Experiments on Parallel Polygon Triangulation Using Ear Clipping

Günther Eder* Martin Held* Peter Palfrader*

Abstract

We present an experimental study of different strategies for triangulating polygons in parallel. As usual, we call three consecutive vertices of a polygon an ear if the triangle that is spanned by them is completely inside of the polygon. Extensive tests on thousands of sample polygons indicate that most polygons have a linear number of ears. This experimental result suggests that polygon-triangulation algorithms based on ear clipping might be well-suited for parallelization.

We discuss three different on-core approaches to parallelizing ear clipping and report on our experimental findings. Extensive tests show that the most promising method achieves a speedup by a factor of roughly k on a machine with k cores.

1 Introduction

An ear of a planar simple polygon P is formed by three consecutive vertices (v_{i-1}, v_i, v_{i+1}) if the open line segment $\overline{v_{i-1}, v_{i+1}}$ is completely contained in the interior of P (see Fig. 1). It is well-known [2] that (v_{i-1}, v_i, v_{i+1}) form an ear of P 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}). Hence, if (v_{i-1}, v_i, v_{i+1}) is an ear of P then the line segment $\overline{v_{i-1}, v_{i+1}}$ forms a diagonal of P. Clipping this ear by inserting this diagonal cuts off the vertex v_i, the "base" of the ear, thus reducing the number of vertices of P by one.

The basic idea of ear clipping is to iteratively cut off ears until the polygon has shrunk to a triangle. The algorithm's correctness hinges upon Meisters' two-ears theorem which states that every simple polygon with four or more vertices has at least two non-overlapping ears [4].

Typically, an implementation of an ear-clipping algorithm will operate in two phases. *Classification:* Iterate along the contour of P to determine all instances of three consecutive vertices that form an ear of P. All potential ears are stored in a queue. *Clipping:* Iteratively pick an ear from the queue and clip it. 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 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. The process ends for an n-vertex input polygon P when $n-3$ ears have been clipped and, thus, the triangle list together with the final triangle forms a complete triangulation of P.

Ear clipping forms the basis of the FIST triangulation algorithm and ANSI-C implementation, Held's fast industrial-strength triangulation tool [2]. Key features of FIST include 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] showed that FIST's careful engineering allows it to run flawlessly on a standard floating-point arithmetic.

Ear clipping is, ostensibly, limited to triangulating simple polygons. FIST, however, also handles polygons with holes by converting them in a preprocessing step: so-called *bridges* are inserted to connect all hole polygons directly or indirectly to the outer boundary polygon, turning a polygon with holes into one (slightly degenerate) simple polygon, which can then be triangulated using ear clipping.

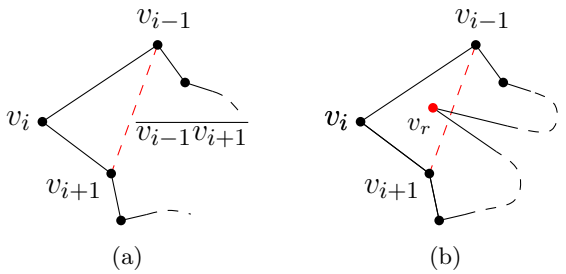


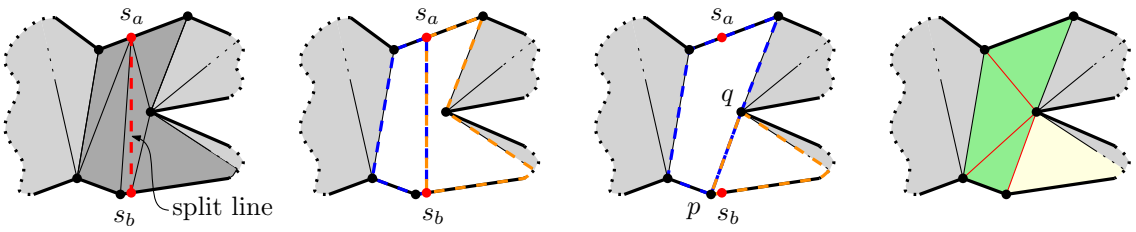
Figure 1: (a) An ear defined by the vertices v_{i-1}, v_i, v_{i+1} where v_i is convex; (b) A reflex vertex v_r violates the second condition.

2 Prior Work

Surprisingly little work has been done on parallel triangulations. The literature focuses mostly on (Delaunay) triangulations of point sets rather than polygons, see for instance Rong et al. [6] and Xin et al. [8].

In 2013, Qi et al. [5] introduced a primarily GPU-based algorithm to compute constrained Delaunay tri-

*Universität Salzburg, FB Computerwissenschaften, 5020 Salzburg, Austria; supported by Austrian Science Fund (FWF) Grant P25816-N15; {geder,held, palfrader}@cosy.sbg.ac.at.



(a) (b) (c) (d)

Figure 2: The repair process used in the divide and conquer algorithm.

angulations. In a first step they compute a Voronoi diagram, i.e., the dual of the Delaunay triangulation. Constraints are then added to obtain the constrained Delaunay triangulation (CDT). Their approach scales well on the GPU and seems to be the currently best solution if an NVIDIA GPU is available.

3 Our Contribution

We study the prevalence of ears in our vast set of test data (see Sec. 5) and find that, on average, about half of all vertices of a given polygon form the bases of ears. If we look only at convex vertices then a vast majority (98 %) of them belong to ears. This is significantly more than the two ears guaranteed by Meisters' theorem [4] and, hence, suggests that clipping many ears simultaneously is feasible.

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

4 Parallel Ear-Clipping Algorithms

In the sequential version of FIST, on average, about 80 % of the time is spent for classification and clipping of the ears, while only approximately 20 % is spent on preprocessing, such as data cleaning and bridge finding to convert polygons with holes into degenerate (weakly-)simple polygons. We therefore concentrate our parallelization efforts on the classification and clipping phases.

4.1 Divide and Conquer

The basic idea is to split the polygon into as many sub-polygons as CPU cores are available. All sub-polygons shall have roughly the same number of vertices. Since it seems costly to determine suitable diagonals that achieve balanced splits, we simply use vertical lines to split the polygon. Using the Sutherland-Hodgman algorithm [7], we can split a polygon along a line ℓ in time $\mathcal{O}(n)$, at a cost of at most $\mathcal{O}(n)$ Steiner points given by the number of intersections between ℓ and the edges of the polygon. (In practice, the number of Steiner points seems to be bounded by $\sqrt{n}$ for almost all but contrived inputs.)

We then run one (sequential) FIST instance per core to obtain a triangulation of each sub-polygon. A concatenation of the triangulations of all sub-polygons yields a triangulation of P, albeit with Steiner points which have to be removed.

Consider a pair of consecutive Steiner points s_a and s_b. We remove them and all their incident triangles, and we repair the contour by re-joining vertices that were adjacent previously. The removal of incident triangles leaves a hole H which 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, see Fig. 2b.

Using s_a as start vertex, we walk counter-clockwise along the boundary of H. If we find a vertex that is not visible from s_a, then we store the preceding vertex as p. If all vertices are visible, then we stop the test when we reach the second Steiner point s_b and store the last vertex before s_b as p. Then we start the same search clockwise, starting again at s_a, and obtain q.

We divide H into two star-shaped polygons H_1, H_2 by adding the diagonal $\overline{pq}$. Each of them has one of the original Steiner points in its nucleus, see Fig. 2c. Now every triangle based at a convex vertex of H_1 and H_2 forms an ear as long as it does not contain either s_a or s_b. Hence, both holes can be triangulated easily.

4.2 Partition and Cut

In this approach, we use the sequential classification step to partition the contour of the polygon into k different sets. We choose k to correspond to the number of cores we want to run on. The sequential FIST walks along the polygon, tests each vertex triple for its ear property, and stores all ears in one queue. To parallelize, we set the number of queues equal to k and split the polygon into k polygonal chains with roughly n/k many vertices each. Additionally, we memorize k contour vertices between the chains. These *corner vertices* are important for the parallel clipping to ensure that no thread oversteps its partition boundaries;

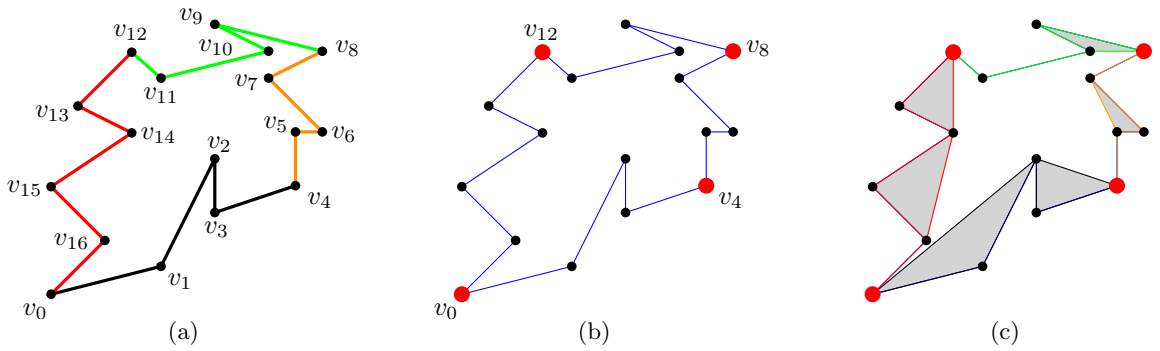


Figure 3: (a) A simple polygon P already partitioned into four chains. (b) The four corner vertices are highlighted. (c) A partition-and-cut triangulation.

see Fig. 3b.

In the parallel 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. If so, and if $v_{i\pm1}$ is no corner vertex, then $v_{i\pm1}$ is added to the queue.

After all k queues are empty, the parallel clipping phase is completed. Now we finish the triangulation with a sequential run of FIST. This will remove the remaining vertices from our polygon.

4.3 Mark and Cut

This algorithm builds on the fact that every second ear along the polygon's contour is non-overlapping. Hence, we can mark those ears and clip them immediately without conflicts. We only need one additional data structure, namely an array A to store the indices of all marked ears.

In the *mark phase*, we walk along the polygon once and store the index of every other vertex in A. Additionally we only take convex vertices and add flags to check if a triangle has already been checked for its ear-property. In the *cut phase*, we check for each vertex v_i in A whether (v_{i-1}, v_i, v_{i+1}) forms an ear. If so, we clip this ear and store the triangle $\Delta(v_{i-1}, v_i, v_{i+1})$ in the triangle array at position i. Since we considered only every other vertex, ears that we find cannot overlap.

Every ear can be clipped only once and for every clipped ear only one vertex is removed from the polygon. Thus, we can use the index i of a removed vertex v_i as an index for the stored triangle and avoid any collisions.

The algorithm is designed to work in parallel without locks or atomics. Initially, we start the mark phase and mark the first half of the contour as described above. Then we let all threads but one run through the first half of the array A in parallel and check each vertex (cut phase). The remaining thread marks the second half of the contour and stores the vertex indices in the second half of A. After all threads are finished, the procedure starts again with marking the first half of the remaining contour and cutting the vertices stored in the second half of A and so on.

Once only a low number of new triangles can be generated in a cut phase, we run the sequential version of FIST on the remaining polygon.

5 Experimental Results

We implemented the parallel variants of FIST as an on-core parallelization by the use of OpenMP/C++.

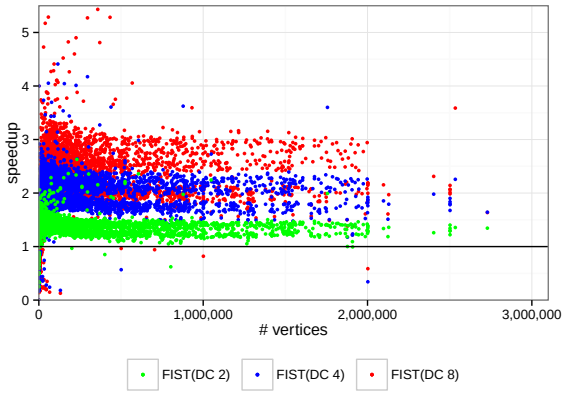
Our test system runs CentOS 6.5 on an 2014 Intel Xeon E5-2667 v3 CPU at 3.20 GHz with 8 cores and 132 GB RAM.

Our implementations were tested on about twenty thousand polygons with up to four million vertices per input, consisting of both real-world and contrived data of different characteristics, including CAD/CAM designs, printed-circuit board layouts, geographic maps, closed fractal and space filling curves, star-shaped and random polygons generated by RPG [1], as well as sampled spline curves and font outlines. 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 (slightly degenerate) simple polygons by inserting bridges.

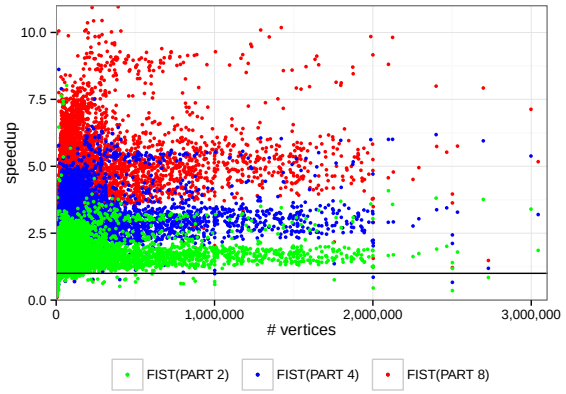
In our tests, we compare the runtime of our parallel algorithms to the runtime of the conventional sequential FIST. The plots of Fig. 4 show the speedups that we achieved.

We observe the overall best result for the mark-and-cut algorithm (Fig. 4c), with an average speedup of 8 when using eight cores or 4 when using four cores. In any case, parallelization seems to pay off once the input polygon has at least about fifteen thousand vertices. Some test sets yield a speedup of over k while employing k cores. This is a result of the different storage structure used in the sequential version of FIST.

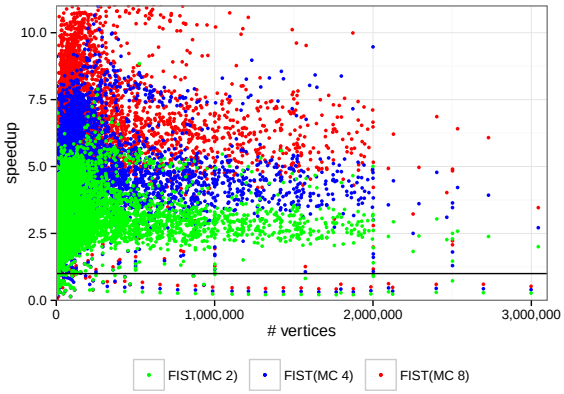
Tests on other systems with up to 64 cores did not



(a) Divide and Conquer



(b) Partition and Cut



(c) Mark and Cut

Figure 4: Speedup of parallelization approaches as a function of input size. In (a), (b), and (c) we see speedups for different numbers of threads. For the divide-and-conquer approach this is the same as the number of sub-polygons used.

scale as well as presumed. However, those systems were multi-CPU systems which seem to behave differently than the multi-core system which we used, for reasons not yet fully understood.

Summarizing, we present experimental evidence that an algorithm for triangulating polygons based on ear clipping can be parallelized for efficient execution on multi-core computers. Since current personal computers are equipped with quad-core processors, the triangulation of a polygon can be accomplished about four times as fast with our parallel variants of FIST in most cases.

References

[1] T. Auer and M. Held. Heuristics for the Generation of Random Polygons. In *Proc. 8th Canad. Conf. Comput. Geom (CCCG 1996)*, pages 38–44, Aug. 1996.

[2] M. Held. FIST: Fast Industrial-Strength Triangulation of Polygons. *Algorithmica*, 30(4):563–596, 2001.

[3] M. Held and W. Mann. An Experimental Analysis of Floating-Point Versus Exact Arithmetic. In *Proc. 23rd Canad. Conf. Comput. Geom. (CCCG 2011)*, pages 489–494, Aug. 2011.

[4] G. H. Meisters. Polygons have Ears. *The American Mathematical Monthly*, 82(6):648–651, June 1975.

[5] M. Qi, T. Cao, and T. Tan. Computing 2D Constrained Delaunay Triangulation Using the GPU. *IEEE Trans. Visualizat. Comput. Graph.*, 19(5):736–748, May 2013.

[6] G. Rong, T.-S. Tan, T.-T. Cao, and Stephanus. Computing Two-Dimensional Delaunay Triangulation using Graphics Hardware. In *Proc. ACM Symp. Interactive 3D Graphics*, I3D '08, pages 89–97, 2008.

[7] I. E. Sutherland and G. W. Hodgman. Reentrant Polygon Clipping. *C. ACM*, 17(1):32–42, Jan. 1974.

[8] S.-Q. Xin, X. Wang, J. Xia, W. Mueller-Wittig, G.-J. Wang, and Y. He. Parallel Computing 2D Voronoi Diagrams using Untransformed Sweepcircles. *Computer-Aided Design*, 45(2):483–493, 2013.