Computational Geometry 73 (2018) 15–23

Contents lists available at ScienceDirect

Computational Geometry: Theory and Applications

www.elsevier.com/locate/comgeo

Parallelized ear clipping for the triangulation and constrained Delaunay triangulation of polygons

Günther Eder, Martin Held *, Peter Palfrader

Universität Salzburg, FB Computerwissenschaften, 5020 Salzburg, Austria

ARTICLE INFO

Article history:
Received 2 July 2015
Received in revised form 19 November 2016
Accepted 10 January 2018
Available online 7 May 2018

Keywords:
Polygon triangulation
Constrained Delaunay triangulation (CDT)
Parallelization
Multi-core
Ear-clipping

ABSTRACT

We present an experimental study of strategies for triangulating polygons in parallel on multi-core machines, including the parallel computation of constrained Delaunay triangulations. As usual, we call three consecutive vertices of a (planar) polygon an ear if the triangle that is spanned by them is completely inside the polygon. Extensive tests on thousands of sample polygons indicate that about 50% of vertices of most polygons form ears. This experimental result suggests that polygon-triangulation algorithms based on ear clipping might be well-suited for parallelization.

We discuss three different approaches to parallelizing ear clipping, and we present a parallel edge-flipping algorithm for converting a triangulation into a constrained Delaunay triangulation. All algorithms were implemented as part of Held's FIST framework. We report on our experimental findings, which show that the most promising method achieves an average speedup of 2–3 on a quad-core processor. In any case, our new triangulation code is faster than the sequential triangulation codes Triangle (by Shewchuk) and FIST.

1. Introduction

1.1. Ear clipping

Consider a simple polygon P in the plane with n vertices $v_0, \ldots, v_{n-1}$. Throughout this paper we take all vertex indices modulo n. As usual, we call a vertex convex (reflex, resp.) if its interior angle is less than (greater than, resp.) 180°. Three consecutive vertices $(v_{i-1},\ v_i,\ v_{i+1})$ form an *ear* if the open line segment $\overline{v_{i-1}, v_{i+1}}$ is completely contained in the interior of P; see Fig. 1. In other words, (v_{i-1}, v_i, v_{i+1}) is an ear of P if and only if the line segment $\overline{v_{i-1}, v_{i+1}}$ forms a diagonal of P, or, equivalently, if and only if (i) v_i is convex and (ii) the closure of the triangle $\Delta(v_{i-1}, v_i, v_{i+1})$ does not contain any reflex vertex of P (except possibly v_{i-1} or v_{i+1}). Clipping such an ear by inserting the diagonal $\overline{v_{i-1}, v_{i+1}}$ cuts off the vertex v_i, the "base" of the ear, thus eliminating v_i from the boundary of P and reducing the number of vertices of P by one.

The basic idea of an ear-clipping algorithm is to iteratively cut off ears until the polygon has shrunk to a triangle. Typically, an implementation of an ear-clipping algorithm will operate in two phases:

Classification: Scan the boundary of P and determine all instances of three consecutive vertices that form an ear of P. These candidate ears are stored in a queue.

* Corresponding author.
E-mail addresses: geder@cosy.sbg.ac.at (G. Eder), held@cosy.sbg.ac.at (M. Held), palfrader@cosy.sbg.ac.at (P. Palfrader).

https://doi.org/10.1016/j.comgeo.2018.01.004

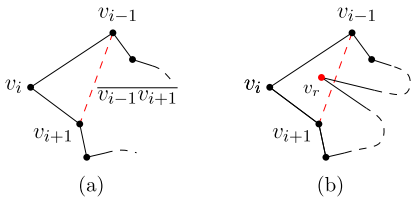


Fig. 1. (a) An ear defined by the vertices v_{i-1}, v_i, v_{i+1}; (b) A reflex vertex v_r is contained in the triangle $\Delta(v_{i-1}, v_i, v_{i+1})$. Thus, $\overline{v_{i-1}v_{i+1}}$ is not a diagonal of P and the triangle is not an ear of P.

Clipping: Iteratively pick a candidate ear from the queue and clip it if it is still valid. As an ear (v_i, v_j, v_k) is clipped and stored in a triangle list, its two outer vertices v_i and v_k have to be checked to determine whether they form the bases of new ears after the clipping of (v_i, v_j, v_k). Every newly found ear is added to the queue.

Note that the queue may contain candidate ears which are no longer part of the polygon, i.e., which have become invalid. For instance, if (v_{i-1}, v_i, v_{i+1}) and (v_i, v_{i+1}, v_{i+2}) both are ears, then the clipping of one of them renders the other ear invalid.

The ear-clipping process ends for an n-vertex input polygon P after $n-3$ ears have been clipped and, thus, the triangle list together with the final triangle forms a complete triangulation of P. The correctness of this approach hinges upon Meisters' two-ears theorem [1], which states that every simple polygon with four or more vertices has at least two non-overlapping ears.

1.2. FIST

Held's fast industrial-strength triangulation framework FIST [2] is a C++ code for polygon triangulation based on ear-clipping. Key features of FIST are its speed and robustness. While the basic ear-clipping algorithm has an $\mathcal{O}(n^2)$ worst-case complexity, FIST employs multi-level geometric hashing to speed up the computation to near-linear time for almost all (real-world and contrived) inputs. Extensive tests [3] show that FIST's careful engineering allows it to run flawlessly on standard floating-point arithmetic.

FIST provides heuristics that try to avoid clipping sliver triangles. In a nutshell, FIST stores all candidate ears in a priority queue and always chooses the best ear for clipping, where the meaning of "best" depends on the specific quality heuristic used. Its default heuristic is "top", an allusion to "top-quality triangulation": The more an ear deviates from an equilateral triangle, the lower is its quality. The "top" heuristic improves the quality of the triangulation at the cost of a slightly increased average run-time. It does, however, also help to avoid excessive (worst-case) run-times by avoiding the clipping of sliver ears with very long diagonals that may cause FIST's geometric hashing to perform poorly.

Ear clipping is, ostensibly, limited to triangulating simple polygons without island contours. FIST, however, also handles multiply-connected polygons by converting them in a pre-processing step: so-called *bridges* are inserted to connect all island polygons directly or indirectly to the outer boundary polygon, turning a polygon with islands into one (slightly degenerate) simple polygon which can then be triangulated using ear clipping. Care has been taken in recent revisions of FIST to ensure that the bridge finding does not have an adverse effect on the run-time even when hundreds of thousands of islands are to be handled.

1.3. Prior work

Surprisingly little work has been done on computing triangulations of simple polygons using a coarse-grain parallelization that would be suitable for multi-core computers. Besides some rather theoretical papers on fine-grain parallelizations, e.g., [4–6], prior work focuses mostly on parallel (Delaunay) triangulations of point sets rather than polygons.

In 2008, Rong et al. introduce a hybrid approach to compute Delaunay triangulations [7]. The major part of the computation takes place on the GPU, and a speedup of 1.5 compared to Shewchuk's Triangle [8] is claimed.

Xin et al. [9] present an algorithm to compute Voronoi diagrams on the GPU by using a sweep-circle, a variant of Fortune's sweep-line geared towards parallelization and well suited for execution on a GPU. The plane is split into cells, and the algorithm launches for each cell independently and correctly computes the Voronoi diagram within its cell with relatively little overhead.

In 2013, Qi et al. [10] introduce a primarily GPU-based algorithm to compute constrained Delaunay triangulations (CDT). They first compute a point Voronoi diagram and then add constraints to obtain the constrained Delaunay triangulation. Their CUDA implementation scales well on the GPU and seems to be the currently best solution if an NVIDIA GPU is available.

A few papers focus on the related field of parallel constrained Delaunay mesh generation [11–13]. E.g., Kot et al. [12] try to speed-up mesh generation for huge data sets. Andrey and Chernikov [13] minimize communication costs in parallel constrained Delaunay meshing on over 100 processors.

1.4. Our contribution

We analyze the prevalence of ears in our vast set of test data (see Sec. 3) and find that, on average, about half of all vertices of a polygon form the bases of ears. Furthermore, it is very likely that a convex vertex belongs to an ear: About 98% of convex vertices form the bases of ears.

We therefore extend the classic FIST ear-clipping algorithm such that it can operate in parallel. We present three different variants: a divide-and-conquer algorithm, an algorithm that uses a partitioning of the boundary, and a mark-and-cut approach. All algorithms were implemented within the FIST framework and compared to the conventional (sequential) FIST.

FIST uses several heuristics to improve the triangulation quality. In our parallel algorithms it is difficult to take triangle quality into consideration. To mitigate this shortcoming, we extend FIST by an additional edge-flipping step which allows FIST to compute a constrained Delaunay triangulation of the polygon.

We conclude our paper with a detailed experimental evaluation of the C++ implementations of our new algorithms.

2. Parallel ear-clipping algorithms

In the sequential version of FIST, on average, about 80% of the time is spent on the classification and clipping of ears, while only the remaining 20% are spent on preprocessing, such as data cleaning and bridge finding (if the input polygon has islands). Therefore, we concentrate our parallelization efforts on the classification and clipping of ears.

Let k denote the number of threads that shall be used in the parallel triangulation. (Typically, this number will match the number of CPU cores of the target computing platform.) In this section we discuss three algorithms for extending FIST such that it can operate in parallel using k threads. The algorithms differ in how they split the polygon and its ears such that the subsequent ear clipping can be carried out in parallel. Care is taken to avoid any need for (possibly costly) synchronization among the threads.

2.1. Divide and conquer

The basic idea of the *divide-and-conquer* approach is to split the polygon into as many roughly equal-sized sub-polygons as parallel threads shall be used. Since it is costly to determine suitable diagonals that achieve balanced splits, we use vertical lines to split the polygon.

Sutherland and Hodgman [14] describe how to clip a polygon P along a vertical line ℓ. Their algorithm uses an initially empty list L to store the resulting, clipped polygon. In order to populate L, one walks along the boundary of P and considers all pairs of consecutive vertices a and b. For each pair the following four cases of positions of a and b relative to an oriented line ℓ can be distinguished: (1) If both a and b are right of ℓ then b is added to L. (2) If a is right of and b is left of ℓ then the intersection $\overline{ab} \cap \ell$ of the line segment $\overline{ab}$ and the line ℓ is added as new vertex to L. (3) If a is left of and b is right of ℓ then the intersection $\overline{ab} \cap \ell$ as well as b is appended to L. (4) Lastly, if both a and b are left of ℓ then nothing is added to L.

We employ a variant of the Sutherland–Hodgman algorithm to split P into k sub-polygons along $k-1$ vertical split-lines. Splitting an n-vertex polygon P once along a line ℓ is done in time $\mathcal{O}(n)$ at a cost of adding at most $\mathcal{O}(n)$ Steiner points which arise from intersections between ℓ and the edges of P. Our tests showed that this is a pessimistic estimate of the number of Steiner points since $\sqrt{n}$ Steiner points seem to suffice for almost all but contrived inputs. Note that the (repeated) application of the Sutherland–Hodgman algorithm may result in degenerate sub-polygons that contain overlapping edges.

In order to achieve balanced splits the ith split-line is positioned right after the m_ith-largest x-coordinate of vertices of P, with $m_i := \lceil i \cdot n/k \rceil$. Since FIST sorts all vertices according to their x-coordinates during the preprocessing, the x-coordinates that define the split-lines are readily obtained. Note that we never position a split-line precisely at a vertex of P to avoid special cases. The first split is carried out using the split-line at $m_{k/2}$ resulting in two sub-polygons of roughly equal size. Next, we split using the split-lines at $m_{k/4}$ and $m_{3k/4}$. We continue this process until we have k sub-polygons. Thus, the splitting of P into k parts can be expected to run in $O(n \log k)$ time, except for a few contrived inputs that require lots of Steiner points. Recall that we are shooting for a coarse-grain parallelization and, thus, k is a small constant factor.

We then run one (sequential) FIST instance per thread to obtain a triangulation of each sub-polygon. As the individual sub-polygons do not overlap, the individual triangulation runs are independent, too. In particular, the parallel triangulation runs do not require any kind of synchronization between them.

Gluing the triangulations of all sub-polygons together yields a triangulation of P, albeit with Steiner points which have to be removed. Consider a pair of Steiner points s_a and s_b such that the line segment $\overline{s_a s_b}$ of ℓ lies inside of P. We create a hole H by removing all triangles incident to s_a or s_b; see Fig. 2a. The boundary of H forms a "double-star-shaped" polygon, where every point of H is visible from at least one of the Steiner points s_a and s_b. Due to this specific property there are several simple approaches to re-triangulate H.

Consider the two triangles (s_a, s_b, p) and (s_a, s_b, q) that previously shared the edge $\overline{s_a s_b}$; see Fig. 2b. Note that $\overline{pq}$ forms a valid diagonal of H. Hence, we can divide H into two star-shaped polygons H_1, H_2 by adding the diagonal $\overline{pq}$. Each of H_1, H_2 has one of s_a, s_b in its nucleus; see Fig. 2c.

Let H_1 be the polygon with s_a on its boundary. Now every triangle (s_i, s_j, s_k) formed by a triple of consecutive vertices of H_1 forms an ear and can be clipped if s_j is a convex vertex and if the triangle does not contain s_a. The triangulation of

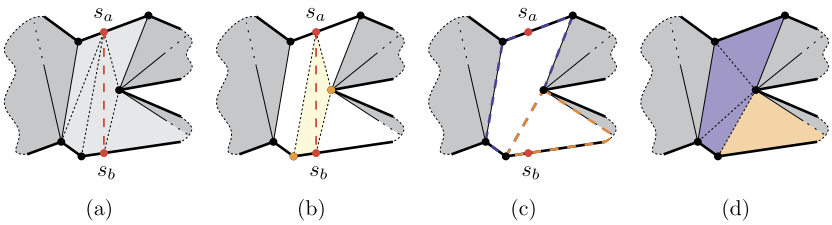


Fig. 2. The repair process used in the divide-and-conquer algorithm.

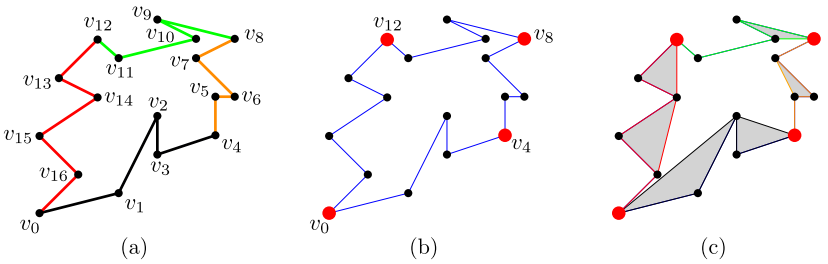


Fig. 3. Partition and cut: (a) A simple polygon P already partitioned into four chains. (b) The four landmark vertices are highlighted. (c) A partial partition-and-cut triangulation.

H_1 is completed by clipping the one remaining triangle that contains s_a. Similarly for H_2 and s_b; see Fig. 2d. Note that if p and q are both adjacent to one of the Steiner points s_a, s_b, then we do not need to introduce $\overline{pq}$ and one of H_1, H_2 is empty.

This re-triangulation of H can be implemented to run in time linear in the number of vertices of H. Hence, H can be triangulated quickly and easily.

2.2. Partition and cut

In the *partition-and-cut* approach we regard the k vertices of P which are given by the index set $\{0, n/k, 2n/k, \ldots, (k-1)n/k\}$ as *landmark* vertices. Then we partition P into k chains at these landmarks, with one chain for each thread; see Fig. 3. Thus, two adjacent chains always share one landmark.

In order to run parallel classification phases, one for each chain, we use a per-thread queue to store candidate ears instead of one single global queue. Then, in the clipping phase, each thread processes all ears from its queue. As usual, clipping the ear (v_{i-1}, v_i, v_{i+1}) involves checking whether $v_{i\pm1}$ has become the basis of a new ear. However, such a vertex is only added to the respective queue of ears if it is not a landmark. Since each thread only clips ears in its own chain and care is taken to never remove landmarks, both the classification and clipping phase of each thread can run independently of other threads; see Fig. 3b. Note that checking the ear property at a vertex only requires read-only access to the initial global vertex list and its hash grid and, thus, no synchronization of access is required.

After all k queues are empty and the parallel clipping threads have ended, some part of P is likely to remain untriangulated. We process this part and complete the triangulation of P by using one final, sequential run of FIST.

2.3. Mark and cut

The key observation of the *mark-and-cut* approach is the following: Every second ear along the boundary of P does not overlap with any other evenly-numbered ear. Therefore, we can process every second ear and clip it independently from all other ears.

The phases used by this approach thus differ slightly from the previous two algorithms: In the *mark phase* we walk along the boundary of P and store the index of every second convex vertex in an array A; see Fig. 4b. In the *cut phase*, which can be run by many threads in parallel, we consider every vertex in A, and if it forms the basis of an ear we clip it immediately; see Fig. 4c.

One thread is tasked with running the mark phase. As soon as it has processed half of the polygon boundary, the remaining threads launch a cut phase on the indices stored in A so far while the first thread continues until it has processed the remainder of the boundary.

Once all threads are finished, the cutting threads are re-launched on the vertices that have since been added to A. The marking thread now revisits what remains after the parallel ear-clipping on the first half of the polygon boundary. This continues until only a small number of ears are found in a cut phase. We then use one sequential run of FIST on the remaining polygon to finish the triangulation. (In our tests we switched to the sequential FIST once fewer than 20 new triangles were generated in one cutting phase.)

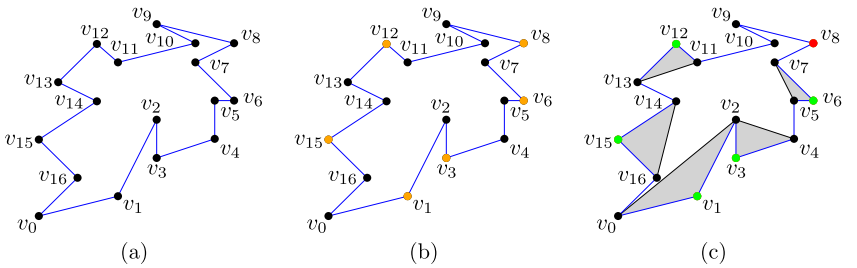


Fig. 4. Mark and cut: (a) A simple polygon P. (b) P after the first mark phase, every other convex vertex marked (orange). (c) P after the first clip phase; the red dot marks a vertex which is not the base of an ear. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

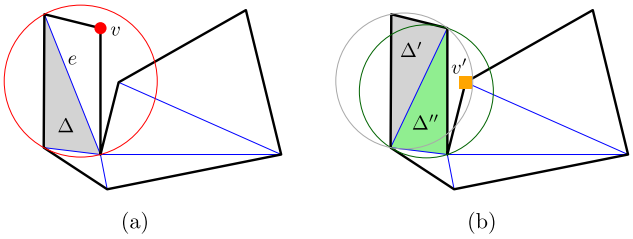


Fig. 5. The triangulation in (a) is not a constrained Delaunay triangulation as the triangle Δ does not fulfill the empty-circle condition: The vertex v is within the circumcircle of Δ and visible from all vertices of Δ. Flipping the triangulation edge e creates Δ' and Δ'', establishing a constrained Delaunay triangulation of the polygon; see (b). Note that the vertex v' is not visible from all vertices of Δ' and Δ'', i.e., it is hidden behind a constraint, and so does not violate the empty-circle conditions for Δ' and Δ''.

To store the triangles without conflict in parallel we use the following property: Every ear can be clipped only once and for every clipped ear only one vertex is removed from the boundary of P. Thus, we can use the index of the vertex at the base of the removed ear as an index to store the triangle and avoid collisions.

2.4. Constrained Delaunay triangulation

FIST provides several heuristics to avoid sliver triangles and produce "good-quality" triangulations. These heuristics do not work well for our parallel ear-clipping variants since we cannot maintain one priority queue from which the ears to be clipped are chosen according to their quality. In particular, in the mark-and-cut variant, we just clip ears without accounting for any measure of quality. In the partition-and-cut variant, each partition is able to employ the same heuristics as in the sequential variant but only within its own subset of the boundary of P.

To avoid sliver triangles as much as possible, we decided to incorporate an optional post-processing step into our parallelized FIST that converts a triangulation computed by one of our parallel ear-clipping algorithms into a constrained Delaunay triangulation (CDT). By standard definition, a triangulation T of a polygon P is a constrained Delaunay triangulation of P if every triangle Δ of T meets the empty-circle condition: The circumcircle of Δ does not contain any vertex of P in its interior that is visible within P from all three vertices of Δ; see Fig. 5 for an example. It is well-known that a constrained Delaunay triangulation P maximizes (in the lexicographic order) the vector of the interior angles of all triangles of a triangulation over all triangulations of P.

A CDT is obtained by applying edge flipping to one of FIST's triangulations. That is, we flip edges of a triangulation T until all triangles of T satisfy the empty-circle condition. We employ Shewchuk's [15] robust in-circle predicate to test for violations of the empty-circle condition. This edge-flipping approach is easy to implement and transforms an arbitrary triangulation T of an n-vertex polygon P into a correct CDT of P in at most $\mathcal{O}(n^2)$ edge flips: This bound is known for point sets where any triangulation can be converted into a Delaunay triangulation by applying $\mathcal{O}(n^2)$ edge flips [16]. It is easy to see that this bound also holds for polygon triangulations.

The idea for parallelizing the edge-flipping transformation of T is to partition T into k disjoint sets of triangles; see Fig. 6b. Edge flipping is then conducted on each triangle set in parallel until the empty-circle condition is fulfilled within each set. Finally, triangles which have at least one neighbor in a different set are checked and the empty-circle condition is established, recursively, using further edge flips.

We tested two algorithms for partitioning the triangles of a triangulation into subsets, with one subset per thread: depth-first and breadth-first traversal of the dual graph of the triangulation. Both algorithms start at a randomly chosen ear triangle of T, mark it with a number that corresponds to the current set, and continue with its neighbors. All neighbors are stored temporarily in either a stack or a queue structure and eventually processed in the same way. We get a depth-first traversal (breadth-first traversal, resp.) if a stack (queue, resp.) is used. Note that both traversals may result in a fragmentation of the triangles, i.e., in several disconnected sets of triangles that are processed by one thread.

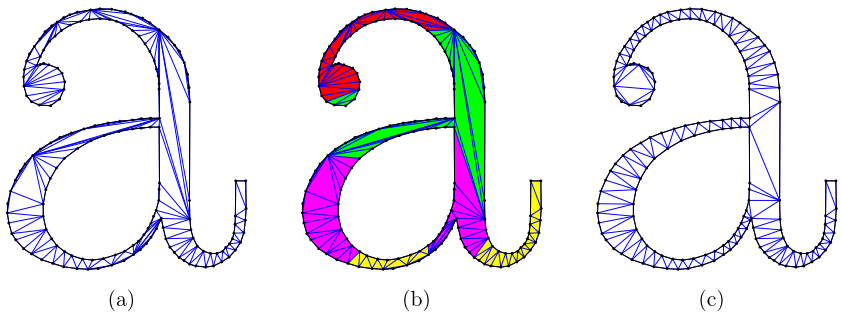


Fig. 6. (a) A triangulation T of a polygon P. (b) T partitioned into four sets. (c) T after the edge flipping, corresponding to the CDT of P.

The example polygon in Fig. 6 contains 169 triangles. To obtain its CDT using four threads we partition T into four sets of triangles and perform 134 parallel edge flips. Finally, we need 15 sequential edge flips on the triangles at the partition boundaries between the triangle sets.

3. Experimental results

We implemented the three parallel variants of FIST and the edge-flipping extension to achieve a CDT as described in the previous section. Our implementations are based on the existing FIST tool, which is written in C++, and thus, likewise, are in C++. To achieve multi-core parallelization, we employed OpenMP, an API that supports shared-memory multiprocessing on several platforms and languages, among them Linux and C++. Our test system runs CentOS 6.5 on an 2014 Intel Xeon E5-2667 v3 CPU at 3.20 GHz with 8 cores and 132 GB RAM.

Our implementations were tested on about 20 000 polygons with up to four million vertices per input, consisting of both real-world and synthetic data that exhibits various characteristics. The test data was collected by Held over the past thirty years and includes proprietary CAD/CAM designs, sampled printed-circuit board layouts, geographic maps, sampled spline curves and font outlines, closed fractal and space-filling curves, as well as star-shaped and various types of "random" polygons generated by RPG [17].

Some datasets contain circular arcs, which we approximated by polygonal chains in a preprocessing step. Similarly, we used the standard sequential FIST to convert all multiply-connected polygonal areas to (degenerate) simple polygons by inserting bridges. It is difficult to trace back the origins of the real-world data sets. As a rule of thumb, small data sets tend to be of CAD/CAM origin while the large real-world data sets represent mostly GIS data and (straight-line approximations of) printed-circuit boards.

3.1. Number of ears and quality of triangles

A key motivation for this work was our observation that most polygons seem to have significantly more ears than the two ears guaranteed by Meisters' theorem [1], suggesting that clipping many ears simultaneously is a feasible means to speed up triangulation. The prevalence of ears in the polygons of our test data is shown in Fig. 7: About half the vertices of almost all polygons form the bases of ears.

We now turn our attention to the "quality" of the triangulations generated by our parallel variants of FIST. We compare the average deviation of the interior angles from 60° of triangulations computed by FIST's default sequential "top" ear-clipping heuristic to those of constrained Delaunay triangulations and the triangulations computed by our new parallel partition-and-cut, mark-and-cut, and divide-and-conquer algorithms; see Table 1. Similarly, we analyze the vector sum of minimal angles from all triangles of an input. This vector sum is then normalized by the triangle count and again averaged over all inputs. While the numbers in Table 1 present a highly condensed assessment of the quality of triangulations, it becomes obvious that FIST's "top" ear clipping generates triangulations whose angles are not much worse than those of a CDT. Of course, since this is nothing but a heuristic, contrived inputs could always lead to bad-quality triangulations.

The mark-and-cut algorithm never stores ears to select the best ear according to some quality heuristic but clips them immediately and, therefore, is not able to apply any heuristic for improving the quality of triangulations. It comes as no surprise that this results in the weakest angular quality. The divide-and-conquer and partition-and-cut algorithms turn out slightly better since they can employ FIST's quality heuristics on a per-thread basis to each chain or sub-polygon.

3.2. Performance observations

To provide a better overall perspective we continue with comparing the run-time of FIST to that of Shewchuk's Triangle [8], which is generally regarded as another leading triangulation code. As mentioned in Sec. 1.2, FIST's default set-up is to use a priority queue to decide which ear is best to clip next according to its "top" heuristic. In our tests we use "top" ear clipping in all sequential and parallel variants of FIST whenever this is possible. In Fig. 8 we plot the run-times of Triangle,

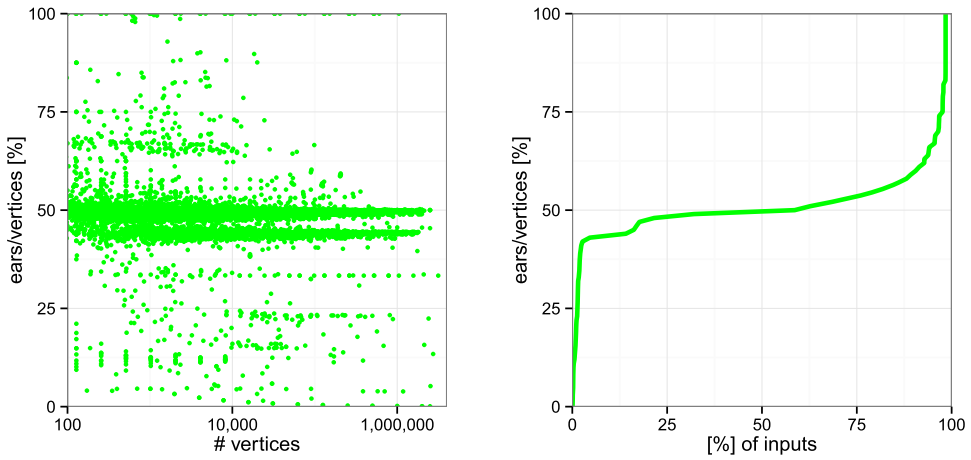


Fig. 7. For the vast majority of our sample inputs about half the vertices form ears.

Table 1
Comparison of FIST's triangulation qualities including the CDT version, the sequential version using the "top" heuristic, and the following parallel versions: Divide and Conquer (D&C), Partition and Cut (P&C), and Mark and Cut (M&C), all evaluated using 8 cores. 1) Average deviation of all interior angles from 60° over all triangulations (smaller is better). 2) Average smallest interior angle of all triangles over all triangulations (larger is better).

	CDT	FIST's "top"	D & C	P & C	M & C
1) avr. dev. 60°	30.79°	31.53°	35.29°	34.97°	38.38°
2) avr. min-vec.	24.60°	23.40°	20.07°	21.32°	21.07°

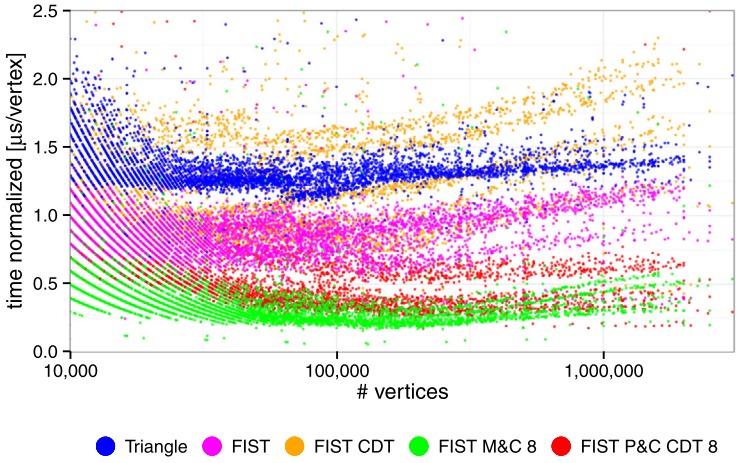
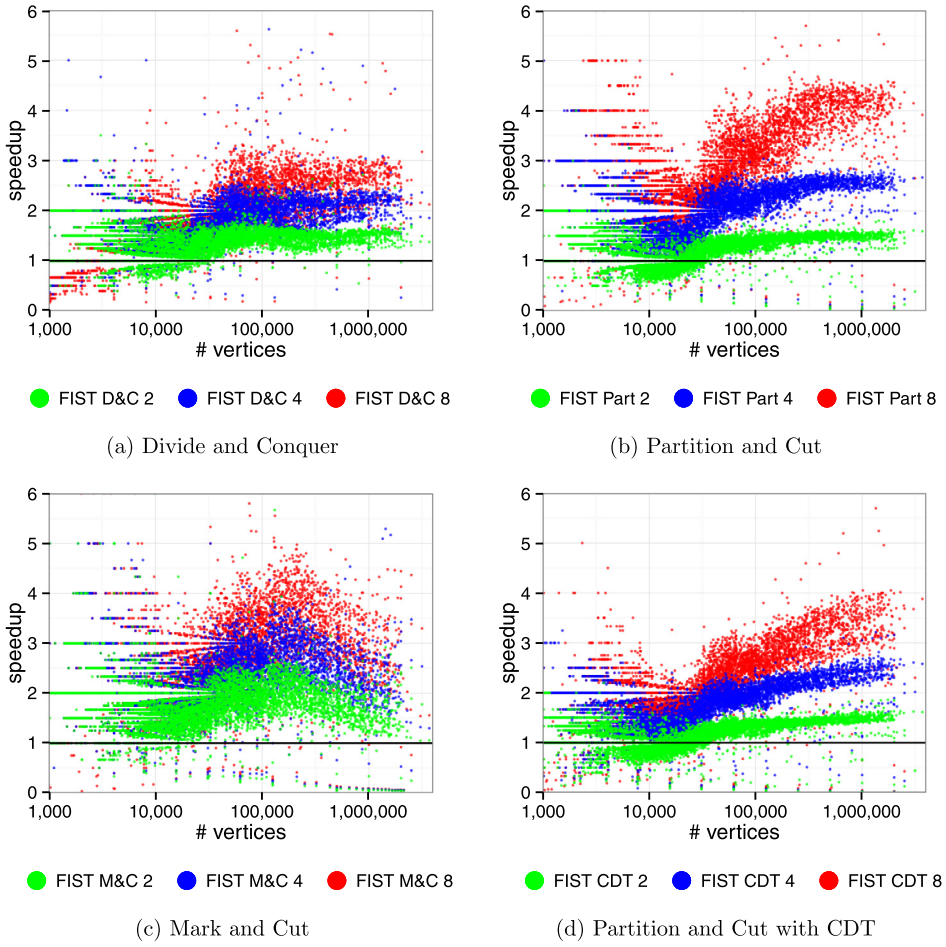


Fig. 8. Performance of Triangle compared to the sequential version of FIST, the sequential version of FIST producing a CDT, the parallel version of FIST using the mark-and-cut approach with 8 cores, and our parallel version of FIST using the partition-and-cut approach with parallel CDT on 8 cores. The y-axis shows the run-time divided by the number of vertices. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

the sequential version of FIST, the sequential version of FIST-CDT, i.e. FIST with sequential edge flipping to achieve a CDT, the parallel version of FIST using the mark-and-cut approach, and the parallel version of FIST-CDT using the partition-and-cut approach with parallel edge flipping; both parallel variants on 8 cores. Note that FIST-CDT by necessity produces the same triangulation as Triangle. While FIST tends to be faster than Triangle for most inputs, the run-times of FIST-CDT and Triangle are (roughly) comparable. A closer inspection of the timings suggests that the additional computational cost incurred by the edge flipping becomes more noticeable as the number of vertices of the input polygons increases. Extrapolating the trend observed, we would assume Triangle to be faster than FIST-CDT for most inputs when the vertex count exceeds a few million. Unfortunately, we do not have enough complex polygons to test this hypothesis in a statistically meaningful way.

We now turn our attention to the speedups achieved by our parallel variants of FIST and FIST-CDT. In our tests we compared the run-time of our parallel variants to the run-time of the conventional sequential FIST. The plots of Fig. 9



(a) Divide and Conquer

(b) Partition and Cut

(c) Mark and Cut

(d) Partition and Cut with CDT

Fig. 9. Speedups of the parallel variants as a function of input size, plotted for different numbers of threads. For the divide-and-conquer approach, (a), this is the same as the number of sub-polygons used.

show the speedups that we achieved. We observe the overall best results for both the partition-and-cut (Fig. 9b) and the mark-and-cut (Fig. 9c) algorithms, while the speedup achieved by the divide-and-conquer algorithm is considerably lower. Our tests make it evident, though, that it is difficult to select an overall winner. For polygons with about 100 000 vertices both methods achieve a speedup of about 2–3 for four threads and 3–4 for eight threads. However, while the speedup achieved by the mark-and-cut algorithm deteriorates for polygons with significantly more than 100 000 vertices, the performance of the partition-and-cut algorithm does not change.

Today's desktop computers tend to have four cores, and in such a setting the partition-and-cut variant yields a speedup of about 2–3 over a wide range of input sizes. In any case, our parallelization efforts seem to pay off only once the input polygon has at least about 15 000 vertices.

A closer look at the plot in Fig. 9 for the divide-and-conquer and mark-and-cut approaches reveals a few outliers, i.e., instances where we observe a speedup larger than k for k cores. A detailed investigation of the datasets in question suggests that these outliers can be explained as follows: The standard FIST iterates over all vertices and for each vertex v first checks whether it is convex and then checks whether it is the base of ear. If v is the base of an ear, this ear is then added to the priority queue. Later, when an ear at v_i gets clipped, this clipping will invalidate the ear properties at v_{i-1} and v_{i+1}. As a result, the clipping of v_i results in a waste of time spent on the classification of v_{i-1} and v_{i+1}. The mark-and-cut approach processes vertices slightly differently, avoiding the potential waste of having positively established an ear property for a vertex which later then gets invalidated. Indeed, a comparison between a sequentialized version of the mark-and-cut algorithm and the regular FIST already shows an average performance increase of 1.5, with an increase of up to 2.5 for some inputs even. The running time of the mark-and-cut algorithm using k cores compared to this sequentialized version shows a stable result without scattering and with the outliers contained in Fig. 9c. Note, though, that this speed-up achieved by the mark-and-cut algorithm over the conventional FIST comes at the expense of more sliver triangles; recall Table 1.

For the divide-and-conquer approach the outliers seem to be caused by FIST's use of a hash grid. FIST does not blindly use some fixed resolution for the hash grid but applies heuristics to determine a good resolution, and even re-sizes the

grid if its resolution turns out to be poor during the actual ear clipping. Still, this is nothing but a heuristic, and generating individual grids for specific sub-polygons — as it is done by the divide-and-conquer approach — may result in a substantially better performance for some specific data set.

The set-up of the parallel CDT variant of FIST is as follows. We use the partition-and-cut algorithm to create the basic triangulation. As indicated by our tests, a triangulation produced by the partition-and-cut variant is a good candidate for a subsequent edge flipping to convert it into a CDT. Our tests show that the partitions computed by a breadth-first traversal of the dual graph of a triangulation are better suited for the parallel edge-flipping phase in practice: The triangle sets obtained by a breadth-first partitioning tend to be less fragmented and, thus, require less work during the final sequential edge flipping after the parallel edge flipping has finished.

In Fig. 9d we see the speedup of the parallel partition-and-cut algorithm with the subsequent parallel edge flipping for CDT conversion compared to the sequential version of FIST with the sequential CDT computation, FIST-CDT. We observe a maximal speedup of about 2–3 when using four threads. We also tested the number of edge flips that occur for the above setting: On average, when four threads are used, FIST-CDT flips every edge about 4 times, with 3.7 flips per edge occurring within the parallel threads.

4. Conclusion

We present experimental evidence that an algorithm for triangulating polygons based on ear clipping can be parallelized for faster execution on standard multi-core computers. We also provide experimental evidence that using FIST-CDT to obtain a constrained Delaunay triangulation via ear clipping followed by edge flipping is a viable alternative to using Shewchuk's Triangle, which computes a Delaunay triangulation on the set of vertices and then inserts the polygon edges as constraints. While the sequential FIST-CDT code is of roughly of the same speed as Triangle, our best parallel variant of FIST-CDT is faster by a factor of about two when using four parallel threads. Likely, there is room for further improvement since our implementations are not yet fully tuned. Summarizing, since current desktop computers are equipped with quad-core processors, the (constrained Delaunay) triangulation of a polygon can be accomplished on standard hardware about 2–3 times as fast with our parallel variants of FIST than when using the sequential FIST.

Acknowledgements

This work was supported by Austrian Science Fund (FWF) Grant P25816-N15. We thank an anonymous reviewer for suggesting a simplified description of the re-triangulation of a hole in the divide-and-conquer algorithm.

References

[1] G.H. Meisters, Polygons have ears, Am. Math. Mon. 82 (6) (1975) 648–651.
[2] M. Held, FIST: fast industrial-strength triangulation of polygons, Algorithmica 30 (4) (2001) 563–596, https://doi.org/10.1007/s00453-001-0028-4.
[3] M. Held, W. Mann, An experimental analysis of floating-point versus exact arithmetic, in: Proc. 23rd Canad. Conf. Comput. Geom., CCCG, 2011, pp. 489–494.
[4] M. Goodrich, Triangulating a polygon in parallel, J. Algorithms 10 (3) (1989) 327–351.
[5] K.L. Clarkson, R. Cole, R.E. Tarjan, Randomized parallel algorithms for trapezoidal diagrams, Int. J. Comput. Geom. Appl. 2 (2) (1992) 117–133.
[6] M. Goodrich, Planar separators and parallel polygon triangulation, J. Comput. Syst. Sci. 51 (3) (1995) 374–389, https://doi.org/10.1006/jcss.1995.1076.
[7] G. Rong, T.-S. Tan, T.-T. Cao, Stephanus, Computing two-dimensional Delaunay triangulation using graphics hardware, in: Proc. ACM Symp. Interactive 3D Graphics, I3D '08, 2008, pp. 89–97.
[8] J.R. Shewchuk, Triangle: engineering a 2D quality mesh generator and Delaunay triangulator, in: Appl. Comp. Geom.: Towards Geom. Eng., in: Lecture Notes in Computer Science, vol. 1148, Springer, ISBN 3-540-61785-X, 1996, pp. 203–222.
[9] S.-Q. Xin, X. Wang, J. Xia, W. Mueller-Wittig, G.-J. Wang, Y. He, Parallel computing 2D Voronoi diagrams using untransformed sweepcircles, Comput. Aided Des. 45 (2) (2013) 483–493, https://doi.org/10.1016/j.cad.2012.10.031.
[10] M. Qi, T. Cao, T. Tan, Computing 2D constrained Delaunay triangulation using the GPU, IEEE Trans. Vis. Comput. Graph. 19 (5) (2013) 736–748, https://doi.org/10.1109/TVCG.2012.307.
[11] L.P. Chew, N. Chrisochoides, F. Sukup, Parallel constrained Delaunay meshing, in: Proc. Joint ASME/ASCE/SES Summer Meeting Special Symp. on Trends in Unstructured Mesh Generation, 1997, pp. 89–96.
[12] A. Kot, A. Chernikov, N. Chrisochoides, Parallel out-of-core constrained Delaunay mesh generation, in: 2005 IEEE Intel. Data Acqu. and Adv. Comp. Systems: Tech. and Appl., IEEE, 2005, pp. 183–190.
[13] A.N. Chernikov, N.P. Chrisochoides, Algorithm 872: parallel 2D constrained Delaunay mesh generation, ACM Trans. Math. Softw. 34 (1) (2008) 6:1–6:20, https://doi.org/10.1145/1322436.1322442.
[14] I.E. Sutherland, G.W. Hodgman, Reentrant polygon clipping, Commun. ACM 17 (1) (1974) 32–42, https://doi.org/10.1145/360767.360802.
[15] J.R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates, Discrete Comput. Geom. 18 (3) (1997) 305–363.
[16] M.W. Bern, D. Eppstein, Mesh generation and optimal triangulation, in: Computing in Euclidean Geometry, in: Lecture Notes Series on Computing, vol. 1, World Scientific, 1992, pp. 23–90.
[17] T. Auer, M. Held, Heuristics for the generation of random polygons, in: Proc. 8th Canad. Conf. Comput. Geom., CCCG, 1996, pp. 38–44.