Computer-Aided Design 130 (2021) 102943

Contents lists available at ScienceDirect

Computer-Aided Design

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

On the Vertex-oriented Triangle Propagation (VTP) Algorithm: Parallelization and Approximation ☆

Jie Du [a], Ying He [a,*], Zheng Fang [a], Wenlong Meng [b], Shi-Qing Xin [b,*]

[a] School of Computer Science and Engineering, Nanyang Technological University, Singapore
[b] School of Computer Science and Technology, Shandong University, China

ARTICLE INFO

Article history:
Received 26 April 2020
Received in revised form 25 August 2020
Accepted 6 September 2020

Keywords:
Discrete geodesics
Parallel computing
Vertex-oriented triangle propagation (VTP)
Accuracy & speed control

ABSTRACT

Computing geodesic distances on polyhedral surfaces is a fundamental problem in digital geometry processing and computer-aided design. Most of the existing exact algorithms partition mesh edges into intervals, called windows, and propagate one window at a time. The state-of-the-art, Vertex-oriented Triangle Propagation (VTP), groups the windows on the same half-edge as a window list, and propagates window lists across triangular faces using a vertex-oriented priority queue. VTP runs much faster than the conventional algorithms thanks to its group nature. However, as a sequential algorithm, VTP is still computationally expensive for large-scale meshes. In this paper, we develop a parallel version of VTP, called Parallel-VTP or PVTP, that can propagate multiple window lists from multiple vertices simultaneously. To avoid data conflicts, PVTP proceeds with 3 steps in each iteration, which are K-window-list selection, parallel window list propagation, and vertex distance updating and window list merging. Extensive evaluation shows that PVTP improves the speed of the sequential VTP by a factor of 2.5∼3 for $T = 4$ and 4∼5 for $T = 8$ for triangular meshes with regular tessellation and over 1 million vertices, where T is the number of threads.

We observe that wavefront propagation in the exact geodesic algorithms slows down when the wavefront has a long circumference, hereby containing a large number of windows pending processing. To improve the efficiency of window propagation, we develop an approximate variant of VTP, called Approximate VTP or AVTP, which trades speed for accuracy by resetting window when the wavefront radius is a multiple of λh, where λ is a user-specified parameter and h is average edge length. We show that AVTP becomes Dijkstra's algorithm when λh is less than the minimal edge length and becomes the exact VTP when λh is greater than the longest geodesic distance on the model. AVTP has a theoretical time complexity $O(n\lambda)$, which is also confirmed by computational results. It is worth noting that the proposed parallelization and approximation techniques can be either used separately or combined together. Our source code is available at https://github.com/djie-0329/PVTP.

1. Introduction

The discrete geodesic problem is to compute geodesic distances on polyhedral surfaces. As a fundamental problem in computational geometry, it has a wide range of applications in robotics, geographic information systems and computer graphics [1]. The classic discrete geodesic algorithms are the Mitchell–Mount–Papadimitriou (MMP) algorithm [2], the Chen–Han (CH) algorithm [3] and their many variants [4–7]. Given a manifold triangle mesh and a user-specified source vertex, these algorithms partition the mesh edges into intervals, called windows, and propagate them from the source to other vertices in a continuous Dijkstra's style. During the propagation, a window can be split and generate one or more child windows. When a window lights a vertex, it updates its geodesic distance. These algorithms compute exact solutions on arbitrary manifold triangle meshes and differ in the way of window organization and the propagation scheme.

Among the above-mentioned algorithms, vertex-oriented triangle propagation (VTP) [7] is the most efficient one. Instead of propagating windows individually, VTP groups the windows of the same half-edge, and propagates the window lists across the triangulated faces in an ascending order of vertex distances, which are maintained by a priority queue. Computational results show that VTP runs 2∼15 times faster than the other exact algorithms [2–6]. Since there are $O(n^2)$ windows on a triangle mesh with n vertices, VTP has a theoretical time complexity $O(n^2)$.

☆ This paper has been recommended for acceptance by Shi-Min Hu.

* Corresponding authors.
E-mail addresses: Yhe@ntu.edu.sg (Y. He), xinshiqing@sdu.edu.cn (S.-Q. Xin).

https://doi.org/10.1016/j.cad.2020.102943
0010-4485/

For regularly tessellated triangle meshes, VTP runs in $O(n^{1.5})$ time empirically.

This paper aims at improving the performance of VTP via parallelization and approximation. The sequential VTP algorithm iteratively takes a vertex v from a priority queue (ordered by the negative distance to the source), and collects the window lists associated to the half-edges incident to v, and propagates them one by one. Our parallel VTP (or PVTP in short) supports parallel window list propagation across multiple triangles. Specifically, it collects K window lists from multiple vertices, and propagates T (specified by the user) of them simultaneously, where T is the number of threads. After all the K window lists are propagated, we adopt a delayed-writing strategy to collect the newly generated window lists and updated vertices so that we can avoid writing conflict. Extensive evaluation on a wide range of 3D models with various degrees of anisotropy shows that PVTP improves the performance of the sequential VTP algorithm by a factor of 2.5∼3 for $T = 4$ and 4∼5 for $T = 8$ on triangular meshes with regular tessellation and over 1 million vertices.

We also develop an approximate variant of VTP. We observe that for all the window propagation based exact algorithms, the propagation slows down when the wavefront has a long circumference, hereby containing a large number of pending windows. To balance speed and accuracy, we propose a window *resetting* strategy. Denote by h the average edge length. When the radius of the wavefront is a multiple of λh, we delete all the windows generated so far and take the vertices on the wavefront as new sources for the subsequent propagation. By resetting window at regular distance intervals, we can significantly reduce the total number of windows generated and improve the propagation speed. The parameter λ balances the accuracy and the speed. When λh is less than the minimal edge length, AVTP reduces to Dijkstra's algorithm. On the other hand, for a sufficiently large λ so that λh is greater than the longest geodesic distance on the mesh, AVTP becomes the exact VTP algorithm. AVTP has a theoretical time complexity $O(n\lambda)$, which is also confirmed by computational results. It is worth noting that the proposed parallelization and approximation techniques can be either used separately or combined together.

The rest of the paper is organized as follows: Section 2 reviews the related work on discrete geodesics, followed by background knowledge on windows and vertex-oriented triangle propagation in Section 3. Sections 4 and 5 document the proposed parallelization and approximation techniques, respectively. Section 6 reports experimental results. Finally, Section 7 concludes the paper and points out a few future directions.

2. Related work

Since Mitchell et al. published the seminal paper in 1987 [2], the discrete geodesic problem has been studied extensively, resulting in a large body of literature. Due to limited space, we review only the most related work here and refer the interested readers to [1,8–10] for comprehensive surveys.

2.1. Sequential algorithms

Mitchell, Mount, and Papadimitriou (MMP) proposed the first practical algorithm for computing exact[1] geodesic distances on arbitrary polyhedral surfaces [2]. The basic data structure of the MMP algorithm is called window [4], which encodes a set of shortest paths sharing the common edge sequence. It adopts a priority queue to organize windows and propagates windows in a continuous Dijkstra's style. Mitchell et al. showed that there are $O(n)$ windows on an edge, where n is the number of mesh vertices. Therefore, the MMP algorithm has an $O(n^2 \log n)$ time complexity, and the $\log n$ factor is due to the overhead of maintaining priority queue. Since it stores all windows to compute the distance from the source to arbitrary point (not necessarily vertices), the MMP algorithm has $O(n^2)$ space complexity.

Chen and Han (CH) developed an algorithm with $O(n^2)$ time complexity and $O(n)$ space complexity [3]. The CH algorithm organizes windows in a tree and propagates them in breadth-first-search order. To avoid exponential explosion, they proposed a window filter, called "one angle, one split": if two windows cover a vertex, at most one of them is allowed to generate two children.

Surazhsky et al. [4] presented an efficient implementation of the MMP algorithm and they observed that for a triangular mesh with regular tessellation, each mesh edge has an average $O(\sqrt{n})$ windows, which is much smaller than the theoretical bound $O(n)$. Therefore, the MMP algorithm runs empirically in $O(n^{1.5} \log n)$ time.

Xin and Wang observed that more than 99% of windows generated by the CH algorithm are redundant, which do not carry shortest distances [5]. To improve the performance of the CH algorithm, they proposed a simple yet effective filter that prunes most of the redundant windows by checking distances with vertices. The improved CH algorithm (or ICH) adopts a priority queue to organize windows, hence runs in $O(n^2 \log n)$ time. Experimental results show that ICH runs much faster than CH on real-world meshes due to much fewer windows generated.

Since maintaining a priority queue is expensive, Xu et al. [6] proposed fast wavefront propagation (FWP) that organizes windows in buckets and propagates the windows of the same bucket in a FIFO order. Although FWP generates more windows than MMP and ICH, its wavefront quality is not compromised too much and FWP is faster than MMP and ICH.

Recently, Qin et al. [7] proposed a totally new method, called vertex-oriented triangle propagation (VTP). VTP distinguishes itself from the other window propagation algorithms in two aspects: first, it takes vertices instead of windows as primitives in a priority queue; second, it groups the windows on a half-edge, and propagates window lists across triangles. Since it maintains a priority queue of vertices instead of windows, VTP is much faster than MMP, ICH and FWP. Moreover, VTP stores only the windows on the wavefront, hence it takes less space than the other algorithms.

2.2. Parallel algorithms

Ying et al. [11] proposed the parallel CH (PCH) algorithm, the first algorithm for parallel computing exact geodesic distances. PCH divides the sequential CH algorithm into four phases, window selection, window propagation, data organization, and event processing so that there is no data dependence in each phase and multiple windows can be propagated at the same time. Among the four phases, three can be carried out in parallel on GPUs. However, in each iteration, event processing, which has an $O(n)$ time complexity, is sequential and has to be done on CPUs. The frequent switch between the CPU and the GPU is the major bottleneck of PCH.

Recently, Ying et al. [12] developed a new parallelization framework, called autonomous wavefront propagation (AWP). AWP defines a propagation dependency graph to characterize data dependency in computing geodesics and adopts an active strategy so that windows are propagated within their own memory space. As a result, all read and write operations can be carried out simultaneously. AWP is elegant and has a perfect efficiency (i.e., doubling the threads doubles the speed). However, since

[1] By exact we mean if numerical computation is exact, the solution is exact.

it does not consider the distance priority, AWP generates much more windows than VTP. Note that the computational power of a CUDA core is not even close to that of a CPU core. As a result, AWP-CH on an NVIDIA Titan Xp GPU with 3840 CUDA cores runs only twice as fast as VTP on an Intel CPU (i7-7700K Quad Core 4.2 GHz, single thread).

VTP is the most efficient sequential algorithm so far for computing exact geodesic distances. Our goal is to further improve its speed by parallelizing VTP. We notice that PCH and AWP take advantage of the parallel structure of GPUs but sacrifice the wavefront quality too much, hereby their runtime performance does not show significant improvement over VTP.

To address this issue, we use buckets, which are a low-cost alternative to priority queue, to maintain the wavefront quality. In each iteration, we pop k_v vertices from the buckets, where k_v is determined in an adaptive manner. Then we propagate window lists that are associated to the selected vertices in parallel. Experimental results show that PVTP generates a similar amount of windows as the sequential VTP algorithm. Therefore, using multiple threads, PVTP can run much faster than VTP.

3. Preliminaries

Our parallelization and approximation techniques are built upon the seminal work of Qin et al. [7]. To make the paper self-contained, we briefly introduce the core of VTP in this section. We refer the readers to [7,10] for details.

3.1. Windows

Let $M = (V, E, F)$ be a manifold triangle mesh with vertex set V, edge set E and face set F. Denote by $s \in V$ the source point. A geodesic path inside a triangle is a straight line. When crossing over an edge, the path also corresponds to a straight line when unfolding the two adjacent faces to a common plane. Mitchell et al. [2] showed that a geodesic path γ is an alternative sequence of vertices and (possibly empty) edges such that the unfolded image of the path along any edge sequence is a line segment. Furthermore, the angle of γ passing through a vertex is greater than or equal to π. Since a geodesic path from s may pass through a few saddle vertices, the last saddle vertex in the face sequence is called a *pseudo source*.

Like MMP [2], CH [3] and their variants [4–6], VTP also simulates wavefront propagation using windows. A window is an interval on an oriented half-edge that tracks geodesic paths on the same face sequence [4]. Fig. 1(a) shows an example of window propagation.

Placing it on the horizontal axis, we define a window w as a 5-tuple $w = (x_0, x_1, x_v, y_v, \sigma)$, where x_0 and x_1 are the x-coordinates of the two endpoints, x_v and y_v are the coordinates of the pseudo-source v, σ is the geodesic distance from pseudo-source v to source s. See Fig. 1(b). Using the window w, one can compute the geodesic distance from s to any point $p \in w$ as

$$d(s, p) = \sqrt{(x_v - x_p)^2 + y_v{}^2} + \sigma. \tag{1}$$

3.2. Window pruning

There would be a lot of useless windows created during propagation, which do not contribute the shortest distance. VTP detects redundant windows by a set of distance-based filters.

Consider a window $w = [a_0, b_0]$ with pseudo-source s_0 on the triangle edge AB. Using the ICH filter [5], w is redundant if one of the following inequalities is satisfied

$$d(s, s_0) + \| \overline{s_0 b_0} \| > d(s, A) + \| \overline{A b_0} \|, \tag{2}$$

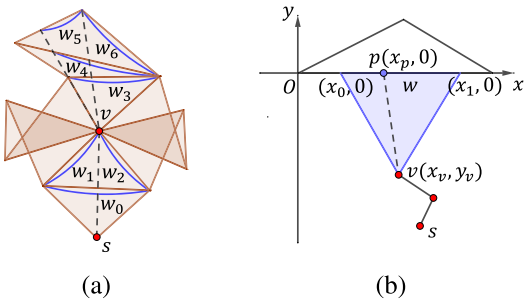

Fig. 1. Windows and pseudo-sources. (a) Starting from the source vertex s, windows are propagated across triangulated faces and new windows are generated during propagation. For example, w_1 and w_2 are the child windows of w_0. When a window lights up a saddle vertex v, v becomes a pseudo-source and the windows are further propagated from v. (b) Using window parameterization, we can compute the geodesic distance from s to any point p in w.

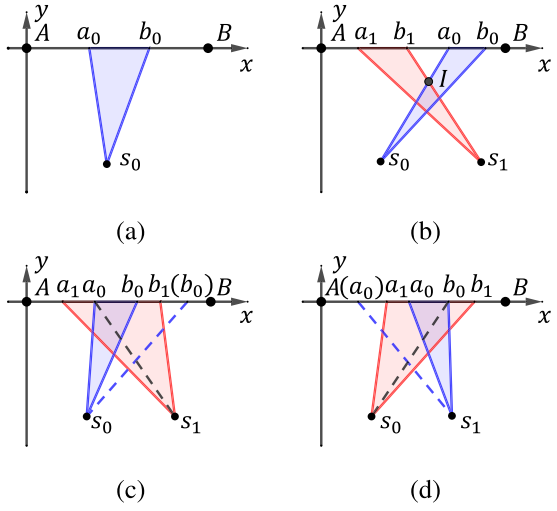

Fig. 2. Window pruning rules. (a) The ICH window filter [5]. (b), (c) and (d) show three situations of pairwise pruning between two overlapped windows [7,13].

or

$$d(s, s_0) + \| \overline{s_0 a_0} \| > d(s, B) + \| \overline{B a_0} \|, \tag{3}$$

where $\|PQ\|$ is the Euclidean distance between P and Q. See Fig. 2(a).

There are 3 pairwise window pruning strategies used in VTP [7,13]. Consider two windows $w_0 = [a_0, b_0]$ and $w_1 = [a_1, b_1]$ on the triangle edge AB. s_0 and s_1 are the pseudo-sources of windows w_0 and w_1 respectively. In Fig. 2(b), let I be the intersection of $\overline{s_0 a_0}$ and $\overline{s_1 b_1}$. If $d(s, s_0) + \| \overline{s_0 I} \| > d(s, s_1) + \| \overline{s_1 I} \|$, delete window w_0; otherwise, delete window w_1. In Fig. 2(c), if $d(s, s_0) + \| \overline{s_0 a_0} \| > d(s, s_1) + \| \overline{s_1 a_0} \|$, delete w_0; otherwise, the part $[a_1, a_0]$ of window w_1 is redundant. In Fig. 2(d), if $d(s, s_0) + \| \overline{s_0 b_0} \| > d(s, s_1) + \| \overline{s_1 b_0} \|$, delete window w_0; otherwise, the part $[b_0, b_1]$ of window w_1 is redundant.

3.3. Synchronized window propagation in a triangle

In order to thoroughly remove abundant windows at early stage, VTP [7] groups the windows on a half-edge and propagates *window lists* in a synchronized manner across triangles. See Fig. 3(a). VTP adopts three rules for window list propagation and window pruning in a triangle.

Rule 1, called “one angle two sides” rule, removes unnecessary propagation during window list splitting. Consider a window

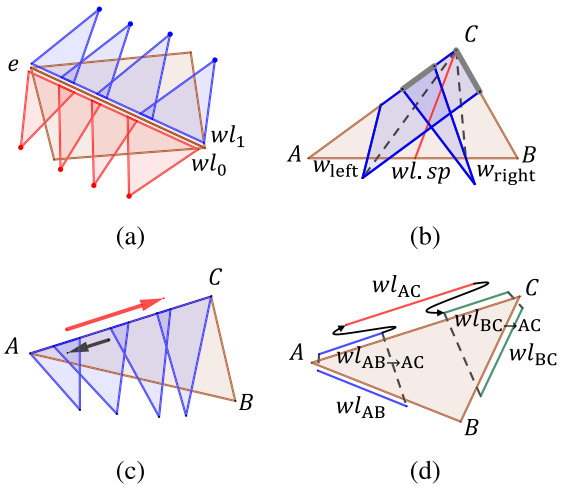

Fig. 3. Window propagation rules in a triangle [7]. (a) Each side of the edge e contains a window list. (b) The propagation of window w_{left} to edge BC is redundant when $w_{\text{left}}.sp < wl.sp$, and the propagation of window w_{right} to edge AC is redundant when $w_{\text{right}}.sp > wl.sp$. The redundant windows are drawn in gray. (c) Pairwise windows pruning during window list propagation. (d) Propagating window lists wl_{AB} and wl_{BC} produce $wl_{\text{AB}\to\text{AC}}$ and $wl_{\text{BC}\to\text{AC}}$ respectively. Update window list wl_{AC} by merging $wl_{\text{AB}\to\text{AC}}$ and $wl_{\text{BC}\to\text{AC}}$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

list $wl_{\text{AB}} = \{w_i | i = 0, 1, 2....\}$ on half-edge AB, which will propagate across ΔABC. For each window $w_i \in wl$, VTP computes a *separating point* sp by minimizing

$$\arg\min_{sp \in w_i} \{\sigma_{s_i} + \|s_i - sp\| + \|sp - C\|\}, \tag{4}$$

where s_i is the pseudo-source of w_i, σ_{s_i} is the geodesic distance between source s and pseudo-source s_i. Denote by w_s the window which supports shortest distance to vertex C. $wl.sp$ is the separating point of w_s. Fig. 3(b) shows an example of the "one angle two sides" rule. For each window $w_i \in wl$ and $w_i \neq w_s$, the propagation of w_i to edge BC is redundant if $w_i.sp < wl.sp$, and the propagation of w_i to edge AC is redundant if $w_i.sp > wl.sp$ [7].

Rule 2 performs the pairwise window pruning for window list wl. It traverses all windows in the outer loop (red arrow) and checks each window against its preceding windows in the inner loop (black arrow). See Fig. 3(c). During window list propagation, VTP utilizes the ICH filter and Rule 2 to prune or remove redundant windows.

The newly generated window lists associated to a half-edge are merged by Rule 3. As Fig. 3(d) shows, for the half-edge (A, C), $wl_{\text{BC}\to\text{AC}}$ is inserted in front of wl_{AC} and $wl_{\text{AB}\to\text{AC}}$ is appended after it.

3.4. Vertex-oriented triangle propagation

One of the key features of VTP is that it takes *vertices* as primitives and propagates geodesic path across triangles around the vertex popped from a priority queue iteratively. Algorithm 1 shows the pseudo code of VTP. Given a source vertex s, VTP initializes the distances $d(s) = 0$ and $d(v) = \infty$ for $v \neq s$. It creates a window for each half-edge opposite to s and stores it in the half-edge's window list. It also pushes the adjacent vertices into a priority queue $\mathcal{Q}$ according to their distances to s.

Denote by wf the wavefront and R the interior region it borders. At each iteration, the VTP algorithm pops up a vertex v_i with shortest geodesic distance value from $\mathcal{Q}$ and extends its one-ring region as follows. First, the wavefront wf and its enclosed area R are extended to the one-ring region of v_i. It pushes the window lists adjacent or opposite to v_i into the FIFO queue $\mathcal{W}$ if they are inside the swept region R. While the window list queue $\mathcal{W}$ is not empty, VTP pops a window list wl_i from $\mathcal{W}$. Then it prunes the redundant windows of wl_i using the pruning rules in Section 3.2 and Rule 2. Next it propagates the windows of wl_i in a synchronized manner and merge the newly-generated windows into the corresponding window lists (Rule 1 and Rule 3 in Section 3.3). If the newly-generated windows are inside the area R, push them into $\mathcal{W}$. Finally, if there exist vertices whose geodesic distances are updated during one-ring region growing of v_i, push the updated vertices into the priority queue $\mathcal{Q}$. Repeat these steps until the priority queue $\mathcal{Q}$ is empty.

In summary, the VTP algorithm iteratively extends the swept area R in a Dijkstra-like manner with the aid of vertices in priority

Algorithm 1: Vertex-oriented Triangle Propagation Algorithm [7]

Input: A triangle mesh $M = (V, E, F)$, source vertex id $s \in \{1, \cdots, |V|\}$.
Output: The geodesic distance d_i for each vertex v_i.

```
Initialize distances d_S ← 0 and d_i ← ∞, ∀i ≠ s.
Create a priority queue Q for vertices.
Create a FIFO queue W for window lists.
Create a window for each edge opposite to s and store them into the corresponding window lists.
Compute the geodesic distance for each vertex adjacent to s and push the vertices into Q.
Update the wavefront wf and its swept area R using the one-ring triangles of s.
while Q ≠ ∅ do
    Pop a vertex v_i from Q.
    if v_i is a saddle vertex then
        Create a window for each edge opposite to v_i and store the windows into the corresponding window lists.
        Update the geodesic distances of vertices adjacent to v_i and push the updated vertices into Q.
    Update the wavefront wf and its enclosed area R using the one-ring triangles of v_i.
    Push the window lists into W if they are inside the swept area R and adjacent or opposite to v_i.
    while W ≠ ∅ do
        Pop a window list wl_i from W.
        Prune the redundant windows of wl_i (Rule 2).
        Propagate wl_i (Rule 1).
        Push updated vertex into Q.
        Merge newly-generated window lists (Rule 3).
        Push the newly-generated window lists into W if they are inside the swept area R.
```

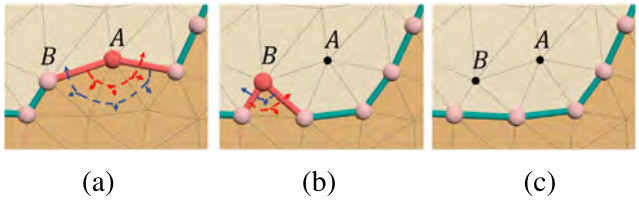

Fig. 4. Vertex-oriented triangle propagation. From (a) to (b): window propagation across triangles around vertex A; From (b) to (c): window propagation across triangles around vertex B. The swept region is light-colored and the unswept region is dark-colored. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

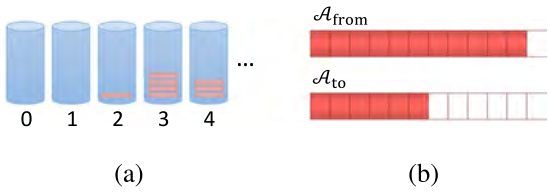

Fig. 5. Data structures of PVTP. (a) Buckets $\mathcal{B}$ store the soon-to-be propagated vertices. (b) Dual arrays $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ control concurrent window list propagation alternately.

queue. Fig. 4 shows examples of VTP iterations. Suppose A is the nearest vertex to source s at the moment. The triangles incident to A are propagated in the current iteration. Let B be next vertex popped up from the priority queue $\mathcal{Q}$. The triangles adjacent to B will be propagated in the next iteration. After window list propagation, the updated vertices on the wavefront are pushed into $\mathcal{Q}$.

4. Parallel VTP

4.1. Overview

The original VTP propagates geodesic distances across triangles around the vertex with the highest priority at a time, organizes windows located on the same half-edge into a window list, and propagates one window list at a time. PVTP aims at processing multiple vertices in each iteration and propagating multiple window lists simultaneously. There are main technical challenges including (a) dealing with data conflicts during concurrent propagation across triangles; and (b) replacing the priority queue by a low-cost alternative without compromising the wavefront quality too much. We use a delayed-writing strategy to avoid data conflicts and borrow the bucket data structure, which is used in the sequential FWP algorithm [6], to address the second challenge.

We list the core data structures to facilitate the description of our algorithm:

- A **window list** wl, as achieved in the original VTP, to organize the windows on a half-edge. See Fig. 3(a).
- A list of **buckets** $\mathcal{B}$ to maintain the priority of soon-to-be propagated vertices. See Fig. 5(a).
- A pair of **arrays** $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ to facilitate parallel propagation of window lists. See Fig. 5(b).
- A **buffer** *buf* to temporarily store the generated window lists and updated vertices. This delayed writing strategy can avoid data conflicts.

The pseudo code of PVTP is shown in Algorithm 2. Given a source vertex s, we first create a window for each half-edge opposite to s and update the geodesic distances of s's neighboring vertices. We then insert the newly generated windows into the corresponding window lists and push the updated vertices into buckets $\mathcal{B}$. Denote by wf the wavefront and R the area it encloses. At each iteration, PVTP selects top k_v vertices from buckets $\mathcal{B}$ and extends the wavefront by a range of one-ring: (a) selecting K window lists around the k_v vertices and storing them in $\mathcal{A}_{\text{from}}$; (b) propagating the selected K window lists in a parallel manner, and saving their child window lists and the swept vertices[2] into buffer *buf*; (c) handling the data in *buf* which includes merging window lists in *buf* by Rule 3, updating the distances of the recently-swept vertices, and pushing the merged window lists into A_{to} and the updated vertices into buckets $\mathcal{B}$; and (d) swapping $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$. If array $\mathcal{A}_{\text{from}}$ is not empty, go to (b); otherwise go to (a). The algorithm terminates when the buckets $\mathcal{B}$ become empty — at this moment, the geodesic distances to all destination vertices have been computed. Fig. 6 visualizes a typical PVTP iteration on the Bunny model with 72K vertices.

[2] The swept vertices include both the vertices inside the wavefront and the ones on the wavefront.

Algorithm 2: Parallel VTP

Input: A triangle mesh $M = (V, E, F)$, source vertex $s \in V$, the number of threads T
Output: Geodesic distance $d(v_i)$ for each vertex v_i

```
d(s) ← 0 and d(v_i) ← ∞, ∀i ≠ s.
Create buckets B for vertices.
Create dual arrays A_from and A_to for window lists.
Create windows for each edge opposite to s and store them into the corresponding window lists.
Compute the geodesic distance for each vertex adjacent to s and push the vertices into B.
Update the wavefront wf and its enclosed area R using the one-ring triangles of s.
while B ≠ ∅ do
    Select k_v vertices from B.
    Update the wavefront wf and its enclosed area R using the one-ring triangles of the k_v selected vertices.
    Push K window lists into A_from if they are inside R and are adjacent or opposite to the k_v selected vertices.
    while A_from ≠ ∅ do
        Parallel propagate window lists in T threads and save the generated window lists and updated vertices in buf.
        Merge newly-generated window lists and update vertices' distance.
        Push the newly-generated window lists into A_to if they are inside the swept area R.
        Push updated vertices into B.
        Swap A_from and A_to.
```

4.2. Buckets for maintaining vertex priorities

In contrast to the original VTP algorithm, PVTP handles multiple vertices at a time. Since the quality of the wavefront highly depends on the order of vertex processing, we use a bucket data structure [6] to maintain the priority in which the vertices are processed. Let L be the maximum geodesic distance, which shall be estimated later. We create $\lceil L\tau/h \rceil$ buckets, where h is the average edge length and τ (≥ 1) is the anisotropy degree [14], which measures how far the input triangulation is from isotropic. A complete isotropic triangular mesh has $\tau = 1$. The ith bucket records the distance ranging from ih/τ to $(i+1)h/\tau$. When the distance of a vertex v_i is updated during the wavefront propagation, we push the vertex's id into the $\lfloor d_i\tau/h \rfloor$-th bucket, where d_i is the updated distance of v_i.

If the wavefront is close to a geodesic isocontour, it has good quality so that many vertices can be processed simultaneously without producing too many redundant windows. On the other hand, if the vertices on the wavefront have distances with large variations, only a few vertices (i.e., the ones with shortest distances to the source) can be processed. Therefore, the throughput of parallel window propagation depends closely on the quality of wavefronts.

Let $P^{(i)}_{\text{st}}$ and $P^{(i)}_{\text{nd}}$ be the distances of the first and last buckets where vertices are popped from in the current iteration. Denote by $c^{(i)}_{\text{large}}$ (resp. $c^{(i)}_{\text{small}}$) the number of updated vertices whose distances are greater (resp. less) than $P^{(i)}_{\text{nd}}$. Denote by $k^{(i)}_{\text{ad}}$ the number

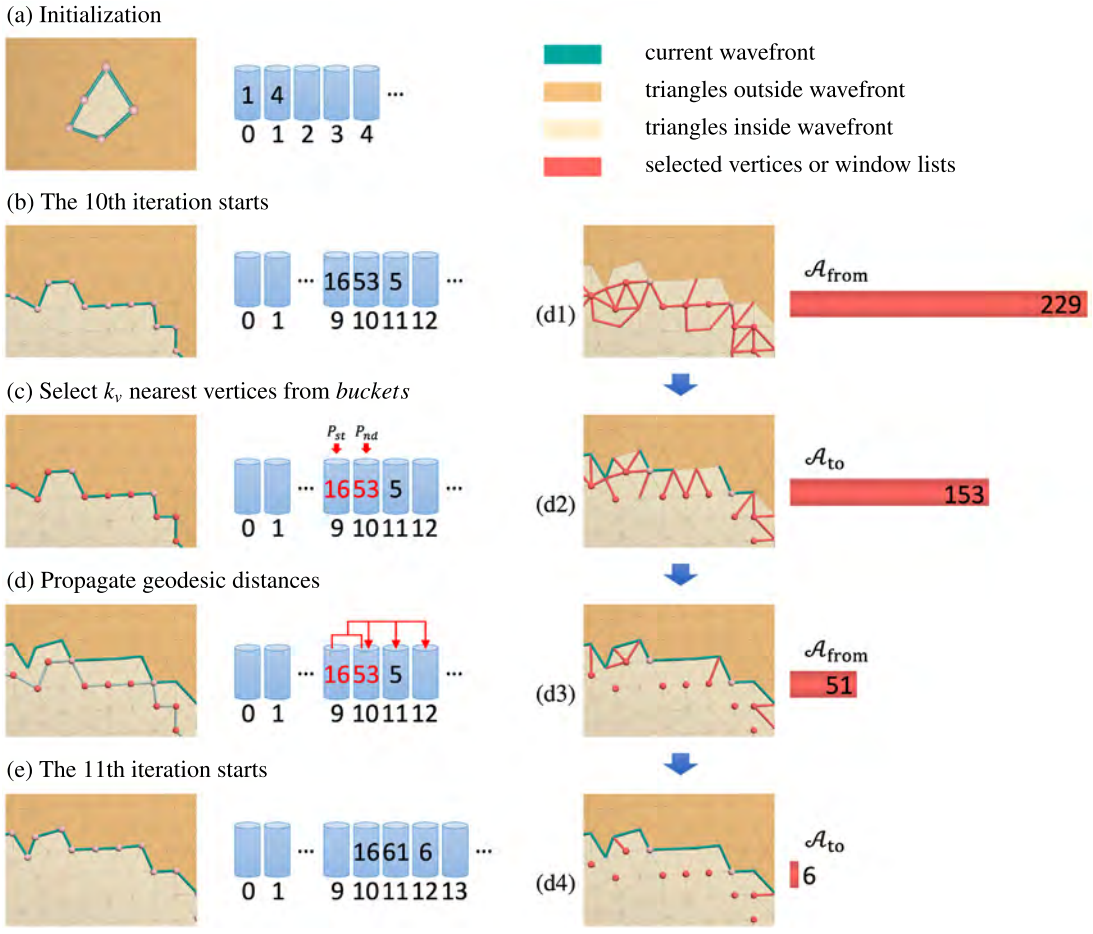

Fig. 6. A typical iteration of PVTP. The group of blue cylinders denotes the buckets $\mathcal{B}$ and the red horizontal bars denote the dual arrays $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ to store the window lists alternately. (a) Create a window for each edge opposite to the source s and store the windows into their corresponding window lists. Then the updated vertices are pushed into the buckets $\mathcal{B}$. Green segments describe the current wavefront. (b) At the beginning of 10^{th} iteration, all to-be-propagated vertices are stored in the buckets $\mathcal{B}$ according to their geodesic distances. (c) Select k_v-nearest vertices from buckets $\mathcal{B}$ so that K window lists can be handled at the same time; See Section 4.3 for the choices of k_v and K. (d) Propagating and merging window lists: (d1) 229 window lists are selected and stored into one of the dual arrays $\mathcal{A}_{\text{from}}$; (d2) After parallel propagation, there are 153 updated window lists inside the wavefront and they are pushed into the other array $\mathcal{A}_{\text{to}}$ for the next parallel propagation; (d3, d4) Operate the data alternatively between $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ until both the dual arrays are empty. At the moment, the new wavefront is reshaped as the green segments. During performing (d1–d4), the vertices with updated geodesic distances would be pushed into the buckets $\mathcal{B}$. (e) The 11th iteration. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

of vertices popped from the buckets $\mathcal{B}$ in the ith iteration. Then in the $(i+1)$-th iteration, $k_{\text{ad}}^{(i+1)}$ is adaptively updated as

$$k_{\text{ad}}^{(i+1)} = k_{\text{ad}}^{(i)} + c_{\text{large}}^{(i)} - c_{\text{small}}^{(i)}. \tag{5}$$

Remark. Priority queue can guarantee the event with the highest priority is handled first, so it can minimize the total number of generated windows. However, inserting or deleting an element from a priority queue takes $O(\log n)$ time, which is expensive. In contrast, maintaining a bucket data structure takes only $O(1)$ time. Experimental results show that PVTP produces only slightly more windows than the original VTP algorithm, which justifies the effectiveness of the bucket data structure and the adaptive popping strategy.

4.3. K-Window-list selection

Window list selection is a trivial operation in the original VTP algorithm since it considers window lists sequentially. However, PVTP is different since it propagates K window lists simultaneously. We observe that choosing a proper number of window lists for propagation is critical to the efficiency of parallelization. Our goal is to select as many window lists as possible to get better parallelization efficiency, and meanwhile not downgrade the wavefront quality too much. We adopt the following strategy to determine the value of K.

Given the $k_{ad}^{(i)}$ vertices computed above, we count the number of window lists around them, which is denoted by $K_{\text{ad}}^{(i)}$. Then, we determine the number of window lists to be propagated $K^{(i)}$ by

$$K^{(i)} = \min(K_{\text{ad}}^{(i)}, 100T), \tag{6}$$

where T is the number of threads. We set an empirical threshold $100T$ to balance the wavefront quality and throughput of parallel propagation. Finally, we compute $k_v^{(i)}$ the number of actual popped vertices from buckets $\mathcal{B}$ which corresponds to the $K^{(i)}$ window lists.

The selected $K^{(i)}$ window lists are stored in array $\mathcal{A}_{\text{from}}$. The red segments in Fig. 6(d1) denote the selected window lists. It is worth noting that if we encounter a saddle vertex that is popped from the buckets, we create windows on the one-ring opposite edges and add them into the corresponding window lists, which is due to the fact that a saddle vertex may serve as a pseudo-source vertex.

4.4. Parallel window list propagation

After selecting K window lists, PVTP propagates window lists in parallel with the aid of dual arrays $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ as shown in Fig. 6(d1–d4) and Algorithm 2 (Line 12–16).

For each window list wl in $\mathcal{A}_{\text{from}}$, we inherit the window pruning strategies and propagation rules of the original VTP algorithm that are summarized in Section 3. During the propagation, if there is a vertex v gets a smaller distance, we push it to the buckets. Let wl be a window list. If the newly-generated window lists wl_{left} and wl_{right}, the next generation of wl, are located inside the newly wavefront as the red segments shown in Fig. 6(d2), we store wl_{left} and wl_{right} into $\mathcal{A}_{\text{to}}$. Considering that there may be possible data conflicts during the parallel window list propagation (different window lists may generate new window lists on the same half-edge; the distance of a vertex may be updated from different window lists), we utilize a buffer *buf* to store the conflicts temporarily and process the events in delay. After pushing the merged window lists into $\mathcal{A}_{\text{to}}$, we swap $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$. In this way, we concurrently propagate window lists alternately with the help of $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ until both of them become empty.

4.5. Correctness

A window-propagation-based algorithm is correct if it does not discard any useful windows. The correctness is independent of the window processing order. The central idea of all algorithms in the family is to effectively organize windows so that they can be propagated in a roughly near-to-far order and prune as many redundant windows as possible. The algorithms differ in the window organization and propagation strategies, and pruning rules. Since PVTP adopts the same pruning rules as VTP, it keeps all the useful windows and computes exact geodesic distances when the algorithm terminates.

4.6. Complexity analysis

Given a manifold triangle mesh with n vertices, there are $O(n)$ half-edges and faces. Since a half-edge contains at most $O(n)$ windows, there are $O(n^2)$ windows generated [2]. As a result, parallel window list propagation takes empirical $O(n^2/T)$ time complexity, where T is the number of threads. Inserting (resp. deleting) an element to (resp. from) buckets takes $O(1)$ time. Therefore, both K-window list selection and pushing updated vertices into buckets have $O(n)$ time complexity. Window list merging in PVTP is as order-free as the original VTP (taking $O(1)$ time for each merging operation), so it also runs in $O(n)$ time. In summary, PVTP has a time complexity $O(n^2/T + n) = O(n^2)$.

We adopt "buckets" structure in our parallel implementation, which could maintain a smooth wavefront as sequential VTP algorithm. Thus, PVTP has space complexity $O(n^2)$, which is the same as VTP.

4.7. Implementation details

Adaptive choice of k_{ad}. The number of vertices popped from the buckets $k_{\text{ad}}^{(i)}$ is a variable. Obviously, if K_{ad}, the number of window lists selected for parallel propagation, is less than T, i.e. the number of threads, the parallel processing cannot bring performance gains. Herein, we require the number of selected vertices popped from the buckets, k_{ad}, must be larger than T. In this case, the number of selected window lists for parallel processing must be also larger than T.

Updated vertices inside wavefront. For the original VTP algorithm, if an updated vertex v is a saddle vertex located in the area enclosed by the wavefront, new windows are created on each one-ring edge opposite to v and these corresponding window lists have to be propagated until they arrive at the wavefront. In fact, for an anisotropic model with a large amount of obtuse triangles, it is quite often that the distance of a vertex v is updated multiple times. Note that the buckets cannot guarantee that the vertices are processed in a strictly ascending order of the distances. Therefore, in our implementation, the buckets contain two types of vertices: vertices on the wavefront and saddle vertices inside the swept region.

Algorithm 3: Approximate VTP

Input: A triangle mesh $M = (V, E, F)$, source vertex $s \in V$, the speed and accuracy control parameter λ.
Output: The geodesic distance d_i for each vertex v_i.

```
1  Initialize as Algorithm 1: Line 1-6;
2  Compute the average edge length h.
3  R_num ← 0.
4  while Q is not empty do
5      v_i = Q.pop().
6      if d_i > (R_num + 1)λh then
           /* N: # of vertices on the current
              wavefront                          */
7          for j ← 0 to N do
8              if d_j > (R_num + 1)λh + λ_g h then
9                  Preserve the windows on the window lists
                   adjacent to v_j.
10             else
11                 Delete the windows on the window lists
                   adjacent to v_j.
12         Mark all vertices on the wavefront as
           pseudo-sources and push them into Q.
13         R_num + +.
14     Propagate the wavefront as in Algorithm 1: Line 8-20.
```

5. Approximate VTP

5.1. Motivation

Accuracy and speed are a pair of indicators for evaluating a geodesic algorithm. In fact, it is necessary to make a balance between accuracy and speed in practice. In this section, we aim at developing a fast and accuracy-controllable approximate discrete geodesic algorithm that does not require pre-computation.

Imagine that C is an isoline of the geodesic distance field. Suppose that the distances inside the area enclosed by C has been fixed. Now we take sufficiently many samples on C and continue propagating the distances. It is easy to show that the final distance field is almost identical to the real distance field. The observation motivates us to replace windows by vertices on the wavefront when the propagation goes across an appropriate distance gap.

Let L be the farthest geodesic distance from the source to all destinations, and D be the prescribed distance gap. Then the number of steps is $\lfloor \frac{L}{D} \rfloor$. Once the minimum distance reported by wavefront climbs to a new step, we clear all the windows and use the vertices on the wavefront as the new sources to continue the distance propagation. In this way, we can trade accuracy for speed.

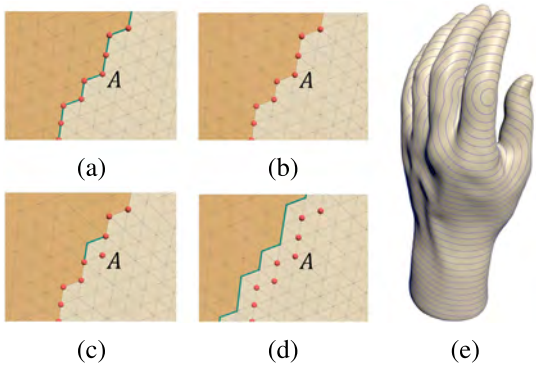

Fig. 7. Illustration of a typical iteration of AVTP on the Hand model with $\lambda = 32$. (a) Let A be the nearest vertex to the source s. The edges containing active windows are drawn in green. (b) When $d(s, A) > \lambda h$, all windows on the wavefront are deleted. (c) Take the vertices on the current wavefront as new sources, from which windows are propagated. (d) A new wavefront is formed. (e) The isolines define the distance steps where we reset windows. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5.2. Key ideas

The approximate geodesic algorithm is implemented by adding one step in the VTP algorithm as shown in Algorithm 3 and Fig. 7. If the geodesic distance of the nearest vertex A on the current wavefront is larger than $(R_{num} + 1)\lambda h$, we delete all windows on the wavefront and take the vertices on wavefront as pseudo-sources, where λ denotes the prescribed distance gap, R_{num} denotes the number of steps that were passed, and h denotes the average edge length for input model. In order to avoid large errors for anisotropic or non-uniform triangle meshes, we set λ_g as a threshold: If the geodesic distance of a vertex on the current wavefront is larger than $(R_{num} + 1)\lambda h + \lambda_g h$, the windows on the edges adjacent to it would be preserved. It is worth noting that AVTP can also be parallelized by the PVTP framework.

5.3. Complexity analysis

Fig. 8 gives comparison statistics on the number of windows on the Hand model. By setting $\lambda = 32$, the relative mean error of AVTP is 0.00497% while the number of windows (red bar) is significantly reduced, thus greatly improving the performance. It is worth noting that AVTP reduces to Dijkstra's algorithm when λh is less than the minimum edge length, while becoming the exact VTP algorithm when λh is greater than the maximum geodesic distance L. The sequential AVTP algorithm has an $O(n\lambda)$ time complexity because there are at most $O(\lambda)$ windows on each edge during window propagation [4].

6. Experimental results and discussions

6.1. Parallel algorithm

6.1.1. Efficiency

To test the efficiency of PVTP, we compare it with the sequential VTP on both isotropic and anisotropic meshes. Table 1 and Fig. 9 report the detailed statistics. We observe that for regularly tessellated models, PVTP is 2∼3 times faster than the sequential VTP on $T = 4$ threads and 3∼5 times faster when $T = 8$. Moreover, the memory consumption and window complexity of PVTP is similar to VTP. As Table 1 shows, the total propagated windows of PVTP is only 2.54% more than VTP under 4 threads and 3.16% more under 8 threads. The memory cost of PVTP is

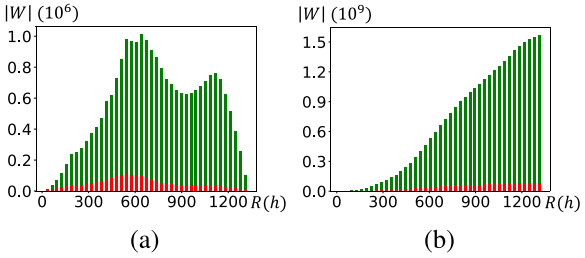

Fig. 8. Window distribution of AVTP with $\lambda = 32$ (red) and the exact VTP (green) on an isotropic hand model with 2.3 million vertices. The horizontal axis is the wavefront radius, expressed as a multiple of the average edge length h. The vertical axes in (a) and (b) are the number of windows on the wavefront and the accumulated number of windows, respectively. AVTP generates significantly fewer windows than VTP. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

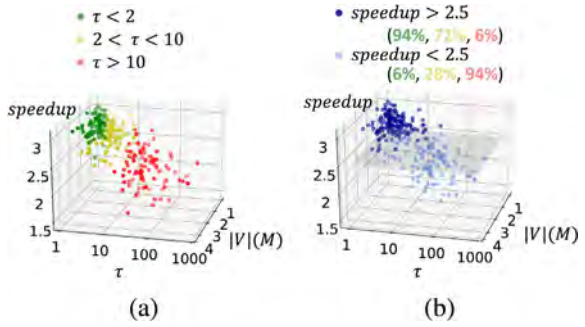

Fig. 9. Performance statistics of PVTP (4 threads) on 300 models with varying anisotropy degree τ, which are randomly selected from the Thingi10k dataset. We plot the speedup factor in 3D diagrams. (a) Meshes with low, middle and high degree of anisotropy are drawn in green, yellow and red, respectively. (b) The horizontal plane denotes the speedup factor 2.5. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

1.97% higher than VTP under 4 threads and 2.74% higher under 8 threads.

It is no wonder that PVTP does not have a big advantage over the sequential VTP if the model resolution is small since there is a large timing cost spent on the sequential computation part; see the Bunny model with 72K vertices in Table 1. Therefore, we subdivide the anisotropic models into at least 1M vertices and observe the improvement of the performance. We randomly choose 300 models from the Thingi10k dataset,[3] including (a) 100 models whose anisotropic degree τ is less than 2, (b) 100 models whose τ is between 2 and 10, and (c) 100 models whose τ is larger than 10.

As an exact algorithm, PVTP produces the same numerical results on all models as the sequential VTP with window interval threshold 10^{-6} [7].[4] The statistics in Fig. 9 show that our parallel framework has an advantage over the original VTP for anisotropic models with $\tau < 10$. In detail, our parallel-VTP is at least 1.5∼3 times faster than VTP under 4 threads on the 300 models. If we consider the models with $\tau < 2$, the speedup amounts to 2.5∼3 times on 94% of the models. If we allow the anisotropic factor τ to range between 2 and 10, a 2.5∼3 times speedup can also be obtained on 72% of the models. It is worth noting that the speedup effect decreases with the increasing of anisotropic degree, which is due to the fact that a high-quality wavefront (close to a isocontour) is difficult to achieve unless we rigorously follow the priorities of vertices.

[3] https://ten-thousand-models.appspot.com/

[4] Windows whose lengths are less than the threshold are discarded.

Table 1
Performance comparison of VTP and the proposed parallel and approximate variants (PVTP, AVTP, PAVTP). τ is the anisotropy degree of the input meshes and error is the relative mean error.

Model	Performance	Algorithms						
		VTP	PVTP		$\lambda = 8$		$\lambda = 32$	
			$T = 4$	$T = 8$	AVTP	PAVTP ($T = 4$)	AVTP	PAVTP ($T = 4$)
Armadillo	Speedup	1	2.443	3.375	2.529	4.276	1.768	3.459
$\|V\|$: 692K	Peak mem.(MB)	6.549	6.736	6.802	1.594	1.654	4.097	4.200
τ: 1.347	Total #windows	66,223,041	68,769,693	69,279,435	11,128,586	11,639,232	28,486,185	29,432,522
	Error (%)	0	0	0	0.080564	0.080564	0.014819	0.014819
Bunny	Speedup	1	2.189	2.526	2.400	3.243	1.590	2.748
$\|V\|$: 72K	Peak mem.(MB)	1.519	1.594	1.643	0.466	0.492	1.195	1.238
τ: 1.258	Total #windows	4,878,831	5,085,895	5,185,930	1,004,357	1,058,105	2,445,322	2,540,435
	Error (%)	0	0	0	0.086441	0.086441	0.013227	0.013227
Gargoyle	Speedup	1	2.672	3.489	3.376	5.900	2.294	4.701
$\|V\|$: 3,200K	Peak mem.(MB)	46.648	46.979	47.080	6.131	6.268	18.242	18.463
τ: 1.407	Total #windows	569,784,945	577,395,049	578,867,166	60,941,660	63,311,663	154,118,728	158,461,594
	Error (%)	0	0	0	0.116999	0.116999	0.012781	0.012781
Golfball	Speedup	1	2.912	4.017	16.130	29.299	10.343	21.928
$\|V\|$: 7,864K	Peak mem.(MB)	203.751	206.407	206.645	5.225	5.388	16.778	17.250
τ: 1.731	Total #windows	8,749,978,577	8,907,732,188	8,922,046,848	169,495,647	178,567,392	451,939,716	470,661,404
	Error (%)	0	0	0	0.045374	0.045374	0.003834	0.003834
Hand	Speedup	1	3.073	5.176	17.559	27.973	11.363	22.960
$\|V\|$: 2,298K	Peak mem.(MB)	75.562	75.644	75.776	2.847	2.892	9.455	9.500
τ: 1.012	Total #windows	1,481,158,276	1,484,109,571	1,485,675,117	31,251,924	31,882,694	85,861,269	86,690,282
	Error (%)	0	0	0	0.059886	0.059886	0.00497	0.00497
Lucy	Speedup	1	2.396	3.318	2.697	4.281	1.855	3.444
$\|V\|$: 1,052K	Peak mem.(MB)	11.733	12.133	12.206	2.702	2.810	7.315	7.552
τ: 1.604	Total #windows	117,416,392	123,902,626	124,886,259	18,976,893	20,123,287	45,660,689	47,860,688
	Error (%)	0	0	0	0.087376	0.087376	0.01201	0.01201
Rockerarm	Speedup	1	2.943	4.267	9.599	17.212	6.207	13.252
$\|V\|$: 2,571K	Peak mem.(MB)	70.588	70.917	70.949	2.822	2.909	8.657	8.849
τ: 1.515	Total #windows	1,447,435,014	1,459,454,219	1,461,652,495	57,004,153	59,758,894	147,407,347	153,163,408
	Error (%)	0	0	0	0.139333	0.139333	0.011661	0.011661

6.1.2. Scalability

We use the Fertility and the Lucy models to observe whether the performance gain exists independent of specific shapes. As Fig. 11 shows, the speedup factor depends on the number of vertices and is insensitive to different geometric variations.

We also compare the total propagated windows and memory consumption on commonly used models in Table 2. The total propagated windows of PCH (resp. AWP-CH) is 4.46 (resp. 3.42) times more than PVTP with $T = 8$ on average. The memory cost of PCH (resp. AWP-CH) is 17.65 (resp. 16.52) times higher than PVTP with $T = 8$ on average.

6.1.3. Profiling

In order to observe the parallelization effect, we divide the total running time into two parts, one spent in sequential operations and the other spent in parallelization operations. Based on the statistics given by Fig. 12, we can see that the timing cost of sequential steps (blue part of histogram) under one thread is lower than 5%. The majority of the operations are executed in a parallel style, which accounts for the fact that the performance improvement of PVTP against VTP correlates with the number of threads in CPU.

6.2. Approximate algorithm

6.2.1. Performance and accuracy

We test our AVTP on regularly tessellated models as shown in Table 1 and anisotropic models from Thingi10k. Recall that we use a single parameter λ to make the balance between performance and accuracy. Fig. 10 plots how λ affects the trade off between performance and accuracy. Here the three test models are with different anisotropic degrees. Based on the experimental results, we find (1) the time complexity is roughly $O(n\lambda)$, and (2) the accuracy could be improved to 2^a times by doubling λ, where $a \in [1, 2]$. The biggest advantage of AVTP/PAVTP is that they allow users to balance performance and accuracy using λ.

6.2.2. Comparison with approximate-ICH

We compare AVTP with AICH [15], which is an approximate variant of the ICH algorithm [5]. Both AVTP and AICH do not require pre-computation. Using a user-specified parameter $\lambda_a \in [0, 1]$, AICH adopted an over-filtering strategy to control the accuracy. Fig. 13 shows the distribution of relative mean error for models with varying degrees of anisotropy. We compute $x_{\bar{\varepsilon}}$ by averaging the mean errors of models with $\tau < 10$ and define $\delta \triangleq x_{\bar{\varepsilon}} \times 0.8$. We observe that 67.7% (resp. 69% and 71%) of the models have an error in the range $[x_{\bar{\varepsilon}} - \delta, x_{\bar{\varepsilon}} + \delta]$ when we set $\lambda = 8$ (resp. 32 and 128) in AVTP. For the comparison purpose, we take $\lambda_A = 0.1$ for the AICH algorithm. With the parameter setting, AICH is not so accurate as our AVTP but more timing consuming. In detail, the average error of AICH is around $x_{\bar{\varepsilon}}$ for 41.3% models and even worse for the remaining models.

Table 2
Memory consumption of PVTP (with 4 and 8 threads), PCH and AWP-CH. Our method generates significantly fewer windows and consumes much less memory than the other parallel algorithms.

Model	τ	Performance	Algorithms			
			PCH	AWP-CH	PVTP	
					$T = 4$	$T = 8$
Armadillo	1.347	Peak mem.(MB)	134.783	167.343	6.736	6.802
$\|V\|$: 692K		Total #windows	243,998,069	441,738,351	68,769,693	69,279,435
Bunny	1.258	Peak mem.(MB)	50.688	25.810	1.594	1.643
$\|V\|$: 72K		Total #windows	33,892,153	17,832,190	5,085,895	5,185,930
Fertility	1.267	Peak mem.(MB)	14.106	11.344	1.369	1.405
$\|V\|$: 30K		Total #windows	7,446,597	4,224,471	2,025,663	2,050,153
Gargoyle	1.025	Peak mem.(MB)	68.822	66.331	2.882	2.922
$\|V\|$: 247K		Total #windows	110,457,175	67,589,361	13,287,453	13,385,292
Golfball	1.731	Peak mem.(MB)	35.945	49.625	2.601	2.676
$\|V\|$: 123K		Total #windows	46,999,069	52,180,166	14,016,515	14,270,891
Hand	1.017	Peak mem.(MB)	58.695	40.434	5.516	5.561
$\|V\|$: 125K		Total #windows	66,360,823	26,118,914	16,215,356	16,300,920
Lucy	1.604	Peak mem.(MB)	186.742	228.298	12.133	12.206
$\|V\|$: 1052K		Total #windows	238,212,237	222,127,471	123,902,626	124,886,259

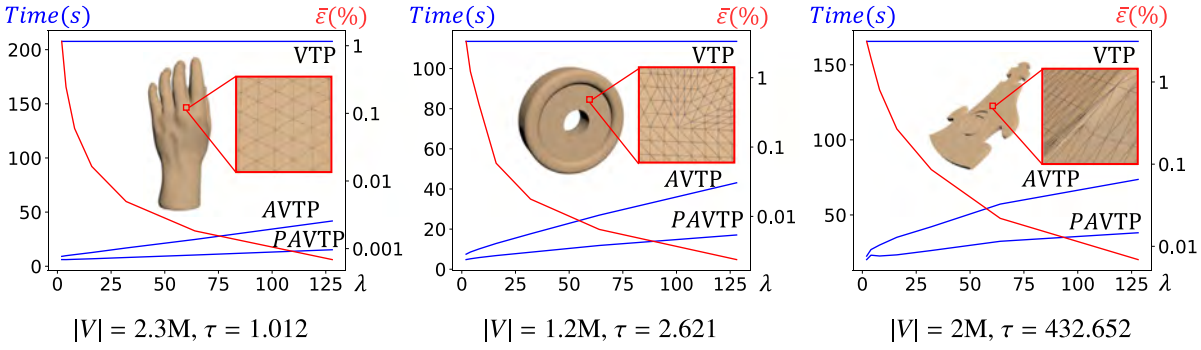

Fig. 10. Performance and accuracy trade-off. We select three representative models with low, middle and high degree of anisotropy for testing. The red lines are the relative mean error of AVTP and PAVTP ($T = 4$), and the blue lines are the running time of VTP, AVTP and PAVTP ($T = 4$). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

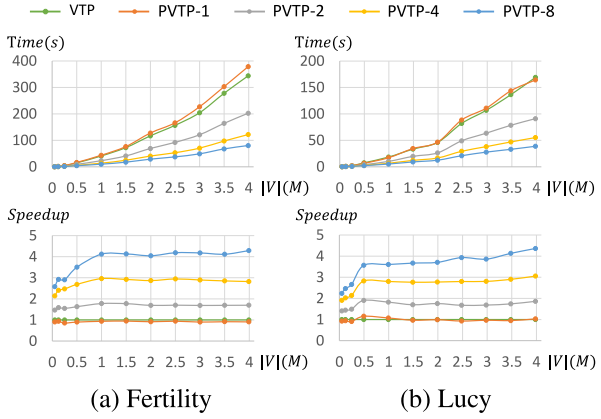

Fig. 11. The time complexity and speedup of PVTP with $T = 1$, 2, 4 and 8 on two representative models. The horizontal axis shows the number of vertices and the vertical axis is the running time/speedup factor.

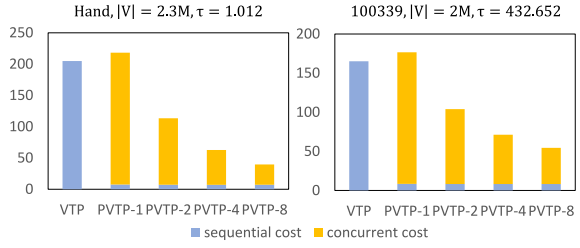

Fig. 12. Profiling of PVTP on an isotropic model (Hand) and an anisotropic model (Model 100339 from the Thingi10k dataset). The yellow and blue parts are the running time of the parallel step and the sequential steps respectively. PVTP-i denotes the parallel-VTP algorithm with $T = i$ threads. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

6.3. Parallel and approximate algorithm

It is worth noting that the proposed parallelization and approximation techniques can be either applied separately or combined together. The previous subsections report the performance of each technique separately. In this subsection, we discuss the combination of the two techniques.

We test our parallel approximate VTP (PAVTP) for regularly tessellated models. See Table 1. Herein, we list the performance of PAVTP with $\lambda = 8, 32$ under 4 threads. The performance advantage of PAVTP is even more significant on models with higher resolution or bulging shapes (with a long wavefront during distance propagation). For example, PAVTP is 29 times faster than VTP on the Golfball model with $\lambda = 8$. Fig. 10 also illustrates the

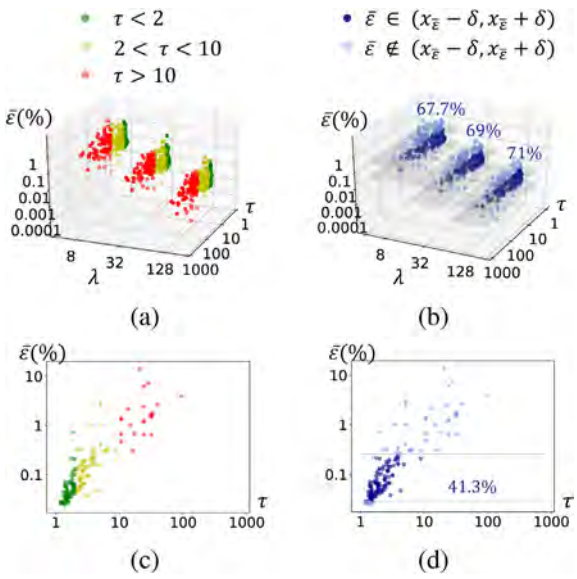

Fig. 13. Relative mean error distribution of AVTP (row 1) and AICH (row 2). (a) Results of AVTP with $\lambda = 8, 32$, and 128, respectively. (b) Models around average value $x_{\bar{\varepsilon}}$ are marked in dark blue for AVTP. (c) Results of AICH with $\lambda_A = 0.1$. (d) Models around average value $x_{\bar{\varepsilon}}$ are marked in dark blue for AICH. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

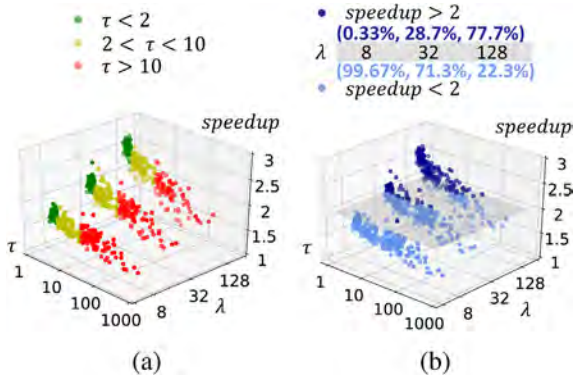

Fig. 14. Speedup factor of PAVTP with $T = 4$ over AVTP. (a) Speedup factor under different λ and anisotropy degree τ. (b) Models whose speedup factor is bigger than 2 are marked in dark blue. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

performance of PAVTP under different λ, which shows that PAVTP runs faster under the same accuracy requirement.

We compare the performance of PAVTP and AVTP on anisotropic models in Fig. 14. We observe that the speedup factor depends on the parameter λ: the larger the parameter λ, the higher the speedup. For example, 77.7% models enjoy a speedup factor greater than 2 with $\lambda = 128$. The performance improvement of PAVTP also depends on the triangulation quality of the input meshes, which is similar to the observations of PVTP. See Section 6.1.1.

7. Conclusion and future work

We extend the vertex-oriented triangle propagation algorithm [7] - the state-of-the-art method for exact discrete geodesics — via parallelization and approximation. To avoid conflicts in data writing, the parallel VTP algorithm proceeds with 3 steps in each iteration: K-nearest window list selection, parallel window list propagation, and vertex distance updating and window list merging. Experimental results show that PVTP runs 2.5∼3 times faster than the sequential VTP algorithm using 4 threads and 4∼5 times faster using 8 threads on triangle meshes with 1 million vertices and fairly regular tessellations. For challenging anisotropic meshes, PVTP is still 1.5∼3 times faster than VTP. We also propose an approximate variant of VTP that balances accuracy and speed by a global parameter λ. Our idea is to reset window on the wavefront when its radius is a multiple of λh, where h is the average length of mesh edges. AVTP becomes Dijkstra's algorithm when λh is less than the minimal edge length and becomes the exact VTP algorithm when λh is greater than the longest geodesic distance on the model. We prove that AVTP has $O(n\lambda)$ time complexity. It is worth noting that AVTP can also be parallelized under the same framework of PVTP. Our source code is available at https://github.com/djie-0329/PVTP.

Future work. We implemented PVTP on CPUs, which have relatively few cores comparing to GPUs. Since the bucket data structure and K-window-list selection are effective to control the total number of propagated windows, it is interesting to explore their potential on GPUs in the near future. Though this paper focuses on the single-source-all-destinations geodesics, applying PVTP *locally* can improve the performance of constructing saddle vertex graphs [16] and discrete geodesic graphs [17,18]. There are also other potential applications such as motion planing with polyhedron obstacles, that require efficient and accurate geodesic computation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank the authors of VTP [7] for releasing their source code and the anonymous reviewers for their constructive comments. This project is partially supported by Singapore MOE Grant RG20/20 and National Natural Science Foundation of China (61772016).

References

[1] Bose P, Maheshwari A, Shu C, Wuhrer S. A survey of geodesic paths on 3D surfaces. Comput Geom 2011;44(9):486–98.
[2] Mitchell JSB, Mount DM, Papadimitriou CH. The discrete geodesic problem. SIAM J Comput 1987;16(4):647–68.
[3] Chen J, Han Y. Shortest paths on a polyhedron. In: Proceedings of the sixth annual symposium on computational geometry; 1990. p. 360–9.
[4] 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.
[5] Xin S-Q, Wang G-J. Improving Chen and Han's algorithm on the discrete geodesic problem. ACM Trans Graph 2009;28(4):104:1–8.
[6] Xu C, 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.
[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] Mitchell JS. Geometric shortest paths and network optimization. In: Handbook of computational geometry. Elsevier Science Publishers B.V. North-Holland; 1998, p. 633–701.
[9] Peyré G, Péchaud M, Keriven R, Cohen LD. Geodesic methods in computer vision and graphics. Found Trends Comput Graph Vision 2010;5(3–4):197–397.

[10] Qin Y. Fast and exact geodesic computation using edge-based windows grouping (Ph.D. thesis), Bournemouth University, National Centre for Computer Animation; 2017.
[11] Ying X, Xin S, He Y. Parallel chen-han (PCH) algorithm for discrete geodesics. ACM Trans Graph 2014;33(1):9:1–9:11.
[12] Ying X, Huang C, Fu X, He Y, Yu R, Wang J, Yu M. Parallelizing discrete geodesic algorithms with perfect efficiency. Comput Aided Des 2019;115:161–71.
[13] Liu Y-J. Exact geodesic metric in 2-manifold triangle meshes using edge-based data structures. Comput Aided Des 2013;45(3):695–704.
[14] Zhong Z, Guo X, Wang W, Lévy B, Sun F, Liu Y, Mao W. Particle-based anisotropic surface meshing. ACM Trans Graph 2013;32(4).
[15] Xin S-Q, Wang G-J. 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.
[16] Ying X, Wang X, He Y. Saddle vertex graph (SVG): A novel solution to the discrete geodesic problem. ACM Trans Graph 2013;32(6):170:1–170:12.
[17] 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 Design 2017;52:262–84, Geometric Modeling and Processing 2017.
[18] Adikusuma YY, Fang Z, He Y. Fast construction of discrete geodesic graphs. ACM Trans Graph 2020;39(2):14:1–14.