Tile & Merge: Distributed Delaunay Triangulations for Cloud Computing

Laurent Caraffa, Pooran Memari, Murat Yirci, Mathieu Brédif

Tile & Merge: Distributed Delaunay Triangulations for Cloud Computing

Laurent Caraffa
Univ. Paris-Est,
LASTIG ACTE, IGN, ENSG,
F-94160 Saint-Mande, France
laurent.caraffa@ign.fr

Pooran Memari
CNRS-LIX
École Polytechnique
Palaiseau, France
memari@lix.polytechnique.fr

Murat Yirci
Univ. Paris-Est,
LASTIG GEOVIS, IGN, ENSG,
F-94160 Saint-Mande, France
murat.yirci@ign.fr

Mathieu Brédif
Univ. Paris-Est,
LASTIG GEOVIS, IGN, ENSG,
F-94160 Saint-Mande, France
mathieu.bredif@ign.fr

***Abstract*—Motivated by the needs of a scalable out-of-core surface reconstruction algorithm available on the cloud, this paper addresses the computation of distributed Delaunay triangulations of massive point sets. The proposed algorithm takes as input a point cloud and first partitions it across multiple processing elements into tiles of relatively homogeneous point sizes. The distributed computation and communication between processing elements is orchestrated so that each one discovers the Delaunay neighbors of its input points within the theoretical overall Delaunay triangulation of all points and computes locally a partial view of this triangulation. This approach prevents memory limitations by never materializing the global triangulation.**

This efficiency is due to our proposed uncentralized model to represent, manage and locally construct the triangulation corresponding to each tile. The point set is first partitioned into non-overlapping tiles, then we construct within each tile the Delaunay triangulation of the local points and a minimal set of replicated foreign points in order to capture the simplices spanning multiple tiles. Inspired by the star splaying approach for Delaunay triangulation computation/repair, communication is limited to exchanging points of potential Delaunay neighbors across tiles. Therefore, our method is guaranteed to reconstruct, within each tile, a triangulation that contains the star of its local points, as though it were computed within the Delaunay triangulation of all points.

The proposed algorithm is implemented with Spark for the scheduling and C++ for the geometric computations. This allows both an optimal scheduling on multiple machines and efficient low-level computation. The results show the efficiency of our algorithm in terms of speedup and strong scaling on a classical Spark configuration with both synthetic and real use case datasets.

***Index Terms*—Computational Geometry, Delaunay, Cloud computing, Spark.**

I. Introduction

The need to compute the Delaunay triangulation (DT) of extremely large point sets in $\mathbb{R}^d$ is present across a large variety of application domains, ranging from computer graphics [FLP14], multimedia [TO00], pattern recognition [XY03], fluid simulation [ATW13] and computer vision [HKLP09] to geology [KM09], [WMRL17] and astrophysics [SOJ14].

Given a point set, the DT has the nice feature of being well defined and unique (in non-degenerate cases) in any dimension. It guarantees reasonable triangle shapes (at least in 2D) which yields a decomposition of space in rather compact triangles, which is paramount for precise numerical computation.

In the Geo-spatial domain, photogrammetric and LiDAR sensors are now routinely acquiring massive 3D point clouds and the computation of their 3D DTs is a common preprocessing step for higher level workflows such as surface reconstruction [LPK09], [CBV16].

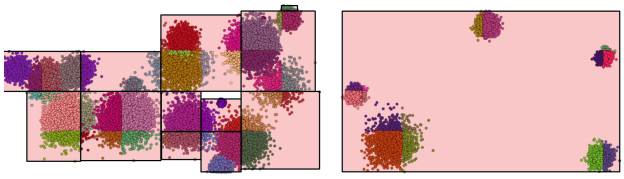
In addition, with the explosion of deep learning and more particularly graph neural networks [ZWa19], the computation of large scale DT combined with efficient data storage becomes mandatory to consider large-scale learning on graphs.

This well-studied computational geometry problem [BY98] is now facing the availability of more and more massive point sets, which asks for scaling out to out-of-core, parallel and distributed methods. This article aims to bridge the gap between efficient single-node DT algorithms and large-scale application on the cloud in a unified framework.

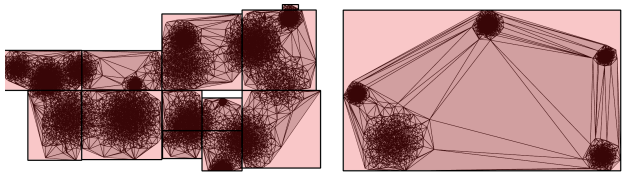
A. Related Work

Many algorithms have been presented to compute the DT of massive point sets. Some algorithms have been developed for only specialized platforms whereas others target a group of systems with certain architectural properties such as shared or distributed memory [FS17]. Specialized platforms may be completely unique or have restricted configurations. For instance, the algorithm proposed by Chen et al. [CCW06] requires the number of processing elements (PE) to be powers of two. In the shared memory systems (e.g. multi-core CPUs), the main memory of the system is accessible to every PE. On the other hand, for the distributed memory systems, each PE possesses its own main memory, e.g. computational clusters. Generally, algorithms designed for shared memory systems [BBK06], [BMPS10] are more effective due to the relatively fast communication between the PEs via the shared memory. On the other hand, algorithms designed for distributed memory systems [SOJ14] are not bounded by the size of the shared memory but by the total number of PEs, their total available memory or their disk space, which can be extended relatively easily, and therefore they can operate on larger data sets.

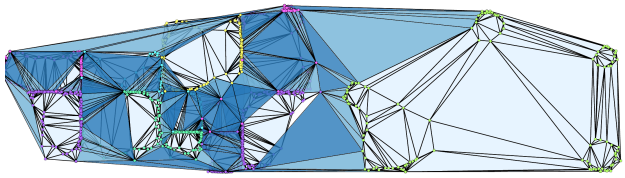
Distributed and parallel computing is not the only way of dealing with large data sets. Out-of-core algorithms can process data that is too large to fit into a computer's main



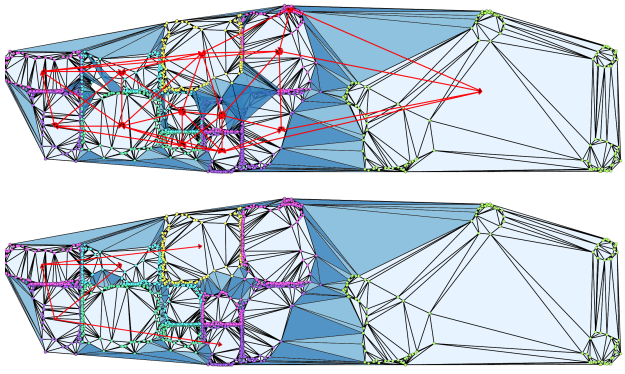
(a) Tiling of the input point cloud : the original grid tiling is shown on the point cloud in color. Grid tiles are merged based on a quad-tree, illustrated by the bounding boxes of the merged tiles.



(b) Local Delaunay triangulation.

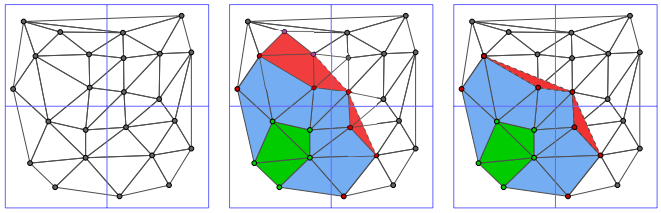


(c) Extreme points broadcast.



(d) Star splaying iterations. Red arrows show the exchanges of points between tiles.

Fig. 1: Example of the full pipeline in 2D. In figure 1c and 1d, only main triangles (in blue) are shown. Foreign triangles are not shown.



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

Fig. 2: The structure of the distributed triangulation: 2a is the full triangulation, 2b shows in color the DT of a supset of the local points of the bottom left corner tile. These extra points, denoted as foreign points are called redundant if they are not adjacent to a local point. Simplices may be categorized as local, mixed or foreign, 2c shows the tile after simplification, by removing redundant foreign points from the local triangulation.

memory at one time [Vit01]. Agarwal et al. [AAY05] proposed an out-of-core approach for the computation of constrained DT. Isenburg et al. [ILSS06] presented a two step out-of-core algorithm. The first step passes over the input data set three times, and gathers spatially coherent points into buckets. In the second step, points from buckets are streamed into an incremental insertion algorithm. The constructed triangles / tetrahedra are classified as finalized if their circumcircles / circumspheres are guaranteed to remain empty. The finalized triangles / tetrahedra are saved to disk, therefore, only the unfinalized parts of the triangulation remain in the memory, which reduces the memory requirements of the algorithm. The algorithm is very efficient for 2D data sets such that it can process 11GB of LiDAR points in 48 minutes using 70MB memory, but the performance decreases with 3D surface point clouds, for which the memory requirements grow due to the increasing number of unfinalized tetrahedra (as very large circumspheres may exist).

The authors of [PMP14] that provides one of the most efficient DT algorithm on a dedicated architecture, introduces Spark-DIY [CLCN*18], a "A Framework for Interoperable Spark Operations with High performance Block-Based Data Models". As OpenMp-MPI architecture is fast, the lack of fault tolerant processes and genericity becomes too disadvantageous regarding to the application and the infrastructures available on frameworks like Spark [ZXW*16].

It is against that background that a highly distributed algorithm that can be easily deployed on Spark framework is proposed in this article. Without being optimal in message passing by using Spark-DIY, the proposed method shows very good results on scalability tests regarding the number of points or the number of cores with synthetic and real data sets.

Next, an overview of the proposed method is outlined. In section II, the algorithm is introduced with the theoretical guarantees. Then, section III presents the evaluation. Finally, the results of our method are shown and discussed in section IV.

B. Overview of the Proposed Method

This algorithm is based on the star playing approach [She05]. The star of a point is composed of its adjacent points and simplices in the overall triangulation. Star splaying maintains independent stars for each point (which may initially be inconsistent : Point A may think he is a neighbor of B, but B may disagree) and iteratively shuffles points between stars until they are all made consistent and converge to the stars of the overall DT. In our approach, as both the input and the computation are distributed, the output DT is itself distributed as well into tiles. The proposed approach ensures that all these tiles encode the overall DT in a distributed way, but this overall triangulation is never materialized at any single location.

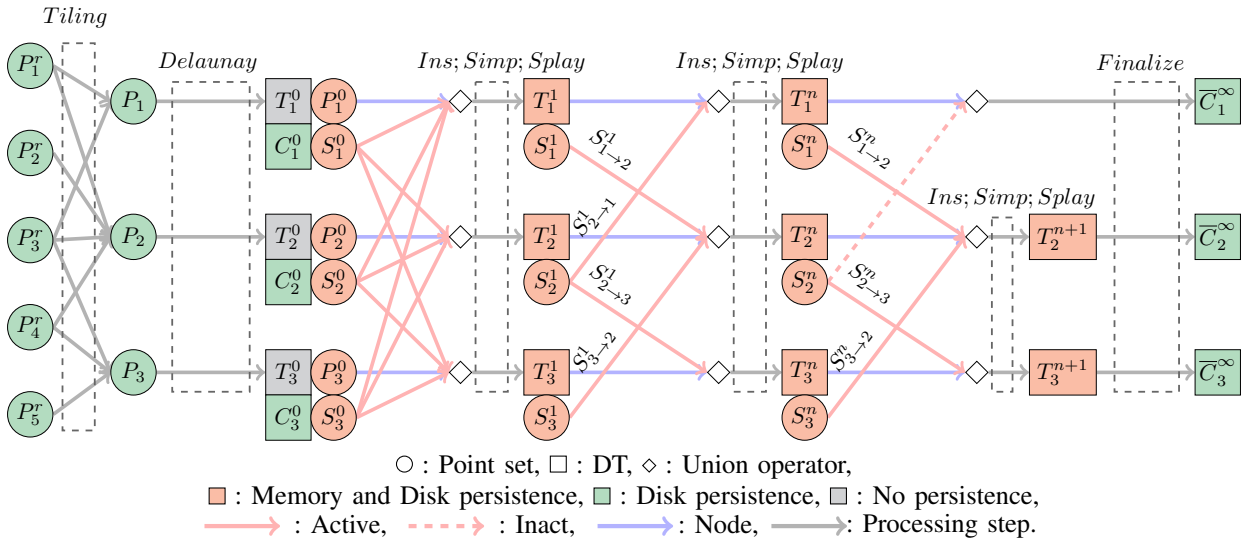


Fig. 3: Proposed distributed Delaunay triangulation workflow in Spark. P^r denotes the input point set, accessible through chunks P^r_k, P_i the tiled point, T^0_i the local triangulation of the tile, $S^0_{i\to all}$ the first broadcasted point set, $S^n_{i\to j}$ the point set sent from tile i to tile j, Solid (resp. dashed) red arrows show active (resp. inactive) connections between two tiles. C^0_i the finalized cells after the first triangulation and $\overline{C}^\infty_i$ the unfinalized cells at the end. Dashed boxes denote geometrical processing transformations (e.g. $Ins; Simp; Splay$ denotes an insertion following by a simplification and starsplaying). The color of the node represents the Spark persistence level.

Figure 1 shows an overview of the algorithm. The input of the algorithm is a point set, that may be stored in multiple files or databases across multiple computers due to massive storage requirements and to benefit from distributed computing. For this article, we process a 120Go LiDAR point cloud of 1.98 billion of points distributed across 12.000 files stored on the Hadoop distributed filesystem (HDFS) [SKRC10].

First, the point set is tiled with a distributed octree approach that consists in choosing a maximum depth of an octree decomposition and then grouping leaf ids while the union of merged tiles do not pass a limit number of points (see figure 1a). The local DT of each tile is computed (see figure 1b). Cells where the circumsphere is inside the bounding box are finalized (i.e extracted and stored), and a point that is only incident to finalized simplices is removed as its Delaunay star is already known. The extreme points of each tile are then extracted and broadcasted to all tiles and inserted with the local points. We call it foreign vertices (see figure 2, 1c). Redundant foreign vertices in the local triangulation can be removed (see figure 2). At this time, a first approximation of the neighbor connections is defined by the remaining foreign vertices in each triangulation (see figure 1d). A connection exists between two tiles when a point has a neighbor with a different tile id. To this point, the iterative scheme starts with a splaying approach that propagates foreign neighbor points and discovers new neighbor relations until no extra point has to be sent (see figure 1c). The result of the algorithm is a set of finalized cells.

As the proposed approach is already functional in concrete applications with classic I/O that belong on C++ code. Because $RDDs$ handle natively *String* type and communicate with external applications by stream, the straightforward approach is to store triangulations and point set serialized in *Strings* and provide an interface that serializes/deserializes *String* streams in C++ for communication. With this approach, the Spark scheduler is still aware of the data to optimize the scheduling over the cluster. Thanks to the C++ implementation, steps that require geometry processing remain fast.

This approach is distinguished by many aspects:

- An out-of-core algorithm based on a fully distributed data structure that provably computes the distributed DT with no bottleneck as the number of executors grow linearly according to the data.
- An hybrid Spark / C++ implementation with evaluations that show the efficiency of our method in terms of speedup on a generic Spark configuration.
- A result that can be directly used with the Spark GraphX framework [GXD*14] for large graph processing.

II. Tile and Merge

A. Definitions

Given a point set P, we note the partition $P_I = (P_i)_{i\in I}$ its decomposition into $|I|$ disjoint subsets P_i, where I denotes a discrete set of tile indices. We can define the **primary** tile of a point $p \in P$ as the unique tile $i \in I$ such that $p \in P_i$.

In dimension N, a cell refers to a $N+1$-simplex, which is a set of $N+1$ vertices (a triangle in 2D, a tetrahedra in 3D...). A triangulation is a set of cells which geometry covers the convex hull of its vertex points. We note $Delaunay(P)$ the DT of the point set P, and $DelaunayIns(P, T)$ the DT resulting from

the insertion of the point set P into the DT T. We define a **tile-triangulation** T_i of tile i as a triangulation of a superset of points $Q_i \supseteq P_i$. A **distributed triangulation** is defined as the set of triangulations $T_I = (T_i)_{i \in I}$. We denote a point $p \in P_i$ as **local** in T_i and **foreign** in T_j if $j \neq i$. By extension, a simplex is classified as **local** (resp. **foreign**) in T_i if all its points are local (resp. foreign) in T_i, and **mixed** otherwise (see figure 2). Finally, we introduce the notation $Star(p, T_i)$ to denote the subcomplex of T_i induced by $p \in P_i$ and all its neighbors in T_i. To simplify notations, a star will be treated as a set of cells (and likewise a cell as a set of vertices) : *e.g.* $\{p\} \cup Neighbors(p, T) = \bigcup_{c \in Star(p,T)} c$. Note that, unless otherwise specified, all triangulations will be Delaunay [BY98] in the following sections.

Triangulation distribution: Given a triangulation $T = DT(P)$ and a point-partitioning P_I, we define the distributed triangulation T_I as $(DT(Q_i))_{i \in I}$, where Q_i corresponds to the local points P_i and their foreign Delaunay neighbors in T: $Q_i = P_i \cup \bigcup_{p \in P_i} Neigbors(p, T)$. T_i is then a local view of T, as it encodes the neighborhood of its local points: $Star(p, T) = Star(p, T_i)$ if $p \in P_i$.

Conversely, given a distributed triangulation T_I of a point-partitioning P_I, the overall triangulation T may be defined as the union of all local and mixed cells: $T = \bigcup_{i \in I} \bigcup_{p \in P_i} Star(p, T_i)$.

Delaunay Simplification: Since T_i is responsible to provide a local view of the overall triangulation through the stars of its local points $Star(p, T_i)$, with $p \in P_i$, its foreign simplices may be discarded without modifying the local stars. The proposed Delaunay simplification operator removes all foreign points that are not adjacent to any local point, and thus disjoint to all local stars. These are denoted as **redundant** foreign points. There might remain foreign simplices (so-called **auxiliary** simplices) which ensure that the simplified triangulations cover the convex hull of their point sets. We introduce $Simplify(T_i)$ the action of removing the non-adjacent foreign point (figure 2).

Algorithm 1: Tile and merge

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 ∈ P do in parallel
    i ← TileId(p)
    P_i ← P_i ⋃ {p}
// Local tile triangulations
for each tile i ∈ I do in parallel
    T_i^0 ← Delaunay(P_i)
    C_i^0, C̄_i^0 ← Finalize(T_i^0)
    P_i^0 ← ⋃_{c ∈ C̄_i^0} c
    S_{i→all}^0 ← ExtremePoints(P_i^0)
// Star splaying initialization
for each tile i ∈ I do in parallel
    R_i^1 ← ⋃_{j≠i} S_{j→all}^0
    P_i^1 ← P_i^0 ∪ R_i^1
    T_i^1 ← Simplify(Delaunay(P_i^1))
    (S_{i→j}^1)_{j≠i} ← StarSplay(R_i^1, T_i^1)
// Star splaying iterations
n ← 2
while ∃ i, j ∈ I such that S_{i→j}^n ≠ ∅ do
    for each tile i ∈ I do in parallel
        R_i^n ← (⋃_{j≠i} S_{j→i}^{n-1}) \ P_i^{n-1}
        P_i^n ← P_i^{n-1} ⋃ R_i^n
        T_i^n ← Simplify(DelaunayIns(R_i^n, T_i^{n-1}))
        (S_i^n)_{j≠i} ← StarSplay(R_i^n, T_i^n)
    n ← n + 1
for each tile i ∈ I do in parallel
    C_i^n, C̄_i^n ← Finalize(T_i^n)
    T_i^∞ = C_i^0 ∪ C̄_i^n
return T_I^∞ = (T_i^∞)_{i ∈ I}
```

B. Algorithm

The proposed algorithm may be decomposed into four main steps (see figure 1 for the overall algorithm and figure 3 for the pipeline overview).

First, the tiling step (line 2) is performed on the whole input point set P. Chunks of the input point set P are processed in parallel to produce partitions according to an oracle function $TileId(p)$, which is here limited to a grid partitioning (e.g., in 2D, $TileId(x, y) = (\lfloor \frac{x}{s} \rfloor, \lfloor \frac{y}{s} \rfloor)$, where s is the grid size). Shown in figure 1a but omitted in algorithm 1 and figure 3 for brevity, the grid-based tiles are then merged in a quadtree/octree procedure until they meet a point size limit.

Then the DT of each tile is performed (line 5). The output of this process is both a first set of finalized cells C_i^0 and sets of points P_i^0 that are incident to at least one non-finalized cell.

At this point, the initialization of the iterative scheme starts by broadcasting and inserting the extreme points of non finalized cells $S_{i \to all}^0$ with the non finalized points of each tile P_i^0. By simplifying each tile triangulation T_i^1, the remaining foreign points give a first approximation of the connections between tiles : points $S_{i \to j}^1$ are adjacent to points of tile j in the local stars of T_i^1, such that communication is required between i and j to make their common stars consistent.

We define $S_{i \to j} = \bigcup_{p_k \in T_i}$ the point set sent from tile i to tile j during the shuffling step such as $p_k \in (Star(T_i, p)\, \forall p \in T_i)_{k \neq j \wedge k \neq j}$. We define $S_i = \bigcup_j S_{i \to j}$ the set of point sent from tile i and $R_j = \bigcup_j S_{i \to j}$ the set of point receives at tile j. We finally introduce the function $StarSplay(T_i) = (S_{i \to j})_{j \neq i}$.

We say there is an **active** connection between two tiles at an iteration if $S_{i \to j}^n \neq \varnothing$ and **inactive** connection if $S_{i \to j}^n = \varnothing \wedge \exists m$ such as $\forall \ \ m < n, S_{j \to j}^m \neq \varnothing$. A special case is to gather the tile-triangulations into a single overall triangulation

by enumerating all local cells and deduplicated mixed cells (e.g., by only reporting the cells from the smallest primary tile of all its incident vertices). We call this cell **main** and note it C^f and introduce $Finalize(T)$ the function that extract the finalized cells.

Finally, the scheme iterates while the number of active connections is not null (line 15). The shuffled points received by each tile R_i^n are inserted into the current triangulation T_i^{n-1} followed by a simplification (line 19). The new exchanged point set $S_{i\to j}$ can be computed. In each tile, a list of points that has already been sent is saved. Thus, only points that has not been sent yet are shuffled (line 17). In this way, the number of inactive connections increases with respect to the number of iterations until the number of active connections being empty. At the end, main cells that have not been finalized during the first triangulation are dumped.

III. Evaluation

The proposed approach is evaluated on a Spark cluster with 28 cores and 100Go memory. [1]

In this section, we analyze the strong scaling ability that shows how the algorithm scales according to the cluster configuration (see figure 4). We use 3 different data sets: points generated from i) a normal distribution, ii) a uniform distribution and iii) from LiDAR acquisitions. 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.

The main parameter of the proposed approach is the maximum number of points per tile that is allowed during the tiling construction. According to the efficiency of the local DT implementation and the cost of the communication between executors 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. To handle a large amount of points per tile, we need to limit the number of simultaneous triangulations performed at the same time. To do that, the number of partitions of P is set to the number of tiles. It follows that a core will never process more that one triangulation at the same time without passing the result to the scheduler and limit the memory footprint.

A stress test is realized to evaluate the maximum number of points that can handle each tile without running out of memory. For that, we increase the number of points triangulated from a uniform distribution on a 4x4x4 grid (64 tiles) step by step. With a uniform distribution, we have an upper bound of the maximum number of points processed at the same time. In this configuration, after 3M points per tiles, memory issues appear.

To test the strong scaling capabilities of the proposed approach. A triangulation of 300M points with normal and uniform distribution is processed with different cloud configurations: a varying number of executors (7,4,2 and 1) and, for each executor, a varying number of cores (4,3,2,1) plus 1 executor with 1 core configuration for a fixed 12Go of RAM per executor in every cases. The total number of cores is thus the number of executors times the number of cores per executor. The result is shown on figure 4. As our application is fully distributed, we compare it to the perfect scale factor of 1. The scale factor is defined by $\frac{t_{1\text{core}}}{n * t_{n_{\text{core}}}}$.

With one core, it takes $9351s$ to process the 300M of points for the normal distribution data set and $18332s$ for the uniform. It gives around 1.9 and 1 million points processed per-minute. This is a very suitable result on the fact that the algorithm is tuned to work with less than 4Go of memory per core and performs multiple I/O.

For a fixed number of executors, the time constantly decreases with a minimum scale factor of 0.4 for the normal and 0.35 for the uniform distribution. It means that for a given number of cores, the time decreases by at least 0.35 times the number of cores. The scale factor globally increases after adding a third core. With four cores per executor, the scale factor remains stable and above 0.5 for every configuration. This result highlights the fact that adding only one executor to a single executor configuration has a significant cost while adding more that one executor re-increase the scale factor. The scale factor is the most important with 2 cores per executor. This is probably the consequence of a biggest memory to core ratio.

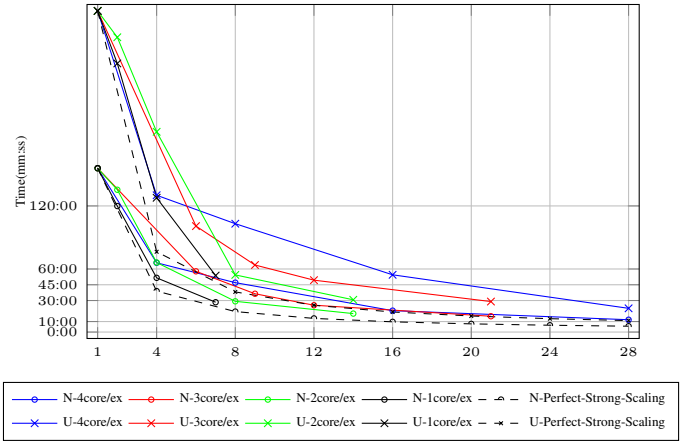


Fig. 4: Strong scaling on processing and persisting the DT according to the number of cores with 4,3 and 2 cores/executors (12Go Ram/executors) on a 300 millions of points test set generated with the normal(N) and uniform(U) random distribution.

IV. Results

The biggest point set processed with this approach is a 1.98 billion point cloud triangulation of a $6km^2$ with a $2cm$ spatial resolution in 2h20 on 28 cores configuration (figure 5). All the finalized simplices have been written on the HDFS after 4h11min for a total of 400Go ply files. The heterogeneous

[1] As a comparison, the *m5* series of the Amazon EMR service dedicated to general usage provides 4Go for 1 core. The *m5.4xlarge* setup is 16 cores for 64Go of RAM and the *m5.8xlarge* is 16 cores for 128Go of RAM for $\$0.8/h$ and $\$1.6/h$ respectively in August 2019.

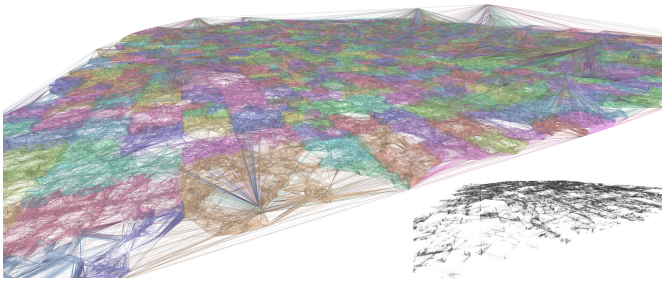


Fig. 5: Result of the proposed approach on 1.98 billion of points $6km^2$ with $2cm$ spatial resolution. This figure shows the whole scene. Only finalized cells during the last step are shown. Triangles of the same tile have the same color.

distribution of the point cloud implies a sparsity of the data set with high density area while other places are empty of point. This case is well handled by the octree structure that allows a good distribution of the point across the tiles.

V. Conclusion

This article presents a new approach for the distributed computation of the Delaunay triangulation. Scientific contributions are multiple: A new tiling and merge approach is proposed based on a distributed tiling representation of the triangulation. Each tile is guaranteed to provide a local view of the global triangulation. An iterative process based on a star splaying approach makes them consistent. This pipeline is fully distributed and then limit the peak memory footprint. This approach is experimentally validated with an implementation on the Spark Framework coupled with efficient C++ executables for computational geometry routines. It shows both good scaling according to an increasing number of cores and number of points. Unlike more hardware-dedicated algorithms, this framework can easily be deployed on a Spark cluster and integrates well a full production pipeline for distributed 3D surface reconstruction from LiDAR point clouds.

References

[AAY05] Agarwal P. K., Arge L., Yi K.: I/o-efficient construction of constrained delaunay triangulations. In *ESA* (2005), vol. 5, Springer, pp. 355–366.

[ATW13] Ando R., Thürey N., Wojtan C.: Highly adaptive liquid simulations on tetrahedral meshes. *ACM Trans. Graph. 32*, 4 (July 2013), 103:1–103:10.

[BBK06] Blandford D. K., Blelloch G. E., Kadow C.: Engineering a compact parallel delaunay algorithm in 3D. *Proceedings of the twenty-second annual symposium on Computational geometry - SCG 06* (2006).

[BMPS10] Batista V. H., Millman D. L., Pion S., Singler J.: Parallel geometric algorithms for multi-core computers. *Computational Geometry 43*, 8 (Oct 2010), 663–677.

[BY98] Boissonnat J.-D., Yvinec M.: *Algorithmic geometry*. Cambridge university press, 1998.

[CBV16] Caraffa L., Brédif M., Vallet B.: 3d watertight mesh generation with uncertainties from ubiquitous data. In *Proceedings of Asian Conference on Computer Vision (ACCV'16)* (Taipei, Taiwan, 2016), no. 7727 in LNCS, Springer.

[CCW06] Chen M.-B., Chuang T.-R., Wu J.-J.: Parallel divide-and-conquer scheme for 2D Delaunay triangulation. *Concurrency and Computation: Practice and Experience 18*, 12 (2006), 1595–1612.

[CLCN*18] Caíno-Lores S., Carretero J., Nicolae B., Yildiz O., Peterka T.: Spark-diy: A framework for interoperable spark operations with high performance block-based data models. In *BDCAT* (2018), IEEE Computer Society, pp. 1–10.

[FLP14] Fuetterling V., Lojewski C., Pfreundt F.-J.: High-performance delaunay triangulation for many-core computers. In *High Performance Graphics* (2014), pp. 97–104.

[FS17] Funke D., Sanders P.: Parallel d-d Delaunay triangulations in shared and distributed memory. *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)* (Jan 2017).

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

[HKLP09] Hiep V. H., Keriven R., Labatut P., Pons J.-P.: Towards high-resolution large-scale multi-view stereo. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on* (2009), IEEE, pp. 1430–1437.

[ILSS06] Isenburg M., Liu Y., Shewchuk J. R., Snoeyink J.: Streaming computation of Delaunay triangulations. *ACM SIGGRAPH 2006 Papers on - SIGGRAPH 06* (2006).

[KM09] Kaufmann O., Martin T.: 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 (2009), 70–82.

[LPK09] Labatut P., Pons J.-P., Keriven R.: Robust and Efficient Surface Reconstruction From Range Data. *Computer Graphics Forum* (2009).

[PMP14] Peterka T., Morozov D., Phillips C.: High-performance computation of distributed-memory parallel 3d voronoi and delaunay tessellation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE Press, pp. 997–1007.

[She05] Shewchuk J. R.: Star splaying: An algorithm for repairing Delaunay triangulations and convex hulls. In *Proceedings of the Twenty-first Annual Symposium on Computational Geometry* (New York, NY, USA, 2005), SCG '05, ACM, pp. 237–246.

[SKRC10] Shvachko K., Kuang H., Radia S., Chansler R.: The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[SOJ14] Starinshak D., Owen J., Johnson J.: A new parallel algorithm for constructing voronoi tessellations from distributed input data. *Computer Physics Communications 185*, 12 (Dec 2014), 3204–3214.

[TO00] Tekalp A. M., Ostermann J.: Face and 2-d mesh animation in mpeg-4. *Signal Processing: Image Communication 15*, 4 (2000), 387–421.

[Vit01] Vitter J. S.: External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys 33*, 2 (Jun 2001), 209–271.

[WMRL17] Wang Y., Ma G., Ren F., Li T.: A constrained delaunay discretization method for adaptively meshing highly discontinuous geological media. *Computers & Geosciences 109* (2017), 134–148.

[XY03] Xiao Y., Yan H.: Text region extraction in a document image based on the delaunay tessellation. *Pattern Recognition 36*, 3 (2003), 799–809.

[ZWa19] Zonghan Wu and Shirui Pan and F. C. a. G. L. a. C. Z. a. P. S. Y.: A comprehensive survey on graph neural networks. *CoRR* (2019).

[ZXW*16] 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.: Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.