One machine, one minute, three billion tetrahedra

Authors: Célestin Marot, Jeanne Pellerin, Jean-François Remacle

02/25/2020

Paper Outline

- Focuses on solving the problem of computing Delaunay triangulations.
- First, presents a highly optimized improvement of the state-of-the-art sequential algorithm (Bowyer-Watson) for computing Delaunay triangulations.
- Second, presents a scalable, parallelized version of the algorithm that avoids many synchronization issues and complexity that previous parallel implementations have faced.
- Third, demonstrates how this triangulation algorithm can be plugged into a Delaunay refinement mesh generator and improve its performance in practice.

Context

- Delaunay Triangulation is a fundamental problem in computational geometry, and is used in many domains (3D scanners, astrophysics, terrain modeling, etc.)
- Although it can be generalized mathematically to any number of dimensions, its main practical applicability comes in 3 dimensions
- As data becomes more abundant, it is becoming more important to be able to triangulate efficiently
- Shared-memory, multi-core machines are fairly common now with even hundreds of cores, so it is sufficient to develop for a single shared-memory machine

Delaunay Triangulation

- Given a set P of points, a Delaunay triangulation is one such that no point in P lies within the circumcircle of any triangle in the triangulation.
- This has many nice locality properties, such as maximizing the minimum angles of all interior angles in the triangulation, that make it a problem of interest.
- It can be calculated numerically through the Bowyer-Watson algorithm, an incremental insertion algorithm.

Bowyer-Watson Algorithm

- An incremental insertion algorithm that builds the triangulation point-by-point, stepwise fashion.
- Given the Delaunay Triangulation DT_k of subset $S_k = \{p_1,\ldots,p_k\}$ of S, the algorithm insert point p_{k+1} into the triangulation and creates DT_{k+1}, a valid triangulation of the subset $S_{k+1} = \{p_1,\ldots,p_{k+1}\}$ of S.
- The insertion first identifies the set of tetrahedra in DT_k whose circumspheres contain p_{k+1}. This is the cavity of p_{k+1} in DT_k.
- It then removes this set, as they violate the definition.
- It then adds back a set of tetrahedra that fill the resulting space but incorporate p_{k+1} (basically p_{k+1} connected to every boundary triangle). This is the Delaunay Ball of p_{k+1} in DT_k.

Bowyer-Watson Algorithm

- The insertion step can be concisely expressed as:

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

- $\mathrm{C}(\mathrm{DT}_k, \mathrm{p}_{k+1})$ is the cavity of p_{k+1} in DT_k
- $\mathrm{B}(\mathrm{DT}_k, \mathrm{p}_{k+1})$ is the is the Delaunay Ball of p_{k+1} in DT_k
- This process is repeated until the full Delaunay triangulation is obtained.
- It can be broken into three parts. First, a walk to find the tetrahedron that contains p_{k+1}. Then, a search to find the cavity. Finally, a constructive process to add the Delaunay Ball.

Bowyer-Watson Algorithm

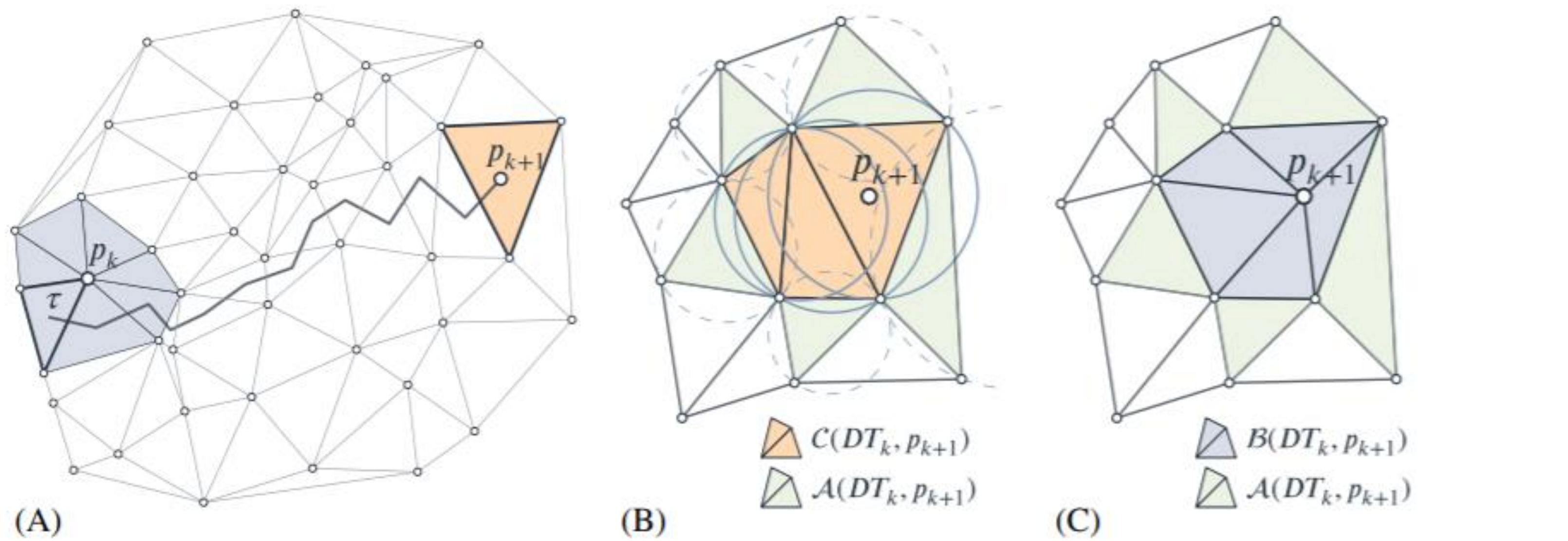
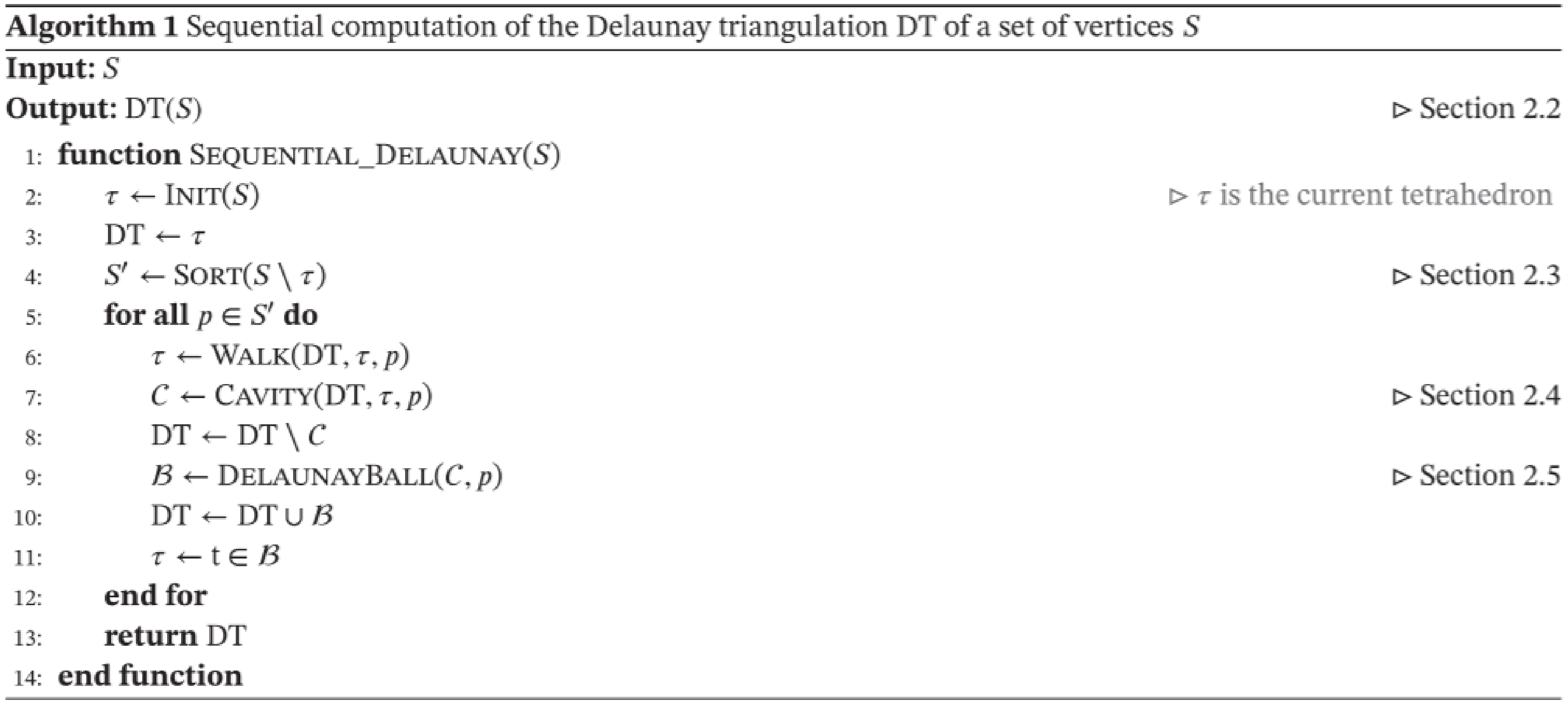

p_{k+1}
p_k
τ
(A)
p_{k+1}
$\mathcal{C}(DT_k, p_{k+1})$
$\mathcal{A}(DT_k, p_{k+1})$
(B)
p_{k+1}
$\mathcal{B}(DT_k, p_{k+1})$
$\mathcal{A}(DT_k, p_{k+1})$
(C)

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, whereas 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

Bowyer-Watson Algorithm



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

Input: S

Output: DT(S) ▷ Section 2.2

```
1:  function SEQUENTIAL_DELAUNAY($S$)
2:      $\tau \leftarrow$ INIT($S$)                      ▷ $\tau$ is the current tetrahedron
3:      DT $\leftarrow \tau$
4:      $S' \leftarrow$ SORT($S \setminus \tau$)                    ▷ Section 2.3
5:      for all $p \in S'$ do
6:          $\tau \leftarrow$ WALK(DT, $\tau$, $p$)
7:          $\mathcal{C} \leftarrow$ CAVITY(DT, $\tau$, $p$)            ▷ Section 2.4
8:          DT $\leftarrow$ DT $\setminus \mathcal{C}$
9:          $\mathcal{B} \leftarrow$ DELAUNAYBALL($\mathcal{C}$, $p$)         ▷ Section 2.5
10:         DT $\leftarrow$ DT $\cup \mathcal{B}$
11:         $\tau \leftarrow$ t $\in \mathcal{B}$
12:     end for
13:     return DT
14: end function
```

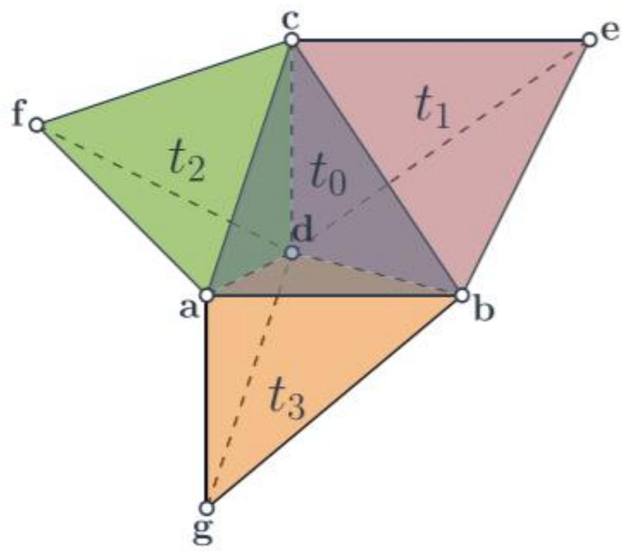
Optimization 1: Mesh Representation

- 32-byte aligned struct to represent vertices (alignment allows cache efficiency because objects don’t span across cache lines)
- Parallel arrays to represent tetrahedra – each tetrahedron gets 4 entries in 3 arrays, representing vertices, neighbors, and sub-determinants respectively.
- The indices of these parallel arrays are specifically set up to make the walk step fast.
- Overall, the mesh struct contains a list of vertices and a list of tetrahedra.

Optimization 1: Mesh Representation

```
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;
```

Optimization 1: Mesh Representation



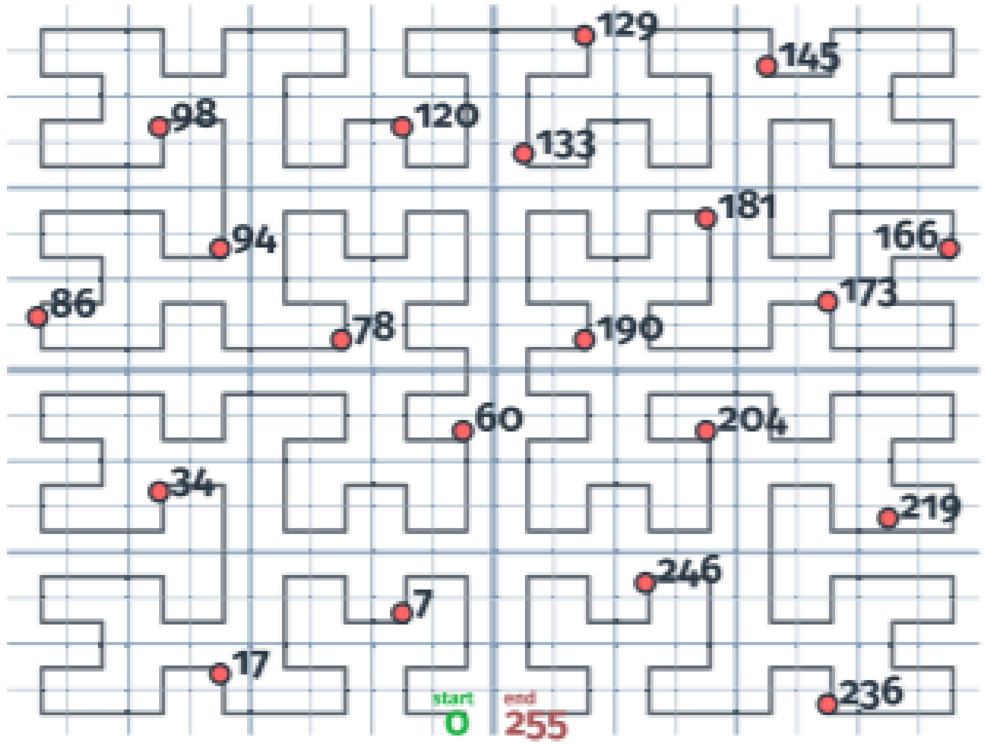
memory index	`vertex_ID`	`neighbor_ID`
$4t_0$	a	$4t_1+3$
$4t_0+1$	b	$4t_2+3$
$4t_0+2$	c	$4t_3+3$
$4t_0+3$	d	–
⋮		
$4t_1$	b	–
$4t_1+1$	c	–
$4t_1+2$	d	–
$4t_1+3$	e	$4t_0+0$
⋮		
$4t_2$	a	–
$4t_2+1$	d	–
$4t_2+2$	c	–
$4t_2+3$	f	$4t_0+1$
⋮		
$4t_3$	a	–
$4t_3+1$	b	–
$4t_3+2$	d	–
$4t_3+3$	g	$4t_0+2$

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

Optimization 2: Sort

- The walk and cavity complexity depends on the order in which the points are inserted.
- It has been shown by Boissonnat, et al that a biased-randomized insertion order (BRIO) achieves a small, constant number of steps during the walks and a small average cavity size.
- BRIO works by separating the points into successively larger rounds of insertions, and ordering each round by spatial locality.
- The spatial sorting is done by assigning the points indices based on their position on a Hilbert/Moore space-filling curve, and then sorting them by their indices (tradeoff in cost vs resolution).
- The Hilbert/Moore indices tend to be relatively small in range, because they use a low-resolution curve with respect to the number of points.

Moore Space-Filling Curves

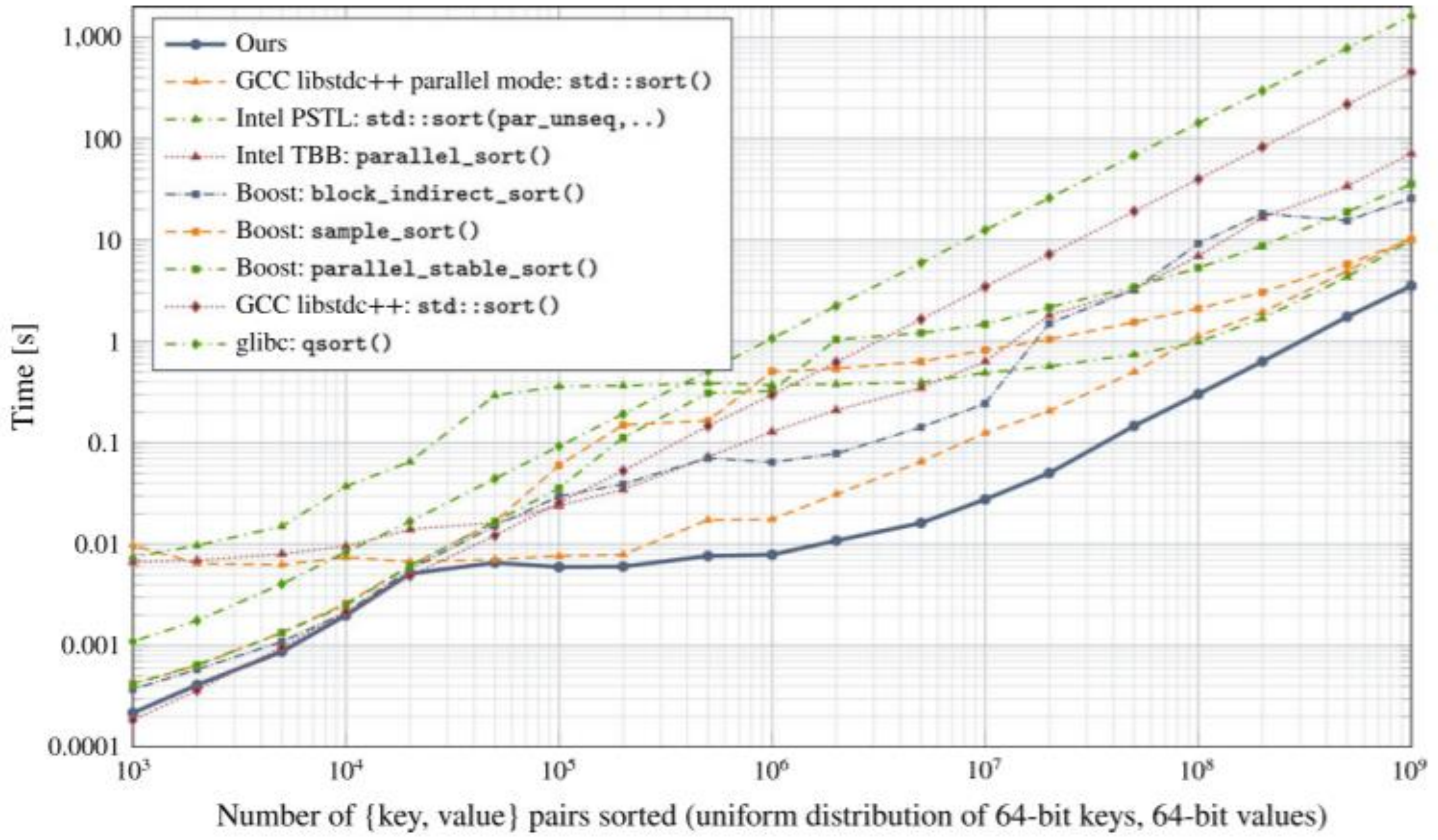


2^mx2^mx2^m lattice, choose m = $k\log_2(n)$ for a constant number of overlaps

Optimization 2: Sort

- The paper provides a highly optimized, parallel radix sort routine that operates specifically over Hilbert/Moore indices and provides essentially linear time sort.
- It is multithreaded and uses vectorized instructions (e.g. AVX 512).
- Because it takes advantage of the short keys, it outperforms existing sort routines.
- The keys are short because the choice of m is O(logn), so the number of keys is logarithmic in the number of actual points to be sorted.

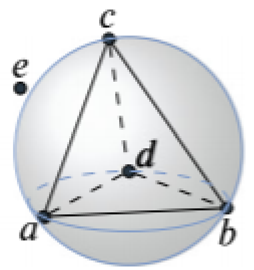
Optimization 2: Sort



Optimization 3: Geometric Predicates

- During the cavity step of the algorithm, we perform a BFS on the tetrahedra, using the inSphere geometric predicate to determine whether the new point is inside the sphere of each tetrahedron.
- This calculation is a very expensive 4x4 matrix determinant
- It is optimized by breaking it into a sum of products of 4 3x3 sub-determinants that only depend on the vertices of the tetrahedron in question. These sub-determinants can be precomputed and cached for each tetrahedron.
- There is also an orient3D geometric predicate that is used during the walk step to determine which face to walk through at each step.

Optimization 3: Geometric Predicates



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

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

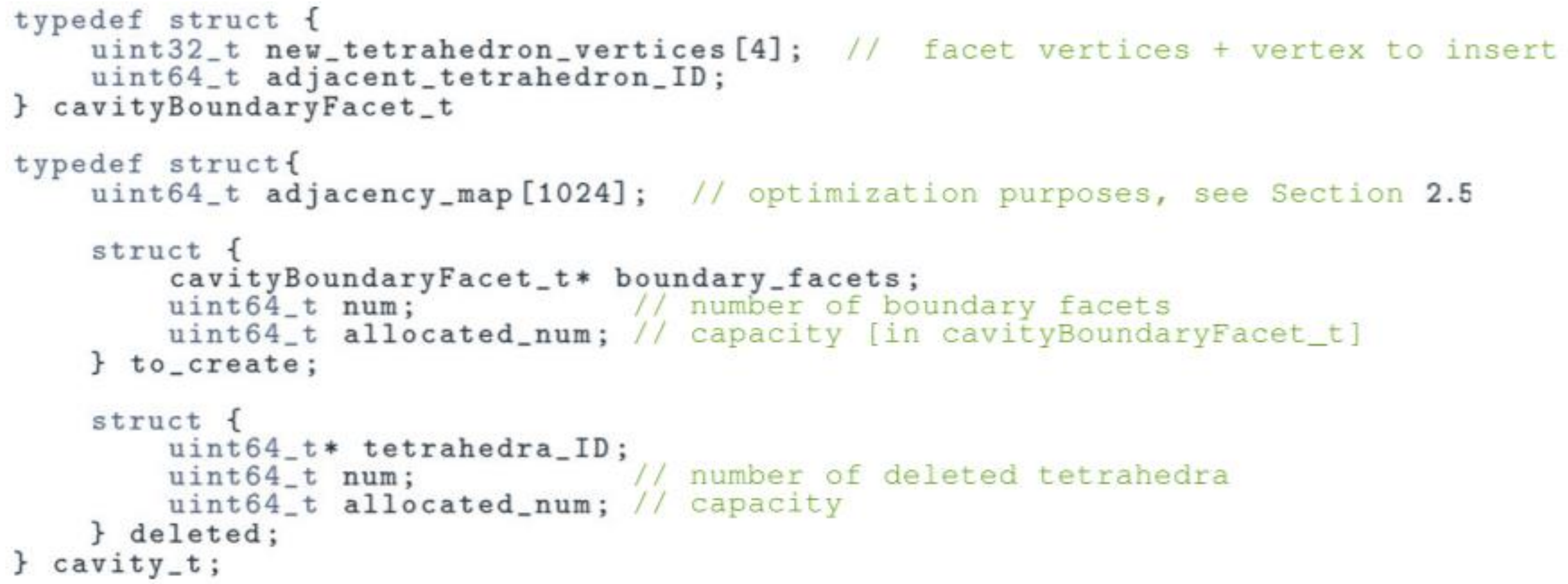
Optimization 4: Cavity Representation

- The most expensive step of the triangulation is recalculating the adjacencies of the newly inserted tetrahedra in the Delaunay Ball.
- The paper proposes a two step process to perform this process very efficiently.
- First, it notes that the adjacency opposite the new point p_{k+1} of every tetrahedron is already known from the BFS step during the cavity exploration (because we would have explored the boundary layer around the cavity).
- For the remainder of the points, it creates an nxn lookup table that allows for the adjacencies to be read out in constant time.

Optimization 4: Cavity Representation

- Each vertex p_j is assigned index i_j in [0, n) for the lookup table.
- Every new tetrahedron t_i consists of points $\{p_{k+1}, p_1, p_2, p_3\}$, where p_1, p_2, p_3 are on the boundary of the cavity. Then, adjacency index $4t_i+1$ is written to position (i_2, i_3). Then, the adjacent tetrahedron that shares edge p_2p_3 will be able to directly read this value out as its adjacency index, and similarly for all the other directed edges on the boundary.
- This lookup is predicated on the idea that the number of boundary vertices of a cavity is small, bounded by a prechosen n (chosen as 32). The paper demonstrates that this value is sufficient for almost all cases, and provides a fallback linear search in the rare case of a larger cavity.

Optimization 4: Cavity Representation



```
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;
```

Implementation Details

- The implementation as described does not work if the newly inserted point does not already lie within the convex hull of already inserted points.
- In order to solve this problem, the paper introduces a ghost point at infinity that forms tetrahedra with the entire boundary of the Delaunay triangulation. In this way, it covers the entire space.
- The orient3D subroutine has to be slightly modified to account for the infinite tetrahedra, but it otherwise works with no modification in the algorithm.

Results

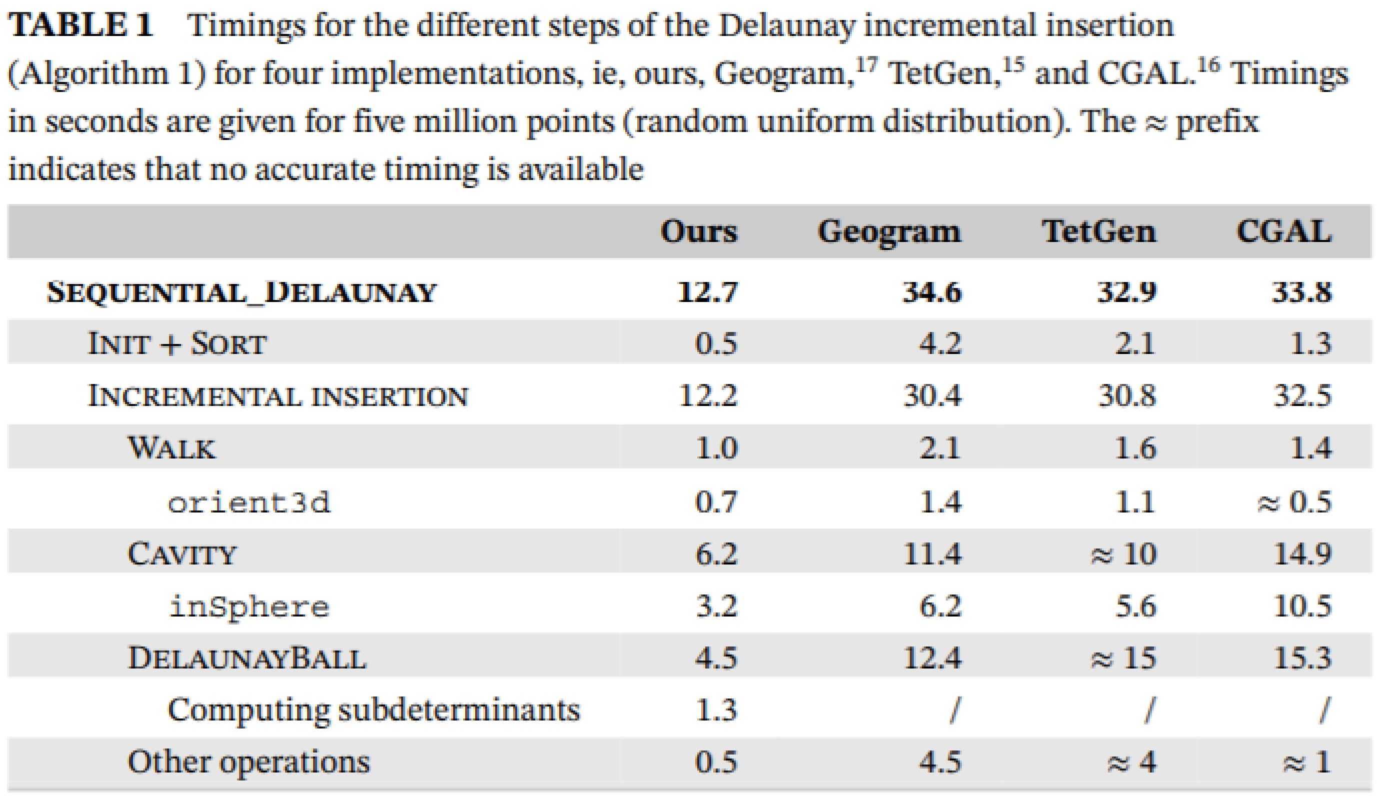
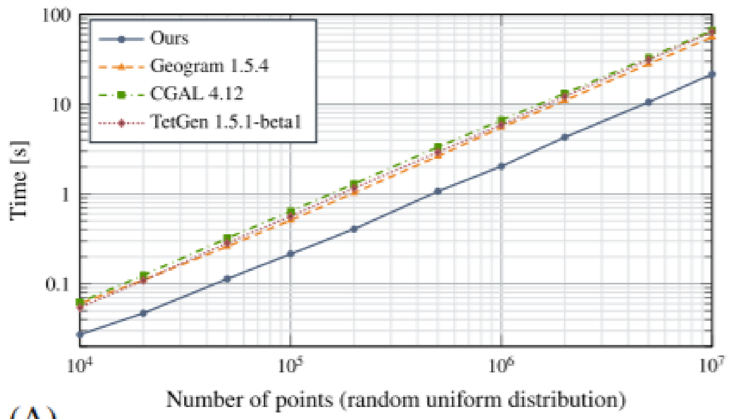


TABLE 1 Timings for the different steps of the Delaunay incremental insertion (Algorithm 1) for four implementations, ie, ours, Geogram,[17] TetGen,[15] and CGAL.[16] Timings in seconds are given for five million points (random uniform distribution). The ≈ prefix indicates that no accurate timing is available

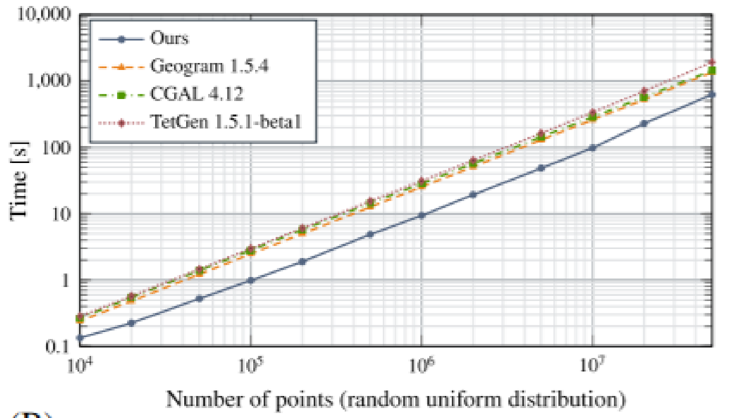
	Ours	Geogram	TetGen	CGAL
SEQUENTIAL_DELAUNAY	**12.7**	**34.6**	**32.9**	**33.8**
INIT + SORT	0.5	4.2	2.1	1.3
INCREMENTAL INSERTION	12.2	30.4	30.8	32.5
WALK	1.0	2.1	1.6	1.4
`orient3d`	0.7	1.4	1.1	≈ 0.5
CAVITY	6.2	11.4	≈ 10	14.9
`inSphere`	3.2	6.2	5.6	10.5
DELAUNAYBALL	4.5	12.4	≈ 15	15.3
Computing subdeterminants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

Results



# 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)



# 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)

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. A, Intel® Core™ i7-6700HQ CPU, maximum core frequency of 3.5Ghz; B, Intel® Xeon Phi™ 7210 CPU, maximum core frequency of 1.5Ghz [Colour figure can be viewed at wileyonlinelibrary.com]

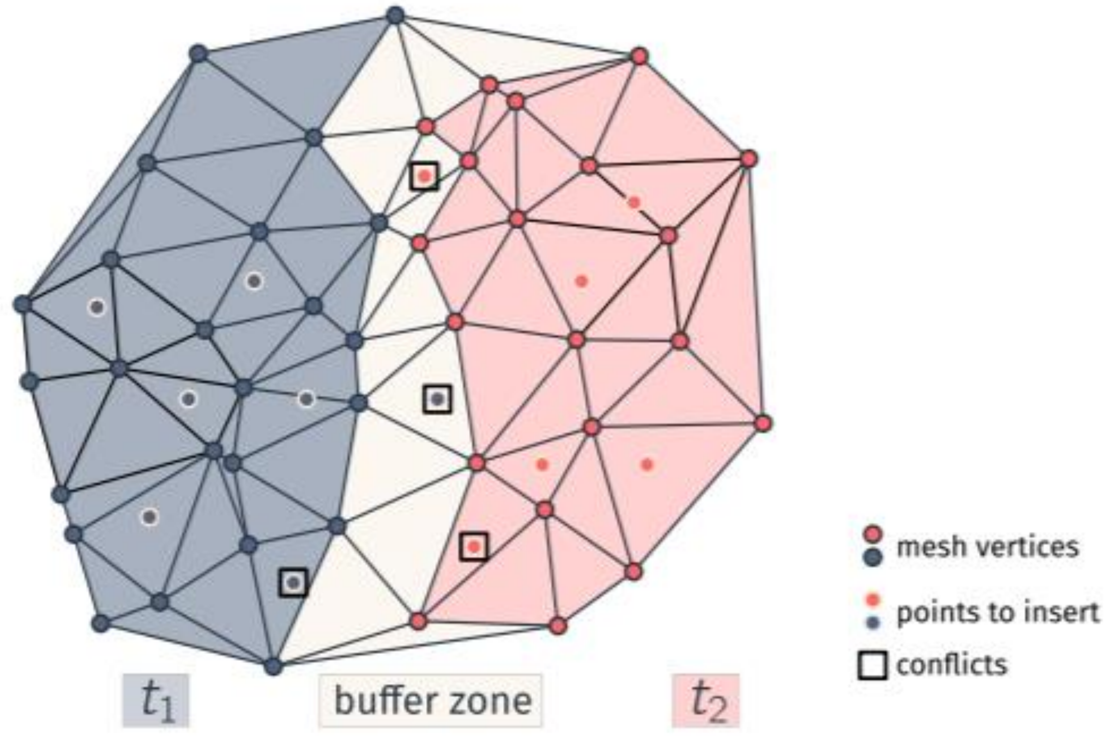
Parallelization

- Focuses on three key ideas to parallelize the sequential algorithm.
- The first is to divide into fully independent subproblems and have a thread work on each of them sequentially.
- The second is to minimize synchronization of shared memory access.
- The third is to minimize memory writes in general to save memory bandwidth.

Partitioning

- In order to properly parallelize without data races, there are two key points to be observed.
- First, the two threads cannot access the same tetrahedra (the cavities must not overlap, the boundaries must not overlap with the other cavities, and the walks cannot go through other thread’s boundaries).
- In order to observe these rules without synchronization overhead, it is sufficient to partition the points into independent sets, and to only allow a thread to process a tetrahedron if it owns at least 3 of its vertices.
- The second is that two threads cannot insert a tetrahedron into the same index of the global mesh array. This is done with explicit synchronization.

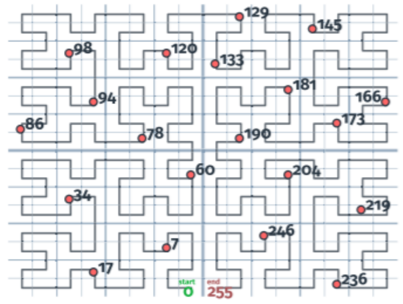
Partitioning



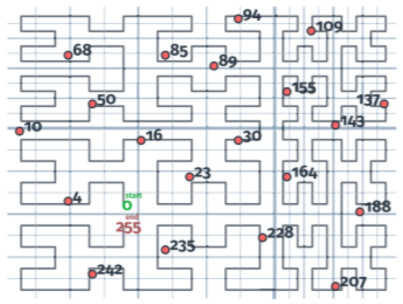
Partitioning

- The partitioning is again done with Moore space-filling curves. The curve is drawn with an explicit start and end, and then the points are assigned indices on the curve. Then, they are split equally amongst the threads.
- Each partition owns the tetrahedra that have at least 3 vertices in that partition.
- After all the insertions, there are some points left over in the buffer regions whose cavities span multiple partitions. The Moore curve is then transformed and rotated, and the points are repartitioned.
- In order to terminate, the number of threads is decreased proportionally with the number of points so that the buffer regions don’t get too large. Eventually, the points are inserted sequentially.

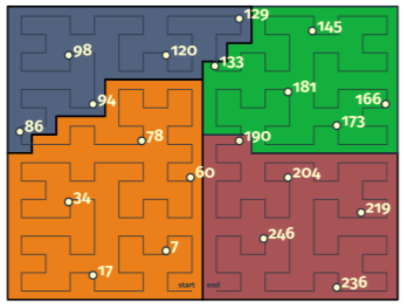
Partitioning



(A)

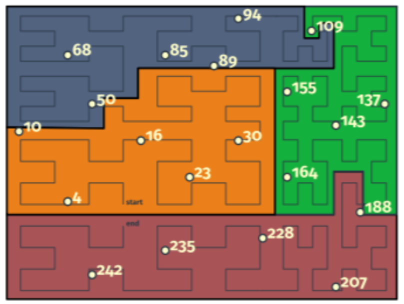


(B)



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

(C)



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

(D)

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

Partitioning

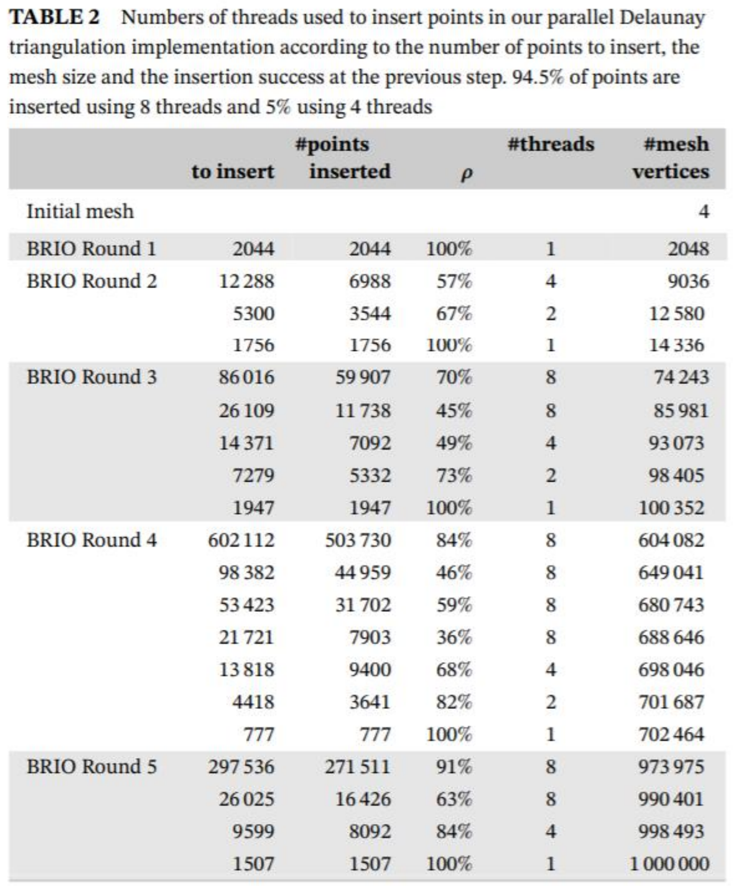


TABLE 2 Numbers of threads used to insert points in our parallel Delaunay triangulation implementation according to the number of points to insert, the mesh size and the insertion success at the previous step. 94.5% of points are inserted using 8 threads and 5% using 4 threads

	#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

Partitioning

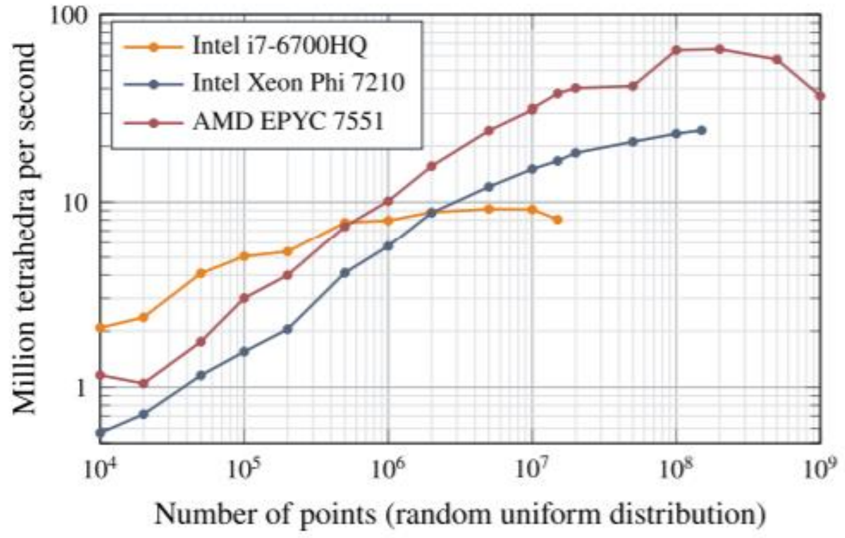


FIGURE 9 Number of tetrahedra created per second by our parallel implementation for different number of points. Tetrahedra are created more quickly when there is a lot of points because the proportion of conflicts is lower. An average rate of 65 million tetrahedra created per second is obtained on the EPYC

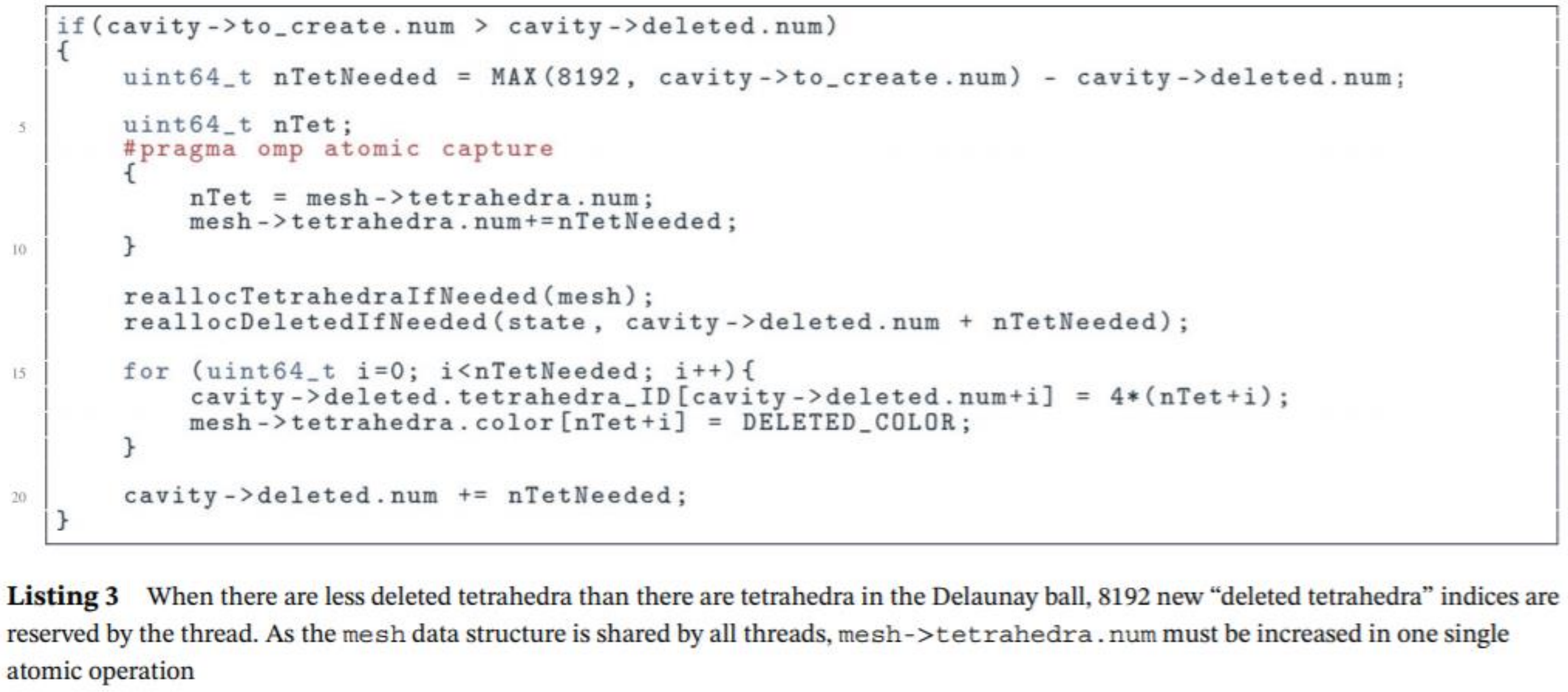
Data Structures

- The sub-determinant optimization is removed for the parallel implementation, because the memory bandwidth becomes a bottleneck (due to all the cores working in parallel) rather than the computational time.
- The extra parallel array is just used to store whether the tetrahedra is deleted or not.

Synchronized Memory Access

- All of the threads have to insert into the global mesh array. This requires memory synchronization.
- As a first order optimization, the threads each have their own cavity object and reuse the deleted tetrahedron spaces to insert new tetrahedra.
- If a thread runs out of deleted space and needs to insert more, it needs to atomically claim memory in the global mesh array.
- The threads claim memory in chunks of 8192 (experimentally chosen) in order to minimize the amount of synchronization required - tradeoff between thread synchronization and wasted memory.
- The thread synchronization is done with OpenMP.

Synchronized Memory Access



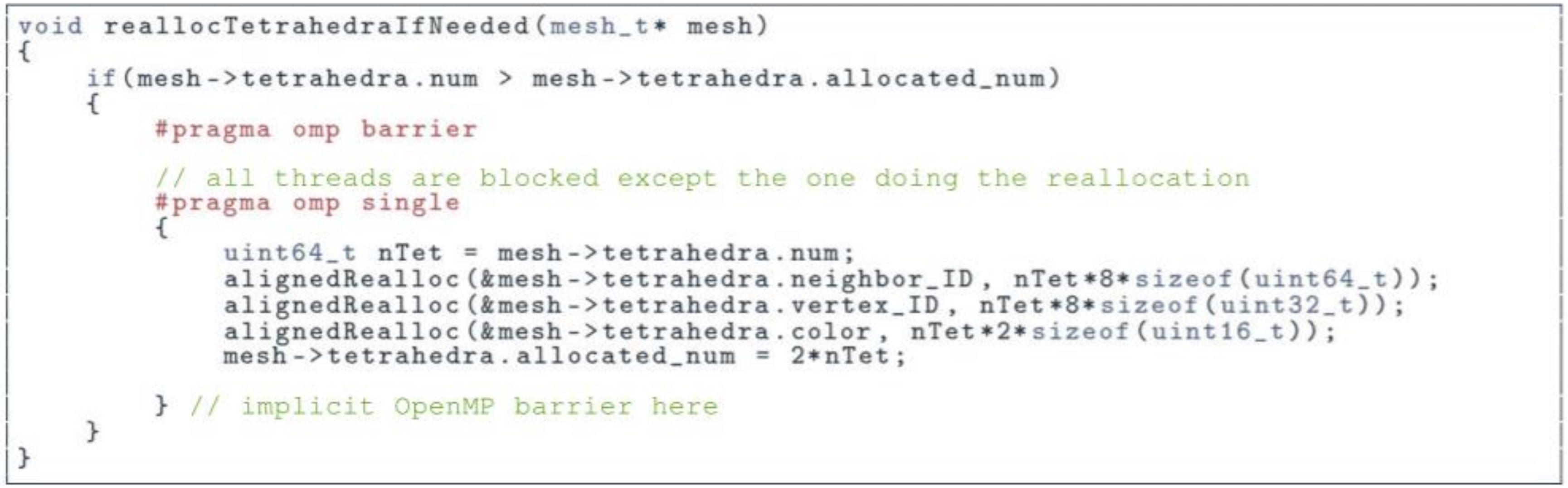
```
if(cavity->to_create.num > cavity->deleted.num)
{
    uint64_t nTetNeeded = MAX(8192, cavity->to_create.num) - cavity->deleted.num;

    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

Synchronized Memory Access

- If the space entirely runs out of the global mesh array, more space needs to be allocated in the global array.
- This is done by a single thread in a locked section.
- This does not occur that often because the static space requirements of the entire algorithm can be very well approximated based on the input size.

Synchronized Memory Access



```
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

Results

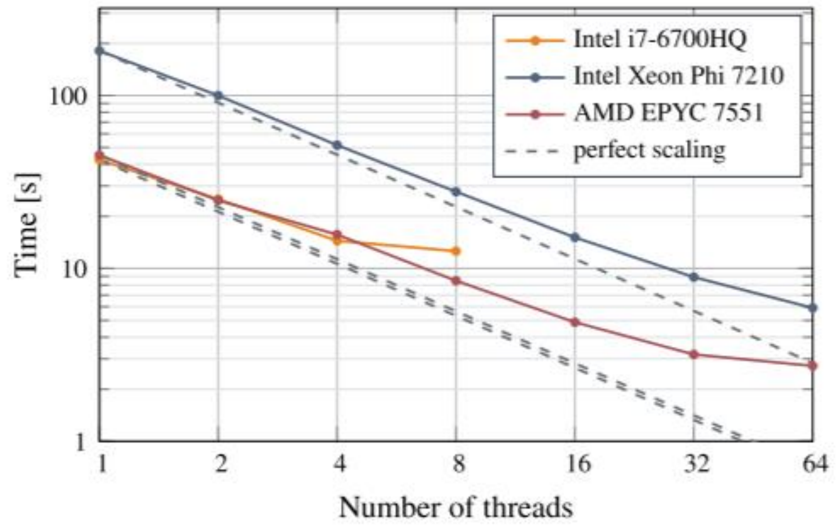
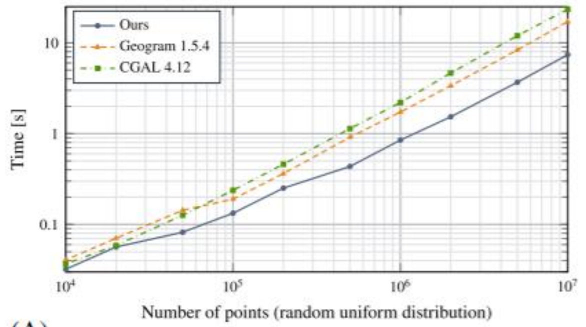


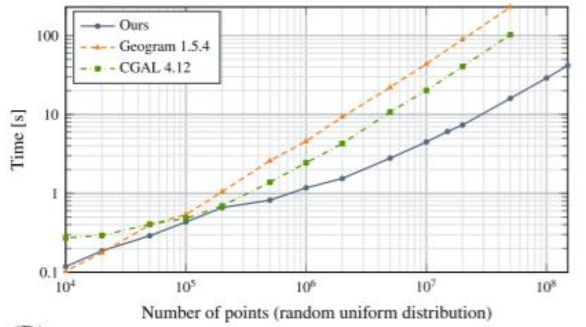
FIGURE 10 Strong scaling of our parallel Delaunay for a random uniform distribution of 15 million points, resulting in over 100 million tetrahedra on three machines, ie, a quad-core laptop, an Intel Xeon Phi with 64 cores, and a dual-socket AMD EPYC 2 × 32 cores

Results



(A)

# 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



(B)

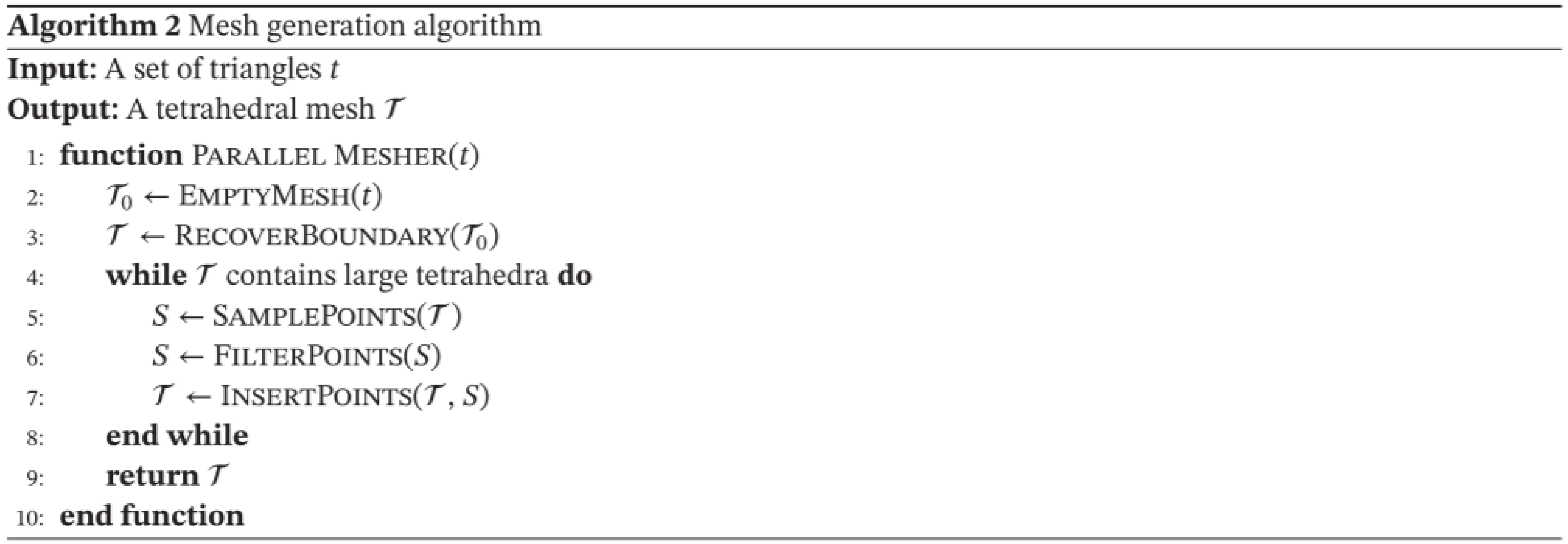
# vertices	10^4	10^5	10^6	10^7	10^8
Ours	0.11	0.43	1.17	4.48	28.95
Geogram	0.10	0.54	4.58	43.70	/
CGAL	0.27	0.48	2.44	20.15	/

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. A, 4-core Intel® Core™ i7-6700HQ CPU; B, 64-core Intel® Xeon Phi™ 7210 CPU [Colour figure can be viewed at wileyonlinelibrary.com]

Mesh Generation

- Mesh generation is the problem of taking a boundary and producing a tetrahedron mesh that has the given boundary.
- It includes Delaunay triangulation as a subprocess.
- The paper introduces a mesh generation routine based on existing work in which they implemented their new triangulation routine and the parallelization techniques discussed above.
- They demonstrate that their parallelization techniques extend to this practical application.

Mesh Generation



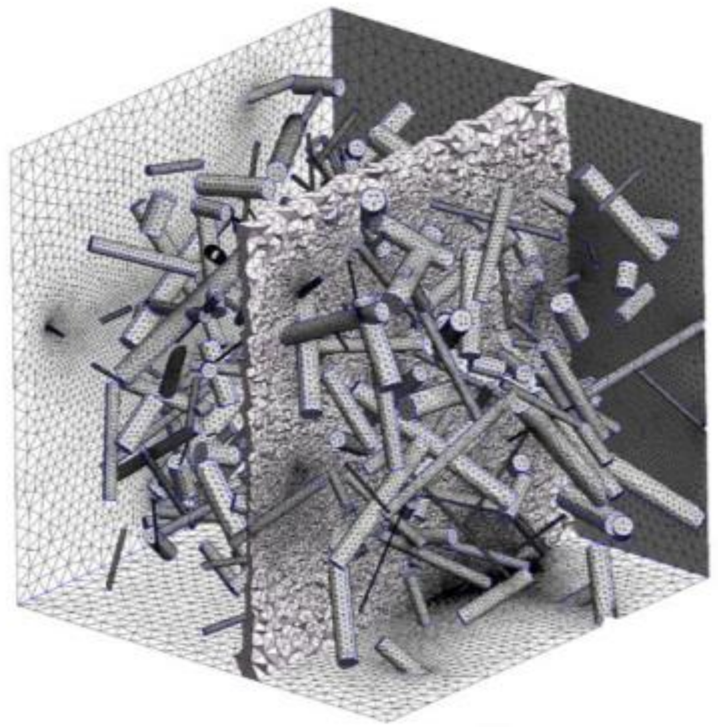
Algorithm 2 Mesh generation algorithm

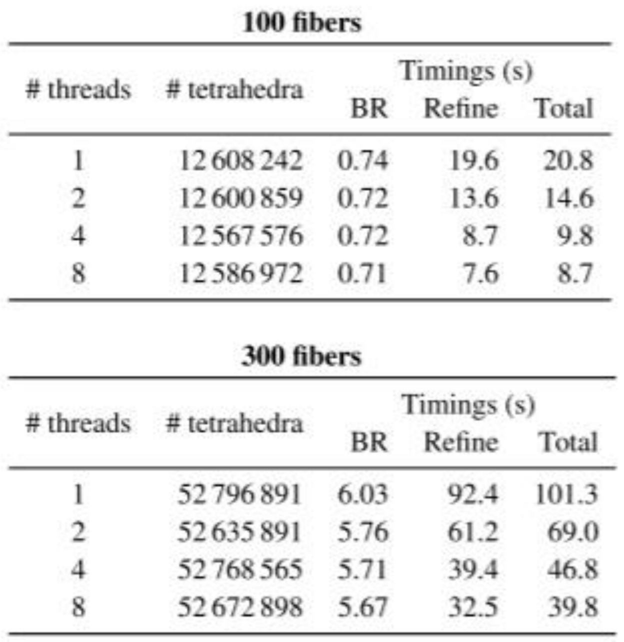
Input: A set of triangles t

Output: A tetrahedral mesh $\mathcal{T}$

```
function ParallelMesher($t$)
    $\mathcal{T}_0 \leftarrow$ EmptyMesh($t$)
    $\mathcal{T} \leftarrow$ RecoverBoundary($\mathcal{T}_0$)
    while $\mathcal{T}$ contains large tetrahedra do
        $S \leftarrow$ SamplePoints($\mathcal{T}$)
        $S \leftarrow$ FilterPoints($S$)
        $\mathcal{T} \leftarrow$ InsertPoints($\mathcal{T}, S$)
    end while
    return $\mathcal{T}$
end function
```

Mesh Generation





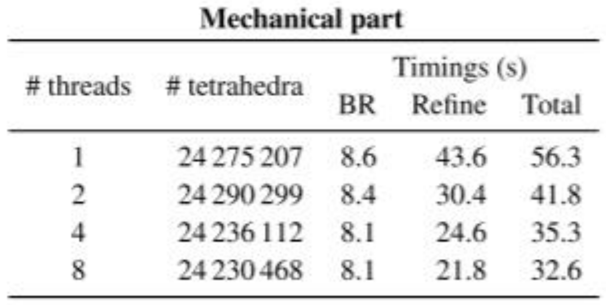
100 fibers

# 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

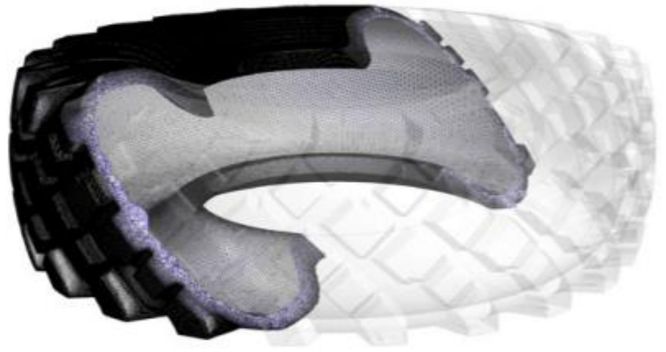
Mesh Generation



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

Mesh Generation

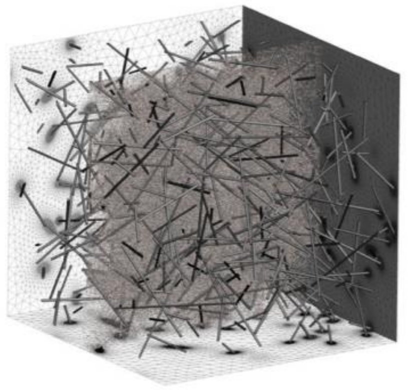


Truck tire

# 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 an Intel® Core™ i7-6700HQ 4-core CPU. Wall clock times are given for the whole meshing process for 1 to 8 threads. They include I/Os (sequential), initial mesh generation (parallel), as well as sequential boundary recovery (BR), and parallel Delaunay refinement for which detailed timings are given [Colour figure can be viewed at wileyonlinelibrary.com]

Mesh Generation

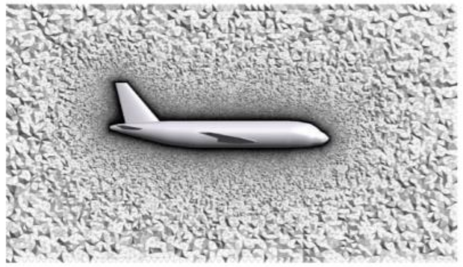


100 thin fibers

# 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 an AMD® EPYC 64-core machine. Wall clock times are given for the whole meshing process for 1 to 127 threads. They include I/Os (sequential), initial mesh generation (parallel), as well as sequential boundary recovery (BR), and parallel Delaunay refinement for which detailed timings are given

Related Work

- There are many existing implementations of the sequential Bowyer-Watson algorithm.
- There are many existing parallelizations as well.
- Many of them perform merge steps to combine independent triangulations; these merge steps tend to be expensive and serial.
- There are many divide-and-conquer strategies as well, but they use locks or barriers (Remacle, et al) to coordinate access to memory. This causes too much overhead on large numbers of cores.
- Loseille, et al do partitioning based on space-filling curves, but do not perform the repartitioning step in between rounds to insert points that couldn’t be inserted previously.

Thoughts

- Strengths
  - This paper has many novel ideas, such as portioning based on Moore curve and repartitioning by using transformations of the indices.
  - It takes advantage of cache operations to design very efficient data structures.
  - It manages to parallelize operations without synchronization overhead, allowing for high scalability.
  - It outperforms state-of-the-art Delaunay triangulation routines.
- Weaknesses
  - This paper focuses specifically on shared-memory machines and does not work on distributed-memory architectures.
  - Does not present the data that led to specific parameter choices.

Discussion

- How could these ideas be extended to distributed-memory architectures?
- What new architectural features would be most beneficial to speeding up this algorithm?
- Does the work capacity of an extra thread always outweigh the potential loss of memory bandwidth?