ACUT: Out-Of-Core Delaunay Triangulation of Large Terrain Data Sets

Josef Kohout, Ivana Kolingerová

Centre of Computer Graphics and Data Visualization, University of West Bohemia, Plzeň, Czech Republic
Email: {besoft, kolinger}@kiv.zcu.cz

Abstract

In the last couple of years, very detailed high-resolution terrain data sets have become available thanks to new acquisition techniques, e.g., the airborne laser scanning. Such data sets contain, typically, several millions of points and, therefore, several gigabytes are required just to store them, which disallows their loading into the memory of a common computer. In this paper, we propose a novel out-of-core technique for construction of the Delaunay triangulation of such large data sets. It is based on the method of incremental insertion with flipping that is simple, robust and can be easily extended for weights of points, constraints, etc. The proposed technique was tested on various data sets with sizes up to 128M points on a commodity hardware (P4 3.2GHz, 2GB RAM, 250GB SATA disk). The largest data set was processed in about 2.5 hours.

1 Introduction

Given a point set S, the Delaunay triangulation [16] in \( \mathbb{R}^2 \) is a triangulation which satisfies the Delaunay criterion for each triangle: the circum-circle of the triangle does not contain any input point \( p \in S \) in its interior. One of the most important properties of the Delaunay triangulation is that it maximizes the minimal angle and, therefore, it contains the most equiangular triangles of all triangulations (i.e., it limits the number of too narrow triangles that may cause problems in further processing). Due to its properties [21], the Delaunay triangulation is used in many research areas, e.g., in terrain modeling (GIS) [22], scientific data visualization and interpolation [17], [5], [36], pattern recognition [38], meshing for finite element methods (FEM) [15], [8], [34], natural sciences [33], [2], computer graphics and multimedia [17], [35], etc.

In the past, many researchers focused on the problem how to compute the Delaunay triangulation in a reasonable time. The pressure was put on the exploitation of sophisticated data structures to speed up the computation, e.g., Directed Acyclic Graph [9], hierarchical tree [19], multi level uniform grid [40], skip lists [39], etc. Many parallel solutions have been also developed; most of them, e.g., [13], [30] were designed for specialized parallel architectures, often with hundreds of processors. Parallel algorithms suitable for low-cost clusters of workstations also exist, e.g., [14], [29], [25].

Modern computer architectures allow us to compute the Delaunay triangulation of data sets with thousands of points by any of existing sequential algorithms in a reasonable time. The problem is that current data sets are, due to the progress in acquisition techniques, much larger, e.g., typical data sets produced by airborne laser scanning contain several millions of uniformly distributed points, and they cannot be, therefore, loaded into the physical memory of a common computer - either because the computer is a 32-bits architecture that is able to address up to 4GB of memory only, or simply because it is not equipped by the requested amount of memory. Majority of popular sequential or parallel algorithms thus fail to process these data sets.

Hardwick [25] describes a parallel algorithm based on the divide-and-conquer strategy. The input points are subdivided recursively into two groups according to the median of their x or y-coordinate. Points are projected onto the perpendicular plane going through the median, the lower convex hull of projected points is found using quickhull algorithm and the back projection of the convex hull gives a set of Delaunay edges that separates the input region with the input points into two non-convex sub-regions. Both sub-regions are simultaneously triangulated by Dwyer’s algorithm [20] and the Delaunay triangulation is obtained by a union of both local triangulations. Similar approach was also proposed by Lee et al. [30]. As both approaches need to keep all points in the memory for the computation
of Delaunay separators, they are not suitable for the processing of large data sets. This drawback, however, can be easily removed.

Another parallel approach was proposed by Chrisochoides et al. [14]. It starts by a construction of a coarse triangulation of a subset of points. The created triangles are partitioned into k continuous regions and distributed over k processors. After that, the processors insert points lying in their regions using the Watson’s approach [37]. The triangles constructed on boundaries are redistributed heuristically in order to balance the load of the processors. The algorithm is again not suitable for large data sets because every processor needs all points but its extension seems to be possible.

Recently, Blandford et al. [10] proposed, in our opinion, a very complex parallel algorithm based also on the Watson’s approach. It exploits a special data structure that maintains the triangulation in a compressed format in the memory and dynamically decompresses a small region of this triangulation whenever a point has to be inserted into this region. The authors were able to process a data set with billions of points on parallel architecture with 64 processors (each processor had 4GB RAM).

Chen et al. [13] proposed a parallel algorithm that distributes points according to their coordinates over k processors and let processors to triangulate their points. After that, each processor computes an “interface” using an incremental construction approach, i.e., a set of triangles that crosses the area boundaries. For this operation, the processor must have available all points lying in the areas adjacent to the area assigned to this processor. In the final step, interfaces are merged together in order to obtain the resulting triangulation. As the processor does not need the whole input set, the algorithm is able to triangulate huge data sets.

Kohout at al. [28] proposed an application independent software layer that supports processing of large data sets by the simulation of the shared memory on a cluster of workstations. The layer provides universal routines for the manipulation with data, no matter whether the data is stored locally or remotely. It was used for the construction of Delaunay triangulation by a parallel approach described in [27]. Due to an intensive network communication required to fetch the remote simplices, this solution is not, however, effective.

Parallel processing is not the only way how to handle large data sets. Recent research has been also focused on external memory algorithms (acknowledged also as out-of-core algorithms) that use disks for temporary storage of data structures that are too large to fit in the memory and load them into the memory when necessary. Many theoretical papers discussing the optimal strategy to minimize data movement have been published, e.g., [1], [4], etc. A good survey of these approaches can be found in [6]. Surprisingly only a few practical papers dealing with the construction of Delaunay triangulation of large data sets exist.

Agarwal et al. [3] designed an out-of-core algorithm for the construction of Constrained Delaunay triangulation based on the divide-and-conquer approach. It sorts all points in such a way that they lie on a space-filling Hilbert curve and splits them into subsets that are successively triangulated. Whenever the triangulation of a subset is completed, the algorithm checks each of the remaining points if it does not violate the Delaunay criterion and if the outcome is positive, the triangulation is altered by an appropriate way. Although the authors claim that their algorithm is practical, we think that it is quite difficult to be understood and implemented. Nevertheless, the proposed algorithm is quite efficient. According to the results, it was able to compute the Delaunay triangulation of a data set with several millions of points in a couple of minutes.

A data streaming approach described by Isenburg et al. [26] constructs the Delaunay triangulation in two steps. In the first step, the input stream with points is read (several times) and points are partitioned into buckets written into an output stream. In the second pass, this output stream is read and the Delaunay triangulation is successively constructed using the Watson’s approach. When all points from one bucket are processed, the constructed triangles that will not change in the future are removed from the memory (and their points as well) and written into the output stream. The algorithm is easy to be understood but its implementation may be more difficult as many singular cases must be handled. Nevertheless, it is very efficient: a data set with billion points can be processed in several hours on a common computer.

In this paper, we propose an out-of-core algorithm called ACUT (Area CUTting) that is suitable for the construction of Delaunay triangulation of large \( \mathbb{E}^2 \) data sets. It is very simple to be understood
as well as implemented. It is based on the method of incremental insertion with local transformations (i.e., edge flipping) [23], thus it offers better robustness than algorithms based on other approaches. As far as we know, it is the first algorithm based on this approach. It does not need to sort points in preprocessing (but it performs a spatial reorganization of points), is insensitive on distribution of points and can be generalized to incorporate constraints given in the form of prescribed edges, to use non-Euclidian metrics or weights of points, etc.

The paper is structured as follow. An overview of our approach (called ACUT) is given in Section 2; Sections 3 and 4 describe its steps in detail. Section 5 brings results of the performed experiments and a comparison with existing approaches. Section 6 concludes the paper.

2 Overview of ACUT

The proposed approach for the construction of Delaunay triangulation of large data sets is based on the obvious idea to split the input data into several smaller sets, compute local triangulation for these sets and merge these triangulations together.

As many existing solutions, our approach has two main steps. In the first one, the points from the input file are reorganized into cells of a uniform grid and stored into a temporary grid file. In the second step, these cells (and their points, indeed) are partitioned into the requested number of regions and an appropriate star shaped domain with Delaunay edges on its boundary is computed for every region - see Figure 1. Domains are processed successively: their points, i.e., points lying inside the domain (or on the boundary) are triangulated and the triangulation is stored into the output file. As domain boundaries are formed by Delaunay edges, no complex merge phase is required, all that is needed is to update the connectivity between triangles of adjacent domains, if necessary. The schematic view of the proposed approach is shown in Figure 2.

The most complex part is the construction of domains. Let us describe it in more detail. It starts with a construction of the convex hull of input points. The computed convex hull is the domain appropriate to the initial region that represents the whole grid of points. This region (and its domain) is recursively subdivided by horizontal and vertical cuts using an approach by Mueller [32]. More information about it is given in the following section. Whenever a cut for the region is found, the technique proposed by Blelloch et al. [11] is used to construct a poly-line of Delaunay edges along the cutting edge (details are described in the Section 3). This poly-line is combined together with separators forming the boundary of the appropriate domain to create two new domains. The recursion stops when the requested number of domains is reached.

3 Details of ACUT

In this section, we describe both steps of our approach (see the previous section) in detail.

3.1 Grid Construction

In this first step of our approach, the input points are subdivided into cells of a uniform grid that covers the min-max box of points and has \( \sqrt{c \cdot \sqrt{N} \cdot \frac{H}{W}} \) cells, where \( N \) is the number of input points, \( H \) and \( W \) are height and width of the
min-max box of the input points and \( c \) is a constant (we use 0.5). For each cell of the grid, a small point buffer is created in the memory. Its capacity depends on the number of points to be processed, number of cells in the grid and on the amount of memory available for this first step. The input points are successively read and inserted into the corresponding buffer according to their coordinates. If the buffer is full, its points are written at the end of a temporary grid file and the position of the written block in the file is enlisted in a list of fragments stored in the cell structure. When the entire input is read, all buffers are flushed. At the end, the grid structure, i.e., the matrix of lists of fragments is also written into the grid file.

The problem is that very often we do not know the min-max box of points in advance and sometimes even the precise number of points is also unknown. To avoid multiple reading of input file, we propose a data-streaming algorithm that works as follows. At the beginning, a chunk of \( M (M < N) \) points is loaded into the memory, the overall number of points in the file is estimated from the size of input file and the size of currently loaded number of points, the min-max box of the loaded points is computed, an initial uniform grid is created using the formula written above and points are subdivided into the grid. After that another chunk of points is loaded, their min-max box is computed, the current grid is enlarged by adding some rows and/or columns, if necessary, and points are again subdivided into cells. If the grid dimension is larger than some given threshold, pairs of cells are merged together. Let us point out that this merge stage only concatenates lists of fragments, no point is moved. So it goes until all points are processed.

An advantage of this partitioning is that it subdivides input points into cells in one pass. On the other hand, points lying in one cell are very often fragmented into blocks (especially, if cell buffers are too small), which slows down the processing due to an inefficient use of spatial coherence.

### 3.2 Cells Partition

The points have to be subdivided into \( k \) subsets in such a manner that not only the almost equal number of points in each subset is ensured but also the bounding boxes of these subsets have minimal intersection and the total length of boundaries is minimal. To achieve this, we use the strategy proposed by Mueller [32] that is based on a recursive subdivision of the summed-area table using horizontal and vertical cuts. The summed-area table is constructed from a matrix of values corresponding to numbers of points lying in appropriate cells of a uniform grid covering the bounding box of all points - see Figure 3. Let us note that we already created this grid in the first step of our approach. An advantage of the Mueller approach is that the summed-area table can be efficiently found in \( O(R) \), where \( R \) is the total number of cells and split into \( k \) regions in \( O(k \cdot \log(R)) \) using a binary search algorithm. Detailed description is out of the scope of this paper.

![Figure 3: The subdivision of cells into three regions.](image)

### 3.3 Construction of Delaunay Separators

When, in the Mueller algorithm (see the previous section), a cut for the current region is computed, we construct the convex hull of points transformed by the following formulas:

\[
P(P_x, P_y) \rightarrow P'(P_y - C_y, \|P - C\|^2),\]

if the cut is the vertical one and

\[
P(P_x, P_y) \rightarrow P'(P_x - C_x, \|P - C\|^2),\]

otherwise, where \( C \) is the centre of the cut. Unlike Hardwick et al., not all points are processed but only points lying in cells covered by the domain appropriate to the region to be cut. The lower part of the constructed convex hull is taken and its edges give the Delaunay separators between corresponding untransformed points - see Figure 4.

All that remains is to combine the constructed poly-line of Delaunay separators with the domain separators and to create two new domains. Starting from the first point from the poly-line, we search in the chain of vertices of the domain polygon to find this point. If the corresponding point is not found,
the next point is taken and the search restarts. So it goes until the match is found. The other end of the poly-line is also processed, the domain polygon is split into two poly-lines at the positions of matches and they are connected to the appropriate part of the constructed poly-line to form two new domains. The complexity of this brute-force combination is $O(\sqrt{N})$ in the worst-case. The performance could be improved by using a hash table - in our current implementation, we do not use it. Figure 5 shows the domains constructed for a real data set.

3.4 Construction of Convex Hull

As there is not enough memory to load every point into the memory, we use an incremental construction algorithm that works as follows. Starting with an initial convex hull, points are successively tested whether they lie outside the current convex hull. If the result of the test is positive, the convex hull has to be updated in such a manner that the point lying outside belongs now to the new convex hull.

In our implementation, the location is speeded up by red-black trees [7]; one is used for the lower part of the convex hull, another one for the upper part. An internal node stores a pair of key $x$ and an associated value $p$, where $p$ is a point on the convex hull and $x$ is its x-coordinate. An external node represents an edge or an empty half-space. For each given point $q$, we search its x-coordinate in both trees to find either vertex $v$ or an edge $e$ and we test whether the given point lies above or below the found primitive - see Figure 6a. If the point lies outside the current convex hull, it is inserted into the appropriate red-black tree and points that no longer belong to the convex hull are removed from this tree - see Figure 6b. Let us note that the convex hull can be computed by this algorithm in $O(N \cdot \log(M))$ expected time, where $M$ is the number of points on the convex hull (it is usually much less than $N$).

3.5 Domain Triangulation

Domain points are triangulated using the method of incremental insertion with flipping. We decided to
use this method because of its simplicity and robustness: in the case of an incorrect or inconsistent Delaunay criterion evaluation caused by numerical inaccuracy, a triangulation with two or more non-Delaunay triangles is obtained, but it is still a valid triangulation. The method can be also simply modified to incorporate constraints given in the form of prescribed edges, to use non-Euclidian metrics or weights of points, etc.

Starting with an auxiliary triangle containing all domain points in its interior, points are inserted successively into the triangulation as follows. The triangle containing the point to be inserted is located (we use the remembering stochastic walking [18]) and subdivided. Afterwards, the empty circum-sphere criterion is tested recursively on all triangles adjacent to the new ones, and if necessary, their edges are flipped, i.e., local transformation are applied. Figure 7 shows an example of the insertion.

Figure 7: The insertion of point into the DT.

In our approach, points on the boundary of the domain are inserted first and the remaining domain points (they must lie in some cell of the corresponding rectangular region [11]) are afterwards inserted in a pseudorandom order as follows. While the insertion order of points from one cell is randomized, cells are ordered according to the space-filling Hilbert curve [12] and all points from one cell must be processed before the algorithm may advance to another cell - see Figure 8. This pseudorandom order of insertion has two important features. First, it exploits the spatial coherence in data and, therefore, the walking needs \( O(1) \) in the expected case to locate the triangle to be subdivided. Next, the randomness lowers the sensitivity of the approach to numerical errors.

In our implementation, an array is used as a compact and efficient data structure to hold the triangulation. Let us note that as planar triangulations of \( N \) points have at most \( 2N \) triangles, the array can be allocated at the beginning to hold this amount.

When all points are processed, every outer triangle, i.e., the triangle that lies outside the domain, has to be removed and the remaining triangles must be stored into the output file. The removal starts with searching for any triangle that has a vertex of the auxiliary initial triangle. Usually, only a few triangles have to be checked (the worst-case we have experienced was 1% of triangles). The reference on this triangle is pushed into a stack and the following steps are repeated until the stack is empty. The triangle is popped from the stack and every adjacent triangle not sharing edge that belongs to the Delaunay separators is pushed into the stack. A hash table is used to speed-up this test. The triangle is marked then as processed and disconnected from the mesh.

4 Singular Cases

The proposed approach, as it was described in previous sections, may not work perfectly for all data sets. In this section, we discuss singular cases that must be handled in order to have a robust solution.

First problem roots in the ambiguity of Delaunay triangulation for data sets containing four points lying on a common circle - see Figure 9. As this quadrilateral, say \( p_a, p_b, p_c, p_d \), may be triangulated by two different ways, it may happen that while the diagonal \( p_b, p_d \) is chosen during the construction of Delaunay separators, the triangulator creates the other diagonal, i.e., the edge \( p_a, p_c \). Due to this inconsistency, we can walk from an outer triangle to an inner triangle without crossing any Delaunay separator, which leads to that no triangle is stored in the extraction stage.

Fortunately, this problem can be easily solved by adding an additional test to the extraction algorithm.
When the triangle is popped, we check whether all of its vertices belong to a set of vertices forming the Delaunay separators (this can be done quickly using another hash table). If the outcome of this test is negative, i.e., the triangle contains some inner domain point, the appropriate adjacent triangle forming the quadrilateral is found and their common edge is swapped.

The second problem is caused by a numerical inaccuracy during the construction of convex hull. It may happen that a point that should be on the convex hull is due to round-off errors located inside the convex hull, which leads to the construction of incorrect Delaunay separators. The result is that this point lies outside the domain into which it is assigned - see Figure 10. Therefore, its incident triangles are removed during the extraction, which means that this point will not be triangulated in the final Delaunay triangulation.

This problem is automatically detected when the algorithm handles the problem of ambiguity and is unable to find an appropriate adjacent triangle to perform the swap edge. Unfortunately, its solving is extremely difficult (if not impossible). Indeed, we could redistribute these points assigned to incorrect domain, but their detection would harm the performance significantly. As we detected, in our experiments, up to 5 such cases per one domain (35 in total for a data set with 64M points) when single float precision was used, we cease to correct this problem, thus allowing a small amount of unconnected points in the resulting mesh. Let us note that we have not experienced this problem when we switched to double precision.

5 Experiments & Results

The proposed approach was implemented in C++ using Microsoft Visual Studio.NET 2005. Loading/storing is written generally (e.g., it can load data into various structures from various file formats), which has a significant negative impact on the performance. Also some profiling is included in the code, e.g., counting time spent in important routines. It, indeed, slows down the computation. Single precision arithmetics was used in the code.

We tested the implemented solution on various data sets (generated and real) in a binary format on Dell Optiplex GX620: Intel Pentium Processor 3.2 GHz with Hyper-Threading Technology and EM64, 2GB Dual Channel DDR2 RAM (but we used only 1GB), 250GB SATA Disk, MS Windows XP. For each data size several different data sets were tested and the presented times were calculated as average of measured total times. Let us note that in the total time everything is included, i.e., the time needed for loading of points, for the grid construction, convex hull computation, triangulation and for the storing of the output mesh.

Figure 11 shows dependency of the total time on the number of points for generated data sets with various point distributions: uniform, gauss, clusters and for real data sets (mostly GTOPO30 [24]). As it can be seen, the proposed approach is insensitive to point distributions. Moreover, it can process data sets in almost linear time.

A runtime profiling for tested uniform data sets is given in Figure 12. If we consider larger data sets only, i.e., 16M+, more than 50% of the overall time is consumed by I/O operations (grid construction - 10–20%, points loading - up to 10%, domains construction 30%). The time required for the domains construction, i.e., for the cells partition and the construction of Delaunay separators, grows proportionally to the number of points. For a 128M data
Figure 11: The runtime for data sets with various point distributions.

set, when 32 domains were used, ACUT needed as much time for this operation as for the triangulation itself. Let us note that the domains construction also performs some I/O operations.

Figure 12: The profiling for uniform data sets.

We compared our results with the results achieved by Agarwal et al. [3]. According to published graphs, their approach is about three times faster. However, this comparison is skewed because Agarwal et al. do not include the time consumed by the required sorting of input points into the published total time needed for the Delaunay triangulation. They also used a more powerful computer for their experiments: Intel Pentium XEON 2.4 GHz with Hyper-Threading Technology, 1GB RAM (but only 128 MB was used), 4x72 GB SCSI (10000 rpm) disks in RAID-0 configuration running Linux with kernel 2.4.5-smp. Therefore, we daresay that both approaches are more or less competitive in the performance, however, our approach is, in our opinion, much easier to be implemented.

We also compared the results with the results by Isenburg et al. [26]. Although authors claim that their approach is 12 times faster than the approach by Agarwal et al, the experiments that we performed with their software on our data sets show that for large uniform data sets our approach outpaces theirs - see Figure 13. For real data sets, however, Isenburg et al. achieved much better performance (about 4 times for 20M) - see Figure 14. The reason for this behavior is that points in tested real data sets are already ordered according to their coordinates, while points for uniform data sets are unordered. Let us note that this comparison is, however, imprecise because of two following reasons. First, unlike our approach, Isenburg et al. do not store the connectivity between triangles, which is, indeed, unacceptable for many applications. According to our additional experiments, the triangulation extraction (in ACUT) runs much faster if we do not store the connectivity (about 12 times for a 16M uniform data set, which speed ups the overall process 2 times). On the other hand, for experiments with the software by Isenburg et al., the input was given in text format, which is, indeed, quite inefficient format (our approach requires about 70% more time to process text files than to process binary files, e.g., 74.2% for a 16M uniform data set). Let us, therefore, conclude the comparison by the claiming that both approaches are more or less competitive, each has its pros and cons.

Figure 13: The runtime comparison of our and Isenburg’s approach [26] for uniform data sets.

6 Conclusion

In this paper, we have proposed an out-of-core approach for the construction of Delaunay triangulation in $\mathbb{R}^2$ based on the incremental insertion with local transformations, which, as far as we know, has not been used for the processing of large data sets yet. The approach is easy to implement and robust.
Figure 14: The runtime comparison of our and Isenburg’s approach [26] for real data sets.

It can be also generalized to incorporate constraints given by a set of prescribed edges into the triangulation or to use weights of points. All that is needed is to modify the transformation of points (see Section 3.4) using the idea proposed by Maur et al. [31]. It transforms constrained edges (or non Delaunay edges introduced by given weights of points) onto a lower convex hull and to modify the Delaunay criterion test in such a way that constrained edges are always considered valid, i.e., they are never flipped.

The proposed approach was implemented in C++ and tested on various data sets with up to 128 millions of points. According to the our experiments, our solution processes data sets in an almost linear time (a 128M uniform data set was processed in 2.5 hours on a common hardware) and, due to its simplicity, insensitivity to point distribution and generalization possibilities, it is, in our opinion, an interesting alternative to existing more efficient approaches by Agarwal et al. [3] and Isenburg et al. [26].

Acknowledgment

This work was supported by the Ministry of Education of The Czech Republic - project LC 06008. We would also like to thank Prof. V. Skala from the University of West Bohemia for providing conditions in which this work has been possible.

References


