Received: 00 Month 0000 | Revised: 00 Month 0000 | Accepted: 00 Month 0000

DOI: xxx/xxxx

ARTICLE TYPE

One machine, one minute, three billion tetrahedra

Célestin Marot* | Jeanne Pellerin | Jean-François Remacle

[1]Université catholique de Louvain, iMMC, Avenue Georges Lemaitre 4, bte L4.05.02, 1348 Louvain-la-Neuve, Belgium

Correspondence
*Corresponding author: Email: celestin.marot@uclouvain.be

Summary

This paper presents a new scalable parallelization scheme to generate the 3D Delaunay triangulation of a given set of points. Our first contribution is an efficient serial implementation of the incremental Delaunay insertion algorithm. A simple dedicated data structure, an efficient sorting of the points and the optimization of the insertion algorithm have permitted to accelerate reference implementations by a factor three. Our second contribution is a multi-threaded version of the Delaunay kernel that is able to concurrently insert vertices. Moore curve coordinates are used to partition the point set, avoiding heavy synchronization overheads. Conflicts are managed by modifying the partitions with a simple rescaling of the space-filling curve. The performances of our implementation have been measured on three different processors, an Intel core-i7, an Intel Xeon Phi and an AMD EPYC, on which we have been able to compute 3 billion tetrahedra in 53 seconds. This corresponds to a generation rate of over 55 million tetrahedra per second. We finally show how this very efficient parallel Delaunay triangulation can be integrated in a Delaunay refinement mesh generator which takes as input the triangulated surface boundary of the volume to mesh.

KEYWORDS:
3D Delaunay triangulation; Tetrahedral mesh generation; Parallel Delaunay; Radix sort; SFC partitioning

1 | INTRODUCTION

The Delaunay triangulation is a fundamental geometrical object that associates a unique triangulation to a given point-set in general position. This triangulation and its dual, the Voronoi diagram, have locality properties that make them ubiquitous in various domains[1]: mesh generation[2], surface reconstruction from 3D scanners point clouds[3], astrophysics[4,5], terrain modeling[6] etc. Delaunay triangulation algorithms now face a major challenge: the availability of more and more massive point sets. LiDAR or other photogrammetry technologies are used to survey the surface of entire cities and even countries, like the Netherlands[7]. The size of finite element mesh is also growing with the increased availability of massively parallel numerical solvers. It is now common to deal with meshes of over several hundred millions of tetrahedra[8–10]. Parallelizing 3D Delaunay triangulations and Delaunay-based meshing algorithms is however very challenging and the mesh generation process is nowadays considered a technological bottleneck in computational engineering[11].

The most part of today's clusters have two levels of parallelism. Distributed memory systems contain thousands of nodes, each node being itself a shared memory system with multiple cores. In recent years, nodes have seen their number of cores and the size of their memory increase, with some clusters already featuring 256-core processors and up to 12TB of RAM[12,13]. As many-core shared memory machines are becoming standard, Delaunay triangulation algorithms designed for shared memory should not only scale well up to 8 cores but to several hundred cores. In this paper, we complete the approach presented in

our previous research note[14] and show that a billion tetrahedra can be computed very efficiently on a single many-core shared memory machine.

Our first contribution is a sequential implementation of the Delaunay triangulation algorithm in 3D that is able to triangulate a million points in about 2 seconds. In comparison, the fastest 3D open-source sequential programs (to our best knowledge): Tetgen[15], CGAL[16] and Geogram[17] all triangulate a million points in about 6 seconds on one core of a high-end laptop. Our implementation is also based on the incremental Delaunay insertion algorithm, but the gain in performance is to ascribe to the details of the specific data structures we have developed, to the optimization of geometric predicates evaluation, and to specialized adjacency computations (Section 2).

Our second contribution is a scalable parallel version of the Delaunay triangulation algorithm devoid of heavy synchronization overheads (Section 3). The domain is partitioned using the Hilbert curve and conflicts are detected with a simple coloring scheme. The performances and scalability are demonstrated on three different machines: a high-end four core laptop, a 64-core Intel® Xeon Phi Knight's Landing, and a recent AMD® EPYC 64-core machine (Section 3.7). On the latter computer, we have been able to generate three billion tetrahedra in less than a minute (about 10^7 points per second).

We finally demonstrate how this efficient Delaunay triangulation algorithm can be easily integrated in a tetrahedral mesh generation process where the input is the boundary surfaces of the domain to mesh (Section 4).

Our reference implementation is open-source and available in Gmsh 4 at http://gmsh.info.

2 | SEQUENTIAL DELAUNAY

The Delaunay triangulation DT(S) of a point set S has the fundamental geometrical property that the circumsphere of any tetrahedron contains no other point of S than those of the considered tetrahedron. More formally, a triangulation[1] $T(S)$ of the n points $S = \{p_1, \dots, p_n\} \in \mathbb{R}^3$ is a set of non overlapping tetrahedra that covers exactly the convex hull $\Omega(S)$ of the point set, and leaves no point p_i isolated. If the empty circumsphere condition is verified for all tetrahedra, the triangulation $T(S)$ is said to be a Delaunay triangulation. If, additionally, S contains no group of 4 coplanar points and no group of 5 cospherical points, then it is said to be in general position, and the Delaunay triangulation DT(S) is unique.

The fastest serial algorithm to build the 3D Delaunay triangulation DT(S) is probably the Bowyer-Watson algorithm, which works by incremental insertion of points in the triangulation. The Bowyer-Watson algorithm, presented in (§2.1), was devised independently by Bowyer and Watson in 1981[18, 19]. Efficient open-source implementations are available: Tetgen[15], CGAL[16] and Geogram[17]. They are designed similarly and offer therefore similar performances (Table 1).

	Ours	Geogram	TetGen	CGAL
SEQUENTIAL_DELAUNAY	**12.7**	**34.6**	**32.9**	**33.8**
INIT + SORT	0.5	4.2	2.1	1.3
INCREMENTAL INSERTION	12.2	30.4	30.8	32.5
WALK	1.0	2.1	1.6	1.4
`orient3d`	0.7	1.4	1.1	≈ 0.5
CAVITY	6.2	11.4	≈ 10	14.9
`inSphere`	3.2	6.2	5.6	10.5
DELAUNAYBALL	4.5	12.4	≈ 15	15.3
Computing sub-determinants	1.3	/	/	/
Other operations	0.5	4.5	≈ 4	≈ 1

TABLE 1 Timings for the different steps of the Delaunay incremental insertion (Algorithm 1) for four implementations: Ours, Geogram[17], Tetgen[15] and CGAL[16]. Timings in seconds are given for 5 million points (random uniform distribution). The $\approx$ prefix indicates that no accurate timing is available.

[1]This paper is about 3D meshing. Still we use the generic term triangulation instead of tetrahedralization.

In the remaining of this section, the incremental insertion algorithm is recalled and we describe the dedicated data structures that we developed as well as the algorithmic optimizations that make our sequential implementation three times faster than reference ones.

2.1 | Algorithm overview

Let DT_k be the Delaunay triangulation of the subset $S_k = \{p_1, \dots, p_k\} \subset S$. The *Delaunay kernel* is the procedure to insert a new point $p_{k+1} \in \Omega(S_k)$ into DT_k, and construct a new valid Delaunay triangulation DT_{k+1} of $S_{k+1} = \{p_1, \dots, p_k, p_{k+1}\}$. The *Delaunay kernel* can be written in the following abstract manner:

$$\mathrm{DT}_{k+1} \leftarrow \mathrm{DT}_k - \mathcal{C}(\mathrm{DT}_k, p_{k+1}) + \mathcal{B}(\mathrm{DT}_k, p_{k+1}), \tag{1}$$

where the Delaunay cavity $\mathcal{C}(\mathrm{DT}_k, p_{k+1})$ is the set of all tetrahedra whose circumsphere contains p_{k+1} (Figure 1b), whereas the Delaunay ball $\mathcal{B}(\mathrm{DT}_k, p_{k+1})$ is a set of tetrahedra filling up the polyhedral hole obtained by removing the Delaunay cavity $\mathcal{C}(\mathrm{DT}_k, p_{k+1})$ from DT_k (Figure 1c).

The complete state-of-the-art incremental insertion algorithm (Algorithm 1) that we implemented has five main steps that are described in the following.

Init

The triangulation is initialized with the tetrahedron formed by the first four non-coplanar vertices of the point set S. These vertices define a tetrahedron τ with a positive volume.

Sort

Before starting point insertion, the points are sorted so that two points that have close indices are close in space. Used alone, this order would result in cavities $\mathcal{C}(\mathrm{DT}_k, p_{k+1})$ containing a pathologically large number of tetrahedra. Insertions are organized in different randomized stages to avoid this issue[20]. The three kernel functions Walk, Cavity and DelaunayBall thereby have a constant complexity in practice. We have implemented a very fast sorting procedure (Section 2.3).

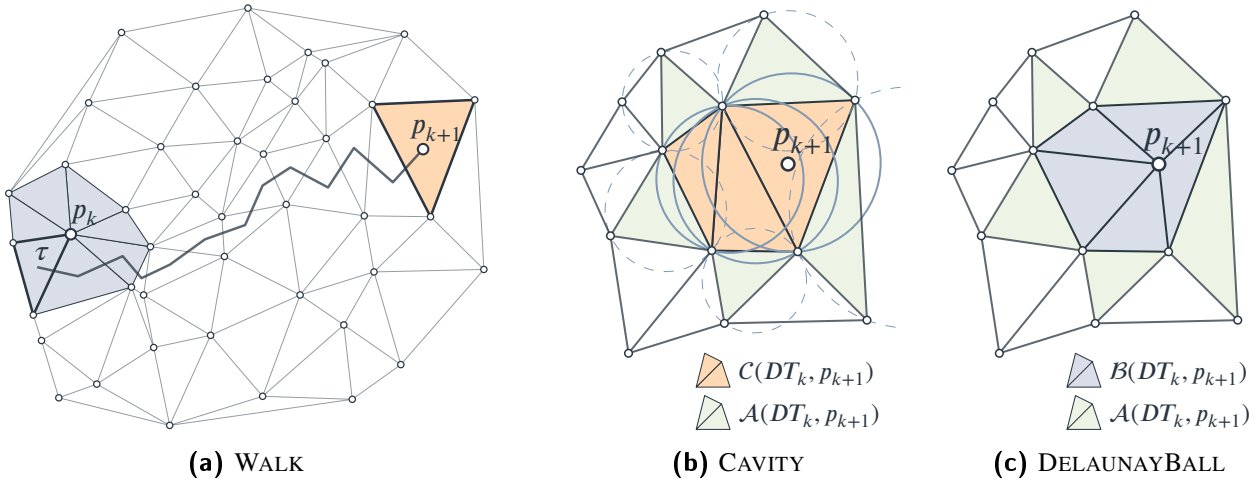
Walk

The goal of this step is to identify the tetrahedron τ_{k+1} enclosing the next point to insert p_{k+1}. The search starts from a tetrahedron τ_k in the last Delaunay ball $\mathcal{B}(\mathrm{DT}_k, p_k)$, and walks through the current triangulation DT_k in direction of p_{k+1} (Figure 1a). We say that a point is visible from a facet when the tetrahedron defined by this facet and this point has negative volume. The Walk function thus iterates on the four facets of τ, selects one from which the point p_{k+1} is visible, and then walks across this facet to

Algorithm 1 Sequential computation of the Delaunay triangulation DT of a set of vertices S

Input: S
Output: $\mathrm{DT}(S)$ ▷ Section 2.2

```
1:  function SEQUENTIAL_DELAUNAY(S)
2:      τ ← INIT(S)                                  ▷ τ is the current tetrahedron
3:      DT ← τ
4:      S' ← SORT(S \ τ)                             ▷ Section 2.3
5:      for all p ∈ S' do
6:          τ ← WALK(DT, τ, p)
7:          C ← CAVITY(DT, τ, p)                     ▷ Section 2.4
8:          DT ← DT \ C
9:          B ← DELAUNAYBALL(C, p)                   ▷ Section 2.5
10:         DT ← DT ∪ B
11:         τ ← t ∈ B
12:     end for
13:     return DT
14: end function
```



(a) WALK **(b)** CAVITY **(c)** DELAUNAYBALL

FIGURE 1 Insertion of a vertex p_{k+1} in the Delaunay triangulation DT_k. **(a)** The triangle containing p_{k+1} is obtained by walking toward p_{k+1}. The WALK starts from $\tau \in \mathcal{B}_{p_k}$. **(b)**The CAVITY function finds all cavity triangles (orange) whose circumcircle contains the vertex p_{k+1}. They are deleted, while cavity adjacent triangles (green) are kept. **(c)** The DELAUNAYBALL function creates new triangles (blue) by connecting p_{k+1} to the edges of the cavity boundary.

the adjacent tetrahedron. This new tetrahedron is called τ and the WALK process is repeated until none of the facets of τ sees p_{k+1}, which is equivalent to say that p_{k+1} is inside τ (see Figure 1a).

The visibility walk algorithm is guaranteed to terminate for Delaunay triangulations[21]. If the points have been sorted, the number of walking steps is essentially constant[22]. Our implementation of this robust and efficient walking algorithm is similar to other available implementations.

CAVITY

Once the tetrahedron $\tau \leftarrow \tau_{k+1}$ that contains the point to insert p_{k+1} has been identified, the function CAVITY finds all tetrahedra whose circumsphere contain p_{k+1} and deletes them. The Delaunay cavity $\mathcal{C}(\mathrm{DT}_k, p_{k+1})$ is simply connected and contains τ[23], it is then built using a breadth-first search algorithm. The core and most expensive operation of the CAVITY function is the `inSphere` predicate, which evaluates whether a point e is inside/on or outside the circumsphere of given tetrahedron. This CAVITY function is thus an expensive function of the incremental insertion, which accounts for about 33% of the total computation time (Table 1). To accelerate this, we propose in Section 2.4 to precompute sub-components of the `inSphere` predicate.

DELAUNAYBALL

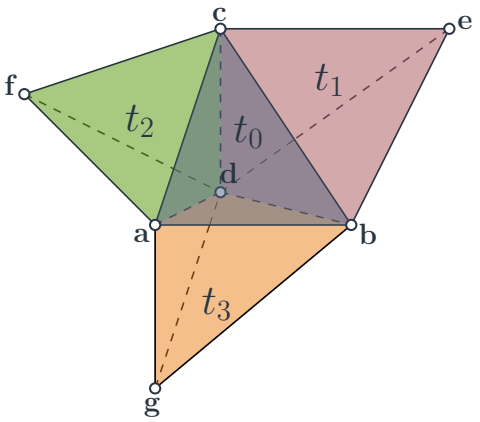
Once the cavity has been carved, the DELAUNAYBALL function first generates a set of new tetrahedra adjacent to the newly inserted point p_{k+1} and filling up the cavity, and then updates the mesh structure. In particular, the mesh update consists in the computation of adjacencies between the newly created tetrahedra. This is the most expensive step of the algorithm, with about 40% of the total computation time (Table 1). To accelerate this step, we replace general purpose elementary operations like tetrahedron creation/deletion or adjacency computation, with batches of optimized operations making benefit from a cavity-specific data structure (Section 2.5).

2.2 | Mesh data structure

One key for enhancing performances of a Delaunay triangulation algorithm resides in the optimization of the data structure used to store the triangulation. Various data structure designs have been proposed that are very flexible and allow representing hybrid meshes, high order meshes, add mesh elements of any type, or manage adjacencies around vertices, edges, etc. The versatility of such general purpose data structures has a cost, both in terms of storage and efficiency. Here, our aim is to have a structure as lightweight and fast as possible, dealing exclusively with 3D triangulations. Our implementation is coded in plain `C` language, with arrays of doubles, floats, and integers to store mesh topology and geometry. This seemingly old-style coding has important advantages in terms of optimization and parallelization because it compels us to use simple and straightforward algorithms.

```
typedef struct {
    double coordinates[3];
    uint64_t padding;
} point3d_t;

typedef struct {
    struct {
        uint32_t* vertex_ID;
        uint64_t* neighbor_ID;
        double* sub_determinant;
        uint64_t num;             // number of tetrahedra
        uint64_t allocated_num;   // capacity [in tetrahedra]
    } tetrahedra;

    struct {
        point3d_t* vertex;
        uint32_t num;            // number of vertices
        uint32_t allocated_num; // capacity [in vertices]
    } vertices;
} mesh_t;
```

Listing 1: The aligned mesh data structure `mesh_t` we use to store the vertices and tetrahedra of a Delaunay triangulation in 3D.



memory index	`vertex_ID`	`neighbor_ID`
$4t_0$	a	$4t_1+3$
$4t_0+1$	b	$4t_2+3$
$4t_0+2$	c	$4t_3+3$
$4t_0+3$	d	–
⋮		
$4t_1$	b	–
$4t_1+1$	c	–
$4t_1+2$	d	–
$4t_1+3$	e	$4t_0+0$
⋮		
$4t_2$	a	–
$4t_2+1$	d	–
$4t_2+2$	c	–
$4t_2+3$	f	$4t_0+1$
⋮		
$4t_3$	a	–
$4t_3+1$	b	–
$4t_3+2$	d	–
$4t_3+3$	g	$4t_0+2$

FIGURE 2 Four adjacent tetrahedra : t_0, t_1, t_2, t_3 and one of their possible memory representations in the `tetrahedra` data structure given in Listing 1. `tetrahedra.neighbor_ID`$[4t_i + j]/4$ gives the index of the adjacent tetrahedron opposite to `tetrahedra.vertex_ID`$[4t_i + j]$ in the tetrahedron t_i and `tetrahedra.neighbor_ID`$[4t_i + j]$ gives the index where the inverse adjacency is stored.

The mesh data only contains vertices and tetrahedra explicitly, and all topological and geometrical information can be deduced from it. However, in order to speed up mesh operations, it is beneficial to store additional connectivity information. A careful trade-off needs however to be made between computational time and memory space. The only connectivity information we have chosen to store is the adjacency between tetrahedra, as this allows walking through the mesh using local queries only.

Vertices Vertices are stored in a single array of structures `point3d_t` (see Listing 1). For each vertex, in addition to the vertex coordinates, a `padding` variable is used to align the structure to 32 bytes (3 doubles of 8 bytes each, and an additional padding variable of 8 bytes sum up to a structure of 32 bytes) and conveniently store temporarily some auxiliary vertex related values at different stages of the algorithm. Memory alignment ensures that a vertex does not overlap two cache lines during memory transfer. Modern computers usually work with cache lines of 64 bytes. The padding variable in the vertex structure ensures that a vertex is always loaded in one single memory fetch. Moreover, aligned memory allows to take advantage of the vectorization capabilities of modern microprocessors.

Tetrahedra Each tetrahedron knows about its 4 vertices and its 4 neighboring tetrahedra. These two types of adjacencies are stored in separate arrays. The main motivation for this storage is flexibility and, once more, memory alignment. Keeping good memory alignment properties on a tetrahedron structure evolving with the implementation is cumbersome. In addition, it provides little to no performance gain in this case. On the other hand with parallel arrays, additional information per tetrahedron (e.g. a color for each tetrahedron, sub-determinants etc.) can be added easily without disrupting memory layout. Each tetrahedron is identified by the indices of its four vertices in the `vertices` structure. Vertex indices of tetrahedron `t` are read between positions `4*t` and `4*t+3` in the global array `tetrahedra.vertex_ID` storing all tetrahedron vertices. Vertices are ordered so that the volume of the tetrahedron is positive. An array of double, `sub_determinant`, is used to store 4 values per tetrahedron. This space is used to speed up geometric predicate evaluation (see Section 2.4).

Adjacencies By convention, the i-th facet of a tetrahedron is the facet opposite the i-th vertex, and the i-th neighbor is the tetrahedron adjacent to that facet. In order to travel efficiently through the triangulation, facet indices are stored together with the indices of the corresponding adjacent tetrahedron, thanks to an integer division and its modulo. The scheme is simple. Each adjacency is represented by the integer obtained by multiplying by four the index of the tetrahedron and adding the internal index of the facet in the tetrahedron. Take, for instance, two tetrahedra t_1 and t_2 sharing facet f, whose index is respectively i_1 in t_1 and i_2 in t_2. The adjacency is then represented as `tetrahedra.neighbor_ID[4t`$_1$`+i`$_1$`]=4t`$_2$`+i`$_2$ and `tetrahedra.neighbor_ID[4t`$_2$`+i`$_2$`]=4t`$_1$`+i`$_1$. This multiplexing avoids a costly looping over the facets of a tetrahedron. The fact that it reduces by a factor 4 the maximum number of elements in a mesh is not a real concern since, element indices and adjacencies being stored as 64 bytes unsigned integers, the maximal number of element in a mesh is $2^{62} \simeq 4.6\ 10^{18}$, which is huge. Note also that division and modulo by 4 are very cheap bitwise operations for unsigned integers: $i/4 = i \gg 2$ and $i\%4 = i\&3$.

Memory footprint The key to an efficient data structure is the balance between its memory footprint and the computational cost of its modification. We do not expect to generate meshes of more than `UINT32_MAX` vertices, i.e. about 4 billion vertices on one single machine. Each vertex therefore occupies 32 bytes, 24 bytes for its coordinates and 8 bytes for the padding variable. On the other hand, the number of tetrahedra could itself be larger than 4 billion, so that a 64 bits integer is needed for element indexing. Each tetrahedron occupies 80 bytes, $4 \times 4 = 16$ bytes for the vertices, $4 \times 8 = 32$ bytes for the neighbors, 32 bytes again for the sub-determinants. In average a tetrahedral mesh of n vertices has a little more than $6n$ tetrahedra. Thus, our mesh data structure requires $\approx 6 \times 80 + 32 = 512$ bytes per vertex.

2.3 | Fast spatial sorting

The complexity of the Delaunay triangulation algorithm depends on the number of tetrahedra traversed during the walking steps, and on the number of tetrahedra in the cavities. An appropriate spatial sorting is required to improve locality, i.e., reduce the walking steps, while retaining enough randomness to have cavity sizes independent of the number of vertices already inserted in the mesh (by cavity size, we mean the number of tetrahedra in the cavity, not the volume of the cavity).

An efficient spatial sorting scheme has been proposed by Boissonnat et al.[24] and is used in state-of-the-art Delaunay triangulation implementations. The main idea is to first shuffle the vertices, and to group them in rounds of increasing size (a first round with, e.g., the first 1000 vertices, a second with the next 7000 vertices, etc.) as described by the Biased Randomized Insertion Order (BRIO)[20]. Then, each group is ordered along a space-filling curve. The space-filling curve should have the property that successive points on the curve are geometrically close to each other. With this spatial sorting, the number of walking steps between two successive cavities remains small and essentially constant[22]. Additionally, proceeding by successive rounds according to the BRIO algorithm tends to reduce the average size of cavities.

The Hilbert curve is a continuous self-similar (fractal) curve. It has the interesting property to be space-filling, i.e., to visit exactly once all the cells of a regular grid with $2^m \times 2^m \times 2^m$ cells, $m \in \mathbb{N}$. A Moore curve is a closed Hilbert curve. Hilbert and Moore curves have the sought spatial locality properties, in the sense that points close to each other on the curve are also close to each other in $\mathbb{R}^3$ [25, 26]. Any 3D point set can be ordered by a Hilbert/Moore curve according to the order in which the curve visits the grid cells that contains the points. The Hilbert/Moore index is thus an integer value $d \in \{0, 1, 2, \ldots, 2^{3m} - 1\}$, and several points might have the same index. The bigger m is for a given point set, the smaller the grid cells are, and hence the lower the probability of having two points with the same Hilbert/Moore index. Given a 3D point set with n points, there are in average $n/2^{3m}$ points per grid cell. Therefore, choosing $m = k\ log_2(n)$ with k constant ensures the average number of points in grid cells to be independent of n. If known in advance, the minimum mesh size can also be taken into account: m can be chosen such that a grid cell never contain more than a certain number of vertices, thus capturing non-uniform meshes more precisely.

For Delaunay triangulation purposes, the objective is both to limit the number of points with the same indices (in the same grid cell) and to have indices within the smallest range as possible to accelerate subsequent sorting operations. There is thus a balance to find, so that Hilbert indices are neither too dense nor too sparse.

Computing the Hilbert/Moore index of one cell is a somewhat technical point. The Hilbert/Moore curve is indeed fractal, which means recursively composed of replicas of itself that have been scaled down by a factor two, rotated and optionally reflected. Those reflections and rotations can be efficiently computed with bitwise operations. Various transformations can be applied to point coordinates to emulate non-regular grids, a useful functionality we will resort to in Section 3.3. Hamilton gives extensive details on how to compute Hilbert indices[27] and we provide open-source implementation.

Once the Hilbert/Moore indices have been computed, points can be sorted accordingly in a data structure where the Hilbert/-Moore index is the key, and the point the associated value. An extremely well suited strategy to sort bounded integer keys is the radix sort[28–31], a non-comparative sorting algorithm working digit by digit. The base of the digits, the radix, can be freely chosen. Radix sort has a $O(wn)$ computational complexity, where n is the number of $\{key, value\}$ pairs and w the number of digits (or bits) of the keys. In our case, the radix sort has a complexity in $O(mn) = O(n\,log(n))$, where n is the number of points and $m = k\,log_2(n)$ the number of levels of the Hilbert/Moore grid. In general, m is small because a good resolution of the space-filling curve is not needed. Typically, the maximum value among keys is lower than the number of values to be sorted. We say that keys are *short*. Radix-sort is able to sort such keys extremely quickly.

Literature is abundant on parallel radix sorting and impressing performances are obtained on many-core CPUs and GPus[32–37]. However, implementations are seldom available and we were not able to find a parallel radix sort implementation properly designed for many-core CPUs. We implemented `HXTSort`, that is available independently as open source at https://www.hextreme.eu/hxtsort. Figure 3 compares the performances of `HXTSort` with `qsort`, `std::sort` and the most efficient implementations that we are aware of for sorting 64-bit key and value pairs. Our implementation is fully multi-threaded and takes advantage of the vectorization possibilities offered by modern computers such as the AVX512 extensions on the Xeon PHI. It has been developed primarily for sorting Hilbert indices, which are typically short. We use different strategies depending on the key bit size. These are the reason why `HXTSort` outperforms the Boost Sort Library, GNU's and Intel's parallel implementation of the standard library and Intel TBB when sorting Hilbert indices.

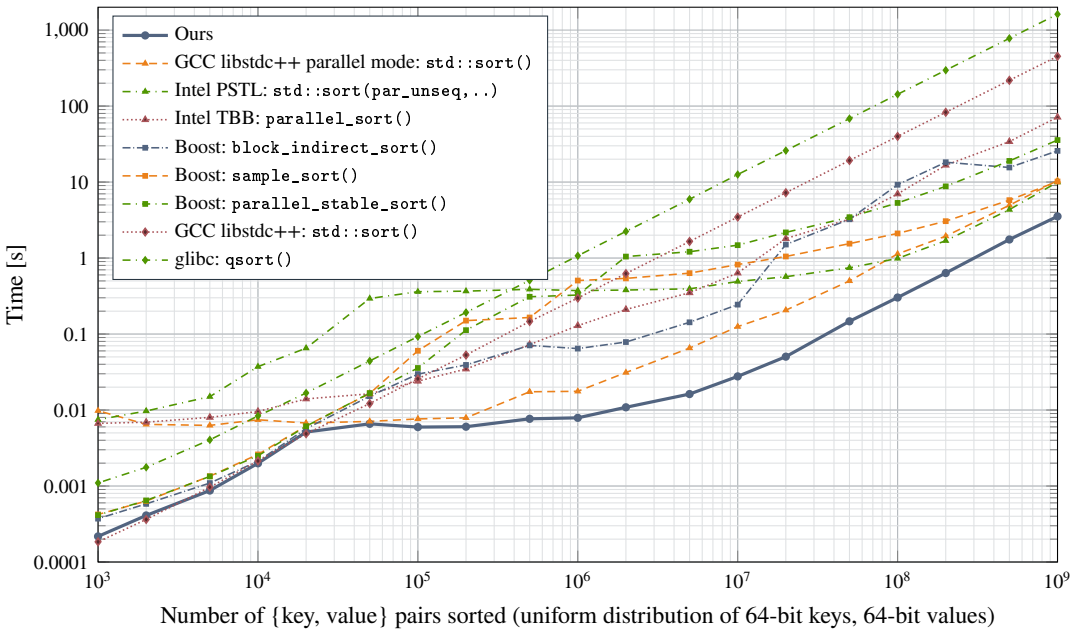


FIGURE 3 Performances of `HXTSort` for sorting $\{key, value\}$ pairs on an Intel® Xeon Phi™ 7210 CPU and comparison with widely used implementations.

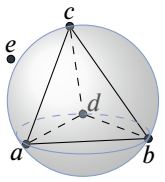
2.4 | Improving CAVITY: spending less time in geometric predicates

The first main operation of the Delaunay kernel is the construction of the cavity $\mathcal{C}(\mathrm{DT}_k, p_{k+1})$ formed by all the tetrahedra $\{a, b, c, d\}$ whose circumscribed sphere encloses p_{k+1} (Figure 1). This step of the incremental insertion represents about one third of the total execution time in available implementations (Table 1). The cavity is initiated with a first tetrahedron τ containing p_{k+1} determined with the WALK function (Algorithm 1 and Figure 1a), and then completed by visiting the neighboring tetrahedra with a breadth-first search algorithm.

Optimization of the `inSphere` predicate

The most expensive operation of the CAVITY function is the fundamental geometrical evaluation of whether a given point e is inside, exactly on, or outside the circumsphere of a given tetrahedra $\{a, b, c, d\}$ (Table 1). This is evaluated using the `inSphere` predicate that computes the sign of the following determinant:

$$\texttt{inSphere}(a,b,c,d,e) = \begin{vmatrix} a_x & a_y & a_z & \|a\|^2 & 1 \\ b_x & b_y & b_z & \|b\|^2 & 1 \\ c_x & c_y & c_z & \|c\|^2 & 1 \\ d_x & d_y & d_z & \|d\|^2 & 1 \\ e_x & e_y & e_z & \|e\|^2 & 1 \end{vmatrix} = \begin{vmatrix} b_x - a_x & b_y - a_y & b_z - a_z & \|b-a\|^2 \\ c_x - a_x & c_y - a_y & c_z - a_z & \|c-a\|^2 \\ d_x - a_x & d_y - a_y & d_z - a_z & \|d-a\|^2 \\ e_x - a_x & e_y - a_y & e_z - a_z & \|e-a\|^2 \end{vmatrix}$$



This is a very time consuming computation, and to make it more efficient, we propose to expand the 4×4 determinant into a linear combination of four 3×3 determinants independent of point e.

$$\begin{aligned} \texttt{inSphere}(a,b,c,d,e) = \quad & -(e_x - a_x) \begin{vmatrix} b_y - a_y & b_z - a_z & \|b-a\|^2 \\ c_y - a_y & c_z - a_z & \|c-a\|^2 \\ d_y - a_y & d_z - a_z & \|d-a\|^2 \end{vmatrix} + (e_y - a_y) \begin{vmatrix} b_x - a_x & b_z - a_z & \|b-a\|^2 \\ c_x - a_x & c_z - a_z & \|c-a\|^2 \\ d_x - a_x & d_z - a_z & \|d-a\|^2 \end{vmatrix} \\ & -(e_z - a_z) \begin{vmatrix} b_x - a_x & b_y - a_y & \|b-a\|^2 \\ c_x - a_x & c_y - a_y & \|c-a\|^2 \\ d_x - a_x & d_y - a_y & \|d-a\|^2 \end{vmatrix} + \|e-a\|^2 \begin{vmatrix} b_x - a_x & b_y - a_y & b_z - a_z \\ c_x - a_x & c_y - a_y & c_z - a_z \\ d_x - a_x & d_y - a_y & d_z - a_z \end{vmatrix} \end{aligned}$$

Being completely determined by the tetrahedron vertex coordinates, the four 3×3 determinants can be pre-computed and stored in the tetrahedron data structure when it is created. The cost of the `inSphere` predicate becomes then negligible. Notice also that the fourth sub-determinant is minus the tetrahedron volume. We can set it to a positive value to flag deleted tetrahedra during the breadth-first search, thereby saving memory space.

The maximal improvement of the CAVITY function obtained with this optimization depends on the number of times the `inSphere` predicate is invoked per tetrahedron. First, in order to simplify further discussion, we will assume that the number of tetrahedra is about 6 times the number of vertices in the final triangulation[38]. This means that each point insertion results in average in the creation of 6 new tetrahedra. On the other hand, we have seen in Section 2.3 that an appropriate point ordering ensures an approximately constant number of tetrahedra in the cavities. This number is close to 20 in a usual mesh generation context[22]. One point insertion thus results in the deletion of 20 tetrahedra, and the creation of 26 tetrahedra (all figures are approximations). This number is also the number of triangular faces of the cavity, since all tetrahedra created by the DELAUNAYBALL function associate a cavity facet to the inserted vertex p_{k+1}. The `inSphere` predicate is therefore evaluated positively for the 20 tetrahedra forming the cavity and negatively for the 26 tetrahedra adjacent to the faces of the cavity, a total of 46 calls for each vertex insertion.

When n points have been inserted in the mesh, a total of $26n$ tetrahedra were created and the predicate has been evaluated $46n$ times. Thus, we may conclude from this analysis that the `inSphere` predicate is called approximately $46n/26n = 1.77$ times per tetrahedron. In consequence, the maximal improvement that can be obtained from our optimization of the `inSphere` predicate is of $1 - 1/1.77 = 43\%$. Storing the sub-determinants has a memory cost (4 `double` values per tetrahedron) and a bandwidth cost (loading and storing of sub-determinants). For instance, for $n = 4.10^6$, we observe a speedup of 32% in the `inSphere` predicate evaluations, taking into account the time spent to compute the stored sub-determinants.

Note that a second geometric predicate is extensively used in Delaunay triangulation. The `orient3d` predicate evaluates whether a point d is above/on/under the plane defined by three points $\{a, b, c\} \in \mathbb{R}^3$[39]. It computes the triple product $(b-a) \cdot ((c-a) \times (d-a))$, i.e. the signed volume of tetrahedron $\{a, b, c, d\}$. This predicate is mostly used in the WALK function.

Implementation details

Geometrical predicates evaluated with standard floating point arithmetics may lead to inaccurate or inconsistent results. To have a robust and efficient implementations of the `inSphere` and `orient3d` predicates, we have applied the strategy implemented in Tetgen[15].

1. Evaluate first the predicate with floating point precision. This gives a correct value in the vast majority of the cases, and represents only 1% of the code.

2. Use a filter to check whether the obtained result is certain. A static filter is first used, then, if more precision is needed, a dynamic filter is evaluated. If the result obtained by standard arithmetics is not trustworthy, the predicate is computed with exact arithmetics[39].

3. To be uniquely defined, Delaunay triangulations requires each point to be inside/outside a sphere, under/above a plane. When a point is exactly on a sphere or a plane, the point is in a non-general position that is slightly perturbed to "escape" the singularity. We implemented the symbolic perturbations proposed by Edelsbrunner[40].

2.5 | Improving DELAUNAYBALL: spending less time computing adjacencies

The DELAUNAYBALL function creates the tetrahedra filling the cavity (Figure 1c), and updates the tetrahedron structures. In particular, tetrahedron adjacencies are recomputed. This is the most expensive step of the Delaunay triangulation algorithm as it typically takes about 40% of the total time (Table 1).

In contrast to existing implementations, we strongly interconnect the cavity building and cavity retriangulation steps. Instead of relying on a set of elegant and independent elementary operations like tetrahedron creation/deletion or adjacency computation, a specific cavity data structure has been developed, which optimizes batches of operations for the specific requirements of the Delaunay triangulation (Listing 2).

Use cavity building information The tetrahedra reached by the breadth-first search during the CAVITY step (Section 2.4), are either inside or adjacent to the cavity. Each triangular facet of the cavity boundary is thus shared by a tetrahedron $t_1 \in \mathcal{A}(\mathrm{DT}_k, p_{k+1})$ outside the cavity, by a tetrahedron $t_2 \in C(\mathrm{DT}_k, p_{k+1})$ inside the cavity, and by a newly created tetrahedron inside the cavity $t_3 \in \mathcal{B}(\mathrm{DT}_k, p_{k+1})$ (Figure 1). The facet of t_1 adjacent to the surface of the cavity defines, along with the point p_{k+1}, the tetrahedron t_3. We thus know a priori that t_3 is adjacent to t_1, and store this information in the `cavityBoundaryFacet_t` structure (Listing 2).

Fast adjacencies between new tetrahedra During the cavity construction procedure, the list of vertices on the cavity as well as adjacencies with the tetrahedra outside the cavity are computed. The last step is to compute the adjacencies between the new tetrahedra built inside the cavity, i.e. the tetrahedra of the Delaunay ball.

The first vertex of all created tetrahedra is set to be p_{k+1}, whereas the other three vertices $\{p_1, p_2, p_3\}$ are on the cavity boundary, and are ordered so that the volume of the tetrahedral element is positive, i.e., $\texttt{orient3d}(p_{k+1}, p_1, p_2, p_3) > 0$. As explained in the previous section, the adjacency stored at index $4t_i + 0$, which corresponds to the facet of tetrahedron t_i opposite to the vertex p_{k+1}, is already known for every tetrahedron t_i in $\mathcal{B}(\mathrm{DT}_k, p_{k+1})$. Three neighbors are thus still to be determined for each tetrahedron t_i, which means practically that an adjacency index has to be attributed to $4t_i + 1$, $4t_i + 2$ and $4t_i + 3$. The internal facets across which these adjacencies have to be identified are made of the common vertex p_{k+1} and one oriented edge of the cavity boundary.

Using an elaborated hash table with complex collision handling would be overkill. We prefer to use a double entry lookup table of dimension $n \times n$, whose rows and columns are associated with the n vertices $\{p_j\}$ of the cavity boundary, to which auxiliary indices $0 \leq i_j < n$ are affected for convenience and stored in the padding variable of the vertex, With this, the unique index of an oriented edge p_1p_2 is set to be $n \times i_1 + i_2$, which corresponds to one position in the $n \times n$ lookup table. So, for each tetrahedron t_i in the Delaunay ball with vertices $\{p_{k+1}, p_1, p_2, p_3\}$, the adjacency index $4t_i + 1$ is stored at position $n \times i_2 + i_3$ in the lookup table, and similarly $4t_i + 2$ at position $n \times i_3 + i_1$ and $4t_i + 3$ at position $n \times i_1 + i_2$. A square array with a zero diagonal is built proceeding this way, in which the sought adjacencies are the pairs of symmetric components.

Theoretically, there is no maximal value for the number n of vertices in the cavity but, in practice, we can take advantage of the fact that it remains relatively small. Indeed, the cavity boundary is the triangulation of a topological sphere, i.e. a planar graph in which the number of edges is $E = 3F/2$, where F is the number of faces, and whose Euler characteristic is $n - E + F = 2$.

```
typedef struct {
    uint32_t new_tetrahedron_vertices[4];  //  facet vertices + vertex to insert
    uint64_t adjacent_tetrahedron_ID;
} cavityBoundaryFacet_t

typedef struct{
    uint64_t adjacency_map[1024];  // optimization purposes, see Section 2.5

    struct {
        cavityBoundaryFacet_t* boundary_facets;
        uint64_t num;            // number of boundary facets
        uint64_t allocated_num; // capacity [in cavityBoundaryFacet_t]
    } to_create;

    struct {
        uint64_t* tetrahedra_ID;
        uint64_t num;            // number of deleted tetrahedra
        uint64_t allocated_num; // capacity
    } deleted;
} cavity_t;
```

Listing 2: Cavity specific data structure.

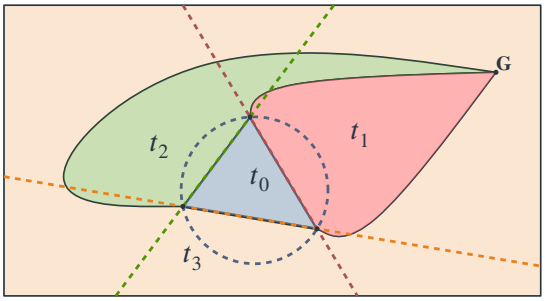


FIGURE 4 Triangle t_0 surrounded by three ghost triangles t_1, t_2 and t_3. Circumcircles are shown in dash lines of the respective color.

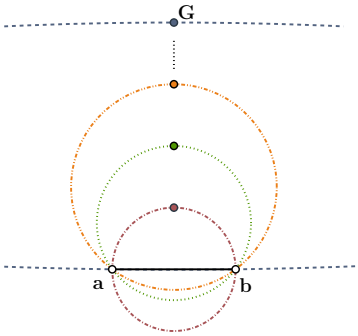


FIGURE 5 Circumcircles of an edge $\overline{ab}$ and increasingly distant vertices. For a vertex G infinitely far away, the degenerate circle approaches a line.

Hence, the number of vertices is linked to the number of triangular faces by $n = F/2 + 2$. As we have shown earlier that F is about 26, we deduce that there are about $n = 15$ vertices on a cavity boundary. Hence, a $n_{max} \times n_{max}$ lookup table, with maximum size $n_{max} = 32$ is sufficient provided there are at most $F_{max} = 2(n_{max} - 2) = 60$ tetrahedra created in the cavity. If the cavity exceptionally contains more elements, the algorithm smoothly switches to a simple linear search.

Once all adjacencies of the new tetrahedra have been properly identified, the DELAUNAYBALL function is then in charge of updating the mesh data structure. This ends the insertion of point p_{k+1}. The space freed by deleted tetrahedra is reused, if needed additional tetrahedra are added. Note that almost all steps of adjacencies recovery are vectorizable.

2.6 | About a ghost

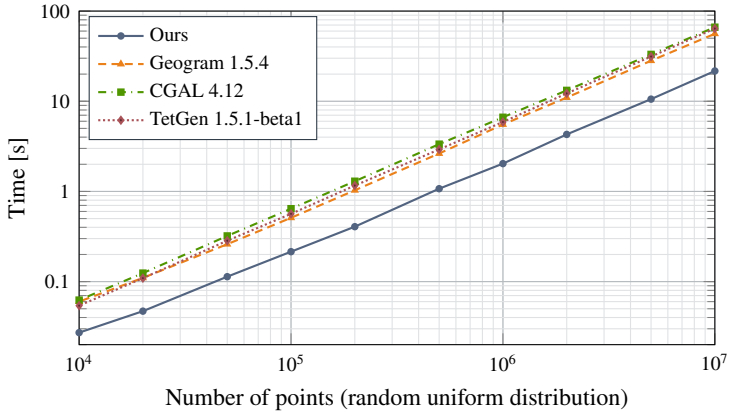
The Bowyer–Watson algorithm for Delaunay triangulation assumes that all newly inserted vertices are inside an element of the triangulation at the previous step (Algorithm 1). To insert a point p_{k+1} outside the current support of the triangulation, one possible strategy is to enclose the input vertices in a sufficiently large bounding box, and to remove the tetrahedra lying outside the convex hull of the point set at the end of the algorithm. A more efficient strategy, adopted in TetGen [15], CGAL [16], Geogram [17], is to work with the so-called ghost tetrahedra connecting the exterior faces of the triangulation with a virtual ghost vertex G. Using this elegant concept, vertices can be inserted outside the current triangulation support with almost no algorithmic change.

The ghost vertex G is the vertex "at infinity" shared by all ghost tetrahedra (Figure 4). The ghost tetrahedra cover the whole space outside the regular triangulation. Like regular tetrahedra, ghost tetrahedra are stored in the mesh data structure, and

are deleted whenever a vertex is inserted inside their circumsphere. The accurate definition of the circumsphere of a ghost tetrahedron, in particular with respect to the requirements of the Delaunay condition, is however a more delicate question.

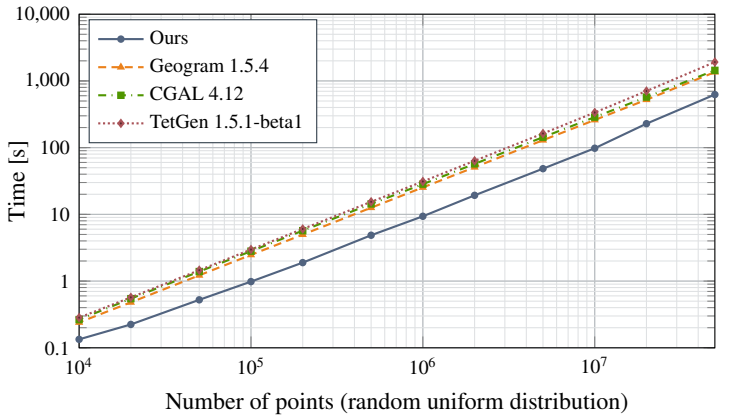
From a geometrical point of view, as the ghost vertex G moves away towards infinity from an exterior facet abc of the triangulation, the circumscribed sphere of tetrahedron $abcG$ tends to the half space on the positive side of the plane determined by the points abc, i.e., the set of points x such that `orient3d`(a, b, c, d) is positive. The question is whether this inequality should be strict or not, and the robust answer is neither one nor the other. This is illustrated in 2D in Figure 5. The circumcircle of a ghost triangle abG actually contains not only the open half-plane strictly above the line $\overleftrightarrow{ab}$, but also the line segment $[ab]$ itself; the other parts of the line $\overleftrightarrow{ab}$ being excluded. In 3D, the circumsphere of a ghost tetrahedron $abcG$ contains the half-space L such that for any point $d \in L$, `orient3d`(a, b, c, d) is strictly positive, plus the disk defined by the circumcircle of the triangle abc. If the point d is in the plane abc, i.e. `orient3d`(a, b, c, d)`=0`, we thus additionally test if d is in the circumsphere of the adjacent regular tetrahedron sharing the triangle abc. This composite definition of the circumscribed circle makes this approach robust with minimal changes: only the `inSphere` predicate is modified in the implementation.

Because the 3×3 determinant of `orient3d` is used instead of the 4×4 determinant of `inSphere` for ghost tetrahedra, the approach with a ghost vertex is faster than other strategies[23]. Note that the WALK cannot evaluate `orient3d` on the faces of ghost tetrahedra that connect edges of the convex hull to the ghost vertex. If the starting tetrahedron is a ghost tetrahedron, the WALK directly steps into the non-ghost adjacent tetrahedron. Then, if it steps in an other ghost tetrahedron $abcG$ while walking toward p_{k+1}, we have `orient3d`$(a, b, c, p_{k+1}) > 0$ and the ghost tetrahedron is inside of the cavity. The walk hence stops there, and the algorithm proceeds as usual.



# vertices	10^4	10^5	10^6	10^7
Ours	0.027	0.21	2.03	21.66
Geogram	0.060	0.51	5.53	56.02
CGAL	0.062	0.64	6.65	66.24
TetGen	0.054	0.56	5.89	63.99

(a) Intel® Core™ i7-6700HQ CPU, maximum core frequency of 3.5Ghz.



# vertices	10^4	10^5	10^6	10^7
Ours	0.134	0.98	9.36	97.97
Geogram	0.240	2.47	25.34	259.74
CGAL	0.265	2.81	28.36	286.54
TetGen	0.283	2.97	31.13	336.21

(b) Intel® Xeon Phi™ 7210 CPU, maximum core frequency of 1.5Ghz.

FIGURE 6 Performances of our sequential Delaunay triangulation implementation (Algorithm 1) on a laptop **(a)** and on a slow CPU having AVX-512 vectorized instructions **(b)**. Timings are in seconds and exclude the initial spatial sort.

2.7 | Serial implementation performances

Our reference implementation of the serial Delaunay triangulation algorithm has about one thousand lines (without Shewchuk's geometric predicates[39]). It is open-source and available in Gmsh (www.gmsh.info). Overall, it is almost three time faster than other sequential implementations. Figure 6a and 6b show the performance gap between our implementation and concurrent software on a laptop with a maximum core frequency of 3.5Ghz, and on a many-core computer with a maximum core frequency of 1.5Ghz and wider SIMD instructions. Table 1 indicates that the main difference in speed comes from the more efficient adjacencies computation in the `DelaunayBall` function. Other software use a more general function that can also handle cavities with multiple interior vertices. But this situation does not happen with Delaunay insertion, where there is always one unique point in the cavity, p_{k+1}. Since our `DelaunayBall` function is optimized for Delaunay cavities, it is approximately three time faster, despite the additional computation of sub-determinants. The remaining performance gain is explained by our choice of simple but efficient memory aligned data structure.

3 | PARALLEL DELAUNAY

3.1 | Related work

To overcome memory and time limitations, Delaunay triangulations should be constructed in parallel making the most of both distributed and shared memory architectures. A triangulation can be subdivided into multiple parts, each constructed and stored on a node of a distributed memory cluster. Multiple methods have been proposed to merge independently generated Delaunay triangulations[41–44]. However, those methods feature complicated merge steps, which are often difficult to parallelize. To avoid merge operations, other distributed implementations maintain a single valid Delaunay triangulation and use synchronization between processors whenever a conflict may occur at inter-processor boundaries[45, 46]. In finite-element mesh generation, merging two triangulations can be simpler because triangulations are not required to be fully Delaunay, allowing algorithms to focus primarily on load balancing[47].

On shared memory machines, divide-and-conquer approaches remain efficient, but other approaches have been proposed since communication costs between different threads are not prohibitive to the contrary of distributed memory machines. To insert a point in a Delaunay triangulation, the kernel procedure operates on a cavity that is modified to accommodate the inserted point (Figure 1). Two points can therefore be inserted concurrently in a Delaunay triangulation if their respective cavities do not intersect, $C(DT_k, p_{k1}) \cap C(DT_k, p_{k2}) = \emptyset$, otherwise there is a conflict. In practice, other types of conflicts and data-races should possibly be taken into account depending on the chosen data structures and the insertion point strategy. Conflict management strategies relying heavily on locks [2] lead to relatively good speedups on small numbers of cores[48–51]. Remacle et al. presented an interesting strategy that checks if insertions can be done in parallel by synchronizing threads with barriers[22]. However, synchronization overheads prevent those strategies from scaling to an high number of cores. More promising approaches rely on a partitioning strategy[52, 53]. Contrarily to pure divide-and-conquer strategies for distributed memory machines, partitions can change and move regularly, and they do not need to cover the whole mesh at once.

In this section, we propose to parallelize the Delaunay kernel using partitions based on a space-filling curve, similarly to Loseille et al.[53]. The main difference is that we significantly modify the partitions at each iteration level. Our code is designed for shared memory architecture only, we leave its integration into a distributed implementation for future work.

3.2 | A parallel strategy based on partitions

There are multiple conditions a program should ensure to avoid data-races and conflicts between threads when concurrently inserting points in a Delaunay triangulation. Consider thread t_1 is inserting point p_{k1} and thread t_2 is simultaneously inserting point p_{k2}.

1. Thread t_1 cannot access information about any tetrahedron in $C(\mathrm{DT}, p_{k2})$ and inversely. Hence:
   (a) $C(\mathrm{DT}, p_{k1}) \cap C(\mathrm{DT}, p_{k2}) = \emptyset$
   (b) $\mathcal{A}(\mathrm{DT}, p_{k1}) \cap C(\mathrm{DT}, p_{k2}) = \emptyset$ and $\mathcal{A}(\mathrm{DT}, p_{k2}) \cap C(\mathrm{DT}, p_{k1}) = \emptyset$

[2] A lock is a synchronization mechanism enforcing that multiple threads do not access a resource at the same time. When a thread cannot acquire a lock, it usually waits.

(c) Thread t_1 cannot walk into $\mathcal{C}(\mathrm{DT}, p_{k2})$ and reciprocally, t_2 cannot walk into $\mathcal{C}(\mathrm{DT}, p_{k1})$

2. A tetrahedron in $\mathcal{B}(\mathrm{DT}, p_{k1})$ and a tetrahedron in $\mathcal{B}(\mathrm{DT}, p_{k2})$ cannot be created at the same memory index.

To ensure rule (1) it is sufficient to restrain each thread to work on an independent partition of the mesh (Figure 7). This lock-free strategy minimizes synchronization between threads and is very efficient. Each point of the Delaunay triangulation is assigned a partition corresponding to a unique thread. A tetrahedron belongs to a thread if at least three of its vertices are in that thread's partition.

To ensure (1a) and (1b), the insertion of a point belonging to thread is aborted if the thread accessed a tetrahedron that belongs to another thread or that is in the buffer zone. To ensure (1c), we forbid threads to walk in tetrahedra belonging to another thread. Consequently, a thread aborts the insertion of a vertex when (i) the WALK reaches another partition, or when (ii) the CAVITY function reaches a tetrahedron in the buffer zone. To insert these points, the vertices are re-partitioned differently (Section 3.3), a procedure repeated until there is no more vertices to insert or until the rate of successful insertions has become too low. In that case, the number of threads is decreased (Section 3.4). When the number of points to insert has become small enough, the insertion runs in sequential mode to insert all remaining vertices. The first BRIO round is also inserted sequentially to generate a first base mesh. Nevertheless, the vast majority of points are inserted in parallel.

Rule (2) is obvious from a parallel memory management point of view. It is however difficult to ensure it without requiring an unreasonable amount of memory. As explained in Section 3.6, synchronization between threads is required punctually.

3.3 | Partitioning and re-partitioning with Moore curve

We subdivide the n_v points to insert such that each thread inserts the same number of points. Our partitioning method is based on the Moore curve, i.e. on the point insertion order implemented for the sequential Delaunay triangulation (Section 2.3). Space-filling curves are relatively fast to construct and have already been used successfully for partitioning meshes [53–56]. Each partition of the n_{thread} partitions is a set of grid cells that are consecutive along the Moore curve. To compute the partitions, i.e. its starting and ending Moore indices, we sort the points to insert according to their Moore indices (see Section 2.3). Then, we assign the first $n_v/n_{threads}$ points to the first partition, the next $n_v/n_{threads}$ to the second, etc (Figure 8c). The second step is to partition the current Delaunay triangulation in which the points will be inserted. We use once more the Moore indices to assign the mesh vertices to the different partitions. The ghost vertex is assigned a random index. The partition owning a tetrahedron is determined from the partitions of its vertices, if a tetrahedron has at least three vertices in a partition it belong to this partition, otherwise the tetrahedron is in the buffer zone. Each thread then owns a subset of the points to insert and a subset of the current triangulation.

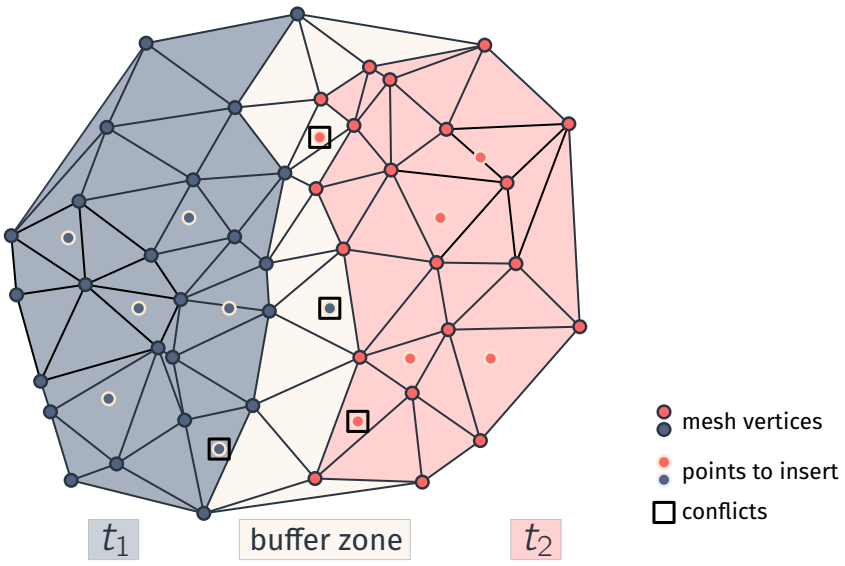
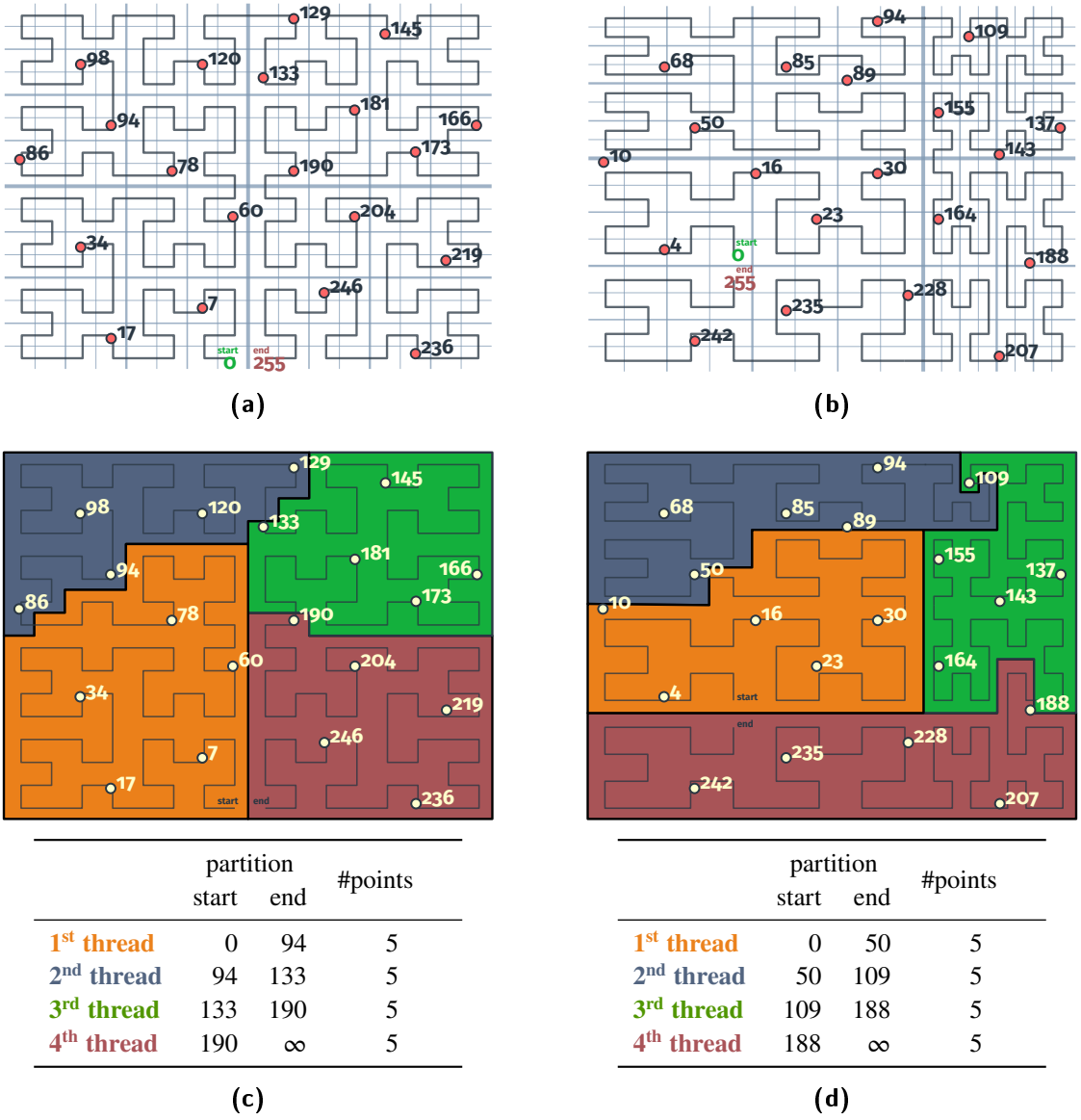


FIGURE 7 Vertices are partitioned such that each vertex belongs to a single thread. A triangle can only be modified by a thread that owns all of its three vertices. Triangles that cannot be modified by any thread form a buffer zone.

Once all threads attempted to insert all their points, a large majority of vertices is generally inserted. To insert the vertices for which insertion failed because the point cavity spans multiple partitions (Figure 7), we modify significantly the partitions by modifying the Moore indices computation. We apply a coordinate transformation and a circular shift to move the zero index around the looping curve (Figure 8). Coordinates below a random threshold are linearly compressed, while coordinates above the threshold are linearly expanded.



	partition start	partition end	#points
1st thread	0	94	5
2nd thread	94	133	5
3rd thread	133	190	5
4th thread	190	∞	5

	partition start	partition end	#points
1st thread	0	50	5
2nd thread	50	109	5
3rd thread	109	188	5
4th thread	188	∞	5

FIGURE 8 Partitioning of 20 points in 2D using the Moore indices, on the right the supporting grid of the Moore curve is transformed and the curve is shifted. In both cases, each partition contains 5 points. Indeed, the starting and ending Moore index of each partition are defined in a way that balances the point insertions between threads.

3.4 | Ensuring termination

When the number of vertices of the current Delaunay triangulation is small, typically at the first steps of the algorithm, the probability that the mesh vertices belong to different partitions is very high and none of the point insertions may succeed. To avoid wasting precious milliseconds, the first BRIO round is always inserted sequentially.

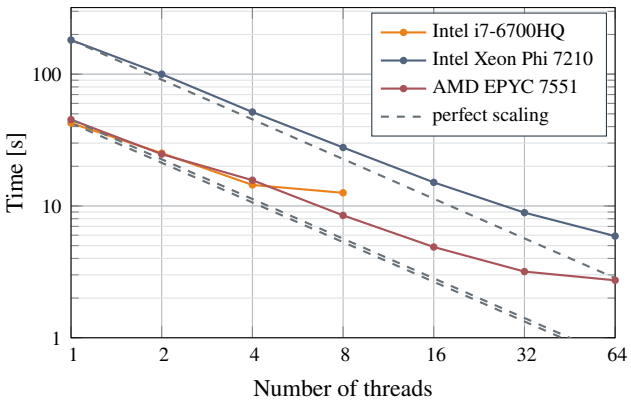


FIGURE 9 Strong scaling of our parallel Delaunay for a random uniform distribution of 15 million points, resulting in over 100 million tetrahedra on 3 machines: a quad-core laptop, an Intel Xeon Phi with 64 cores and a dual-socket AMD EPYC 2×32 cores.

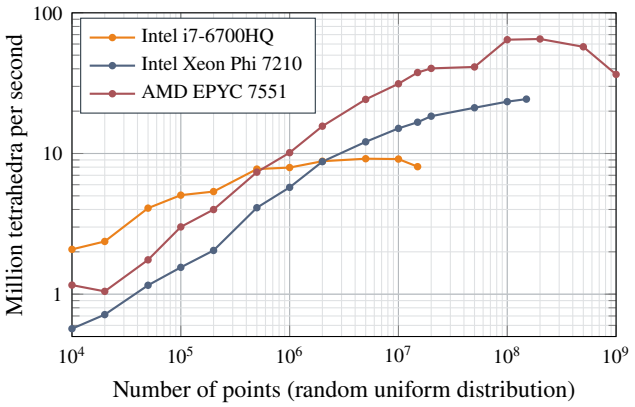


FIGURE 10 Number of tetrahedra created per second by our parallel implementation for different number of points. Tetrahedra are created more quickly when there is a lot of points because the proportion of conflicts is lower. An average rate of 65 million tetrahedra created per second is obtained on the EPYC.

Moreover, there is no guarantee that re-partitioning will be sufficient to insert all the points. And even when a large part of the triangulation has already been constructed, parallel insertion may enter an infinite failure loop. After a few rounds, the remaining points to insert are typically located in restricted volumes (intersection of previous buffer zones). Because re-partitioning is done on the not-yet-inserted points, the resulting partitions are also small and thin. This leads to inevitable conflicts as partitions get smaller than cavities. In practice, we observed that the ratio ρ of successful insertions decreases for a constant number of threads. If 80 out of 100 vertices are successfully inserted in a mesh ($\rho_k = 0.8$), less that 16 of the 20 remaining vertices will be inserted at the following attempt ($\rho_{k+1} < 0.8$). Note that this effect is more important for small meshes, because the bigger the mesh, the relatively smaller the buffer zone and the higher the insertion success rate. This difference explains the growth of the number of tetrahedra created per second with the number of points in the triangulation (Figure 10).

To avoid losing a significant amount of time in re-partitioning and trying to insert the same vertex multiple times we gradually decrease the number of threads. Choosing the adequate number of threads is a question of balance between the potential parallelization gain and the partitioning cost that comes with each insertion attempt. When decreasing the number of threads, we decrease the number of attempts needed. When the ratio of successful insertions is too low, $\rho < 1/5$, or when the number of points to insert per thread is under 3000, we divide the number of threads by two. Furthermore, if $\rho_k \leq 1/n_{threads}$, the next insertion attempt will not benefit from multi-threading and we insert the points sequentially.

In Table 2 are given the number of threads used at each step of the Delaunay triangulation of a million vertices depending on the number of points to insert, the size of the current mesh, and the success insertion ratio ρ. Note that the computation of Moore indices and the sorting of vertices to insert are always computed on the maximal number of threads available (8 threads in this case) even when the insertion runs sequentially.

3.5 | Data structures

The data structure for our parallel Delaunay triangulation algorithm is similar to the one used by our sequential implementation (Section 2.2). There are two small differences. First, the parallel implementation does not compute sub-determinants for each tetrahedron. Actually, the bandwidth usage with the parallel insertions is already near its maximum. Loading and storing sub-determinant becomes detrimental to the overall performance. Instead we store a 16-bit[3] color flag to mark deleted tetrahedra. Second, each thread has its own `Cavity_t` structure (Listing 2) to which are added two integers identifying the starting and ending Moore indices of the thread's partition.

[3] An 8-bit char would also work (not less or it would create data races) but the color flag is also used to distinguish volumes in our mesh generator described in §4

	#points			#threads	#mesh
	to insert	inserted	ρ		vertices
Initial mesh					4
BRIO Round 1	2044	2044	100%	1	2048
BRIO Round 2	12 288	6988	57%	4	9036
	5300	3544	67%	2	12 580
	1756	1756	100%	1	14 336
BRIO Round 3	86 016	59 907	70%	8	74 243
	26 109	11 738	45%	8	85 981
	14 371	7092	49%	4	93 073
	7279	5332	73%	2	98 405
	1947	1947	100%	1	100 352
BRIO Round 4	602 112	503 730	84%	8	604 082
	98 382	44 959	46%	8	649 041
	53 423	31 702	59%	8	680 743
	21 721	7903	36%	8	688 646
	13 818	9400	68%	4	698 046
	4418	3641	82%	2	701 687
	777	777	100%	1	702 464
BRIO Round 5	297 536	271 511	91%	8	973 975
	26 025	16 426	63%	8	990 401
	9599	8092	84%	4	998 493
	1507	1507	100%	1	1 000 000

TABLE 2 Numbers of threads used to insert points in our parallel Delaunay triangulation implementation according to the number of points to insert, the mesh size and the insertion success at the previous step. 94.5% of points are inserted using 8 threads and 5% using 4 threads

Memory footprint Because we do not store four sub-determinants per tetrahedra anymore but only a 2-bytes color, our mesh data structure is lighter. Still assuming that there is approximately $6n$ tetrahedra for n vertices, it requires a little more than $6 \times 50 + 32 = 332$ bytes per vertex. Thanks to this low memory footprint, we were able to compute the tetrahedralization of $N = 10^9$ vertices (about 6 billion of tetrahedra) on an AMD EPYC machine that has 512 GB of RAM. The experimental memory usage shown in Table 3 differ slightly from the theoretical formula because (i) more memory is allocated than what is used (ii) measurements represent the maximum memory used by the whole program, including the stack, the text and data segment etc.

# vertices	10^4	10^5	10^6	10^7
Ours	6.9 MB	43.8 MB	404.8 MB	3.8 GB
Geogram	6.7 MB	30.5 MB	268.6 MB	2.7 GB
CGAL	14.1 MB	66.7 MB	578.8 MB	5.7 GB

TABLE 3 Comparison of the maximum memory usage of our parallel implementation, CGAL[57] and Geogram[17] when using 8 threads.

```
if(cavity->to_create.num > cavity->deleted.num)
{
    uint64_t nTetNeeded = MAX(8192, cavity->to_create.num) - cavity->deleted.num;

    uint64_t nTet;
    #pragma omp atomic capture
    {
        nTet = mesh->tetrahedra.num;
        mesh->tetrahedra.num+=nTetNeeded;
    }

    reallocTetrahedraIfNeeded(mesh);
    reallocDeletedIfNeeded(state, cavity->deleted.num + nTetNeeded);

    for (uint64_t i=0; i<nTetNeeded; i++){
        cavity->deleted.tetrahedra_ID[cavity->deleted.num+i] = 4*(nTet+i);
        mesh->tetrahedra.color[nTet+i] = DELETED_COLOR;
    }

    cavity->deleted.num += nTetNeeded;
}
```

Listing 3: When there are less deleted tetrahedra than there are tetrahedra in the Delaunay ball, 8192 new "deleted tetrahedra" indices are reserved by the thread. As the `mesh` data structure is shared by all threads, `mesh->tetrahedra.num` must be increased in one single atomic operation.

3.6 | Critical operations

When creating new tetrahedra in the Delaunay ball of a point, a thread first recycles unused memory space by replacing deleted tetrahedra. The indices of deleted tetrahedra are stored in the `cavity->deleted.tetrahedra_ID1` array of each thread. When the `cavity->deleted.tetrahedra_ID` array of a thread is empty, additional memory should be reserved by this thread to create new tetrahedra.

This operation is a critical part of the program requiring synchronization between threads to respect the rule (2). We need to capture the current number of tetrahedra and increment it by the requested number of new tetrahedra in one single atomic operation. *OpenMP* provides the adequate mechanism, see Listing 3.

To reduce the number of time this operation is done, the number of tetrahedra is incremented atomically by at least 8192, and the `cavity->deleted.tetrahedra_ID` is filled with the new indices of tetrahedra. Those tetrahedra are conceptually deleted although they have never been in the mesh. The number 8192 was found experimentally among multiples of 512[4]. Increasing it would reduce the number of time reservation of new tetrahedra is performed but it is not necessary. Indeed, since the critical operation occurs then in average every 1000^+ insertions, the time wasted is very low. Therefore, increasing the default number of deleted tetrahedra would only wastes memory space for little to no gain.

When the space used by tetrahedra exceeds the initially allocated capacity, reallocation is implemented that doubles the capacity of the arrays of mesh tetrahedra. During that operation, memory is potentially moved at another location. No other operation can be performed at that time on the mesh. Therefore, the reallocation code is placed in between two OpenMP barriers (Listing 4). This synchronization event is very rare and does not impact performances. In general, a good estimation of the needed memory needed to reserve is possible and that critical section is never reached.

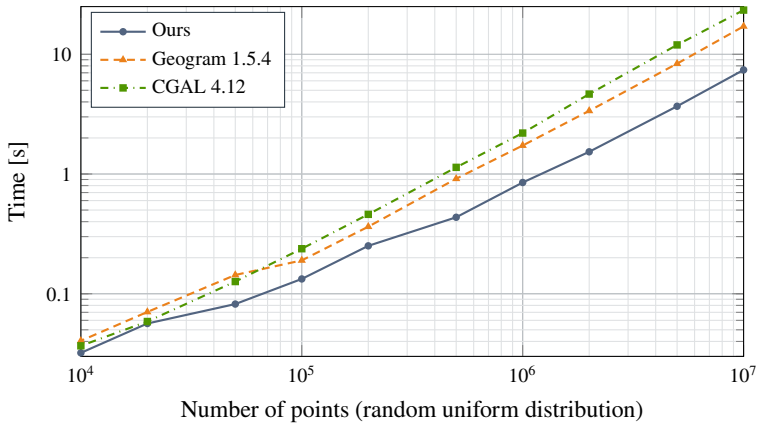
3.7 | Parallel implementation performances

We are able to compute the Delaunay triangulation of over one billion tetrahedra in record-breaking time: 41.6 seconds on the Intel Xeon Phi and 17.8 seconds on the AMD EPYC. These timings do not include I/Os. As for the title of this article, we are able to generate three billion tetrahedra in 53 seconds on the EPYC. The scaling of our implementation regarding the number of threads is detailed in Figure 9. We obtain a good scaling until the number of threads reach the number of cores, i.e. 4 cores for the Intel i7-6700HQ, 64 cores for the Intel Xeon Phi 7210 and the AMD EPYC 7551. To our best knowledge, CGAL[57] and Geogram[17] are the two fastest open-source CPU implementations available for 3D Delaunay triangulation. We compare their performances to ours on a laptop (Figure 11a) and on the Intel Xeon Phi (Figure 11b).

[4]It is common to choose a multiple of the page size (usually 4096 bytes) to minimize TLB misses.

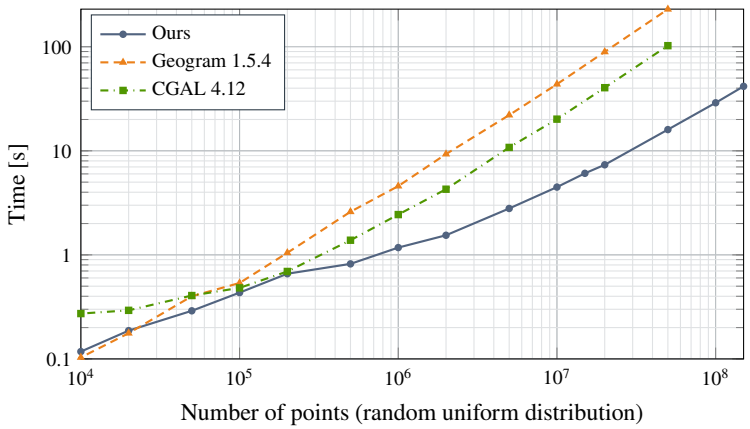
```
void reallocTetrahedraIfNeeded(mesh_t* mesh)
{
    if(mesh->tetrahedra.num > mesh->tetrahedra.allocated_num)
    {
        #pragma omp barrier

        // all threads are blocked except the one doing the reallocation
        #pragma omp single
        {
            uint64_t nTet = mesh->tetrahedra.num;
            alignedRealloc(&mesh->tetrahedra.neighbor_ID, nTet*8*sizeof(uint64_t));
            alignedRealloc(&mesh->tetrahedra.vertex_ID, nTet*8*sizeof(uint32_t));
            alignedRealloc(&mesh->tetrahedra.color, nTet*2*sizeof(uint16_t));
            mesh->tetrahedra.allocated_num = 2*nTet;

        } // implicit OpenMP barrier here
    }
}
```

Listing 4: Memory allocation for new tetrahedra is synchronized with OpenMP barriers.



# vertices	10^4	10^5	10^6	10^7
Ours	0.032	0.13	0.85	7.40
Geogram	0.041	0.19	1.73	17.11
CGAL	0.037	0.24	2.20	23.37

(a) 4-core Intel® Core™ i7-6700HQ CPU.



# vertices	10^4	10^5	10^6	10^7	10^8
Ours	0.11	0.43	1.17	4.48	28.95
Geogram	0.10	0.54	4.58	43.70	/
CGAL	0.27	0.48	2.44	20.15	/

(b) 64-core Intel® Xeon Phi™ 7210 CPU.

FIGURE 11 Comparison of our parallel implementation with the parallel implementation in CGAL[57] and Geogram[17] with on a high-end laptop **(a)** and a many-core computer **(b)**. All timings are in seconds.

4 | TETRAHEDRAL MESH GENERATION

The parallel Delaunay triangulation algorithm that we presented in the previous section is integrated in a Delaunay refinement mesh generator. A tetrahedral mesh generator is a procedure that takes as input the boundary of a domain to mesh, defined by set of triangles t that defines the boundary of a closed volume, and that returns a finite element mesh, i.e. a set of tetrahedra $\mathcal{T}$ of controlled sizes and shapes which boundary is equal to the input triangulation: $\partial\mathcal{T} = t$. From a coarse mesh that is conformal to t, a Delaunay-refinement based mesher inserts progressively vertices until element sizes and shapes follow the prescribed ranges. Generating a mesh is an intrinsically more difficult problem than constructing the Delaunay triangulation of given points because (1) the points are not given, (2) the boundary triangles must be facets of the generated tetrahedra, and (3) the tetrahedra shape and size should be optimized to maximize the mesh quality. Our objective in this section is to demonstrate that our parallel Delaunay point insertion may be integrated in a mesh generator. The interested reader is referred to the book by Frey and George[58] for a complete review of finite-element mesh generation.

The mesh generation algorithm 2 proposed in this section follows the approach implemented for example in Tetgen[15]. First, all the vertices of the boundary triangulation t are tetrahedralized to form the initial "empty mesh" $\mathcal{T}_0$. Then, $\mathcal{T}_0$ is transformed into a conformal mesh $\mathcal{T}$ which boundary $\partial\mathcal{T}$ is equal to $t : \partial\mathcal{T} = t$. The triangulation $\mathcal{T}$ is then progressively refined by (i) creating vertices S at the circumcenters of tetrahedra for which the circumsphere radius, r_τ, is significantly larger than the desired mesh size, h, i.e. $r_\tau/h > 1.4$[23]. (ii) inserting the vertices in the mesh using our parallel Delaunay algorithm. The sets of points S to insert are filtered *a priori* in order not to generate short edges, i.e. edges of size smaller than $0.7h$. We first use Hilbert coordinates to discard very close points on the curve, and implemented a slightly modified cavity algorithm that aborts if a point of the cavity is too close from the one to insert (Section 2.4). Points are thus discarded both in the filtering process and in the insertion process. Note that contrary to existing implementations, we insert as large as possible point sets S during the refinement step to take advantage of the efficiency of our parallel point insertion algorithm (Section 3).

We parallelized all steps of the mesh generator (Algorithm 2) except the boundary recovery procedure that is the one of Gmsh[59] which is essentially based on Tetgen[15].

The cavity algorithm has also been modified to accommodate possible constraints on the faces and edges of tetrahedra. With this modification, mesh refinement never breaks the surface mesh and constrained edges and faces can be included in the mesh interior. As a result, the mesh may not satisfy the Delaunay property anymore. Therefore, we must always ensure that cavities are star-shaped. This is achieved by a simple procedure that checks if boundary facets of the cavity are oriented such that they form a positive volume with the new point. If a boundary facet is not oriented correctly, indicating that the cavity is not star-shaped, the tetrahedron containing it is removed from the cavity and the procedure is repeated. In practice, these modifications do not affect the speed of point insertions significantly.

To generate high-quality meshes, an optimization step should be added to the algorithm to improve the general quality of the mesh elements and remove sliver tetrahedra. Mesh improvements are out of the scope of this paper. However, first experiments have shown that optimization step should slow down the mesh generation by a factor two. For example, mesh refinement and improvement take approximately the same time in Gmsh[59].

Algorithm 2 Mesh generation algorithm

Input: A set of triangles t
Output: A tetrahedral mesh $\mathcal{T}$

```
1:  function ParallelMesher(t)
2:      T_0 ← EmptyMesh(t)
3:      T ← RecoverBoundary(T_0)
4:      while T contains large tetrahedra do
5:          S ← SamplePoints(T)
6:          S ← FilterPoints(S)
7:          T ← InsertPoints(T, S)
8:      end while
9:      return T
10: end function
```

4.1 | Small and medium size test cases on standard laptops

In order to verify the scalability of the whole meshing process, meshes of up to one hundred million tetrahedra were computed on a 4 core 3.5 GHz Intel Core i7-6700HQ with 1, 2, 4 and 8 threads. Those meshes easily fit within the 8Gb of RAM of this modern laptop.

Three benchmarks are considered in this section: (i) a cube filled with cylindrical fibers of random radii and lengths that are randomly oriented, (ii) a mechanical part and (iii) a truck tire. Surface meshes are computed with Gmsh[59]. Mesh size on the surfaces is controlled by surface curvatures and mesh size inside the domain is simply interpolated from the surface mesh.

Illustrations of the meshes, as well as timings statistics are presented in Figure 12. Our mesher is able to generate between 40 and 100 million tetrahedra per minute. Using multiple threads allows some speedup, the mesh refinement process is accelerated by a factor ranging between 2 and 3 on this 4 core machine.

The last test case (truck tire) is defined by more than 7000 CAD surfaces. Recovering the 27 892 triangular facets missing from $\mathcal{T}_0$ takes more than a third of the total meshing time with the maximal number of threads. Parallelizing the boundary recovery process is clearly a priority of our future developments. On this same example, the surface mesh was done with Gmsh using four threads. The surface mesher of Gmsh is not very fast and it took about the same time to generate the surface mesh of 6 881 921 triangles as to generate the volume mesh that contains over one hundred million tetrahedra using the same number of threads. The overall meshing time for the truck tire test case is thus about 6 minutes.

4.2 | Large mesh generation on many core machine

We further generated meshes containing over 300,000,000 elements on a AMD® EPYC 64 core machine. Three benchmarks are considered: (i) two cubes filled with many randomly oriented cylindrical fibers of random radii and lengths, and (ii) the exterior of an aircraft. Surface meshes were also generated by Gmsh.

Our strategy reaches its maximum efficiency for large meshes. In the 500 thin fibers test case, over 700,000,000 tetrahedra were generated in 135 seconds. This represents a rate of 5.2 million tetrahedra per second. In the 500 thin fibers test case, boundary recovery cost was lower and a rate of 6.2 million tetrahedra per second was reached.

5 | CONCLUSION

This paper introduces a scalable Delaunay triangulation algorithm and demonstrates that inserting points concurrently can be performed effectively on shared memory architectures without the need of heavy synchronization mechanisms. Our reference implementation is open-source and available in Gmsh 4 (www.gmsh.info). We optimized both the sequential and the parallel algorithm by specifying them exclusively for Delaunay triangulation purposes. Our parallel implementation of 3D Delaunay triangulation construction has one important drawback: partitioning is more costly for non-uniform point sets because Moore indices are not adaptive. Adaptive Hilbert or Moore curve computation is relatively simple[27, 60] but may not be faster than our refinement and sorting in one batch approach. Another drawback of our implementation is that it might not be well suited for distributed memory architectures. However, nothing prevents our strategy from being included in any larger divide-and-conquer approach.

We additionally presented a lightweight mesh generator based on Delaunay refinement. This mesh generator is almost entirely parallel and is able to produce computational meshes with a rate above 5 million tetrahedra per second. One issue when parallelizing mesh generation is to ensure reproducibility. For mesh generation, it is usually admitted that mesh generation should be reproducible for a given number of threads. Our mesh generator has been made reproducible in that sense, with a loss of 20% in overall performance. This feature has been implemented as an option. On the other hand, Figures 12 and 13 show that, starting from the same input, the number of elements varies. Making the code reproducible independently of the number of threads would dramatically harm its scalability.

Note that our mesh generator does not include the final optimization step of the mesh, *mesh improvement*, to obtain a mesh adequate for finite-element simulations. This crucial step of any industrial-grade meshing procedure is one major limitation of our mesh generator. The basic operations to perform: edge swap, edge collapse, edge split, vertex relocation and face swap operations will be done using the same parallel framework than the one we described to build the Delaunay triangulation. The final challenge is the parallelization of the boundary recovery procedure.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement ERC-2015-AdG-694020).

References

1. Aurenhammer F. Voronoi diagrams—a survey of a fundamental geometric data structure. ACM Computing Surveys. 1991 sep;23(3):345–405. Available from: http://portal.acm.org/citation.cfm?doid=116873.116880.

2. Cheng SW. Delaunay mesh generation. Chapman & Hall/CRC computer and information science series. Boca Raton: CRC Press; 2013.

3. Berger M, Tagliasacchi A, Seversky LM, Alliez P, Guennebaud G, Levine JA, et al. A Survey of Surface Reconstruction from Point Clouds: A Survey of Surface Reconstruction from Point Clouds. Computer Graphics Forum. 2017 jan;36(1):301–329. Available from: http://doi.wiley.com/10.1111/cgf.12802.

4. Cautun MC, van de Weygaert R. The DTFE public software - The Delaunay Tessellation Field Estimator code. arXiv:11050370 [astro-ph]. 2011 may;ArXiv: 1105.0370. Available from: http://arxiv.org/abs/1105.0370.

5. Ramella M, Boschin W, Fadda D, Nonino M. Finding galaxy clusters using Voronoi tessellations. Astronomy & Astrophysics. 2001 mar;368(3):776–786. Available from: http://www.aanda.org/10.1051/0004-6361:20010071.

6. Garland M, Heckbert PS. Fast polygonal approximation of terrains and height fields. Report CMU-CS-95-181, Carnegie Mellon University; 1995.

7. Martinez-Rubi O, Verhoeven S, van Meersbergen M, van Oosterom P. Taming the beast: Free and open-source massive point cloud web visualization; 2016. p. 12.

8. Rasquin M, Smith C, Chitale K, Seol S, Matthews BA, Martin JL, et al. Scalable fully implicit finite element flow solver with application to high-fidelity flow control simulations on a realistic wing design. Computing in Science and Engineering. 2014;16:7.

9. Ibanez DA, Seol ES, Smith CW, Shephard MS. PUMI: Parallel Unstructured Mesh Infrastructure. ACM Transactions on Mathematical Software. 2016 May;42(3):1–28. Available from: http://dl.acm.org/citation.cfm?doid=2935754.2814935.

10. Vázquez M, Houzeaux G, Koric S, Artigues A, Aguado-Sierra J, Arís R, et al. Alya: Multiphysics engineering simulation toward exascale. Journal of Computational Science. 2016 May;14:15–27. Available from: http://www.sciencedirect.com/science/article/pii/S1877750315300521.

11. Slotnick J, Khodadoust A, Alonso J, Darmofal D, Gropp W, Lurie E, et al. CFD vision 2030 study: a path to revolutionary computational aerosciences; 2014. NASA.

12. Nystrom NA, Levine MJ, Roskies RZ, Scott JR. Bridges: A Uniquely Flexible HPC Resource for New Communities and Data Analytics. In: Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure. XSEDE '15. New York, NY, USA: ACM; 2015. p. 30:1–30:8. Available from: http://doi.acm.org/10.1145/2792745.2792775.

13. Fu H, Liao J, Yang J, Wang L, Song Z, Huang X, et al. The Sunway TaihuLight supercomputer: system and applications. Science China Information Sciences. 2016;59(7):072001.

14. Marot C, Pellerin J, Lambrechts J, Remacle JF. Toward one billion tetrahedra per minute. In: 26th International Meshing Roundtable, Research Notes. Barcelona, Spain; 2017. .

15. Si H. TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator. ACM Trans Math Softw. 2015;41(2):11:1–11:36. Available from: http://doi.acm.org/10.1145/2629697.

16. Boissonnat JD, Devillers O, Teillaud M, Yvinec M. Triangulations in CGAL. In: Proceedings of the sixteenth annual symposium on Computational geometry. ACM; 2000. p. 11–18.

17. Levy B. Geogram; 2015 (accessed August 23, 2018). http://alice.loria.fr/software/geogram/doc/html/index.html. Available from: http://alice.loria.fr/software/geogram/doc/html/index.html.

18. Bowyer A. Computing Dirichlet tessellations*. The Computer Journal. 1981 jan;24(2):162–166. Available from: http://dx.doi.org/10.1093/comjnl/24.2.162.

19. Watson DF. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. The Computer Journal. 1981 jan;24(2):167–172. Available from: https://academic.oup.com/comjnl/article/24/2/167/338200.

20. Amenta N, Choi S, Rote G. Incremental constructions con BRIO. In: Fortune S, editor. Proceedings of the 19th ACM Symposium on Computational Geometry, San Diego, CA, USA, June 8-10, 2003. ACM; 2003. p. 211–219. Available from: http://doi.acm.org/10.1145/777792.777824.

21. De Floriani L, Falcidieno B, Nagy G, Pienovi C. On sorting triangles in a delaunay tessellation. Algorithmica. 1991 Jun;6(1):522–532. Available from: https://doi.org/10.1007/BF01759057.

22. Remacle J. A two-level multithreaded Delaunay kernel. Computer-Aided Design. 2017;85:2–9. Available from: https://doi.org/10.1016/j.cad.2016.07.018.

23. Shewchuk JR. Delaunay refinement mesh generation. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE; 1997.

24. Boissonnat JD, Devillers O, Hornus S. Incremental Construction of the Delaunay Triangulation and the Delaunay Graph in Medium Dimension. In: Proceedings of the Twenty-fifth Annual Symposium on Computational Geometry. SCG '09. New York, NY, USA: ACM; 2009. p. 208–216. Available from: http://doi.acm.org/10.1145/1542362.1542403.

25. Haverkort HJ, van Walderveen F. Locality and bounding-box quality of two-dimensional space-filling curves. Comput Geom. 2010;43(2):131–147. Available from: https://doi.org/10.1016/j.comgeo.2009.06.002.

26. Abel DJ, Mark DM. A Comparative Analysis of some 2-Dimensional Orderings. International Journal of Geographical Information Systems. 1990;4(1):21–31. Available from: https://doi.org/10.1080/02693799008941526.

27. Hamilton CH, Rau-Chaplin A. Compact Hilbert indices: Space-filling curves for domains with unequal side lengths. Information Processing Letters. 2008;105(5):155–163. Available from: https://doi.org/10.1016/j.ipl.2007.08.034.

28. Blelloch GE. Scans as Primitive Parallel Operations. IEEE Trans Computers. 1989;38(11):1526–1538. Available from: https://doi.org/10.1109/12.42122.

29. Zagha M, Blelloch GE. Radix sort for vector multiprocessors. In: Martin JL, editor. Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991. ACM; 1991. p. 712–721. Available from: http://doi.acm.org/10.1145/125826.126164.

30. Sohn A, Kodama Y. Load balanced parallel radix sort. In: Proceedings of the 12th international conference on Supercomputing. ACM; 1998. p. 305–312.

31. Satish N, Harris MJ, Garland M. Designing efficient sorting algorithms for manycore GPUs. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009. IEEE; 2009. p. 1–10. Available from: https://doi.org/10.1109/IPDPS.2009.5161005.

32. Wassenberg J, Sanders P. Engineering a Multi-core Radix Sort. In: Jeannot E, Namyst R, Roman J, editors. Euro-Par 2011 Parallel Processing. Berlin, Heidelberg: Springer Berlin Heidelberg; 2011. p. 160–169.

33. Polychroniou O, Ross KA. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In: Dyreson CE, Li F, Özsu MT, editors. International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014. ACM; 2014. p. 755–766. Available from: http://doi.acm.org/10.1145/2588555.2610522.

34. Satish N, Kim C, Chhugani J, Nguyen AD, Lee VW, Kim D, et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In: Elmagarmid AK, Agrawal D, editors. Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010. ACM; 2010. p. 351–362. Available from: http://doi.acm.org/10.1145/1807167.1807207.

35. Merrill D, Grimshaw A. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. Parallel Processing Letters. 2011;21(02):245–272. Available from: http://www.worldscinet.com/ppl/21/2102/S0129626411000187.html.

36. Bell N, Hoberock J. Thrust: A productivity-oriented library for CUDA. In: GPU computing gems Jade edition. Elsevier; 2011. p. 359–371.

37. Sengupta S, Harris MJ, Zhang Y, Owens JD. Scan primitives for GPU computing. In: Segal M, Aila T, editors. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2007, San Diego, California, USA, August 4-5, 2007. Eurographics Association; 2007. p. 97–106. Available from: https://doi.org/10.2312/EGGH/EGGH07/097-106.

38. Remacle JF, Shephard MS. An algorithm oriented mesh database. International Journal for Numerical Methods in Engineering. 2003;58(2):349–374.

39. Shewchuk JR. Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete & Computational Geometry. 1997;18(3):305–363.

40. Edelsbrunner H, Mücke EP. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. ACM Trans Graph. 1990;9(1):66–104. Available from: http://doi.acm.org/10.1145/77635.77639.

41. Cignoni P, Montani C, Perego R, Scopigno R. Parallel 3D Delaunay Triangulation. Computer Graphics Forum. 1993 aug;12(3):129–142. Available from: http://doi.wiley.com/10.1111/1467-8659.1230129.

42. Chen M. The Merge Phase of Parallel Divide-and-Conquer Scheme for 3D Delaunay Triangulation. In: International Symposium on Parallel and Distributed Processing with Applications; 2010. p. 224–230.

43. Funke D, Sanders P. Parallel d -D Delaunay Triangulations in Shared and Distributed Memory. In: 2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX). Society for Industrial and Applied Mathematics; 2017. p. 207–217. Available from: http://epubs.siam.org/doi/10.1137/1.9781611974768.17.

44. Blelloch GE, Miller GL, Hardwick JC, Talmor D. Design and Implementation of a Practical Parallel Delaunay Algorithm. Algorithmica. 1999 jul;24(3-4):243–269. Available from: http://link.springer.com/10.1007/PL00008262.

45. Okusanya T, Peraire J. 3D PARALLEL UNSTRUCTURED MESH GENERATION. Citeseer; 1996.

46. Chrisochoides N, Nave D. Parallel Delaunay mesh generation kernel. International Journal for Numerical Methods in Engineering. 2003 sep;58(2):161–176. Available from: http://doi.wiley.com/10.1002/nme.765.

47. Lachat C, Dobrzynski C, Pellegrini F. Parallel mesh adaptation using parallel graph partitioning; 2014. p. 13.

48. Kohout J, Kolingerová I, Žára J. Parallel Delaunay triangulation in E^2 and E^3 for computers with shared memory. Parallel Computing. 2005;31(5):491–522. Available from: http://linkinghub.elsevier.com/retrieve/pii/S0167819105000323.

49. Blandford DK, Blelloch GE, Kadow C. Engineering a compact parallel delaunay algorithm in 3D. ACM Press; 2006. p. 292. Available from: http://portal.acm.org/citation.cfm?doid=1137856.1137900.

50. Batista VHF, Millman DL, Pion S, Singler J. Parallel geometric algorithms for multi-core computers. Computational Geometry. 2010 Oct;43(8):663–677. Available from: http://linkinghub.elsevier.com/retrieve/pii/S0925772110000362.

51. Foteinos P, Chrisochoides N. Dynamic Parallel 3D Delaunay Triangulation. In: Quadros WR, editor. Proceedings of the 20th International Meshing Roundtable. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 3–20. Available from: http://link.springer.com/10.1007/978-3-642-24734-7_1.

52. Lo SH. Parallel Delaunay triangulation in three dimensions. Computer Methods in Applied Mechanics and Engineering. 2012 sep;237-240:88–106. Available from: http://linkinghub.elsevier.com/retrieve/pii/S0045782512001570.

53. Loseille A, Alauzet F, Menier V. Unique cavity-based operator and hierarchical domain partitioning for fast parallel generation of anisotropic meshes. Computer-Aided Design. 2017 apr;85:53–67. Available from: https://linkinghub.elsevier.com/retrieve/pii/S0010448516301142.

54. Lieber M, Grützun V, Wolke R, Müller MS, Nagel WE. FD4: A Framework for Highly Scalable Load Balancing and Coupling of Multiphase Models. AIP Conference Proceedings. 2010;1281(1):1639–1642. Available from: https://aip.scitation.org/doi/abs/10.1063/1.3498143.

55. Aluru S, Sevilgen FE. Parallel domain decomposition and load balancing using space-filling curves. In: Proceedings Fourth International Conference on High-Performance Computing; 1997. p. 230–235.

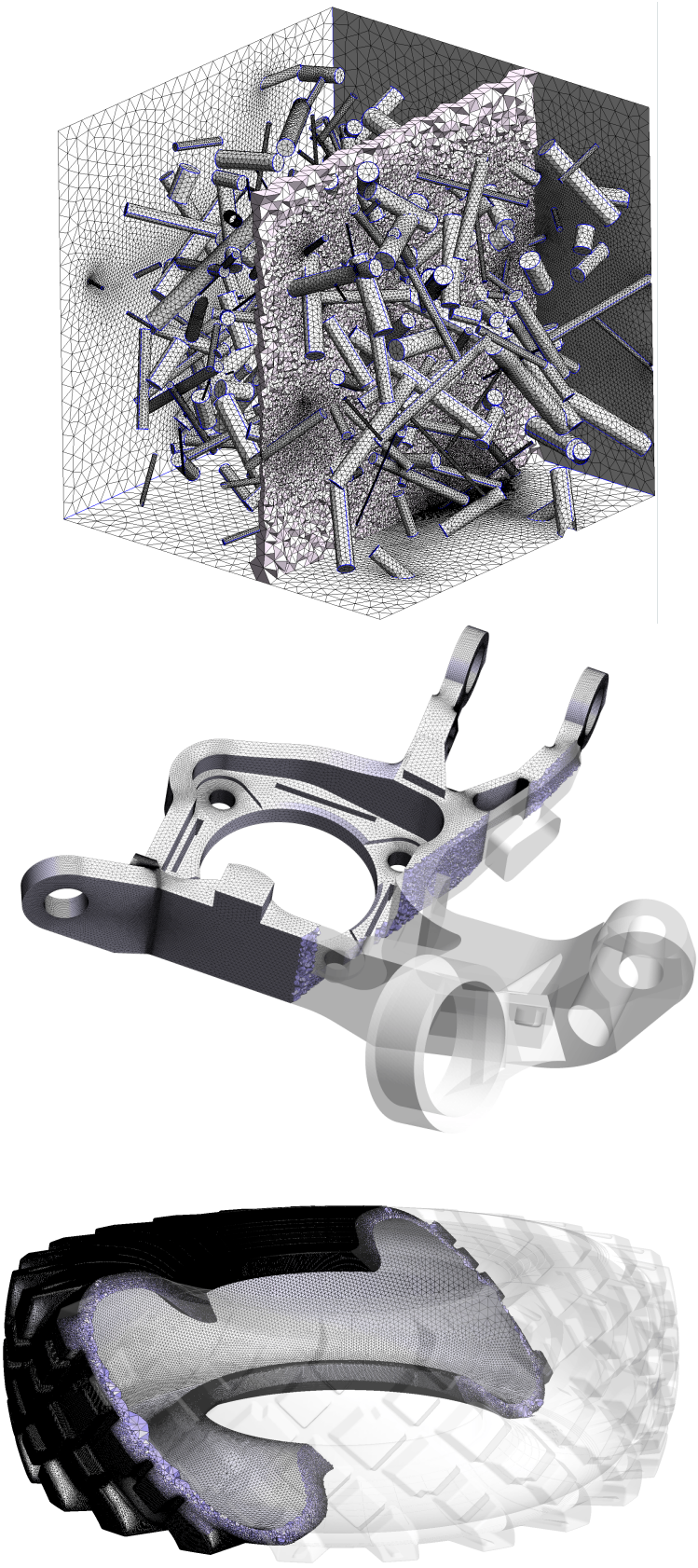
56. Devine KD, Boman EG, Heaphy RT, Hendrickson BA, Teresco JD, Faik J, et al. New challenges in dynamic load balancing. Applied Numerical Mathematics. 2005;52(2-3):133–152.

57. Jamin C, Pion S, Teillaud M. 3D Triangulations. In: CGAL User and Reference Manual. 4.12 ed. CGAL Editorial Board; 2018. Available from: https://doc.cgal.org/4.12/Manual/packages.html#PkgTriangulation3Summary.

58. Frey PJ, George PL. Mesh Generation: Application to Finite Elements. ISTE; 2007.

59. Geuzaine C, Remacle JF. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. International Journal for Numerical Methods in Engineering. 2009 sep;79(11):1309–1331. Available from: http://doi.wiley.com/10.1002/nme.2579.

60. Su T, Wang W, Lv Z, Wu W, Li X. Rapid Delaunay triangulation for randomly distributed point cloud data using adaptive Hilbert curve. Computers & Graphics. 2016 feb;54:65–74. Available from: http://linkinghub.elsevier.com/retrieve/pii/S0097849315001223.



100 fibers

# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	12 608 242	0.74	19.6	20.8
2	12 600 859	0.72	13.6	14.6
4	12 567 576	0.72	8.7	9.8
8	12 586 972	0.71	7.6	8.7

300 fibers

# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	52 796 891	6.03	92.4	101.3
2	52 635 891	5.76	61.2	69.0
4	52 768 565	5.71	39.4	46.8
8	52 672 898	5.67	32.5	39.8

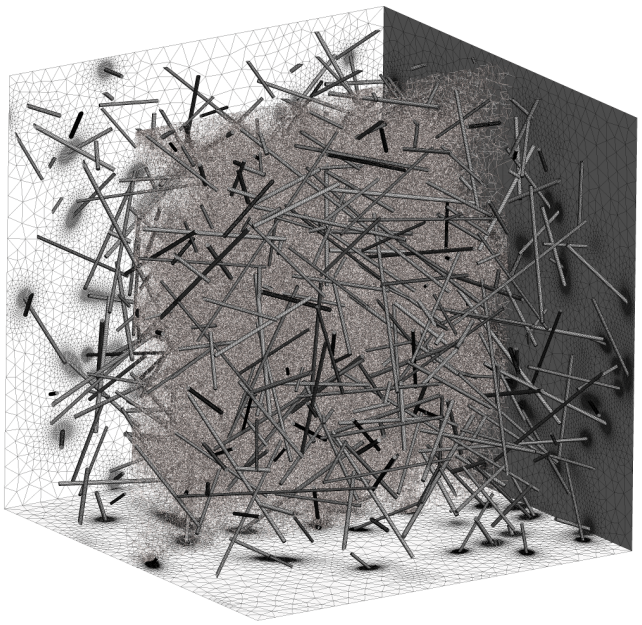
Mechanical part

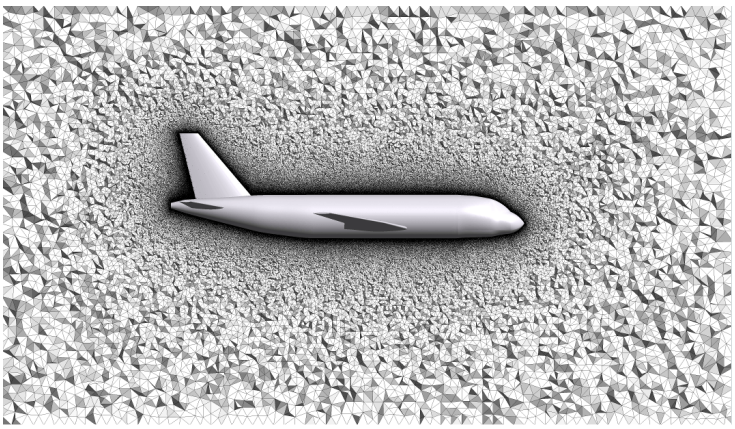
# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	24 275 207	8.6	43.6	56.3
2	24 290 299	8.4	30.4	41.8
4	24 236 112	8.1	24.6	35.3
8	24 230 468	8.1	21.8	32.6

Truck tire

# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	123 640 429	75.9	259.7	364.7
2	123 593 913	74.5	166.8	267.1
4	123 625 696	74.2	107.4	203.6
8	123 452 318	74.2	95.5	190.0

FIGURE 12 Performances of our parallel mesh generator on a Intel® Core™ i7-6700HQ 4-core CPU. Wall clock times are given for the whole meshing process for 1 to 8 threads. They include IOs (sequential), initial mesh generation (parallel), as well as sequential boundary recovery (BR), and parallel Delaunay refinement for which detailed timings are given.





100 thin fibers

# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	325 611 841	3.1	492.1	497.2
2	325 786 170	2.9	329.7	334.3
4	325 691 796	2.8	229.5	233.9
8	325 211 989	2.7	154.6	158.7
16	324 897 471	2.8	96.8	100.9
32	325 221 244	2.7	71.7	75.8
64	324 701 883	2.8	55.8	60.1
127	324 190 447	2.9	47.6	52.0

500 thin fibers

# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	723 208 595	18.9	1205.8	1234.4
2	723 098 577	16.0	780.3	804.8
4	722 664 991	86.6	567.1	659.8
8	722 329 174	15.8	349.1	370.1
16	723 093 143	15.6	216.2	236.5
32	722 013 476	15.6	149.7	169.8
64	721 572 235	15.9	119.7	140.4
127	721 591 846	15.9	114.2	135.2

Aircraft

# threads	# tetrahedra	Timings (s)		
		BR	Refine	Total
1	672 209 630	45.2	1348.5	1418.3
2	671 432 038	42.1	1148.9	1211.5
8	665 826 109	39.6	714.8	774.8
64	664 587 093	38.7	322.3	380.9
127	663 921 974	38.1	255.0	313.3

FIGURE 13 Performances of our parallel mesh generator on a AMD® EPYC 64-core machine. Wall clock times are given for the whole meshing process for 1 to 127 threads. They include IOs (sequential), initial mesh generation (parallel), as well as sequential boundary recovery (BR), and parallel Delaunay refinement for which detailed timings are given.