Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Engineering 00 (2015) 000–000

Procedia Engineering

www.elsevier.com/locate/procedia

A Two-Level Multithreaded Delaunay Kernel

Jean-François Remacle[a,c], Vincent Bertrand[a], Christophe Geuzaine[b]

[a] *Université catholique de Louvain, Institute of Mechanics, Materials and Civil Engineering (iMMC), Bâtiment Euler, Avenue Georges Lemaître 4, 1348 Louvain-la-Neuve, Belgium*

[b] *Université de Liège, Department of Electrical Engineering and Computer Science, Montefiore Institute B28, Grande Traverse 10, 4000 Liège, Belgium*

[c] *Department of Computational and Applied Mathematics, Rice University*

Abstract

This paper presents a fine grain parallel version of the 3D Delaunay Kernel procedure using the OpenMP (Open Multi-Processing) API. A set $S = \{p_1, \ldots, p_n\}$ of n points is taken as input. S is initially sorted along a space-filling curve so that two points that are close in the insertion order are also close geometrically. The sorted set of point is then divided into M subsets S_i, $1 \leq i \leq M$ of equal size n/M. The multithreaded version of the Delaunay kernel inserts M points at a time in the triangulation. OpenMP barriers provide the required synchronization that is needed after each multiple insertion in order to avoid data races. This simple approach exhibits two standard problems of parallel computing: load imbalance and parallel overheads. Those two issues are addressed using a two-level version of the multithreaded Delaunay kernel. Tests show that triangulations of about a billion tetrahedra can be generated on a 32 core machine (Intel Xeon E5-4610 v2 @ 2.30GHz with with 128 GB of memory) in less that 3 minutes of wall clock time, with a speedup of 18 compared to the single-threaded implementation.

Peer-review under responsibility of organizing committee of the 24th International Meshing Roundtable (IMR24).

Keywords: Delaunay triangulation, parallel computing, OpenMP

1. Introduction

In the last decades, the size of the finite element meshes that are used in industry for scientific computing has grown considerably due to the availability of massively parallel computers. It is nowadays not uncommon to generate meshes that have over 100 million tetrahedra. From a user's perspective, however, generating a mesh of a complex domain usually involves the generation of several intermediary meshes that are progressively enhanced in order to fulfill some adequate design requirements. Todays best 3D meshing algorithms are able to generate about five million tetrahedra per minute on one single core [11]. Each iteration in the meshing process thus takes long minutes and users eventually spend a significant portion of their time waiting for the mesh generator to provide outputs.

Computation platforms are increasingly based on multicore architectures with a large common shared memory but relatively low performance individual computing cores. The performance of today's mesh generation procedures, based on serial kernels, is thus largely stalling, and may actually degrade in the future.

Parallel mesh generation procedures have been developed for several years. Up to ten years ago, the great majority of parallel meshing algorithms worked at a coarse grain level (see [6] for an exhaustive survey). In such coarse grain algorithms, the 3D domain is first partitioned into subdomains; the internal 2D boundaries between the subdomains are then meshed in order to ensure the compatibility of the 3D meshes in adjacent subdomains; and serial meshing procedures are finally applied at the subdomain level [5,8,16]. The focus of these parallel algorithms is mostly about mesh size and not so much about fast meshing: indeed, generating very large meshes cannot be done on one single

Peer-review under responsibility of organizing committee of the 24th International Meshing Roundtable (IMR24).

* Jean-François Remacle.

CPU and it is thus very important to distribute the meshing generation process in order to have access to sufficient memory. Multicore platforms offer the advantage of low latency communications through shared memory, leading to the possibility of fine-grain parallelism in the meshing kernels. Recently, CGAL [3] developers have proposed a fine-grained multithreaded version of the Delaunay triangulation in 3D based on locks [3]. Different locking strategies have been analyzed, with typical speedups of 5 with 8 computing cores, but at the cost of a fairly complex algorithmic implementation.

This paper is essentially a preliminary work in multithreading the meshing process of Gmsh [12]. The overall aim is similar to CGAL's, i.e., to increase the speed of the central piece of the mesh generator through a fine-grained parallelization of the Delaunay kernel. The main difference is that we focus here on the simplicity of the implementation, using the OpenMP (Open Multi-Processing) Application Programming Interface (API).

The paper is organized as follows. In Section §2, we give the main ingredients that allow us to build an efficient 3D Delaunay triangulator:

1. The points are initially separated in groups of increasing sizes. Then, points are sorted within each group using a Hilbert space-filling curve [1,13].
2. While this sorting process is done, the insertion of the points has a linear complexity in most of the situations.
3. Geometrical predicates are carefully crafted in order both to ensure the robustness of the triangulation and good computational efficiency.

With those ingredients, we demonstrate that our algorithm compares well with the best serial implementations available: a set of one million points in $\mathbb{R}^3$ is tetrahedralized in less that 11 seconds of wall clock time on one single core of an Intel Xeon E5-4610 v2 @ 2.30GHz.

In §3 a fine grain-parallelization of this Delaunay kernel is then proposed. The Hilbert curve passing through all the points is snipped into M equal parts and each of the M computational threads take care of one part of the curve. M points are inserted at once, in parallel. The procedure is implemented using OpenMP, the Open Multi-Processing API. A significant speedup is observed at that stage: the time for tetrahedralizing one million points can be reduced by a factor 4.6 on 8 cores. Yet, using more cores, the parallel performance decreases: 32 cores are required to obtain a speedup of 9.33.

In §3, we show that the degradation of the parallel efficiency of this simple approach when the number of cores increases is predictable: the overhead of OpenMP constructs is magnified and the load gets less and less well balanced when the number of cores increases.

Mitigation solutions for these two problems are presented in n §4, using a two-level extension of the multithreaded Delaunay kernel, where $M_2 > 1$ points are inserted at once by each of the cores. This second level of insertion has the effect of reducing the number of OpenMP barriers by a factor M_2 and to better balance the load by averaging the computations over M_2 point insertions. This two-level approach has however the disadvantage of reducing data locality on each thread. Moreover, M_2 cannot be increased above a certain level (typically $M_2 \leq 8$). Nevertheless, the two-level multithreaded Delaunay kernel allows to speed up the triangulation by a factor of 13.2 on 32 cores, leading to the tetrahedralization of about 1 million points per second. On a 32 core Intel Xeon E5-4610 v2 @ 2.30GHz with with 128 GB of memory, the two-level multithreaded Delaunay kernel with $M_2 = 8$ generates more than a billion tetrahedra (150 million points) in 142.8 seconds of wall clock time.

2. The Delaunay Kernel

In what follows, a triangle is the generic term for a triangle in 2D or a tetrahedron in 3D. A triangulation $T(S)$ of S is a set of non overlapping triangles that exactly covers the convex hull $\Omega(S)$ with all points of S being among the vertices of the triangulation. The Delaunay triangulation $\mathrm{DT}(S)$ is such that the empty circumcircle of any triangle in $\mathrm{DT}(S)$ is empty, i.e., it contains no point of S. Delaunay triangulations are popular in the meshing community because fast algorithms exist that allow to generate $\mathrm{DT}(S)$ in $O(n \log n)$ complexity.

The fastest algorithms that allow to build $\mathrm{DT}(S)$ are based on the Delaunay kernel [10]. Let DT_k be the Delaunay triangulation of a point set $S_k = \{p_1, \ldots, p_k\} \subset \mathbb{R}^d$. The *Delaunay kernel* is a procedure that allow the incremental insertion of a given point $p_{k+1} \in \Omega(S_k)$ into DT_k and to build the Delaunay triangulation DT_{k+1} of $S_{k+1} = \{p_1, \ldots, p_k, p_{k+1}\}$. The *Delaunay kernel* can be written in the following abstract manner:

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

where the Delaunay cavity $C(\mathrm{DT}_k, p_{k+1})$ is the set of all triangles whose circumcircles contain the new point p_{k+1} (see Figure 1; the triangles of the cavity cannot belong to DT_{k+1}) and the Delaunay ball $\mathcal{B}(\mathrm{DT}_k, p_{k+1})$ is a set of triangles that fill the polyhedral hole that has been left empty while removing the Delaunay cavity $C(\mathrm{DT}_k, p_{k+1})$ from DT_k.

The most important building block in any implementation of the Delaunay kernel is the computation of the simply connected Delaunay cavity $C(\mathrm{DT}_k, p_{k+1})$ [14]. One seed triangle t should be found that has its circumcircle containing p_{k+1}. Then, computing the Delaunay cavity can be done locally using a depth-first search technique.

There are essentially two ways to compute the seed triangle t efficiently. The history of all triangles may be maintained and a search is made into the history dag to insert a new point [2]. This solution leads to logarithmic complexity for finding the seeding triangle t. It also requires some memory overhead because all triangles of all stages have to be maintained. A more straightforward algorithm consists in doing a "walk" into the triangulation: starting from any triangle τ, finding the next triangle in the path to t consists in choosing one of the three neighbors τ_j of τ in such a way that p_{k+1} is on the other side of τ of the edge that is common to τ and τ_j. This walk can be shown to always terminate using Edelsbrunner's acyclic theorem [9]. Figure 2 presents the construction of a Delaunay cavity using this approach. The walking path that allows to find the seed t is depicted with a thick black line. The Delaunay cavity of point p_{k+1} is shown in red. In our implementation, we choose τ as the first element of the Delaunay ball $\mathcal{B}(\mathrm{DT}_{k-1}, p_k)$. We will see later on why this is actually a good idea. Listing 1 shows a typical implementation of the Delaunay kernel.

The top half of Table 1 presents the average number of walking steps N_{walk} required to find an initial triangle of the Delaunay cavity in terms of the number of points to insert n, when the points are inserted randomly. In 2D N_{walk} is of the order $O(n^{1/2})$ as expected. It is interesting to see that the CPU time for computing the triangulation is asymptotically lower in 3D because N_{walk} is of the order $O(n^{1/3})$.

This behaviour can be improved dramatically by sorting the points in such a way that two successive points in the set are close to each other geometrically. On Figure 3, a set of 10^5 points are sorted using a Hilbert curve (two successive points in the sorted list are linked with a solid line). In the context of the Delaunay kernel, this kind of data locality can decrease the number of local searches N_{walk} that are required to find the next invalid triangle: one can use as a starting triangle t for point p_{k+1} one of the triangles of cavity $C(\mathrm{DT}_{k-1}, p_k)$. In practice, points are initially separated in groups of increasing sizes. Then, points are sorted within each group using a Hilbert space-filling curve. This has the advantage to produce smaller Delaunay cavities during the incremental insertion process than a simple Hilbert sort. Adding some randomness in the insertion process allows to reduce the average cavity size. Here, we

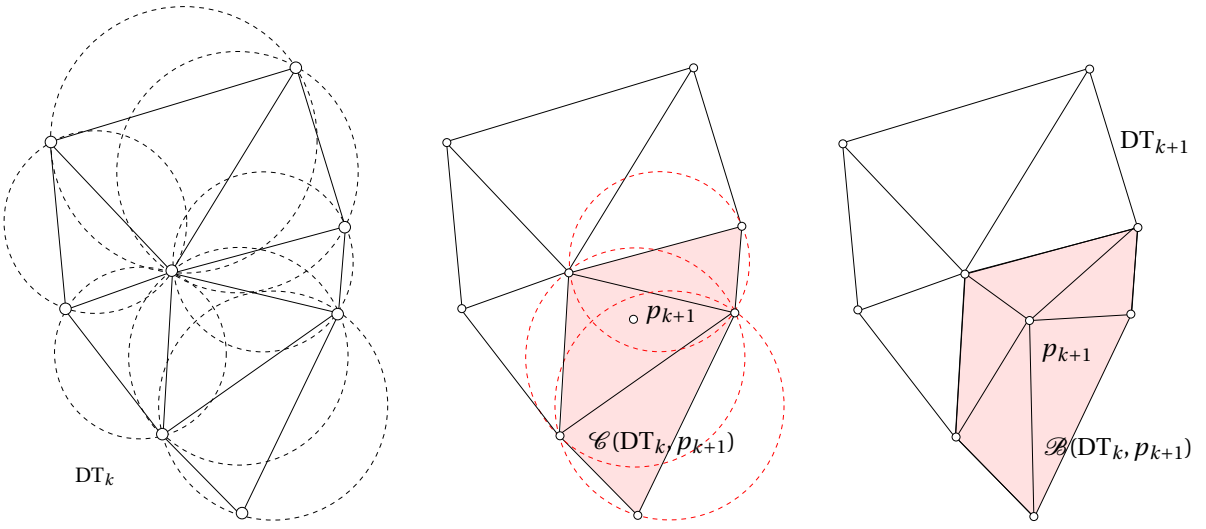

Fig. 1. Delaunay triangulation DT_k (left), Delaunay cavity $C_p(\mathrm{DT}_k, p_{k+1})$ (center) and $\mathrm{DT}_{k+1} = \mathrm{DT}_k - C(\mathrm{DT}_k, p_{k+1}) + \mathcal{B}(\mathrm{DT}_k, p_{k+1})$ (right).

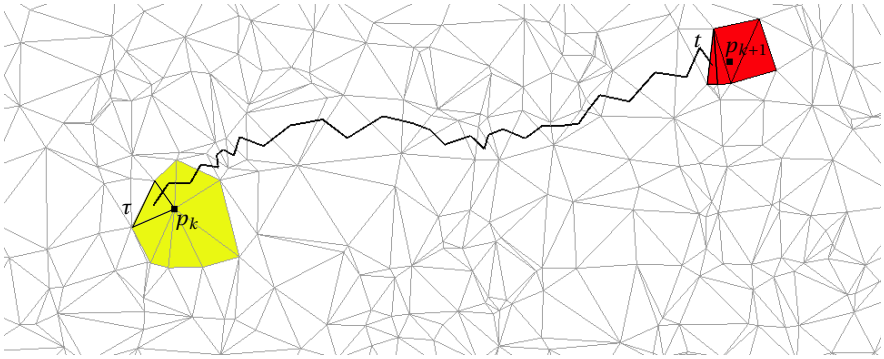

Fig. 2. Delaunay cavity $C(\mathrm{DT}_k, p_{k+1})$ in red and Delaunay cavity $C(\mathrm{DT}_{k-1}, p_k)$ in yellow. A seed t for building $C(\mathrm{DT}_k, p_{k+1})$ is found by starting from an arbitrary triangle of $C(\mathrm{DT}_{k-1}, p_k)$ and walking through the triangulation. Here, 32 walking steps were necessary to find t.

```
void delaunaySerial (vector<Vertex> &v, vector<Tetrahedron> &t)
{
  sortPoints(v);
  Tetrahedron tau = T[0];

  for(int k = 0; k < v.size(); k++){
    Tetrahedron t = walk(tau, v[k]);
    vector<Tetrahedron> cavity = delaunayCavity(t, v[k]);
    vector<Tetrahedron> ball = delaunayBall(cavity, v[k]);
    tau = ball[0];
  }
}
```

Listing 1. Serial Delaunay procedure

n	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6
	2D (random insertion)				3D (random insertion)			
N_{walk}	23	73	230	727	17	38	85	186
t(sec)	$3.6\ 10^{-3}$	$9.1\ 10^{-2}$	3.98	187	$1.2\ 10^{-2}$	$1.8\ 10^{-1}$	3.42	73
	2D (Hilbert curve)				3D (Hilbert curve)			
N_{walk}	2.3	2.4	2.5	2.5	2.9	3.0	3.1	3.1
t(sec)	$2\ 10^{-3}$	$1.5\ 10^{-2}$	$1.5\ 10^{-1}$	1.47	$9.0\ 10^{-3}$	$7.5\ 10^{-2}$	$7.8\ 10^{-1}$	7.81

Table 1. Results of the Delaunay Triangulation algorithm applied to a set of random points uniformly distributed in $[0,1]^d$. The table compares timings and average number of local searches in the case where points are inserted in a random fashion and in the case where points are inserted along a Hilbert curve.

use the Biased Randomized Insertion Order (BRIO) approach [1] that allows to recover enough randomness in the process.

The bottom half of Table 1 presents the average number of walking steps N_{walk} as well as timings in the case where points are inserted along a Hilbert curve. The number of walks N_{walk} is now almost independent of n and both 2D and 3D point insertion algorithms have an overall linear complexity. The difference between 2D and 3D timings is essentially due to the size of Delaunay cavities: 4 on average in 2D and above 20 in 3D. Assuming that geometrical predicates are slightly more expensive to compute in 3D, a factor of about 6 between the 2D and 3D timings is

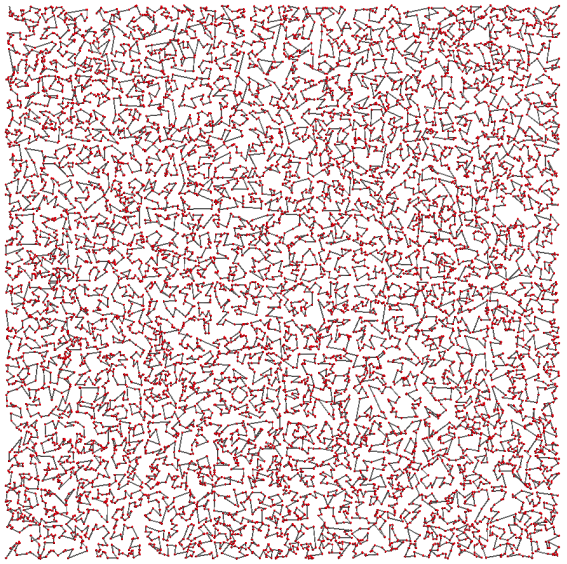

Fig. 3. Hilbert sort of a sets of 10^5 random points.

to be expected. Note that the Hilbert sorting has a $O(n \log n)$ complexity: the overall complexity of the Delaunay triangulation is $O(n \log n)$ as well. Yet, this is an asymptotic bound: with 1 million points, the time required to sort the points is still typically 100 times lower than the time for generating the mesh.

Before proceeding to the description of the proposed fine-grained parallelization of the Delaunay kernel, let us note that its serial performance is comparable to state of the art algorithms. For the same point set and for the same machine (same compiler and compiler options), Tetgen 1.5 [17] takes 7.7 seconds to tetrahedralize the set of points while ours takes 7.81 sec (a difference of less than 2%). The source code of both the 2D and the 3D version of the algorithm is available on Gmsh's website `www.gmsh.info`.

3. A Multithreaded Delaunay Kernel

Assume M computational threads that aim at inserting M points in the triangulation at the same time. At the end, each thread is going to insert $\frac{n}{M}$ points and our hope is of course to obtain a speedup close to M. The situation is of course not that simple: two points p_i and p_j can only be inserted at the same time in DT_k if their respective Delaunay cavities $C(\mathrm{DT}_k, p_i)$ and $C(\mathrm{DT}_k, p_j)$ do not overlap, i.e., if they do not have triangles in common:

$$C(\mathrm{DT}_k, p_i) \cap C(\mathrm{DT}_k, p_j) = \emptyset.$$

A non-overlapping situation is more likely to happen if points p_i and p_j are not close geometrically. For that purpose, we split the Hilbert curve into M equal parts and assign each part to one thread. Threads process their assigned points in order. A first chunk of points that correspond to about $20 \times M$ points is inserted in a serial fashion at first [3]. This allows to avoid inevitable cavities overlap in the first stages of the algorithm. Then, the rest of the points is inserted in parallel. Figure 4 shows different stages of the algorithm. Delaunay cavities are far apart most of the time thanks to the property of the Hilbert curve.

The *multithreaded Delaunay kernel* can be written in the following abstract manner:

$$\mathrm{DT}_{k+1} = \mathrm{DT}_k + \sum_{i=0}^{M-1} \left[-C(\mathrm{DT}_k, p_{k+i\frac{n}{M}}) + \mathcal{B}(\mathrm{DT}_k, p_{k+i\frac{n}{M}}) \right]. \tag{2}$$

We have implemented the multithreaded Delaunay kernel using OpenMP [7]. Two OpenMP barriers were used at each iteration k. A first barrier is used after the computation of the M cavities: every thread i has to complete its

```
  void delaunayParallel(vector<Vertex> &v, vector<Tetrahedron> &t, const int M)
  {
    sortPoints(v, M);
    Tetrahedron tau[M];
    const int n = v.size();
    for(int i = 0; i < M; i++) tau[i] = T[0];
#pragma omp parallel num_threads(M)
    {
      int i = omp_get_thread_num();
      for(int k = 0; k < n/M; k++){
        int index = k + i * n / M;
        Tetrahedron t = walk (tau[i], v[index]);
        vector<Tetrahedron> cavity = delaunayCavity(t, v[index]);
#pragma omp barrier
        if(noOverlap(cavity)){
          vector<Tetrahedron> ball = delaunayBall(cavity, v[index]);
          tau[i] = ball[0];
        }
#pragma omp barrier
      }
    }// end of omp pragma
  }
```

Listing 2. Parallel Delaunay procedure

cavity $C(\mathrm{DT}_k, p_{k+i\frac{n}{M}})$ at iteration k in order to be able to verify that the cavity does not overlap other cavities. When several cavities overlap, only the point corresponding to the smallest thread number is processed. The other points are delayed to the next iteration. A second barrier is used after the construction of $\mathcal{B}(\mathrm{DT}_k, p_{k+i\frac{n}{M}})$: every thread has to finish computing the Delaunay kernel in order to start iteration $k+1$ with a valid mesh. Listing 2 presents a simplified version of the code that has been used to do the computations. Even though this version is simplified, it has the main features of the actual code that is available in `www.gmsh.info`. It is important to note that Listings 1 and 2 are very similar: the multithreaded version of the Delaunay kernel that is proposed here only implies moderate modifications of an existing serial Delaunay insertion procedure.

Three potential threats can definitively harm the parallel speedup:

1. **Load balancing**: at stage k, the size of the M Delaunay cavities $C(\mathrm{DT}_k, p_{k+i\frac{n}{M}})$ may vary from one thread to another, leading to some load imbalance. Threads with small cavities will wait at the barrier for the thread that has the largest cavity.
2. **Overlaps**: it should be verified that cavity overlaps are rare events: each overlap adds one iteration in the insertion process.
3. **Overheads**: The OpenMP parallelization adds its own overhead: two barriers are used per iteration.

Let us investigate how those three threats may impact the scaling of the multithreaded Delaunay kernel.

3.1. Load Balancing

Figure 5 shows the distribution of sizes of 3D Delaunay cavities for a set of on million points that are uniformly distributed on the unit cube. The average size of Delaunay cavities is 21.2. Note that the way points are distributed does not change the cavity size distribution significantly: normal and Kuzmin point distribution have been tried with no spectacular change in the computations.

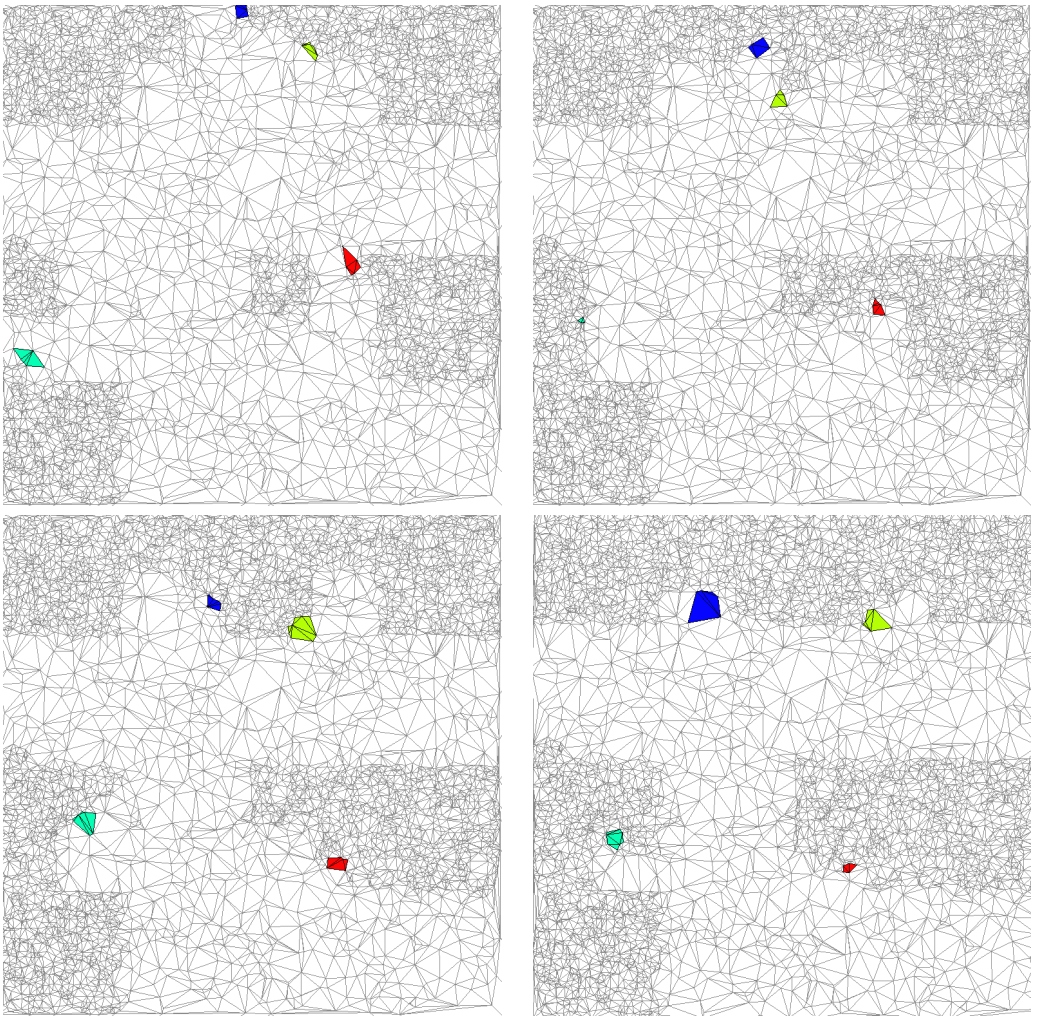

Fig. 4. Different stages of the multithreaded Delaunay insertion ($M = 4$). Delaunay cavities at the four different stages are colored w.r.t. their thread number.

The workload for thread i at a given step k is proportional to the cavity size $\dim(C(\mathrm{DT}_k, p_{k+i\frac{n}{M}}))$. The maximal theoretical speedup of the parallel algorithm is

$$S_{\max} = \frac{\sum_{k=1}^{n/M} \sum_{i=0}^{M-1} \dim(C(\mathrm{DT}_k, p_{k+i\frac{n}{M}}))}{\sum_{k=1}^{n/M} \max_{i=0}^{M-1} \dim(C(\mathrm{DT}_k, p_{k+i\frac{n}{M}}))},$$

which represents the total work done by all the threads divided by the maximal work done at every iteration k. Theoretical speedups $S_{\max}$ are reported in the top part of Table 2. The discrepancy between cavity sizes clearly has an impact on the theoretical speedup, and the parallel efficiency $S_{\max}/M$ decreases with the number of threads.

3.2. Overheads

The rest of Table 2 presents computational results, performed a 32 core Intel(R) Xeon(R) CPU E5-4610 v2 @ 2.30GHz with with 128 GB of memory. Microbenchmarks in [4] provide measurements of overheads incurred by OpenMP 3.0 constructs. The overhead $\tau_{barrier}$ of introducing one OpenMP barrier grows from 0.2 microseconds for 2 threads to 10 microseconds for 32 threads with a very irregular pattern in between (see Table 2). The total time spent

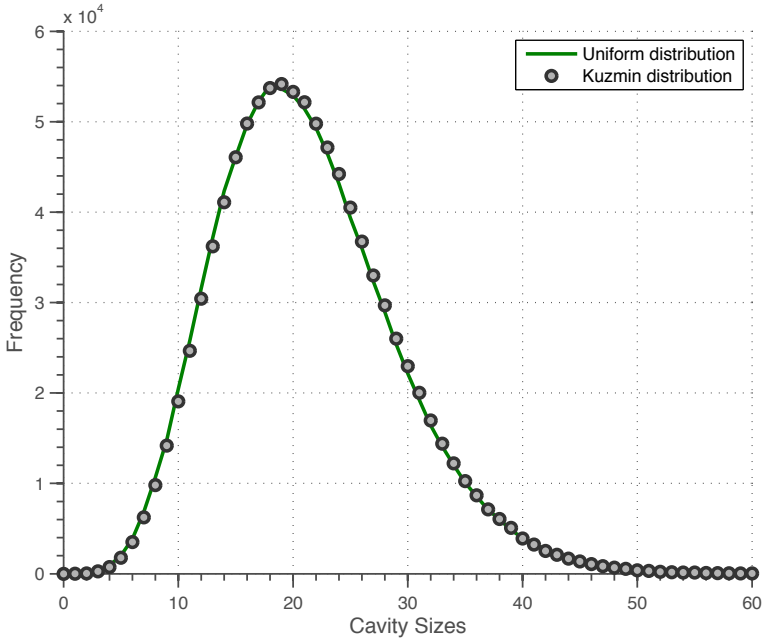

Fig. 5. Size of 3D Delaunay cavities for two different distributions (uniform and Kuzmin) of one million points.

M	1	2	4	8	16	32
$S_{\max}$	–	1.66	2.86	5.03	8.69	15.7
$S_{\max}/M$	–	0.83	0.71	0.62	0.54	0.49
$\tau_{barrier}$ (μsec)	–	0.2	0.4	2.0	3.5	10
$t_{barrier}$ (sec)	–	0.2	0.2	0.5	0.43	0.62
N_{miss}	–	8	15	42	184	392
S	–	1.63	2.76	4.60	6.62	9.33
S/M	–	0.81	0.69	0.57	0.41	0.29
t (sec)	11.2	6.84	4.05	2.43	1.69	1.20

Table 2. Scaling results for the multithreaded Delaunay kernel for triangulating 10^6 points (uniform distribution). M is the number of cores, $S_{\max}$ is the maximal speedup that can be attained, $\tau_{barrier}$ is the overhead per OpenMP Barrier, $t_{barrier}$ is the total time spent at barriers, N_{miss} is the number of times a point cannot be inserted, S is the actual speedup and t is the wall clock time. All computations performed on a 32 core Intel(R) Xeon(R) CPU E5-4610 v2 @ 2.30GHz with with 128 GB of memory.

at barriers is computed as

$$t_{barrier} = \frac{2\, n\, \tau_{barrier}}{M}.$$

This overhead is not decreasing with the number of threads which may cause serious drops in parallel efficiency.

3.3. Overlaps

We have reported in Table 2 the number of times N_{miss} one point cannot be inserted because its Delaunay cavity overlaps the cavity of another point that is inserted by another thread. This number is negligible compared to $n = 10^6$. On $M = 32$ cores, $N_{miss} = 392$ points were delayed which only represents 0.39% of the total. For a larger number of points $n = 10^7$, the number of overlaps is decreasing: $N_{miss} = 270$.

```
  void delaunayParallel2(vector<Vertex> &v, vector<Tetrahedron>& t,
                         const int M, const int M2)
  {
    sortPoints(v, M, M2);
    Tetrahedron tau[M][M2];
    const int n = v.size();
    for (int i = 0; i < M; i++)
      for (int j = 0; j < M2; j++) tau[i][j] = T[0];
#pragma omp parallel num_threads(M)
    {
      int i = omp_get_thread_num();
      for (int k = 0; k < n / M / M2; k++) {
        vector<Tetrahedron> cavity[M2];
        for (int j = 0; j < M2; j++) {
          int index = k + i * n / (M * M2) + j * n / M2;
          Tetrahedron t = walk (tau[i][j], v[index]);
          cavity[j] = delaunayCavity(t, v[index]);
        }
#pragma omp barrier
        for (int j = 0; j < M2; j++) {
          if(noOverlap(cavity[j])){
            vector<Tetrahedron> ball = delaunayBall(cavity[j], v[index]);
            tau[i][j] = ball[0];
          }
        }
#pragma omp barrier
      }
    }// end of omp pragma
  }
```

Listing 3. Parallel Delaunay procedure

3.4. Results

The last three rows in Table 2 show the actual speedups S and the computational efficiency S/M for computing the Delaunay triangulation of one million points, as well as the overall wall clock time t. These results reflect the two problems that have been described above. On 32 cores, a theoretical speedup of $S_{\max} = 14.9$ is expected but a true speedup of $S = 9.33$ is attained. The difference is caused by the overhead $t_{barrier} = 0.62$ sec caused by OpenMP barriers. Without this overhead, wall clock time would reduce to $1.20 - 0.62 = 0.58$ seconds, which is close to the serial wall clock time (11.2 seconds) divided by the theoretical maximum speedup (15.7). Even though the speedup S is always increasing with the number of threads M, barrier overheads and cavity size discrepancies clearly tend to become problematic when the number of threads is large. In the following sections, those two issues are addressed by using a two-level approach.

4. A Two-Level Multithreaded Delaunay Kernel

A two-level strategy is a partial solution to both issues of cavity size discrepancy and barrier overheads. At iteration k, we assume that each thread i inserts M_2 points at a time in a serial manner: M_2 Delaunay cavities are computed by thread i which leads to a Delaunay kernel that inserts $M \times M_2$ points at each iteration k. Figure 6 shows how M_2 affects the distribution of sizes of 3D Delaunay cavities for a set of on million points that are uniformly distributed on the unit cube. The distribution gets sharper around the average cavity size, which is definitively advantageous for scalability.

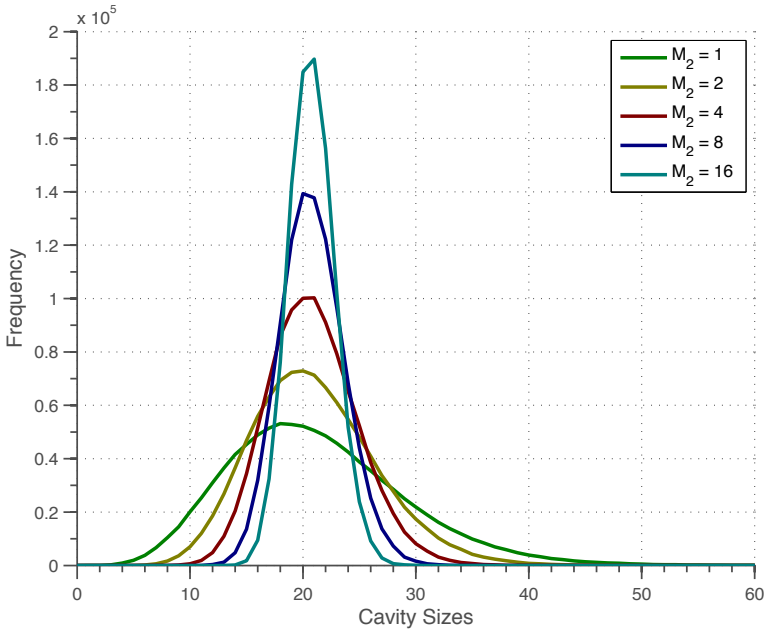

Fig. 6. Average size of 3D Delaunay cavities for $M_2 = 1$ to $M_2 = 16$.

The *two-level multithreaded Delaunay kernel* can be written in the following abstract manner:

$$\mathrm{DT}_{k+1} = \mathrm{DT}_k + \sum_{i=0}^{M-1} \sum_{j=0}^{M_2-1} \left[-\mathcal{C}(\mathrm{DT}_k, p_{k+i\frac{n}{MM_2}+j\frac{n}{M_2}}) + \mathcal{B}(\mathrm{DT}_k, p_{k+i\frac{n}{MM_2}+j\frac{n}{M_2}}) \right]. \tag{3}$$

This procedure generates exactly the same insertion sequence as the multithreaded Delaunay kernel with $M \times M_2$ threads.

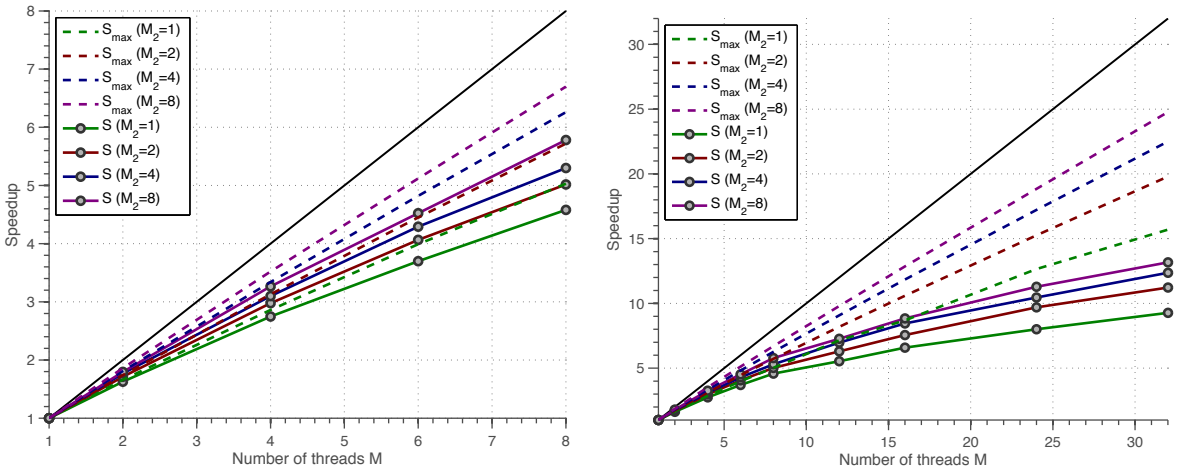

Fig. 7. Strong scaling of the two-level multithreaded 3D Delaunay kernel. Left figure is a zoom ($1 \leq M \leq 8$) of the right Figure ($1 \leq M \leq 32$).

Listing 3 presents a simplified version of the code that has been used to do the computations.

Figure 7 reports strong scaling results for the two-scale multithreaded Delaunay kernel for $M_2 = 1, 2, 4, 8$. Maximal theoretical speedups $S_{\max}$ are reported as well as true speedups S. The computer that has been used in is the same

4 socket node with a 8-cores processor on each socket for a total of 32 cores. The 8 first threads were executed within the scope of one single processor, allowing to maintain the nearness of threads and their data. Figure 7 shows that speedups increase with M_2 as predicted, and that for $M \leq 8$ the actual speedup of our procedure is close to the maximal possible one. A speedup of $S = 5.8$ is attained for $M = 8$ and $M_2 = 8$, which is slightly better that the ones observed in [3] and which is close to $S_{\max} = 6.7$. For a higher number of cores, speedups are moving away from their maximal values even though we made sure that thread affinity was maintained by allocating vertices and tetrahedra on the local node [15].

Yet, the good news is that speedups always increase with M: a triangulation of one million points in 3D was performed in 0.89 seconds of wall clock time on 32 cores for a total speedup of $S = 13.2$. This is one order of magnitude faster than the fastest procedure available.

Figure 8 gives scaling results for different sizes for the point set: it starts from $n = 10^4$ and increases that number to $n = 1.5\ 10^8$ points, i.e., to about 10^9 tetrahedra. With $M > 8$ speedups are even more important for very large sets of points. This is essentially due to the fact that data locality is better on 32 cores than using one single node for such a large amount of points.

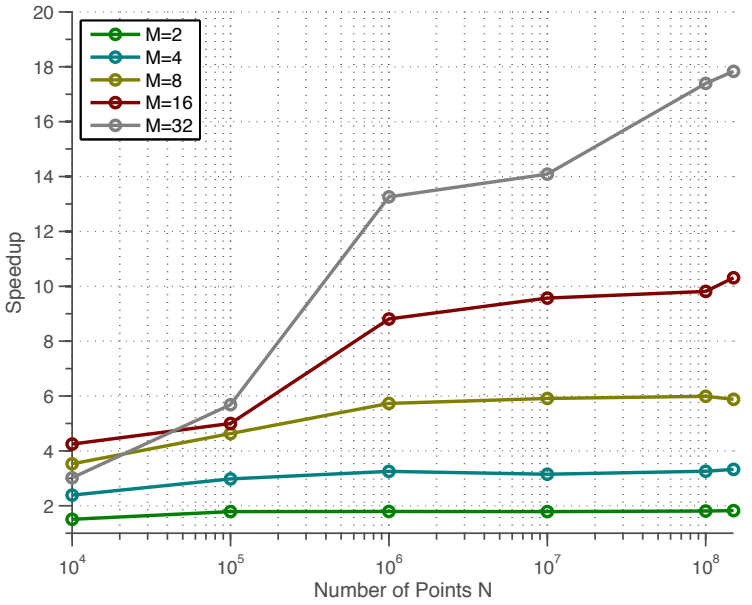

Fig. 8. Scaling of the two-level multithreaded Delaunay kernel with respect to N.

In our implementation, one tetrahedron requires 72 bytes of memory. Less than 100 GigaBytes of memory were necessary for 10^9 tetrahedra and it took 142.8 seconds of wall clock time for generating such a mesh on $M = 32$ cores with $M_2 = 8$. This corresponds to a rate of about 7 million of tetrahedra per second.

5. Conclusions

This paper presents some preliminary results on shared-memory parallel mesh generation. The central piece of the mesh generator, its Delaunay kernel, has been successfully parallelized using very simple OpenMP constructs. The parallel 3D delaunay procedure is available at `www.gmsh.info`. It has about 500 lines of code and only uses 3 OpenMP constructs.

This approach is presently extended to Graphical Processor Units. GPUs are the ultimate multicore platforms, with thousand of computing cores that provide massive throughputs. The two-level approach that is presented here actually fits very well with the two-level parallelism that is used in GPUs (blocks/threads).

The next step towards parallel mesh generation is to use this approach for Delaunay refinement.

References

[1] N. Amenta, S. Choi, and G. Rote. Incremental constructions con brio. In *Proceedings of the nineteenth annual symposium on Computational geometry*, pages 211–219. ACM, 2003.

[2] C. B. Barber and H. Huhdanpaa. Qhull, softwarepackage, 1995.

[3] V. H. Batista, D. L. Millman, S. Pion, and J. Singler. Parallel geometric algorithms for multi-core computers. In *Proceedings of the twenty-fifth annual symposium on Computational geometry*, pages 217–226. ACM, 2009.

[4] J. M. Bull, F. Reid, and N. McDonnell. A microbenchmark suite for openmp tasks. In *OpenMP in a Heterogeneous World*, pages 271–274. Springer, 2012.

[5] N. Chrisochoides and D. Nave. Parallel delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 58(2):161–176, 2003.

[6] N. P. Chrisochoides et al. A survey of parallel mesh generation methods. *Brown University, Providence RI-2005*, 2005.

[7] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[8] H. L. De Cougny, M. S. Shephard, and C. Ozturan. Parallel three-dimensional mesh generation. *Computing Systems in Engineering*, 5(4):311–323, 1994.

[9] H. Edelsbrunner. *Geometry and topology for mesh generation*. Cambridge University Press, 2001.

[10] P. J. Frey and P. L. George. *Mesh generation: application to finite elements*. ISTE London, 2008.

[11] P. L. George, F. Hecht, and E. Saltel. Automatic mesh generator with specified boundary. *Computer methods in applied mechanics and engineering*, 92(3):269–288, 1991.

[12] C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[13] S. Hornus and J.-D. Boissonnat. An efficient implementation of delaunay triangulations in medium dimensions. 2008.

[14] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of delaunay triangulations. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 1049–1056. ACM, 2006.

[15] A. Kleen. A numa api for linux. *Novel Inc*, 2005.

[16] R. Löhner, J. Camberos, and M. Merriam. Parallel unstructured grid generation. *Computer Methods in Applied Mechanics and Engineering*, 95(3):343–357, 1992.

[17] H. Si. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)*, 41(2):11, 2015.