Utility” stands for the ratio between the total number of generated elements and the maximum possible element number $P \cdot T$ (where $T$ stands for the total number of iterations):

$$u = \frac{N_{\text{elem}}}{P \cdot T}. \quad (14)$$

Utility measures whether all the $P$ threads are fully utilized during the mesh generation process. Since it is not possible to keep all $P$ new candidates generated in each iteration (due to intersection between candidates), utility $u$ is always smaller than 1. Ideally, $u \approx 1$, meaning that for most of the time in $T$ iterations, nearly all new elements found by all $P$ threads have been kept. Our algorithm is front-based, therefore when the number of threads is close to or larger than the number of remaining faces, utility $u$ tends to drop. At this point, increasing the number of threads will not lead to a significant reduction in total computation time.

### 3.2.1 Larger violin

Instead of testing multiple domains with totally different shapes, we create several larger domains based on the small “violin” case mentioned previously. Multiple examples can be constructed by enlarging the problem size. Based on the
original “violin” case (containing 18,157 elements), we introduce a parameter \( p_{\text{mult}} \), which controls the size of “larger violin.” A given position \((x, y)\) in the original 2D domain is mapped to \((x', y') = (p_{\text{mult}} x, p_{\text{mult}} y)\). The element size distribution \( h(x, y) \), on the other hand, transforms as follows:

\[
h'(x', y') = h\left(\frac{x'}{p_{\text{mult}}}, \frac{y'}{p_{\text{mult}}}\right). \tag{15}
\]

Therefore, the number of elements contained in “larger violin” case should be \( p_{\text{mult}}^2 \) times larger compared with that contained in the original “violin” case. Note that the generated mesh is nonuniform: the element size \( h \), as well as \( h' \), ranges from \( 3 \times 10^{-3} \) to \( 2.8 \times 10^{-2} \).

We conduct mesh generation for three different cases: \( p_{\text{mult}} = 2, 8, 32 \). The number of threads \( P \) also varies. Detailed timing results are given in Table 1. For the largest case with \( p_{\text{mult}} = 32 \), the mesh generation speed reaches 112k elements per second. Considering that these “larger violin” cases are not very large, here \( P \) does not exceed 1536.

3.2.2 Ring

Similar to the “larger violin” case, we use a single domain named “ring”, and construct multiple examples by varying the size of this domain. This domain is simply a ring-shaped area whose boundary consists of two concentric circles, as shown in the two insets of Figure 7. The inner circle’s radius \( r_{\text{in}} \) is half the radius of outer circle, which is denoted as \( r_{\text{out}} \). To change the problem size, we introduce a parameter \( p_{\text{mult}} \), which controls the radii of both circles:

\[
r_{\text{out}} = 0.5 \times (\sqrt{2})^{p_{\text{mult}}}, \quad r_{\text{in}} = 0.25 \times (\sqrt{2})^{p_{\text{mult}}}. \tag{16}
\]

The element size is uniform and does not change when the problem size changes (\( h = 0.05 \) for all \( p_{\text{mult}} \)'s).

We conduct mesh generation for three different cases: \( p_{\text{mult}} = 11, 14, 17 \). In order to show the scalability of SAFT, the number of threads \( P \) also varies. Detailed timing results are given in Table 2. For the largest case with \( p_{\text{mult}} = 17 \), the mesh generation speed reaches 176k elements per second. For a smaller case (\( p_{\text{mult}} = 14 \)) a generation speed of 233k elements per second can be achieved with a single GPU, demonstrating the effectiveness of our proposed algorithm.

In order to show the scalability of SAFT, timing curves of four different cases (\( p_{\text{mult}} = 10, p_{\text{mult}} = 12, p_{\text{mult}} = 14, \) and \( p_{\text{mult}} = 16 \), respectively) have been plotted in Figure 7. Here we are using the “strong scaling” metric,\(^{12}\) which is often used when the goal of parallelization is to minimize the time consumption of solving a given problem. More specifically, with the problem size fixed, we check the relationship between time consumption and the number of threads \( P \).

| \( p_{\text{mult}} \) | \( P \) | Utility | \( N_{\text{elem}} \) | Find new | Intersect | Choose | Update | Move | Total time |
|-------------------|---------|---------|-----------------|---------|----------|--------|--------|------|-----------|
| 2                 | 32      | 95%     | 72.3k           | 3.97    | 0.57     | <0.01  | 0.27   | 0.95 | 5.76      |
| 64                | 90%     | 72.2k   | 2.57            | 0.32    | <0.01    | 0.16   | 0.55   | 3.60 |          |
| 128               | 83%     | 72.2k   | 1.98            | 0.18    | <0.01    | 0.10   | 0.31   | 2.58 |          |
| 256               | 70%     | 72.2k   | 1.43            | 0.12    | <0.01    | 0.07   | 0.21   | 1.83 |          |
| 512               | 55%     | 72.2k   | 1.14            | 0.11    | <0.01    | 0.06   | 0.16   | 1.47 |          |
| 8                 | 128     | 95%     | 1.138M          | 23.81   | 2.75     | 0.01   | 4.08   | 8.71 | 39.35     |
| 256               | 91%     | 1.138M  | 14.16           | 1.45    | 0.01     | 2.39   | 4.78   | 22.79 |          |
| 512               | 83%     | 1.138M  | 9.39            | 1.00    | 0.01     | 1.47   | 2.79   | 14.65 |          |
| 1024              | 72%     | 1.137M  | 7.25            | 0.91    | 0.02     | 1.04   | 1.84   | 11.06 |          |
| 1536              | 64%     | 1.137M  | 6.86            | 1.18    | 0.03     | 0.92   | 1.54   | 10.53 |          |
| 32                | 512     | 95%     | 18.080M         | 149.71  | 14.03    | 0.16   | 67.91  | 106.28 | 338.10 |
| 1024              | 91%     | 18.073M | 89.81           | 11.28   | 0.17     | 40.33  | 59.72  | 201.31 |          |
| 1536              | 87%     | 18.068M | 73.04           | 12.61   | 0.20     | 32.16  | 43.43  | 161.44 |          |

Note: All timing results are measured in seconds. The results are averaged over 5 runs with different random seeds.
be concluded that our implementation shows good scalability when the number of threads $P$ is not too large. This is reasonable, considering that when $P$ is not small compared with the number of faces, utility $u$ becomes low, and increasing $P$ will not lead to a significant reduction in total computational time. We emphasize that to achieve good scalability and avoid wasting computing resources, it is always important to keep $P$ consistent with the problem size.

### 3.3 Mesh quality

In this part, to show that SAFT is able to generate 2D meshes with high quality, three different quality measures are evaluated: the angle distribution, skewness value $\epsilon$, as well as radius-edge ratio $\rho$. All the results shown in this part are based on the “ring” test case mentioned in the previous part, with $p_{\text{mult}} = 12$. The final mesh contains about 2.25 million elements.

**Angle distribution**

The angles formed by each pair of edges of a triangular element are customary measures to look at. For good elements that are close to equilateral triangles, the angles should be close to $60^\circ$. The histogram of angle distribution is plotted in Figure 8. Obviously the angle distribution peak becomes much narrower after applying Laplacian smoothing.
### TABLE 2  Detailed timing results of the “ring” case

| $p_{\text{mult}}$ | $P$  | Utility | $N_{\text{elem}}$ | Find new | Intersect | Choose | Update | Move | Total time |
|-------------------|------|---------|-------------------|----------|-----------|--------|--------|------|------------|
| 11                | 256  | 93%     | 1.128M            | 8.30     | 1.30      | 0.01   | 1.81   | 3.80 | 15.22      |
|                   | 512  | 88%     | 1.128M            | 4.65     | 0.93      | 0.01   | 1.18   | 2.32 | 9.09       |
|                   | 1024 | 78%     | 1.128M            | 2.82     | 0.81      | 0.01   | 0.87   | 1.57 | 6.09       |
|                   | 2048 | 62%     | 1.126M            | 1.94     | 1.16      | 0.02   | 0.74   | 1.17 | 5.03       |
| 14                | 512  | 95%     | 8.999M            | 47.14    | 7.56      | 0.08   | 19.23  | 35.63| 109.66     |
|                   | 1024 | 91%     | 8.999M            | 27.18    | 5.92      | 0.08   | 12.41  | 20.62| 66.23      |
|                   | 2048 | 84%     | 8.996M            | 14.55    | 7.48      | 0.09   | 9.01   | 13.14| 44.26      |
|                   | 3072 | 77%     | 8.994M            | 9.78     | 10.43     | 0.10   | 7.93   | 10.32| 38.56      |
| 17                | 1024 | 96%     | 72.416M           | 228.65   | 49.72     | 0.65   | 220.04 | 332.33| 831.40     |
|                   | 2048 | 93%     | 72.640M           | 123.42   | 59.44     | 0.61   | 137.57 | 179.11| 500.19     |
|                   | 3072 | 90%     | 72.765M           | 88.87    | 82.95     | 0.68   | 108.42 | 131.88| 412.81     |

*Note: All timing results are measured in seconds. The results are averaged over 5 runs with different random seeds.*

**FIGURE 8**  Histogram of angle distribution. Dashed blue lines represent the average value, while dashed red lines represent the minimum/maximum value. (A) The angle distribution of the original mesh, without smoothing. (B) The angle distribution of the final mesh, post-processed with Laplacian smoothing.

**Skewness**

The definition of skewness value has been given in Equation (13). If both $\theta_{\max}$ and $\theta_{\min}$ are close to $\theta_e = 60^\circ$, this element is considered good, with skewness $\epsilon \approx 0$. On the other hand, a skewness value close to 1 indicates bad element quality, which might lead to problems during numerical simulation. The histogram of skewness distribution is plotted in Figure 9. Skewness distribution has been improved much after Laplacian smoothing.

**Radius-edge ratio**

In order to show the effect of Laplacian smoothing in a more explicit way, we have also evaluated radius-edge ratio, which is a common criterion to evaluate mesh quality.\(^\text{13}\) For a given triangular element $e$, the corresponding radius-edge ratio $\rho(e)$ is defined as

$$\rho(e) = \frac{R_{\text{circ}}}{L_{\text{min}}}$$  \hspace{1cm} (17)

where $R_{\text{circ}}$ is the radius of element $e$’s circumscribed circle, while $L_{\text{min}}$ denotes length of $e$’s shortest edge. Since we work with 2D cases, the value of $\rho \in \left[ \frac{1}{\sqrt{3}}, +\infty \right)$. The histogram of radius-edge ratio distribution is plotted in Figure 10. It can be concluded that after applying Laplacian smoothing, the distribution of $\rho$ has also improved.
FIGURE 9  Histogram of skewness distribution. Dashed blue lines represent the average value, while dashed red lines represent the maximum value. (A) The skewness distribution of the original mesh, without smoothing. (B) The skewness distribution of the final mesh, post-processed with Laplacian smoothing

FIGURE 10  Histogram of radius-edge ratio distribution. Dashed blue lines represent the average value, while dashed red lines represent the maximum value. (A) The radius-edge ratio distribution of the original mesh, without smoothing. (B) The radius-edge ratio distribution of the final mesh, post-processed with Laplacian smoothing

4 | CONCLUSION

This article introduces SAFT, a front-based parallel advancing front algorithm for large-scale mesh generation. Each thread handles one face at a moment. To achieve maximum parallelism, we do not follow previous works which use hundreds of CPU cores. Instead, our implementation relies on GPU, which has thousands of CUDA cores. By taking advantage of the large number of CUDA cores that can work concurrently, SAFT achieves a much higher degree of parallelism compared with conventional parallel AFT methods. Another important advantage of SAFT is that it does not require domain decomposition, which greatly simplifies the whole mesh generation process. By discarding illegal new elements which intersect with each other, conflicts can be resolved efficiently.

We have implemented SAFT using CUDA C++. Its performance has been tested using a personal computer, which is equipped with an NVIDIA GeForce RTX2060 graphics card, as well as an Intel Core i7-9750 CPU. Detailed timings show that our approach is scalable: with one single GPU, we have been able to generate 72.7M triangular elements in less than 7 min (=176k elements per second); if tested with a smaller case, an even higher generation speed of 233k elements per second can be achieved. With proper post-processing (Laplacian smoothing, in our case), the proposed algorithm is guaranteed to generate high-quality mesh. The peak speed may not be very fast, considering that Löhner reached a much higher speed in 3D domain (403k elements per second, with 512 CPU cores). Another potential issue is that, although all the experiments have been carried out using a single GPU, it would be difficult to fit a massive 3D mesh into GPU global
memory (typically a few GB). Therefore, while our work reveals GPU’s potential for mesh generation tasks, developing a multi-GPU version of the AFT algorithm is still desired.

Note that it’s possible to extend SAFT to more general mesh generation tasks with nontriangular elements, so long as AFT is used. Although only 2D cases are implemented in this article, we do not consider this as a major weakness, because our strategy can be readily applied to 3D cases. Based on SAFT, one can construct a parallel 3D mesh generator by simply replacing the rules of finding new vertices with those rules used in 3D cases.

ACKNOWLEDGMENTS
This project has received funding from Donald E. and Esther E. Procknow Wisconsin Distinguished Graduate Fellowship (WDGF), granted by University of Wisconsin-Madison.

CONFLICT OF INTEREST
The authors declare no potential conflict of interests.

ORCID
Qingyi Zhou https://orcid.org/0000-0001-7261-1060

REFERENCES
1. Löhner R. A parallel advancing front grid generation scheme. *Int J Numer Meth Eng*. 2001;51(6):663-678.
2. Chrisochoides N. *Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer; 2006.
3. Ito Y, Shih A, Erukala A, et al. Parallel unstructured mesh generation by an advancing front method. *Math Comput Simul*. 2007;75(5-6):200-209.
4. Löhner R. Recent advances in parallel advancing front grid generation. *Arch Comput Methods Eng*. 2014;21(2):127-140.
5. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J Sci Comput*. 1998;20(1):359-392.
6. Veenaadeeve N, Chithra R, Manju M, Soman K. Efficiency of GPU over CPU in geometric processing of 3D objects. *Int J Eng Res Technol*. 2013;2(7):1884-1889.
7. Blazek J. *Computational Fluid Dynamics: Principles and Applications*. Elsevier; 2015.
8. Löhner R, Camberos J, Merriam M. Parallel unstructured grid generation. *Comput Method Appl Mech Eng*. 1992;95(3):343-357.
9. Zienkiewicz O, Taylor R, Zhu J. *The Finite Element Method: Its Basis and Fundamentals*. Elsevier; 2005.
10. Chernikov A, Chrisochoides N. Parallel guaranteed quality Delaunay uniform mesh refinement. *SIAM J Sci Comput*. 2006;28(5):1907-1926.
11. Cavendish J. Automatic triangulation of arbitrary planar domains for the finite element method. *Int J Numer Meth Eng*. 1974;8(4):679-696.
12. Hager G, Wellein G. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press; 2010.
13. Hang S. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.* 2015;41(2):11.

How to cite this article: Zhou Q, Wang Q, Yu Z. SAFT: Shotgun advancing front technique for massively parallel mesh generation on graphics processing unit. *Int J Numer Methods Eng*. 2022;123(18):4391-4406. doi: 10.1002/nme.7038