Fast Parallel Triangulation Algorithms of Large Data Sets in E2 and E3 for In-Core and Out-Core Processing,
ICCSA 2014 conference, 978-3-319-09128-0, pp.301-314, Springer, 2014

Fast Parallel Triangulation Algorithm of Large Data Sets in E^2 and E^3 for In-Core and Out-Core Memory Processing

Michal Smolik[1], Vaclav Skala[1]

[1] Faculty of Applied Sciences, University of West Bohemia,
22 Univerzitni, 30614 Pilsen, Czech Republic

Abstract. A triangulation of points in E^2, or a tetrahedronization of points in E^3, is used in many applications. It is not necessary to fulfill the Delaunay criteria in all cases. For large data (more then $5 \cdot 10^7$ points), parallel methods are used for the purpose of decreasing time complexity. A new approach for fast and effective parallel CPU and GPU triangulation, or tetrahedronization, of large data sets in E^2 or E^3, is proposed in this paper. Experimental results show that the triangulation/tetrahedralization, is close to the Delaunay triangulation/tetrahedralization. It also demonstrates the applicability of the method presented in applications.

1 Introduction

Today's applications need to process large data sets using several processors with shared memory, i.e. in parallel processing, or/and on systems using distributed processing. In this paper we describe an approach applicable for effective triangulation in E^2 and E^3 (tetrahedralization) using CPU and/or GPU parallel or distributed systems, e.g. on computational clusters, for large data sets.

Many algorithms for triangulation in E^2 and E^3 have been developed and described with different criteria [1], [2], [4], [5]; mostly Delaunay triangulation in E^2 is used due to the duality with the Voronoi diagrams. The Delaunay triangulation maximizes the minimum angle; on the other hand, it does not minimize the maximum angle, which is required in some fields, like CAD systems etc. Moreover, if the points form a squared mesh, algorithms are sensitive to the numerical precision of computation. It is well known that the Delaunay triangulation (DT) contains $O\left(N^{\lceil d/2 \rceil}\right)$ simplicities where d is dimensionality. The computational complexity of the DT is $(N^{\lceil d/2 \rceil+1})$, i.e. for $d = 2$ is $O(N^2)$ and for $d = 3$ is $O(N^3)$.

1.1 Motivation

However, in many cases we do not need exact Delaunay triangulation nor another specific triangulation, as triangulation "close enough" to the required type is

acceptable. Weakening this strict requirement enables us to formulate a simple algorithm based on "divide and conquer (D&C)" strategy and the approach is independent from the triangulation property requirements.

There are the following critical issues to be solved if triangulation is to be applicable for large data sets:

- how to store data so as to especially have fast access on parallel/distributed system,
- how the triangulation is made on a data subset – we expect that each processor will process the given data subset resulting in a triangulated subset,
- how to join triangulated subsets in order to get the final large triangulation in E^2 or E^3.

Of course, implementation on CPU should be simple and implementation on GPU should be simple as well.

2 Proposed Algorithm

In this section, we will introduce a new fast parallel triangulation algorithm in E^2 and E^3. The main idea of this algorithm is to divide all input points into several subsets, perform a triangulation in each of them and then join them together.

First, in sections 2.1-6, we will introduce the proposed algorithm for parallel triangulation. In section 2.7, we will show how to divide data between multiple GPUs and/or cluster PCs. Finally, in section 2.8, we will propose an approach for large data processing.

2.1 Points Division

The approach proposed is based on D&C strategy and therefore input data set has to be split to several subsets. In our case, we will use rectangular grid of size $n \times m$ domains in E^2 (see Fig. 1), resp. $n \times m \times p$ domains in E^3. The grid does not have to be necessarily regular and we can adjust it according to the properties of the input data set. However, we will use orthogonal grid in our approach: it is not necessary because domains can be triangular or tetrahedral, etc.

In the case when a domain does not contain any point, we have to generate a random one and place it into this domain. This restriction is necessary because of the joining procedure which will be introduced later.

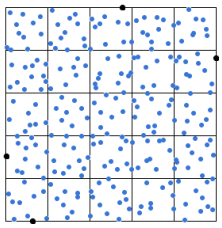


Fig. 1. Division of points into a rectangular grid.

The virtual corner points of the grid are included in the domains. It means that now each data subset contains the original points plus the virtual corner points of the appropriate domain.

2.2 Domains Triangulation

Now, each domain can be triangulated using any triangulation library. Properties of the final triangulation will depend on which triangulation will be used. It should be noted that in some applications, it is inappropriate to use DT, as some other triangulations are more appropriate.

Each domain contains added virtual points. This is a great advantage because the convex hull of domain triangulation will only contain these virtual points (see Fig. 2).

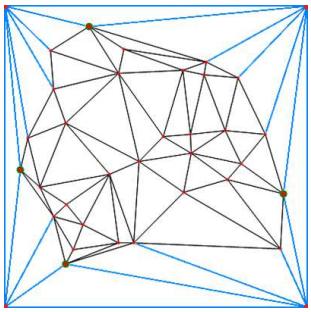


Fig. 2. Domain triangulation (in E^2).

In case of using triangulation library that constructs triangulation with incremental insertion, we do not have to create initial big triangle/tetrahedron. We can directly construct triangles/tetrahedra from virtual corner points.

It should be noted that domain triangulations are totally independent and thus can be done in parallel. We have $n \times m$ independent processes in E^2, resp. $n \times m \times p$ in E^3.

2.3 Domains Joining

After domains triangulation, we have $n \times m$ triangulations in E^2, or $n \times m \times p$ in E^3, and we have to join them to only one triangulation. The process of joining two domains triangulations is very simple. We only have to swap common edge EF to edge AB (see Fig. 3).

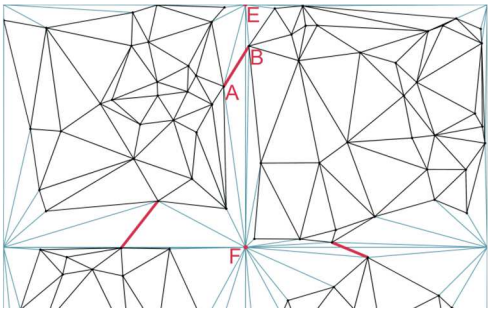


Fig. 3. Joining triangulated domains by edges $EF \rightarrow AB$ swapping.

Situation in E^3 is identical to in E^2. Two domains share one common side with vertices E, F, G and H, and thus we only have to swap edges EG and FH to edge AB (see Fig. 4). It can be seen that the connection of triangulated subsets is extremely simple in the E^2 case. In the E^3 case the situation is straightforward and not complicated as well.

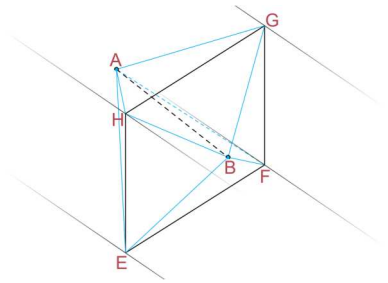


Fig. 4. Joining tetrahedralized domains by edges EG & $FH \rightarrow AB$ swapping.

Joining two domains is totally independent from joining another two domains. Therefore, joining of all triangulations to one triangulation can be done in parallel without any conflicts.

2.4 Removing or Retaining of Virtual Corner Points

If the triangulation is used for scalar potential field in E^2 or E^3, or $2\&1/2D$ applications in GIS systems, the value in the virtual corner points can be approximated from the neighbors using Radial Basis Function Interpolation (RBF) [7]. Virtual corner points can be retained in the triangulation and thereby the triangulation is done. Otherwise the corner points have to be removed.

If the corner points have to be removed, there are several algorithms to manage deletion of vertices from triangulation/tetrahedralization [8], [3]. Simply removing a vertex together with its incident simplices leaves a star-shaped hole in the triangulation, which is not necessarily convex. This approach will be described in the

next subsection. Another approach is to move the vertex towards its nearest neighbor in several steps; each followed by a sequence of flips restoring the triangulation until the simplices between the two vertices are very flat and can be clipped out of the triangulation [6]. This approach will be described in the second subsection.

The process of removing one virtual corner point from triangulation is totally independent from removing any other virtual corner point. Thus removing of virtual corner points in the middle part of triangulation can be done totally in parallel.

Star-shape Polygon Re-Triangulation. This algorithm removes a vertex from the triangulation and thus creates a star-shape hole (polygon/polyhedron) which has to be re-triangulated. The polygon can be divided into several parts (see Fig. 5). We have one center part and four "arms" in E^2, resp. six "arms" in E^3.

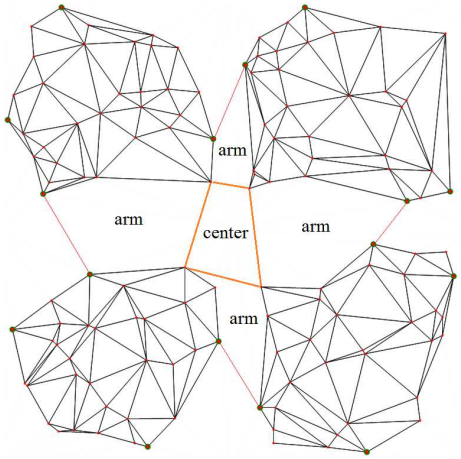


Fig. 5. Star-shape polygon (hole in triangulation).

The center part of the star-shape polygon contains the closest vertex from each surrounding domains. However, the number of vertices is usually four, or eight in E^3; more vertices can be included, e.g. the situation in Fig. 6. The center polygon can be triangulated using ear clipping algorithm, which is of computational complexity $O(N^2)$, but the number of vertices N is very small.

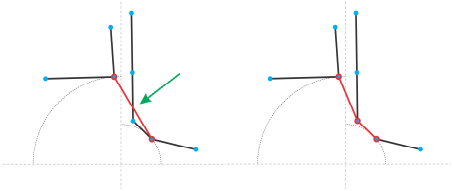


Fig. 6. Intersection of two edges (left) and the solution (right) in E^2.

The arms of the star-shape polygon are monotone polygons in respect to axis x or y, resp. x, y or z. Monotone polygon can be triangulated in $O(N)$ time and thus triangulation of the star-shape hole is a really fast process.

Moving and Deleting Vertex. This algorithm moves the virtual corner point towards its nearest neighbor vertex in triangulation ($A \rightarrow A'$) [6]. The main question is how far a vertex v can be moved into a certain direction without invalidating the triangulation, i.e. without creating overlapping simplices. We can define the pseudo-orientation of a simplex $S = (A, B, C)$, resp. $S = (A, B, C, D)$, as follows:

$$v = \begin{vmatrix} A_x - B_x & B_x - C_x \\ A_y - B_y & B_y - C_y \end{vmatrix}, \text{ resp.}$$
$$v = \begin{vmatrix} A_x - B_x & B_x - C_x & B_x - D_x \\ A_y - B_y & B_y - C_y & B_y - D_y \\ A_z - B_z & B_z - C_z & B_z - D_z \end{vmatrix}. \quad (1)$$

Now suppose one of the vertices is moved along the direction of Δ, i.e. $A \rightarrow A' = A + \lambda\Delta$ with $\lambda \in \langle 0; 1 \rangle$. The maximum size of λ is the minimum value of all λ for all simplices incident the moving vertex A. λ is calculated using the formula:

$$\lambda = \frac{|v|}{\text{abs}\begin{vmatrix} \Delta_x & B_x - C_x \\ \Delta_y & B_y - C_y \end{vmatrix}}, \text{ resp.}$$
$$\lambda = \frac{|v|}{\text{abs}\begin{vmatrix} \Delta_x & B_x - C_x & B_x - D_x \\ \Delta_y & B_y - C_y & B_y - D_y \\ \Delta_z & B_z - C_z & B_z - D_z \end{vmatrix}}. \quad (2)$$

If $\lambda \geq 1$, then the vertex can simply be moved along the complete path Δ, whereas if $\lambda < 1$, the vertex A can only be moved by a fraction $\lambda\Delta$ and the triangulation has to be validated using a sequence of flips. After triangulation validation, we have to recalculate parameters λ and repeat the algorithm until vertex A is equal to A'.

2.5 Removing of Extra Inserted Points

Some domains did not contain any vertex and thus an extra vertex was inserted into such each domain. Now the vertices inserted have to be removed from triangulation. This situation is the same as when removing virtual corner points. Algorithms for removing extra inserted vertices were presented in the previous section.

2.6 Convex Hull Creation

The union of all simplices forms a convex hull. To create a convex hull of triangulation, we have to remove all virtual corner points at the border of the created grid. The vertices have to be removed and preserve the convex hull. There exist many algorithms how to do it. One of them is the ear clipping algorithm. We remove all simplices containing one virtual corner point and then re-triangulate the border. Another way how to do it is to use the approach presented in section "Moving and Deleting a Vertex".

The process of triangulation from input vertices is done after removing all remaining virtual grid points.

2.7 Multiple GPUs or PCs

Today's applications need to process data sets in a short time. Therefore we may use several processors with shared memory, i.e. in parallel processing, or/and on systems using distributed processing, or/and systems using multiple GPUs.

When using several PCs, or/and GPUs, we have to find out how to divide the work and how to join results into one triangulation. Triangulations of domains are totally independent so there is no problem with work distribution. Joining of domains triangulations is, again, totally independent. In the case of retaining virtual corner points in final triangulation, there is no challenge in work distribution between PCs, or/and GPUs. Otherwise in the case of removing virtual corner points, we have to distribute work between PCs, or/and GPUs, according to Fig. 7. Both GPUs need triangulations of yellow domains for removing virtual corner points.

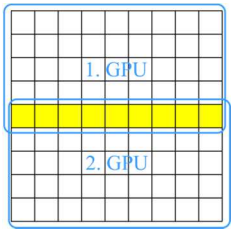


Fig. 7. Distribution of domains per GPU.

Division of work between more PCs, or/and GPUs is no problem and can be easily implemented. Triangulation time can be easily reduced while using more PCs, or/and GPUs.

2.8 Large Data Processing

However there are many algorithms for triangulation/tetrahedralization and only a few of them can be used for large data processing. The main problem is available memory, which is usually no more than tens of gigabytes. The number of points

which can be triangulated/tetrahedralized, is limited by the available memory.

The approach proposed does not have this restriction on the maximal number of points. We can triangulate large data sets which cannot fit at once into the available memory. For one domain triangulation, we do not need any information about other domains. The situation in joining is almost the same. We only need information about two domains which will be joined. And finally, when removing one virtual corner point, we only need information about domains which contain this virtual corner point, i.e. four domains in E^2, or eight domains in E^3.

The input data set can be processed by parts. We can load input data only for some domains, perform parallel triangulation according the approach proposed, and save resulting triangulation/tetrahedralization, in a file. Then we can load data for following domains and perform the same operations. This is a very small change in the approach proposed and is easy to implement. Using this approach, we are able to perform a triangulation/tetrahedralization, on large input data sets with more than 10^7 vertices. The most important feature is that we are not restricted by the limited size of the maximal available memory.

3 Implementation

We implemented the approach proposed in C++ with using OpenMP for parallelization and in CUDA for GPU implementation. The implementation of the approach proposed has been fairly simple in both E^2 and E^3.

It is appropriate to save a copy of points into domains rather than only references to points. Then a full advantage of cache memory use can be taken, and speedup your implementation.

4 Experimental Results

The approach proposed has been tested in several criteria. First of all, we tested the optimal number of points per domain for the purpose of low time requirements. In the second part, we tested time performance of triangulation/tetrahedralization, for a different number of input points. After that, we tested the quality of triangulation/tetrahedralization. Finally, we tested our approach on both synthetic and real data sets.

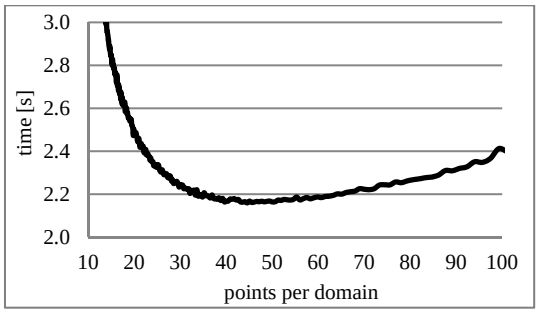
The approach proposed has been tested on data sets using PC with the following configuration:

- CPU: Intel(R) Core(TM) i7 920 (4 × 2,67GHz) with 8 HyperThreads,
- GPU: 2 × GeForce GTX 295
  - 30 multiprocessors × 8 CUDA Cores per multiprocessor 1,38GHz
  - memory 896MB 1,05GHz
- memory: 12 GB RAM,
- operating system Microsoft Windows 7 64bits

4.1 Number of Points per Domain

The first part in the approach proposed is the division of all vertices into a grid. We need to know what the average number of points per domain is. According to that, we can compute parameters n and m, or n, m and p, to split input vertices into $n \times m$ domains in E^2, or $n \times m \times p$ domains in E^3.

We measured time complexity of triangulation/tetrahedralization, for different numbers of input vertices with uniform distribution and different numbers of points per domain. One example of the time measured for 10^7 points and a different number of points per domain can be seen in Graph 1. It can be seen that with an increasing number of points per domain time complexity decreases. This happens up to an optimal number of points per domain where the time complexity is minimal. From this number of points, the time complexity increases with an increasing number of points per domain.



Graph 1. Number of points per domain for 10^7 points (in E^2) with uniform distribution.

An optimal number of points per grid depends on the exact implementation of triangulation/tetrahedralization, which is used for domains triangulation. The next factor is the number of threads used during parallel triangulation. In our case, we used eight hyper-threads and two different implementation of triangulation/tetrahedralization. In the case of using a brutal-force implementation, the optimal number of points per domain is 45 in E^2, or 171 in E^3. In the case of using an optimized implementation, the optimal number of points per domain is 2 000 in E^2, resp. 400 in E^3.

4.2 Time Performance

In some applications, time performance is one of an important criterion. We measured running times for triangulation/tetrahedralization, for different numbers of points with uniform distribution. Running times were measured for 8 threads running and for only 1 thread. The times of 1 thread running, have been compared with running times of

Fast Parallel Triangulation Algorithms of Large Data Sets in E2 and E3 for In-Core and Out-Core Processing,
ICCSA 2014 conference, 978-3-319-09128-0, pp.301-314, Springer, 2014

publicly available serial library for triangulation called Fade[1], or serial library for tetrahedralization called TetGen[2].

Triangulation. Tab. 1 presents running times of triangulationon on CPU. Running times of triangulation on GPU in comparison with running times of publicly available GPU library GPU DT[3] can be seen in Tab. 2.

Table 1. Running times of triangulations in E^2 (using CPU).

	8 threads (4 cores)	1 thread		
Number of points	Parallel triangulation	Parallel triangulation	Fade library	
	Time [s]	Time [s]	Time [s]	
316 227	0.06	0.20	0.27	
1 000 000	0.18	0.67	0.88	
3 162 277	0.65	2.23	2.96	
10 000 000	2.16	7.33	9.58	
31 622 776	7.99	24.88	35.66	
100 000 000	28.21	81.94		

The running time for 10^8 points using Fade triangulation library could not be measured because of high memory requirements. However, we do not have time of triangulation for 10^8 points: we can see that the parallel triangulation is always faster, even when using serial execution of our parallel triangulation. The time required for triangulation of 10^8 vertices is $28.21\ [s]$ on CPU.

Table 2. Running times of triangulations in E^2 (using GPU).

Number of points	GPU parallel triangulation	GPU DT library
	Time [s]	Time [s]
1 000	0.010	0.149
3 162	0.012	0.173
10 000	0.015	0.186
31 622	0.019	0.260
100 000	0.034	0.317
316 227	0.093	0.620
1 000 000	0.253	1.625

According to the results from Tab. 2, it can be seen that our GPU triangulation is much faster than publicly available library for GPU triangulation called GPU DT. The time required for triangulation of 10^6 vertices is $0.253\ [s]$ on GPU.

Tetrahedralization. Running times of tetrahedralization in comparison with publicly available serial library TetGen can be seen in Tab. 3.

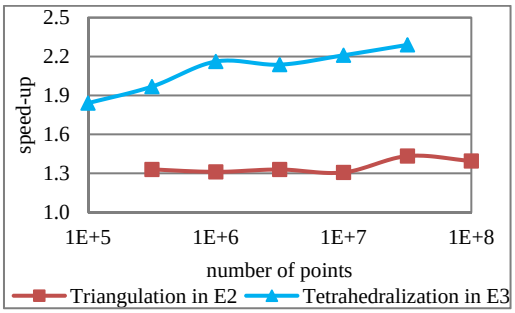
[1] Kornberger, B., Fade2D & Fade2.5D, Geom e.U. Software Development.
[2] Si, H., TetGen: A Quality Tetrahedral Mesh Generator and a 3D Delaunay Triangulator, Weierstrass Institute for Applied Analysis and Stochastics.
[3] GPU-DT: A 2D Delaunay Triangulator using Graphics Hardware, National University of Singapore.

Table 3. Running times of tetrahedralizations in E^3.

Number of points	8 threads (4 cores)	1 thread	
	Parallel tetrahedraliza. Time [s]	Parallel tetrahedraliza. Time [s]	TetGen library Time [s]
100 000	0.29	0.97	1.78
316 227	0.85	2.92	5.75
1 000 000	2.52	8.77	18.97
3 162 277	8.11	28.35	60.60
10 000 000	25.72	88.69	196.00
31 622 776	81.70	278.45	

The running time for $\sqrt{10} \cdot 10^7$ points using TetGen tetrahedralization library could not be measured because of high memory requirements. Although we do not have time of tetrahedralization for $\sqrt{10} \cdot 10^7$ points, we can see that the parallel tetrahedralization is always faster, even when using serial execution of our parallel tetrahedralization, and the speed-up is increasing with the increasing number of input vertices. The time required for tetrahedralization of $\sqrt{10} \cdot 10^7$ vertices is 81.7 $[s]$.

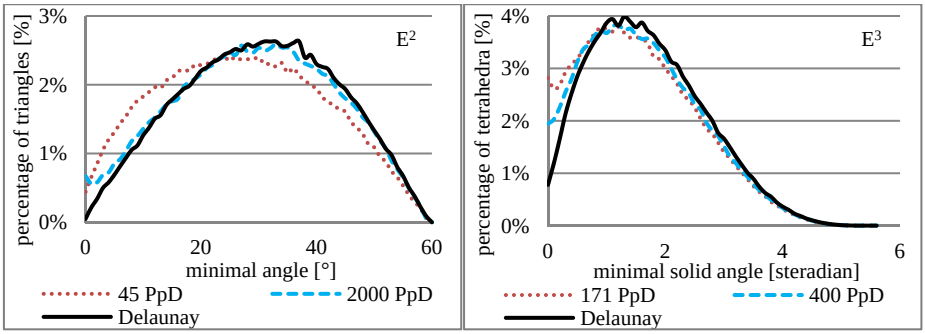
Speed-up. Using Tab. 1 and Tab. 3, we can calculate the speed-up of parallel triangulation/tetrahedralization when using only one thread, in respect to the publicly available serial library for triangulation called Fade, or for tetrahedralization called TetGen, see Graph 2.



Graph 2. Speed-up of parallel triangulation (using 1 thread) to Fade library and speed-up of parallel tetrahedralization (using 1 thread) to TetGen library.

4.3 Triangulation Quality

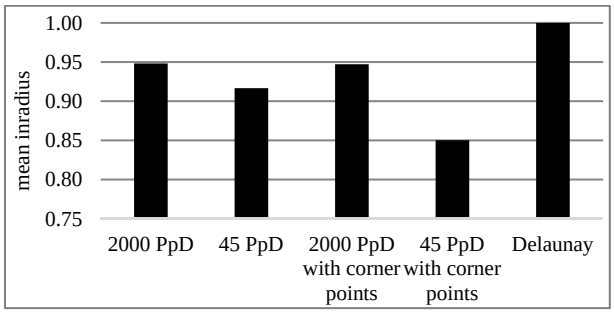
Delaunay triangulation maximizes the minimal internal angle in triangulation. Therefore, one test of triangulation quality is the distribution of minimal internal angles in triangulation. We measured the internal angle in degrees in E^2 and the internal solid angle in steradians in E^3 (see Graph 3).



Graph 3. Distribution of minimal internal angles (PpD = Points per Domain).

According to the results, a triangulation/tetrahedralization created with the algorithm proposed is very close to Delaunay triangulation, see Graph 2. Moreover, the inner parts of the domains are Delaunay's. The more points per domain are used, the closer to the Delaunay triangulation the triangulation proposed is.

The Delaunay triangulation maximizes the mean incircle radii. Due to this criterion, we calculated Graph 4. We can see a similar behavior like in Graph 3. The more points per domain are used, the closer to the Delaunay triangulation our triangulation is. If we retain corner points in triangulation, then the quality of triangulation is a bit worse. However, for 2 000 vertices there is almost no difference in the mean inradius for triangulation with and without virtual corner points.



Graph 4. Mean inradius of triangles for different triangulations (mean inradius of Delaunay triangulation was normalized to size 1.0).

4.4 Synthetic and Real Data Sets

In many applications, we do not need to triangulate only uniformly distributed data sets, but real data sets. An example of real data sets may be sets for geographic information system applications. Triangulation of one such set is shown in Fig. 8. It is

a GIS data set of South America. The set contains $1.1 \cdot 10^6$ points and the triangulation time of our parallel triangulation proposed was $0.42\ [s]$.

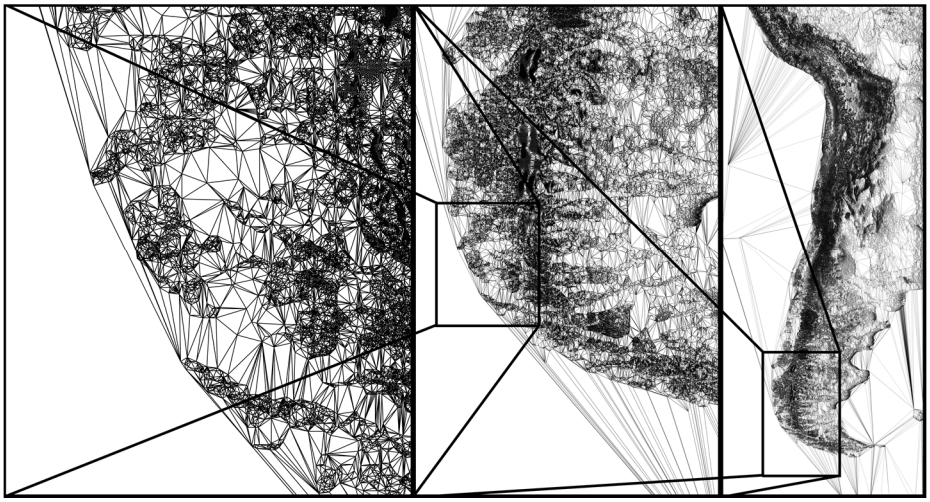
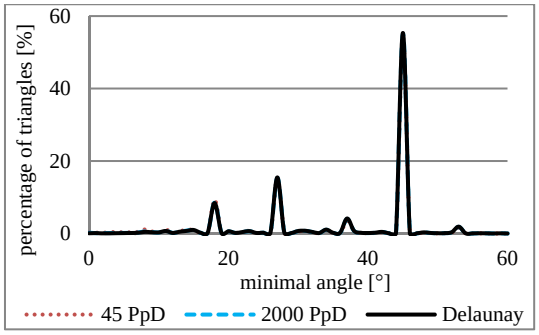


Fig. 8. Triangulation of South Americas GIS data set.

We compared the obtained triangulation from Fig. 8 with the Delaunay triangulation of the same data set. We used the distribution of minimal internal angles in triangulation for comparison (see Graph 5). It can be seen that there is almost no difference and all graphs are overlapped over each other. The highest maxima are for angles $45°$ (edges of the triangle in the proportion of $1:1:\sqrt{2}$), $26.6°$ (edges of the triangle in the proportion of $1:2:\sqrt{5}$) and $18.4°$ (edges of the triangle in the proportion of $1:3:\sqrt{10}$).



Graph 5. Distribution of minimal internal angle (PpD = Points per Domain).

5 Conclusion

A new fast parallel triangulation algorithm in E^2 and E^3 has been presented. It is based on the "Divide & Conquer" strategy. It can be easily implemented on parallel environments with shared and/or distributed memory using both CPU and GPU. As it is scalable; the proposed algorithm is especially convenient for large data sets processing. The approach proposed has been implemented and tested using both CPU and GPU. An additional speed-up can be expected if the data structures are carefully implemented for the given HW.

Acknowledgments. The authors would like to thank their colleagues at the University of West Bohemia, Plzen, for their comments and suggestions, and anonymous reviewers for their valuable comments and hints provided. The research was supported by MSMT CR projects LG13047, LH12181 and SGS 2013-029.

References

1. Chen, M.-B.: A Parallel 3D Delaunay Triangulation Method. 9th ISPA, IEEE, 2011, pp.52-56.
2. Cignoni, P., Montani, C., Scopigno, R.: DeWall: A Fast Divide & Conquer Delaunay Triangulation Algorithm in Ed. Computer Aided Design, 1998, Vol.30, No.5, pp.333-341.
3. Ledoux, H., Gold, Ch. -M., Baciu, G.: Flipping to Robustly Delete a Vertex in a Delaunay Tetrahedralization. Computational Science and Its Applications – ICCSA, 2005, Vol. 3480, pp. 737-747.
4. Liu, Y.-X., Snoeying, J.: A Comparison of Five Implementations 3D Delaunay Tessellations. Combinatorial and Computational Geometry, MSRI publ., 2005, Vol.52, pp.439-458.
5. Rong, G.D., Tan, T.S., Cao, T.-T.: Computing Two-dimensional Delaunay Triangulation Using Graphics Hardware. ACM Symposium on Interactive 3D Graphics and Games, Redwood City, CA, USA, 2008, pp. 89-9.
6. Schaller, G., Meyer-Hermann, M.: Kinetic and Dynamic Delaunay Tetrahedralization in Three Dimensions. Computer Physics Communications, 2004, Vol.162, No.1, pp.9-23.
7. Skala, V.: Radial Basis Functions for High Dimensional Visualization. VisGra - ICONS12, Saint Gilles, Reunion Island, IARIA, 2012, ISBN: 978-1-61208-184-7, pp. 218-222.
8. Zemek, M., Kolingerova, I.: Hybrid Algorithm for Deletion of a Point in Regular and Delaunay Triangulation. ICCSA, 2009, ISBN: 978-1-4503-0769-7, pp. 137-144.