PROVABLY CONSISTENT DISTRIBUTED DELAUNAY TRIANGULATION

Mathieu Brédif [1]* Laurent Caraffa[1] Murat Yirci[1]
Pooran Memari[2]

[1] LASTIG, Univ Gustave Eiffel, ENSG, IGN, F-94160 Saint-Mande, France - firstname.lastname@ign.fr
[2] CNRS, LIX, Ecole Polytechnique, IP Paris, France - memari@lix.polytechnique.fr

Commission II, WG II/3

KEY WORDS: Computational Geometry, Delaunay, Cloud computing, Spark, Point Cloud

ABSTRACT:

This paper deals with the distributed computation of Delaunay triangulations of massive point sets, mainly motivated by the needs of a scalable out-of-core surface reconstruction workflow from massive urban LIDAR datasets. Such a data often corresponds to a huge point cloud represented through a set of tiles of relatively homogeneous point sizes. This will be the input of our algorithm which will naturally partition this data across multiple processing elements. The distributed computation and communication between processing elements is orchestrated efficiently through an uncentralized model to represent, manage and locally construct the triangulation corresponding to each tile. Initially inspired by the star splaying approach, we review the Tile& Merge algorithm for computing Distributed Delaunay Triangulations on the cloud, provide a theoretical proof of correctness of this algorithm, and analyse the performance of our Spark implementation in terms of speedup and strong scaling in both synthetic and real use case datasets. A HPC implementation (*e.g.* using MPI), left for future work, would benefit from its more efficient message passing paradigm but lose the robustness and failure resilience of our Spark approach.

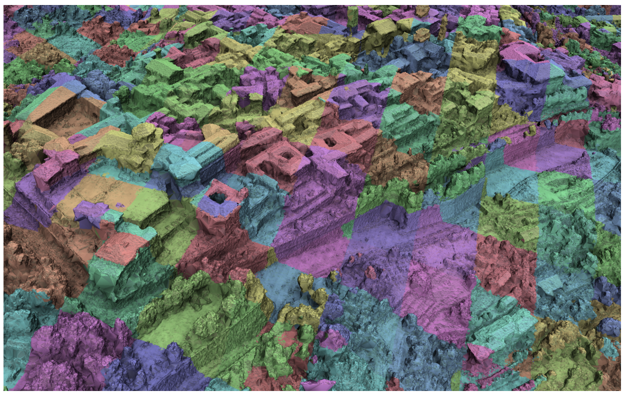
Figure 1. Target application: 3D surface reconstruction based on the proposed out-of-core Delaunay algorithm on 54 million points. Triangles of the resulting mesh are colored according to the tile that contains one of their vertices.

1. INTRODUCTION

Delaunay triangulation is arguably known as the most common discrete representation tool for complex 3D shape models. The problem of computing the Delaunay Triangulation (DT) of large point sets has been considered in different scientific and engineering fields from computer graphics [Fuetterling et al., 2014], scientific visualization [Antaki et al., 2000], fluid simulation [Ando et al., 2013], computer vision [Hiep et al., 2009], multimedia [Tekalp, Ostermann, 2000], pattern recognition [Xiao, Yan, 2003], to geology [Kaufmann, Martin, 2009,Wang et al., 2017] and astrophysics [Sousbie, 2011] [Starinshak et al., 2014] [Zhao et al., 2016].

A DT is a well-defined topological object in any dimension. Among all possible triangulations of a given point cloud, the DT is the unique triangulation that has optimal simplices (*i.e.* triangles in 2D and tetrahedra in 3D) in the following sense : the interior of the sphere (circle in 2D) circumscribing each simplex is empty of other points [Boissonnat, Yvinec, 1998]. This empty sphere property ensures some compactness on the simplices, which is a must in numerical computation (although it is not sufficient to eliminate elongated tetrahedra, called Slivers). Now that research and industry are commonly facing massive point clouds, far beyond the available memory of existing computers, there is a pressing demand for scaling out the computation of the Delaunay Triangulation using out-of-core, streaming, parallel or distributed approaches.

In this project, the initial motivation of this distributed computational problem was from the Geospatial and photogrammetric viewpoint, where we were looking for both efficiency and theoretical guarantees on the robustness of computation, under the technical constraint of using the Spark Big Data framework [Zaharia et al., 2016] to implement the scheduling and communication of the computation.

The robustness of DT computation is required for further applications in 3D reconstructions. Thus, in this framework, given that recent LiDAR sensors are routinely acquiring massive 3D point clouds, the computation of a 3D DT is a challenging preprocessing step for higher level workflows such as surface reconstruction [Labatut et al., 2009, Caraffa et al., 2016]. Moreover, with the explosion of deep learning and graph neural networks [Wu et al., 2019], the need for large scale DT algorithm combined with efficient data storage becomes crucial.

Figure 1 shows a surface reconstruction based on a distributed 3D DT : each tetrahedron of the DT is assigned a boolean occupation value and the surface is reconstructed as the set of 3D triangles between tetrahedra with differing occupation values. This approach ensures watertightness of the surface on the

* Corresponding author

whole scene thanks to the global DT that guarantees the consistency across tiles, without any cracks at tile boundaries. This watertightness property is obviously primordial for flood simulation or visualization.

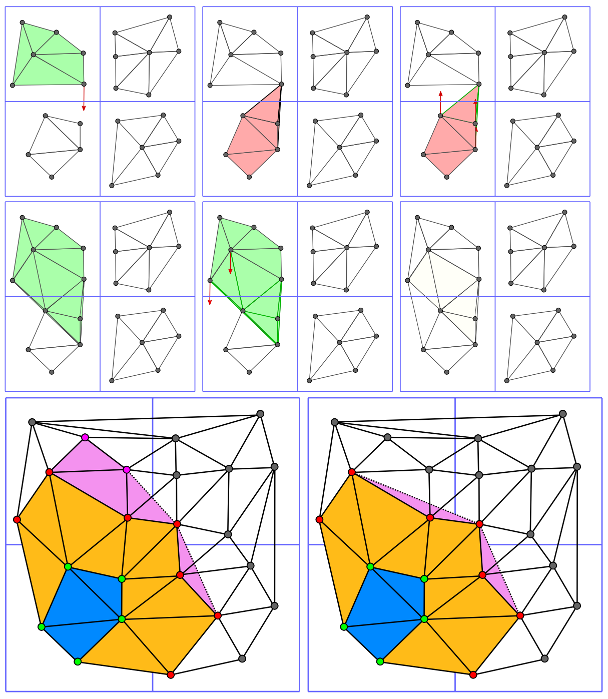
●: Local vertices, ●: Foreign vertices, ●: Redundant foreign vertices, ▲: Local cells, ▲: Mixed cells, ▲: Foreign cells.

Figure 2. The first two rows show the principle of the algorithm, that iteratively exchange points between tiles. The green and red triangulations exchange points of Delaunay neighbor candidates until convergence. The last row shows the triangulation of the bottom left tile before (left) and after (right) simplification.

1.1 Related Work

Given the ever-increasing data resolution in various application fields, computational problems related to Delaunay triangulations, as a common mesh-based representation tool, are more and more challenging and have been tackled in different contexts such as scientific data visualization, surface construction, finite element analysis, train modeling or abstract data-mining.

There is a huge corpus of research tackling the scaling of Delaunay Triangulation computations to point sets of ever increasing sizes. A first approach is the streaming approach of [Isenburg et al., 2006] which proposes an out-of-core algorithm with a small number of passes over the point cloud. This streaming ensures that the peak memory usage is limited as the dataset is loaded and loaded as needed. The first passes analyze the point cloud distribution and characteristic so that the following passes may have some guarantees that whole subsets of the input point cloud are not necessary to compute the Delaunay neighborhood of currently loaded points. This way, the following pass may use these guarantees to perform incremental Delaunay insertion considering the loaded points only, and write down triangles and unload (so-called finalized) points, knowing that subsequent points will not affect them. This approach works well in practice in 2D but in higher dimensions a much smaller fraction of the computed simplices may be unloaded early due to their huge circumsphere, which limits the performance.

Among distributed methods, that leverage multiple Processing Elements (PE), we can distinguish shared memory and distributed memory approaches [Peterka et al., 2014], [Funke et al., 2018], [Chen, Gotsman, 2012], [Si, 2015]. PEs with shared memory may communicate through a common memory address space and are thus relatively fast, allowing fine grain parallelization approaches and often relying on synchronization primitives and locks [Kohout et al., 2005, Blandford et al., 2006, Batista et al., 2010]. A popular framework from HPC computing on high end clusters is MPI, which can take advantage of low level synchronizations and efficient message passing to design very efficient algorithms [Remacle et al., 2015], even reaching the milestone of three billion tetrahedra produced in a minute on a single (very high end) machine [Marot et al., 2018]. In contrast, distributed approaches consider separate memory segments for each PE and thus require explicit communications [Lee, Lam, 2008, Starinshak et al., 2014]. These approach tend to be less efficient in processing speed but scale out to larger datasets at reasonnable hardware costs.

It is a common approach to tackle large data structures by operating on a spatial partitioning of them, for instance for combinatorial maps [Damiand et al., 2018] or 3D triangular meshes [Cabiddu, Attene, 2015]. The Tile & Merge approach [Caraffa et al., 2019] works similarly by decomposing the overall Delaunay Triangulation into a set of DT local to each input tile. However, contrary to other approaches, it is not a proper partition as cells (triangles in 2D) across tile boundaries are replicated in the triangulations of their neighboring tiles and points are replicated in the tiles of all their Delaunay neighbors.

In this paper, we extend the preliminary work of [Caraffa et al., 2019] on distributed Delaunay triangulation computation, by providing a (i) **theoretical proof of the consistency** for the resulting Distributed Delaunay Triangulation. In addition, we also discuss (ii) implementation details on the Spark architecture and propose (iii) a deeper scalability analysis of the overall algorithm in the results section. After providing an overview of the method proposed by [Caraffa et al., 2019], we review their algorithm and propose a theoretical proof of consistency in section 2. Then, section 3 presents the implementation details, followed by the evaluation 4. Finally, the results of our method are shown and discussed in section 5.

1.2 Overview of the Proposed Method

Although the proposed Distributed Delaunay Triangulation computation could be implemented on a HPC cluster (*e.g.* using MPI), which would benefit from its more efficient message passing paradigm, this is left for future work and we implemented the computation using the Spark big data framework for the scheduling and communication, as it is widely deployed in the targeted engineering contexts, and offers robustness and failure resilience guarantees.

As the input of our algorithm as well as the computation are distributed, the output Delaunay triangulation is itself distributed into tiles efficiently so that the overall triangulation is never materialized at any single location. This is done through a method initially inspired by the star playing approach [Shewchuk, 2005] where we consider an iterative shuffle of the so-called **star** points between tiles until reaching a global consistent DT (the star of a point in a triangulation is the set of all its neighbors).

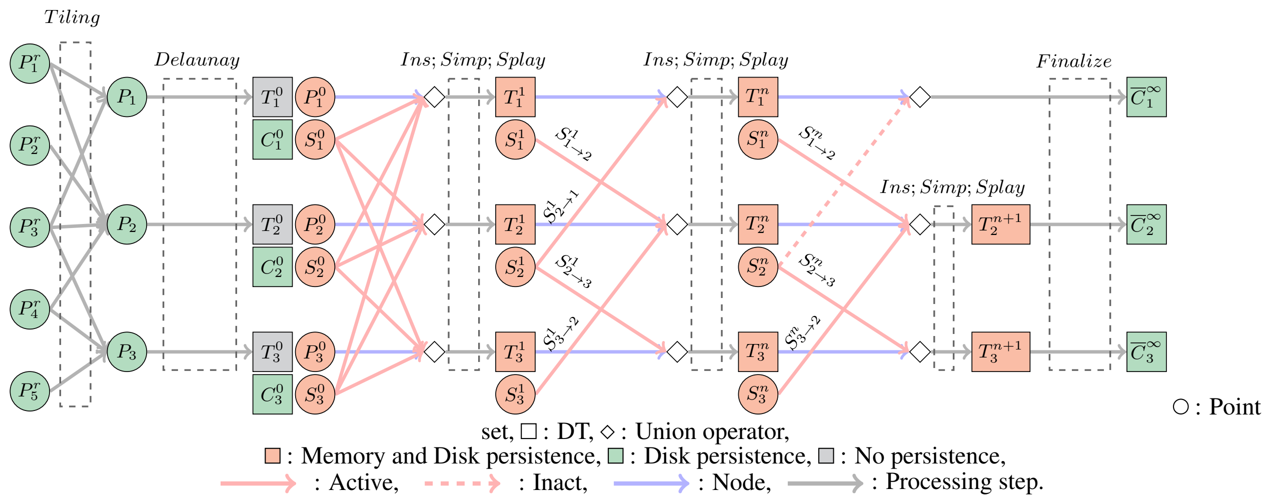
Tiling
Delaunay
Ins; Simp; Splay
Ins; Simp; Splay
Ins; Simp; Splay
Finalize
○ : Point set, □ : DT, ◇ : Union operator,
▧ : Memory and Disk persistence, 🟩 : Disk persistence, ▨ : No persistence,
: Active, : Inact, : Node, : Processing step.

Figure 3. Overview of the distributed Delaunay triangulation workflow in Spark [Caraffa et al., 2019]. The node colors denote the Spark persistence levels of the various datasets.

Points are provided partitioned into tiles and the point set of each tile is triangulated in parallel independently. Figure 2 shows the overall principle on a toy example from the point of view the triangulation of a single tile (in green). This tile sends one point to its lower neighbor. This point is then inserted in the red triangulation, local to the lower left tile. This triangulation then sends back points it thinks should be adjacent to the received point. These red points are now received and inserted in the green triangulation, which discovers their potential neighbors in the green tile. The algorithm exchange points similarly until convergence.

In the end, each tile has a Delaunay triangulation of (i) its own points and (ii) the minimal set of points from other tiles (so-called foreign points) so that each local point has the same Delaunay neighborhood in the local triangulation of the tile than in the theoretical overall DT of all points. The last row shows in color a classification of the final DT of the lower left tile. Blue triangles are local to the tile, orange triangles are mixed, or shared (and represented in the different tiles of their points) and the pink triangles are foreign cells which are not part of the overall DT but present in local DTs to ensure that the convex hull of all local and received points are triangulated. Foreign points adjacent to foreign triangles only may be deleted without modifying the Delaunay neighborhood of local points.

2. DISTRIBUTED DELAUNAY TRIANGULATION

2.1 Definitions

Let us recall some main notions and terminologies introduced in the context of Tile and Merge algorithm. If P is a point set, $P_I = (P_i)_{i\in I}$ will be its partition into $|I|$ disjoint subsets P_i, where I is a discrete set of tile indices. A $N+1$-simplex is a set of $N+1$ vertices in N dimension (a triangle in 2D, a tetrahedron in 3D...). A triangulation is then a set of cells which geometry covers the convex hull of its vertex points. $Delaunay(P)$ denotes the DT of the point set P, and $DelaunayIns(P,T) = Delaunay(P \bigcup_{c\in T} c)$ the DT of the union of the points set of T and the point set P.

A **tile-triangulation** T_i is defined a triangulation of a supset Q_i of the local points P_i. $Q_i \setminus P_i$ are foreign points in T_i. A collection of tile-triangulations $T_I = (T_i)_{i\in I}$ constitutes a **distributed triangulation**. We define a simplex as local if all its points are local, foreign if all its points are foreign and mixed otherwise (see figure 2). Lastly, the star $Star(p,T_i)$ of point p is the subcomplex of T_i induced by $p \in P_i$ and all its neighbors in T_i. For brevity, we consider stars as sets of cells and cells as set of vertices, such that for instance $\{p\} \cup Neighbors(p,T) = \bigcup_{c\in Star(p,T)} c$.

Triangulation distribution: The distributed triangulation T_I of a triangulation $T = DT(P)$ according to a point-partitioning P_I is defined as $(DT(Q_i))_{i\in I}$, with $Q_i = P_i \cup \bigcup_{p\in P_i} Neigbors(p,T)$. T_i is thus a local view of T, since it contains all its local stars: $Star(p,T) = Star(p,T_i)$ if $p \in P_i$. Likewise, the overall triangulation T can be reconstructed from a distributed triangulation T_I with a partitioning P_I as $T = \bigcup_{i\in I} \bigcup_{p\in P_i} Star(p,T_i)$

2.2 Algorithm

Figure 3 and algorithm 1 [Caraffa et al., 2019] describe the Tile & Merge approach. P_i are point sets available after tiling the input data. Then each tile computes in parallel the DT T_i^0 of its local points. The extreme points $S^0_{i\to all}$ of each tile are then broadcast, which is a rather limited set of maximum $2N$ points per tile in N dimensions (1 per axis direction). Then an iteration inspired by Star Splaying is performed [Shewchuk, 2005] : each tile iteratively receives points, insert them to their DT and send Delaunay neighbor candidate points to their neighboring tiles.

Note that instead of working on the whole point cloud, the initial tile triangulation step detect which points have already provably found their Delaunay neighborhood so as to concentrate on the reduce point set of more problematic points. These initially finalized cells C_i^0 are at last merged with a subset $\overline{C}_i^\infty$ of the distributed triangulation of the reduced point set to produce the final set of cells of the overall distributed Delaunay triangulation.

A further feature is that tile triangulations are simplified by removing foreign points that are not adjacent to any local points in order to limit the growth of the tile triangulations resulting from receiving candidate Delaunay neighbors that are eventually not Delaunay neighbors of their local point at convergence.

Algorithm 1: Tile and merge [Caraffa et al., 2019]

```
Input: Point set $P$
Output: Distributed DT of $P$: $T_I^\infty = (T_i^\infty)_{i\in I}$
// Point cloud tiling
for each point $p \in P$ do in parallel
    $i \leftarrow TileId(p)$
    $P_i \leftarrow P_i \bigcup \{p\}$
// Local tile triangulations
for each tile $i \in I$ do in parallel
    $T_i^0 \leftarrow Delaunay(P_i)$
    $C_i^0, \overline{C}_i^0 \leftarrow Finalize(T_i^0)$
    $P_i^0 \leftarrow \bigcup_{c\in\overline{C}_i^0} c$
    $S_{i\to all}^0 \leftarrow ExtremePoints(P_i^0)$
// Star splaying initialization
for each tile $i \in I$ do in parallel
    $R_i^1 \leftarrow \bigcup_{j\neq i} S_{j\to all}^0$
    $P_i^1 \leftarrow P_i^0 \cup R_i^1$
    $T_i^1 \leftarrow Simplify(Delaunay(P_i^1))$
    $(S_{i\to j}^1)_{j\neq i} \leftarrow StarSplay(R_i^1, T_i^1)$
// Star splaying iterations
$n \leftarrow 2$
while $\exists i, j \in I$ such that $S_{i\to j}^n \neq \varnothing$ do
    for each tile $i \in I$ do in parallel
        $R_i^n \leftarrow \left(\bigcup_{j\neq i} S_{j\to i}^{n-1}\right) \setminus P_i^{n-1}$
        $P_i^n \leftarrow P_i^{n-1} \bigcup R_i^n$
        $T_i^n \leftarrow Simplify(DelaunayIns(R_i^n, T_i^{n-1}))$
        $(S_i^n)_{j\neq i} \leftarrow StarSplay(R_i^n, T_i^n)$
    $n \leftarrow n + 1$
for each tile $i \in I$ do in parallel
    $C_i^n, \overline{C}_i^n \leftarrow Finalize(T_i^n)$
    $T_i^\infty = C_i^0 \cup \overline{C}_i^n$
return $T_I^\infty = (T_i^\infty)_{i\in I}$
```

2.3 Theoretical Proof

Star Splaying To prove the correctness of the Tile&Merge algorithm, we cast it as a batch-processing version of the original star splaying approach [Shewchuk, 2005] and ensure that we meet the requirements of its theorem 3 on correctness. Namely, we prove that the distributed triangulation $T_I^n = (T_i^n)_{i\in I}$ converges to the distributed Delaunay triangulation of $P_I^0 = (P_i^0)_{i\in I}$.

Instead of working on a running DT as for incremental Delaunay insertion approaches, star splaying is an iterative algorithm that maintains a star for each input point and exchanges points between stars such that these stars eventually get consistent (i.e. agree on shared simplices) and converge to the stars of the DT of the whole point set. The star splaying operator updates the star of each point p by iteratively performing Delaunay insertions of each incoming point q and sending p to the star of q, q to the star of each of its neighbors (except p) in the updated star and vice-versa the neighbors of both p and q to the star of q.

The Tile&Merge algorithm maintains a distributed triangulation $(T_i)_{i\in I}$. Thus, it maintains a star $Star(p, T_i)$ for each point p where $p \in P_i$. By construction, these stars are consistent within a tile as they are extracted from a common triangulation T_i, but not necessarily across tiles. The Delaunay insertions of the incoming points R_i^n into the tile triangulation T_i^n has the effect of splaying the stars of vertices of T_i^n. The simplification then Delaunay-removes points outside of the stars $Star(p, T_i^n)$ of the local points $p \in P_i^0$. $StarSplay(R_i^n, T_i^n)$ then prepares points according to the original star splaying algorithm, so that they can be sent to other tiles to splay the stars of their local points. Thus, after convergence and for each $p \in P_i^0$, the initial stars $Star(p, T_i^1)$ are splayed into stars $Star(p, T_i^n)$ as if they were processed by the original sequential star splaying algorithm.

Theorem 3 of [Shewchuk, 2005] states a sufficient condition on the initial stars to guarantee the consistency and correctness of the DT resulting from aggregating the iteratively splayed stars. Namely, the initial (non necessarily Delaunay) star of each point except the lexicographically minimum point should contain at least one point that lexicographically precedes it. Note that, this condition is trivially met if the lexicographically minimum point of the whole point set is in the initial star of each point. Since the extreme points are broadcast across tiles, the star of each point is thus confronted to the lexicographically minimum point of each tile, so that the correctness theorem applies: the stars of the local points within the local triangulations T_i^n are indeed equal to the stars of the DT of the initial point set : at convergence, for each $i \in I$ and each $p \in P_i^0$, $Star(p, T_i^n) = Star(p, DT(\bigcup_{i\in I} P_i^0)$ and T_I^n is indeed the distributed Delaunay triangulation of P_I^0.

Finalization The cells of the target DT of the whole point set $P = \bigcup_{i\in I} P_i$ may be divided into two categories: C, the local cells which circumsphere is fully contained in the bounding box of its tile and $\overline{C} = DT(P) \setminus C$, the other cells that may not be finalized locally. Indeed, cells in C may not be in conflict with foreign points as long as bounding boxes of the tiles are disjoint, so that they may be computed independently upfront using the DT of each point set P_i: $C = \bigcup_i C_i^0$ (line 6).

Conversely, since T_I^n is the distributed triangulation of $DT(\bigcup_{i\in I} P_i^0)$, its non-finalized cells $\bigcup_{i\in I} \overline{C}_i^n$ are also the non-finalized cells of $DT(\bigcup_{i\in I} P_i^0)$. By definition of P_i^0, points in $F_i = P_i \setminus P_i^0$ are adjacent to finalized cells of T_i^0 only and their incident cells in T_i^0 (or, equivalently, in $DT(P)$) are all local. Thus, they may be removed from $DT(P)$ without modifying mixed cells. This proves that the mixed cells of T_i^n, which are the ones of $DT(\bigcup_{i\in I} P_i^0)$, are indeed the mixed cells of $DT(P)$. The insertion of a local point of F_i in a local cell may only create local cells. Thus, if it were in conflict with a non-finalized local cells of T_i^n, its insertion would create at least one non-finalized cell. This cell being local, it should already be present in T_i^0, where the point is only adjacent to finalized cells, which is a contradiction. Thus non-finalized local cells of T_i^n are not in conflict with the finalized points, and finally, $\overline{C} = \bigcup_i \overline{C}_i^n$.

3. IMPLEMENTATION DETAILS

All the communications between executors are preformed by a streaming architecture scheduled with Spark. Data sets are both persisted on memory and disk. Large data sets like input point clouds and finalized cells are stored only on disk (green color in figure 3) and lightweight data sets like simplified triangulation during the iterative scheme that are likely to have multiple I/O are stored both in memory and disk.In this section, the implementation choices are detailed. For implementing, both C++ and Spark are used. On a higher level, Spark provides a fault-tolerant and lazy programming language that distributes efficiently the operations on the cluster according to the data location thanks to the use of resilient distributed data sets (RDD).

C++ C++ is widely used in computational geometry because of the low-level architecture that allows high speed computational time. Each geometric operation is implemented with a C++ executable leveraging the CGAL library. Once a triangulation is computed, the C++ sets are encoded with the *Base64* encoding, consequently, the data integrity is guarantied across transformations.

Spark The Scala interface of Spark is used for the scheduling process. Each point set P_I and triangulation T_I are stored in a RDD of size $|I|$ serialized with a *Base64* encoding in a *String*. The key/value formalism is used. Each element of the RDD is represented by a key K and a value V which is a list of sets (Point Set, local view of a triangulation, etc.). The value is then $V = List[String]$, finally we have $RDD[(K, V)]$. To implement the shuffling $S_{i\to j}$, the GraphX library [Gonzalez et al., 2014] is used. An exchange is stored as an edge of a graph with the triplet $RDD[(K_i, K_j, V)])$ where K_i is the key of the source and K_j the key of the target. For each RDD transformation that requires geometric processing, the content of a RDD is encoded in base64 and streamed to a C++ executable by using the transformation *pipe* operator for Spark RDD. Since each serialized value encodes its own key, the C++ thread can interpret and build the local view of the triangulation. The union operator ($\cup$ in alg:1 and $\diamond$ in figure 3) is a union following by a *ReduceByKey* in Spark. As an example, the star splaying step (lines 13 and 20) can be written as follows:

$$(\ RDD[(K_i, K_j, S_{i\to j})].map(e \to (e.K_j, e.V)) \cup RDD[(K, T_j)] \ ).reduceByKey((a, b) \to (a \cup b)).pipe(\text{./StarSplay})$$

Where ./StarSplay is a C++ executable that takes as input the union of the previous step triangulation and the received points, do the insertion, the simplification, extract the stars and produce as output the new triangulation with the new points to send. Finally, a filter operator is used to separate the output stream of each C++ call.

4. EVALUATION

In this section, we provide a deeper analysis of the results of [Caraffa et al., 2019], which were produced on a Spark cluster with 28 cores and 100GB memory (to be compared with Amazon EMR service with *e.g.* 16 cores for 128GB of RAM with *m5.8xlarge*) Two aspects are analyzed here: the strong scaling that shows how the algorithm scales according to the cluster configuration (see figure 6) and how one configuration scales according to an increasing number of points. We use 3 different data sets: points generated from i) a normal distribution, ii) a uniform distribution and iii) from LiDAR acquisitions (see figure 5). For these tests, the clock starts when point sets are generated / loaded from HDFS and stops when all the finalized cells of the triangulation are persisted on cluster nodes.

Figures 4 and 5 show qualitatively some example results on small data sets: figure 4 on 1 million random points generated with a normal distribution with a 3x3x3 tiling. Figure 5 on a LiDAR cloud of 3 million points. A cut in the mesh is done for visibility purpose. The gray cells are the local cells finalized after the first triangulation step. The black cells with colored edges are non-finalized cells from the last step. Cells with the same colors belong to the same tile.

Parameter tuning Among all possible parameters (at the Spark or JVM level), we only focus here on the following parameters that impact the most the behaviour of the algorithm.

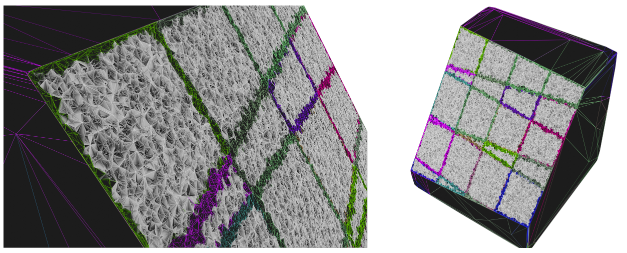
Figure 4. 1M random points triangulation on a 3x3x3 grid.

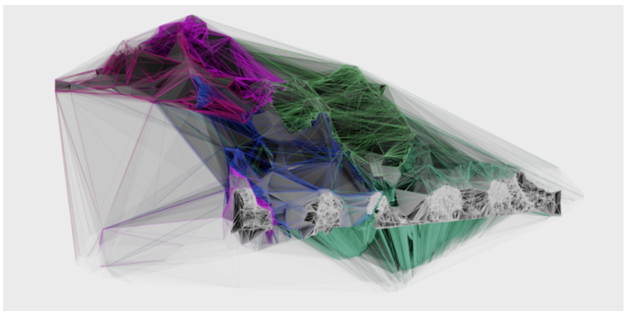
Figure 5. 3M LiDAR points triangulation on a 3x3x3 grid.

- At the algorithm level with:
  - The maximum number of points per tiles,
  - The number of points extracted in the first step.
- At the RDD storage level with:
  - Number of partitions,
  - Persistence type.
- At the cloud initialization level with:
  - Number of executors,
  - Amount of memory of an executor,
  - Number of cores per executor.

[Caraffa et al., 2019] discussed that the main parameter is the maximum number of points per tile that is allowed during the tiling construction. It is clear that, as the batch Delaunay insertions is a coarse grain step with in-memory inputs and no need for synchronisation, any overly optimized and state of the art implementation may be used. Thus, the larger the number of points per tile, the better, until the memory limit is reached. The octree structure allows a constant number of points per tile. Since our local DT implementation is very efficient thanks to the *C++* implementation compared to the cost of iterative scheme caused by the transfer of non-finalized cells, the higher the number of points per tile during the first triangulation (line 5), the faster the algorithm. With a maximum number of 4GB of memory per executor, each tile can handle during the first iteration 3M points per tile. With a bigger values, memory issues appear in the tested configuration. Figure 4 shows an example of 1M points generated with a uniform distribution tiled on a 4x4x4 grid.

To have a generic configuration that is both efficient with a small number of points but can also handle an large point cloud with a small number of executors, two persistence levels are chosen. Input point sets and finalized cells are persisted only on disk as we only need to read them one time and their number can be huge. Point sets that are broadcasted and triangulations during the star splaying iterations are stored in memory and disk as the operation is repeated several times. One can set the persistence in memory for every RDD with small data sets and force everything into disk for very large data sets to avoid memory issues. But as the algorithm can handle 2 billion of points with only 7 executors, we strongly advise to increase the number of executors and keep the persistence both in memory and disk.

Scalability To see the ability of the proposed approach to handle an large number of points with a constant configuration, a varying number of points, from 100M to 2G is triangulated with the 7 executors and 4 cores configuration on 3 data sets: Normal distribution and Random distribution on a 4x4x4 grid, and LiDAR on a 8x8x8 grid. The algorithm scales well and has no particular difficulty to handle a large amount of point with the a low memory to core ratio configuration.

In detail comparison Figure 8 shows the time spent in the main steps in average for the previous tests . Even if the standard deviation (*std*) does not have a deep statistic meaning in this case, it gives us a clue on the time variation in each step. First, the time spent during the local insertion is more important for the normal distribution than for the LiDAR or the uniform distribution. Moreover, the *std* value is greater than in the two other data sets where it remains stable. This makes sense according to the sparsity and the variation of the point cloud distribution according to the 3 axes. However, it has the advantage to produce a lightweight triangulation with a big interface with the result of a small number of cells and shared points. Indeed, the time spend during the iterative scheme is way shorter compared to the rest and makes the algorithm much faster according to the previous section.

The algorithm on the LiDAR data set has mixed behaviour compared to the two others, the local triangulation remains quick, but some case can take a while. On the contrary, the sparsity can result in small interface and high speed iterative scheme in some cases, such as roads and planar regions, or, by contrast, create an large interface with a high density of shared simplices which may slow down the process significantly. An important aspect is the maximum number of 7 star splaying iterations required for the biggest data set. This indicates that the number of edges grows in a logarithmic fashion. Moreover, the last iteration has a lower cost as the number of active connections fall drastically. As the algorithm persists all the finalized cells at the $finalization$ step, time to save finalized cells on HDFS are also shown. As expected, saving C^0 takes half the time of the whole process when persisting. To conclude, this result shows a relative good stability of the proposed approach despite of the heterogeneity of the data sets.

5. RESULTS

Figure 1 shows how challenging computing the global triangulation of a LiDAR point cloud is with a mobile mapping acquired LiDAR point set of 1.98 billion points over a $6km^2$ area. The computation took 2h20 on a machine with 28 cores and wrote after 4h11 the resulting local triangulations as binary ply to the HDFS distributed file system (400GB) Note how the octree structure handles well the non-uniform distribution of points. The point cloud is distributed mainly on the x,y axis, this implies an large number of active connections (28402) at the first iteration. In LiDAR datasets, outliers generated by non-Lambertian materials create points at wrong positions (e.g. way under the ground or above the city) that increases the maximum degree of the connection graph. Thanks to the fully distributed

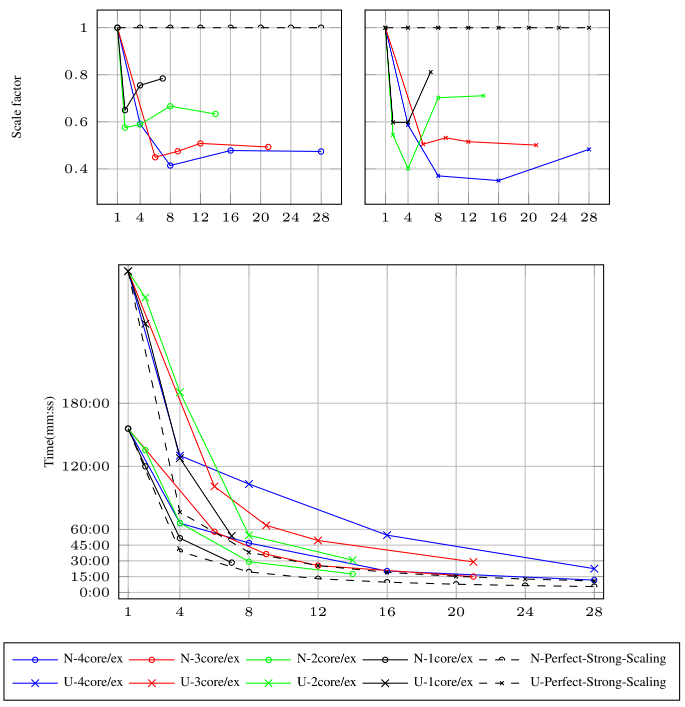

Figure 6. Scaling on processing and persisting the DT according to the number of cores with 4,3 and 2 cores/executors(12Go Ram/executors) on a 300 millions of points test set generated with the normal(N) and uniform(U) random distribution. The two first figures show the scale factor according to the number of core, last figure the time.

framework proposed by the approach, neighbor tiles within a clique of high dimension are never all loaded at the same time and the construction of this case is iteratively solved.

6. CONCLUSION

This article provides a theoretical basis for the distributed computation of the Delaunay triangulation of massive tiled point set proposed in [Caraffa et al., 2019]. We prove that the resulting local triangulation in each tile is consistent with the global triangulation thanks to an iterative process based on a star splaying approach. The pipeline, being fully distributed, efficiently limits the peak memory footprint, and is implemented on the Spark Framework coupled with efficient C++ executables for computational geometry routines. This coarse-grain parallelization is a key that enables this very modular approach to wrap any Delaunay batch insertion implementation with any heavily optimized code Based on our experiments, the scaling performs well according to an increasing number of cores and number of points. As claimed before, the main advantage is that, unlike more hardware dedicated algorithms, this framework can easily be deployed on a Spark cluster and integrates in a full production pipeline for distributed 3D surface reconstruction from LiDAR point clouds. This is justified by the reconstruction experiments we performed on the urban dataset of figure 7. At the implementation level, Spark-DIY [Cano-Lores et al., 2018] defines itself as "A Framework for Interoperable Spark Operations with High performance Block-Based Data Models", which may provide an abstraction layer over spark to design a more efficient Spark implementation. In parallel, the algorithm, while being coarse-grain by design is not tied to its Spark implementation, it would be very interesting to implement an MPI

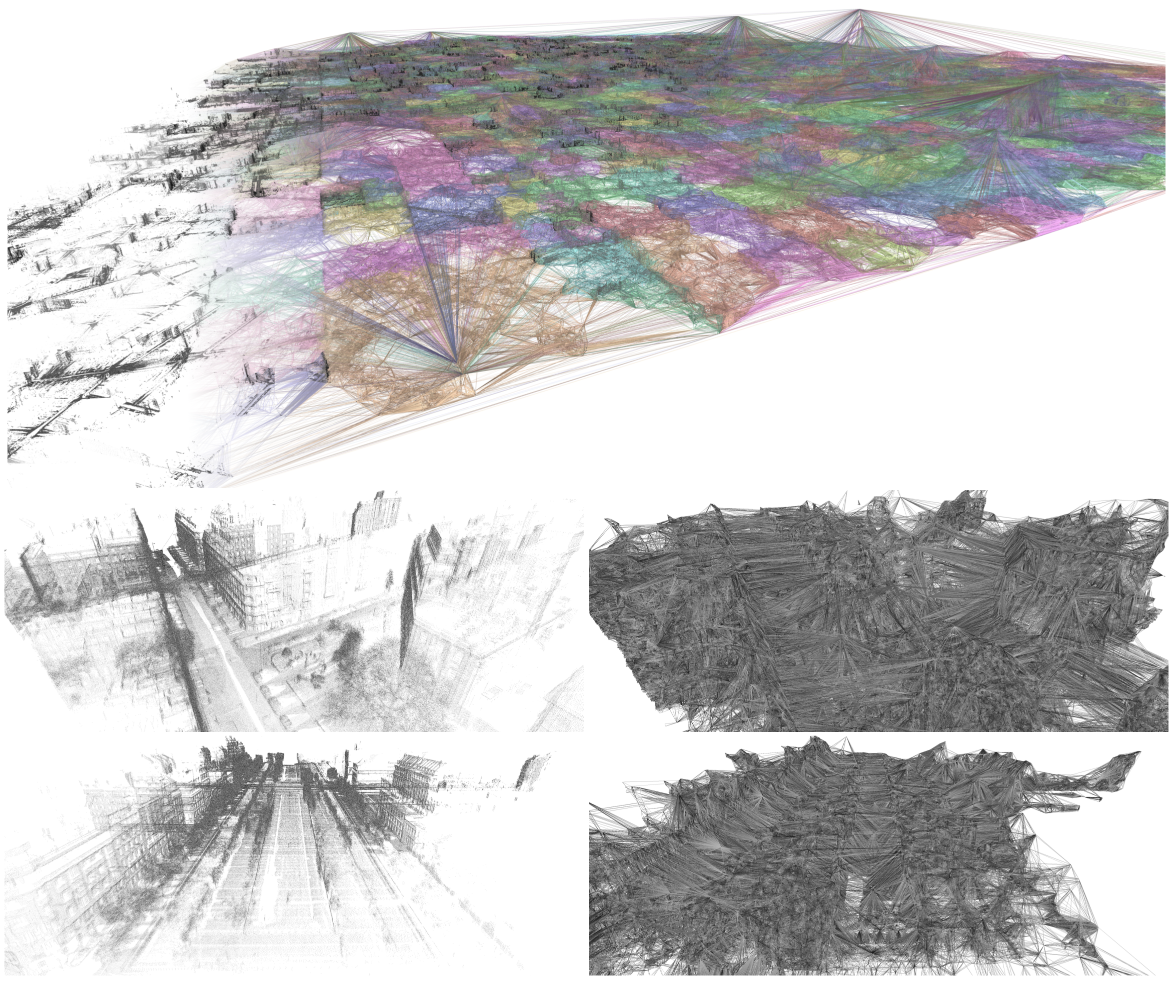

Figure 7. Result of the proposed approach on 1.98 billion of points $6km^2$ with $2cm$ spatial resolution. First line: the whole scene. Only finalized cells during the last step are shown. Triangles of the same tile have the same color. Second and third line: two tiles, only local finalized simplex at the first step are shown.

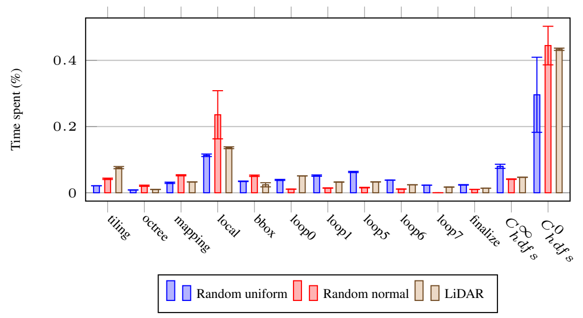

Figure 8. Percentage of time spend on the main step on 3 data sets (in average). Some intermediate iterations are not shown for visibility purpose.

version and evaluate its performance on a HPC cluster. As an interesting future work direction, the proposed framework seems to be extendable in a recursive way to nested or adaptative tiling procedures, where a distributed computation inside tiles containing high number of vertices is considered. Lastly, if an efficient computation of the set of tile pairs that need to exchange points were available (*ie*, the ones that have Delaunay neighbors), then this could be performed as a preprocessing step to drive the Delaunay computation distribution more efficiently. This could also enable an efficient on-demand computation of subsets of the Delaunay triangulation.

ACKNOWLEDGMENTS

This work was partially supported by the European Commissions Seventh Framework Programme under the grant agreement FP7-ICT-2011-318787 (IQmulus: A High-Volume Fusion and Analysis Platform for Geospatial Point Clouds, Coverages and Volumetric Data Sets).

REFERENCES

Ando, R., Thürey, N., Wojtan, C., 2013. Highly Adaptive Liquid Simulations on Tetrahedral Meshes. *ACM Trans. Graph.*, 32(4), 103:1–103:10.

Antaki, J. F., Blelloch, G. E., Ghattas, O., Malcevic, I., Miller, G. L., Walkington, N. J., 2000. A parallel dynamic-mesh lagrangian method for simulation of flows with dynamic interfaces. *Supercomputing, ACM/IEEE 2000 Conference*, IEEE, 26–26.

Batista, V. H., Millman, D. L., Pion, S., Singler, J., 2010. Parallel geometric algorithms for multi-core computers. *Computational Geometry*, 43(8), 663–677.

Blandford, D. K., Blelloch, G. E., Kadow, C., 2006. Engineering a compact parallel delaunay algorithm in 3D. *Proceedings of the twenty-second annual symposium on Computational geometry - SCG 06*.

Boissonnat, J.-D., Yvinec, M., 1998. *Algorithmic geometry*. Cambridge university press.

Cabiddu, D., Attene, M., 2015. Large mesh simplification for distributed environments. *Computers & Graphics*, 51(Supplement C), 81 - 89. International Conference Shape Modeling International.

Caraffa, L., Brédif, M., Vallet, B., 2016. 3d watertight mesh generation with uncertainties from ubiquitous data. *Proceedings of Asian Conference on Computer Vision (ACCV'16)*, LNCS, Springer, Taipei, Taiwan.

Caraffa, L., Memari, P., Yirci, M., Brédif, M., 2019. Tile & Merge: Distributed Delaunay Triangulations for Cloud Computing. *IEEE Big Data 2019*, Los Angeles, United States.

Cano-Lores, S., Carretero, J., Nicolae, B., Yildiz, O., Peterka, T., 2018. Spark-diy: A framework for interoperable spark operations with high performance block-based data models. *BDCAT*, IEEE Computer Society, 1–10.

Chen, R., Gotsman, C., 2012. Localizing the delaunay triangulation and its parallel implementation. *Voronoi Diagrams in Science and Engineering (ISVD), 2012 Ninth International Symposium on*, IEEE, 24–31.

Damiand, G., Gonzalez-Lorenzo, A., Zara, F., Dupont, F., 2018. Distributed Combinatorial Maps for Parallel Mesh Processing. *Algorithms*, 11(7), 105.

Fuetterling, V., Lojewski, C., Pfreundt, F.-J., 2014. High-performance delaunay triangulation for many-core computers. *High Performance Graphics*, 97–104.

Funke, D., Lamm, S., Sanders, P., Schulz, C., Strash, D., von Looz, M., 2018. Communication-free massively distributed graph generation. *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 336–347.

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., Stoica, I., 2014. Graphx: Graph processing in a distributed dataflow framework. *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, USENIX Association, Berkeley, CA, USA, 599–613.

Hiep, V. H., Keriven, R., Labatut, P., Pons, J.-P., 2009. Towards high-resolution large-scale multi-view stereo. *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, IEEE, 1430–1437.

Isenburg, M., Liu, Y., Shewchuk, J. R., Snoeyink, J., 2006. Streaming computation of Delaunay triangulations. *ACM SIGGRAPH 2006 Papers on - SIGGRAPH 06*.

Kaufmann, O., Martin, T., 2009. Reprint of 3D geological modelling from boreholes, cross-sections and geological maps, application over former natural gas storages in coal mines[Comput. Geosci. 34 (2008) 278–290]. *Computers & geosciences*, 35(1), 70–82.

Kohout, J., Kolingerová, I., ára, J., 2005. Parallel Delaunay triangulation in E2 and E3 for computers with shared memory. *Parallel Computing*, 31(5), 491 - 522.

Labatut, P., Pons, J.-P., Keriven, R., 2009. Robust and Efficient Surface Reconstruction From Range Data. *Computer Graphics Forum*.

Lee, D.-Y., Lam, S. S., 2008. Efficient and accurate protocols for distributed delaunay triangulation under churn. *2008 IEEE International Conference on Network Protocols*.

Marot, C., Pellerin, J., Remacle, J., 2018. One machine, one minute, three billion tetrahedra. *CoRR*, abs/1805.08831. http://arxiv.org/abs/1805.08831.

Peterka, T., Morozov, D., Phillips, C., 2014. High-performance computation of distributed-memory parallel 3d voronoi and delaunay tessellation. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Press, 997–1007.

Remacle, J.-F., Bertrand, V., Geuzaine, C., 2015. A two-level multithreaded delaunay kernel. *Procedia Engineering*, 124, 6–17.

Shewchuk, J. R., 2005. Star splaying: An algorithm for repairing Delaunay triangulations and convex hulls. *Proceedings of the Twenty-first Annual Symposium on Computational Geometry*, SCG '05, ACM, New York, NY, USA, 237–246.

Si, H., 2015. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)*, 41(2), 11.

Sousbie, T., 2011. The persistent cosmic web and its filamentary structure–i. theory and implementation. *Monthly Notices of the Royal Astronomical Society*, 414(1), 350–383.

Starinshak, D., Owen, J., Johnson, J., 2014. A new parallel algorithm for constructing Voronoi tessellations from distributed input data. *Computer Physics Communications*, 185(12), 3204–3214.

Tekalp, A. M., Ostermann, J., 2000. Face and 2-D mesh animation in MPEG-4. *Signal Processing: Image Communication*, 15(4), 387–421.

Wang, Y., Ma, G., Ren, F., Li, T., 2017. A constrained Delaunay discretization method for adaptively meshing highly discontinuous geological media. *Computers & Geosciences*, 109, 134–148.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P. S., 2019. A Comprehensive Survey on Graph Neural Networks. *IEEE transactions on neural networks and learning systems*.

Xiao, Y., Yan, H., 2003. Text region extraction in a document image based on the Delaunay tessellation. *Pattern Recognition*, 36(3), 799–809.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I., 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 59(11), 56–65.

Zhao, C., Tao, C., Liang, Y., Kitaura, F.-S., Chuang, C.-H., 2016. dive in the cosmic web: voids with Delaunay triangulation from discrete matter tracer distributions. *Monthly Notices of the Royal Astronomical Society*, 459(3), 2670–2680.