Dynamic Parallel 3D Delaunay Triangulation

Panagiotis Foteinos[1,2] and Nikos Chrisochoides[2]

[1] Department of Computer Science, College of William and Mary
pfot@cs.wm.edu
[2] Department of Computer Science, Old Dominion University
nikos@cs.odu.edu

Summary. Delaunay meshing is a popular technique for mesh generation. Usually, the mesh has to be refined so that certain fidelity and quality criteria are met. Delaunay refinement is achieved by dynamically inserting and removing points in/from a Delaunay triangulation. In this paper, we present a robust parallel algorithm for computing Delaunay triangulations in three dimensions. Our triangulator offers fully dynamic parallel insertions and removals of points and is thus suitable for mesh refinement. As far as we know, ours is the first method that parallelizes point removals, an operation that significantly slows refinement down. Our shared memory implementation makes use of a custom memory manager and light-weight locks which greatly reduce the communication and synchronization cost. We also employ a contention policy which is able to accelerate the execution times even in the presence of high number of rollbacks. Evaluation on synthetic and real data shows the effectiveness of our method on widely used multi-core SMPs.

Key words: dynamic Delaunay triangulation, parallel, mesh generation

1 Introduction

1.1 Motivation

Mesh generation is a fundamental step for finite element analysis or visualization. A popular meshing technique is the *Delaunay mesh generation*, since it is able to mesh domains bounded by polyhedral surfaces [10, 12, 26, 35], curved surfaces [19, 32], or non-manifold surfaces consisted of a single [8, 34] or even multiple materials [18, 33].

The *quality* and *fidelity* of the mesh elements affect the speed and accuracy of the subsequent finite element analysis. Fidelity measures how well the mesh boundary describes the surface of the object to be modelled and quality regards the shape of the elements. Poor fidelity or poor quality meshes undermine the stability of the numerical solvers.

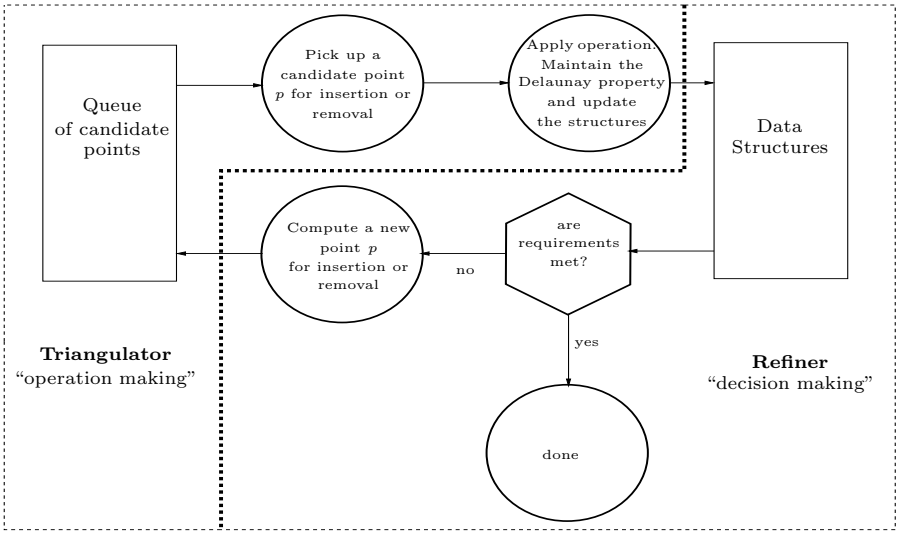

Fig. 1. The mesh refinement procedure illustrating the relationship between the refiner and the triangulator. The refiner computes the points that will potentially improve the mesh, while the triangulator is responsible for the actual operations, i.e, it maintains the Delaunay property and the data structures.

Therefore, after having computed an initial discretization of the input domain, meshers ought to refine it, until the specified quality and fidelity criteria are met. In the literature, refinement is achieved by incrementally inserting in or deleting points from an underlying *Delaunay triangulation* [8, 18, 19, 33, 34]. The triangulator, the backbone of any *Delaunay refiner* (see Figure 1), is responsible for updating the interconnectivity of mesh elements after the insertion or removal of points while maintaining the so called *Delaunay property*. The triangulator has to be able to support *dynamic* (i.e., *on line*) operations, simply because the sequence of the point insertions/removals (that improve the mesh) is not known a priori. In contrast, existing state of the art parallel Delaunay triangulators [6, 7] assume that the points are known before the parallel algorithm actually starts.

Figure 1 illustrates how the refiner and the triangulator cooperate with each other in a mesh refinement algorithm. Although this is the design we adopted for our own sequential mesh refinement algorithm [19], the high-level concept remains valid for all the other Delaunay mesh refinement algorithms in the literature. The key concept is that the refiner computes the point that will have to be inserted or removed, i.e., decides the sequence of the points that will improve the quality and the fidelity of the mesh, while the actual insertion or removal is performed by the triangulator. As stated above, that sequence of points is dynamically revealed by the refiner; decisions made in the past determine the ones that will be made in the future.

In [19], we developed a High Quality mesh Refinement algorithm (abbreviated to HQR hereafter) for medical images. Image-to-Mesh (I2M) conversion software [18, 19, 22, 33, 38] is essential for visualization or medical simulations. HQR at any time maintains the triangulation of points that have already been inserted inside, outside, or precisely on the boundary of the manifold (i.e., curved object) to be meshed. Therein, we prove that after the end of the refinement, a subset of the final triangulation forms a good topological and geometric approximation of the object and is of good quality. That subset (which is uniquely defined and extracted) constitutes the actual mesh of the object (this is a common technique used in the literature for meshing volumes bounded by surfaces [8, 18, 32–34]). Note that mesh generators for images need to also recover the surface of the object; the surface of the object is unknown and not given as a polyhedral domain. HQR meshes the surface and the volume of the object at the same time.

In this paper, towards the parallelization of our sequential I2M algorithm [19], we present a dynamic *Parallel Delaunay Triangulator* (PDT). Unlike all the other parallel implementations in the literature, we support Delaunay removal of points, an operation that has been shown to significantly improve the quality [26] and fidelity of the final meshes [19]. It will become obvious in the next sections that removing points is not only a slow operation (slower than insertions) but it is also a challenging task.

PDT follows the *master-workers* approach, because in this way we facilitate the integration of the triangulator with any refiner for manifold surfaces. In fact, the triangulator sees the refiner as a black box, which offers great flexibility since no assumptions for the location of the points to be inserted/removed are made. The integration of our parallel triangulator to a parallel refiner is the next step and is outside the scope of this paper.

1.2 Related work

In the literature, there has been extensive work on developing parallel Delaunay methods in two and three dimensions. Their main limitation is the fact that they do not support removal of points, and therefore, they are not suitable for mesh refinement codes where deletions are needed (see the work by Klinger and Shewchuk [26] and Foteinos *et al.* [19] for few such refinement schemes).

Blelloch *et al.* [7], Hardwick [23], and Amato *et al.* [3] compute the Delaunay triangulation of a given point set by solving the corresponding convex hull problem using a parallel divide and conquer scheme. Teng *et al.* [36] construct the triangulation by expanding faces on every vertex in parallel. Cignoni *et al.* [14] present a divide and conquer and a construction parallel algorithm and compare their performance on uniformly distributed points. Blandford *et al.* [6] present a 3D incremental triangulator that first associates the uninserted vertices to the tetrahedra that contains them. Kohout *et al.* [27] give two incremental randomized insertion schemes (i.e., the points to be inserted

have to be reordered) for computing the Delaunay triangulation in 2D. It is worth noting that all the techniques described so far are not dynamic, since the point set has to be known before the algorithm starts; therefore, they are not suitable for mesh refinement because the list of uninserted vertices constantly changes.

A dynamic parallel refinement algorithm for distributed memory platforms is given by Okusanya and Peraire [31]. The observed speed up, however, is very low: the execution time of 8 processes on uniformly distributed data is higher than that of 2. Dynamic parallel mesh refinement methods are also proposed by Chernikov and Chrisochoides [11]. Therein, the synchronization between processors is greatly reduced by choosing to insert points in parallel only if the insertions do not cause conflicts. Kadow [25] extends the work of Blelloch *et al.* [7] in order to support dynamic insertions in 2D. Galtier and George [20] propose a domain decomposition scheme. The domain is subdivided by partitioning the polyhedral surface mesh. That partitioning is performed by computing a suitable separator of the domain boundary. Linardakis and Chrisochoides [28] present a 2D decoupling parallel method. The communication between workers is eliminated by inserting extra points on the medial axis of the domain and meshing each resulting subdomain in parallel. Note that the dynamic refinement algorithms mentioned so far target exclusively polyhedral domains, that is, the surface of the object is already represented as a set of polyhedral facets. As already explained in Section 1.1, our parallel triangulator prepares the ground for a parallel guaranteed quality and fidelity mesh generator for objects whose surface and volumes are meshed at the same time, an essentially different problem.

Nave and Chrisochoides [30] parallelize the Bowyer-Watson kernel. The new inserted points, however, are restricted to be the circumcenters of poorly shaped triangles. That implies that *locating* the new point is not necessary, since the first element in the cavity (i.e., the poor element) is known and it needs not to be found. In the literature, meshing curved objects necessitates the insertion of points that are not circumcenters of poor elements. See for example the work in [8, 18, 32–34]. Therein, points on the voronoi edges are also inserted to guarantee that the mesh boundary is a good approximation of the object. As another example, our sequential refinement algorithm [19] sometimes inserts points outside the circumball of poor elements. For these reasons, our parallel triangulator makes no assumptions about the location of the points to be inserted, and therefore it supports (in fact, it has to support) *parallel locating* as well. Another difference is that Nave and Chrisochoides [30] start their parallel Bowyer-Watson kernel after the sequential construction of an initial mesh. Therefore, there is enough parallelism (i.e., no contention) in the early stages of the refinement. We follow a different approach. Our parallel Bowyer-Watson kernel tries to exploit parallelism in the early stages of the triangulation via the help of a contention manager policy. Furthermore, our implementation supports parallel removals.

1.3 Our method

In this paper, we present a parallel dynamic Delaunay triangulation algorithm. Its main features include support for dynamic removal of points and ease of integration with any refinement schemes, especially refinement schemes that work directly on manifold surfaces. This makes our implementation suitable for mesh refinement schemes that (a) do not rely only on insertion of points for quality/fidelity improvement but also on Delaunay deletions [19, 26] and (b) decide to insert non-trivial points, i.e., points other than circumcenters or midpoints [18, 32–34]. Indeed, our parallel triangulator imposes no restrictions on the location of the points to be inserted or removed.

As stated above, we make use of a master-worker approach. More precisely, the triangulator, instead of applying the operation for each point in its global queue sequentially, launches the master thread. The master thread inspects the global queue and as long as there are points left, it moves them to the appropriate private queue of the workers. Each worker is responsible for only the points that are in its private queue. If a worker, during the insertion or removal of a point, encounters a locked vertex, then it aborts the operation (*rollback*) [7, 30] and tries to operate on another vertex in its private queue.

Rollbacks, however, may cause livelocks which result in system-wide starvation [24], especially when there is high contention. Kohout *et al.* [27] overlook this fact and that is the reason the resulting Delaunay triangulation is not always valid: they report that few elements are not Delaunay. Blandford *et al.* [6] deal with livelocks by bootstrapping: they insert the first $500,000$ vertices sequentially and therefore the chances for high contention are minimized. As we have already mentioned, however, we are not given any point a priori, and therefore bootstrapping cannot be applied. Generally, dynamic triangulators cannot rely on any pre-processing strategy. We solve the starvation issue by employing a contention management policy. It guarantees correctness even in cases of extreme contention (i.e., when the number workers is very large with respect to the size of the triangulation), without compromising speed. On the contrary, the contention manager always yields faster execution times than these achieved by our sequential algorithm.

The rest of the paper is organized as follows: Section 2 describes our sequential implementation and Section 3 elaborates on the key aspects of our parallel implementation. Section 4 evaluates our parallel code on synthetic and real data, and Section 5 concludes the paper.

2 Sequential Implementation

The Delaunay triangulation $\mathcal{D}(V)$ of a set of vertices $V \subset \mathcal{R}^3$ is a triangulation that satisfies the *Delaunay property*. More precisely, let $\mathcal{B}(t)$ denote the open circumscribing ball (a.k.a *circumball*) of tetrahedron t. Then, t belongs in $\mathcal{D}(V)$ if $\mathcal{B}(t)$ contains no vertex of V. The insertion of a new vertex $v \notin V$

Algorithm 1: The Bowyer-Watson insertion kernel.

Algorithm: `Insert`(V, $\mathcal{D}(V)$, v)

Input : V is the current set of vertices,
$\mathcal{D}(V)$ is the Delaunay triangulation of V,
v is the new vertex.

Output: The Delaunay triangulation of set $V \cup v$.

Compute $\mathcal{C}(v) = \{t \in \mathcal{D}(V) \mid v \in \mathcal{B}(t)\}$;
Delete all tetrahedra in $\mathcal{C}(v)$;
Compute $\partial\mathcal{C}(v)$, the set of triangles incident to a tetrahedron that belongs to $\mathcal{C}(v)$ and to a tetrahedron that does not belong to the cavity;
Connect v with all the vertices of $\partial\mathcal{C}(v)$;
$V = V \cup v$;

or the removal of an existing vertex $v \in V$ necessitates local transformations such that the Delaunay property is maintained and the triangulation is still valid (i.e., the tetrahedra form a partition of the convex hull of the vertices).

2.1 Sequential Insertion

Let v be the new vertex inserted into V. The triangulation is updated using the well known *Bowyer-Watson* kernel [9, 37]. See Algorithm 1 for an illustration. First, the cavity $\mathcal{C}(v)$ of v is computed. The cavity contains all the tetrahedra in $\mathcal{D}(V)$ whose circumball contains v. Clearly, the elements composing the cavity have to be deleted because they violate the Delaunay property. Let us denote with $\partial\mathcal{C}(v)$ the boundary of the the cavity. As noted in [9, 37], $\partial\mathcal{C}(v)$ is a convex polyhedron and therefore, every vertex on the cavity's boundary is visible from v. Hence, connecting the vertices of $\partial\mathcal{C}(v)$ to v constitutes a valid triangulation. It can also be shown that the new elements created in this way respect the Delaunay property [21].

Computing the cavity of v is trivial, as long as we know one tetrahedron in $\mathcal{D}(V)$ which actually contains v (or one tetrahedron that belongs to the cavity). Thus, we first have to traverse part of the triangulation to locate that tetrahedron before we proceed to cavity computation. For this reason, our algorithm implements the *visibility walk* as described in [16]: starting from an element, we perform orientation checks which will dictate the next element of the walk, until we find the tetrahedron the contains v. In order to launch the location from a good starting element, we implement the *jump and walk* technique as described in [29]. Specifically, a small subset of the vertices in the triangulation is sampled, and the starting element is an element incident to the sample which is closest to v. Although, the jump and walk technique is slower than more elaborate schemes [15], it is an ideal candidate because it achieves fairly good complexity and its parallelization introduces no global synchronization.

Algorithm 2: The removal kernel.

Algorithm: `Remove`$(V, \mathcal{D}(V), v)$

Input : V is the current set of vertices,
$\mathcal{D}(V)$ is the Delaunay triangulation of V,
v is the vertex to be removed.

Output: The Delaunay triangulation of set $V - v$.

Compute $\mathcal{H}(v) = \{t \in \mathcal{D}(V) \mid t \text{ is incident to } v\}$;
Delete all tetrahedra in $\mathcal{H}(v)$;
Compute $\partial\mathcal{H}(v)$, the set of triangles incident to a tetrahedron that belongs to $\mathcal{H}(v)$ and to a tetrahedron that does not belong to the hole;
Compute the small triangulation of the vertices of $\partial\mathcal{H}(v)$;
Merge the small triangulation with $\mathcal{D}(V)$;
$V = V - v$;

It is worth noting that the Bowyer-Watson kernel never creates flat tetrahedra (which must not exist in legal triangulations). And this is due to the fact that v cannot be coplanar with any of the facets of $\partial\mathcal{C}(v)$.

2.2 Sequential Removal

Removing a vertex $v \in V$ from $\mathcal{D}(V)$ involves re-triangulating the hole $\mathcal{H}(v)$ created by the tetrahedra incident to v. See Algorithm 2. First, we compute the Delaunay triangulation of the vertices on the boundary $\partial\mathcal{H}(v)$ of the hole. We shall refer to that triangulation as "small" triangulation to distinguish it from $\mathcal{D}(V)$ (the "big" triangulation). Then, we sew the small triangulation back to $\mathcal{D}(V)$.

Extra care has to be taken, however, to ensure that the small and big triangulations actually match. The problem is that $\partial\mathcal{H}(v)$ may not appear as a set of facets in the small triangulation due to degenerate cases. As explained in [17], if there are more than 3 cospherical and coplanar vertices in $\partial\mathcal{H}(v)$, then their triangulation in the plane is not unique, and therefore, parts of $\partial\mathcal{H}(v)$ might fail to appear in the small triangulation. Hence, we need a way to resolve ties, so that the small triangulation always matches the boundary of the hole. We do so by keeping track of the order in which the vertices were inserted in $\mathcal{D}(V)$. We use that information when triangulate the vertices of $\partial\mathcal{H}(v)$: they are inserted into the small triangulation according to the order they were inserted earlier in $\mathcal{D}(V)$. In this way, we can guarantee that sewing always gives valid triangulations. The proof of correctness is omitted in this paper.

Removals are more expensive than insertions. During the evaluation of our sequential triangulator, we observed that the removal of 3000 uniformly distributed points is about 6 to 7 times slower than their insertion. See Table 1 for a comparison. (Removals are 6 to 7 times slower than insertions in the

CGAL triangulator [2] as well, the fastest dynamic triangulator we are aware of.) This result is counter-intuitive because insertions involve visibility walks, while removals do not. Indeed, each inserted point v stores a pointer to an incident element. Therefore, if v is removed, the hole can be found without triangulation traversals. Although removals do not require triangulations walks, the cost associated with memory management increases. And the reason is that the small triangulation does not contain and maintain only the elements needed to fill the hole. If the hole $\mathcal{H}(v)$ is not convex, then all the elements of the small triangulation that are outside $\mathcal{H}(v)$ but inside the convex hull of the vertices of $\partial\mathcal{H}(v)$ will never be a part of the big triangulation. Therefore, more bookeeping is introduced for maintaining and sewing the small triangulation.

3 Parallel Implementation

Our parallel implementation makes use of the C++ *Boost threads* [1]. When a thread wants to lock an element, it does so by locking its 4 vertices. If the operation is a read-only operation, then the thread asks for a shared ownership. For example, locating the element that contains a vertex does not modify the triangulation. Therefore, multiple location operations might overlap without any blocking.

3.1 Master-Workers Scheme

Each worker is responsible for a specific region. It inserts or removes a vertex as long as it lies inside its region. A worker either inserts or removes vertices. In the former case, it is referred to as an *inserter*, and in the latter as a *remover*. The master thread keeps scanning the global queue and moves vertex v (if any) to a worker's private queue (implemented as a thread safe single linked list) only if v lies inside the worker's region. In order to assign each worker a region, we assume that we are given the positions of the extreme points that will come in the future. This is a reasonable assumption, since in most cases the domain of interest is known. Then, the domain is logically divided into 3D structured blocks, each of which is assigned to a specific worker. Note that there is no global synchronization for the private queues. The master thread places new vertices at the head of the queues and the workers draw them from the tail of their queues, locking each time 2 exactly queue nodes and not all the queue.

3.2 Parallel Operations

When a cavity $\mathcal{C}(v)$ is explored by a worker W_1, all of the cavity's elements are locked exclusively. If another worker W_2 visits any of $\mathcal{C}(v)$'s elements, then W_2 aborts the operation (*rollback*) and moves on to the next point in its private queue [7, 30].

When a worker attempts to remove a vertex v, it acquires an exclusive lock on the elements of the hole $\mathcal{H}(v)$. Similarly to insertions, if another worker happens to be on an element of $\mathcal{H}(v)$, it aborts and tries to operate on another point in its private queue. Also, note that a remover might try to remove a vertex which belongs to an inserter's sample list (recall that the jump and walk location technique implemented by the inserters requires some processing of a small subset of already inserted vertices). Therefore, we require that the inserters exclusively lock their samples. If an inserter finds a sample exclusively locked (by either an another inserter or a remover), then it tries to find another one to start the locating from.

In order to decrease the synchronization (as a result of locking) and communication cost (as a result of reading and writing the shared memory) among the threads, we developed our own custom memory manager and light-weight locks. According to the findings presented by Antonopoulos *et al.* [4], efficient memory utilization and locking greatly reduces the overhead of maintaining multiple threads.

- *Memory manager*: Allocating and deallocating cells and vertices in multi-threaded implementation is costly, because now the kernel calls a thread safe implementation of the *new/delete* operator. This overhead can be reduced if each thread has its own (private) memory pool from which it asks blocks. If a block is to be deleted, then it throws it back to its memory pool. When the pool does not contain any blocks, only then the thread expands its pool by calling the kernel's *new* operator. In order to further expedite things and exploit localization, when the pool needs to be expanded, a whole chunk of blocks is allocated. We set the chunk size equal to 12 pages for cells and 3 pages for vertices (each page is 4KB). Higher chunk sizes yielded similar results.
- *Light-weight locks*: Locking mechanisms using POSIX mutexes can waste hundreds of cycles. On the contrary, the built-in atomic operations implemented by the C++ GNU compiler just add few stall cycles. Therefore, we decided to implement our own shared and exclusive try-locks using the atomic *fetch_and_add* operation. More precisely, each vertex has a specific flag which we atomically increment or decrement. That flag denotes the number of readers (associated with the vertex) if the value is non-negative. A non-negative flag implements a shared lock. If the flag is negative, then it is exclusively owned by one and only one worker. By replacing the pthread mutexes (that we used in the early stages of the development of our parallel code) with our light-weight locks, the removals sped up significantly. For example, the single-threaded removal of $10,000$ random points was faster by 24%. Interestingly, the insertions were not substantially affected. We think that this happens because the number of cells incident to a point (to be removed) tends to be higher than the number of cells of a point's cavity (to be inserted); hence, removing involves more locking and it is more sensitive to the locking mechanism employed. In uniform data, for

example, we counted that on the average the insertion of a point requires 21 locks, while its removal 30 locks.

3.3 Contention Management

When the number of workers is high and the number of inserted vertices is small, increased contention is introduced which hampers the performance of the triangulator. What is even worse, the workers will most likely suffer from *livelocks*, a common pitfall of *non-blocking parallel* algorithms [24]. Livelocks are caused by continuous rollbacks: workers try to lock overlapping sets of vertices for an undefined period of time. Kohout *et al.* [27] overlook that problem and, for that reason, invalid Delaunay elements are reported. Blandford *et al.* [6] solve the problem by bootstrapping: they insert the first $500,000$ vertices sequentially. We believe that even in the beginning (i.e., when there are not many vertices inserted) there is parallelism that can be exploited. Therefore, instead of bootstrapping, we decided to follow a more dynamic (yet simple to implement and with little overhead; see Section 4.1) contention policy. This is also one of the differences with the previous work of Nave and Chrisochoides [30]: therein, an initial mesh is first constructed sequentially and then the work is distributed among the workers.

We experimentally found that livelocks are always present in our algorithm when the number of vertices already inserted into the triangulation is less than $750 \times N$, where N is the number of parallel workers. And that fact did not only cause specific threads to starve but also it prevented system-wide throughput: none of the threads did useful work for a long (and thus undefined) time. To solve the problem, each worker W_i keeps track of its progress by calculating its *progress ratio* $u_i = \frac{C_i}{A_i}$, where C_i is the number of completed operations (i.e. amount of useful work) and A_i is the number of attempted operations. If A_i is large compared to C_i, then that means that the worker spends most of its time rolling back. When u_i drops below a specified threshold u^-, then W_i goes to sleep, releasing all its resources. A low ratio implies that W_i finds it difficult to cooperate with other threads, and therefore, it becomes inactive in order to help the other workers do useful work. Conversely, if u_i exceeds a specified threshold u^+ ($\geq u^-$), then it signals a sleeping worker, say W_j (if any). A high ratio implies that W_i does not conflict with other threads often, which indicates that a inactive worker might be able to do some useful work now. The awaken worker W_j can now inspect its private queue to find a point to work on. (Note that when a worker W_j awakes, its counters C_j and A_i are reset to 0, clearing in this way its progress history. We find no good reason why the contention manager should not be memoryless.) In cases of high contention, only one worker will be active, simulating a sequential algorithm.

Although the contention manager's primary goal is to insure correctness (that is, absence of livelocks), we observed that it speeds up the triangulator in its very early stages, i.e., when the triangulation does not contain

Table 1. Contention management on extreme cases

#Threads	Insertions		Removals	
	1	12	1	12
Time (secs)	0.06	0.04	0.39	0.12
Speedup	1	1.5	1	3.3
Max $\frac{\#\text{Rollbacks}}{\#\text{completed operations}}$ (%)	0	80	0	3500

many vertices. Table 1 shows the speed up achieved by 12 inserters and 12 removers. For that experiment, only 3000 points are about to be inserted into and removed from the triangulation, and therefore, we push our algorithm to extremes. The points are normally distributed. (Note that without the contention manager, the presence of livelocks prevented the insertion of any vertex for more than 1 hour.) The thresholds u^- and u^+ were set to 0.7 and 0.9 respectively. Different configurations of the thresholds yielded the same behavior. We observed similar results on different distributions (e.g., uniform distribution, line distribution, points on a box, and grid points) too. We can see from the last row that the contention is very intense. For example, a value of 3500% for $\frac{\#\text{Rollbacks}}{\#\text{completed operations}}$ means that a remover had to roll back 35 times on the average for every single point it removed. Despite that fact, the parallel implementation not only did not encounter any livelocks but also, it yielded faster executions.

4 Experimental Evaluation

In this section, we evaluate the performance of our Parallel Delaunay Triangulator (PDT) on both synthetic and real data. Throughout the evaluation, we used an Intel and an AMD machine. The Intel machine is equipped with a 12-core Xeon X5690 CPU at 3.47GHz and 96GB of memory, and the AMD machine with a 48-core Opteron 6174 CPU at 2.2GHz and 96GB of memory. Both the sequential and the parallel code is written in C++.

4.1 Synthetic Data

We first evaluated our parallel implementation on synthetic data. More precisely, we dynamically feed the global queue with points to be inserted or removed according to three distributions: uniform, normal, and line as described in [7]. For all distributions, we simulate the dynamic insertion of 12M points and the dynamic removal of the first 1.2M points (which account for 10% of the inserted points).

Table 2, Table 3, and Table 4 illustrate the results. The experiments were run on the Intel machine. The reported Speedup1 is the speedup with respect to our single-threaded parallel implementation. The $\frac{\#\text{Rollbacks}}{\#\text{compl. ops}}$ row shows

Table 2. Uniform

	Insertions					Removals				
#Threads	1	2	4	8	12	1	2	4	8	12
Time (secs)	743	355	186	89	61	167	85	46	24	17
Speedup1	1.00	2.09	3.99	8.35	12.18	1.00	1.96	3.63	6.96	9.82
Max $\frac{\#\text{Rollbacks}}{\#\text{compl. ops}}$ (%)	0.000	0.001	0.008	0.013	0.032	0.000	0.000	0.000	0.000	0.003
CGAL time (secs)	402	-	-	-	-	131	-	-	-	-
Speedup2	0.54	1.13	2.16	4.52	6.59	0.78	1.54	2.85	5.46	7.71

Table 3. Normal

	Insertions					Removals				
#Threads	1	2	4	8	12	1	2	4	8	12
Time (secs)	739	356	185	90	87	166	85	46	24	24
Speedup1	1.00	2.08	3.99	8.21	8.49	1.00	1.95	3.61	6.92	6.92
Max $\frac{\#\text{Rollbacks}}{\#\text{compl. ops}}$ (%)	0.000	0.003	0.006	0.018	0.022	0.000	0.000	0.000	0.000	0.001
CGAL time (secs)	400	-	-	-	-	131	-	-	-	-
Speedup2	0.54	1.12	2.16	4.44	4.60	0.79	1.54	2.85	5.46	5.46

Table 4. Line

	Insertions					Removals				
#Threads	1	2	4	8	12	1	2	4	8	12
Time (secs)	1,182	507	240	110	72	169	86	47	25	17
Speedup1	1.00	2.33	4.93	10.75	16.42	1.00	1.97	3.60	6.76	9.94
Max $\frac{\#\text{Rollbacks}}{\#\text{compl. ops}}$ (%)	0.000	0.001	0.035	0.097	0.252	0.000	0.000	0.006	0.030	2.641
CGAL time (secs)	612	-	-	-	-	133	-	-	-	-
Speedup2	0.52	1.21	2.55	5.56	8.50	0.79	1.55	2.83	5.32	7.82

the number of rollbacks with respect to the number of points that were inserted/removed (see Section 3.3). High values imply that the worker spent most of its time rolling back instead of doing useful work (i.e., instead of actually completing an operation). In fact, that row reports the higher value among the workers.

First of all, notice that the penalty introduced by the contention manager is negligible. In most cases, the maximum percentage of rollbacks with respect to the total number of inserted/removed points is less than 0.1%. Also, the total number of seconds that the workers sleep (not shown in the tables) were less than 3 seconds for all the experiments. Therefore, it is safe to look at the $\frac{\#\text{Rollbacks}}{\#\text{compl. ops}}$ to determine whether the synchronization cost is high or not.

For the uniform (Table 2) and line distribution (Table 4), we observe excellent speedups. With 12 workers, the uniform insertions are more than 12 times faster than the single-threaded execution; also, to our surprise, the line insertions are more than 16 times faster. The reason for such superlinear improvements is memory locality and memory reuse achieved by our custom memory manager. For the normal distribution (Table 3), we observe superlinear speedups up to 8 threads. Increasing the number of threads do not result in considerably less execution, although the number of rollbacks is still very

small. This behavior is attributable to load imbalance. In fact, only 9 out of the 12 launched workers have work to do (the same applies for the parallel removals too). Load balancing ought to be considered on its own merit (see for example the work by Barker and Chrisochoides [5]), and it is left as future work.

For the parallel removals, notice that the speedups are smaller than those achieved by the inserters. For example, the improvement with 12 uniform removers is 9.82, while the improvement with 12 inserters is superlinear. We believe that this happens because removals are more memory intensive than insertions. During a removal the number of cells to be updated is higher than in the case of an insertion (more on this shortly). Therefore, memory management overhead per removal is higher than that per insertion. When the number of removers increases, that communication cost hampers the speedup of the inserters.

The increased number of cells to be updated during a removal is not only due to the fact that the cells in a cavity of a point to be inserted are fewer than the cells incident to a point to be removed, as reported in Section 3.2. It is also due to the fact that the hole has to be retriangulated which introduces extra memory overhead. For example, during the single threaded experiment of uniform data, we counted that on the average 211 cells are touched per removal, but only 19 cells are touched per insertion.

For comparison, the last row (Speedup2) of the tables above report the speedup of our parallel algorithm over the *Computational Geometry Algorithms Library* (CGAL) [2]. To our knowledge, CGAL triangulator is the fastest sequential dynamic algorithm and it is also open-source software. Note that CGAL triangulator has been highly optimized and tested over the last years and it is the state of the art implementation to date.

Observe that although the time of our single-threaded removal of points is comparable to that of CGAL, our single-threaded insertion is a little bit less than 2 times slower. The reason for such a difference is not only the extra cost associated with locking the elements in the cavity, but also the fact that the jump and walk location technique we employed is asymptotically slower than the optimal *Delaunay hierarchy* used by CGAL. (The worst-case cost of the jump and walk technique is $O(n^{\frac{2}{3}})$, while the cost of the hierarchy is $O(\log n)$.) As explained in Section 2, we used the jump and walk algorithm because it can be parallelized without extra synchronization cost, has a good expected complexity, and requires little memory. Despite the slower single-threaded execution times of our algorithm, it soon outperforms CGAL. The uniform insertions and removals by 12 inserters and 12 removers are 6.59 and 7.71 times faster, respectively. For the line insertions and removals, the speed up over CGAL is 8.5 and 7.82. Lastly, the normal insertions and removals by 8 inserters and 8 removers are 4.6 and 5.46 times faster, respectively.

The last experiment on synthetic data is performed on the AMD machine. In this experiment, we wanted to see the speedup of our algorithm when a larger number of cores is used. We insert 1M uniformly distributed points per

Table 5. Results for the AMD machine

	Insertions							Removals						
#Threads	1	2	4	8	16	32	48	1	2	4	8	16	32	48
Time (secs)	69	86	107	105	128	181	217	24	27	32	33	34	34	35
Max $\frac{\#\text{Rollbacks}}{\#\text{compl. ops}}$ (%)	0.00	0.00	0.01	0.03	0.09	0.20	0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CGAL time(secs)	40	87	194	418	898	1873	2869	16	37	66	132	264	526	800
Speedup	0.58	1.01	1.81	3.98	7.02	10.35	13.22	0.67	1.37	2.06	4.00	7.76	15.47	22.86

inserter and remove 0.1M points per remover. Table 5 depicts the results. The last row shows the speedup of our implementation over CGAL on the same input. Notice that the number of rollbacks is negligible, and therefore synchronization cost is not a problem. However, we observe that after 8 workers both inserters and removers stop scaling well. For example, with 48 inserters the speedup over CGAL is 13.22, while if the scaling was perfect it should be $48 \times 0.58 = 27.84$. Recall that we increase the problem size per worker and therefore one would expect a good scaling, since the number of rollbacks is negligible. Note, however, that the application is memory intensive and that when we increase the problem size, the memory management overhead also increases (the memory reads/writes with 48 workers is at least 48 times more than the memory reads/writes with 1 worker). For these reasons, we believe that after a certain number of workers, the communication cost dominates and the scaling is suboptimal.

4.2 Real Data

We also simulated the sequence of points produced by our high quality sequential I2M algorithm HQR (see Section 1 for some information on HQR) [19]. The quality and fidelity guarantees that HQR proves can be found in [19]. Note that in [19], we developed only our own refiner; the (sequential) triangulator used there was built on top of a third party library (CGAL [2]). We ran HQR on a segmented medical image and we traced the sequence of points that were inserted and removed. Then, we fed that data to our parallel triangulator to simulate a real case. Figure 2 shows the data produced by HQR.

Table 6 depicts the timings. During the simulation, $219,031$ points were inserted and $63,356$ were removed in parallel. Although the insertions scale well, the removals hit a wall after 4 workers. This is expected because there is a little concurrency to be exploit, i.e., the number of points to be removed per remover is low. (Note that load imbalance is not a problem in this simulation, since we verified that each worker attempted to make roughly the same number of operations, that is, the number of points to be removed is approximately the same for all removers.) This also agrees with the huge rollback percentages: for example, when 12 removers are launched, we noticed that a thread had to try $1,069,400$ times just to remove $5,319$ points. Clearly, the computation for such a little work of removals is not enough to compensate for the synchronization cost.

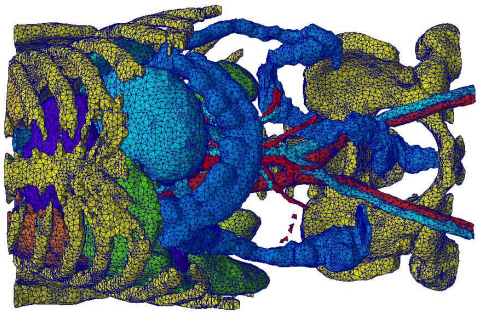

Fig. 2. The result after the termination of our mesh refinement algorithm. This sequence of points that were inserted/removed was fed into our parallel triangulator.

Table 6. Evaluation on real data. Notice the high rollback percentages associated with the removals.

	Insertions					Removals				
#Threads	1	2	4	8	12	1	2	4	8	12
Time (secs)	7.3	4.2	2.5	1.4	0.9	32.0	16.7	10.8	7.1	11.6
Speedup1	1.00	1.74	2.92	5.21	8.11	1.00	1.92	2.96	4.51	2.76
Max $\frac{\#\text{Rollbacks}}{\#\text{compl. ops}}$ (%)	0.0	0.2	1.0	4.7	6.1	0.0	0.1	16.8	42.9	19837.0
CGAL time (secs)	3.9	-	-	-	-	16	-	-	-	-
Speedup2	0.53	0.93	1.56	2.79	4.33	0.50	0.96	1.48	2.25	1.38

5 Conclusions and Future Work

In this paper, we presented a dynamic parallel Delaunay triangulator. Its main feature is its ability to support parallel removal of points, an operation that is much slower than the insertion as we have already explained (Section 2.2). For synthetic data, the execution time of our parallel insertions on the 12-core Intel machine is $4.6 - 8.5$ times faster than the fastest sequential triangulator (CGAL [2]) we are aware of. The corresponding speedup over CGAL for our parallel removals is 5.46 to 7.82. The overall speedup (taking into account both insertions and removals) over CGAL ranges from 4.78 to 8.37 when 12 workers are launched. Removals do not scale as well as insertions, but the reason is not the synchronization cost. In fact, our light-weight lock implementation (see Section 3.2) greatly reduced the time spent for locking by 24%. Indeed, as Table 2, Table 3, and Table 4 show, the time spent on rollbacks is negligible when compared with the time of useful work. We noticed, however, that removals are 11 times more memory intensive than insertions (see Section 4.1). Therefore, removals exhibit higher communication cost which limits scalability. We believe that communication cost is also the reason that our algorithm stop scaling well after 8 workers on the 48-core AMD machine (see Section 4.1).

A case of increased synchronization cost is shown in Table 6. The high $\frac{\#\text{Rollbacks}}{\#\text{compl. ops}}$ numbers indicate exactly that. For example, the value 19837%

implies that a thread had to roll back 198.37 times on average for every point it tried to remove. As explained in Section 4.2, the reason for so much contention is because of the little concurrency that can be exploit. This experiment shows that our contention manager (see Section 3.3) is able to remove livelocks under extreme circumstances.

Our next goal is to combine our parallel triangulator with a parallel refiner for medical images and exploit parallelism in two levels: in the level of the triangulation (operation making) and in the level of the refiner (decision making, see Figure 1). To ease the integration, the triangulator employs a master-workers scheme and sees the refiner (i.e., the master or the masters if the refiner is parallel) as a black box. Note that if the parallel refiners are synchronized, they will never attempt to improve an element that was previously removed.

The experimental evaluation presented in this paper focuses on shared-memory multi-core machines. We are also planning to parallelize our algorithm for larger distributed-memory machines according to the work of Chrisochoides *et al.* [13]. Therein, a hybrid mesh generation framework is developed that takes advantage of both the smaller shared memory layer and the larger distributed memory layer.

Acknowledgements

The authors would like to thank Andrey Chernikov and Andriy Kot for the fruitful discussions and constructive comments and the anonymous reviewers who helped us improve the paper. This work is supported in part by NSF grants: CCF-1139864, CCF-1136538, and CSI-1136536 and by the John Simon Guggenheim Foundation and the Richard T. Cheng Endowment.

References

1. Boost C++ libraries. `http://www.boost.org/`.
2. CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org`.
3. Nancy M. Amato, Michael T. Goodrich, and Edgar A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *IEEE Symposium on Foundations of Computer Science*, pages 683 –694, November 1994.
4. Christos Antonopoulos, Filip Blagojevic, Andrey Chernikov, Nikos Chrisochoides, and Dimitris Nikolopoulos. Algorithm, software, and hardware optimizations for delaunay mesh generation on simultaneous multithreaded architectures. *Journal on Parallel and Distributed Computing*, 69(7):601–612, 2009.
5. Kevin Barker and Nikos Chrisochoides. Practical performance model for optimizing dynamic load balancing of adaptive applications. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2005.
6. Daniel K. Blandford, Guy E. Blelloch, and Clemens Kadow. Engineering a compact parallel delaunay algorithm in 3d. In *Proceedings of the* 22nd *Symposium on Computational Geometry*, SCG '06, pages 292–300, New York, NY, USA, 2006. ACM.

7. Guy E. Blelloch, Gary L. Miller, Jonathan C. Hardwick, and Dafna Talmor. Design and implementation of a practical parallel delaunay algorithm. *Algorithmica*, 24(3):243–269, 1999.
8. Jean-Daniel Boissonnat and Steve Oudot. Provably good sampling and meshing of surfaces. *Graphical Models*, 67(5):405–451, 2005.
9. Adrian Bowyer. Computing Dirichlet tesselations. *Computer Journal*, 24:162–166, 1981.
10. Andrey Chernikov and Nikos Chrisochoides. Three-Dimensional Semi-Generalized Point Placement Method for Delaunay Mesh Refinement. In *Proceedings of the* 16$^{\text{th}}$ *International Meshing Roundtable*, pages 25–44, Seattle, WA, October 2007. Elsevier.
11. Andrey Chernikov and Nikos Chrisochoides. Three-dimensional delaunay refinement for multi-core processors. In *ACM International Conference on Supercomputing*, number 22, pages 214–224, Island of Kos, Greece, June 2008.
12. L. Paul Chew. Guaranteed-quality Delaunay meshing in 3D. In *Proceedings of the* 13$^{\text{th}}$ *ACM Symposium on Computational Geometry*, pages 391–393, Nice, France, 1997.
13. Nikos Chrisochoides, Andrey Chernikov, Andriy Fedorov, Andriy Kot, Leonidas Linardakis, and Panagiotis Foteinos. Towards exascale parallel delaunay mesh generation. In *International Meshing Roundtable*, number 18, pages 319–336, Salt Lake City, Utah, October 2009.
14. P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3d delaunay triangulation. *Computer Graphics Forum*, 12(3):129–142, 1993.
15. Olivier Devillers. The delaunay hierarchy. *Internat. J. Found. Comput. Sci*, 13:163–180, 2002.
16. Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. In *Proceedings of the* 17$^{\text{th}}$ *annual Symposium on Computational geometry*, SoCG '01, pages 106–114, New York, NY, USA, 2001. ACM.
17. Olivier Devillers and Monique Teillaud. Perturbations and vertex removal in a 3d delaunay triangulation. In *Proceedings of the* 14$^{\text{th}}$ *ACM-SIAM Symposium on Discrete algorithms*, SODA '03, pages 313–319, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.
18. Dobrina Boltcheva, Mariette Yvinec, and Jean-Daniel Boissonnat. Mesh Generation from 3D Multi-material Images. In *Medical Image Computing and Computer-Assisted Intervention*, pages 283–290. Springer, September 2009.
19. Panagiotis Foteinos, Andrey Chernikov, and Nikos Chrisochoides. Guaranteed Quality Tetrahedral Delaunay Meshing for Medical Images. In *Proceedings of the* 7$^{\text{th}}$ *International Symposium on Voronoi Diagrams in Science and Engineering*, pages 215–223, Quebec City, Canada, June 2010.
20. Jérôme Galtier and Paul-Louis George. Prepartitioning as a way to mesh subdomains in parallel. In *Special Symposium on Trends in Unstructured Mesh Generation*, pages 107–122. ASME/ASCE/SES, 1997.
21. Paul-Louis George and Houman Borouchaki. *Delaunay triangulation and meshin, Application to finite elements*. HERMES, 1998.
22. Orcun Goksel and Septimiu E. Salcudean. Image-based variational meshing. *IEEE Transactions on Medical Imaging*, 30(1):11–21, 2011.
23. Jonathan C. Hardwick. Implementation and evaluation of an efficient parallel delaunay triangulation algorithm. In *Proceedings of the* 9$^{\text{th}}$ *ACM symposium on Parallel algorithms and architectures*, pages 239–248, New York, NY, USA, 1997. ACM.

24. William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the* 24^{th} *annual ACM symposium on Principles of distributed computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.
25. Clemens Martin Joachim Kadow. *Parallel Delaunay Refinement Mesh Generation*. 2004. PhD Thesis, Carnegie Mellon University.
26. Bryan Matthew Klingner and Jonathan Richard Shewchuk. Aggressive tetrahedral mesh improvement. In *Proceedings of the International Meshing Roundtable*, pages 3–23. Springer, 2007.
27. Josef Kohout, Ivana Kolingerová, and Jiří Žára. Practically oriented parallel delaunay triangulation in E^2 for computers with shared memory. *Computers & Graphics*, 28(5):703–718, 2004.
28. Leonidas Linardakis and Nikos Chrisochoides. Graded delaunay decoupling method for parallel guaranteed quality planar mesh generation. *SIAM Journal on Scientific Computing*, 30(4):1875–1891, March 2008.
29. Ernst P. Mücke, Isaac Saias, and Binhai Zhu. Fast randomized point location without preprocessing in two- and three-dimensional delaunay triangulations. In *Proceedings of the* 12^{th} *ACM Symposium on Computational Geometry*, pages 274–283, 1996.
30. Demian Nave, Paul Chew, and Nikos Chrisochoides. Guaranteedquality parallel delaunay refinement for restricted polyhedral domains. In *ACM Symposium on Computational Geometry (SoCG)*, pages 135–144, July 2002.
31. T. Okusanya and J. Peraire. 3d parallel unstructured mesh generation, 1997. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.7898`.
32. Steve Oudot, Laurent Rineau, and Mariette Yvinec. Meshing volumes bounded by smooth surfaces. In *Proceedings of the International Meshing Roundtable*, pages 203–219, San Diego, California, USA, September 2005. Springer-Verlag.
33. Jean-Philippe Pons, Florent Ségonne, Jean-Daniel Boissonnat, Laurent Rineau, Mariette Yvinec, and Renaud Keriven. High-Quality Consistent Meshing of Multi-label Datasets. In *Information Processing in Medical Imaging*, pages 198–210, 2007.
34. Laurent Rineau and Mariette Yvinec. Meshing 3d domains bounded by piecewise smooth surfaces. In *Proceedings of the International Meshing Roundtable*, pages 443–460, 2007.
35. Jonathan Richard Shewchuk. Tetrahedral mesh generation by delaunay refinement. In *Proceedings of the* 14^{th} *ACM Symposium on Computational Geometry*, pages 86–95, Minneapolis, MN, 1998.
36. Y. Ansel Teng, Francis Sullivan, Isabel Beichl, and Enrico Puppo. A data-parallel algorithm for three-dimensional delaunay triangulation and its implementation. In *ACM Conference on Supercomputing*, pages 112–121, New York, NY, USA, 1993. ACM.
37. David F. Watson. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. *Computer Journal*, 24:167–172, 1981.
38. Yongjie Zhang, Thomas J.R. Hughes, and Chandrajit L. Bajaj. An automatic 3d mesh generation method for domains with multiple materials. *Computer Methods in Applied Mechanics and Engineering*, 199(5-8):405 – 415, 2010. Computational Geometry and Analysis.