Computer-Aided Design 115 (2019) 161–171

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

Parallelizing discrete geodesic algorithms with perfect efficiency☆,☆☆

Xiang Ying [a], Caibao Huang [a], Xuzhou Fu [a], Ying He [b,*], Ruiguo Yu [a,*], Jianrong Wang [a], Mei Yu [a]

[a] College of Intelligence and Computing, Tianjin University, Tianjin, China
[b] School of Computer Science & Engineering, Nanyang Technological University, Singapore

ARTICLE INFO

Article history:
Received 26 April 2019
Accepted 4 May 2019

Keywords:
Discrete geodesics
Autonomous wavefront propagation
The Chen–Han algorithm
The fast marching method
Parallel algorithm
Perfect efficiency

ABSTRACT

This paper presents a new method for parallelizing geodesic algorithms on triangle meshes. Using the half-edge data structure, we define the propagation dependency graph to characterize data dependency in computing geodesics. Then, we design an *active* strategy such that the vertices and half-edges on the wavefront take the initiative to collect their input data and then propagate windows and update geodesic information in their own memory space. As a result, all the read and write operations can be carried out simultaneously. Our method, named AWP, works for both exact (e.g., the CH algorithm) and approximate (e.g., the fast marching method) geodesic algorithms. Our implementation on various NVIDIA GPUs exhibit perfect linear speedup, i.e., doubling the computational power (i.e., FLOPS) doubles the speed. We prove that the AWP-CH algorithm runs in $O(n^2/\min(C, n))$ time, where n and C are the numbers of faces and cores, respectively. Evaluation on GTX Titan XP shows that AWP-CH empirically runs in n^p time, $p \in [1.25, 1.35]$, for real-world models with $n \leq 10^7$ and anisotropy measure $\tau \leq 2.0$. Thanks to its perfect efficiency and the trend of increasing the number of processors in graphics hardware, we believe that the actual performance of AWP can be further improved in the near future.

1. Introduction

Computing geodesic distances on triangle meshes is a fundamental problem in computational geometry and computer graphics. As proposed in Mitchell et al.'s seminal paper [1], the discrete geodesic problem can be solved by discretizing wavefront into a set of intervals on mesh edges, which are called windows, and propagating them in a Dijkstra-like sweep. Given a manifold triangle mesh with n vertices, two notable algorithms, the Mitchell–Mount–Papadimitriou (MMP) algorithm [1] and the Chen–Han (CH) algorithm [2], can compute exact geodesic distances in $O(n^2 \log n)$ and $O(n^2)$ time, respectively. However, the high computational cost diminishes their application to large-scale models.

Given the importance of discrete geodesics, there has been a considerable effort to accelerate the MMP and CH algorithms. One typical way is to design simple rules involved only *local* information to prune redundant windows, such as [3–7]. Another approach is to develop effective ways to organize windows so that multiple windows can be propagated at a time, e.g., [7,8]. The VTP algorithm [7], elegantly combining the two types of techniques, is so far the most efficient algorithm for exact geodesics and it also incurs the least memory usage. However, due to the *tight* lower bound $\Omega(n^{1.5})$ of window complexity [9], none of the algorithms in the MMP/CH family is able to run faster than $O(n^{1.5})$. Moreover, limited by the physical bottlenecks of CPUs, the rate of progress of CPU speed would reach saturation, implying that the room for acceleration becomes less and less.

On the other hand, there is a growing trend of increasing the number of processors, especially for graphics hardware. This paper aims at improving the performance of the window propagation algorithm by GPU parallelization. This problem, however, is challenging due to 1) complicated data dependencies, such as distance update, window clipping, and window propagation, and 2) conflicts between the order of window processing and parallelism. Specifically, to maintain the best quality of wavefront, one always processes the window which is closest to the source at a time. On the other hand, when a large number of windows are propagated simultaneously, the wavefront quality is often poor,

☆ This paper has been recommended for acceptance by Pierre Alliez, Yong-Jin Liu & Xin Li.

☆☆ No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.cad.2019.05.023.

* Corresponding authors.
E-mail addresses: xiang.ying@tju.edu.cn (X. Ying), cbhuang@tju.edu.cn (C. Huang), fuxuzhou@tju.edu.cn (X. Fu), yhe@ntu.edu.sg (Y. He), rgyu@tju.edu.cn (R. Yu), wjr@tju.edu.cn (J. Wang), yumei@tju.edu.cn (M. Yu).

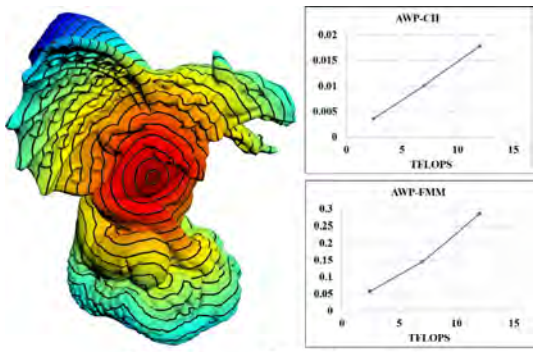
https://doi.org/10.1016/j.cad.2019.05.023
0010-4485/

resulting in many redundant windows which do not carry the shortest distances at all. Therefore, one needs to find a tradeoff between the throughput and the wavefront quality.

To our knowledge, the only parallel geodesic algorithm for triangle meshes is the parallel Chen–Han (PCH) algorithm [10]. It divides the classic CH algorithm into 4 phases, window selection, window propagation, data organization, and events processing, in which the operations have no data dependency so that they can be carried out simultaneously. It also adopts a simple window selection strategy to maintain the wavefront quality. PCH sacrifices some of the ordered nature of wavefront propagation to process a large number of windows at a time. However, PCH is not entirely parallel, since the data organization phase is executed on the CPU. Computational results show that the sequential part of PCH takes 20%–25% of the running time for models with 1 million vertices. By Amdahl's law [11], the speedup of a parallel program is limited by the sequential fraction of the program. Since only 75%–80% of PCH can be parallelized, the theoretical maximum speedup using parallel computing would be 5×, no matter how many processors are used.

To overcome the limitations of PCH, we present an entirely new parallelization framework, called autonomous wavefront propagation (AWP). Using the half-edge data structure, we define the propagation dependency graph to characterize data dependency in computing geodesics. Then, we design an *active* strategy such that the vertices and half-edges on the wavefront take the initiative to collect their input data and then propagate windows and update geodesic information in their own memory space. As a result, all the read and write operations can be carried out simultaneously. Thanks to its active nature, AWP is fully parallel, and it has perfect efficiency. Moreover, AWP is a general framework in that it works for both exact algorithm (e.g., the CH algorithm [2]) and approximate algorithm (e.g., the fast marching method [12]).

We prove that the AWP-CH algorithm runs in $O(n^2/\min(C, n))$ time, where n and C are the numbers of faces and cores, respectively. Our implementation on various NVIDIA GPUs exhibit perfect linear speedup, i.e., doubling the computational power doubles the speed. Evaluation on GTX Titan XP shows that AWP-CH empirically runs in n^p time, $p \in [1.25, 1.35]$, for real-world models with $n \leq 10^7$ and anisotropy measure $\tau \leq 2.0$. Fig. 1 shows the results of our method and popular approaches on 6.9-million-face Gargoyle model. Thanks to its perfect efficiency and the trend of increasing the number of processors in graphics hardware, we believe that the actual performance of AWP can be further improved in the near future.



Method	Time (s)	Error
ICH [4]	1736.99	0%
FWP-ICH [8]	402.13	0%
PCH [10]	176.68	0%
VTP [7]	106.76	0%
AWP-CH	56.29	0%
FMM [12]	20.88	1.73%
AWP-FMM	3.51	1.74%

Fig. 1. AWP enables shared-memory parallelization of popular discrete geodesic algorithms such as the fast marching method and the CH algorithm, with perfect linear speedup, i.e., doubling the computational power (i.e., FLOPS) doubles the speed. The vertical axis shows the inverse of time. AWP-CH runs 2–30 times faster than the existing exact geodesic algorithms on the Gargoyle model with 6.9 million faces. The running time of sequential algorithms (FMM, MMP, ICH, FWP–ICH and VTP) and parallel algorithms (PCH and AWP) are measured on cutting edge CPU (Intel i7-7700k Quad Core 4.2 GHz with 16 GB memory) and GPU (NVIDIA Titan Xp with 3840 CUDA cores), respectively. The MMP and FWP-MMP algorithms fail due to short of memory.

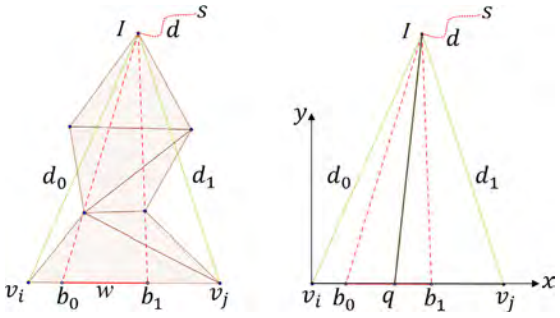


Fig. 2. A window $w = (e_{ij}, b_0, b_1, d_0, d_1, d)$ is an interval associated to a half-edge $e_{ij} = (v_i, v_j)$. The pseudo source I is the last saddle vertex on the face sequence, and d is the geodesic distance between source s and pseudo source I. Parametrizing w to $\mathbb{R}^2$, we can compute the geodesic distance for any point $q \in w$ as $d(s, q) = d + \|Iq\|$.

2. Background and related work

There is a large body of literature in discrete geodesics. We refer readers to [13–15] for comprehensive surveys. In this section, we review the most relevant work according to their sequential/parallel nature.

2.1. Sequential algorithms

Given a source vertex $s \in V$, imagine that we place a light at s, and then visualize geodesic paths as light rays emanating from s in all tangent directions. To track together groups of shortest paths that can be parameterized atomically, the algorithm partition each mesh edge into a set of intervals called windows [3]. A window is a segment on an edge such that the geodesic path to any point on the segment trespasses the *same* sequence of faces and vertices. See Fig. 2. The last saddle vertex[1] (if exists) on the face/vertex sequence is called *pseudo-source*. Mathematically speaking, a window w associated to a half-edge $e_{ij} = (v_i, v_j)$ is a 6-tuple $(e_{ij}, b_0, b_1, d_0, d_1, d)$, where b_0, b_1 are the endpoints of w, d_0, d_1 are the distances from the pseudo-source to v_i and v_j, and d is the distance from the source to the pseudo-source.

Most sequential algorithms adopt a Dijkstra-like framework: starting from the source, the algorithm gradually advances the discrete wavefront. In each iteration, it propagates one or more active windows across triangular faces and updates the distance for vertices that are lit by windows. The algorithm continues until wavefront vanishes and there are no active windows.

They differ in the way of window organization and the order of window propagation. MMP [1] and ICH [4] maintain windows in a priority queue, and only the window closest to the source is propagated at a time. Due to few redundant windows produced, they have the "smoothest" wavefront. However, since updating

[1] A saddle vertex is a vertex with negative Gaussian curvature.

priority queue takes $O(\log n)$ time in each iteration, they have $O(n^2 \log n)$ time complexity. CH [2] organizes windows in a tree structure and processes them in a breadth-first-search order. Again, only one window is propagated in each iteration. Although CH has a theoretical $O(n^2)$ time complexity, its runtime performance is not as good as those using a priority queue. FWP [8] stores windows in a FIFO queue and adopts a label-correcting scheme to determine the active windows.

In contrast to the above-mentioned algorithms, VTP [7] adopts triangles instead of windows as the primitive. In each iteration, it propagates a collection of windows from one triangle edge to its two opposite edges. Using an exhaustive list of rules, VTP can effectively identify redundant windows, hereby significantly reducing computational cost and memory usage at later stages.

The fast marching method [16] is a popular numerical method for solving the Eikonal equation. Starting from the known information, i.e. the boundary values, it builds a solution outwards strictly following the causality of the equation. The mesh-based FMM [12] can be considered as a simplified CH/MMP algorithm if taking the *entire* edge as a window.

2.2. Parallel algorithms

Weber et al. [17] proposed parallel marching method (PMM), a raster scan-based version of the FMM on geometry images. They showed that PMM has a linear time complexity and it outperforms the sequential FMM by several orders of magnitude. However, such a nice parallel structure exists only in the Cartesian grids. In practice, it is expensive to construct geometry images for surfaces of complicated geometry and/or topology. The PCH algorithm [10] is the only known parallel algorithm for exact geodesic distances. As mentioned above, PCH is not fully parallel: when mesh complexity increases, its sequential part becomes dominant, leading to a capped speedup. There are also a few approaches for parallelizing the fast marching method, such as [18–20]. However, these methods are limited to regular grids.

2.3. Other algorithms

There are other methods with mixed sequential and parallel features. The heat method [21] computes geodesic distances by solving a pair of standard linear elliptic problems, which can be pre-factored once and subsequentially solved in near-linear time. The graph-based methods, such as saddle vertex graph method [22] and discrete geodesic graph [23], encode geodesic information into a sparse, undirected graph, which is constructed in parallel, then compute geodesic distances using Dijkstra's algorithm.

3. Autonomous wavefront propagation

3.1. Propagation dependency graph

Denote by M a manifold triangle mesh. Let V, E and F be its vertex, edge and face sets. We represent M by the half-edge data structure, i.e., each edge is split into two half-edges with opposite directions and the three half-edges that border a triangular face form a counterclockwise loop. To simplify the presentation, we assume that M is closed, i.e., every half-edge borders a face, and consider the single-source geodesic distances. Our method can be easily extended to meshes with boundaries and multiple sources.

Consider a half-edge e bordering a face f. Denote by $v(e) \in f$ the vertex facing e, $e_l(e)$ (resp. $e_r(e)$) the half-edge bordering the face on the left (resp. right) side of f (see Fig. 3(a)). The geodesic distance of half-edge e comes from the current windows on e, the opposite vertex $v(e)$ and the two half-edges $e_l(e)$ and $e_r(e)$ bordering f's neighboring triangles. If one of these elements is changed, we update the windows on e. Similarly, the geodesic distance of vertex v is determined by its incident and opposite half-edges (see Fig. 3(b)).

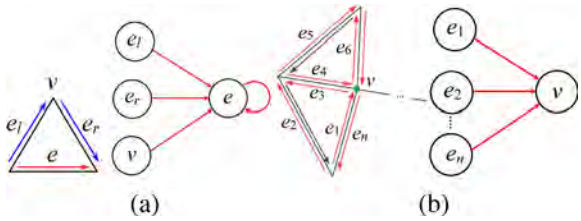


Fig. 3. Propagation dependency of half-edges (a) and vertices (b). See the text for explanation.

We encode the propagation dependency into a directed graph to characterize data flow in geodesic computation. Such a graph G, called *Propagation Dependency Graph* (PDG), has $|V| + 2|E|$ nodes and $6|E|$ edges, where each node is a vertex or a half-edge, and each directed edge is one of the above-mentioned dependencies. Note that we do not store PDG explicitly since all the dependencies are readily available from the half-edge data structure.

3.2. Algorithm

Algorithm 1 Autonomous Wavefront Propagation

Input: Manifold triangle mesh $M = (V, E, F)$ and source $s \in V$
Output: Geodesic distance for each vertex $v \in V$

```
Initialize the global variables;
for each opposite half-edge e = (v_i, v_j) of s do
    Create a window w and add it to rBuffer;
    rEdgeStatus[id(e)] = active;
    rVertexStatus[i] =rVertexStatus[j] = active;
    rWinCnt[id(e)] = 1;
    rWinAddr[id(e)] = sizeof(window);
end for
exclusive_scan(rWinAddr);        ▷ Call CUDA's prefix sum to compute the addresses of windows
while rBuffer is not empty do
    parallel Allocate space in write buffer;
    exclusive_scan(wWinAddr);
    parallel Update distances;
    parallel Propagate windows;
    parallel Collect windows;
    swap the read and write buffers;
end while
```

We call the to-be-propagated windows *active* windows. After propagating, an active window becomes inactive. We call a vertex $v \in V$ *active*, if it generates active window(s) in the current iteration; We call a half-edge e *active*, if it contains active window(s).

If window $w \subseteq e$ covers some vertex, say $u \in V$, and updates the geodesic distance at u, we say e *activates vertex* u. If a vertex v or a half-edge e generates window on half-edge e', we say v *or* e *activates half-edge* e'.

With propagation dependency graph, each active vertex/half-edge takes the initiative to update geodesic distance from its inward paths and then provides the result to nodes through the outward paths. We maintain two separate buffers to store windows, for the read and write operations, respectively. The read buffer contains windows for the active elements, and multiple

active elements can read the same window without conflict. Each active element then writes the output in its own space in the write buffer. Therefore, we can process all active elements in a completely independent manner. Due to its autonomous nature, we call our algorithm *autonomous wavefront propagation*.

AWP defines the following global variables to organize windows:

- r/wWinCnt[2|E|]: the number of windows on a half-edge;
- r/wWinAddr[2|E|]: the address of the first window on a half-edge;
- r/wEdgeStatus[2|E|]: boolean value to indicate whether a half-edge is active;
- r/wVertexStatus[|V|]: boolean value to indicate whether a vertex is active;
- r/wBuffer[]: memory buffer for active windows;
- LeftChn[], RightChn[]: memory buffer for left and right children propagated from active windows.

We use the prefixes r and w to distinguish the variables for the read and write operations. We denote by $id(x)$ the index of an element x (either a half-edge or a vertex).

Let $s \in V$ be the source vertex. Initially, for each half-edge e facing s, we create a window and activate e. These active half-edges then activate their dependent vertices. Taking these active half-edges and vertices as initial active elements, the algorithm enters a loop to process active elements. Each iteration consists of the following 4 steps:

(Step 1) **Edge oriented memory allocation.** Each thread processes an active half-edge e. If $v(e)$, $e_l(e)$ or $e_r(e)$, is active, new windows will be generated on e in this iteration. As a result, we allocate memory in the write buffer to store them. Observe that each active vertex produces a window for its opposite edge, and each active edge produces a window on its left and right half-edges. So we can estimate the maximal number of windows generated for each half-edge. After all active half-edges have been processed, we call CUDA's prefix sum function to compute window addresses.

(Step 2) **Vertex oriented distance update.** Each thread processes an active vertex v. If v is a saddle vertex and its distance is updated, we create a child window for every opposite side of v and update the window count and address for the write buffer.

(Step 3) **Edge oriented window propagation.** Each thread takes an active half-edge e, propagates all its windows, and adds the child windows into LeftChn and RightChn buffers.

(Step 4) **Edge oriented window collection.** Each thread takes an active half-edge e, collects its windows from $e_l(e)$'s RightChn and $e_r(e)$'s LeftChn, then write them into the write buffer and activates elements for the next iteration.

The algorithm continues until the read buffer is empty, i.e., no new windows are generated. Algorithm 1 shows the pseudo-code of AWP and Fig. 5 illustrates a typical iteration of AWP on a toy model. Note that PDG is not loop-free, due to the dependency relation between vertices and half-edges in a triangle. For example, the loop $v_3 \rightarrow e_4 \rightarrow v_2 \rightarrow e_5 \rightarrow v_3$ for the toy mesh. The loops in PDG, however, do not lead to endless loop in window propagation. The reason is a vertex v propagates windows only when v is saddle and its distance has just been updated (see line 11 in function VertexOrientedDistanceUpdate. If the distance does not update any longer, the loop is then broken.

3.3. Time complexity

The core of AWP is the propagation dependency graph, which models the discrete geodesic problem as information propagation on directed graphs. In each iteration, AWP passes geodesic distances (carried by windows) of the current level to the next level. For a triangle mesh with n faces, PDG has at most n levels, since the longest geodesic may span n faces. Therefore, AWP terminates in at most n iterations. For real-world meshes, the longest geodesic is of length $O(\sqrt{n})$, so AWP terminates in empirical $\sqrt{n}$ iterations. In each iteration, all active half-edges are processed independently; therefore, windows are also propagated in a fully parallel manner without writing conflicts.

It is worth noting that this active updating strategy is not available to PCH. In one iteration of PCH, several windows may light a common vertex v, so they will update the geodesic distance of v simultaneously, causing writing conflicts. Therefore, PCH is not fully parallel.

In each iteration, there are at most n windows produced. Note that the windows associated to a half-edge are stored in contiguous memory, so we can divide the windows evenly to GPU cores. Thus, each iteration takes $O(n/\min(C, n))$ time, where C is the number of cores. Putting it all together, AWP has a theoretical time complexity $O(n^2/\min(C, n))$.

3.4. Window complexity

For the ith iteration, the function EdgeOrientedWindowPropagation takes the windows generated by the $(i-1)$th iteration as input, and then filter them by the local window filters. The newly created windows either propagate and generate child windows, or they are deleted by the filters. Therefore, each window is processed only once, which can guarantee that there are no more than $O(n^2)$ windows produced by AWP (see Fig. 6).

```
1: function EdgeOrientedMemoryAllocation(e)
2:     wWinAddr[id(e)] = 0;
3:     if rVertexStatus[id(v(e))] == active then
4:         wEdgeStatus[id(e)] = active;
5:         wWinAddr[id(e)] += sizeof(window);
6:         wWinCnt[id(e)] = 0;
7:     end if
8:     if rEdgeStatus[id(e_l)] == active then
9:         wEdgeStatus[id(e)] = active;
10:        for each window w ⊆ e_l do  ▷ w produces child(ren) on e
11:            wWinAddr[id(e)] += sizeof(window) ;
12:            wWinCnt[id(e)] = 0;
13:        end for
14:    end if
15:    Process the right half-edge e_r in the same way
16: end function
```

4. Implementation

AWP is a general framework for parallelizing discrete geodesic algorithms. This subsection presents the implementation details for two popular algorithms, the CH algorithm [2] and the fast marching method [12].

4.1. AWP-based CH algorithm

The CH algorithm [2] is a classic exact algorithm. It organizes windows in a tree structure and propagates windows in the breadth-first-search order. Propagating a window produces one

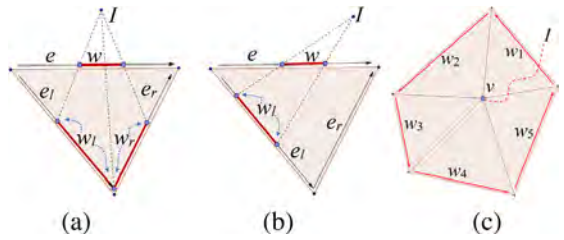


Fig. 4. Window propagation. (a)–(b) Propagating a window $w \subseteq e$ produces child window(s) on e's left and/or right half-edges. (c) When the distance of a saddle vertex v is updated, we create a window for each opposite half-edge of v, and replace the pseudo source I by v.

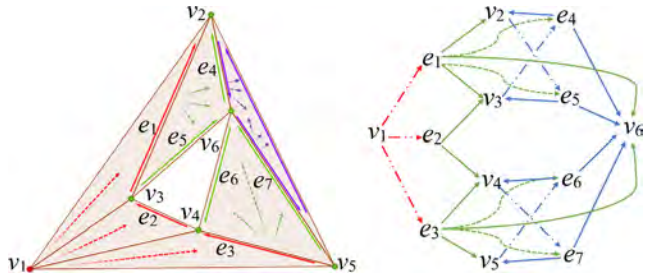


Fig. 5. Illustrating AWP on a 6-vertex toy mesh. Let v_1 be the source vertex. At the beginning, set $d(v_1) = 0$ and $d(v_i) = \infty$, $2 \le i \le 6$, and create a window to each opposite half-edge. Now there are 3 active half-edges e_i, $1 \le i \le 3$ and 4 active vertices v_i, $2 \le i \le 5$. Then, we illustrate one iteration of AWP. Each active vertex updates its distances from its parent half-edges and writes their windows into the write buffer (step 2). Next, each active half-edge e_i, $1 \le i \le 3$ propagates its windows to neighboring faces (step 3). For simplicity, we skip memory management (steps 1 and 4) in the description and show only the most important half-edges in this figure. Color scheme: red — initialization; green — the current iteration; purple — the next iteration.

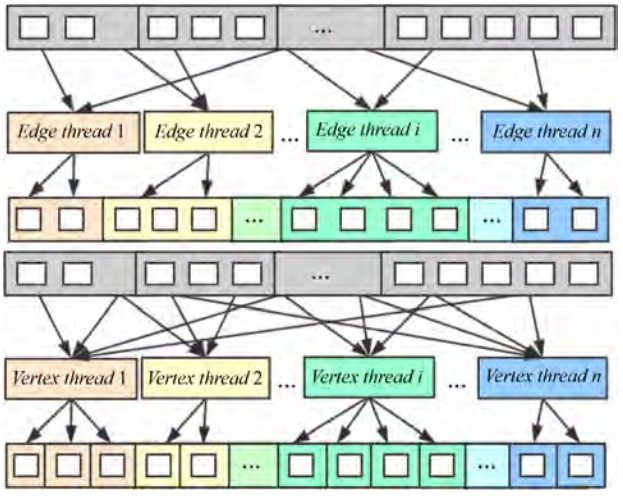


Fig. 6. Each edge/vertex thread gets its input from the read buffer (gray), and then writes its output in its own space in the write buffer. Since there is no write conflict, all threads can be carried out in a fully parallel manner. The white rectangles are windows.

or two child windows (see Fig. 4(a) and (b)). To avoid exponential explosion in the tree, Chen and Han [2] adopted a simple one-angle-one-split window filter: if two windows light a vertex, only one can have two children. See Fig. 7(a). Since there are at most $O(n)$ windows in each level and the depth of the tree cannot exceed the number of faces of M, the CH algorithm runs in $O(n^2)$ time. Also, it stores only the $O(n)$ key windows that light vertices, leading to an $O(n)$ space complexity.

```
1: function VertexOrientedDistanceUpdate(v)
2:     Updated = false;
3:     for each half-edge e on which v depends do
4:         for each window w ∈ e do
5:             if w provides a shorter distance to v then
6:                 Update v's distance;
7:                 Updated = true;
8:             end if
9:         end for
10:    end for
11:    if v is saddle and Updated == true then
12:        for each v's opposite half-edge e do
13:            Create a window w for the entire half-edge e;
14:            wBuffer[wWinAddr[j]+wWinCnt[j] × s] = w;
15:            wWinCnt[j] ++;      ▷ j = id(e); s = sizeof(window);
16:        end for
17:    end if
18: end function
```

```
1: function EdgeOrientedWindowPropagation(e)
2:     LeftWinCnt=RightWinCnt=0;
3:     for each window w ⊆ e do
4:         Propagate w;            ▷ j = id(e); s = sizeof(window);
5:         for each child window w′ do
6:             if w′ belongs to the left half-edge e_l then
7:                 LeftChn[rWinAddr[j]+s×LeftWinCnt]= w′;
8:                 LeftWinCnt ++;
9:             else
10:                RightChn[rWinAddr[j]+s×RightWinCnt]= w′;
11:                RightWinCnt ++;
12:            end if
13:        end for
14:    end for
15: end function
```

```
1: function EdgeOrientedWindowCollection(e)
2:     if rEdgeStatus[id(e_l(e))] == true then
3:         for each window w ∈ LeftChn do          ▷ j = id(e);
   s = sizeof(window);
4:             wBuffer[wWinAddr[j]+s×wWinCnt[j]]= w;
5:             wWinCnt[j] ++;
6:             if w updates the distance of e's endpoint p then
7:                 wVertexStatus[id(p)] = true;
8:             end if
9:         end for
10:    end if
11:    Process the right half-edge e_r(e) in the same way
12: end function
```

Xin and Wang [4] observed that the majority of windows produced in the CH algorithm do not contribute to the shortest distance. To prune the redundant windows, they proposed a simple yet effective "checking with vertex" rule (see Fig. 7(b)). Consider a window $w = (I, A, B, d_0, d_1, d) \subseteq (v_0, v_1)$ generated by its parent window on half-edge (v_1, v_2). Let g_i be the *current* shortest distance for vertex v_i, $i = 0, 1, 2$. Denote by $\varepsilon \ge 0$ the accuracy parameter. Window w can be removed, if one of the following inequalities holds,

$$d + \|IB\| + \varepsilon \ge g_0 + \|v_0B\|,$$
$$d + \|IA\| + \varepsilon \ge g_1 + \|v_1A\|,$$

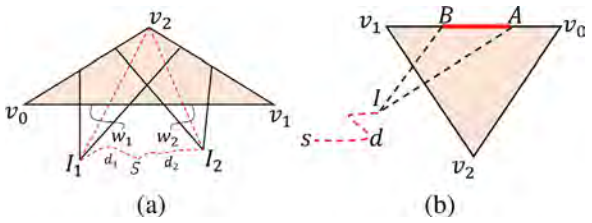


Fig. 7. Window pruning in the CH algorithm. (a) The "one-angle-one-split" filter [Chen and Han [2]]. Although both windows w_1 and w_2 occupy vertex v_2, only the one who provides shorter distance to v_2 has two children. (b) The "checking with vertices" rule [Xin and Wang [4]] can effectively prune redundant windows using local information stored at vertices. See the text for explanation.

$$d + \|IB\| + \varepsilon \geq g_2 + \|v_2B\|.$$

The parameter ε controls the accuracy: the algorithm is exact if $\varepsilon = 0$ [4], and it becomes approximate otherwise [5]. In practice, Xin and Wang [5] suggested $\varepsilon = \lambda\bar{E}$, where $\bar{E}$ is the average edge length and $\lambda \in [0, 1]$ controls the accuracy.

Qin et al. [7] proposed an exhaustive list of rules for pairwise window cross checking, which is highly effective to remove redundant windows. Among them, cases 1, 2 and 3 do not involve data dependency, hereby can be used in AWP.

Since all the above-mentioned window pruning strategies are based on local information only, they can be easily added to function EdgeOrientedWindowProagation. In function EdgeOrientedWindowCollection, we check all windows on a half-edge. If two windows overlap, we adopt MMP's window clipping strategy [1] to partition them.

4.2. AWP-based FMM

The fast marching method [16] is a numerical method for solving boundary value problems of the Eikonal equation on 2D regular grids. The algorithm is similar to Dijkstra's algorithm and uses the fact that information only flows outward from the seeding area. The FMM can be extended to triangulated surfaces [12], unstructured meshes [24] and broken meshes [25].

Imagine that we set a "prairie fire" at the source at time $t = 0$. The fire evolutes towards directions where the surface has not yet been "burnt out". The FMM computes the time values t for each vertex at which the advancing fire front reaches it. It maintains a priority queue Q of all non-source vertices sorted by their times of arrival. The basic operation of fast marching is the update step, which computes the time of arrival of the wavefront to a vertex based on the times of arrival to its neighbors.

There are two common update strategies, based on planar wavefront approximation [12] and spherical wavefront approximation [26]. For the former, a vertex is updated by simulating a planar wavefront propagating inside the triangle, then the values of the two supporting vertices allow to compute the front direction. For the latter, a vertex is updated with its Euclidean distance from a virtual point source, whose coordinates are estimated from the times of arrival to the two supporting vertices.

To parallelize FMM in the proposed AWP framework, we take each half-edge as a window, which is called edge window. Then the window $w = (e_{ij}, d_i, d_j)$ is simply a 3-tuple, where $e_{ij} = (v_i, v_j)$ is an oriented half-edge, and d_i and d_j are the geodesic distances from s to v_i and v_j, respectively.

There are two major differences between FMM and CH. First, propagating a window in FMM produces exactly two edge windows; Second, saddle vertices in FMM do not propagate windows. Keeping these differences in mind, we modify Steps 2 and 3 in the AWP framework to tailor FMM.

```
1: function FMM_VertexOrientedDistanceUpdate(v)
2:     for each opposite half-edge e_ij do
3:         if (e_ij, d_i, d_j) provides v a shorter distance then
4:             Update v's distance;
5:         end if
6:     end for
7: end function
```

```
1: function FMM_OrientedWindowPropagation(e_ij)          ▷ e_ij
   borders face f_ijk = {v_i, v_j, v_k}
2:     Compute the distance for the opposite vertex v_k using d_i
   and d_j;                                  ▷ s = sizeof(edge_window);
3:     LeftChn[rWinAddr[id(e_kj)] + s] = (e_kj, d_k, d_j);
4:     RightChn[rWinAddr[id(e_ik)] + s]= (e_ik, d_i, d_k);
5: end function
```

5. Results & comparison

We implemented AWP-CH (exact), AWP-CH (approximate) and AWP-FMM in CUDA 8.0 (see the supplementary material) and tested our programs on 3 NVIDIA GPUs, including

- GTX 970 with 1664 CUDA cores and 2.44 Tflops,
- GTX Titan X (Maxwell) with 3072 CUDA cores and 7.0 Tflops, and
- Titan Xp with 3840 CUDA cores and 12 Tflops.

We compare our methods with popular discrete geodesic algorithms, MMP[2] [3], ICH[3] [4], PCH[4] [10], FWP[5] [8], and VTP[6] [7]. The timing of the sequential algorithms were measured on a cutting-edge CPU, Intel i7-7700k Quad Core 4.2 GHz with 32 GB memory. This section reports the time performance, efficiency, window and time complexities.

Performance & memory usage. Table 2 in the supplementary material reports the mesh complexity, runtime performance, peak memory (excluding the memory for storing meshes), and the number of window propagations of the testing algorithms. Since the performance of geodesic algorithms is sensitive to the location of source point, we repeat each test with 100 randomly-selected sources and then report the average.

We observe that AWP-CH runs 8–30 times faster than the classic MMP and ICH algorithms, 3–10 times faster than FWP, 1–2 times faster than VTP which is the state-of-the-art.

VTP produces the least number of windows and incurs the least memory usage due to its highly effective window pruning strategy. Rather than controlling the window complexity, AWP-CH aims at improving the time complexity by parallelization. Thanks to its fully parallel nature, AWP-CH is able to process a large number of windows simultaneously. However, the price to pay for such a large throughput is the high number of redundant windows. We observe that AWP-CH propagates 5–10 times more windows than VTP.

The reason that AWP does not show significant advantage over VTP is the difference of computational power of individual CPU core and GPU core. Take the Nvidia Titan Xp and Intel i7-7700k CPU 4.20 GHz as an example. A CPU core is able to process 2.97 million windows per second, whereas a CUDA core can process only 0.072 million windows per second.

2 https://code.google.com/p/geodesic/.
3 https://sites.google.com/site/xinshiqing.
4 http://acm.tju.edu.cn/yingxiang.
5 http://cg.cs.tsinghua.edu.cn/people/~Yongjin.
6 https://github.com/YipengQin/VTP_source_code.

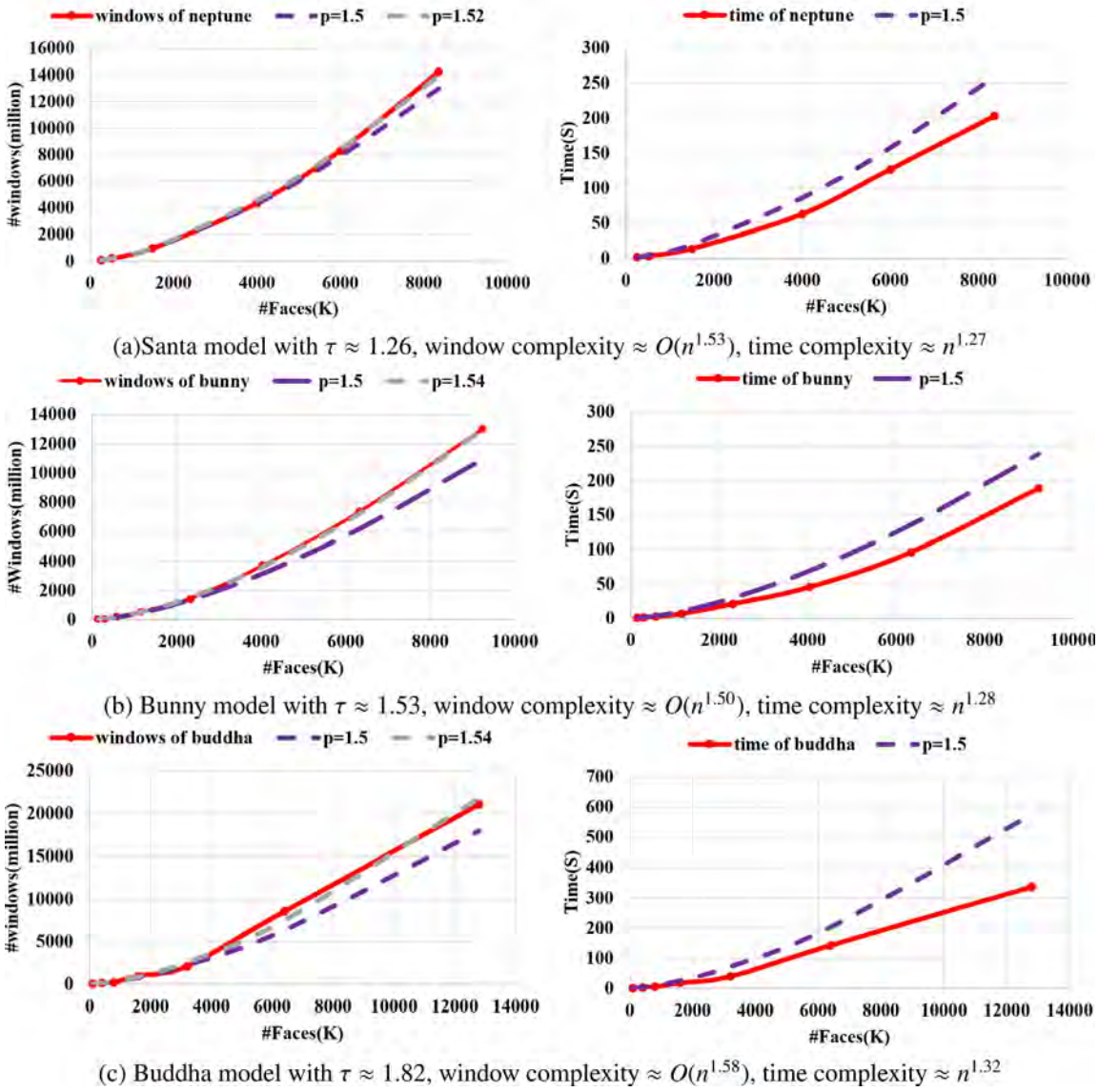


Fig. 8. Window and time complexities of AWP-CH. We show 3 representative models with different anisotropy measures τ. Each model is subdivided into multiple resolutions with τ unchanged. We evaluate both window complexity and time complexity with respect to the growth of resolution. The dashed curves are the fitted $O(n^p)$ curves.

As a member of the CH family, AWP-CH processes at most $O(n)$ windows in an iteration and it is guaranteed to stop after $|F|$ iterations. As a result, the peak memory of AWP-CH is comparable to ICH and FWP-CH. As shown in Fig. 9, the peak memory cost of AWP-CH grows linearly with respect to the growth of mesh resolution (see Fig. 8).

Complexity. We measure the time and window complexities of the testing algorithms. As shown in [9], there are at least $\Omega(n^{1.5})$ windows of a triangle mesh. The lower bound is tight, which happens only when the input mesh is completely isotropic. As a result, none of the exact sequential algorithms is able to break the $O(n^{1.5})$ time complexity. In practice, real-world meshes have window complexity $O(n^p)$, where the exponent $p \in [1.5, 2.0]$ depends on the degree of anisotropy. The higher the degree of anisotropy, the closer p to 2.0.

Similar to [8], we define $\tau(f) = \frac{H}{2\sqrt{3}S}$ the quality measure of a triangular face f, where S and H are the area and perimeter of f. Then we define $\tau = \frac{\sum_{f\in F}\tau(f)}{|F|}$ to measure the anisotropy degree of mesh M. Obviously, $\tau \geq 1$ for all triangle meshes. $\tau = 1$ for isotropic meshes. In general, the larger the value of τ, the higher the degree of anisotropy, hereby the more the number of windows produced.

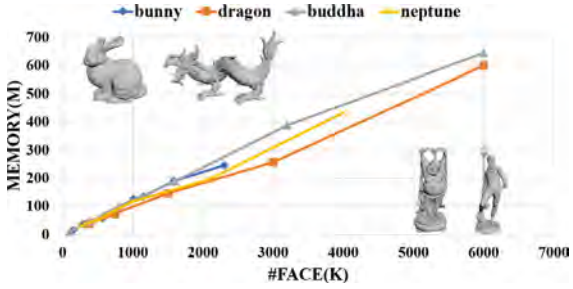


Fig. 9. The peak memory cost of windows for different models with different resolutions.

As shown in Section 3.3, AWP-CH has a worst-case time complexity $O(n^2/\min(C, n))$ when a geodesic path spans $O(n)$ faces. For real-world meshes, the longest geodesic path crosses only

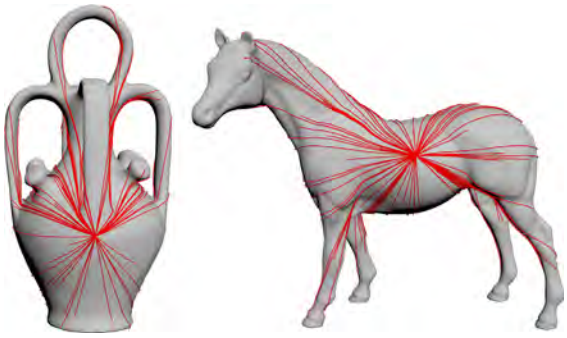


Fig. 10. AWP-CH algorithm can also compute exact geodesic paths.

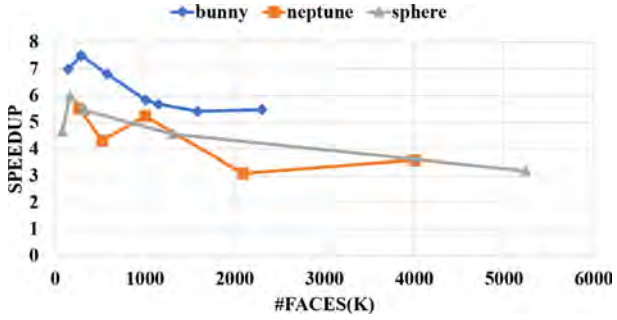


Fig. 11. AWP-FMM runs 3–8 times faster than the sequential FMM. The horizontal and vertical axes show the mesh resolution and the ratio of execution time of FMM to AWP-FMM.

$O(\sqrt{n})$ faces, so AWP-CH runs in an empirical $O(n^{1.5}/\min(C, n))$ time. For the *ideal* case when C is sufficiently large (e.g., $> n$), propagating the wavefront once takes only $O(1)$ time. With n iterations, the wavefront sweeps all mesh vertices and the algorithm stops. Evaluation on GTX Titan XP shows that AWP-CH empirically runs in n^p time, $p \in [1.25, 1.35]$, for real-world models with $n \leq 10^7$ and anisotropy measure $\tau \leq 2.0$ (see Fig. 15(b)(d)(f)). We also observe that AWP-CH produces no more than $O(n^2)$ windows (Fig. 15(a)(c)(e), which matches our theoretical analysis in Section 3.4.

Geodesic paths. As the other exact algorithms, AWP-CH can also compute *exact* geodesic paths. By "one-angle-one-split" rule, each angle in a triangular face is associated one window w, which allows us to back-trace the path to w's pseudo-source. Storing those windows takes only $O(3|F|)$ space. Given an arbitrary vertex $v \in V$, to compute the geodesic path $\gamma(s, v)$, we unfold the sequence of windows starting from the one providing v's shortest distance until reaching the source s. See Fig. 10 for two examples of geodesic paths.

Approximation algorithms. The priority queue based FMM runs in $O(n \log n)$ time, due to the $O(\log n)$ overhead of each iteration. Although there exist faster implementations of FMM, e.g., the linear-time algorithm [27], the highly scalable massively parallel FMM [20], they work only for 2D regular grids. We observe AWP-FMM runs 3–8 times faster than the sequential FMM. We also parallelize the approximate CH algorithm [5] using our AWP framework. The approximate CH and exact CH algorithms differ in an error term $\varepsilon = \lambda \overline{E}$ in the "checking with vertices" rule, where $\overline{E}$ is the average edge length and $\lambda \in [0, 1]$. Setting $\lambda = 0$ and $\lambda = 1$, we obtain the exact CH algorithm and Dijkstra's algorithm, respectively. As Fig. 12 shows, approximate AWP-CH consistently outperforms its sequential counterpart by a factor of 4–9 for all $\lambda \in [0.01, 0.2]$. Fig. 14 reports the root mean square error of the approximate AWP-CH algorithm (see Figs. 11, 13 and 16).

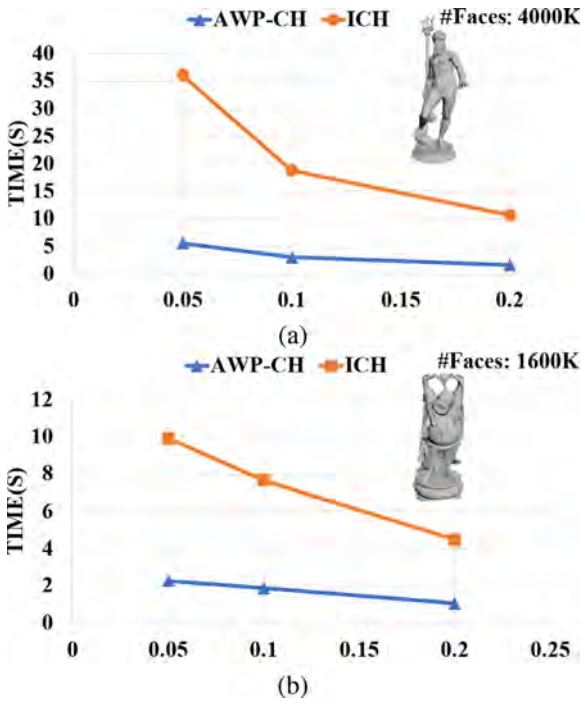
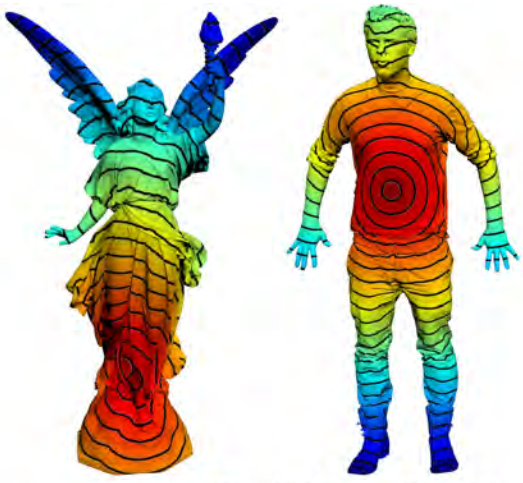


Fig. 12. Performance of approximate AWP-CH and approximate ICH. The horizontal axis shows the accuracy parameter λ. Our AWP algorithm runs 4–9 faster than its sequential counterpart.



	Lucy $\|F\| = 16M$	Samwithbag $\|F\| = 15M$
AWP-CH (exact)	315.97s	198.58s
AWP-CH (approx)	52.67s	30.95s
AWP-FMM	33.87s	17.30s

Fig. 13. Large-scale models. The accuracy parameter for the approximate AWP-CH algorithm is $\lambda = 0.1$.

Efficiency. To quantitatively measure the efficiency, we test AWP-CH and AWP-FMM on NVIDIA GPUs with increasing Tflops. Since our algorithm is fully parallel, the more powerful the GPU, the faster the algorithm, hereby its running time is inversely linearly proportional to the computational power of GPU. Computational results confirm that our method has perfect efficiency (see Fig. 15(a)–(b)).

We also measure the efficiency of PCH [10]. As an "upgraded" version of ICH [4], PCH can propagate a large number of windows in one iteration. At the same time, it uses k-selection to

Table 1
Model complexity and performance comparison with the state-of-the-art geodesic algorithm on running time, peak memory and total number of windows propagations. $|F|$ is the number of faces.

Models	Performance	Algorithms						
		MMP	ICH	FWP-MMP	FWP-CH	VTP	PCH	AWP-CH
Bunny	Time (s)	4.23	5.52	1.40	2.33	0.745	1.04	0.46
	Peak memory (MB)	219.08	2.19	320.54	2.19	1.4	8.73	54.40
($|F|$:144k)	Window count	4,173,450	6,708,539	6,106,040	6,268,906	4,836,790	19,454,764	21,713,401
Golf ball	Time (s)	13.46	30.98	4.13	9.59	1.75	3.68	0.94
	Peak memory (MB)	642.78	2.75	598.23	2.75	2.15	10.04	94.50
($|F|$:244k)	Window count	19,334,362	27,825,888	17,994,177	22,735,411	13,214,211	71,234,273	55,264,195
Armadillo	Time (s)	6.98	8.41	2.65	4.67	1.75	1.95	1.37
	Peak memory (MB)	586.74	5.27	510.14	5.28	1.73	18.23	120.08
($|F|$:346k)	Window count	11,089,425	10,543,826	9,641,604	10,005,269	7,998,976	36,159,081	53,212,243
Sphere	Time (s)	197.71	1540.29	239.5	224.09	40.20	270.17	6.22
	Peak memory (MB)	8062.98	20	6825.30	20.01	28.67	106.22	452.95
($|F|$:1M)	Window count	148,072,389	852,275,198	125,343,035	470,679,673	299,924,498	340,059,807	411,274,370
Santa	Time (s)	87.59	236.82	44.53	65.76	14.47	64.12	6.92
	Peak memory (MB)	8075.51	18.5	7022.19	18.6	9.7	72.38	433.97
($|F|$:1.2M)	Window count	153,767,085	165,532,008	133,710,509	127,128,734	101,800,145	465,125,728	464,688,161
Gargoyle	Time (s)		1398.78		339.97	79.61		55.05
	Peak memory (MB)		85.44		85.44	28.43		1919.58
($|F|$:5.6M)	Window count	Out of	699,045,508	Out of	587,544,410	465,322,440	Out of	1,516,012,094
Lucy	Time (s)	memory	5983.11	memory	1608.25	341.61	memory	186.62
	Peak memory (MB)		220.70		220.7	70.76		4996.05
($|F|$:14.4M)	Window count		3,106,980,366		2,663,347,390	1,911,210,881		17,278,093,986

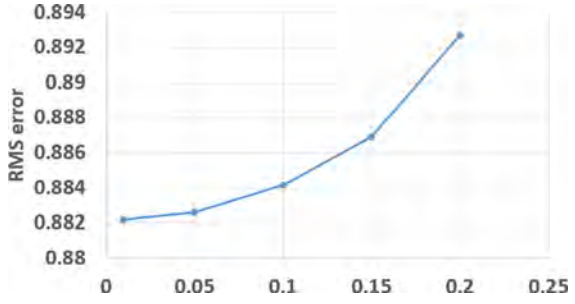


Fig. 14. Root mean square error of the AWP-CH (approximate) algorithm. The horizontal axis shows the accuracy parameter λ.

Table 2
Profiling. Timings are measured in seconds.

Models	Stage					Total time	Core/Total
	Windows collection	Prefix sum	Edge event	Vertex event	Init.		
Bunny $|F|$= 144k	0.1909	0.1495	0.0614	0.0427	0.0111	0.4557	64.76%
Golf ball $|F|$= 244k	0.4857	0.2779	0.1364	0.1158	0.0183	1.0341	71.36%
Armadillo $|F|$= 346k	0.7579	0.2849	0.1764	0.1307	0.0234	1.3734	77.55%
Sphere $|F|$= 1M	3.6806	1.0053	0.7638	0.6762	0.0354	6.1613	83.11%
Santa $|F|$= 1.2M	3.6706	0.9557	0.7818	0.9685	0.0348	6.4115	84.55%
Gargoyle $|F|$= 5.6M	32.7327	4.5813	4.7626	7.99567	0.0523	50.1246	90.76%
Lucy $|F|$= 14.4M	114.6697	23.3288	23.1114	28.1989	0.1313	189.4401	87.62%

maintain wavefront quality, hereby controlling the total number of windows. Therefore, PCH is a tradeoff between the *order* and the *parallelism* of window propagation. However, PCH is not fully parallel, since data organization runs on the CPU. For meshes with 1 million vertices, the sequential part of PCH takes 20% to 25% of the total running time, implying that the speedup of PCH has a maximal ×5 theoretical speedup by Amdahl's law [11]. Fig. 15(c) shows that the efficiency curve of PCH tends to reach saturation. In sharp contrast, AWP-CH is fully parallel and we observe perfect efficiency on all testing models.

Profiling. The memory bandwidth of NVIDIA TitanXP is 547.7 GB/s. Take Stanford Bunny with 144k faces as an example. As shown in Table 1, there are totally 21,713,401 windows generated in AWP-CH. Since one window takes 27 bytes, there are approximately 560M data and the time for reading/writing is 0.001 s. As a result, data reading and writing is not the bottleneck of our algorithm. Also note that in order to achieve perfect parallelism, we use the prefix sum to organize the data. Table 2 reports the time for each stage. It can be seen that as the size of the model increases, the ratio of the core stages in AWP to the total time increases.

Anisotropic meshes. Our current implementation does not handle skinny or degenerate triangles, because of numerical problems in the floating point computation. AWP is a type of exact algorithm, and its dependency to triangulation quality is not as high as approximate algorithms, such as the fast marching method. In the paper, we evaluated AWP on models with low and middle degrees of anisotropy $\tau \in [1, 2]$. As shown in Figs. 17 and 18, the timing of AWP on anisotropic models does not change much.

Comparison with sequential algorithms. The priority queue based algorithms, such as MMP and ICH, can propagate one window at a time, producing wavefronts of the *best* quality. Their best and worst-case window complexities are $O(n^{1.5})$ and $O(n^2)$, respectively [9]. Although priority queue is inherently sequentially, we borrow ICH's "checking with vertices" filter and MMP's window clipping to AWP-CH for pruning redundant windows.

FWP [8] organizes windows in an FIFO queue and can propagate a large number of windows in an iteration. It uses a label-correcting scheme to find a tradeoff between throughput and wavefront quality. However, due to strong data dependency within each iteration, it is non-trivial to parallelize FWP algorithms.

VTP [7] is the state-of-the-art for exact geodesics. Based on an exhaustive list of scenarios where one window can make another

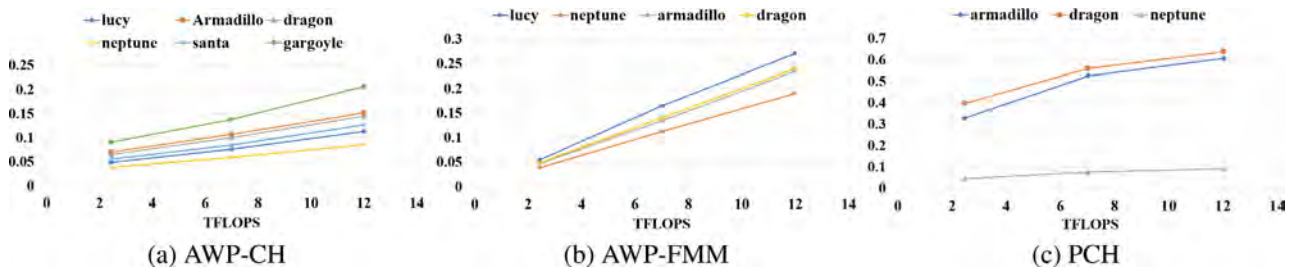


Fig. 15. Efficiency of various parallel algorithms.

Fig. 16. Results of AWP-CH. Images are rendered in high resolution, allowing zoom-in examination.

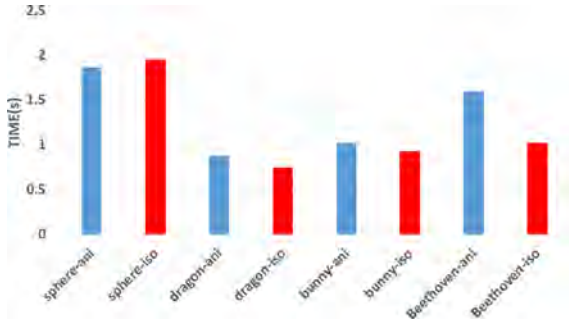


Fig. 17. Performance of AWP-CH for isotropic and anisotropic triangulation models.

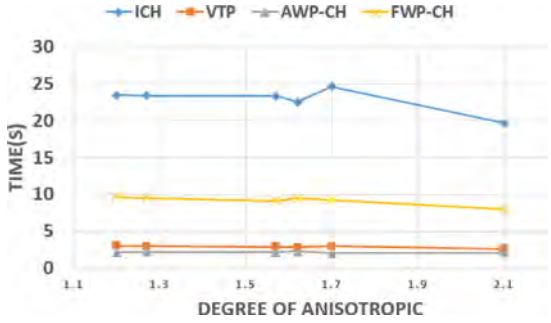


Fig. 18. Performance of AWP-CH, ICH, VTP and FWP-CH for different degrees of anisotropic.

window partially or completely redundant, VTP simultaneously propagating a collection of windows from one triangle edge to its two opposite edges. Computational results show that VTP can effectively identify redundant windows, hereby propagating the least number of windows among all exact algorithms. However, VTP adopts a priority queue to maintain wavefront, making it difficult to parallelize.

We observe that FWP, VTP and AWP-CH have the same theoretical window complexity $O(n^2)$ and empirical window complexity $O(n^p)$, $1.5 \leq p \leq 2$, depending on the anisotropic degree. Since FWP and VTP are sequential algorithms, their time complexities are the same as the window complexity. AWP-CH is fully parallel, hereby its time complexity depends on both window complexity and the number of CUDA cores.

Our implementation of AWP-CH on NVIDIA Titan Xp runs 1–2 times faster than VTP on the cutting edge CPU i7-7700k 4.2 GHz. The reason that AWP does not show a significant advantage over VTP is the difference in computational power of *individual* CPU core and GPU core. A CPU core is able to process 2.97 million windows per second, whereas a CUDA core can process only 0.072 million windows per second. Since the rate of CPUs progress has already reached saturation, with an increasing number of GPU cores we believe that AWP can outperform VTP to a large extent in the next few years. Moreover, AWP has the potential to extend to distributed systems, which is certainly an interesting and promising direction for future research.

6. Conclusion & future work

Autonomous wavefront propagation is a novel framework for parallelizing discrete geodesic algorithms on triangle meshes. Using the half-edge data structure, we define the propagation

dependency graph to characterize data dependency in computing geodesics. Then, we design an *active* strategy such that all active elements (vertices and half-edges) take the initiative to collect their input data and then propagate windows and update geodesic information in their own memory space. As a result, all the read and write operations can be carried out simultaneously. AWP works for both the exact algorithm (the CH algorithm) and approximate algorithm (the fast marching method). We prove that the AWP-CH algorithm runs in $O(n^2/\min(C, n))$ time, where n and C are the numbers of faces and cores, respectively. Evaluation on GTX Titan XP shows that AWP-CH empirically runs in n^p time, $p \in [1.25, 1.35]$, for real-world models with $n \leq 10^7$ and anisotropy measure $\tau \leq 2.0$. Thanks to its perfect efficiency and the trend of increasing the number of processors in graphics hardware, we believe that the actual performance of AWP can be further improved in the near future. The source code of AWP is available at https://github.com/openawp/awp.

For future work, we plan to extend AWP-FMM to tetrahedral meshes using the half-face based PDG. We will also parallelize other discrete geodesic algorithms, such as the MMP family, and extend AWP to distributed systems, which is another promising and challenging direction.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. This project is partially supported by the National Natural Science Foundation of China (Grant No. 61402322), Tianjin Research Program of Application Foundation and Advanced Technology, China (Grant No, 15JCQNJC00300) and Singapore Ministry of Education Grant RG26/17.

References

[1] Mitchell JS, Mount DM, Papadimitriou CH. The discrete geodesic problem. SIAM J Comput 1987;16(4):647–68.
[2] Chen J, Han Y. Shortest paths on a polyhedron. In: Proceedings of the sixth annual symposium on computational geometry. ACM; 1990, p. 360–9.
[3] Surazhsky V, Surazhsky T, Kirsanov D, Gortler SJ, Hoppe H. Fast exact and approximate geodesics on meshes. ACM Trans Graph 2005;24(3):553–60.
[4] Xin S-Q, Wang G-J. Improving Chen and Han's algorithm on the discrete geodesic problem. ACM Trans Graph 2009;28(4):104.
[5] Xin S, Wang G. Applying the improved Chen and Han's algorithm to different versions of shortest path problems on a polyhedral surface. Comput Aided Des 2010;42(10):942–51.
[6] Liu Y. Exact geodesic metric in 2-manifold triangle meshes using edge-based data structures. Comput Aided Des 2013;45(3):695–704.
[7] Qin Y, Han X, Yu H, Yu Y, Zhang J. Fast and exact discrete geodesic computation based on triangle-oriented wavefront propagation. ACM Trans Graph 2016;35(4):125:1–125:13.
[8] Xu C-X, Wang TY, Liu Y-J, Liu L, He Y. Fast wavefront propagation (FWP) for computing exact geodesic distances on meshes. IEEE Trans Vis Comput Graphics 2015;21(7):822–34.
[9] Liu Y-J, Fan D, Xu C-X, He Y. Constructing intrinsic Delaunay triangulations from the dual of geodesic Voronoi diagrams. ACM Trans Graph 2017;36(2):15:1–15:15.
[10] Ying X, Xin S-Q, He Y. Parallel Chen-Han (PCH) algorithm for discrete geodesics. ACM Trans Graph 2014;33(1):9:1–9:11.
[11] Amdahl GM. Validity of the single processor approach to achieving large scale computing Capabilities. In: Proceedings of spring joint computer conference ('67). 1967, p. 483–5.
[12] Kimmel R, Sethian J. Computing geodesic paths on manifolds. Proc Natl Acad Sci 1998;95:8431–5.
[13] Mitchell JSB. Shortest paths and networks. In: Handbook of discrete and computational geometry. 2nd ed.. 2004, p. 607–41.
[14] Peyré G, Péchaud M, Keriven R, Cohen LD. Geodesic methods in computer vision and graphics. Found Trends Comput Graph Vis 2010;5(3&4):197–397.
[15] Bose P, Maheshwari A, Shu C, Wuhrer S. A survey of geodesic paths on 3D surfaces. Comput Geom 2011;44(9):486–98.
[16] Sethian J. A fast marching level set method for monotonically advancing fronts. Proc Natl Acad Sci 1996;93:1591–5.
[17] Weber O, Devir YS, Bronstein AM, Bronstein MM, Kimmel R. Parallel algorithms for approximation of distance maps on parametric surfaces. ACM Trans Graph 2008;27(4):104:1–104:16.
[18] Breuß M, Cristiani E, Gwosdek P, Vogel O. An adaptive domain-decomposition technique for parallelization of the fast marching method. Appl Math Comput 2011;218(1):32–44.
[19] Chacon A, Vladimirsky A. A parallel two-scale method for eikonal equations. SIAM J Sci Comput 2015;37(1):A156–80.
[20] Yang J, Stern F. A highly scalable massively parallel fast marching method for the eikonal equation. J Comput Phys 2017;332:333–62.
[21] Crane K, Weischedel C, Wardetzky M. Geodesics in heat: A new approach to computing distance based on heat flow. ACM Trans Graph 2013;32(5):152.
[22] Ying X, Wang X, He Y. Saddle vertex graph (SVG): A novel solution to the discrete geodesic problem. ACM Trans Graph 2013;32(6):170:1–2.
[23] Wang X, Fang Z, Wu J, Xin S-Q, He Y. Discrete geodesic graph (DGG) for computing geodesic distances on polyhedral surfaces. Comput-Aided Geom Des 2017;52:262–84.
[24] Sethian J, Vladimirsky A. Fast methods for the Eikonal and related Hamilton-Jacobi equations on unstructured meshes. Proc Natl Acad Sci 2000;97:5699–703.
[25] Campen M, Kobbelt L. Walking on broken mesh: Defect-tolerant geodesic distances and parameterizations. Comput Graph Forum 2011;30(2):623–32.
[26] Novotni M, Klein R. Computing geodesic distances on triangular meshes. In: Proc. of WSCG '02. 2002, p. 341–7.
[27] Yatziv L, Bartesaghi A, Sapiro G. $O(N)$ Implementation of the fast marching algorithm. J Comput Phys 2006;212(2):393–9.