High Quality Real-Time Image-to-Mesh Conversion for Finite Element Simulations

Panagiotis A. Foteinos[a,b], Nikos P. Chrisochoides[b]

[a] *Department of Computer Science, College of William & Mary, Williamsburg, 23185, Virginia, USA*
[b] *Department of Computer Science, Old Dominion University, Norfolk, 23569, Virginia, USA*

Abstract

In this paper, we present a parallel Image-to-Mesh Conversion (I2M) algorithm with quality and fidelity guarantees achieved by dynamic point insertions and removals. Starting directly from an image, its implementation is capable of recovering the isosurface and meshing the volume with tetrahedra of good shape. Our tightly-coupled shared-memory parallel speculative execution paradigm employs carefully designed contention managers, load balancing, synchronization and optimizations schemes. These techniques are shown to boost not only the parallel but also the single-threaded efficiency of our code. Specifically, our single-threaded performance is faster than both CGAL and TetGen, the state of the art sequential open source meshing tools we are aware of. The effectiveness of our method is demonstrated on Blacklight, the Pittsburgh Supercomputing Center's cache-coherent NUMA machine. We observe a more than 82% strong scaling efficiency for up to 64 cores, and a more than 82% weak scaling efficiency for up to 144 cores, reaching a rate of more than 14.3 million elements per second. This is the fastest 3D Delaunay mesh generation and refinement algorithm, to the best of our knowledge.

Keywords: parallel Delaunay mesh refinement; scalability; quality; fidelity; shared-memory

Email addresses: pfot@cs.wm.edu (Panagiotis A. Foteinos), nikos@cs.odu.edu (Nikos P. Chrisochoides)

1. Introduction

Image-to-mesh (I2M) conversion enables patient-specific Finite Element (FE) modeling in image guided diagnosis and therapy [1, 2]. This has significant implications in many areas, such as imaged-guided therapy, development of advanced patient-specific blood flow simulations for the prevention and treatment of stroke, patient-specific interactive surgery simulation for training young clinicians, and study of bio-mechanical properties of collagen nano-straws of patients with chest wall deformities, to name just a few.

In this paper, we present a 3D Delaunay parallel Image-to-Mesh conversion algorithm (abbreviated as PI2M) that (a) recovers the isosurface of the biological object with geometric and topological guarantees and (b) meshes the underlying volume with tetrahedra of high quality. These two characteristics render our method suitable for subsequent FE analysis, since the robustness and accuracy of the solver rely on the quality of the mesh [3–5].

PI2M recovers the tissues' boundaries and generates quality meshes through a sequence of dynamic insertion and deletion of points which is computed on the fly and in parallel during the course of refinement. To the best of our knowledge, none of the parallel Delaunay refinement algorithms support point removals. Point removal, however, offers new and rich refinement schemes which are shown in the sequential meshing literature [6, 7] to be very effective in practice.

Our implementation employs low level locking mechanisms, carefully designed contention managers, and well-suited load balancing schemes that not only boost the parallel performance, but they exhibit very little overhead: our single threaded performance is more than 10 times faster than our previous sequential prototype [7, 8] and it is faster than CGAL [9] and TetGen [10], the state of the art optimized sequential open source meshing tools. Specifically, PI2M is consistently 40% faster than CGAL. We also compare PI2M with TetGen [10] and show that PI2M is faster on generating large meshes (i.e., meshes consisting of more than $900,000$ tetrahedra) by 35%. Considering the fact that both CGAL and TetGen perform insertions via the Bowyer-Watson kernel [11, 12], as is the case of PI2M, such a comparison is quite insightful.

Parallel Delaunay refinement is a highly irregular and data-intensive application and as such, it is very dynamic in terms of resource management. Implementing an efficient parallel Delaunay refinement would help the community gain insight into a whole family of problems characterized by unpredictable communication patterns [13]. We test and show the effectiveness

of PI2M on the cc-NUMA architecture. Demonstrating the performance of mesh refinement on cc-NUMA architectures illuminates the characteristic challenges of irregular applications on the many-core chips featuring dozens of cores. But even the biggest distributed-memory machines consist of groups of cores that, from our application's point of view and supporting software, can be treated as cc-NUMA. The efficient utilization of such deep architectures can be achieved by employing a tightly-coupled approach inside each group (i.e., the ideas of this paper), and by being less explorative in the other layers, as we stated in more detail in [14].

Specifically, we used the Pittsburgh Supercomputing Center's Blacklight, employing BoostC++ threads. Although the ideas of this paper could be programmed using the more general MPI programming model, we chose threads, since the maintenance of threads is typically faster in shared-memory machines [15].

Experimental evaluation shows a more than 82% strong scaling efficiency for up to 64 cores, and a more than 82% weak scaling efficiency for up to 144 cores, reaching a rate of more than 14.3 million elements per second. We are not aware of any 3D parallel Delaunay refinement method achieving such a performance, on either distributed or shared-memory architectures. However, for a higher core count, our method exhibits considerable performance degradation. We argue that this deterioration is not because of load imbalance or high thread contention, but because of the intensive and hop-wise slower communication traffic involved in increased problem sizes, large memories, and cache coherency protocols. This problem could be potentially alleviated by using hybrid approaches to explore network hierarchies [14, 16]. However, this is outside the scope of the paper. Our goal is to develop the most efficient and scalable method on a moderate number (~ 100) of cores. Our long term goal is to increase scalability by exploiting concurrency at different levels [14].

In summary, the method we present (PI2M):

- Exhibits the best single-threaded performance, to the best of our knowledge.
- Supports parallel Delaunay insertions and removals; past methods including ours dealt only with point insertions, a much easier task.
- Conforms to user-specified quality, and most importantly, to the newly added and novel, for parallel mesh generation, fidelity constraints.

- Recovers and meshes the isosurface with topological and geometric guarantees from the beginning of mesh refinement, and thus exploits parallelism earlier (see Figure 6).
- Fills the volume of the underlying biological object with millions of high-quality tetrahedra within seconds, achieving an unprecedented rate of 14.3 million elements per second.

Section 2 presents the related work. Section 3 covers the background and briefly describes the Sequential Delaunay Refinement for Smooth Surfaces. Section 4 outlines the basic building blocks of our parallel implementation. Section 5 presents the Contention Managers. Section 6 presents the strong and weak scaling results together with load balancing improvements. Section 7 is dedicated to single-threaded evaluation. Section 8 summarizes our findings and concludes the paper.

2. Related Work

Volume mesh generation methods can be divided into two categories: *PLC-based* and *Isosurface-based*. The PLC-based methods assume that the *surface* $\partial\mathcal{O}$ of the object $\mathcal{O}$ is given as a *Piecewise Linear Complex* (PLC) which contains linear segments and polygonal facets in 3 dimensions [10, 17–20]. The challenge of this method is that the success of meshing depends on the quality of the given PLC: if the PLC forms very small angles, then termination might be compromised [17, 21]. In Computed Aided Design (CAD) applications [5, 17, 21, 22], the surface is usually given as a PLC. In biomedical Computer Aided Simulations (CAS), however, there is no reason to use this approach, since it would add the additional small input angle limitation.

The Isosurface-based methods assume that $\mathcal{O}$ is known through a function $f : \mathbb{R}^3 \to \mathbb{R}$, such that points in different regions of interest evaluate f differently. This assumption covers a wide range of inputs used in modeling and simulation, such as parametric surfaces/volumes [23], level-sets and segmented multi-labeled images [24–26]. Of course, these type of functions can also represent PLCs [24], a fact that makes the Isosurface-based method a general approach. Isosurface-based methods ought to recover and mesh both the *isosurface* $\partial\mathcal{O}$ and the volume. This method does not suffer from any small input angle artifacts introduced by the initial conversion to PLCs, since $\partial\mathcal{O}$ is recovered and meshed during refinement.

In this paper, we present a parallel high quality Delaunay Image-to-Mesh Conversion (PI2M) Isosurface-based algorithm which, starting from a multi-labeled segmented image, recovers the isosurface(s) $\partial\mathcal{O}$ of the object(s) $\mathcal{O}$ and meshes the volume simultaneously. Our method is able to produce millions of high-quality elements within seconds respecting at the same time the exterior and interior boundaries of tissues. As far as we know, PI2M is the first parallel Isosurface-based finite element mesh generation method.

In the parallel mesh generation literature, only PLC-based methods have been considered. That is, either $\mathcal{O}$ is given as an initial mesh [27–30] or $\partial\mathcal{O}$ is already represented as a polyhedral domain [31–34]. We, on the contrary, mesh both the volume and the isosurface directly from an image and not from a polyhedral domain. This flexibility offers great control over the trade-off between quality and fidelity: parts of the isosurface of high curvature can be meshed with more elements of better quality. Moreover, our method is able to satisfy both surface and volume custom element densities, as dictated by the user-specified size functions. This is not the case of algorithms that treat the surface voxels as the PLC of the domain [20, 35, 36], since the size of the elements is determined by the voxel spacing, a fact that offers little control over the mesh density. In the future, we also plan to incorporate in our parallel framework the computational intensive smoothing of the mesh boundary for CFD applications, e.g. lung modeling [37–39].

In our previous work [40], we implemented a parallel *Triangulator* able to support fully dynamic insertions and removals. Our parallel Triangulator, however, has one major limitation: as is the case with all Triangulators [40–43], it tessellates only the convex hull of a set of points, and it is not concerned with any quality or fidelity constraints imposed by the input geometry and the user. Also, in parallel triangulation literature [41–43], the pointset, whose convex hull is to be constructed, is static and given before the algorithm starts. In this paper, we extend our previous work [40], such that the discovery of the dynamically changing set of points, which are being inserted or removed in order to satisfy the quality and fidelity constraints, is performed in parallel as well: a very dynamic process that increases parallel complexity even more. This is neither incremental nor a trivial extension.

There is extensive previous work on parallel mesh generation, including various techniques, such as: Delaunay, Octree, or Advancing Front meshing. Parallel mesh generation/refinement should not be confused with parallel triangulation [40–43]. Triangulation tessellates the convex hull of a given, static set of points. Mesh generation focuses on element quality and the

conformity to the tissues' boundary, which necessitates the parallel insertion or removal of points which are gradually and concurrently discovered through refinement.

One of the main differences between our method and previous work is that in the literature the surface of the domain is either given as a polyhedron, or the extraction of the polyhedron is done sequentially, or refinement starts from an initial background octree. As explained in this Section above, our method constructs the polyhedral representation of the object's surface from scratch, and therefore, it adds extra functionality. This surface recovery is also performed in parallel, together with the volume meshing, thus taking advantage of another degree of parallelism.

Given an initial mesh, de Cougny and Shephard [27] dynamically repartition the domain such that every processor has equal work. They also describe "vertex snapping", a method that can be used for the representation of curved boundaries, but they give no guarantees about the achieved fidelity (both geometrically and topologically).

In our past work [33], we implemented a tightly-coupled method like ours. However, in this paper, we take extra care to greatly reduce the number of rollbacks (see Section 5), and thus achieve scalability for a higher core count. In [44] and [34], our group devised a partially-coupled and a decoupled method for distributed-memory systems based on Medial Axis decomposition. However, Medial Axis decomposition for general 3D domains is a challenging problem and still open. In contrast, the method presented in this paper does not rely on any domain decomposition, and as such, it is flexible enough to be extended to arbitrary dimensions, a goal that is left for future work. In [45], our group presented a method which allows for safe insertion of points independently without synchronization. Although the method in [45] improves data locality and decreases communication, it exhibits little scalability on more than 8 cores because the initial bootstrapping, needed as a pre-processing step, is performed sequentially and not in parallel.

Kadow [32] starts from a polygonal surface (PSLG) and offers tightly coupled refinement schemes in 2D only. In our case, the polyhedral representation of the object's surface is performed in parallel, which adds extra functionality and available parallelism. Galtier and George [31] compute a smooth separator and distribute the subdomains to distinct processors. However, the separators they create might not be Delaunay-admissible and thus they need to restart the process from the beginning. Weatherill *et al.* [46]

subdivide the domain into decoupled blocks. Each block then is meshed with considerably less communication and synchronization. Nevertheless, the generated mesh is not Delaunay, a property that is critical to applications like large scale electro-magnetics [47].

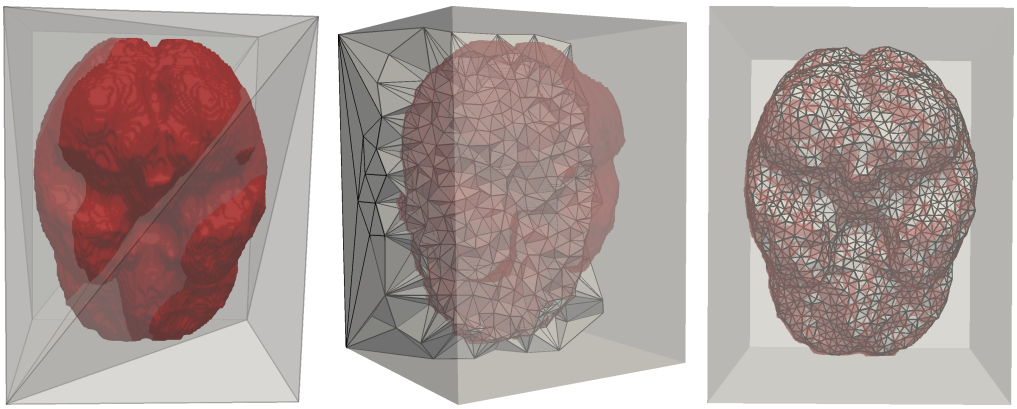
Tu *et al.* [29] describe a parallel octree method that interacts with the solver in parallel and efficiently, but the fidelity and conformity of the meshes to complex multi-material junctions/interfaces (one of this paper's goals) was not their main focus. The work of Zhou *et al.* [48], and the *Forest-of-octrees* method of Burstedde *et al.* [28] offer techniques for fair and efficient data migration and partitioning in parallel. In our application, however, we show that the main bottleneck that hampers scalability is not load imbalance (see Subsection 6.1), but the rollbacks (see Section 5) and the memory pressure in the switches (see Section 6.3). Load balancing and data migration is also used by Okusanya and Peraire [49] to distribute bad elements across processors, but the performance reported is rather low, as the speedup achieved on 8 cores is shown to be less than 2.4.

Ito *et al.* [30] start from an initial mesh and Löhner [50] from a PLC for subsequent parallel mesh generation in advancing front fashion. It should be noted, however, that advancing front methods guarantee neither termination nor good quality meshes.

Oliker and Biswas [51] employ three different architectures to test the applicability of 2D adaptive mesh refinement. They conclude that unstructured mesh refinement is not suitable for cc-NUMA architectures: irregular communication patterns and lack of data locality deteriorate performance sometimes even on just 4 cores. In this paper, we show that this becomes a problem on a much higher core count (more than 144 cores); i.e., with this work, we push the envelop even further. Clearly, this approach has its own limitations, but a highly scalable and efficient NUMA implementation combined with the decoupled and partially coupled approaches we developed in the past can allow us to explore concurrency levels in the order of at least 10^8 to 10^{10} [14].

3. Background: Delaunay Refinement for Smooth Surfaces

Sequential Delaunay Refinement for smooth surfaces is presented in detail in the literature [23, 25] and in our previous work [7, 8]. In this Section, we briefly outline the main concepts.



(a) (b) (c)

Figure 1: (a) The virual box is meshed into 6 tetrahedra. It encloses the volumetric object. (b) During refinement, the final mesh is gradually being carved according to the Rules. (c) At the end, the set of the tetrahedra whose circumcenter lies inside $\mathcal{O}$ is the geometrically and topologically correct mesh $\mathcal{M}$.

As is usually the case in the literature [24, 25, 52], we assume that the surface of the object $\partial\mathcal{O}$ to be meshed is a closed smooth 2-manifold. To prove that the boundary $\partial\mathcal{M}$ of the final mesh $\mathcal{M}$ is geometrically and topologically equivalent with $\partial\mathcal{O}$, we make use of the *sample theory* [52]. Omitting the details, it can be proved [52, 53] that the Delaunay triangulation of a dense pointset lying precisely on the isosurface $\partial\mathcal{O}$ contains (as a subset) the correct mesh $\mathcal{M}$. That mesh consists of the tetrahedra t whose circumcenter $c(t)$ lies inside $\mathcal{O}$. Formally, the sample theorem could be stated as follows [53–55]:

Theorem 1. *Let V be samples of $\partial\mathcal{O}$. If for any point $p \in \partial\mathcal{O}$, there is a sample $v \in V$ such that $|v - p| \leq \delta$, then the boundary triangles of $\mathcal{D}_{|\mathcal{O}}(V)$ is a topologically correct representation of $\partial\mathcal{O}$. Also, the 2-sided Hausdorff distance between the mesh and $\partial\mathcal{O}$ is $O(\delta^2)$.*

Typical values for δ are usually fractions of the *local feature size* of $\partial\mathcal{O}$. See [23, 53–55] for well defined δ parameters. In our application, δ values equal to multiples of the voxel size is sufficient.

Therefore, one of the goals of the refinement is to sample the isosurface densely enough. To achieve that, our algorithm first constructs a *virtual*

box which encloses $\mathcal{O}$. The box is then triangulated into 6 tetrahedra, as shown in Figure 1. This is the only sequential part of our method. Next, it dynamically computes new points to be inserted into or removed from the mesh maintaining the Delaunay property. This process continues, until certain fidelity and quality criteria are met. Specifically, the vertices removed or inserted are divided into 3 groups: *isosurface* vertices, circumcenters, and *surface-centers*.

The isosurface vertices will eventually form the sampling of the surface so that Theorem 1 holds together with its theoretical guarantees about the fidelity of the mesh boundary. Let $c(t)$ be the circumcenter of a tetrahedron t. In order to guarantee termination, our algorithm inserts the isosurface vertex which is the closest to $c(t)$. In the sequel, we shall refer to the *Closest IsoSurface* vertex of a point p as $\hat{p} \in \partial\mathcal{O}$. The isosurface vertices (like the circumcenters) are computed during the refinement dynamically with the help of a parallel Euclidean Distance Transformation (EDT) presented and implemented in [56]. Specifically, the EDT returns the *surface voxel* q which is closest to p. A surface-voxel is a voxel that lies inside the foreground and has at least one neighbor of different label. Then, we traverse the ray $\overrightarrow{pq}$ on small intervals and we compute $\hat{p} \in \partial\mathcal{O}$ by interpolating the positions of different labels [57]. The density of the inserted isosurface vertices is defined by the user by a parameter $\delta > 0$. A low value for δ implies a denser sampling of the surface, and therefore, according to Theorem 1, a better approximation of $\partial\mathcal{O}$.

The circumcenter $c(t)$ of a tetrahedron t is inserted when t has low quality (in terms of its radius-edge ratio [17]) or because its circumradius $r(t)$ is larger than a user-defined size function $\mathrm{sf}(\cdot)$. Circumcenters might also be chosen to be removed, when they lie close to an isosurface vertex, because in this case termination is compromised.

Consider a facet f of a tetrahedron. The *Voronoi* edge $V(f)$ of f is the segment connecting the circumcenters of the two tetrahedra that contain f. The intersection $V(f) \cap \partial\mathcal{O}$ is called a *surface-center* and is denoted by $c_{\mathrm{surf}}(f)$. During refinement, surface-centers are computed similarly to the isosurfaces (i.e., by traversing $V(f)$ on small intervals and interpolating positions of different labels) and inserted into the mesh to improve the planar angles of the boundary mesh triangles [22] and to ensure that the vertices of the boundary mesh triangles lie precisely on the isosurface [23].

In summary, tetrahedra and faces are refined according to the following *Refinement Rules*:

- **R1**: Let t be a tetrahedron whose circumball intersects $\partial\mathcal{O}$. Compute the closest isosurface point $z = c\,\hat{(t)}$. If z is at a distance not closer than δ to any other isosurface vertex, then z is inserted.
- **R2**: Let t be a tetrahedron whose circumball intersects $\partial\mathcal{O}$. If its radius $r(t)$ is larger than $2 \cdot \delta$, then $c\,(t)$ is inserted.
- **R3**: Let f be a facet whose Voronoi edge $V(f)$ intersects $\partial\mathcal{O}$ at $c_{\text{surf}}\,(f)$. If either its smallest planar angle is less than 30° or a vertex of f is not an isosurface vertex, then $c_{\text{surf}}\,(f)$ is inserted.
- **R4**: Let t be a tetrahedron whose circumcenter lies inside $\mathcal{O}$. If its radius-edge ratio is larger than 2, then $c\,(t)$ is inserted.
- **R5**: Let t be a tetrahedron whose circumcenter lies inside $\mathcal{O}$. If its radius $r(t)$ is larger than $\text{sf}\,(c\,(t))$, then $c\,(t)$ is inserted.
- **R6**: Let t be incident to an isosurface vertex z. All the already inserted circumcenters closer than 2δ to z are deleted.

Rules R1 and R2 are responsible for creating the appropriate dense sample so that the boundary triangles of the resulting mesh satisfies Theorem 1 and thus the fidelity guarantees. R3 and R4 deal with the quality guarantees, while R5 imposes the size constraints of the users. R6 is needed so termination can be guaranteed. See [7, 8, 23] for more details. When none of the above rules applies, then refinement is complete. In our previous work [7, 8], we prove that termination is guaranteed, the radius-edge ratio of all elements in the mesh is less than 2, and the planar angles of the boundary mesh triangles is less than 30°.

4. Parallel Delaunay Refinement for Smooth Surfaces

As explained in Section 3, before the mesh generation starts, the Euclidean Distance Transform (EDT) of the image is needed for the on-the-fly computation of the appropriate iso-surface vertices. For this pre-processing step, we make use of the publicly available parallel Maurer filter presented and implemented by Staubs *et al.* [56]. It can be shown [56, 58] that this parallel EDT scales linearly with the respect to the number of threads.

The rest of this section describes the main aspects of our parallel code. Algorithm 1 illustrates the basic building blocks of our multi-threaded mesh-generation design. Note that our tightly-coupled parallelization does not

```
1  Algorithm: GenerateMesh(I, δ, ρ̄, sf(·), tid)
   Input  : I is the image containing O,
            δ is the parameter that determines the density of the surface sampling,
            ρ̄ (≥ 2) is the target radius-edge ratio,
            sf(·) is the size function,
            tid is the unique identifier of the thread.
   Output: A Delaunay mesh M that is guaranteed to (a) approximate ∂O in a correct topological way with
            Hausdorff distance within O(δ^2), (b) be composed of elements with radius-edge ratio less than ρ̄ and
            (c) have boundary facets with planar angles larger than 30°.
2  if tid == 0 then                                                          /* If it is the main thread */
       /* At this moment, both the mesh and all PELs are empty.                                          */
3      Insert the 8 vertices of a box which contains O;
4      PEL_0 = PEL_0 ∪ NewElements;
5  end
6  while PEL_tid ≠ ∅ do
7      t =PEL_tid → next();
8      if locking t's vertices is not successful then
9          Unlock related vertices; Invoke Contention Manager; continue;
10     end
11     if t is an intersecting tetrahedron then
12         Compute z = c(t);                                          /* potential R1 element */
13         if there is an iso-surface vertex closer than δ to z then
14             if r(t) ≥ 2δ then
15                 Compute z = c(t);                                          /* R2 element */
16             end
17         end
18     else
19         if t is adjacent to a restricted facet f, such that ρ(f) ≥ 1 or f's vertices do not lie on ∂O then
20             Compute z = c_surf(f);                                  /* R3 applies.  */
21         else
22             if c(t) lies inside O and either ρ(t) ≥ ρ̄ or r(t) ≥ sf(c(t)) then
23                 Compute z = c(t);                                  /* R4 or R5 apply.  */
24             else
25                 PEL_tid = PEL_tid − t;                          /* t is not a poor element */
26                 Unlock all the related vertices; continue;
27             end
28         end
29     end
30     if z is a isosurface vertex then
31         Prepare to delete all the free vertices that are closer than 2δ to z.
32     end
33     if locking the vertices for the operation is not successful then
34         Rollback; Unlock related vertices; Invoke Contention Manager; continue;
35     end
36     Insert z and delete the vertices (if any); Unlock all the related vertices;
37     if BeggingList ≠ ∅ then
38         other_tid = BeggingList→first();
39         PEL_other_id = PEL_other_id ∪ NewElements;          /* Give work to begging Thread other_id */
40         Wake Thread other_id;                  /* Notify Thread other_id that it can check its PEL again */
41         BeggingList = BeggingList - {other_id};
42     end
43 end
44 if BeggingList→size() != #Threads -1 then                  /* If I am NOT the last Thread to ask for work */
45     BeggingList→push_at_end(tid);
46     Wait;
47     continue ;          /* Now some other thread gave Thread tid work, so PEL_tid is not empty any more */
48 else                                                        /* The mesh is ready, all PELs are empty */
49     Let the final mesh M be equal to the set of the tetrahedra whose circumcenter lies inside O;
50 end
```

Algorithm 1: The parallel mesh generation algorithm. It is executed by each thread.

alter the fidelity (Theorem 1) and the quality guarantees described in the previous section.

4.1. Poor Element List (PEL)

Each thread T_i maintains its own *Poor Element List (PEL)* PEL_i. PEL_i contains the tetrahedra that violate the Refinement Rules and need to be refined by thread T_i accordingly.

4.2. Operation

An operation that refines an element can be either an insertion of a point p or the removal of a vertex p. In the case of insertion, the cavity $\mathcal{C}(p)$ needs to be found and re-triangulated according to the well known Bowyer-Watson kernel [11, 12]. Specifically, $\mathcal{C}(p)$ consists of the elements whose circumsphere contains p. These elements are deleted (because they violate the Delaunay property) and p is connected to the vertices of the boundary of $\mathcal{C}(p)$. In the case of a removal, the ball $\mathcal{B}(p)$ needs to be re-triangulated. As explained in [59], this is a more challenging operation than insertion, because the re-triangulation of the ball in degenerate cases is not unique which implies the creation of illegal elements, i.e., elements that cannot be connected with the corresponding elements outside the ball. We overcome this difficulty by computing a *local Delaunay triangulation* $\mathcal{D}_{\mathcal{B}(p)}$ (or $\mathcal{D}_{\mathcal{B}}$ for brevity) of the vertices incident to p, such that the vertices inserted earlier in the shared triangulation are inserted into $\mathcal{D}_{\mathcal{B}}$ first. In order to avoid races associated with writing, reading, and deleting vertices/cells from a PEL or the shared mesh, any vertex touched during the operation of cavity expansion, or ball filling needs to be locked. We utilize GCC's atomic built-in functions for this goal, since they perform faster than the conventional pthread try_locks. Indeed, replacing pthread locks (our first implementation) with GCC's atomic built-ins (current implementation) decreased the execution time by 3.6% on 1 core and by 4.2% on 12 cores.

In the case a vertex is already locked by another thread, then we have a *rollback*: the operation is stopped and the changes are discarded [33]. When a rollback occurs, the thread moves on to the next bad element in its PEL.

4.3. Update new and deleted cells

After a thread T_i completes an operation, new cells are created and some cells are invalidated. The new cells are those that re-triangulate the cavity (in case of an insertion) or the ball (in case of a removal) of a point p and the

invalidated cells are those that used to form the cavity or the ball of p right before the operation. T_i determines whether a newly created element violates a rule. If it does, then T_i pushes it back to PEL_i (or to another thread's PEL, see below) for future refinement. Also, T_i removes the invalidated elements from the PEL they have been residing in so far, which might be the PEL of another thread. To decrease the synchronization involved for the concurrent access to the PELs, if the invalidated cell c resides in another thread T_j's PEL_j, then T_i removes c from PEL_j only if T_j belongs to the same socket with T_i. Otherwise, T_i raises cell c's invalidation flag, so that T_j can remove it when T_j examines c.

As Line 49 of Algorithm 1 shows, the final mesh $\mathcal{M}$ reported consists of the subset of tetrahedra whose circumcenter lies inside the object $\mathcal{O}$. To expedite the process of finding those elements, each thread maintains a linked list of those elements on the fly, i.e., from the beginning of mesh generation and refinement. Thus, collecting those elements at the end costs constant time O(#Threads). These linked lists are updated similarly to the update of the Poor Element Lists (PELs) described in the previous paragraph.

4.4. Load Balancer

Right after the triangulation of the virtual box and the sequential creation of the first 6 tetrahedra, only the main thread might have a non-empty PEL. Clearly, Load Balancing is a fundamental aspect of our implementation. Our base (not optimized) Load Balancer is the classic *Random Work Stealing* (RHW) [60] technique, since it best fits our implementation design. In Section 6.1, we implement an optimized work stealing balancer that takes advantage of the NUMA architecture and achieves an excellent performance.

If the poor element list PEL_i of a thread T_i is empty of elements, T_i "pushes back" its ID to the *Begging List*, a global array that tracks down threads without work. Then, T_i is busy-waiting and can be awaken by a thread T_j right after T_j gives some work to T_i. A running thread T_j, every time it completes an *operation* (i.e., a Delaunay insertion or a Delaunay removal), it gathers the newly created elements and places the ones that are poor to the PEL of the first thread T_i found in the begging list. The classification of whether or not a newly created cell is poor or not is done by T_j. T_j also removes T_i from the Begging List.

To decrease unnecessary communication, a thread is not allowed to give work to threads, if it does not have enough poor elements in its PEL. Hence, each thread T_i maintains a counter that keeps track of all the poor and *valid*

cells that reside in PEL_i. T_i is forbidden to give work to a thread, if the counter is less than a threshold. We set that threshold equal to 5, since it yielded the best results. When T_i invalidates an element c or when it makes a poor element c not to be poor anymore, it decreases accordingly the counter of the thread that contains c in its PEL. Similarly, when T_i gives extra poor elements to a thread, T_i increases the counter of the corresponding thread.

4.5. Contention Manager (CM)

In order to eliminate livelocks caused by repeated rollbacks, threads talk to a Contention Manager (CM). Its purpose is to pause on run-time the execution of some threads making sure that at least one will do useful work so that system throughput can never get stuck [61]. See Section 5 for approaches able to greatly reduce the number of rollbacks and yield a considerable speedup, even in the absence of enough parallelism. Contention managers avoid energy waste because of rollbacks and reduce dynamic power consumption, by throttling the number of threads that contend, thereby providing an opportunity for the runtime system to place some cores in deep low power states.

5. Contention Manager

The goal of the Contention Manager (CM) is to reduce the number of rollbacks and guarantee the absence of livelocks, if possible [61, 62].

We implemented and compared four contention techniques: the *Aggressive Contention Manager* (Aggressive-CM) [61], the *Random Contention Manager* (Random-CM), the *Global Contention Manager* (Global-CM), and the *Local Contention Manager* (Local-CM).

The Aggressive-CM and Random-CM are non-blocking schemes. As is usually the case for non-blocking schemes [33, 61–64], we do not prove absence of livelocks for these techniques. Nevertheless, they are useful for comparison purposes as Aggressive-CM is the simplest to implement, and Random-CM has already been presented in the mesh generation literature [33, 63, 64].

The Global-CM is a blocking scheme and we prove that does not introduce any deadlock. (Blocking schemes are guaranteed not to introduce livelocks [65]).

The last one, Local-CM, is semi-blocking, that is, it has both blocking and non-blocking parts. Because of its (partial) non-blocking nature, we found it difficult to prove starvation-freedom [62, 66], but we could guarantee absence of deadlocks and livelocks. It should be noted, however, that we have never

experience any thread starvation when using Local-CM: all threads in all case studies are making progress concurrently for about the same period of time.

Note that none of the earlier Transactional Memory techniques [61, 62] and the Random Contention Managers presented in the past [33, 63, 64] solve the livelock problem. In this section, we show that if livelocks are not provably eliminated in our application, then termination is compromised on high core counts.

For the next of this Section assume that (without loss of generality) each thread always finds elements to refine in its Poor Element List (PEL). This assumption simplifies the presentation of this Section, since it hides several details that are mainly related to Load Balancing. The interaction between the Load Balancing and the Contention Manager techniques does not invalidate the proofs of this Section.

5.1. Aggressive-CM

The Aggressive-CM is a brute-force technique, since there is no special treatment. Threads greedily attempt to apply the operation, and in case of a rollback, they just discard the changes, and move on to the next poor element to refine (if there is any). The purpose of this technique is to show that reducing the number of rollbacks is not just a matter of performance, but a matter of correctness. Indeed, experimental evaluation (see Section 5.5) shows that Aggressive-CM very often suffers from livelocks.

5.2. Random-CM

Random-CM has already been presented (with minor differences) in the literature [33, 63, 64, 67] and worked fairly well, i.e, no livelocks were observed in practice. This scheme lets "randomness" choose the execution scenario that would eliminate livelocks. We implement this technique as well to show that our application needs considerably more elaborate CMs. Indeed, recall that in our case, there is no much parallelism in the beginning of refinement and therefore, there is no much randomness that can be used to break the livelock.

Each thread T_i counts the number of consecutive rollbacks r_i. If r_i exceeds a specified upper value r^+, then T_i sleeps for a random time interval t_i. If the consecutive rollbacks break because an operation was successfully finished then r_i is reset to 0. The time interval t_i is in milliseconds and is a randomly generated number between 1 and r^+. The value of r^+ is set to 5. Other values

yielded similar results. Note that lower values for r^+ do not necessarily imply faster executions. A low r^+ decreases the number of rollbacks much more, but increases the number of times that a contented thread goes to sleep (for t_i milliseconds). On the other hand, a high r^+ increases the number of rollbacks, but randomness is given more chance to avoid livelocks; that is, a contented thread has now more chances to find other elements to refine before it goes to sleep (for t_i milliseconds).

Random-CM cannot guarantee the absence of livelocks. As noted in [65], this randomness can rarely lead to livelocks, but it should be rejected as it is not a valid solution. We also experimentally verified that livelocks are not that rare (see Section 5.5).

5.3. Global-CM

Global-CM maintains a global *Contention List* (CL). If a thread T_i encounters a rollback, then it writes its id in CL and it busy waits (i.e., it blocks). Threads waiting in CL are potentially awaken (in FIFO order) by threads that have made a lot of progress, or in other words, by threads that have not recently encountered many rollbacks. Therefore, each thread T_i computes its "progress" by counting how many consecutive successful operations s_i have been performed without an interruption by a rollback. If s_i exceeds a upper value s^+, then T_i awakes the first thread in CL, if any. The value for s^+ is set to 10. Experimentally, we found that this value yielded the best results.

Global-CM can never create livelocks, because it is a blocking mechanism as opposed to random-CM which does not block any thread. Nevertheless, the system might end up to a deadlock, because of the interaction with the Load Balancing's Begging List BL (see the Load Balancer in Section 4).

Therefore, at any time, the number of *active threads* needs to be tracked down, that is, the number of threads that do not busy wait in either the CL or the Begging List. A thread is forbidden to enter CL and busy wait, if it sees that there is only one (i.e., itself) active thread; instead, it skips CL and attempts to refine the next element in its Poor Element List. Similarly, a thread about to enter the Begging List (because it has no work to do) checks whether or not it is the only active thread at this moment, in which case, it awakes a thread from the CL, before it starts idling for extra work. In this simple way, the absence of livelocks and deadlocks are guaranteed, since threads always block in case of a rollback and there will always be at least one active thread. The disadvantage of this method is that there is

global communication and synchronization: the CL, and the number of active threads are global structures/variables that are accessed by all threads.

5.4. Local-CM

The local Contention Manager (local-CM) distributes the previously global Contention List (CL) across threads. The Contention List CL_i of a thread T_i contains the ids of threads that encountered a rollback because of T_i (i.e, they attempted to acquire a vertex already acquired by T_i) and now they busy wait. As above, if T_i is doing a lot of progress, i.e., the number of consecutive successful operations exceed s^+, then T_i awakes one thread from its local CL_i.

Extra care should be taken, however, to guarantee not only the absence of livelocks, but also, the absence of deadlocks. It is possible that T_1 encounters a rollback because of T_2 (and we symbolize this relationship by writing $T_1 \rightarrow T_2$), and T_2 encounters a rollback because of T_1 (i.e., $T_2 \rightarrow T_1$): both threads write their ids to the other thread's CL, and no one else can wake them up. Clearly, this *dependency cycle* ($T_1 \rightarrow T_2 \rightarrow T_1$) leads T_1 and T_2 to a deadlock, because under no circumstances these threads will ever be awaken again.

To solve these issues, each thread is now equipped with two extra variables: conflicting_id and busy_wait. See Figure 2 for a detailed pseudo-code of local-CM.

The algorithm in Figure 2c is called by a T_i every time it does not finish the operation successfully (i.e., it encounters a rollback). Suppose T_i attempts to acquire a vertex already locked by T_j ($T_i \rightarrow T_j$). In this case, T_i does not complete the operation, but rather, it rollbacks by disregarding the so far changes, unlocking all the associated vertices, and finally executing the `Rollback_Occurred` function, with conflicting_id equal to j. In other words, the conflicting_id variables represent dependencies among threads: $T_i \rightarrow T_j \Leftrightarrow T_i.\text{conflicting\_id} = j$.

For example, if T_i encounters a rollback because of T_j and T_j encounters a rollback because of T_k, then the dependency path from T_i is $T_i \rightarrow T_j \rightarrow T_k$, which corresponds to the following values: $T_i.\text{conflicting\_id} = j, T_j.\text{conflicting\_id} = k, T_k.\text{conflicting\_id} = -1$ (where -1 denotes the absence of dependency).

Lines 4-14 of `Rollback_Occurred` decide whether or not T_i should block (via busy-waiting). T_i is not allowed to block if $T_{\text{conflicting\_id}}$ has already decided to block (Lines 6-10). Threads communicate their decision to block by setting their busy_wait flags to true. If $T_{\text{conflicting\_id}}.\text{busy\_wait}$ has already

```
1  Algorithm: Initialization(T,i)
   Input : T is the array of threads,
           i (≥ 0) is the id of the running
   thread T_i.
   /* s tracks down the progress of
      T_i. It counts the number of
      consecutive operations that
      finished successfully without
      rollback.                          */
2  T[i].s = 0;
   /* conflicting_id establishes
      dependencies. If conflicting_id
      is not a negative number that
      means T_i rollbacks because it
      attempted to acquire a vertex
      already owned by T_conflicting_id.
      */
3  T[i].conflicting_id = -1;
   /* busy_wait implements the busy
      waiting.                           */
4  T[i].busy_wait = false;
```

(a) It is called by each thread, before refinement starts.

```
1  Algorithm:
   Rollback_Not_Occurred(T,i)
   Input : T is the array of threads,
           i (≥ 0) is the id of the running
   thread T_i which completed an operation
   successfully, i.e., without rollbacks.
2  T[i].s++;
3  if T[i].s ≤ s+ then
       /* T_i does not awake any thread
          yet.                           */
4      return;
5  end
6  T[i].mutex.lock();
7  j = T[i].CL.pop_front();
8  T[i].mutex.unlock();
   /* Flip T_j's flag, so it can be
      awaken.                            */
9  T[j].busy_wait = false;
```

(b) T_i completed the operation.

```
1  Algorithm: Rollback_Occurred(T, i,
   conflicting_id)
   Input : T is the array of threads,
           i (≥ 0) is the id of the running
   thread T_i which attempted to acquire a
   vertex already locked by the thread
   T_conflicting_id.
   /* The number of consecutive
      successful operations is reset to
      0.                                 */
2  T[i].s = 0;
3  T[i].conflicting_id = conflicting_id;
4  T[min(i,conflicting_id)].mutex.lock();
5  T[max(i,conflicting_id)].mutex.lock();
6  if T[conflicting_id ].busy_wait then
       /* T_conflicting_id is very likely to be
          busy waiting; to avoid cyclic
          dependencies, T_i is forbidden
          to busy wait.                  */
7      T[i].conflicting_id = -1;
8      T[max(i,conflicting_id)].mutex.unlock();
9      T[min(i,conflicting_id)].mutex.unlock();
10     return;
11 end
   /* T_conflicting_id is not busy waiting;
      atomically, T_i will.              */
12 T[i].busy_wait = true;
13 T[max(i,conflicting_id)].mutex.unlock();
14 T[min(i,conflicting_id)].mutex.unlock();
   /* T_i writes its id in T_conflicting_id's
      Contention_List (CL).              */
15 T[conflicting_id ].mutex.lock();
16 T[conflicting_id ].CL.push_back(i);
17 T[conflicting_id ].mutex.unlock();
18 while T[i].busy_wait do
       /* T_i is busy waiting until thread
          T_conflicting_id wakes it up.  */
19 end
20 T[i].conflicting_id = -1;
```

(c) T_i did not complete the operation because it encountered a rollback.

Figure 2: Pseudocode elaborating on the implementation of the Local Contention Manager (Local-CM).

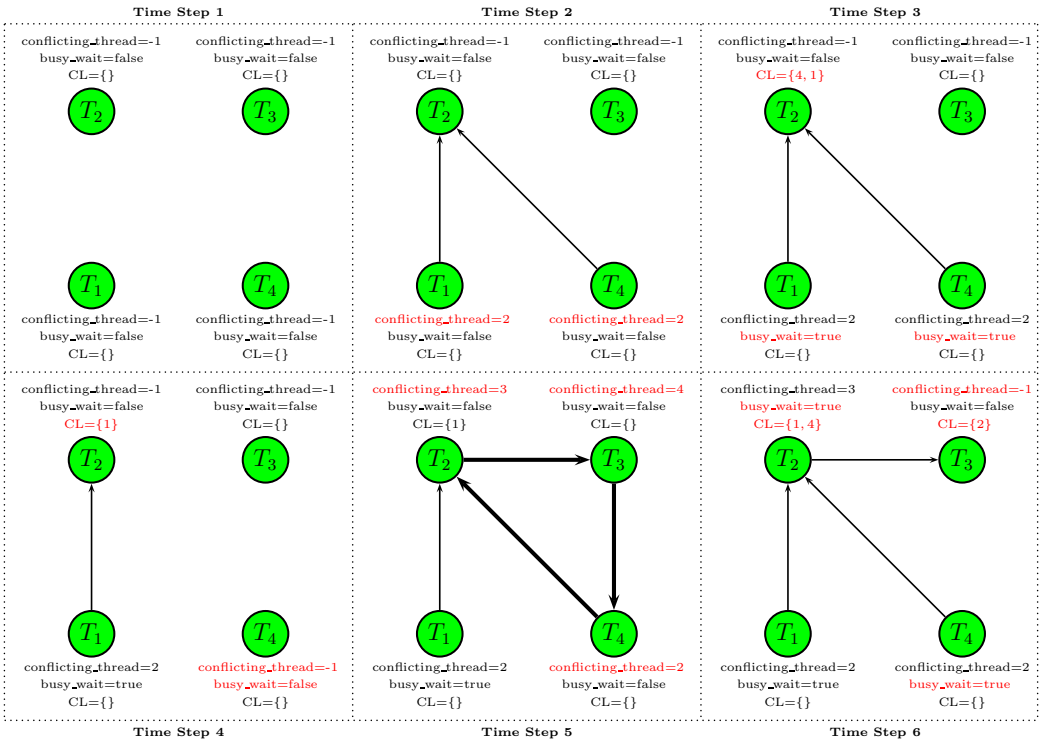


Figure 3: Illustration of the local Contention Manager (local-CM). Six Time Steps demonstrate the interaction among four threads. The contents of their Contention List (CL), the value of the conflicting_thread variable, and the value of the busy_wait flag are shown.

been set to true, it is imperative that T_i is not allowed to block, because it might be the case that the dependency of T_i forms a cycle. By not letting T_i to block, the dependency cycle "breaks". Otherwise, T_i writes its id to $\text{CL}_{\text{conflicting\_id}}$ (Lines 15-17) and loops around its busy_wait flag (Line 18).

The algorithm in Figure 2b is called by a T_i every time it completes an operation, i.e., every time T_i does not encounter a rollback. If T_i has done a lot of progress (Lines 2-5 of `Rollback_Not_Occurred`), then it awakes a thread T_j from its Contention List CL_i by setting T_j's busy_wait flag to false. Therefore, T_j escapes from the loop of Line 18 in `Rollback_Occurred` and is free to attempt the next operation.

Figure 3 illustrates possible execution scenarios for local-CM during six

Time Steps. Below, we describe in detail what might happen in each step:

- **Time Step 1:** All four threads are making progress without any rollbacks.
- **Time Step 2:** T_1 and T_4 attempted to acquire a vertex already owned by T_2. Both T_1 and T_4 call the code of Figure 2c. Their conflicting_id variables represent those exact dependencies (Line 3 of `Rollback_Occurred`).
- **Time Step 3:** T_1 and T_4 set their busy_wait flag to true (Line 12 of `Rollback_Occurred`), they write their ids to CL_2 (Lines 15-17), and they block via a busy wait (Line 18).
- **Time Step 4:** T_2 has done lots of progress and executes the Lines 6-9 of `Rollback_Not_Occurred`, awaking in this way T_4.
- **Time Step 5:** A dependency cycle is formed: $T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2$. Lines 4-14 of `Rollback_Occurred` will determine which threads block and which ones do not. Note that the mutex locking of Lines 4-5 cannot be executed at the same time by these 3 threads. Only one thread can enter its critical section (Lines 6-14) at a time.
- **Time Step 6:** Here it is shown that T_4 executed its critical section first, T_2 executed its critical section second, and T_3 was last. Therefore, T_4 and T_2 block, since the condition in Line 6 was false: their conflicting threads at that time had not set their busy_wait to true. The last thread T_3 realized that its conflicting thread T_4 has already decided to block, and therefore, T_3 returns at Line 10, without blocking.

Note that in Time Step 6, T_2 blocks without awaking the threads in its CL, and that is why both CL_2 and CL_3 are not empty. It might be tempting to instruct a thread T_i to awake all the threads in CL_i, when T_i is about to block. This could clearly expedite things. Nevertheless, such an approach could easily cause a livelock as shown in Figure 4.

Local-CM is substantially more complex than global-CM, and the deadlock-free/livelock-free guarantees are not very intuitive. The rest of this Subsection is devoted to prove that local-CM indeed can never introduce deadlocks or livelocks.

The following two Remarks follow directly from the definition of deadlock and livelock [65].

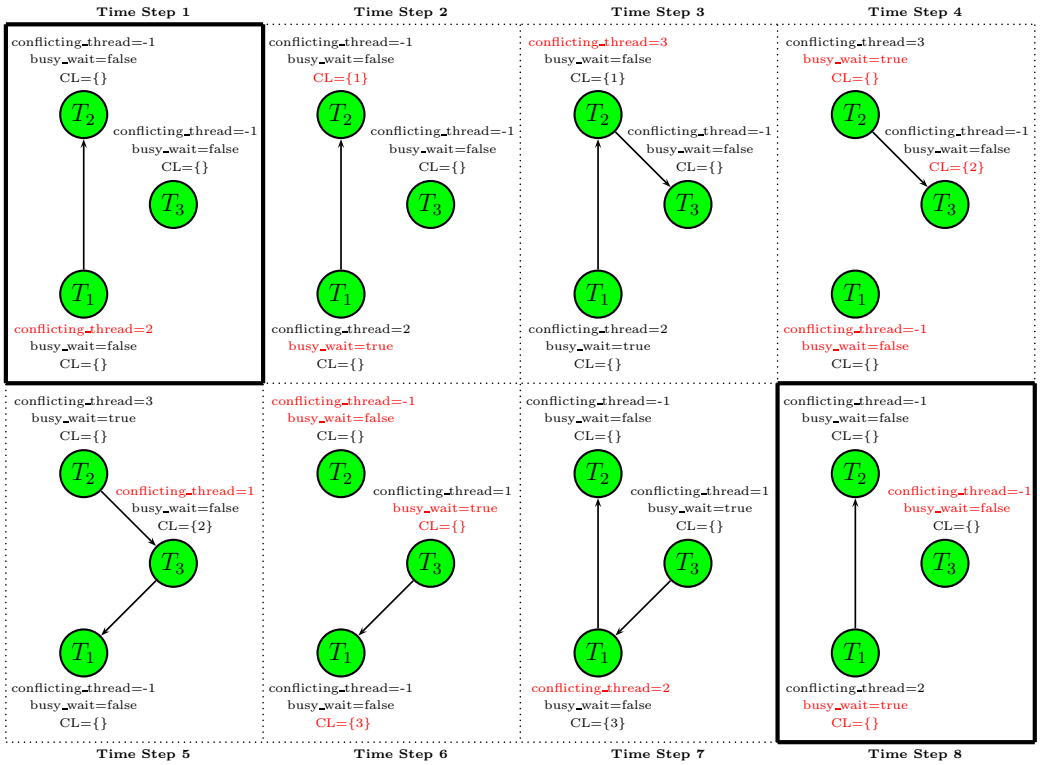


Figure 4: A thread about to busy-wait on another thread's Contention List (CL) should not awake the threads already in its own CL. Otherwise, a livelock might happen, as illustrated in this Figure. Time Step 8 leads the system to the same situation of Time Step 1: this can be taking place for an undefined period of time with none of the threads making progress.

Remark 1. *If a deadlock arises, then there has to be a dependency cycle where all the participant threads block. Only then these blocked threads will never be awaken again.*

Remark 2. *If a livelock arises, then there has to be a dependency cycle where all the participant threads are not blocked. Since all the participant threads break the cycle without making any progress, this "cycle breaking" might be happening indefinitely without completing any operations. In the only case where the rest threads of the system are blocked waiting on these participant threads' Contention Lists (or all the system's threads participate in such a cycle), then system-wide progress is indefinitely postponed.*

The next Lemmas prove that in a dependency cycle, at least one thread will block and at least one thread will not block. This is enough to prove absence of deadlocks and livelocks.

Lemma 1 (Absence of deadlocks)**.** *In a dependency cycle at least one thread will not block.*

Proof. For the sake of contradiction, assume that the threads $T_{i_1}, T_{i_2}, \ldots, T_{i_n}$ participate in a cycle, that is, $T_{i_1} \to T_{i_2} \to \cdots \to T_{i_n} \to T_{i_1}$, such that all threads block. This means that all threads evaluated Line 6 of Figure 2c to false. Therefore, since T_{i_1}'s conflicting_id is T_{i_2}, right before T_{i_1} decides to block (Line 12), T_{i_2}'s busy_wait flag was false. The same argument applies for all the pairs of consecutive threads: $\{T_{i_2}, T_{i_3}\}, \{T_{i_3}, T_{i_4}\}, \ldots, \{T_{i_n}, T_{i_1}\}$. But T_{i_n} could not have evaluated Line 6 to false, because, by our assumption, T_{i_1} had already decided to block and T_{i_1}.busy_wait had been already set to true when T_{i_n} acquired T_{i_1}'s mutex. A contradiction: T_{i_n} returns from `Rollback_Occurred` without blocking. □

Lemma 2 (Absence of livelocks)**.** *In a dependency cycle at least one thread will block.*

Proof. For the sake of contradiction, assume that the threads $T_{i_1}, T_{i_2}, \ldots, T_{i_n}$ participate in a cycle, that is, $T_{i_1} \to T_{i_2} \to \cdots \to T_{i_n} \to T_{i_1}$, such that all threads do not block. This means that all threads evaluated Line 6 of Figure 2c to true. Consider for example T_{i_1}. When T_{i_1} acquired T_{i_2}'s mutex, it evaluated Line 6 to true. That means that T_{i_2} had already acquired and released its mutex having executed Line 12: a contradiction because T_{i_2} blocks. □

5.5. Comparison

For this case study, we evaluated each CM on the CT abdominal atlas of IRCAD Laparoscopic Center (http://www.ircad.fr/) using 128 and 256 Blacklight cores (see Table 2 for its specification). The final mesh consists of about 150×10^6 tetrahedra. The single-threaded execution time on Blacklight was 1,080 seconds. See Table 1.

There are three direct sources of wasted cycles in our algorithm, and all of them are shown in Table 1:

Table 1: Comparison among Contention Managers (CM). A 150 Million element mesh is generated.

(a) 128 cores

	Aggressive-CM	Random-CM	Global-CM	Local-CM
time (secs)	n/a	64.2	23.7	19.3
rollbacks	n/a	2.48251×10^7	728,087	680,338
contention overhead (secs)	n/a	4330.9	1081.4	545.80
load balance overhead (secs)	n/a	872.48	134.62	126.22
rollback overhead (secs)	n/a	516.81	3.0	2.9
total overhead (secs)	n/a	5720.9	1219.6	675.11
speedup	n/a	16.8	45.6	56.0
livelock	yes	no	not possible	not possible
deadlock	not possible	not possible	not possible	not possible

(b) 256 cores

	Aggressive-CM	Random-CM	Global-CM	Local-CM
time (secs)	n/a	n/a	22.3	14.1
rollbackss	n/a	n/a	882,768	1.71197×10^6
contention overhead (secs)	n/a	n/a	3095.9	1377.1
load balance overhead (secs)	n/a	n/a	285.44	239.98
rollback overhead (secs)	n/a	n/a	3.6	7.6
total overhead (secs)	n/a	n/a	3385.1	1624.9
speedup	n/a	n/a	48.4	76.6
livelock	yes	yes	not possible	not possible
deadlock	not possible	not possible	not possible	not possible

- **contention overhead time**: it is the total time that threads spent busy-waiting on a Contention List (or busy-waiting for a random number of seconds as is the case of Random-CM) and accessing the Contention List (in case of Global-CM),

- **load balance overhead time**: it is the total time that threads spent busy-waiting on the Begging List waiting for more work to arrive (see Section 4) and accessing the Begging List, and
- **rollback overhead time**: it is the total time that threads had spent for the partial completion of an operation right before they decided that they had to discard the changes and roll back.

Observe that Aggressive-CM was stuck in a livelock on both 128 and 256 cores. We know for sure that these were livelocks because we found out that no tetrahedron was refined, i.e., no thread actually made any progress, in the time period of an hour.

Random-CM terminated successfully on 128 cores, but it was very slow compared to Global-CM and Local-CM. Indeed, Random-CM exhibits a large number of rollbacks that directly increases both the contention overhead and the rollback overhead. Also, since threads' progress is much slower, threads wait for extra work for much longer, a fact that also increases the load balance overhead considerably. As we have already explained above, Random-CM does not eliminate livelocks, and this is manifested on the 256 core experiment, where a livelock did occur.

On both 128 and 256 cores, Local-CM performed better. Indeed, observe that the total overhead time is approximately twice as small as Global-CM's overhead time. This is mainly due to the little contention overhead achieved by Local-CM. Since Global-CM maintains a global Contention List, a thread T_i waits for more time before it gets awaken from another thread for two reasons: (a) because there are more threads in front of T_i that need to be awaken first, and (b) because the Contention List and the number of active threads are accessed by all threads which causes longer communication latencies.

Although Local-CM is the fastest scheme, observe that it introduces higher number of rollbacks on 256 cores than Global-CM. This also justifies the increased rollback overhead (see Table 1b). In other words, fewer rollbacks do not always imply faster executions, a fact that renders the optimization of our application a challenging task. This result can be explained by the following observation: the number of rollbacks (and subsequently, the rollback overhead) and the contention overhead constitute a tradeoff. The more a thread waits in a Contention List, the more its contention overhead is, but the fewer the rollbacks it encounters are, since it does not attempt to perform any operation. Conversely, the less a thread waits in a Contention List, the less its contention overhead is, but since it is given more chances

Table 2: The specifications of the cc-NUMA machines we used.

	Model	cores per socket	sockets per blade	blades	memory per socket	max hops
Blacklight	Intel Xeon X7560	8	2	128	64GB	5
CRTC	Intel Xeon X5690	6	2	1	48GB	0

to apply an operation, it might encounter more rollbacks. Nevertheless, Table 1 suggests that Local-CM does a very good job balancing this tradeoff on runtime.

Although there are other elaborate and hybrid contention techniques [61, 62], none of them guarantees the absence of livelocks. Therefore, we chose Local-CM because of its efficiency and correctness.

6. Performance

In this Section, we describe a load balancing optimization and present the strong and weak scaling performance on Blacklight. See Table 2 for its specifications.

6.1. Hierarchical Work Stealing (HWS)

In order to further decrease the communication overhead associated with remote memory accesses, we implemented a *Hierarchical Work Stealing* scheme (HWS) by taking advantage of the cc-NUMA architecture.

We re-organized the Begging List into three levels: BL1, BL2, and BL3. Threads of a single socket that run out of work place themselves into the first level begging list BL1 which is shared among threads of a single socket. If the thread realizes that all the other socket threads wait on BL1, it skips BL1, and places itself to BL2, which is shared among threads of a single blade. Similarly, if the thread realizes that BL2 already accommodates a thread from the other socket in its blade, it asks work by placing itself into the last level begging list BL3. When a thread completes an operation and is about to send extra work to an idle thread, it gives priority to BL1 threads first, then to BL2, and lastly to BL3 threads. In other words, BL1 is shared among the threads of a single socket and is able to accommodate up to $number\_of\_threads\_per\_socket - 1$ idle threads (in Blacklight, that is 7 threads). BL2 is shared among the sockets of a single blade and is able to accommodate up to $number\_of\_sockets\_per\_blade - 1$ idle threads (in Blacklight, that is 1 thread). Lastly, BL3 is shared among all the allocated

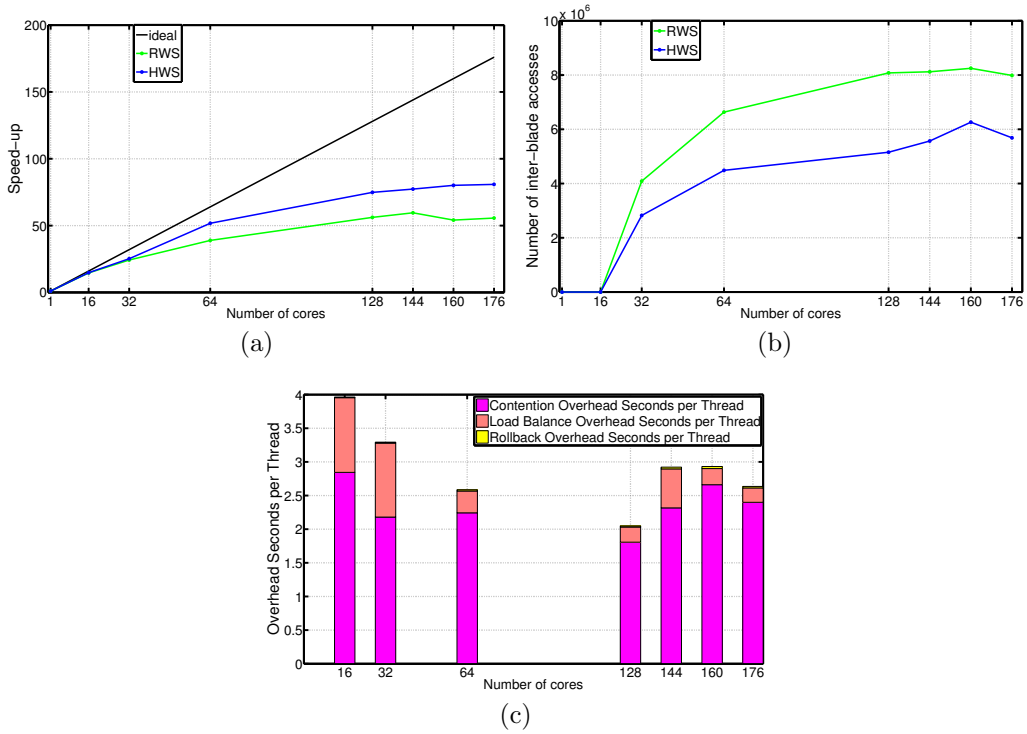


Figure 5: Strong scaling performance achieved by the classic Random Work Stealing (RWS) and Hierarchical Work Stealing (HWS). (a)-(b) Comparison between RWS and HWS on speed-up ($\frac{\mathbf{time_1}}{\mathbf{time_{\#Threads}}}$) and on the number of inter-blade accesses. (c) Breakdown of the overhead time for HWS.

blades and can accommodate at most one thread per blade. In this way, an idle thread T_i tends to take work first from threads inside its socket. If there is none, T_i takes work from a thread of the other socket inside its blade, if any. Finally, if all the other threads inside T_i's blade are idling for extra work, T_i places its id to BL3, asking work from a thread of another blade.

6.2. Strong Scaling Results

Figure 5 shows the strong scaling experiment demonstrating both the Random Work Stealing (RWS) load balance and the Hierarchical Work Stealing (HWS). The input image we used is the CT abdominal atlas obtained from IRCAD Laparoscopic Center. Information about this input image is

Table 3: Information about the three input images used for the scaling results of Section 6 and the single-threaded performace comparison of Section 7.

	voxels	spacing (mm^3)	tissues	download from
abdominal atlas	$512 \times 512 \times 219$	$0.96 \times 0.96 \times 2.4$	23	http://www.ircad.fr/softwares/3Dircadb/3Dircadb2/3Dircadb2.2.zip
knee atlas	$512 \times 512 \times 119$	$0.27 \times 0.27 \times 1$	49	http://www.spl.harvard.edu/publications/item/view/1953
head-neck atlas	$255 \times 255 \times 229$	$0.97 \times 0.97 \times 1.4$	60	http://www.spl.harvard.edu/publications/item/view/2271

shown in Table 3. The final mesh generated consists of 124×10^6 elements. On a single Blacklight core, the execution time was 1100 seconds.

Observe that the speed-up of RWS deteriorates by a lot for more than 64 cores (see the green line in Figure 5a). In contrast, HWS manages to achieve a (slight) improvement even on 176 cores. This could be attributed to the fact that the number of inter-blade (i.e., remote) accesses are greatly reduced by HWS (see Figure 5b), since begging threads are more likely to get poor elements created by threads of their own socket and blade first. Clearly, this reduces the communication involved when a thread reads memory residing in a remote memory bank. Indeed, on 176 cores, 98.9% of all the number of times threads asked for work, they received it from a thread of their own blade, yielding a 28.8% reduction in inter-blade accesses, as Figure 5b shows.

Figure 5c shows the breakdown of the overhead time per thread for HWS across runs. Note that since this is a strong scaling case study, the ideal behavior is a linear increase of the height of the bars with the respect to the number of threads. Observe, however, that the overhead time per thread is always below the overhead time measured on 16 threads. This means that Local-CM and the Hierarchical Work Stealing method (HWS) are able to serve threads fast and tolerate congestion efficiently on runtime.

6.3. Weak Scaling Results

In this section, we present the weak scaling performance of PI2M on two inputs, the information of which is presented in Table 3. The first is the same CT abdominal atlas already used in the previous strong scaling Section. The second input image is the knee atlas obtained from Brigham & Women's Hospital Surgical Planning Laboratory [68]. Other inputs exhibit very similar results on comparable mesh sizes.

We measure the number of tetrahedra created per second across the runs. Specifically, let us define with $\text{Elements}(n)$ and $\text{Time}(n)$, the number of elements created and the time elapsed, when n threads are employed. Then,

Table 4: Weak scaling performance. Across runs, the number of elements per thread remains approximately constant.

(a) abdominal atlas

#Threads	1	16	32	64	128	144	160	176
#Elements	1.07E+07	1.72E+08	3.49E+08	7.44E+08	1.32E+09	1.51E+09	1.67E+09	1.85E+09
Time (secs)	90.37	80.03	87.50	99.23	93.00	103.26	150.03	181.10
Elements per second	1.18E+05	2.15E+06	3.99E+06	7.50E+06	1.42E+07	**1.46E+07**	1.11E+07	1.02E+07
Speedup	1.00	18.19	33.71	63.33	119.56	123.67	94.10	86.36
Efficiency	1.00	1.14	1.05	0.99	**0.93**	0.86	0.59	0.49
Overhead secs per thread	0.90	1.60	2.41	2.98	4.42	4.76	8.74	10.55

(b) knee-atlas

#Threads	1	16	32	64	128	144	160	176
#Elements	1.06E+07	1.66E+08	3.70E+08	8.06E+08	1.31E+09	1.58E+09	1.70E+09	1.91E+09
Time (secs)	87.26	80.67	98.36	110.72	97.79	110.00	167.08	190.00
Elements per second	1.22E+05	2.05E+06	3.76E+06	7.28E+06	1.34E+07	**1.43E+07**	1.02E+07	1.01E+07
Speedup	1.00	16.89	30.92	59.90	110.61	117.92	83.77	82.81
Efficiency	1.00	1.06	0.97	0.94	**0.86**	0.82	0.52	0.47
Overhead secs per thread	0.87	1.46	2.77	3.41	5.47	6.58	8.90	11.07

the speedup is defined as $\frac{\text{Elements}(n)\times\text{Time}(1)}{\text{Time}(n)\times\text{Elements}(1)}$. With n threads, a perfect speedup would be equal to n [69].

We can directly control the size of the problem (i.e., the number of generated tetrahedra) via the parameter δ (see Section 3). This parameter sets an upper limit on the volume of the tetrahedra generated. With a simple volume argument, we can show that a decrease of δ by a factor of x results in an x^3 times increase of the mesh size, approximately.

See Table 4. Each reported Time is computed as the average among three runs. Although the standard deviation for up to 128 cores is practically zero on both inputs, the same does not apply for higher core counts. Indeed, the standard deviation on the 144-, 160-, and 176-core executions is about 10, 15, and 29 seconds respectively, for both inputs. We attribute this behavior to the fact that in those experiments, the network switches responsible for the

cache coherency were close to the root of the fat-tree topology and therefore, they were shared among many more users, affecting in this way the timings of our application considerably. (Note that the increased bandwidth of the upper level switches does not alleviate this problem, since the bottleneck of our application is latency.) This conjecture agrees with the fact that the the maximum number of hops on the experiments for up to 128 cores was 3, while for 144, 160 and 176 cores, this number became 5.

Nevertheless, observe the excellent speedups for up to 128 threads. On 144 cores, we achieve an unprecedented efficiency of more than 82%, and a rate of more than 14.3 Million Elements per second for both inputs. It is worth mentioning that CGAL [9], the fastest sequential publicly available Isosurface-based mesh generation tool, on the same CT abdominal (http://www.ircad.fr/softwares/3Dircadb/3Dircadb2/3Dircadb2.2.zip) image input, is 81% slower than our single-threaded performance. Indeed, CGAL took 548.21 seconds to generate a similarly-sized mesh (1.00×10^7 tetrahedra) with comparable quality and fidelity to ours (see Section 7 for a more thorough comparison case study). Thus, compared to CGAL, the speedup we achieve on 144 cores is 751.25.

Observe, however, that our performance deteriorates beyond this core count. We claim that the main reason of this degradation is not the overhead cycles spent on rollbacks, contention lists, and begging lists (see Section 5.5), but the congested network responsible for the communication. Below, we support our claim.

First of all, notice that the total overhead time per thread increases. Since this is a weak scaling case study, the best that can happen is a constant number of overhead seconds per thread. But this is not happening. The reason is that in the beginning of refinement, the mesh is practically empty: only the six tetrahedra needed to fill the virtual box are present (see Figure 1). Therefore, during the early stages of refinement, the problem does not behave as a weak scaling case study, but as a strong scaling one: more threads, but in fact the same size, which renders our application a very challenging problem. See Figure 6 for an illustration of the 176-core experiment of Table 4a. X-axis shows the wall-time clock of the execution. The Y-axis shows the total number of seconds that threads have spent on useless computation (i.e., rollback, contention, and load balance overhead, see Section 5.5) so far, cumulatively. The more straight the lines are, the more useful work the threads perform. Rapidly growing lines imply lack of parallelism and intense contention. Observe that in the first 14 seconds of refinement (Phase_1), there

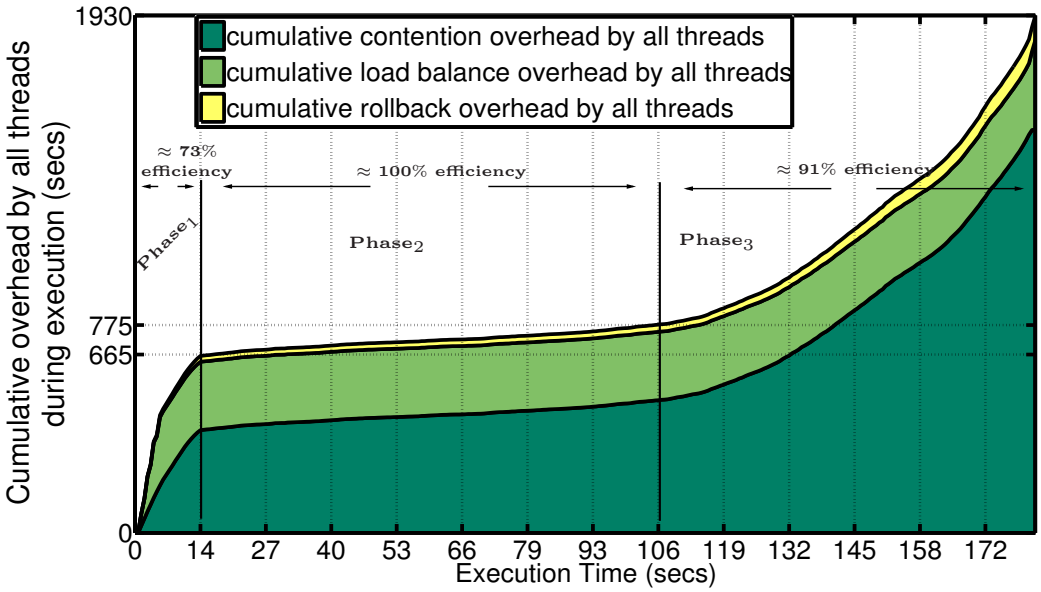


Figure 6: Overhead time breakdown with respect to the wall time for the experiment on 176 cores of Table 4a. A pair (x, y) tells us that up to the xth second of execution, threads have not been doing useful work so far for y seconds all together.

is high contention and severe load imbalance. Nevertheless, even in this case, $\frac{176\times14-665}{176\times14} \approx 73\%$ of the time, all 176 threads were doing useful work, i.e., the threads were working on their full capacity.

However, this overhead time increase cannot explain the performance deterioration. See for example the numbers on 176 threads of Table 4a. 176 threads run for 181.10s each, and, on average, they do useless work for 10.55s each. In other words, if there were no rollbacks, no contention list overhead, and no load balancing overhead, the execution time would have to be 181.10s-10.55s =170.55s. 170.55s, however, is far from the ideal 90.37s (that the first column with 1 thread shows) by 170.55s-90.37=80.18s. Therefore, while rollbacks, contention management, and load balancing introduce a merely 10.55s overhead, the real bottleneck is the 80.18s overhead spent on memory (often remote) loads/stores. Indeed, since the problem size increases linearly with respect to the number of threads, either the communication traffic per network switch increases across runs, or it goes through a higher number of hops (each of which adds a 2,000 cycle latency penalty [70]), or both. It

Table 5: Hyper-threaded execution of the case study shown in Table 4a. The columns of the Speedup, TLB misses, LLC misses, and Resource stall cycles reported here are relative to the non hyper-threaded execution of Table 4a on the same number of cores.

#Cores (2 threads per core)	1	16	32	64	128	144	160	176
#Elements	1.07E+07	1.72E+08	3.49E+08	7.44E+08	1.32E+09	1.51E+09	1.67E+09	1.85E+09
Time (secs)	58.03	55.98	61.57	67.28	240.36	342.91	436.72	480.83
Elements per second	1.84E+05	3.08E+06	5.67E+06	1.11E+07	5.48E+06	4.41E+06	3.83E+06	3.85E+06
Speedup	1.56	1.43	1.42	**1.47**	0.39	0.30	0.34	0.38
Overhead secs per thread	1.16	2.55	3.64	4.55	39.60	111.18	91.85	143.37
TLB misses increase per thread	-13.20%	-16.79%	-18.21%	-16.63%	-22.68%	-28.87%	-34.38%	-34.49%
LLC misses increase per thread	81.72%	-39.72%	-34.81%	-46.63%	-67.71%	-58.01%	-72.98%	-63.08%
Resource stall cycles increase per thread	-46.73%	-50.24%	-47.94%	-48.12%	-38.38%	-37.18%	-49.44%	-43.26%

seems that after 144 cores, this pressure on the switches slows performance down. A hybrid approach [14] able to scale for larger network hierarchies is left for future work.

6.3.1. Hyper-threading

Table 5 shows the performance achieved by the hyper-threaded version of our code. For this case study, we used the same input and parameters as the ones used in the experiment shown in Table 4a. The only difference is that now there are twice as many threads as there were in Table 4a.

Since the hardware threads share the TLB, the cache hierarchy, and the pipeline, we report the impact of hyper-threading on TLB misses, Last Level Cache (LLC) misses, and Resource stall cycles. Specifically, we report the increase of those counters relatively to the non hyper-threaded experiment of Table 4a. The reported Speedup is also relative to the non hyper-threaded experiment.

The last three rows of Table 5 suggest that the hyper-threaded version utilized the core resources more efficiently. Surprisingly enough, the TLB and LLC misses actually decrease (notice the negative sign in front of the percentages) when two hardware threads are launched per core. Also, as expected,

the pipeline in the hyper-threaded version is busier executing micro-ops, as the decrease of resource stall cycles suggest.

Although hyper-threading achieves a better utilization of the TLB, LLC, and pipeline, there is a considerable slowdown after 64 cores (i.e., 128 hardware threads). Observe that hyper-threading expedited the execution for up to 64 cores. Indeed, the hyper-threaded version is 47% faster on 64 cores compared to the non hyper-threaded version. Beyond this point, however, there is a considerable slowdown. This slowdown cannot be explained by merely the increase in the number of overhead seconds.

See for example the overhead secs per thread on 176 cores in Table 5. It is indeed 13 times higher than its non hyper-threaded counterpart; this is, however, expected because the size of the problem is the same but now we use twice as many hardware threads as before. If we subtract the overhead time of the hyper-threaded version on 176 cores, we get that for $480.83s - 143.37s = 337.46s$, all hardware threads were doing useful work. But this is still way longer than the $181.10s - 10.55s = 170.55s$ useful seconds of the non hyper-threaded execution (see Table 4a).

We attribute this behavior to the increased communication traffic caused not by the increased problem size (as was mostly the case in the non hyper-threaded version), but by the increased number of "senders" and "receivers". That is, even though the problem size is the same, the hyper-threaded version utilizes more threads. This means that at a given moment, there will be more packages (originated by the more than before threads) in the switches waiting to be routed than before. This phenomenon increases the communication latency. It seems that the network cannot handle this pressure for more than 64 cores, or equivalently, 128 hardware threads. Note that this agrees with the fact that in the non hyper-threaded version, the slowdown occurred on more than 128 cores, which is again 128 threads (see Table 4).

7. Single-threaded evaluation

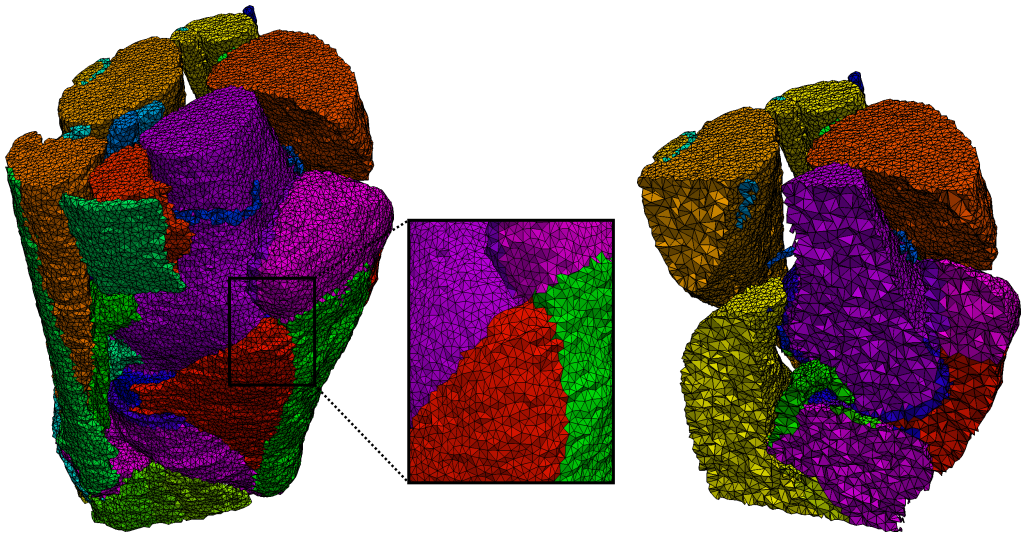
Although PI2M introduces extra overhead due to locking, synchronization, contention management bookkeeping (see Section 5), and hierarchical load balance (see Section 6.1), in this Section we show that the single-threaded performance of our method (PI2M) is better than the performance of CGAL [9] and TetGen [10], the state-of-the-art sequential open source mesh generation tools. Moreover, PI2M has comparable quality with CGAL and much better quality than TetGen. PI2M, CGAL, and TetGen are very

robust Delaunay methods, since they all use exact predicates. Specifically, PI2M adopts the exact predicates as implemented in CGAL [9, 71].

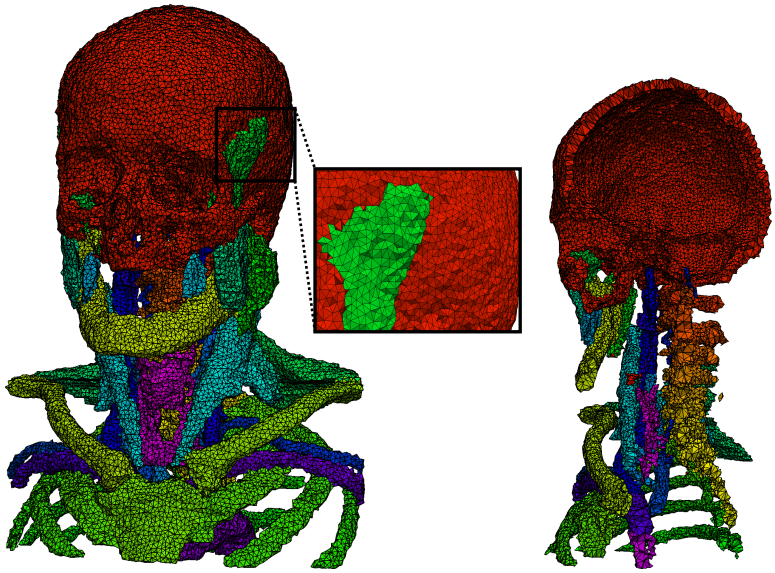
It should be mentioned that although CGAL is able to operate directly on segmented multi-tissue images (i.e., it is an Isosurface-based method), TetGen is a PLC-based method (see Section 2). That is, TetGen's inputs are triangulated domains that separate the different tissues. For this reason, we pass to TetGen the triangulated iso-surfaces as recovered by our method, and then let TetGen to fill the underlying volume.

We ran PI2M, CGAL, and TetGen on two different multi-tissue 3D input images obtained from Brigham & Women's Hospital Surgical Planning Laboratory (http://www.spl.harvard.edu/). The first is the MR knee-atlas [68] used in the previous Section and the second is a CT head-neck atlas [72]. Information about these two inputs is displayed in Table 3. The resulting output meshes generated by our method PI2M are illustrated in Figure 7. We should emphasize that we do not perform any smoothing as a post-processing step, since smoothing tends to deteriorate quality. In fact, in our previous work [73, 74], we show that quality is of great importance in the speed and accuracy of certain applications, such as non-rigid brain registration, and it should not be compromised. Nevertheless, mesh boundary smoothing is desirable for CFD simulations, such as respiratory airway modeling [37–39]. The extension of our framework to support the computationally expensive step of volume-conserving smoothing [37] and scale invariance [38] in parallel is left for future work.

For fair comparison, we also show the resulting output meshes generated by CGAL and TetGen in Figure 8 and Figure 9, respectively. A close investigation of the meshes generated by TetGen (Figure 9) reveals that there are fewer labels than the labels recovered by PI2M and CGAL. In other words, the labels of TetGen do not correspond to the same labels of PI2M or CGAL. This is attributable to the way TetGen groups elements together [10] for visualization purposes. As mentioned earlier, the input PLC for TetGen is the set of the triangulated isosurfaces as recovered by PI2M. This PLC divides the domain into the subdomains that constitute the different tissues. In order for the elements of a subdomain A to be colored by a different label than the elements of a subdomain B, the user needs to specify two seed points p_A and p_B, such that p_A lies strictly in the interior of A and p_B lies strictly in the interior of B. A straightforward (perhaps not the best) way to compute these seeds is to traverse the input image and to assign a seed point per tissue. The unfortunate discrepancy with such an approach is that seeds

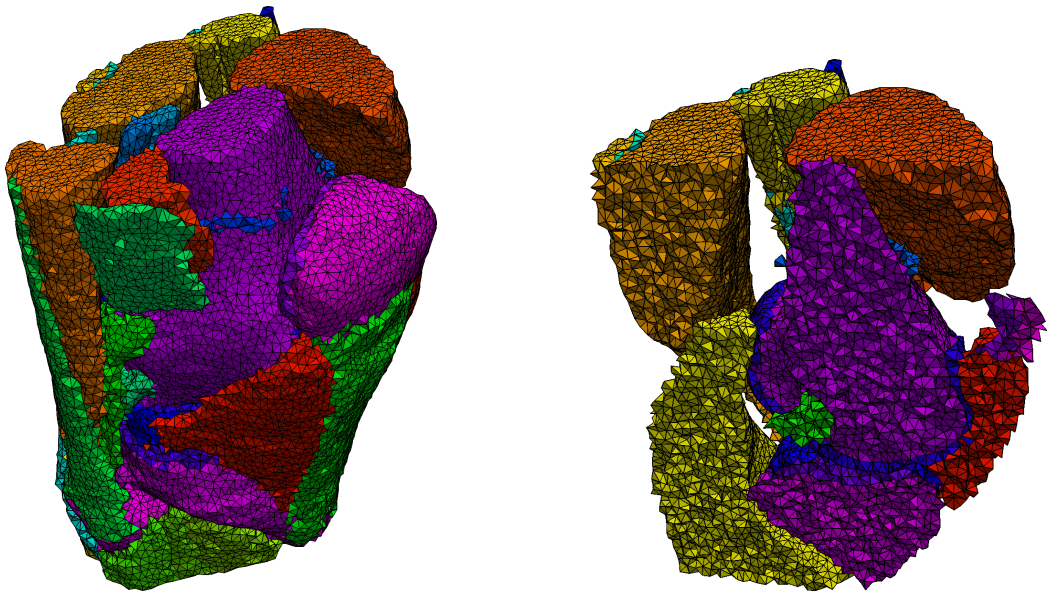


(a) The 439,458 element mesh generated for the MR knee atlas.

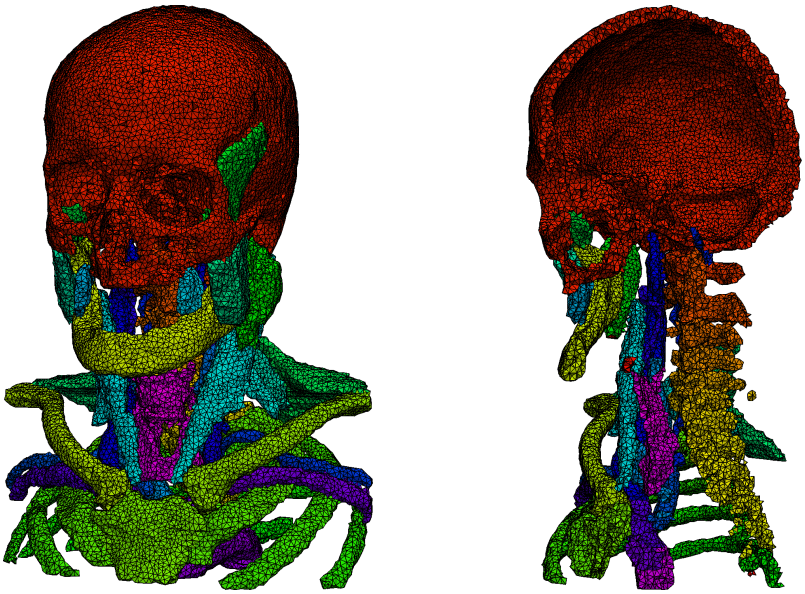


(b) The 993,583 element mesh generated for the CT head-neck atlas.

Figure 7: Output meshes generated by PI2M on the MR knee atlas and on the CT head-neck atlas.

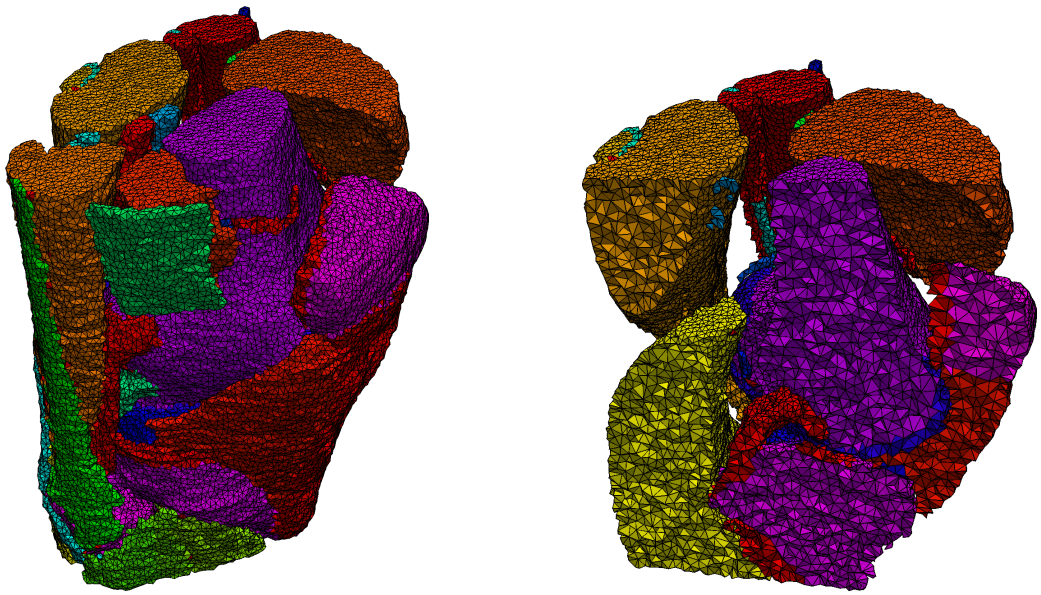


(a) The 436,749 element mesh generated for the MR knee atlas.

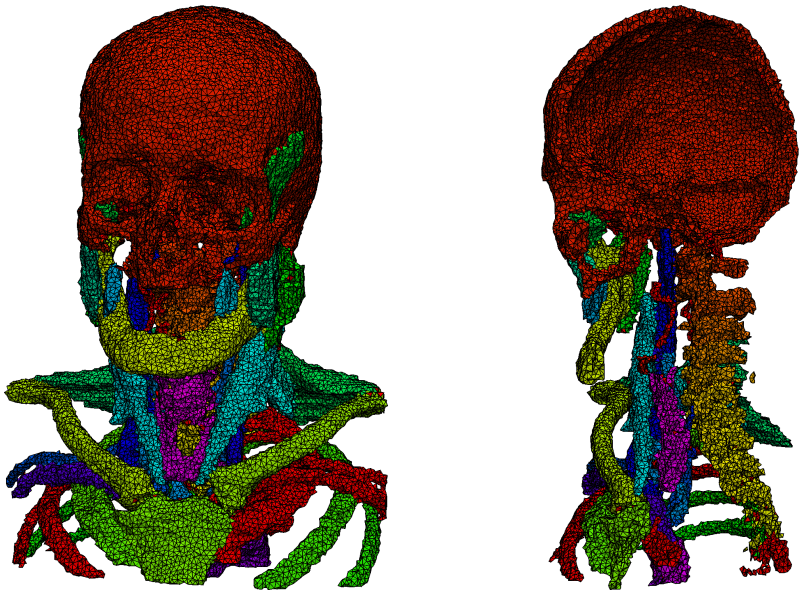


(b) The 991,509 element mesh generated for the CT head-neck atlas.

Figure 8: Output meshes generated by CGAL on the MR knee atlas and on the CT head-neck atlas.



(a) The 434,095 element mesh generated for the MR knee atlas.



(b) The 990,446 element mesh generated for the CT head-neck atlas.

Figure 9: Output meshes generated by TetGen on the MR knee atlas and on the CT head-neck atlas.

Table 6: Statistics regarding the single-threaded performance and the quality/fidelity achieved by PI2M and CGAL. PI2M includes the extra overhead introduced by synchronization, contention management, and load balancing to support the (potential) presence of other threads.

	knee atlas			head-neck atlas		
	PI2M	CGAL	TetGen	PI2M	CGAL	TetGen
#tetrahedra / seconds	67,609	40,069	98,658	96,464	29,077	61,903
time	6.5 secs	10.9 secs	4.4 secs	10.3 secs	34.1 secs	16.0 secs
#tetrahedra	439,458	436,749	434,095	993,583	991,509	990,446
max radius-edge ratio	2	4.4	18.6	2	11.2	93.4
smallest boundary planar angle	17.4°	24.6°	18.0°	15.8°	2.4°	15.3°
(min, max) dihedral angles	(4.6°, 170.1°)	(2.5°, 176.3°)	(2.9°, 173.0°)	(4.5°, 170.2°)	(4.1°, 173.9°)	(0.4°, 172.0°)
Hausdorff distance	10.7 mm	10.3 mm	-	15.3 mm	15.2 mm	-

might not lie in the intended PLC subdomains, simply because the recovered isosurfaces (that form the PLC) represent the actual tissue geometry within a tolerance (see Theorem 1). This problem affects only the visualization of TetGen meshes and it becomes more acute in our case, because there are many tissues that have very little volume, a reality that renders the computation of the appropriate seed points less accurate and robust in general. This fact alters the coloring of the TetGen meshes and this is the reason TetGen coloring does not completely agree with the coloring of the meshes generated by PI2M and CGAL.

Table 6 shows timings and quality statistics for PI2M, CGAL, and TetGen. We used CRTC (see Table 2 for its specifications) for this case study. The timings reported account for everything but for disk IO operations. The execution time reported for PI2M incorporates the 1.9 seconds and 1.2 seconds time interval needed for the computation of the Euclidean distance transform (see Section 3) for the knee atlas and the head-neck atlas, respectively.

We set the sizing parameters of CGAL and TetGen to values that produced meshes of similar size to ours, since generally, meshes with more elements exhibit better quality and fidelity. We access the achieved quality of these methods in terms of radius-edge ratio and dihedral angles. Those metrics are of great important to us, because they are shown to improve the speed and robustness of medical application solvers dealing with isotropic materials [3, 4, 73–75]. Ideally, the radius-edge ratio should be low, the minimum dihedral angle should be large, and the maximum dihedral angle should be low. We also report the smallest boundary planar angles. This

measures the quality of the mesh boundary. Large smallest boundary planar angles imply better boundary quality.

PI2M, CGAL, and TetGen allow users to specify the target radius-edge ratio. Apart from TetGen, these methods also allow users to specify the target boundary planar angles. We set the corresponding parameters accordingly, so that the maximum radius-edge ratio is 2 (for PI2M, CGAL, and TetGen), and the smallest boundary planar angle is more than 30° (for PI2M and CGAL only, since TetGen does not give this parameter).

Fidelity measures how well the mesh boundary represents the iso-surfaces. We access the fidelity achieved by these methods in terms of the symmetric (double-sided) Hausdorff distance. A low Hausdorff distance implies a good representation. Notice that we do not report the Hausdorff distance for TetGen, since the triangular mesh that represents the iso-surfaces is given to it as an input. For the input images we used for Table 6, the Hausdorff distances achieved by both PI2M and CGAL are far from ideal. This happens because the values chosen for the sizing parameters at this comparison did not recover isolated clusters of voxels which seem to be artifacts of the segmentation anyway. Nevertheless, Theorem 1 guarantees (both in theory and in practice) that if the sample is very dense, then the Hasdorff distance approaches to zero. The goal of this Section is not to generate meshes of high fidelity, but to demonstrate the effectiveness of PI2M by comparing PI2M with the state of the art open source meshers.

We access the speed of the methods above by comparing the rate of generated tetrahedra per second. Note that since our method not only inserts but also removes points from the mesh (thus reducing the number of mesh elements), a perhaps fairer way to access speed is to compare the rate of performed operations per second. Nevertheless, we do not report this metric for two reasons. First, a high rate of operations does not always imply a high rate of generated tetrahedra. The later, however, is the only thing that matters, since comparing the quality/fidelity achieved by meshes of very different mesh sizes makes no sense. Second, the number of removals performed by PI2M accounts for only 2% over the total number of operations. Thus, the rate of generated tetrahedra is very close the rate of operations per second; indeed, we experimentally found out that those two rates are practically the same.

Observe that the PI2M and CGAL generate meshes of similar dihedral angles, and fidelity, but our method is much faster. Indeed, the rate of the single-threaded PI2M is 68.7% higher than CGAL on the knee atlas and more

than 3 times higher on the head-neck atlas. Also note that both PI2M and CGAL prove that the smallest boundary planar angles are more than 30° and that radius-edge ratio is less than 2 [7]. Due to numerical errors, however, these bounds might be smaller in practice than what theory suggests. Nevertheless, observe that PI2M yields much better boundary planar angles and radius-edge ratio than CGAL on the head-neck atlas.

TetGen is faster than PI2M only on the knee atlas by a couple of seconds. For larger meshes (as is the case with the head-neck atlas), TetGen is slower. Indeed, for small meshes, the computation of the Euclidean Distance Transform (EDT) accounts for a considerable percentage over the total execution time, a fact that slows down the overall execution time by a lot. For example, the actual meshing time on the knee atlas was just 4.6 secs, very close to TetGen's time and rate. Another notable observation is that our method generates meshes with much better dihedral angles and radius-edge ratio than TetGen. The achieved boundary planar angles are similar simply because the PLC that is given to TetGen was in fact the triangular boundary mesh of PI2M.

8. Discussion, Conclusions, and Future Work

In this paper, we present PI2M: the first parallel Image-to-Mesh (PI2M) Conversion Isosurface-based algorithm and its implementation. Starting directly from a multi-label segmented 3D image, it is able to recover and mesh both the isosurface $\partial\mathcal{O}$ with geometric and topological guarantees (see Theorem 1) and the underlying volume $\mathcal{O}$ with quality elements.

This work is different from parallel Triangulators [40–43], since parallel mesh generation and refinement focuses on the quality of elements (tetrahedra and facets) and the conformal representation of the tissues' boundaries/isosurfaces by computing on demand the appropriate points for insertion or deletion. Parallel Triangulators tessellate only the convex hull of a set of points.

Our tighly-coupled method greatly reduces the number of rollbacks and scales up to a much higher core count, compared to the tightly-coupled method our group developed in the past [33]. The data decomposition method [45] does not support Delaunay removals, a technique that it is shown to be effective in the sequential mesh generation literature [7, 8]. The extension of partially-coupled [44] and decoupled [34] methods to 3D is a very difficult task, since Delaunay-admissible 3D medial decomposition

is an unsolved problem. On the contrary, our method does not rely on any domain decomposition, and could be extended to arbitrary dimensions as well. Indeed, we plan to extend PI2M to 4 dimensions and generate space-time elements (needed for spatio-temporal simulations [76, 77]) in parallel, thus, exploiting parallelism in the fourth dimension. As future work, we also leave the mesh boundary smoothing required for CFD simulations, such as respiratory airway modeling [37–39].

Our code is highly optimized through carefully designed contention managers, and load balancers which take advantage of NUMA architectures. Our Global Contention Manager (Global-CM) and Local Contention Manager (Local-CM) provably eliminate deadlocks and livelocks. They achieve a speedup even on 256 cores, when other traditional contention managers, found in the mesh generation literature, fail to terminate. Local-CM also reduced the number of overhead cycles by a factor of 2 compared to the Global-CM on 256 cores, improving energy-efficiency by avoiding energy waste because of rollbacks. Lastly, our Hierarchical Work Stealing load balancer (HWS) sped up the execution by a factor of 1.45 on 176 cores, as a result of a 22.8% remote accesses reduction.

All in all, PI2M achieves a strong scaling efficiency of more than 82% on 64 cores. It also achieves a weak scaling efficiency of more than 82% on up to 144 cores. We are not aware of any 3D parallel Delaunay mesh refinement algorithm achieving such a performance.

It is worth noting that PI2M exhibits excellent single-threaded performance. Despite the extra overhead associated with synchronization, contention management, and load balancing, PI2M generates meshes 40% faster than CGAL and with similar quality. Moreover, PI2M achieves better quality than TetGen, and it is also faster than TetGen for large mesh sizes.

Recall that in our method, threads spend time idling on the contention and load balancing lists. And this is necessary in our algorithm for correctness and performance efficiency. This fact offers great opportunities to control the power consumption, or alternatively, to maximize the $\frac{\text{Elements}}{\text{second} \times \text{Watt}}$ ratio. Since idling is not the time critical component in our algorithm, the CPU frequency could be decreased during such an idling. Nevertheless, the appropriate frequency drop, the amount of idling, and performance is a trade-off, and its investigation is left as future work.

As already explained, for core counts higher than 144, weak scaling performance deteriorates because communication traffic (per switch) is more intense and passes through a larger number of hops. In the future, we plan

to increase scalability by employing a hierarchically layered (distributed and shared memory) implementation design [14] and combine this tightly-coupled method with the decoupled and partially coupled methods we developed in the past, exploring in this way different levels of concurrency.

Acknowledgments

The authors are deeply grateful to PSC's system group for its priceless and prompt support. Special thanks to the reviewers for their constructive suggestions and to Dimitris Nikolopoulos, Andrey Chernikov, and Andriy Kot for their instructive comments and insightful discussions. 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] N. Archip, O. Clatz, A. Fedorov, A. Kot, S. Whalen, D. Kacher, N. Chrisochoides, F. Jolesz, A. Golby, P. Black, S. K. Warfield, Non-rigid alignment of preoperative MRI, fMRI, DT-MRI, with intra-operative MRI for enchanced visualization and navigation in image-guided neurosurgery, Neuroimage 35 (2007) 609–624.

[2] Y. Liu, C. Yao, L. Zhou, N. Chrisochoides, A point based non-rigid registration for tumor resection using iMRI, in: IEEE International Symposium on Biomedical Imaging: From Nano to Macro, IEEE Press, 2010, pp. 1217–1220.

[3] J. R. Shewchuk, What is a Good Linear Element? - Interpolation, Conditioning, and Quality Measures, in: Proceedings of the 11^{th} International Meshing Roundtable, Sandia National Laboratories, 2002, pp. 115–126.

[4] O. Goksel, S. E. Salcudean, High-quality model generation for finite element simulation of tissue deformation, in: 12^{th} International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI), MICCAI '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 248–256.

[5] P.-L. George, H. Borouchaki, Delaunay triangulation and meshing, Application to finite elements, HERMES, 1998.

[6] B. M. Klingner, J. R. Shewchuk, Aggressive tetrahedral mesh improvement, in: Proceedings of the International Meshing Roundtable, Springer, 2007, pp. 3–23.

[7] P. Foteinos, A. Chernikov, N. Chrisochoides, Guaranteed Quality Tetrahedral Delaunay Meshing for Medical Images, in: Proceedings of the 7th International Symposium on Voronoi Diagrams in Science and Engineering, IEEE Computer Society, 2010, pp. 215–223.

[8] P. Foteinos, N. Chrisochoides, High-quality multi-tissue mesh generation for finite element analysis, in: Y. J. Zhang (Ed.), Image-Based Geometric Modeling and Mesh Generation, volume 3 of *Lecture Notes in Computational Vision and Biomechanics*, Springer Netherlands, 2013, pp. 159–169.

[9] CGAL, Computational Geometry Algorithms Library, http://www.cgal.org, v4.0.

[10] H. Si, TetGen, A Quality Tetrahedral Mesh Generator and a 3D Delaunay Triangulator, http://tetgen.berlios.de/, v1.4.3.

[11] A. Bowyer, Computing Dirichlet tesselations, Computer Journal 24 (1981) 162–166.

[12] D. F. Watson, Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes, Computer Journal 24 (1981) 167–172.

[13] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, A view of the parallel computing landscape, Commun. ACM 52 (2009) 56–67.

[14] N. Chrisochoides, A. Chernikov, A. Fedorov, A. Kot, L. Linardakis, P. Foteinos, Towards exascale parallel Delaunay mesh generation, in: International Meshing Roundtable, 18, Springer Berlin Heidelberg, Salt Lake City, Utah, 2009, pp. 319–336.

[15] R. A. Kendall, M. Sosonkina, W. D. Gropp, R. W. Numrich, T. Sterling, Parallel programming models applicable to cluster computing and beyond, in: A. Bruaset, A. Tveito (Eds.), Numerical Solution of Partial Differential Equations on Parallel Computers, Springer, 2005, pp. 3–55.

[16] P. Foteinos, D. Feng, A. Chernikov, N. Chrisochoides, Multi-layered unstructured mesh generation, in: Proceedings of the 27th international ACM conference on International conference on supercomputing, ICS '13, ACM, New York, NY, USA, 2013, pp. 471–472.

[17] J. R. Shewchuk, Tetrahedral mesh generation by Delaunay refinement, in: Proceedings of the 14$^{\text{th}}$ ACM Symposium on Computational Geometry, ACM, Minneapolis, MN, 1998, pp. 86–95.

[18] G. L. Miller, D. Talmor, S.-H. Teng, N. Walkington, A Delaunay based numerical method for three dimensions: generation, formulation, and partition, in: Proceedings of the 27th Annu. ACM Sympos. Theory Comput, ACM, 1995, pp. 683–692.

[19] A. Chernikov, N. Chrisochoides, Generalized insertion region guides for Delaunay mesh refinement, SIAM Journal on Scientific Computing 34 (2012) A1333–A1350.

[20] A. Chernikov, N. Chrisochoides, Multitissue tetrahedral image-to-mesh conversion with guaranteed quality and fidelity, SIAM Journal on Scientific Computing 33 (2011) 3491–3508.

[21] H. Si, Constrained Delaunay tetrahedral mesh generation and refinement, Finite Elements in Analysis and Design 46 (2010) 33–46.

[22] J. R. Shewchuk, Delaunay refinement algorithms for triangular mesh generation, Computational Geometry: Theory and Applications 22 (2002) 21–74.

[23] S. Oudot, L. Rineau, M. Yvinec, Meshing volumes bounded by smooth surfaces, in: Proceedings of the International Meshing Roundtable, Springer-Verlag, 2005, pp. 203–219.

[24] F. Labelle, J. R. Shewchuk, Isosurface stuffing: fast tetrahedral meshes with good dihedral angles, ACM Transactions on Graphics 26 (2007) 57.1–57.10.

[25] J.-P. Pons, F. Ségonne, J.-D. Boissonnat, L. Rineau, M. Yvinec, R. Keriven, High-Quality Consistent Meshing of Multi-label Datasets, in: Information Processing in Medical Imaging, Springer Berlin Heidelberg, 2007, pp. 198–210.

[26] D. Boltcheva, M. Yvinec, J.-D. Boissonnat, Mesh Generation from 3D Multi-material Images, in: Medical Image Computing and Computer-Assisted Intervention, Springer, 2009, pp. 283–290.

[27] H. L. D. Cougny, M. S. Shephard, Parallel refinement and coarsening of tetrahedral meshes, International Journal for Numerical Methods in Engineering 46 (1999) 1101–1125.

[28] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, L. Wilcox, Extreme-scale amr, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, IEEE Computer Society, 2010, pp. 1–12.

[29] T. Tu, D. R. O. Hallaron, O. Ghattas, Scalable parallel octree meshing for terascale applications, in: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05, IEEE Computer Society, 2005, pp. 4–.

[30] Y. Ito, A. Shih, A. Erukala, B. Soni, A. Chernikov, N. Chrisochoides, K. Nakahashi, Parallel mesh generation using an advancing front method, Mathematics and Computers in Simulation 75 (2007) 200–209.

[31] J. Galtier, P.-L. George, Prepartitioning as a way to mesh subdomains in parallel, in: Special Symposium on Trends in Unstructured Mesh Generation, ASME/ASCE/SES, 1997, pp. 107–122.

[32] C. M. J. Kadow, Parallel Delaunay Refinement Mesh Generation, 2004. PhD Thesis, Carnegie Mellon University.

[33] D. Nave, N. Chrisochoides, P. Chew, Parallel Delaunay refinement for restricted polyhedral domains, Computational Geometry: Theory and Applications 28 (2004) 191–215.

[34] L. Linardakis, N. Chrisochoides, Graded Delaunay decoupling method for parallel guaranteed quality planar mesh generation, SIAM Journal on Scientific Computing 30 (2008) 1875–1891.

[35] U. Hartmann, F. Kruggel, A Fast Algorithm for Generating Large Tetrahedral 3D Finite Element Meshes from Magnetic Resonance Tomograms, in: Proceedings of the IEEE Workshop on Biomedical Image Analysis, WBIA, IEEE Computer Society, Washington, DC, USA, 1998, pp. 184–192.

[36] P. Hu, H. Chen, W. Wu, P.-A. Heng, Multi-tissue tetrahedral mesh generation from medical images, in: International Conference on Bioinformatics and Biomedical Engineering (iCBBE), IEEE, 2010, pp. 1–4.

[37] A. Kuprat, A. Khamayseh, D. George, L. Larkey, Volume conserving smoothing for piecewise linear curves, surfaces, and triple lines, Journal of Computational Physics 172 (2001) 99–118.

[38] A. P. Kuprat, D. R. Einstein, An anisotropic scale-invariant unstructured mesh generator suitable for volumetric imaging data, J. Comput. Phys. 228 (2009) 619–640.

[39] V. Dyedov, D. R. Einstein, X. Jiao, A. P. Kuprat, J. P. Carson, F. del Pin, Variational generation of prismatic boundary-layer meshes for biomedical computing, International Journal for Numerical Methods in Engineering 79 (2009) 907–945.

[40] P. Foteinos, N. Chrisochoides, Dynamic parallel 3D Delaunay triangulation, in: International Meshing Roundtable, Springer Berlin Heidelberg, Paris, France, 2012, pp. 3–20.

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

[42] D. K. Blandford, G. E. Blelloch, C. Kadow, Engineering a compact parallel Delaunay algorithm in 3D, in: Proceedings of the 22$^{\text{nd}}$ Symposium on Computational Geometry, SCG '06, ACM, New York, NY, USA, 2006, pp. 292–300.

[43] G. E. Blelloch, G. L. Miller, J. C. Hardwick, D. Talmor, Design and implementation of a practical parallel Delaunay algorithm, Algorithmica 24 (1999) 243–269.

[44] A. Chernikov, N. Chrisochoides, Algorithm 872: Parallel 2D constrained Delaunay mesh generation, ACM Transactions on Mathematical Software 34 (2008) 6–25.

[45] A. N. Chernikov, N. P. Chrisochoides, Three-dimensional Delaunay refinement for multi-core processors, in: Proceedings of the 22nd annual international Conference on Supercomputing, ICS '08, ACM, New York, NY, USA, 2008, pp. 214–224.

[46] R. Said, N. Weatherill, K. Morgan, N. Verhoeven, Distributed parallel Delaunay mesh generation, Computer Methods in Applied Mechanics and Engineering 177 (1999) 109–125.

[47] P. Sewell, T. Benson, C. Christopoulos, D. W. P. Thomas, A. Vukovic, J. Wykes, Transmission-line modeling (TLM) based upon unstructured tetrahedral meshes, Microwave Theory and Techniques, IEEE Transactions on 53 (2005) 1919–1928.

[48] M. Zhou, O. Sahni, T. Xie, M. S. Shephard, K. E. Jansen, Unstructured mesh partition improvement for implicit finite element at extreme scale, J. Supercomput. 59 (2012) 1218–1228.

[49] T. Okusanya, J. Peraire, 3D parallel unstructured mesh generation, 1997. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.7898.

[50] R. Löhner, A 2nd generation parallel advancing front grid generator, in: X. Jiao, J.-C. Weill (Eds.), Proceedings of the 21st International Meshing Roundtable, Springer Berlin Heidelberg, 2013, pp. 457–474.

[51] L. Oliker, R. Biswas, Parallelization of a dynamic unstructured algorithm using three leading programming paradigms, IEEE Trans. Parallel Distrib. Syst. 11 (2000) 931–940.

[52] N. Amenta, M. Bern, Surface reconstruction by Voronoi filtering, in: SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry, ACM, New York, NY, USA, 1998, pp. 39–48.

[53] N. Amenta, S. Choi, R. K. Kolluri, The power crust, in: Proceedings of the sixth ACM symposium on Solid modeling and applications, SMA '01, ACM, New York, NY, USA, 2001, pp. 249–266.

[54] T. K. Dey, W. Zhao, Approximate medial axis as a voronoi subcomplex, Computer-Aided Design 36 (2004) 195–202.

[55] J.-D. Boissonnat, S. Oudot, Provably good sampling and meshing of surfaces, Graphical Models 67 (2005) 405–451.

[56] R. Staubs, A. Fedorov, L. Linardakis, B. Dunton, N. Chrisochoides, Parallel n-dimensional exact signed euclidean distance transform, The Insight Journal (2006).

[57] W. E. Lorensen, H. E. Cline, Marching cubes: A high resolution 3D surface construction algorithm, SIGGRAPH Computer Graphics 21 (1987) 163–169.

[58] C. R. Maurer, Q. Rensheng, V. Raghavan, A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions, IEEE Transactions on Pattern Analysis and Machine Intelligence 25 (2003) 265 – 270.

[59] O. Devillers, M. 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, SIAM, 2003, pp. 313–319.

[60] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, Cilk: an efficient multithreaded runtime system, in: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '95, ACM, New York, NY, USA, 1995, pp. 207–216.

[61] W. N. Scherer, III, M. L. Scott, Advanced contention management for dynamic software transactional memory, in: Proceedings of the 24$^{\text{th}}$ annual ACM symposium on Principles of distributed computing, PODC '05, ACM, 2005, pp. 240–248.

[62] M. Herlihy, V. Luchangco, M. Moir, Obstruction-free synchronization: Double-ended queues as an example, in: Proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS '03, IEEE Computer Society, 2003, pp. 522–.

[63] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, L. P. Chew, Scheduling strategies for optimistic parallel execution of irregular programs, in: Proc. Symp. on Parallelism in algorithms and architectures (SPAA), ACM, New York, NY, USA, 2008, pp. 217–228.

[64] C. Antonopoulos, X. Ding, A. Chernikov, F. Blagojevic, D. Nikolopoulos, N. Chrisochoides, Multigrain parallel Delaunay mesh generation: Challenges and opportunities for multithreaded architectures, in: ACM International Conference on Supercomputing, 19, ACM, 2005, pp. 367–376.

[65] M. Ben-Ari, Principles of concurrent programming, Chapter 3, pages 30-43, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[66] M. Herlihy, J. E. B. Moss, Transactional memory: architectural support for lock-free data structures, SIGARCH Comput. Archit. News 21 (1993) 289–300.

[67] D. Nave, P. Chew, N. Chrisochoides, Guaranteed-quality parallel Delaunay refinement for restricted polyhedral domains, in: ACM Symposium on Computational Geometry (SoCG), ACM, 2002, pp. 135–144.

[68] J. Richolt, M. Jakab, R. Kikinis, SPL Knee Atlas (2011). Available at: http://www.spl.harvard.edu/publications/item/view/1953.

[69] J. L. Gustafson, Reevaluating Amdahl's law, Communications of the ACM 31 (1988) 532–533.

[70] SGI UV 100/1000 system specifications, http://www.sgi.com/products/servers/uv/specs.html, 2012. Available online.

[71] O. Devillers, S. Pion, Efficient exact geometric predicates for delaunay triangulations, in: Proc. 5th Workshop Algorithm Eng. Exper., SIAM, 2003, pp. 37–44.

[72] M. Jakab, R. Kikinis, Head and neck atlas (2012). Available at: http://www.spl.harvard.edu/publications/item/view/2271.

[73] P. Foteinos, Y. Liu, A. Chernikov, N. Chrisochoides, An Evaluation of Tetrahedral Mesh Generation for Non-Rigid Registration of Brain

MRI, in: Computational Biomechanics for Medicine V, 13th International Conference on Medical Image Computing and Computer Assisted Intervention (MICCAI) Workshop, Springer, 2010, pp. 126–137.

[74] A. Fedorov, N. Chrisochoides, Tetrahedral Mesh Generation for Non-rigid Registration of Brain MRI: Analysis of the Requirements and Evaluation of Solutions, in: International Meshing Roundtable, Springer Verlag, 2008, pp. 55–72.

[75] N. Chentanez, R. Alterovitz, D. Ritchie, L. Cho, K. K. Hauser, K. Goldberg, J. R. Shewchuk, J. F. O'Brien, Interactive simulation of surgical needle insertion and steering, in: Proceedings of ACM SIGGRAPH 2009, pp. 88:1–10.

[76] M. Behr, Simplex space-time meshes in finite element simulations, International Journal for Numerical Methods in Fluids 57 (2008) 1421–1434.

[77] T. C. S. Rendall, C. B. Allen, E. D. C. Power, Conservative unsteady aerodynamic simulation of arbitrary boundary motion using structured and unstructured meshes in time, International Journal for Numerical Methods in Fluids 70 (2012) 1518–1542.