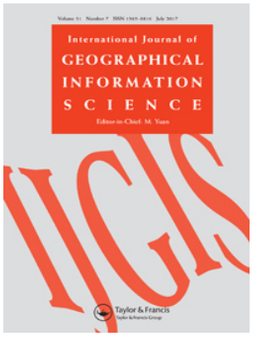

International Journal of Geographical Information Science

ISSN: 1365-8816 (Print) 1362-3087 (Online) Journal homepage: http://www.tandfonline.com/loi/tgis20

Parallel constrained Delaunay triangulation on the GPU

Narcís Coll & Marité Guerrieri

To cite this article: Narcís Coll & Marité Guerrieri (2017) Parallel constrained Delaunay triangulation on the GPU, International Journal of Geographical Information Science, 31:7, 1467-1484, DOI: 10.1080/13658816.2017.1300804

To link to this article: https://doi.org/10.1080/13658816.2017.1300804

View supplementary material

Published online: 13 Mar 2017.

Submit your article to this journal

Article views: 237

View related articles

View Crossmark data

Full Terms & Conditions of access and use can be found at
http://www.tandfonline.com/action/journalInformation?journalCode=tgis20

INTERNATIONAL JOURNAL OF GEOGRAPHICAL INFORMATION SCIENCE, 2017
VOL. 31, NO. 7, 1467–1484
http://dx.doi.org/10.1080/13658816.2017.1300804

Taylor & Francis
Taylor & Francis Group

RESEARCH ARTICLE

Check for updates

Parallel constrained Delaunay triangulation on the GPU

Narcís Coll and Marité Guerrieri

Geometry and Graphics Group, Universitat de Girona, Girona, Spain

ABSTRACT

In this paper, we propose a new graphics processing unit (GPU) method able to compute the 2D constrained Delaunay triangulation (CDT) of a planar straight-line graph consisting of points and segments. All existing methods compute the Delaunay triangulation of the given point set, insert all the segments, and then finally transform the resulting triangulation into the CDT. To the contrary, our novel approach simultaneously inserts points and segments into the triangulation, taking special care to avoid conflicts during retriangulations due to concurrent insertion of points or concurrent edge flips. Our implementation using the Compute Unified Device Architecture programming model on NVIDIA GPUs improves, in terms of running time, the best known GPU-based approach to the CDT problem.

ARTICLE HISTORY
Received 3 November 2015
Accepted 25 February 2017

KEYWORDS
Constrained Delaunay triangulation; parallel computing; GPU

1. Introduction

The Delaunay triangulation (DT) of a set of planar points is a triangulation such that the circumcircle of any triangle does not contain any other point, or equivalently, every edge is locally Delaunay, that is, the two triangles sharing it have no vertex in each other's circumcircles. A planar straight-line graph (PSLG) is a collection of points and non-crossing segments. The constrained Delaunay triangulation (CDT) of a PSLG is a triangulation of its points such that every segment is an edge of the triangulation and the remaining edges are locally Delaunay.

The DT and the CDT are fundamental topics in computational geometry and they have a great variety of applications in geographic information systems that require generating triangulated irregular network models. Although these triangulations can be computed in $O(n \log n)$ time, where n is the number of input points, they still consume a great deal of time especially for current applications that often need to work with millions of points. In such cases, parallel algorithms are well suited to accelerate their computation. To this end, several approaches have been studied by a number of researchers in recent years. However, most of them only achieve a significant speed-up over uniform distributions of the input points because they are based on a balanced distribution of the points among the processors.

CONTACT Narcís Coll coll@imae.udg.edu
The supplemental data for this article can be accessed here
© 2017 Informa UK Limited, trading as Taylor & Francis Group

Graphics processing units (GPUs) are specialized processors which use a highly parallel structure that makes them perfect for solving problems that can be partitioned into independent and smaller parts. The development of Compute Unified Device Architecture (CUDA) and some programming languages such as OCL (Open Computing Language) make GPUs attractive for solving problems in a parallel way.

In the CUDA architecture, a code to be executed on the GPU is called a *kernel*. A kernel is executed in parallel by several blocks of threads. Each thread computation is performed by a CUDA core. A thread has its own local memory and can access the device's global memory or communicate with other threads from the same block through the shared memory. Each thread has an identifier that it uses to compute memory addresses and make control decisions.

In this paper, we propose a novel GPU-based approach for computing the DT of a set of points or the CDT of a PSLG. Our algorithm is easy to implement under the CUDA architecture, works well for any type of distribution of points and segments, and is competitive in terms of running time.

2. Related work

Here, we review the most relevant work for constructing in parallel the DT and the CDT in 2D.

2.1. *Delaunay triangulation*

A number of methods (Cignoni *et al*. 1993, Lee *et al*. 1997, Blelloch *et al*. 1999, Chen *et al*. 2006, Wu *et al*. 2011, 2014) for parallel 2D DT use the divide and conquer strategy to construct partial DTs in subregions, and finally merge these partial triangulations to obtain a result. However, these kinds of algorithms have some drawbacks. First, the merge phase is quite complex because it involves building the faces connecting the subregions and identifying the faces in the subregions that have to be changed to satisfy the Delaunay criterion. In the worst case, these corrections affect the whole triangulation. Second, the merge phase is done by just one processor and thus limiting the efficiency of the algorithm. In Lo (2012), a divide-and-conquer method that does not require a merging phase is presented. However, the output of this method is not a triangulation of the whole region, but rather just a set of triangles.

Parallel 2D DT algorithms avoiding the divide-and-conquer strategy (Chrisochoides and Sukup 1996, Chrisochoides and Nave 2003, Antonopoulos et al. 2009a, 2009b) are inspired by Bowyer–Watson's algorithm to insert the points. Firstly, these algorithms create a coarse triangulation of a subset of points. Secondly, the triangles are partitioned into areas and assigned to processors and then the parallel insertion begins. Synchronizations across area boundaries are necessary.

In Kohout *et al*. (2005), a randomized insertion is used as a base for the parallel algorithm. The triangulation is structured in a directed acyclic graph (DAG) that stores the history of its changes. This algorithm consists of three phases: the location, the subdivision, and the legalization, where the edges are forced to satisfy the Delaunay criterion. Each processor needs to access the DAG structure along all three phases of the algorithm. Then, implementing some synchronization mechanism between the

processors is required to prevent simultaneous accesses to the same DAG node, which may create artefacts in the resulting triangulation.

There are methods (Fischer and Gotsman 2006, Rong *et al*. 2008) based on computing with the GPU a discrete Voronoi diagram of the points, storing it in a texture, and using it to derive the DT. The drawback to these methods is that the accuracy of the triangulation strongly depends on the resolution of the texture.

In Navarro *et al*. (2011), a GPU algorithm based on edge flipping (a not locally Delaunay edge is changed by the edge that connects its opposite vertices) is proposed to transform any 2D triangulation of a set of points into the DT of them.

2.2. *Constrained Delaunay triangulation*

The CDT of a PSLG can be constructed in the following way: begin with an arbitrary triangulation of the PSLG points, next examine each PSLG segment in turn to see if it is an edge, next force each missing PSLG segment to be an edge, and finally flip not locally Delaunay edges until all edges are locally Delaunay with the provision that PSLG segments cannot be flipped. There are two ways to force a PSLG segment to be an edge of the triangulation. The first consists of deleting all the edges it crosses, then inserting the segment and retriangulating the two resulting polygons (one from each side of the segment). The second approach consists of iteratively flipping the edges that cross the segment until the segment is an edge. Qi *et al*. (2013) presented a GPU method for computing the CDT of a PSLG. This method has three phases. The first phase computes the DT of the PSLG points, while the second inserts the PSLG segments into the triangulation using edge flipping, and the third flips the remaining not locally Delaunay edges. Experimental results show that the method achieves a significant speed-up with respect to the best CPU methods (Shewchuk 1996). However, since the DT is constructed using a discrete Voronoi diagram computed on a uniform grid, the method needs huge memory space to allocate the grid and is less efficient when the distribution of the input points is far from being uniform.

3. Our approach

Our proposal uses the same strategies for both triangulations, DT and CDT. Then, from now on, we will assume that we want to compute the CDT of a PSLG with the provision that the set of segments could be empty. Our algorithm is based on an iterative process that finishes when all PSLG elements (points and segments) are inserted into the DT and no edge needs to be flipped. In each iteration as many elements as possible are inserted and as many edges as possible are flipped with the restrictions that only one point can be inserted into one single triangle and only one edge per triangle can be flipped. Each iteration is divided into four steps:

- **Location**: The triangle containing every non-inserted point is determined by a walking process.
- **Insertion**: At most one point per each triangle is inserted into the triangulation.
- **Marking**: Some edges of the triangulation are either marked to be a segment or marked to be flipped because they are crossed by a segment or they are not locally Delaunay.
- **Flipping**: At most one marked edge per each triangle is flipped.

Each step is performed by independent kernel functions that run in parallel. In Algorithm 1 we show the pseudocode of the process.

Algorithm 1 PCDT

```
Initialization;
repeat
  Location:
  for every not inserted point in parallel do
    determine the triangle containing the point by walking through the
    triangulation;
  end for
  Insertion:
  for every triangle containing a point in parallel do
    insert the point into the triangulation;
    mark the edges of the triangle as candidates to be not locally Delaunay;
  end for
  Marking:
  for every not inserted segment whose endpoints have been inserted in paral-
  lel do
    if the segment is an edge then
      mark the edge as a segment;
    else
      starting from the first endpoint launch a walking process until an edge
      crossed by the segment is detected as a candidate to be flipped;
      mark the edge to be flipped;
      mark each traversed triangle with the index of the segment;
      do the same starting from the second endpoint;
    end if
  end for
  for every triangle not crossed by a segment in parallel do
    mark the not locally Delaunay edges of the triangle to be flipped;
  Flipping:
  for every triangle in parallel do
    determine at most one edge per triangle to be flipped;
    if the edge exists then
      flip the edge;
      mark the edges of the quadrilateral determined by the union of the two
      triangles sharing the edge as candidates to be not locally Delaunay;
    end if
  end for
until all points and all segments have been inserted and no edge can be flipped
```

To perform each step correctly, the key point to take into consideration is that the threads have to coordinate their activity in order to maintain a consistent triangulation. The following example illustrates a possible problem: suppose that during the insertion step a thread representing a triangle t is responsible for inserting a point in it. Each newly generated triangle has a neighbour that may be a neighbour of t or a descendant of a neighbour. Then, since it is not possible to know a priori if the neighbour has been processed, is being processed or will be processed by another thread, without a coordination between threads it will not be possible to assign the correct neighbour to the newly created triangle. To avoid this kind of problem, we have adopted the following strategy to coordinate the threads. We have subdivided the insertion of a point in a triangle and the flip of an edge into two parts. In the first part, each thread representing a triangle t '*informs*' each of the neighbours of t if t will remain being their neighbour or, if not, who the new neighbour of each neighbour will be once the step has been finished. In the second part, the triangles are effectively updated according to the information gathered in the first part. Therefore, proper data structures have to be used to perform each step correctly and use the GPU's resources as efficiently as possible.

Next, we explain in detail the data structures, some notation, and each step of the algorithm.

3.1. *Notation and data structures*

Let n be the number of points and m be the number of segments. The CDT is initialized with a large auxiliary triangle that contains all the points. Then, the points are indexed by p from 0 to $n + 2$ (0, 1, and 2 being the indices of the vertices of the auxiliary triangle), the segments are indexed by s from 0 to $m-1$, and the triangles are indexed by t from 0 to $2n$ (0 being the index of the auxiliary triangle). Moreover, the following arrays allocated in the global memory are used to handle points, segments, and triangles:

- **Points**. Three arrays to store:
- the position (x,y) of the point,
- a binary flag to determine whether the point has been inserted or not,
- and the index of the triangle in which the point is contained.

The first three positions of these arrays correspond to the three vertices of the large auxiliary triangle that contains all the points. Moreover, initially all the points are contained in the large auxiliary triangle. Then, for each point, the triangle index is initialized with 0. This index is updated in each location step if the point has not already been inserted into the triangulation. Once the point has been inserted, the triangle index is updated in the insertion and flipping steps and it is used to determine the initial triangle of any straight walk in the triangulation starting at the point.

- **Segments**. Two arrays to record:
- the indices of the endpoints of the segment,
- and a binary flag to establish whether the segment has been inserted or not.
- **Triangles**. Seven arrays to register:

- the indices of the three vertices of the triangle,
- the indices of the three triangles that are the current neighbours of the triangle,
- the indices of the three triangles, called *future neighbours*, that will be the neighbours of the triangle after executing a point insertion or a flip in a current neighbour,
- the index of a point to be inserted in the triangle,
- the index of a segment intersecting the triangle,
- the labels assigned to the three edges of the triangle to discern if they need to be flipped or not,
- and the index of the next edge to be flipped in the triangle.

Here, we remark that the three vertices of a triangle are indexed with 0, 1, and 2 in a counterclockwise order. The edges and neighbours of a triangle are also indexed with 0, 1, and 2 in such a way the edge j connects the vertex j with the vertex $j+1 \bmod 3$ and the neighbour j shares the edge j. Moreover, it is important to establish priorities over the edges marked for flipping. For instance, when a point lies on an edge, its insertion into the triangulation creates a degenerate triangle. This problem can be solved if the longest edge of the degenerate triangle is immediately flipped. Moreover, if a triangle is crossed by at least one segment, no edge of it has to be flipped for being not locally Delaunay. Accordingly, we use one of the following labels and priorities for each edge of each triangle:

- S – Priority 0. The edge is a segment.
- D – Priority 0. The edge is locally Delaunay and not crossed by a segment.
- X – Priority 0. The edge is crossed by a segment but it does not have to be flipped.
- ND – Priority 1. The edge is candidate to be not locally Delaunay and not crossed by a segment.
- X – Priority 2. The edge is crossed by a segment and it has to be flipped.
- P – Priority 3. A point lies on the edge.

Observe, then, that an edge is marked to be flipped only when its flipping priority is greater than zero. Moreover, we use the flags null, 0, 1, 2, and n to record which edge of a triangle is to be flipped. If the flag is null that means no edge has to be flipped, if the flag is 0, 1, or 2 then the triangle is responsible for flipping the corresponding edge, if the flag is n this means that an edge will be flipped but the triangle in charge of the flip is a neighbour.

3.2. *Location step*

The objective of this step is twofold, on the one hand, to determine for each non-inserted point the triangle of the current triangulation containing the point and, on the other, to select for each triangle the best point to be inserted in it. Each point P of index p records the index t of a triangle T in which the point should be contained. This index is initialized with the index 0 of the auxiliary triangle for all the points, but this triangle index needs to be updated after the insertion and flipping steps. For example, if the triangle T was adjacent to a triangle T' of index t' and the common edge between both

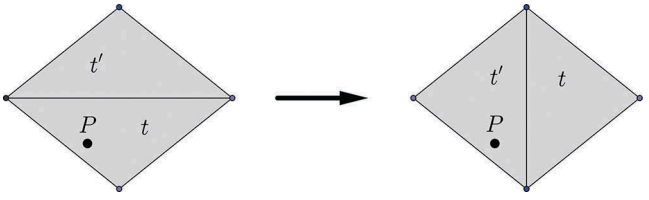

Figure 1. A point P may lie within another triangle after an edge flip.

triangles was flipped, the point P may lie within the new triangle of index t' (see Figure 1). Likewise, if the triangle T was subdivided into three new triangles due to the insertion of a point in it, point P may lie within a triangle with index different from t.

So, in this step, each thread represents a point of index p and, in case that point P has not yet been inserted into the triangulation, updates the triangle index t that is currently associated to p with the index t' of the triangle of the current triangulation that contains P. It works as follows: a straight walk in the triangulation starting from the centroid C of T is launched until the triangle T' that contains P is reached (see Figure 2). If P lies on an edge shared by two triangles, T' is taken between both as that with the lowest triangle index. Then, t is updated with the index t' of T'.

As explained in Section 3.1, each triangle records an index of a point contained in it which will be inserted in the triangle in a following insertion step. Then, the point index p' associated to t' should also be updated. For example, if p' was updated for any point reaching T', this index would be updated simultaneously by distinct processors and, in this manner, it would register the last point reaching T'. However, this is not a good strategy to follow if one wants to minimize the running time of the succeeding location steps. Let us study the mean number of triangles visited by a straight walk. The probability of detecting a convex set domain through a linear search is studied in Santaló (1976) (p. 103). Given a convex domain K_0 within which lies another convex domain K, the probability of intersecting K by a segment S of length s included in K_0 is

$$\frac{2\pi A + 2sL}{m(S \subset K_0)},$$

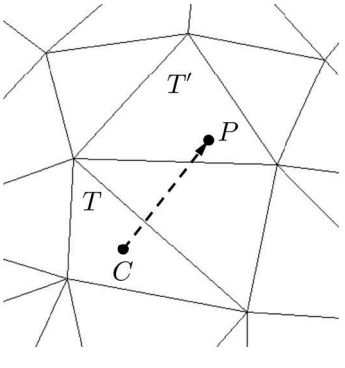

Figure 2. Straight walk in the triangulation for locating the triangle containing a point.

where A and L are, respectively, the area and the perimeter of K and $m(S \subset K_0)$ is the measure of all segments of length s included in K_0. Consequently, given a triangle T_0 of area A_0 partitioned into a set of k triangles T_i with area A_i and perimeter L_i, the mean number of these triangles that are intersected by a segment S of length s included in T_0 is

$$\frac{2\pi \sum_{i=1}^{k} A_i + 2s \sum_{i=1}^{k} L_i}{m(S \subset T_0)} = \frac{2\pi A_0 + 2s \sum_{i=1}^{k} L_i}{m(S \subset T_0)}.$$

Thus, the running time of the straight walk for point P is proportional to the number of triangles intersected by the segment CP, and the mean value of this number is proportional to the length $\| CP \|$ and to the sum of the edge lengths of the triangulation. So, the way to accelerate the location step is, on the one hand, to make the segment CP as small as possible by updating the triangle index t and, on the other, to choose, among all the points contained in T', the point that minimizes its total distance to the vertices of T' for insertion in T' . That is, to choose for insertion the point closer to the Fermat point of T'. However, the Fermat point has two inconveniences. First, it is expensive to compute and second, it can lie on a vertex of the triangle. Consequently, the insertion of a point closer to the Fermat point could create strongly degenerate triangles which would cause numerical problems. Then, instead of the Fermat point, we use the centroid of the triangle that, in fact, minimizes the sum of squares of the distances from a point to the three vertices of the triangle. Accordingly, we have adopted the following strategy for updating p'. Let P' be the point of index p' associated to t' and C' be the centroid of T'. Then, p' is updated with p if the distance between P and C' is less than the distance between P' and C'. In the Results section, we numerically analyse the goodness of this strategy.

3.3. *Insertion step*

In this step, each thread manages a triangle index t and performs the insertion into the triangulation of the point of index p associated to it as follows. Let p_i $(i = 0, 1, 2)$ be the point indices of the vertices of T. Then, the triangle T will be triangulated to the triangle t with vertex indices p_0, p_1, and p; the triangle $2p + 1$ with vertex indices p_1, p_2, and p; and the triangle $2p + 2$ with vertex indices p_2, p_0 and p (see Figure 3).

As pointed out at the beginning of Section 3, when a triangle is newly created or updated, its neighbours have to be known. However, since the insertion process is done in parallel, at the time when a triangle is being processed, it is not possible to know if its neighbours were processed, are currently being processed or will be processed. So, without a proper strategy, it is not possible to know the neighbours of the triangles t, $2p + 1$, and $2p + 2$ after the whole insertion process.

Our way to overcome this problem is to subdivide the insertion step into two parts. In the first, for each triangle t with a point of index p to be inserted, the triangle $2p + 1$ is assigned as a future neighbour of the triangle n_1 and the triangle $2p + 2$ is assigned as a future neighbour of the triangle n_2. In addition, if an edge of t contains p then the edge is marked with label P in both triangles sharing this edge in order to have the maximum priority of being selected in the flipping step. Moreover, the edges of t that do not

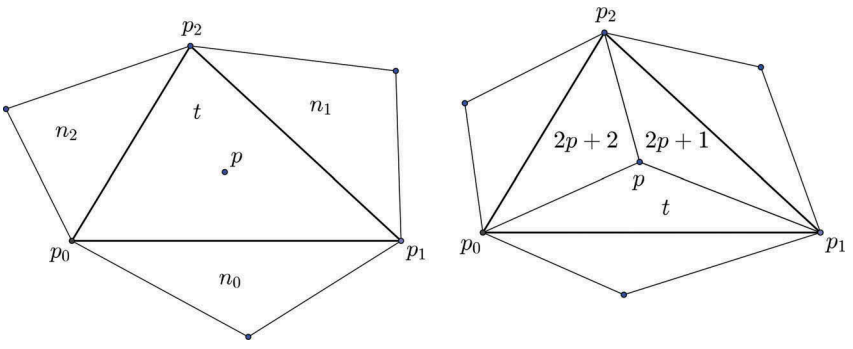

Figure 3. Triangle of index t before and after the insertion of the point of index p in it.

contain p are candidates to be not locally Delaunay after the insertion of p. So, each of these edges is marked as possibly not locally Delaunay with the label ND in both triangles sharing it.

In the second part, for each triangle t, if t has a point p to be inserted, the arrays corresponding to the triangles t, $2p+1$, and $2p+2$ are updated according to the insertion of p in t. Notice that, thanks to the first part, the neighbours of these triangles will have been correctly updated with the information stored in the future neighbours array. Otherwise, if t has no point to be inserted, the neighbours of t are simply updated with the triangles previously stored in the future neighbours array. Moreover, if the triangle index of p_2 is t, this triangle index is updated with $2p+1$.

3.4. *Marking step*

All existing methods for computing a CDT compute first the DT of the given point set, then insert all the segments, and then finally transform the resulting triangulation into the desired CDT. Following this procedure, Qi *et al.* (2013) presented a parallel method for inserting segments based on edge flipping. They showed that, if all the flippable edges crossed by a segment were candidates for flipping, an infinite loop would be formed. Then, to avoid such loops, they established the criterion to determine the edges crossed by a segment that can be flipped. The criterion is stated as follows. Let $C = (T_0, T_1, \ldots, T_k)$ be the ordered sequence of triangles crossed by a segment S. Let e_i $(i = 0, \ldots, k-1)$ be the edge of T_i sharing T_{i+1}. The edges e_0 and e_{k-1} are candidates if the quadrilaterals $T_0 \cup T_1$ and $T_{k-1} \cup T_k$ are strictly convex. An edge e_i $(i = 1, \ldots, k-2)$ is a candidate if the quadrilateral $T_i \cup T_{i+1}$ is strictly convex and satisfies at least one of these three not mutually exclusive conditions:

(1) Flipping e_i would result in S not intersecting one of the two resulting triangles (see Figure 4).
(2) The polygon $T_{i-1} \cup T_i \cup T_{i+1}$ is strictly convex (see Figure 5).
(3) The quadrilateral $T_{i-1} \cup T_i$ is concave and flipping e_i would result in the quadrilateral $T_{i-1} \cup T_i'$ being strictly convex (see Figure 6).

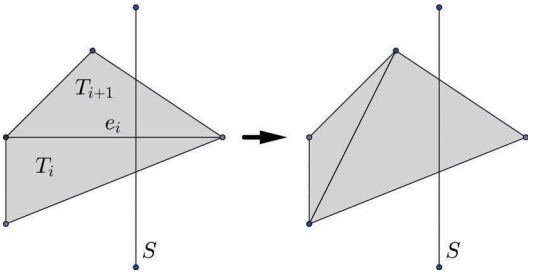

Figure 4. Case 1.

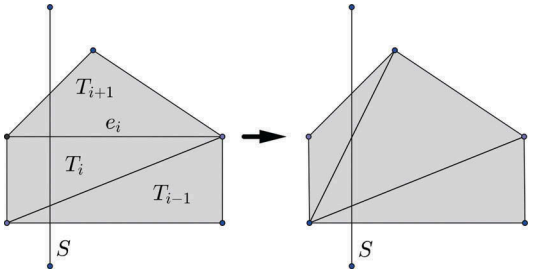

Figure 5. Case 2.

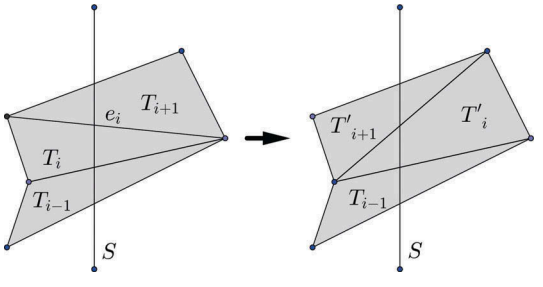

Figure 6. Case 3.

Qi *et al*. (2013) proved that at least one edge crossed by a segment is a candidate to be flipped, and they also noted that, if no further measures were taken, the flipping phase would easily fall into an infinite loop because two different segments may intersect some common triangles. An example of this situation is illustrated in Figure 7. In the first step, the edge depicted in red is selected for flipping by the segment S_2. While in the second step, its flipped edge is selected by the segment S_1, which drives it into an infinite loop.

To avoid these kinds of problems, Qi *et al*. (2013) proposed marking each triangle with the segment of lowest index crossing it, and only allowing to flip edges whose adjacent triangles received the same mark. By doing this, they proved that in each segment insertion iteration at least one segment is inserted into the triangulation.

Our approach differs from all the existing methods in that we simultaneously insert points and segments into the triangulation. Then, the goal of this step is to mark edges

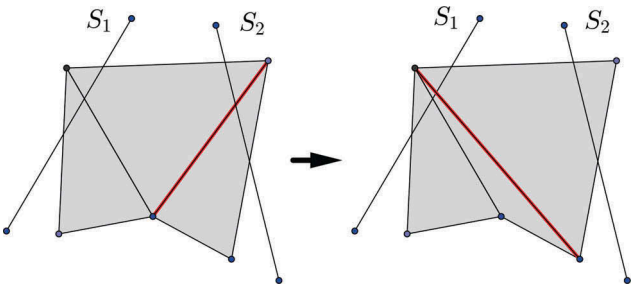

Figure 7. Loop.

as constrained because they correspond to PSLG segments, or to mark edges to be flipped because they are crossed by segments or they are not locally Delaunay. Accordingly, this step is subdivided into two parts: part one for the segments and part two for the not locally Delaunay edges.

In part one, each thread handles a segment S of index s and only works for segments that have not yet been inserted into the triangulation, although both endpoints have. If the endpoints are connected by an edge, the edge is marked as constrained (label S) in both triangles sharing it. If not, at most two edges crossed by S are marked to be flipped as follows. A straight walk starting from the first endpoint of S is launched until an edge e_i satisfying the criterion adopted by Qi *et al*. (2013) is found. Each traversed triangle T_k ($k = 0, \ldots, i+1$) is marked with s if no edge of T_k contains a point (no edge label equals to P) and s is lower than the segment index previously stored in T_k. Then, if the triangles T_i and T_{i+1} are currently marked with s, the edge e_i is marked to be flipped (edge label updated with label X-Priority 2) in both triangles T_i and T_{i+1}. Furthermore, an analogous process takes place starting from the second endpoint.

Observe that in Qi *et al*. (2013) the segments are inserted in parallel after computing the DT of the whole set of the input PSLG points, and for each segment the complete sequence of crossed triangles is marked. In contrast, in our approach, the process for inserting a segment begins as soon as both endpoints of the segment have been inserted into the triangulation. In addition, in each step of the process only the first and the last candidate edge (there are at least one of them) in the sequence of edges crossed by the segment are marked for flipping. In this way, we avoid workload imbalance among GPU threads due to significant differences in the number of edges crossed by the segments, we also avoid unnecessary Delaunay tests and we ensure that the algorithm finishes because in each flipping step at least one edge crossed by the non-inserted segment with lowest index is flipped. In the Results section, we numerically analyse the goodness of this strategy.

In the second part, each thread handles a triangle T of index t and works only for triangles whose edges do not contain any point. If T is not crossed by any segment, each edge of T marked as a candidate to be not locally Delaunay (edge label equals to ND) is checked to be locally Delaunay. In that case, the edge is relabelled D in order not to be selected for flipping in the next step. Otherwise, if T is crossed by a segment S, each neighbour of T adjacent to an edge marked as a candidate to be flipped (edge label equals to X-Priority 2) is checked to be also crossed by S. If not, the edge is relabelled X-Priority 0 in order not to be selected for flipping in the next step.

3.5. *Flipping step*

During this step, at most one edge of each triangle t is flipped. To avoid conflicts between concurrent flips, this step needs to be subdivided into three stages. In all three stages, each thread handles a triangle of index t.

The first stage corresponds to the selection of, at most, one edge per triangle according to the established priorities over the edges marked for flipping and works as follows. First, the highest flipping priority h between the edges of t is determined. Second, if $h > 0$, that is, t has at least one edge really marked for flipping, let e be an edge of t with priority h such that e is the unique edge in the other triangle t' adjacent to e with priority h. If this edge exists, let nh be the number of edges of t with priority h. Then, if $nh \geq 2 >$ or ($nh = 1 >$ and $t < t'$), e is selected as the edge to be flipped in t. Otherwise, null is stored as the edge to be flipped in t.

In the second stage, if t has an edge e of index j to be flipped, the future neighbours of the triangles adjacent to the quadrilateral determined by the edge are updated according to the future flip of the edge (see Figure 8 where $i = j - 1 \bmod 3$ and $k = j + 1 \bmod 3$). Otherwise, if an edge of t is stored to be flipped in a triangle adjacent to t, the label n is stored as the edge to be flipped in t. Moreover, after flipping an edge, the edges of the quadrilateral containing the edge are candidates to be not locally Delaunay. Consequently, in this stage each of these edges is also marked with label ND in both triangles sharing it.

The third stage works for each triangle t that falls into one of these two cases:

(1) The label j of the edge to be flipped is 0, 1, or 2. Then, the arrays corresponding to the triangle t and to the neighbour triangle t' sharing the edge to be flipped are updated according to the flip (see Figure 8). Moreover, if the triangles t and t' were marked with the index of a segment, this mark is set to null. Finally, the triangle index of p_j is updated with t only if its current value is t', and the triangle index of $p_k >$ is updated with t' only if its current value is t.
(2) The label of the edge to be flipped is null. That is, no edge of t has to be flipped. Then, the neighbours of t are updated with the future neighbours of t.

Again, the key to do this task is the information stored in the second stage in the future neighbours array.

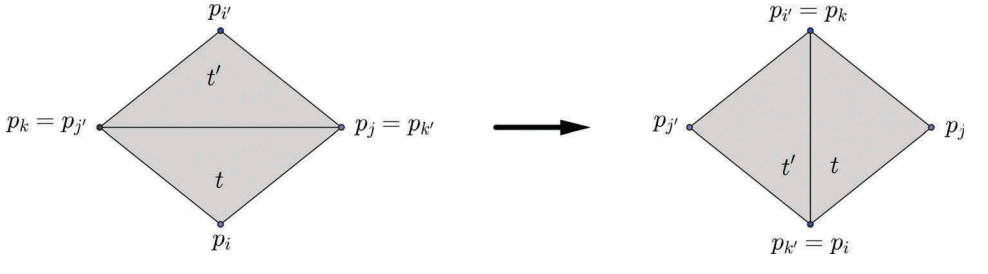

Figure 8. Flip.

4. Results

In this section, we present a number of experimental results on both real and synthetic data sets. They were obtained by implementing our algorithm PCDT in CUDA and the implementation of GPU-DT (Qi *et al*. 2013), also in CUDA, that can be freely downloaded at http://www.comp.nus.edu.sg/~tants/delaunay2DDownload.html. The running times of the algorithms presented in this section are obtained as the median of 10 executions. Both algorithms were tested on a computer equipped with an Intel(R) Pentium(R) D CPU 3.00 GHz, 3.5 GB RAM, and a GPU NVidia GeForce GTX 580. A zip file with a Linux version of our code can be found in the supplementary material of this paper. This file also contains a model that was used in our tests.

We have divided the experimental results into two parts. In the first, we study the practical performance of PCDT in order to evaluate the effectiveness of our strategies. In the second, we compare the performance of our algorithm against GPU-DT.

4.1. *Performance of PCDT*

We tested PCDT against three other variants:

(1) This variant corresponds to the classical approach for computing the CDT of a PSLG that first computes the DT of the given point set, inserts all the segments, and then finally transforms the resulting triangulation into the CDT.
(2) In Section 3.2, we explained two ways for updating the point to insert in a triangle. In each triangle, PCDT inserts the closest point to the centroid of the triangle, while this variant inserts the last point reaching the triangle.
(3) As explained in Section 3.4, for each segment, PCDT marks for flipping at most two edges crossed by the segment and marks only the triangles visited until the two edges are reached. This variant marks all the edges crossed by the segment satisfying the flipping criterion and the complete sequence of the triangles crossed by the segment.

To test the goodness of our final approach we implemented the four strategies, and the four implementations have been applied to the models Australia 1 (see Figure 9), Australia 2 (see Figure 10), and Victoria (see Figure 11). Notice from Table 1 that our approach achieves the best running times over the other three alternatives. Moreover, it

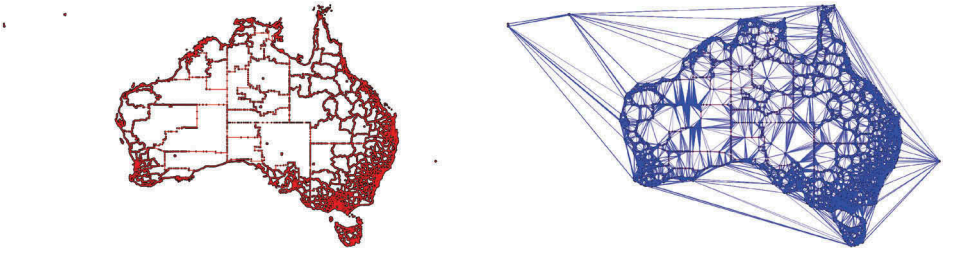

Figure 9. Model Australia 1 (left) and its CDT (right).

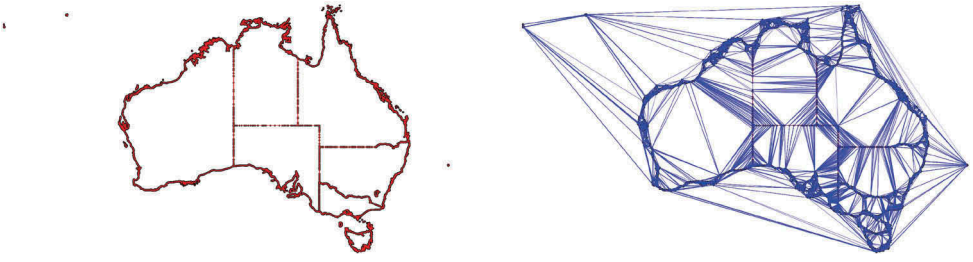

Figure 10. Model Australia 2 (left) and its CDT (right).

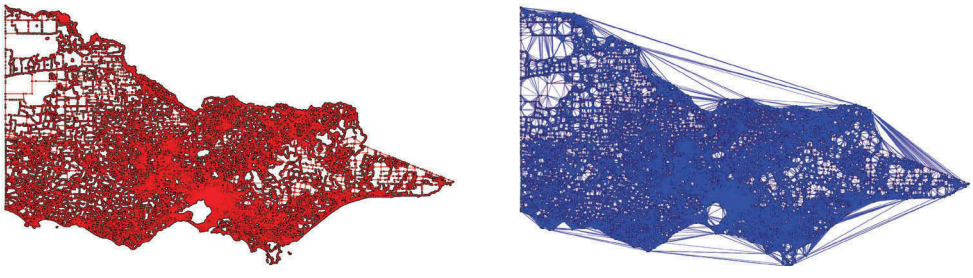

Figure 11. Model Victoria (left) and its CDT (right).

Table 1. Behaviour of the proposed algorithm against other alternatives.

	Australia 1	Australia 2	Victoria
Num. Vertices	4,164,582	1,836,693	3,431,125
Num. Segments	4,162,386	1,836,684	3,349,746
PCDT (ms)	3982	1547	2343
Variant 1 (ms)	4833	1667	3243
Variant 2 (ms)	8696	3855	5998
Variant 3 (ms)	4266	1759	3788

can be also concluded that PCDT achieves significant speed-up versus Variant 2, which proves the effectiveness of the centroid strategy for inserting points in the triangulation.

We have also studied the relationship between running time and number of input segments for a fixed number of points. The results from two different models are depicted in Figure 12 (model Tasmania corresponds to model number 5 in Table 2). From these results, it can be concluded that the performance of our approach depends on the number of input points but it almost never depends on the number of segments. This is because there is not much difference in the number of flips required to transform a triangulation of points into a DT or into a CDT.

4.2. *Comparison with GPU-DT*

We applied our approach and GPU-DT to eight real-life models. These models represent boundaries of states and they can be freely downloaded from the Australian Bureau of Statistics. Australia 1, Australia 2, and Victoria are three of these models. The results are presented in Table 2 and the plot of the mean running times versus the number of

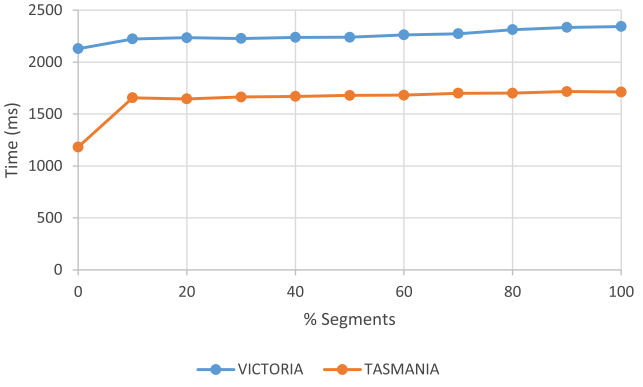

Figure 12. Running time versus number of segments.

Table 2. Behaviour of the proposed algorithm against GPU-DT over real-life data.

	1	2	3	4	5	6	7	8
Num. Vertices	1,048,659	1,244,635	1,725,325	1,836,693	2,083,213	2,739,569	3,431,125	4,164,582
Num. Segments	1,045,426	1,240,732	1,684,815	1,836,684	2,070,237	2,671,693	3,349,746	4,162,386
PCDT (ms)	715	886	1205	1547	1712	2033	2343	3982
GPU-DT (ms)	8486	4592	16,940	23,227	20,150	6209	6131	12,728

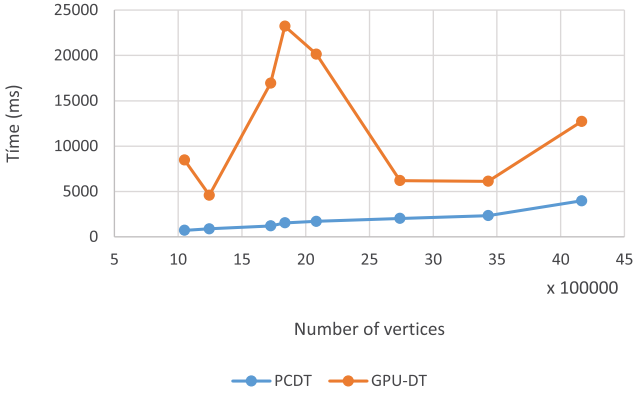

Figure 13. Running time versus the number of vertices.

vertices is depicted in Figure 13. Notice that our approach achieves much better results in all models. Moreover, two features can be observed in Figure 13. First, the behaviour of GPU-DT strongly depends on the distribution of the input points. Note that the worst GPU-DT running time is achieved by model number 4 (Australia 2 in Figure 10) whose input points distribution is far from being uniform. Second, the expected running time of our parallel approach can be considered linear with respect to the size of the input.

We also experimented with synthetic data. To this end, we used three different distributions that have been used to evaluate the performance of DT algorithms (Blelloch *et al*. 1999, Chen *et al*. 2006): Uniform, the Kuzmin, and Line Singularity. The Kuzmin distribution is a radial distribution with distribution function, $M(r) = 1 - 1/\sqrt{1+r^2}$ where r is the distance to the centre. The points in a Line Singularity

Table 3. Behaviour of the proposed algorithm against GPU-DT over synthetic data over a model of 2 million points.

Model	PCDT (ms)	GPU-DT (ms)
Uniform-Random	1294	1747
Uniform-Lexicographically	695	26,129
Kuzmin-Random	2134	2901
Kuzmin-Lexicographically	1161	24,927
Line-Random	1674	2339
Line-Lexicographically	770	48,918

Table 4. Behaviour of the proposed algorithm against GPU-DT over synthetic data over a model of 2 million points and 1 million segments.

Model	PCDT (ms)	GPU-DT (ms)
Uniform-Random	1006	2054
Uniform-Lexicographically	746	25,910
Kuzmin-Random	2139	2293
Kuzmin-Lexicographically	1515	20,015
Line-Random	1831	2343
Line-Lexicographically	1340	53,070

distribution converge to a line instead of a point and are obtained by the transformation, $(u, b/(v(1-b)+b))$ where $b \in [0,1]$ is a parameter and u and v are two independent uniform variables in the unit interval [0,1]. In our experiments, b was set to 0.01. Both Kuzmin and Line Singularity are highly skewed distributions, and so standard space partition techniques do not work well. Two models of 2 million points were created for each distribution. The points of the first model were sorted randomly, while the points of the second model were sorted lexicographically. Similarly, two further models of 2 million points and 1 million segments were created for each distribution. We applied our approach and GPU-DT to them and the results are presented in Tables 3 and 4. Observe that our approach once again achieves much better results in all models and, moreover, GPU-DT does not achieve competitive running times when the points are not sorted randomly.

5. Conclusions

We presented an algorithm and its CUDA implementation for a parallel generation of the 2D CDT of a PSLG. Our algorithm is based on the concurrent insertion of points and segments into the triangulation with the restrictions that, in each iteration, only one point can be inserted into one single triangle and only one edge per triangle can be flipped. The algorithm is easy to implement and our experimental results show that it is competitive in terms of running time with respect to other state-of-the-art methods. Moreover, since it works well for general distributions, our algorithm is suitable for GIS applications.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work was supported by the Spanish Ministerio de Economía y Competitividad [grant TIN2014-52211-C2-2-R].

ORCID

Narcís Coll http://orcid.org/0000-0001-8111-9158

References

Antonopoulos, C.D., *et al*., 2009a. A multigrain Delaunay mesh generation method for multicore smt-based architectures. *Journal Parallel Distrib Computation*, 69, 589–600. doi:10.1016/j.jpdc.2009.03.009

Antonopoulos, C.D., *et al*., 2009b. Algorithm, software, and hardware optimizations for Delaunay mesh generation on simultaneous multithreaded architectures. *Journal Parallel Distrib Computation*, 69, 601–612. doi:10.1016/j.jpdc.2009.03.005

Blelloch, G.E., *et al*., 1999. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24, 243–269. doi:10.1007/PL00008262

Chen, M.B., Chuang, T.R., and Wu, J.J., 2006. Parallel divide-and-conquer scheme for 2D Delaunay triangulation. *Concurrency Computat.: Practical Exper*, 18, 1595–1612. doi:10.1002/cpe.1007

Chrisochoides, N. and Nave, D., 2003. Parallel Delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 58, 161–176. doi:10.1002/(ISSN)1097-0207

Chrisochoides, N. and Sukup, F., 1996. Task parallel implementation of the Bowyer-Watson algorithm. *In: Proceedings of Fifth International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, Mississippi: Mississippi State University, 773–782.

Cignoni, P., *et al*., 1993. Parallel 3D Delaunay triangulation. *Computer Graphics Forum*, 12, 129–142. doi:10.1111/cgf.1993.12.issue-3

Fischer, I. and Gotsman, C., 2006. Fast approximation of high-order voronoi diagrams and distance transforms on the GPU. *Journal Graphics Tools*, 11 (4), 39–60. doi:10.1080/2151237X.2006.10129229

Kohout, J., Kolingerová, I., and Zára, J., 2005. Parallel Delaunay triangulation in E^2 and E^3 for computers with shared memory. *Parallel Computing*, 31, 491–522. doi:10.1016/j.parco.2005.02.010

Lee, S., Park, C.I., and Park, C.M., 1997. An improved parallel algorithm for Delaunay triangulation on distributed memory parallel computers. *In: Proceedings of the 1997 Advances in Parallel and Distributed Computing Conference (APDC '97)*, Washington, DC: IEEE Computer Society, 131–138.

Lo, S.H., 2012. Parallel Delaunay triangulation – Application to two dimensions. *Finite Elements in Analysis and Design*, 62, 37–45. doi:10.1016/j.finel.2012.07.003

Navarro, C., Hitschfeld, N., and Scheihing, E., 2011. A parallel GPU-based algorithm for Delaunay edge-flips. *Abstracts from 27th European Workshop on Computational Geometry*, 2011, 75–78.

Qi, M., Cao, T.T., and Tan, T.S., 2013. Computing 2D constrained Delaunay triangulation using the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 19 (5), 736–748. doi:10.1109/TVCG.2012.307

Rong, G., Tan, T.S., and Cao, T.T., and Stephanus, 2008. Computing two-dimensional Delaunay triangulation using graphics hardware. *In: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, New York, NY: ACM, 89–97.

Santaló, L.A., 1976. *Integral geometry and geometric probability*. Reading, MA: Addison-Wesley.

Shewchuk, J.R., 1996. Triangle: engineering a 2D quality mesh generator and delaunay triangulator. *In: Applied Computational Geometry: Towards Geometric Engineering, Lecture Notes in Computer Science*, Berlin: Springer-Verlag, 1148, 203–222.

Wu, H., Guan, X., and Gong, J., 2011. ParaStream: a parallel Delaunay triangulation algorithm for LiDAR points on multicorearchitectures. *Computers and Geosciencies*, 37, 1355–1363. doi:10.1016/j.cageo.2011.01.008

Wu, W., *et al*., 2014. Novel parallel algorithm for constructing Delaunay triangulation based on a twofold-divide-and-conquer scheme. *GIScience & Remote Sensing*, 51 (5), 537–554. doi:10.1080/15481603.2014.946666