Fast Algorithms for Computing Self-Avoiding Walks and Mesh Intersections over Unstructured Meshes[1]

PeiZong Lee[2], Chih-Hsueh Yang, and Jeng-Renn Yang
Institute of Information Science
Academia Sinica
Taipei, Taiwan, R.O.C.
Internet: {leepe,dickyang,yangjr}@iis.sinica.edu.tw
TEL: +886 (2) 2788-3799
FAX: +886 (2) 2782-4814

Abstract

This paper is concerned with designing an efficient algorithm for computing the intersection of two unstructured meshes. The algorithm uses a background quadtree of the first unstructured mesh and a self-avoiding walk (SAW) of the second unstructured mesh. Due to the neighboring relationships of consecutive triangles in the triangle sequence of a SAW, we can keep track of the location of each triangle in the second unstructured mesh by means of the background quadtree. This allows us to design a linear time algorithm for computing the mesh intersection. Experimental studies show that our efficient algorithm for computing the mesh intersection can save a lot of execution time in comparison with that needed by other naive algorithms. We also present two new SAW's. Using our first-in-first-out (FIFO) SAW can save an additional 5% of the execution time in comparison with that needed when using other SAW's. This is because our FIFO SAW employs better data locality, which is especially beneficial for the current hierarchical-memory computer architectures.

Keywords: advancing front method, background quadtree, first-in-first-out queue, last-in-first-out queue, mesh intersection, self-avoiding walk, unstructured mesh.

*** A preliminary version of this technical report is accepted to be presented at the **16th AIAA Computational Fluid Dynamics Conference**, Orlando, FL, U.S.A., June 23–26, 2003.

[1]This work was partially supported by the NSC under Grants NSC 91-2213-E-001-010 and NSC 91-2213-E-001-018.

[2]PeiZong Lee is the corresponding author. Internet: leepe@iis.sinica.edu.tw, TEL: +886 (2) 2788-3799 ext. 1812, FAX: +886 (2) 2782-4814.

1 Introduction

To implement numerical simulations of engineering applications, such as engine combustion or computational fluid dynamics, unstructured meshes are tessellated in the computing domain before solving the specific governing equations, which are usually partial differential equations [7]. However, the boundary geometries of many simulated objects, like the valves and chamber of a Direct-Injection Spark-Ignition gasoline engine [14], the blades in a gas turbine, and a deforming droplet in the vicinity of a nozzle, change with time.

Figure 1 shows a period of 128 frames for engine combustion, which involves the processes of fuel and air intake, compression of the fuel-air mixture, ignition and combustion of the charge, expansion of gases, and the removal of waste. For this type of transient (where shapes change with time) application, it is practical to generate a separate unstructured mesh for each frame (of an object geometry within a period of operation). Figure 2 shows parts of unstructured meshes for frame 1 and frame 2. The unstructured mesh is regenerated because the left intake valve moves. When simulating operations, we use interpolation techniques to transfer the status of variables from frame i to frame $i+1$ for $1 \leq i < M$, and from frame M to frame 1, where we assume that a period of operation includes M frames.

To compute interpolations from frame i to frame $i+1$ or from frame M to frame 1, we have to know the intersection of each triangle (element or cell) in the unstructured mesh of frame $i+1$ (or frame 1) with respect to triangles in the unstructured mesh of frame i (or frame M, respectively). A naive implementation of computing mesh intersection requires $O(N_1 N_2)$ time complexity to test whether a triangle in the second unstructured mesh intersects with each of the triangles in the first unstructured mesh, where we assume that the first mesh has N_1 triangles, and that the second mesh has N_2 triangles.

If we construct a binary-search partition tree [6] for the first unstructured mesh in advance, then finding the first triangle $\triangle_1$ in the first unstructured mesh which intersects with a specific triangle $\triangle_2$ in the second unstructured mesh requires only $O(\log N_1)$ time complexity. The

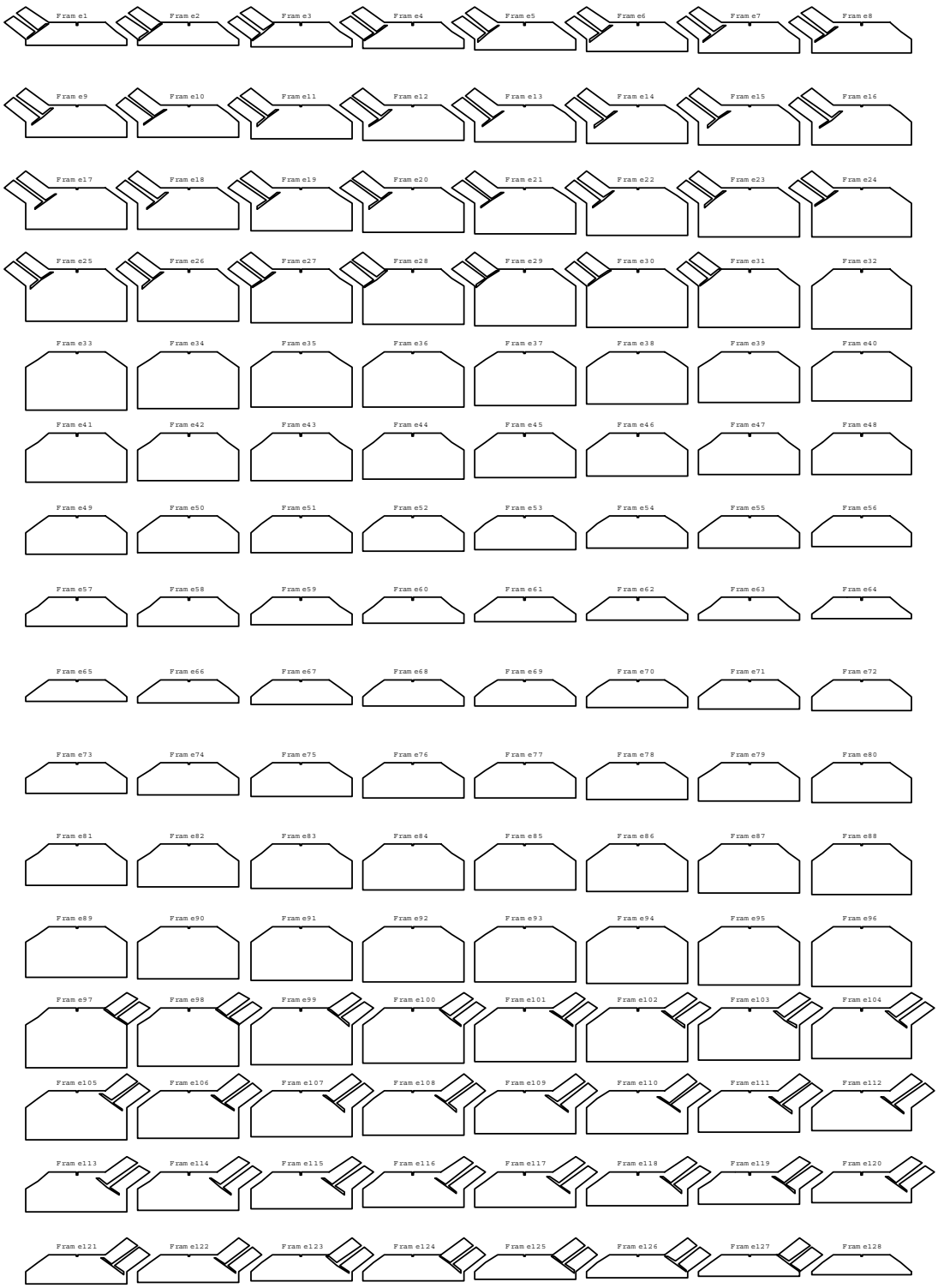

Figure 1: A period of 128 frames for the engine combustion.

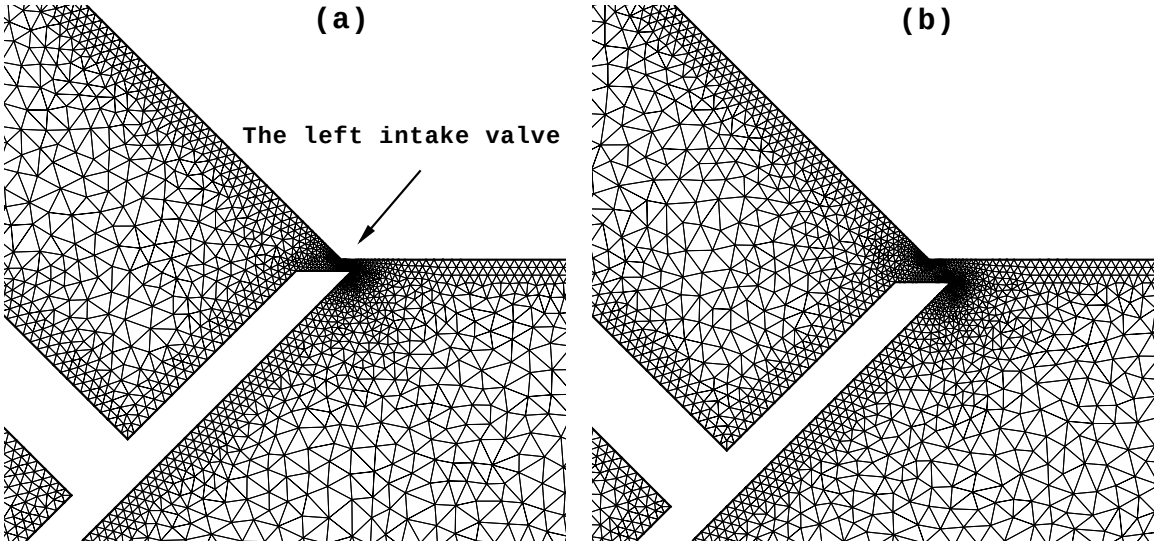

Figure 2: Parts of unstructured meshes near the left intake valve of (a) frame 1 and (b) frame 2 shown in Figure 1.

whole set of triangles in the first unstructured mesh which intersect with $\triangle_2$ can then be found based on the local information of $\triangle_1$ in a constant amount of time. Therefore, the time complexity of computing mesh intersection is reduced to $O(N_2 \log N_1)$.

In this paper, we present an efficient algorithm which can further reduce the time complexity to $O(N_1 + N_2)$ for most cases. Our algorithm requires a *background quadtree* of the first unstructured mesh and a triangle sequence of a *self-avoiding walk* for the second unstructured mesh. The background quadtree, which is defined before unstructured mesh generation to represent a smooth change of density distribution among triangles in the computing domain [9], can be used to identify the location of a triangle. A self-avoiding walk (SAW) over an arbitrary unstructured mesh is an enumeration of all the triangles of that mesh such that two successive triangles share an edge or a vertex [8]. A SAW can be treated as a serialization technique which transforms a two-dimensional unstructured mesh into a sequence of consecutive triangles.

We first construct a SAW sequence for the second unstructured mesh. Following the SAW sequence, after finding the intersection set `ISET1` of the first triangle, we can find the intersection set `ISET2` of the second triangle based on the local information of `ISET1` in a

constant amount of time as the first triangle is adjacent to the second triangle by an edge or by a vertex. Similarly, the intersection set `ISET3` of the third triangle can be found based on the local information of `ISET2`, and so on. Therefore, the time complexity is reduced to $O(N_1 + N_2)$ provided that each triangle in the second unstructured mesh always intersects with triangles in the first unstructured mesh.

However, as the object geometry of the first frame may be different from the object geometry of the second frame, some triangles in the second unstructured mesh may not intersect with any triangle in the first unstructured mesh. Therefore, the local information of the predecessor's intersection set breaks (and therefore requires additional searching). In this case, we use a background quadtree of the first unstructured mesh to keep track of the location of each triangle $\triangle_2$ in the SAW sequence of the second unstructured mesh. Then, we exhaustively test intersections for $\triangle_2$ and those triangles in the first unstructured mesh which fall within the territory of the same quadtree leaf as that of $\triangle_2$. If the territory of each quadtree leaf contains at most a constant number of triangles, then each exhaustive test can be done in a constant amount of time.

The average time complexity of keeping track of the location of each triangle in the SAW sequence (of the second unstructured mesh) over the background quadtree (of the first unstructured mesh) is difficult to analyze, but it is bounded by $O(\log N_1)$, where the height of the background quadtree is $O(\log N_1)$. However, since there is only a slight change of the object geometries from frame i to frame $i + 1$ or from frame M to frame 1, only a small portion of triangles in the second unstructured mesh will not intersect with any triangle in the first unstructured mesh. Therefore, the overhead of keeping track of the locations of all the triangles $\{\triangle_2^{\emptyset}\}$ in the SAW sequence over the background quadtree can be neglected, where the set $\{\triangle_2^{\emptyset}\}$ does not intersect with any triangle in the first unstructured mesh. Therefore, the time complexity of mesh intersection is still $O(N_1 + N_2)$.

The SAW sequences (over unstructured meshes) or space-filling curves (over structured

meshes) [19] are frequently used to enhance data locality, so that data accesses can comply with current hierarchical-memory computer architectures [2, 16]. In this paper, we present two new SAW's and one algorithm for finding mesh intersection. We analyze the cache effects of using these two new SAW's and another two SAW's suggested in [8] when executing mesh intersection. We also present experimental studies of mesh intersection for all 128 frames of engine combustion.

The rest of this paper is organized as follows: Section 2 surveys related works. Section 3 presents the two new SAW's. Section 4 presents our algorithm for computing mesh intersection. Section 5 presents experimental studies, and Section 6 gives some concluding remarks.

2 Related works

Many practical applications are usually time-varying (transient) and have complex geometries. Therefore, more than one mesh can be adopted in the numerical simulations. These meshes may coexist at the same time-step or be built in sequential time-steps. Mesh intersection plays an important role in such numerical simulations. Examples are described in the following.

In multi-physics problems, since variables relevant to multiple physical phenomena are obtained in synchronization steps, the optimal grids for each physical variable need not be the same, so separate grids may be used to solve the appropriate equations for each variable. For example, when the welding of a joint between two parts is simulated, one grid can be used to solve the stress-strain relations to account for the mechanical deformation of the parts, and the other grid can be used for thermal conduction calculations in the system. When both the thermal and mechanical effects are considered, the solution data must be interpolated back and forth between the two grids for each time step [17].

In multi-body simulations, especially for problems with moving bodies or for those having

complex geometries, a series of body-fitted grids separated for each component may overlap. These are called overset grids, and interpolation is used to transmit data between the overset grids in the flow solver. The Chimera scheme is widely used to deal with this kind of problem [3, 18, 22]. It can break complex configurations into components (or regions), generate a series of separate body-conforming grids for each component (or region) of the configuration, and then overset these grids together to form a complete model [20].

In transient problems, like the propagation of a planar shock, local meshes are regenerated as time progresses, and a set of dynamically adaptive meshes are built. The values of the regenerated meshes need to be interpolated from old ones to new ones [7]. The internal combustion engine, which consists of chemical reaction, moving valves and pistons, and fuel injection, is typically a transient problem with changing shapes. The mesh should be regenerated if the valves and pistons move. The intermediate values also need to be transferred from the old mesh to the new one.

Unstructured meshes are becoming important as they can be generated automatically for applications with complex geometries or for those with dynamically moving boundaries [21]. For engine combustion applications, a fast approach might be to regenerate local meshes for the places where boundaries change. However, the quality of the newly generated meshes in these places might be poor in terms of the aspect ratio, area ratio, and edge ratio among the triangles (elements or cells of a mesh) [9]. The quality of a mesh influences the convergence rate of the PDE solvers. Therefore, it is more suitable to generate a separate mesh for each of the frames which represent boundary geometries for a period of operations.

The mesh intersection problem is also called the intergrid communication problem [1, 15, 22], grid transfer problem [17], or interpolation for unstructured grids [13]. Chesshire and Henshaw considered the overlapping of structured grids, where the density distribution of each grid is uniform [4]. They used inverse Cartesian mappings with a neighboring search to find the nearest vertex (called an interpolation point).

Meakin *et al.* also adopted inverse Cartesian mappings to solve the intergrid communication problem [1, 15, 22]. They found that in the highly refined regions, a cell (or a quadrant) of a background Cartesian mesh might enclose a large number of grid elements (triangles). Consequently, the index range of the search region defined by the vertices of the Cartesian cell is likely to be large, and the resulting element (vertex or triangle) search costly. Therefore, multi-level inverse Cartesian mappings were needed for a single grid.

To deal with unstructured meshes, Löhner used a background quadtree to search nearby grid elements [12, 13]. Plimpton *et al.* adopted recursive coordinate bisectioning techniques to search nearby grid elements. Both of their methods can find an independent grid element in a logarithmic amount of time.

SAW's were first introduced by Heber, Biswas, and Gao for renumbering unstructured meshes so that data locality for accessing neighboring data could be improved [8]. As two consecutive triangles in a SAW sequence shared an edge or a vertex, SAW's were also used to do data partitioning for sparse matrix applications over unstructured meshes on parallel computers.

Cuthill and McKee suggested another renumbering method based on breadth-first search on a graph [5]. Starting from a vertex of minimal degree, they applied breadth-first search level-by-level, where vertices with a small degree within each level were numbered first, followed by vertices with a large degree. Cuthill and McKee's sequence is well-known for reducing the bandwidth of a sparse matrix. Liu and Sherman further pointed out that the reverse Cuthill-McKee (RCM) sequence, where level construction was restarted from a vertex of minimal degree in the final level, was found to always be at least as good as its corresponding Cuthill-McKee (CM) sequence in terms of minimizing the bandwidth of a sparse matrix [11].

Most applications found by using SAW's or CM or RCM orderings were related to direct solvers of sparse linear systems or iterative solvers using a conjugate gradient algorithm,

where sparse matrices were symmetric and positive definite. Therefore, different orderings could still get the correct answer because all of these orderings could make the solution convergent. CM and RCM orderings can further minimize the number of non-zero fill-in's in sparse matrices when solving sparse linear systems directly. Note that a large number of non-zero fill-in's may prevent scientists from using direct solvers due to the limitation imposed on the memory size.

However, for some computational fluid dynamic applications, such as Euler and Navier-Stokes equations, due to the hyperbolic property, the resulting sparse matrix is not a symmetric matrix. Therefore, SAW's or CM or RCM orderings might delay the convergence of a solution obtained using iterative solvers. In effect, we have found that for the Euler flow solver, using a diagonal ordering can improve convergence, where in the diagonal ordering, triangle $\triangle_1$ is prior to triangle $\triangle_2$ if their gravity centers (x_1, y_1) and (x_2, y_2) satisfy $x_1 + y_1 < x_2 + y_2$ [10]. This is probably because elements (triangles) in the mesh are iterated along a particular direction, for example, from south-east to north-west, according to the elements' coordinates. Note that CM, RCM, and diagonal orderings are not SAW's. The effectiveness of an ordering depends on its applications.

In this paper, we emphasize that a SAW can be used as a sequence to find mesh intersection. However, SAW's or CM or RCM orderings are not needed to be the ordering of an unstructured mesh, as these orderings will not necessarily converge quickly when a general iterative PDE solver is employed.

3 Generating self-avoiding walks

An unstructured mesh is composed of triangles. Each triangle has three vertices, three edges, and at most three adjacent triangles. Each pair of adjacent triangles share a common edge. Each vertex is surrounded by several triangles; thus, these triangles have a common vertex. Since the computing domain is connected, starting from any triangle, we can use the

advancing front method to traverse all the triangles in the computing domain.

3.1 An algorithm for generating self-avoiding walks

The advancing front method treats each edge as a front. Starting from any boundary edge e, which we define as the first and only active front, we cross edge e and enter the adjacent triangle $\triangle$. Now, in this new triangle $\triangle$, we set the other two edges as two new active fronts if these two edges were not crossed before, and set edge e as an inactive front because this front is now hidden by other new active fronts. We repeatedly cross active fronts as described above until all the fronts are set to be inactive.

In the following, we use a queue to store active fronts. This queue can be implemented as a FIFO (first-in-first-out) queue, a LIFO (last-in-first-out) queue (which is a stack) or any other interesting queue. The FIFO queue corresponds to a breadth-first search, while the LIFO queue corresponds to a depth-first search. For clarity, we use $\langle \texttt{edge}, \triangle_{x_i}, \triangle_a \rangle$ to represent an active front, where $\triangle_{x_i}$ is visited but $\triangle_a$ is not, and where `edge` is their shared (common) edge. According to the direction from $\triangle_{x_i}$ to $\triangle_a$, we also define the left vertex and the right vertex of the front as being the same as those of the front `edge`. Of course, initially, we only have one special front $\langle \texttt{edge}, \emptyset, \triangle_a \rangle$, where `edge` is a boundary edge. We use a double link list to store visited triangles $\triangle_{x_i}$, where one link points to its predecessor $\triangle_{x_{i-1}}$ and the other link points to its successor $\triangle_{x_{i+1}}$. We use $|\triangle_a, \triangle_b, \ldots, \triangle_k|$ to represent the number of triangles in the triangle sequence.

Algorithm 1 for generating self-avoiding walks :

Step 1. Initially, the SAW sequence is empty. We start from an initial boundary front $\langle \texttt{edge}, \emptyset, \triangle_a \rangle$.

Step 2. (Enqueue phase)

Let $\triangle_a$ be the adjacent triangle (which was not visited before as we just crossed a

new active front). We insert $\triangle_a$ into the SAW. Then, we reset the original front to be inactive as it is now hidden by $\triangle_a$. However, we also get either 0 or 1 or 2 new active fronts.

Step 2-1. We get 0 active fronts. The enqueue phase stops, and we continue with Step 3.

Step 2-2. We get 1 active front. We have two cases.

Case 2-2-1: The original left vertex is the left vertex of the new front, and the third vertex of $\triangle_a$ is the right vertex of the new front.

Case 2-2-2: The original right vertex is the right vertex of the new front, and the third vertex of $\triangle_a$ is the left vertex of the new front.

In both cases, we cross the new front and repeat Step 2.

Step 2-3. We get 2 active fronts. The original right vertex is the right vertex of the new right front, and the third vertex of $\triangle_a$ is the left vertex of the new right front. The original left vertex is the left vertex of the new left front, and the third vertex of $\triangle_a$ is the right vertex of the new left front.

We first enqueue the new right front into a front queue; then, we cross the new left front and repeat Step 2.

Step 3. (Dequeue phase)

We dequeue a front from the front queue, called $\langle \texttt{edge}, \triangle_{x_i}, \triangle_a \rangle$.

If there is no front, then the Dequeue phase stops.

If $\triangle_a$ was visited before, then we repeat Step 3.

Otherwise, we reset the front to be inactive. We have four cases.

Case 3-1: When $\triangle_{x_i}$ is not the first triangle in the SAW sequence, if $\triangle_{x_i}$, $\triangle_a$, and $\triangle_{x_{i-1}}$ share a vertex, and moving clockwise along this shared vertex, none of $\triangle_a$, $\triangle_b$, $\ldots$, $\triangle_k$ are visited, and if $|\triangle_a, \triangle_b, \ldots, \triangle_k| > 1$, then we insert $\triangle_k$, $\triangle_{k-1}$, $\ldots$, $\triangle_b$, $\triangle_a$

into the SAW, such that the SAW sequence has the following order: $\triangle_{x_{i-1}}$, $\triangle_k$, $\triangle_{k-1}$, ..., $\triangle_b$, $\triangle_a$, $\triangle_{x_i}$.

In the meantime, we mark $\triangle_k$, $\triangle_{k-1}$, ..., $\triangle_b$, $\triangle_a$ to be visited and enqueue fronts adjacent to $\triangle_k$, $\triangle_{k-1}$, ..., $\triangle_b$, $\triangle_a$ into the front queue. After that, we repeat Step 3.

Case 3-2: When $\triangle_{x_i}$ is not the last triangle in the SAW sequence, if $\triangle_{x_i}$, $\triangle_a$, and $\triangle_{x_{i+1}}$ share a vertex, and moving counterclockwise along this shared vertex, none of $\triangle_a$, $\triangle_b$, ..., $\triangle_k$ are visited, and if $|\triangle_a, \triangle_b, \ldots, \triangle_k| > 1$, then we insert $\triangle_a$, $\triangle_b$, ..., $\triangle_k$, into the SAW, such that the SAW sequence has the following order: $\triangle_{x_i}$, $\triangle_a$, $\triangle_b$, ..., $\triangle_k$, $\triangle_{x_{i+1}}$.

In the meantime, we mark $\triangle_a$, $\triangle_b$, ..., $\triangle_k$ to be visited and enqueue fronts adjacent to $\triangle_a$, $\triangle_b$, ..., $\triangle_k$ into the front queue. After that, we repeat Step 3.

Case 3-3: When $\triangle_{x_i}$ is not the first triangle in the SAW sequence, if $\triangle_{x_i}$, $\triangle_a$, and $\triangle_{x_{i-1}}$ share a vertex, and moving clockwise along this shared vertex, only $\triangle_a$ is not visited, then we insert $\triangle_a$ into the SAW, such that the SAW sequence has the following order: $\triangle_{x_{i-1}}$, $\triangle_a$, $\triangle_{x_i}$.

In the meantime, we mark $\triangle_a$ to be visited and enqueue the front adjacent to $\triangle_a$ into the front queue. After that, we repeat Step 3.

Case 3-4: When $\triangle_{x_i}$ is not the last triangle in the SAW sequence, if $\triangle_{x_i}$, $\triangle_a$, and $\triangle_{x_{i+1}}$ share a vertex, and moving counterclockwise along this shared vertex, only $\triangle_a$ is not visited, then we insert $\triangle_a$ into the SAW, such that the SAW sequence has the following order: $\triangle_{x_i}$, $\triangle_a$, $\triangle_{x_{i+1}}$.

In the meantime, we mark $\triangle_a$ to be visited and enqueue the front adjacent to $\triangle_a$ into the front queue. After that, we repeat Step 3.

Cases 3-1 and 3-2 are prior to Cases 3-3 and 3-4 as we prefer to include more "share-edge" consecutive triangles in the SAW sequence. For example, in Cases 3-1 and 3-2, $\triangle_{x_i}$ and $\triangle_a$

have a shared edge, $\triangle_a$ and $\triangle_b$ have a shared edge, ..., and $\triangle_{k-1}$ and $\triangle_k$ have a shared edge. For unstructured mesh applications, a triangle frequently needs information about its three adjacent triangles. Therefore, "shared-edge" consecutive triangles have better data locality than "shared-vertex" consecutive triangles do. In the following, we show that Algorithm 1 can visit all the triangles; in addition, each triangle appears in the SAW sequence only once. Thus, Algorithm 1 is complete.

Theorem 1 *Algorithm* 1 *can visit all the triangles.*

Proof: Algorithm 1 adopts the advancing front method to visit triangles. Thus, all the triangles behind active fronts are visited. Active fronts are generated in Step 2 and Step 3. In Step 2 (of the enqueue phase), for each active front $\langle \texttt{edge}, \triangle_{x_i}, \triangle_a \rangle$ enqueued into the front queue, the front edge has a common vertex with $\triangle_{x_{i-1}}$ if $\triangle_{x_i}$ is not the first triangle in the SAW sequence, and has a common vertex with $\triangle_{x_{i+1}}$ if $\triangle_{x_i}$ is not the last triangle in the SAW sequence.

In Step 3 (of the dequeue phase), new active fronts $\langle \texttt{edge}_a, \triangle_a, \triangle'_a \rangle$, $\langle \texttt{edge}_b, \triangle_b, \triangle'_b \rangle$, ..., and $\langle \texttt{edge}_k, \triangle_k, \triangle'_k \rangle$ are enqueued into the front queue. In addition, their corresponding SAW sequence is either $\triangle_{x_{i-1}}$, $\triangle_k$, $\triangle_{k_1}$, ..., $\triangle_a$, $\triangle_{x_i}$ or $\triangle_{x_i}$, $\triangle_a$, $\triangle_b$, ..., $\triangle_k$, $\triangle_{x_{i+1}}$. $\texttt{edge}_a$ has a common vertex with $\triangle_{x_i}$ and a common vertex with $\triangle_b$; $\texttt{edge}_b$ has a common vertex with $\triangle_a$ and a common vertex with $\triangle_c$, ...; and $\texttt{edge}_k$ has a common vertex with $\triangle_{k-1}$. Therefore, the edge of each active front has at least one common vertex with its preceding triangle or its succeeding triangle.

Thus, Cases 3-1, 3-2, 3-3, and 3-4 are exhaustive. Because triangles in the unstructured mesh are connected by edges, there exists a triangle path connecting any two triangles, such that any two consecutive triangles in the path have a common edge. An edge can be treated as a front; therefore, all the triangles can be visited by the algorithm. □

Theorem 2 *Each triangle appears in the SAW sequence only once.*

Proof: An active front is represented by $\langle \texttt{edge}, \triangle_{x_i}, \triangle_a \rangle$, where $\triangle_{x_i}$ is visited and $\triangle_a$ is not. In Step 2, if $\triangle_a$ is visited, then $\langle \texttt{edge}, \triangle_{x_i}, \triangle_a \rangle$ is not an active front, and we will not cross the front `edge`. In Step 3, when a front $\langle \texttt{edge}, \triangle_{x_i}, \triangle_a \rangle$ is dequeued from the front queue, we first check whether $\triangle_a$ was visited previously. If $\triangle_a$ was visited previously, we ignore that front. Thus, $\triangle_a$ will not be inserted into the SAW sequence twice. □

We now analyze the algorithm. We assume that the unstructured mesh contains N triangles. Each front (or each edge) can be enqueued into and dequeued from the front queue only once; thus, the time complexity of generating a SAW sequence is proportional to the number of fronts (or edges). Therefore, the time complexity of generating a SAW sequence is $O(N)$.

3.2 FIFO SAW and LIFO SAW

We have implemented a FIFO SAW, which is based on a FIFO queue, and a LIFO SAW, which is based on a LIFO queue. We use the terms "FIFO" SAW and "LIFO" SAW to distinguish between them and BFS SAW and DFS SAW proposed in [8] to avoid confusion. Recall that in Algorithm 1, we have an enqueue phase (Step 2) and a dequeue phase (Step 3). In Step 2 the SAW sequence is generated clockwise along the boundary of the computing domain. Each triangle visited in Step 2 has a common edge with its predecessor if this triangle is not the first triangle in the SAW sequence, and has a common edge with its successor if this triangle is not the last triangle in the SAW sequence. Each triangle has at most three adjacent triangles. Although this triangle and two adjacent triangles are numbered consecutively, the third adjacent triangle may have a number far from theirs.

To improve the average distance between adjacent triangles, we can restrict the number of triangles inserted into the SAW sequence in the enqueue phase (Step 2). We use FIFO(c) SAW or LIFO(c) SAW to represent the SAW with at most c triangles inserted into its sequence in Step 2. To understand the flavors of different SAW's, Figure 3 shows FIFO($\sqrt{N}$) SAW,

FIFO(1) SAW, LIFO(N) SAW, and LIFO(1) SAW over an 8×8 structured mesh, where the basic element in this structured mesh is a "square" cell instead of a "triangle" cell. In these orderings, FIFO($\sqrt{N}$) SAW has the best average distance between adjacent cells in this example.

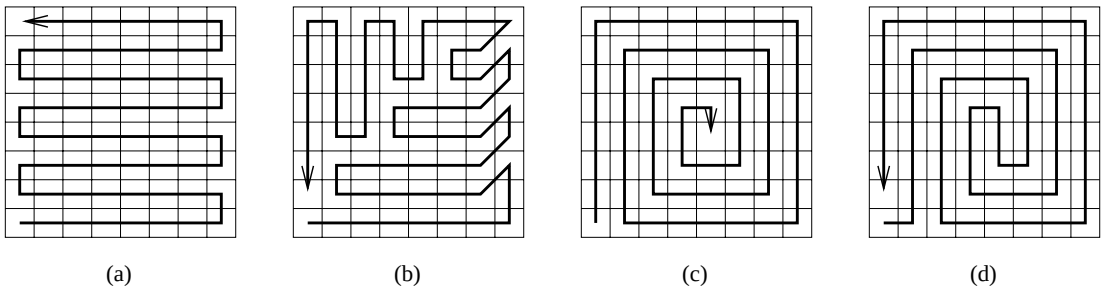

Figure 3: (a) FIFO($\sqrt{N}$) SAW, (b) FIFO(1) SAW, (c) LIFO(N) SAW, and (d) LIFO(1) SAW over an 8×8 structured mesh.

For convenience, we use FIFO SAW to represent FIFO(N) SAW and LIFO SAW to represent LIFO(N) SAW to avoid confusion. Figure 4 shows four SAW's over a sample unstructured mesh, including our FIFO($\sqrt{N}$) SAW, our LIFO SAW, and BFS SAW and DFS SAW suggested in [8].

A SAW is generated based on the graph data structure of an unstructured mesh. Therefore, starting from a different initial front will result in a different SAW. In this paper, we choose a boundary edge at the south-eastern corner as the initial front. In effect, we generate several SAW's starting from different boundary edges; although their average distances between adjacent triangles vary quite a bit, the differences in their execution times for performing mesh intersection are insignificant.

3.3 Quality measure of SAW's

Heber, Biswas, and Gao proposed a quality measure in [8], which computes the average distance between each pair of adjacent triangles of an unstructured mesh based on the numbering of the SAW sequence or the mesh ordering. This measure is a good reference for a SAW sequence. Table 1 shows the average distances obtained by using different mesh order-

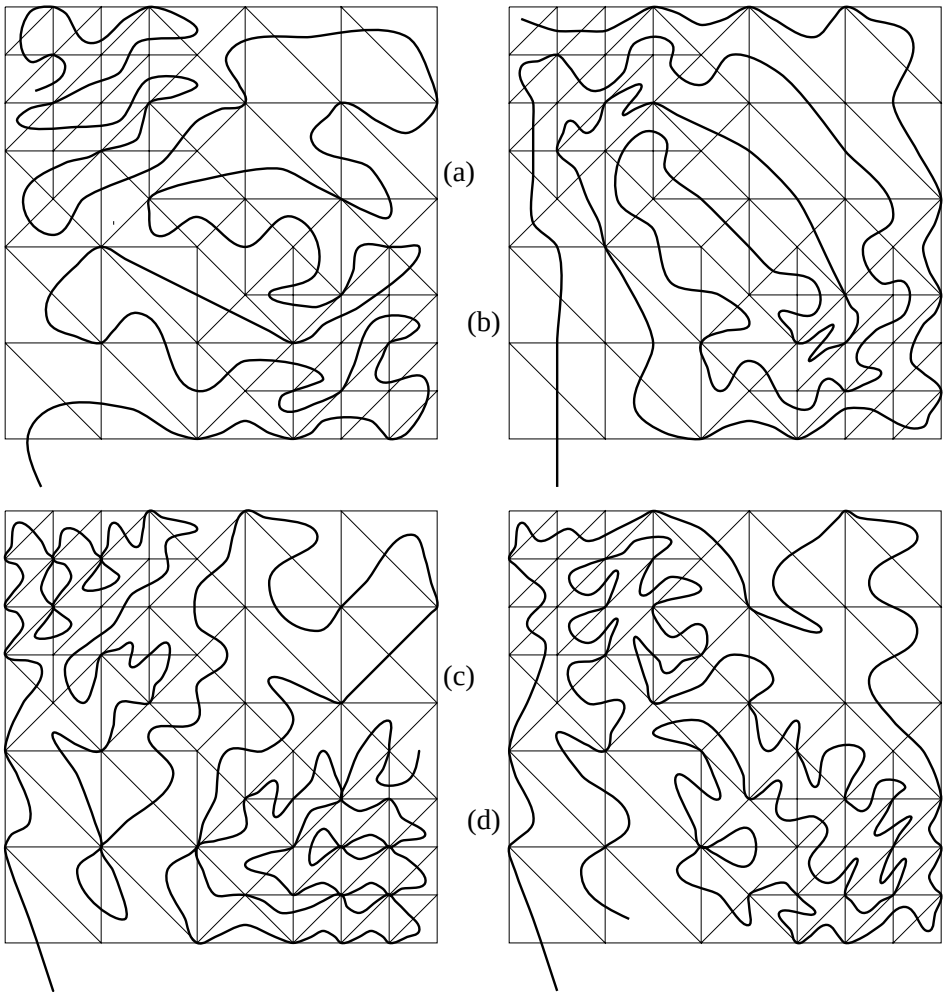

Figure 4: (a) FIFO($\sqrt{N}$) SAW, (b) LIFO SAW, (c) BFS SAW, and (d) DFS SAW over an unstructured mesh.

ings over 128 unstructured meshes of engine combustion shown in Figure 1. Among them, we also apply this measure to three non-SAW cases for comparison: (1) the original mesh ordering obtained from mesh generators or mesh databases, (2) the diagonal ordering, and (3) the reverse Cuthill-McKee ordering (RCM). Among these SAW's, our FIFO($\sqrt{N}$) SAW has the minimum average distance, while our LIFO SAW has the maximum average distance.

Taking the number of walks through edges and vertices into consideration is another interesting measure. The situation *walk through an edge* (edge-walk), which means that two consecutive triangles are tightly connected by a shared edge, may have better data locality for certain PDE solvers than will two consecutive triangles be connected by a shared vertex

Mesh order	Avg. dist.
Original Order	2329.64
Diagonal Order	50.43
RCM Order	60.26
FIFO Order	103.87
FIFO($\sqrt{N}$) Order	53.24
LIFO Order	2880.54
BFS Order	76.23
DFS Order	1297.30

Table 1: The average distance when applying different SAW's on a mesh. The results are averaged again based on the average distances of 128 meshes.

(vertex-walk). Table 2 shows the average counts of walks through edges and vertices for each SAW over 128 unstructured meshes of engine combustion shown in Figure 1. Our LIFO SAW takes an edge-favor walk, so it has a minimum number of vertex-walks, but it also increases the average distance. The counts of edge-walks for our FIFO SAW and FIFO($\sqrt{N}$) SAW are not as good as those for our LIFO SAW but are still better than those of both of H-B-G's SAW's.

	Our SAW			H-B-G's SAW	
Walk count	FIFO	FIFO($\sqrt{N}$)	LIFO	BFS	DFS
#edge-walk	13751.18	14086.21	17300.05	11833.19	11329.07
#vert-walk	4946.37	4611.34	1397.49	6864.36	7368.48

Table 2: The counts of walks through edges and vertices when applying different SAW's on a mesh. The results are averaged again based on the counts of 128 meshes.

The effectiveness of these SAW's in real applications, however, depends on their contribution to saving execution time. We will study their cache effects in Section 5.

4 Computing mesh intersections

Our algorithm requires a background quadtree of the first unstructured mesh and a SAW sequence of the second unstructured mesh. The background quadtree, which was defined before performing unstructured mesh generation, is used to represent a smooth change of density distribution among triangles in the computing domain [9]. If an unstructured mesh is not associated with a background quadtree, we can construct one such that the territory

of each quadtree leaf contains at most a certain constant number of triangles. We say that the territory of a quadtree leaf contains a triangle if the gravity center of that triangle falls within the territory of this quadtree leaf. Figure 5 shows a background quadtree over a sample unstructured mesh, which is traversed by our FIFO($\sqrt{N}$) SAW. Figure 5 illustrates the possibility of tracking the location of each triangle in a SAW sequence by means of a quadtree. Note again that, in our mesh intersection algorithm, we really need a background quadtree for the first unstructured mesh and a SAW sequence for the second unstructured mesh to avoid confusion.

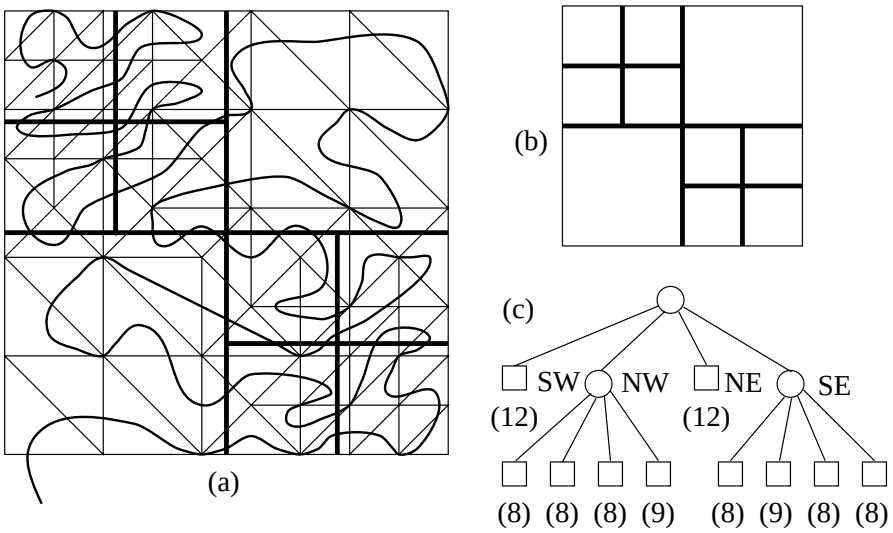

Figure 5: (a) A background quadtree over a sample unstructured mesh, which is traversed by our FIFO($\sqrt{N}$) SAW, (b) the territory of the quadtree, and (c) the quadtree structure, where parentheses enclose the number of triangles in the territory of each quadtree leaf.

The idea behind our algorithm is as follows. If $\triangle_2$ and $\triangle_2'$ are adjacent to each other in the second unstructured mesh, their triangle-intersection sets, with respect to triangles in the first unstructured mesh, have non-empty intersection. Thus, we can follow a SAW sequence of the second unstructured mesh to use the local information of the preceding triangle-intersection set to generate a succeeding triangle-intersection set, provided that each triangle in the second unstructured mesh always intersects with triangles in the first unstructured mesh.

However, if the object geometry of the first unstructured mesh is different from the object geometry of the second unstructured mesh, then some triangles in the second unstructured mesh may not intersect with any triangle in the first unstructured mesh. Thus, the local information of the preceding intersection set breaks (and therefore requires additional searching). In this case, we use a background quadtree of the first unstructured mesh to keep track of the location of each triangle $\triangle_2$ in the SAW sequence of the second unstructured mesh. We exhaustively test intersections for $\triangle_2$ and those triangles in the first unstructured mesh which fall within the territory of the same quadtree leaf as that of $\triangle_2$.

Algorithm 2 for computing the intersection of two meshes:

Pick out one triangle $\triangle_2$ from the SAW sequence of the second unstructured mesh,

If the predecessor's triangle-intersection set is not empty,

then we use the local information of the predecessor's triangle-intersection set to generate the triangle-intersection set of $\triangle_2$;

otherwise we use a background quadtree of the first unstructured mesh to keep track of the location of $\triangle_2$. We exhaustively test intersections for $\triangle_2$ and those triangles in the first unstructured mesh which fall within the territory of the same quadtree leaf as that of $\triangle_2$.

Note that the triangle-intersection set of $\triangle_2$ may fall across the territories of more than one quadtree leaf. However, except for intersecting with boundary triangles, the triangle-intersection set of $\triangle_2$ is connected. Therefore, once we have found a triangle $\triangle_1$ in the first unstructured mesh such that $\triangle_1 \cap \triangle_2 \neq \emptyset$, the remaining triangle-intersection set of $\triangle_2$ can be found using the local information of $\triangle_1$.

As for intersecting with boundary triangles, the resulting triangle-intersection set of $\triangle_2$ may be disconnected. Therefore, we have to consider all the territories of the quadtree leafs

that enclose $\triangle_2$. In effect, we consider all the candidate triangles that fall within the territories of those quadtree leafs in which three vertices and four *range points* of $\triangle_2$ fall. Suppose that three vertices of $\triangle_2$ are (x_1, y_1), (x_2, y_2), and (x_3, y_3). Let $\texttt{xmax} = \texttt{max}(x_1, x_2, x_3)$, $\texttt{xmin} = \texttt{min}(x_1, x_2, x_3)$, $\texttt{ymax} = \texttt{max}(y_1, y_2, y_3)$, and $\texttt{ymin} = \texttt{min}(y_1, y_2, y_3)$. Then, four range points of $\triangle_2$ are (xmin, ymin), (xmin, ymax), (xmax, ymin), and (xmax, ymax).

To evaluate whether a candidate triangle $\triangle_1$ in the first unstructured mesh intersects with $\triangle_2$, we perform the following four tests in turn. If the second, the third, and the fourth tests are not satisfied, then $\triangle_1$ and $\triangle_2$ do not intersect.

Test 1: We perform a range test for $\triangle_1$ and $\triangle_2$. If the range of $\triangle_1$ and the range of $\triangle_2$ do not intersect, then $\triangle_1$ and $\triangle_2$ do not intersect. This is an inexact test, but it is a fast way to prune off many irrelevant candidates.

Test 2: We test whether a vertex of $\triangle_1$ is within $\triangle_2$. If it is, then all the triangles (including $\triangle_1$) surrounding this vertex intersect with $\triangle_2$. We also mark all of their neighboring triangles as candidates.

Test 3: We test whether an edge of $\triangle_1$ intersects with an edge of $\triangle_2$. If it does, then both triangles (including $\triangle_1$) adjacent to this edge intersect with $\triangle_2$. We also mark all of their neighboring triangles as candidates.

Test 4: We test whether a vertex of $\triangle_2$ is within $\triangle_1$. If it is, then $\triangle_1$ intersects with $\triangle_2$. In this case all of $\triangle_2$ is within $\triangle_1$.

Since $\triangle_2$ can intersect with at most a certain constant number of triangles in the first unstructured mesh, we can find the whole set of triangles which intersects with $\triangle_2$ based on the local information of $\triangle_1$ in a constant period of time, after finding the first $\triangle_1$ which intersects with $\triangle_2$. In numerical simulations, the change of object geometries in successive frames is kept small in order to guarantee achievement of convergence. Thus, there are only a few triangles for which the local information of the preceding intersection-set breaks. As mentioned in the Introduction, for this type of changing-shape application, the time

complexity of our algorithm for computing mesh intersection is linear with respect to the number of triangles in the first and second unstructured meshes.

Note that it is possible to compute the range intersection of two target unstructured meshes in a preprocessing step in order to screen out some irrelevant triangles if these two meshes have only a small area of intersection. In our application, two consecutive frames change very slightly; therefore, we ignore the preprocessing step.

5 Experimental studies

Our experimental studies were implemented on a SUN Ultrasparc-3 (750 MHz) workstation. Our benchmark suit contained 128 consecutive unstructured meshes (corresponding to 128 frames) shown in Figure 1. Table 3 lists the numbers of triangles, edges, and vertices of these 128 unstructured meshes. Experimental results show the improvements obtained by using SAW's to compute mesh intersections and also show the impact of using different SAW's.

Table 4 shows the average execution time of mesh intersection using different SAW's based on different mesh orderings. These interesting orderings include: (1) the original mesh ordering obtained from mesh generators or mesh databases, (2) the diagonal ordering, (3) the reverse Cuthill-McKee ordering (RCM), (4) our FIFO SAW ordering, (5) our FIFO($\sqrt{N}$) SAW ordering, (6) our LIFO SAW ordering, (7) the BFS SAW ordering in [8], and (8) the DFS SAW ordering in [8]. We let the first mesh and the second mesh use the same kinds of orderings as listed in the first dimension. These orderings could influence the convergence rate of certain PDE solvers. We then used the different SAW's of the second mesh as listed in the second dimension to compute mesh intersections. We stress again that mesh orderings play an important role in determining the convergence rate of certain PDE solvers; however, SAW's were only used to find mesh intersections in this study.

When we did not use any SAW sequence, we used a quadtree to keep track of the location of preceding triangles, denoted by QT-Track. Otherwise, we simply searched from the root

mesh	#△'s	#edges	#vert.	mesh	#△'s	#edges	#vert.	mesh	#△'s	#edges	#vert.
1	15237	23388	8152	44	16998	25882	8885	87	18033	27446	9414
2	16122	24716	8595	45	16615	25304	8690	88	18090	27535	9446
3	16907	25903	8997	46	16140	24588	8449	89	18524	28190	9667
4	17268	26446	9179	47	15764	24020	8257	90	18976	28872	9897
5	17948	27471	9524	48	15680	23890	8211	91	19447	29582	10136
6	18492	28293	9802	49	15109	23030	7922	92	19634	29866	10233
7	18988	29040	10053	50	14932	22761	7830	93	19856	30203	10348
8	19546	29888	10343	51	14758	22496	7739	94	20452	31101	10650
9	20170	30825	10656	52	14132	21553	7422	95	20619	31355	10737
10	20516	31351	10836	53	13849	21125	7277	96	20866	31729	10864
11	20927	31972	11046	54	13374	20409	7036	97	25615	39068	13454
12	21372	32643	11272	55	13090	19979	6890	98	25747	39260	13514
13	22056	33678	11623	56	12814	19561	6748	99	25747	39264	13518
14	22299	34044	11746	57	12367	18887	6521	100	25631	39083	13453
15	22559	34445	11887	58	12054	18414	6361	101	25901	39486	13586
16	23156	35343	12188	59	11870	18134	6265	102	25488	38864	13377
17	23114	35285	12172	60	11340	17335	5996	103	25361	38668	13308
18	23467	35811	12345	61	11119	17000	5882	104	24854	37910	13057
19	24033	36666	12634	62	10916	16692	5777	105	24903	37978	13076
20	24144	36831	12688	63	10372	15872	5501	106	24838	37882	13045
21	24060	36709	12650	64	9936	15214	5279	107	24424	37257	12834
22	24844	37888	13045	65	10372	15872	5501	108	24232	36965	12734
23	24971	38079	13109	66	10916	16692	5777	109	24136	36822	12687
24	24786	37808	13023	67	11119	17000	5882	110	23528	35903	12376
25	25297	38571	13275	68	11340	17335	5996	111	23283	35536	12254
26	25380	38700	13321	69	11870	18134	6265	112	23163	35355	12193
27	25865	39429	13565	70	12054	18414	6361	113	22602	34507	11906
28	25468	38836	13369	71	12367	18887	6521	114	22306	34055	11750
29	25574	39001	13428	72	12814	19561	6748	115	22245	33963	11719
30	25684	39164	13481	73	13090	19979	6890	116	21516	32861	11346
31	25402	38748	13347	74	13374	20409	7036	117	21049	32157	11109
32	20866	31729	10864	75	13849	21125	7277	118	20476	31294	10819
33	20619	31355	10737	76	14132	21553	7422	119	20130	30766	10637
34	20452	31101	10650	77	14758	22496	7739	120	19600	29969	10370
35	19856	30203	10348	78	14914	22734	7821	121	18984	29035	10052
36	19634	29866	10233	79	15105	23024	7920	122	18622	28490	9869
37	19447	29582	10136	80	10155	15418	5264	123	18178	27819	9642
38	18976	28872	9897	81	15776	24038	8263	124	17507	26807	9301
39	18524	28190	9667	82	16136	24582	8447	125	16938	25953	9016
40	18090	27535	9446	83	16607	25292	8686	126	16235	24887	8653
41	17999	27395	9397	84	17002	25888	8887	127	15242	23396	8155
42	17686	26922	9237	85	17034	25940	8907	128	9936	15214	5279
43	17028	25931	8904	86	17658	26880	9223				

Table 3: Number of triangles, number of edges, and number of vertices in each of 128 unstructured meshes (for 128 frames shown in Figure 1).

Mesh Order	Our SAW			H-B-G's SAW		No SAW	
	FIFO	FIFO($\sqrt{N}$)	LIFO	BFS	DFS	QT-Track	QT-Only
Original Order	238.63	238.12	251.90	238.7	245.76	293.85	297.14
Diagonal Order	237.73	240.07	257.17	240.3	245.46	258.99	262.49
RCM Order	231.5	231.18	246.44	231.87	237.98	249.27	255.96
FIFO Order	226.61	227.42	242.94	227.47	231.87	242.06	250.29
FIFO Order($\sqrt{N}$)	228.36	227.76	241.92	226.86	232.05	238.87	246.97
LIFO Order	229.42	229.11	237.11	229.12	238.44	250.57	258.76
BFS Order	229.62	227.59	236.83	228.57	233.27	241.99	249.96
DFS Order	228.22	227.65	233.31	228.16	233.57	245.11	252.69

Table 4: The average execution time in millisecond of doing mesh intersections for every two consecutive meshes in the 128 frames of engine combustion.

of a quadtree every time to find nearby triangles in the first mesh, denoted by QT-Only. The average execution time was obtained by computing mesh intersections for every two consecutive meshes in the 128 frames of engine combustion, and then averaging these 128 lengths of execution time. We examine the results obtained in the following.

First, the results show that using any SAW improves the execution time of calculating mesh intersections by 8% to 20%, depending on the mesh ordering, compared with not using a SAW. This performance improvement is due to the connectivity of the SAW sequence. In a SAW sequence, two consecutive triangles are connected by an edge or a vertex; therefore, we can use their neighboring relationship to reduce the searching time.

Second, our FIFO SAW, FIFO($\sqrt{N}$) SAW, and Heber-Biswas-Gao's (H-B-G's for short) BFS SAW are the most competitive SAW's for all mesh orderings listed in the first dimension of Table 4, as these three SAW's are all based on some kind of breadth first search using certain FIFO queues. The differences in the execution times required when using these three SAW's are less than 1%. Our LIFO SAW performs the worst, which is about 5% slower than when using other SAW's. The performance differences among the different SAW's can be ascribed to the data locality property of these SAW's.

Recall that as shown in Table 1, the average distances of FIFO SAW, FIFO($\sqrt{N}$) SAW, and BFS SAW are much less than those of DFS SAW and LIFO SAW. Comparing the execution times shown in Table 4, we can see the effect of this average-distance factor with

these five SAW's. However, the average distance is not the absolute factor affecting data locality. In effect, the average distances of the diagonal ordering and RCM ordering are less than those of FIFO SAW and H-B-G's SAW's; however, the former execution times are worse than the latter execution times, as our algorithm is favored to follow a SAW sequence in the second mesh to calculate mesh intersection. The diagonal ordering has the minimum average distance; however, its execution times are worse than those of the RCM ordering. A diagonal ordering is generated according to the triangles' spatially coordinate information, but an RCM ordering is generated based on the graph data structure of the mesh. Therefore, the RCM ordering has better data locality behavior than the diagonal ordering.

When calculating mesh intersection, the mesh ordering (the numbering of triangles in a mesh) has little influence when a SAW sequence is followed in the second mesh. However, the mesh ordering plays an important role when no SAW sequence is followed. In the last column of Table 4, the execution times required by QT-Track are always better than those required by QT-Only. This indicates that all special purpose mesh orderings have some kind of data locality. As for the cases in which QT-Track is used, we can see that when a mesh adopts the numbering of different SAW sequences, the execution time is 11% to 18% faster than that when the original ordering is adopted. This is because the reordered mesh takes advantage of the data locality of that ordering. We can see that for mesh intersections, the best orderings are our FIFO SAW, FIFO($\sqrt{N}$) SAW, and H-B-G's BFS SAW, all of which have better data locality.

Note that despite the fact that our LIFO SAW has the longest average distance, there are some potential applications as mentioned in [8]. For example, walks have a long tradition of applications in Monte Carlo methods used to study long-chain polymer molecules. This new application, however, is beyond the scope of this presentation.

6 Conclusion

In this paper, we have presented an efficient algorithm for computing mesh intersections. We have found that if $\triangle_2$ and $\triangle_2'$ are adjacent to each other in the second unstructured mesh, their triangle-intersection sets with respect to the triangles in the first unstructured mesh have non-empty intersections. Thus, we can follow a SAW sequence to use the local information of a preceding triangle-intersection set to generate the succeeding triangle-intersection set. We can also use a background quadtree to keep track of the location of each triangle in the SAW sequence. This allows us to design a linear time algorithm for computing mesh intersections. According to the results of experimental studies, our algorithm is superior than other naive algorithms in terms of execution time.

For an unstructured mesh, the data of the logically neighboring triangles are not stored together in the physical memory. Thus, data locality has a large impact on performance. We have presented a FIFO SAW and a LIFO SAW. The FIFO SAW employs better data locality, and by using it, we can reduce the execution time by 5% compared with using other SAW's. We believe that this is due to the effect of data locality when operating under hierarchical-memory computer architectures, which result in more page hits (and thus less page faults) and more cache hits (and thus less cache misses).

References

[1] E. Barszcz, S. K. Weeratunga, and R. L. Meakin. Dynamic overset grid communication on distributed memory parallel processors. AIAA paper 93–3311, American Institute of Aeronautics and Astronautics, 1993.

[2] D. A. Burgess and M. B. Giles. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. *Advances in Engineering Software*, 28:189–201, 1997.

[3] J. J. Chattot and Y. Wong. Improved treatment of intersection bodies with the Chimera method and validation with a simple and fast flow solver. *Computers and Fluids*, 27(5–6):721–740, 1998.

[4] G. Chesshire and W. D. Henshaw. Composite overlapping meshes for the solution of partial differential equations. *Journal of Computational Physics*, 90:1–64, 1990.

[5] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th Nat. Conf. of the ACM*, pages 157–172, 1969.

[6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, 1997.

[7] O. Hassan, E. J. Probert, K. Morgan, and N. P. Weatherill. Unsteady flow simulation using unstructured meshes. *Comput. Methods Appl. Mech. Engrg.*, 189:1247–1275, 2000.

[8] G. Heber, R. Biswas, and G. R. Gao. Self-avoiding walks over adaptive unstructured grids. *Concurrency: Practice and Experience*, 12:85–109, 2000.

[9] P.-Z. Lee and C.-H. Chang. Unstructured mesh generation using automatic point insertion and local refinement. In *Proc. National Computer Symposium*, pages B550–B557, Taipei, Taiwan, Dec. 1999.

[10] P.-Z. Lee, C.-H. Chang, and M.-J. Chao. A parallel Euler solver on unstructured meshes. In *Proc. ISCA 13th International Conference on Parallel and Distributed Computing Systems*, pages 171–177, Las Vegas, Nevada, Aug. 2000.

[11] W.-H. Liu and A. H. Sherman. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, Apr. 1976.

[12] R. Löhner. Some useful data structures for the generation of unstructured grids. *Communications in Applied Numerical Methods*, 4:123–135, 1988.

[13] R. Löhner. Adaptive remeshing for transient problems. *Computer Methods in Applied Mechanics and Engineering*, 75:195–214, 1989.

[14] M. M. Maricq, D. H. Podsiadlik, D. D. Brehob, and M. Haghgooie. Particulate emissions from a direct-injection spark-ignition (DISI) engine. SAE Technical Paper 1999-01-1530, 1999.

[15] R. L. Meakin. A new method for establishing intergrid communication among systems of overset grids. AIAA paper 91–1586, American Institute of Aeronautics and Astronautics, 1991.

[16] L. Oliker, X. Li, G. Heber, and R. Biswas. Parallel conjugate gradient: Effects of ordering strategies, programming paradigms, and architectural platforms. In *Proc. ISCA 13th International Conf. on Parallel and Distributed Computing Systems*, pages 178–185, Las Vegas, Nevada, Aug. 2000.

[17] S. Plimpton, B. Hendrickson, and J. Stewart. A parallel rendezvous algorithm for interpolation between multiple grids. In *Proc. Supercomputing'98,* available via WWW at http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Plimpton644/index.htm, Orlando, FL, Nov. 1998.

[18] N. C. Prewitt, D. M. Belk, and W. Shyy. Parallel computing of overset grids for aerodynamic problems with moving objects. *Progress in Aerospace Sciences*, 36:117–172, 2000.

[19] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1998.

[20] J. L. Steger, F. C. Dougherty, and J. A. Benek. A Chimera grid scheme. *ASME FED*, 5:59–69, 1983.

[21] J. F. Thompson, B. K. Soni, and N. P. Weatherill, editors. *Handbook of Grid Generation*. CRC Press, Boca Raton, FL, 1999.

[22] A. M. Wissink and R. L. Meakin. On parallel implementations of dynamic overset grid methods. In *Proc. Supercomputing'97,* available via WWW at http://www.supercomp.org/sc97/proceedings/TECH/WISSINK/INDEX.HTM, San Jose, CA, Nov. 1997.