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

Panagiotis Foteinos
Department of Computer Science
College of William and Mary
Old Dominion University
pfot@cs.wm.edu

Nikos Chrisochoides
Department of Computer Science
Old Dominion University
nikos@cs.odu.edu

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, it is able to recover the isosurface and mesh 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 which boost the parallel efficiency with little overhead: our single-threaded performance is faster than CGAL, the state of the art sequential mesh generation software we are aware of. The effectiveness of our method is shown on Blacklight, the Pittsburgh Supercomputing Center's cache-coherent NUMA machine, via a series of case studies justifying our choices. We observe a more than 82% strong scaling efficiency for up to 64 cores, and a more than 95% weak scaling efficiency for up to 144 cores, reaching a rate of 14.7 Million Elements per second. To the best of our knowledge, this is the fastest and most scalable 3D Delaunay refinement algorithm.

1 Introduction

Image-to-mesh (I2M) conversion enables patient-specific Finite Element modeling in image guided diagnosis and therapy. The ability to tessellate a medical image of multiple tissues into tetrahedra enables Finite Element (FE) Analysis on patient-specific models [6, 38]. This has significant implications in many areas, such as imaged-guided therapy, development of advanced patient-specific blood flow mathematical models 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.

The sequential 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 volume $\mathcal{O}$ (about to be meshed) is given as a *Piecewise Linear Complex* (PLC) which contains linear segments and polygonal facets embedded in 3 dimensions [19, 21, 41, 51]. The limitation 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 [51, 53].

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 [45], level-sets and segmented multi-labeled images [15, 36, 46]. Function f can also represent PLCs [36] (albeit without topological guarantees), 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. Also, since the isosurface is not already meshed into a PLC but it is recovered and meshed during refinement, this method does not suffer from any angle constraints.

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 concurrently. 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 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 [17, 23, 32, 56] or $\partial\mathcal{O}$ is already represented as a polyhedral domain [29, 33, 37, 42]. 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 technique avoids the undesired "stair-case" effect [15, 46] that occurs to algorithms that treat the surface voxels as the PLC of the domain [19] (since those input planar facets meet always at 0 or 90 degrees) or to algorithms that heuristically recover the isosurface [40] without topological guarantees.

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. Note that none of the parallel Delaunay refinement algorithms we know of support point removals. Point removal, however, offers new and rich refinement schemes which are shown in the sequential meshing literature [26, 34] to be very effective in practice.

In our previous work [27], 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 [8, 11, 12, 27], 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 [8, 11, 12], 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 [27], such that the "discovery" of the (dynamically changing) set of points to be inserted or removed (that will eventually force the final mesh 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.

In this paper, we design and implement PI2M, a parallel Isosurface-based Delaunay mesh refinement scheme based on our earlier sequential prototype [26, 28]. 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 code and it is consistently 35% faster than CGAL [2], the state of the art Isosurface-based meshing tool.

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 [7]. We test and show the effectiveness of PI2M on the challenging cc-NUMA architecture; specifically, we used the Pittsburgh Supercomputing Center's Blacklight, employing BoostC++ threads. We observe a more than 82% strong scaling efficiency for up to 64 cores, and also a more than 95% weak scaling efficiency for up to 144 cores, reaching a rate of 14.7 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. For a higher core count, however, our method exhibits considerable performance degradation. We show 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. This problem could be potentially be alleviated by using hybrid approaches to explore network hierarchies [22]. However, this is outside the scope of the paper. Our goal is to develop the most efficient and scalable method on a moderate number of cores. Our long term goal is to increase scalability by exploiting concurrency at different levels [22].

In summary, the method we present (PI2M): **(1)** exhibits the best single-threaded performance, to the best of our knowledge, **(2)** supports parallel Delaunay insertions and removals; past methods including ours dealt only with point insertions, a much easier task, **(3)** conforms to user-specified quality, and most importantly, to the newly added fidelity constraints, **(4)** recovers and meshes the isosurface with topological and geometric guarantees from the beginning of mesh refinement, and thus exploits parallelism earlier (see Figure 4), and **(5)** fills the volume of the domain $\mathcal{O}$ with millions of high-quality tetrahedra within seconds.

Section 2 presents the related work. Section 3 briefly describes the Sequential Delaunay Refinement for Smooth Surfaces and 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

There is extensive previous work on parallel mesh generation, including various techniques, such as: Delaunay, Octree, or Advancing Front meshing. Note that parallel Delaunay mesh generation/refinement should not be confused with parallel Delaunay triangulation [8, 11, 12, 27]. Delaunay 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 the other parallel mesh generation algorithms in the literature is that they either have the surface of the domain given as a polyhedron, or the extraction of the polyhedron is done sequentially, or they start by an initial background octree. As explained in Section 1, our method constructs the polyhedral representation of the object's surface in parallel, together with the volume meshing, thus taking advantage of another degree of parallelism.

Given an initial mesh, de Cougny and Shephard [23] 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 [42], the authors implement a tightly-coupled method like ours. We, however, take extra care to greatly reduce the number of rollbacks (see Section 5), and thus achieve excellent speedups for a higher core count. In [18] and [37], a partially-coupled and a decoupled method for distributed systems is developed based on medial axis decomposition. Medial axis decomposition in 3D inputs, however, is a very challenging and still unsolved task. In contrast, our method does not rely on any domain decomposition, and as such, it is flexible enough to be extended to arbitrary dimensions. Kadow [33] starts from a polygonal surface (PSLG) and offers tightly coupled refinement schemes in 2D, respectively. In our case, the polyhedral representation of the object's surface is done in parallel, which adds extra functionality. In [20], the authors present a method which allows the safe insertion of points independently without synchronization. The extension, however, of their method for point removal support is not straightforward. Galtier and George [29] 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.* [49] subdivide the domain into decoupled blocks. Each block then is meshed with considerably less communication and synchronization. Tu *et al.* [56] describe a parallel octree method that interacts with the solver in parallel and efficiently. The work of Zhou *et*

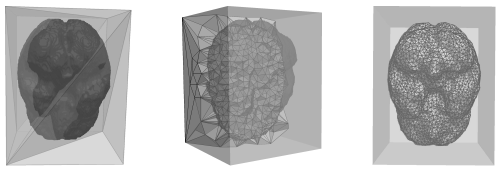
(a) (b) (c)

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 $\mathcal{M}$.

al. [58], and the *Forest-of-octrees* method of Burstedde *et al.* [17] offer techniques for fair and efficient data migration and partitioning in parallel. Load balancing and data migration is also used by Okusanya and Peraire [43] to distribute bad elements across processors. Ito *et al.* [32] start from an initial mesh and Löhner [39] from a PLC for subsequent parallel mesh generation in advancing front fashion. Oliker and Biswas [44] 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 in a much higher core count; 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} [22].

3 Delaunay Refinement for Smooth Surfaces

Sequential Delaunay Refinement for smooth surfaces is presented in detail in the literature [45, 46] and in our previous work [26, 28]. In this Section, we briefly outline the main concepts.

As is usually the case in the literature [3, 36, 46], 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* [3]. Omitting the details, it can be proved [3, 4] 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 [4, 14, 25]:

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 provably good topological approximation 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 [4, 14, 25, 45] 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 $\mathrm{cis}(p) \in \partial\mathcal{O}$. 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 [54]. Specifically, the EDT returns the *surface voxel* p 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 $\overrightarrow{pp'}$ on small intervals and we compute $\mathrm{cis}(p) \in \partial\mathcal{O}$ by interpolating the positions of different labels [40]. 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 [51]) 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 [52] and to ensure that the vertices of the boundary mesh triangles lie precisely on the isosurface [45].

In summary, tetrahedra and faces are refined according to the following *Refinement Rules*: **(R1)** Let t be an intersecting tetrahedron. Compute the closest isosurface point $z = \mathrm{cis}(c(t))$. If z is at a distance not closer than δ to any other isosurface vertex, then z is inserted. **(R2)** Let t be an intersecting tetrahedron. Compute the closest isosurface point $z = \mathrm{cis}(c(t))$. 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 $c_{\mathrm{surf}}(f)$. If either its smallest planar angle is less than 30° or a vertex of f is not an isosurface vertex, then $c_{\mathrm{surf}}(f)$ is inserted. **(R4)** If t is an interior tetrahedron whose radius-edge ratio is larger than 2, then $c(t)$ is inserted. **(R5)** Let t be an interior tetrahedron. If its radius $r(t)$ is larger than $\mathrm{sf}(c(t))$, 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 [26, 28, 45] for more details. When none of the above rules applies, then refinement is complete. In our previous work [26, 28], we prove that termination is guaranteed, the radius-edge ratio of ell 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

In this Section, we outline the main concepts of our parallel implementation. Note that our tightly-coupled parallelization does not alter the fidelity (Theorem 1) and the quality guarantees described in the previous section.

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.

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 [16, 57]. 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 [24], 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 [42]. When a rollback occurs, the thread moves on to the next bad element in its PEL.

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.

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) [13] 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.

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 [50]. 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 rollbacks, if possible [31, 50].

We implemented and compared three contention techniques: the *Aggressive Contention Manager* (Aggressive-CM) [50], the *Random Contention Manager* (Random-CM), and the *Global Contention Manager* (Global-CM).

The Aggressive-CM and Random-CM are non-blocking schemes. As is usually the case for non-blocking schemes [5, 31, 35, 42, 50], 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 [5, 35, 42].

Note that none of the earlier Transactional Memory techniques [31, 50] and the Random Contention Managers presented in the past [5, 35, 42] 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.

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 5.4) 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 [5, 35, 42] 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 efficient 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 [10], 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 5.4).

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 rollbacks. 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.

Table 1: Comparison among Contention Managers (CM). Global-CM greatly reduced the number of rollbacks and the overhead time.

	128 cores		256 cores
	Random-CM	Global-CM	Global-CM
execution time	64s	23s	22s
rollbacks	25×10^6 (52.9%)	728,087 (3.1%)	883,768 (3.6%)
contention overhead	4,317s	1,049	3,054s
load balance overhead	869s	130s	281s
rollback overhead	595s	3s	3s
total overhead	5,703s	1,183s	3,339s
speedup	16.9	47.0	49.1

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.

5.4 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. The **contention overhead time** 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). The **load balance overhead time** 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. Lastly, the **rollback overhead time** 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.

The Aggressive-CM is missing for both core counts, and the Random-CM is missing from the 256-core experiment, because they were stuck in a livelock. Indeed, we experimentally found that they did not make any progress for more than 1 hour, a scenario of livelock.

Observe that Global-CM greatly reduced the number of rollbacks and the associated wasted cycles, as compared to the standard Contention Manager used in the literature. Random-CM encounters a rollback 50% of the total oper-

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

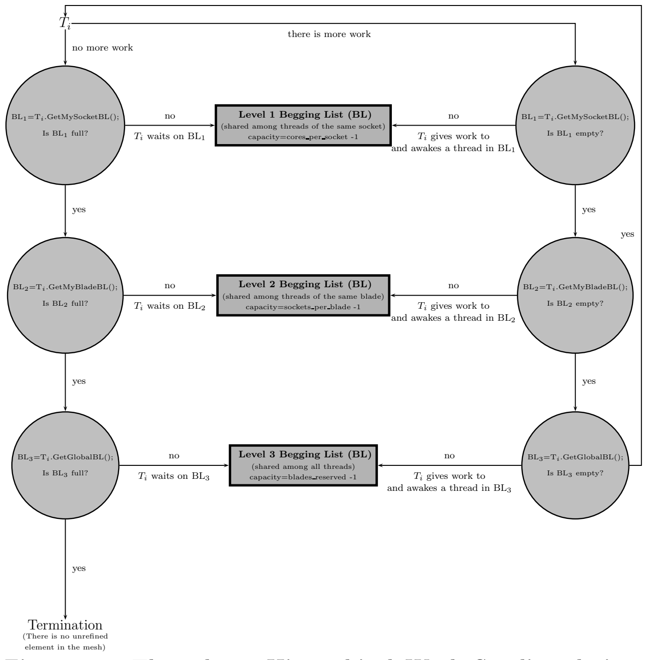
T_i
there is more work
no more work
BL_1=T_i.GetMySocketBL(); Is BL_1 full?
no
T_i waits on BL_1
Level 1 Begging List (BL)
(shared among threads of the same socket)
capacity=cores_per_socket -1
no
T_i gives work to and awakes a thread in BL_1
BL_1=T_i.GetMySocketBL(); Is BL_1 empty?
yes
yes
yes
BL_2=T_i.GetMyBladeBL(); Is BL_2 full?
no
T_i waits on BL_2
Level 2 Begging List (BL)
(shared among threads of the same blade)
capacity=sockets_per_blade -1
no
T_i gives work to and awakes a thread in BL_2
BL_2=T_i.GetMyBladeBL(); Is BL_2 empty?
yes
yes
BL_3=T_i.GetGlobalBL(); Is BL_3 full?
no
T_i waits on BL_3
Level 3 Begging List (BL)
(shared among all threads)
capacity=blades_reserved -1
no
T_i gives work to and awakes a thread in BL_3
BL_3=T_i.GetGlobalBL(); Is BL_3 empty?
yes
Termination
(There is no unrefined element in the mesh)

Figure 2: The 3-layer Hierarchical Work Stealing design. Threads ask work from their closest candidates first.

ations attempted, while Global-CM decreased it down to 3% on 128 cores. See the speedup achieved by these two CMS (over the single-thread execution): Global-CM is 2.7 times faster. Even on 256 cores, where both Aggressive-CM and Random-CM failed to terminate, Global-CM achieved a (slight) improvement over its 128-core performance.

Although there are other elaborate and hybrid contention techniques [31, 50], none of them guarantees the absence of livelocks. Therefore, we chose Global-CM because of its simplicity and efficiency.

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. See Figure 2. 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 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 3 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 by IRCAD Laparoscopic Center (http://www.ircad.fr/). The final mesh generated consists of 147×10^6 elements. On a single Blacklight core, the execution time was 1300 seconds.

Observe that the speed-up of RWS deteriorates by a lot for more than 64 cores (see the green line in Figure 3a). 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 3b), 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% of all the number of times threads asked for work, they received it from a thread of their own blade, yielding a 42% reduction in inter-blade accesses, as Figure 3b shows.

Figure 3c 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 Global-CM and the Hierarchical Work Stealing method (HLB) 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 the CT abdominal atlas (http://www.ircad.fr/softwares/3Dircadb/3Dircadb2/3Dircadb2.2.zip). Other inputs (the knee [48] and the brain atlas [55] for example) exhibit similar results on comparable mesh sizes. We do not report them here, because of space limitations.

We measure the number of tetrahedra created per second across the runs. Specifically, let us define with Elements (n) and Time (n), the number of elements created and the time elapsed, when n threads are employed. Then, the speedup is defined as $\frac{\text{Elements}(n) \times \text{Time}(n)}{\text{Time}(n) \times \text{Elements}(1)}$. With n threads, a perfect speedup would be equal to n [30].

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

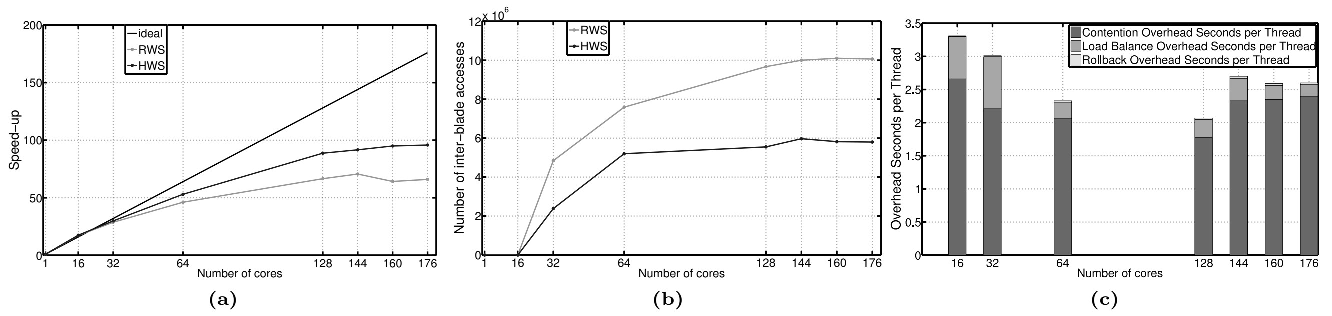

Figure 3: Strong scaling performance achieved by the classic Random Work Stealing (RWS) and Hierarchical Work Stealing (HWS). (a)-(b) Comparison between RWS and HWS on speed-up ($\frac{\text{time}_1}{\text{time}_{\#\text{Threads}}}$) and on the number of inter-blade accesses. (c) Breakdown of the overhead time for HWS.

Table 3: Weak scaling performance. Across runs, the number of elements per thread remains approximately constant.

#Threads	1	32	64	128	144	160	176
#Elements	1.08E+07	3.49E+08	7.44E+08	1.32E+09	1.51E+09	1.67E+09	1.85E+09
Time (secs)	100.80	92.25	102.27	95.91	102.98	135.81	180.96
Elements per second	1.07E+05	3.79E+06	7.28E+06	1.37E+07	**1.47E+07**	1.23E+07	1.02E+07
Speedup	1.00	35.41	68.05	**128.34**	137.24	115.06	95.71
Efficiency	1.000	1.107	1.063	**1.003**	**0.953**	0.719	0.544
Overhead secs per thread	0.00	2.28	3.74	4.62	4.66	6.85	8.54

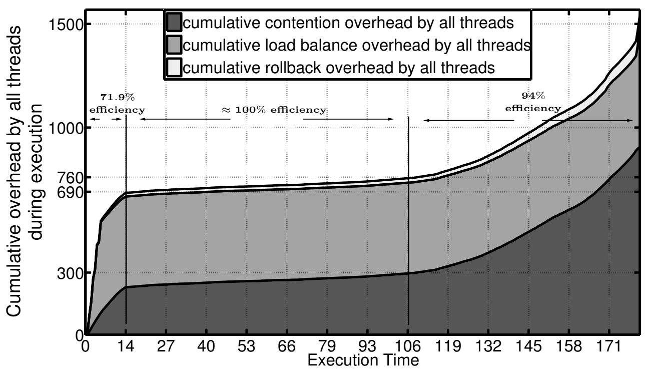

Figure 4: Overhead time breakdown with respect to the wall time for the experiment on 176 cores of Table 3. A pair (x, y) tells us that up to the x^{th} second of execution, threads have not been doing useful work so far for y seconds all together.

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 3. The observed speedup is super-linear for up to 128 threads. On 144 cores, we achieve an unprecedented efficiency of 95%, and a rate of 14.7 Million Elements per second. It is worth mentioning that CGAL [2], 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 746.39.

Nevertheless, 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 5.4), 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 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 4 for an illustration of the 176-core experiment of Table 3. 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 5.4) 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, there is high contention and severe load imbalance. Nevertheless, even in this case, $\frac{176\times14-690}{176\times14} \approx 71.9\%$ 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. 176 threads run for 180.96s each, and, on the average, they do useless work for 8.54s 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 180.96s-8.54s =172.42s. 172.42s, however, is far from the ideal 100.8s (that the first column with 1 thread shows) by 172.42s-100.8=71.62s. Therefore, while rollbacks, contention management, and load balancing introduce a merely 8.54s overhead, the real bottleneck is the 71.62s overhead spent on memory (often remote) loads/stores. Nevertheless, we verified that the number of remote accesses, LLC, and TLB misses remain constant per thread across runs, which is the ideal scenario: our Hierarchical Load Balancer (HLB) suppresses successfully excessive remote accesses thus achieving better data locality. Since, however, 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 [1]), or both. It seems that after 144 cores, this pressure on the switches slows performance down. A hybrid approach [22] able to scale for larger network hierarchies is left for future work.

Table 4: The single-threaded performance of our algorithm. It includes the extra overhead introduced by synchronization, contention management, and load balancing to support the (potential) presence of other threads.

	PI2M	CGAL
#tetrahedra / seconds	51,968	33,462
time	7.1 secs	11.0 secs
#tetrahedra	368,974	368,077
max radius-edge ratio	2	2
smallest boundary planar angle	30°	30°
(min, max) dihedral angles	(4.6°, 170.2°)	(3.5°, 174.7°)
Hausdorff distance	10.7 mm	10.3 mm

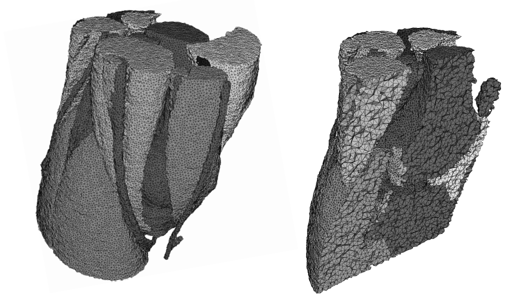
(a) whole mesh (b) cross section

Figure 5: The 368,974 element mesh generated by PI2M with input image the MRI knee atlas [48].

Hyper-threading suffers from the same memory congestion with one exception: the slow-down happens in a lower core count, a fact that needs further investigation. Nevertheless, for low core counts, we obtain a considerable improvement: on 64 cores (128 threads), the execution time is 70.8s, a 30% time decrease compared to the 64 non hyper-threaded core experiment of Table 3.

7 Single-threaded evaluation

In this section, we show that our single-threaded execution time, although it introduces extra overhead due to locking, synchronization, contention management bookkeeping (see Section 5), and hierarchical load balance (see Section 6.1), it is faster than and has similar quality to CGAL, the quickest state-of-the-art Delaunay Isosurface-based mesh generator we are aware of.

See Table 4. For this case study, we set the sizing parameters of CGAL to values that produced meshes of similar size with ours. Observe that the resulting meshes are of similar quality and that the single-threaded PI2M is 35% faster than CGAL. Our timing includes the 1.2 seconds needed for the computation of the Euclidean Distance Transform (see Section 3).

The input image used in this study is the 48-tissue MRI knee atlas freely available at [48]. Other input images yielded the same results: PI2M generates meshes of similar quality to CGAL's and 35% faster. Figure 5 illustrates the mesh generated by PI2M. We used CRTC (see Table 2 for its specifications) for this case study.

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 [8, 11, 12, 27], 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 tighly-coupled method our group developed in the past [42]. The data decomposition method [20] does not support Delaunay removals, a technique that it is shown to be effective in the sequential mesh generation literature [26, 28]. The extension of partially-coupled [18] and decoupled [37] 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 [9, 47]) in parallel, thus, exploiting parallelism in the fourth dimension.

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) provably eliminates deadlocks and livelocks and, at the same time, reduces the number of rollbacks by a factor of 17 compared to the standard Random technique found in the literature. Global-CM also reduced the number of wasted cycles by a factor of 4.8, improving energy-efficiency by avoiding energy waste because of rollbacks. Our Hierarchical Load Balancer (HLB) sped up the execution by a factor of 1.44 on 176 cores, as a result of a 42% remote accesses reduction.

All in all, PI2M achieves a strong scaling efficiency of more than 82% on 64 cores. It also achieves an excellent weak scaling efficiency of more than 95% on up to 144 cores. Other image inputs yielded very similar results. Because of the limited space, we do not report them here. 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 35% faster than CGAL and with similar quality.

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 [22] 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 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.

9 References

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

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

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

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

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

[6] N. Archip, O. Clatz, A. Fedorov, A. Kot, S. Whalen, D. Kacher, N. Chrisochoides, F. Jolesz, A. Golby, P. Black, and 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(2):609–624, 2007.

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

[8] V. H. Batista, D. L. Millman, S. Pion, and J. Singler. Parallel geometric algorithms for multi-core computers. *Computational Geometry*, 43(8):663–677, 2010.

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

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

[11] D. K. Blandford, G. E. Blelloch, and C. Kadow. Engineering a compact parallel Delaunay algorithm in 3D. In *Proceedings of the 22nd Symposium on Computational Geometry*, SCG '06, pages 292–300, New York, NY, USA, 2006. ACM.

[12] G. E. Blelloch, G. L. Miller, J. C. Hardwick, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24(3):243–269, 1999.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and 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, pages 207–216, New York, NY, USA, 1995. ACM.

[14] J.-D. Boissonnat and S. Oudot. Provably good sampling and meshing of surfaces. *Graphical Models*, 67(5):405–451, 2005.

[15] D. Boltcheva, M. Yvinec, and J.-D. Boissonnat. Mesh Generation from 3D Multi-material Images. In *Medical Image Computing and Computer-Assisted Intervention*, pages 283–290. Springer, September 2009.

[16] A. Bowyer. Computing Dirichlet tesselations. *Computer Journal*, 24:162–166, 1981.

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

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

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

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

[21] A. N. Chernikov and N. P. Chrisochoides. Generalized insertion region guides for Delaunay mesh refinement. *SIAM Journal on Scientific Computing (SISC)*, 2011. under revision.

[22] N. Chrisochoides, A. Chernikov, A. Fedorov, A. Kot, L. Linardakis, and P. Foteinos. Towards exascale parallel Delaunay mesh generation. In *International Meshing Roundtable*, number 18, pages 319–336, Salt Lake City, Utah, October 2009.

[23] H. L. D. Cougny and M. S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 46(7):1101–1125, 1999.

[24] O. Devillers and M. Teillaud. Perturbations and vertex removal in a 3D Delaunay triangulation. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete algorithms*, SODA '03, pages 313–319, 2003.

[25] T. K. Dey and W. Zhao. Approximate medial axis as a voronoi subcomplex. *Computer-Aided Design*, 36(2):195–202, 2004.

[26] P. Foteinos, A. Chernikov, and N. Chrisochoides. Guaranteed Quality Tetrahedral Delaunay Meshing for Medical Images. In *Proceedings of the 7th International Symposium on Voronoi Diagrams in Science and Engineering*, pages 215–223, June 2010.

[27] P. Foteinos and N. Chrisochoides. Dynamic parallel 3D Delaunay triangulation. In *International Meshing Roundtable*, pages 9–26, Paris, France, October 2011.

[28] P. Foteinos and N. Chrisochoides. High-quality multi-tissue mesh generation for finite element analysis. In *MeshMed, Workshop on Mesh Processing in Medical Image Analysis (MICCAI)*, pages 18–28, September 2011.

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

[30] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[31] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, 2003.

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

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

[34] B. M. Klingner and J. R. Shewchuk. Aggressive tetrahedral mesh improvement. In *Proceedings of the International Meshing Roundtable*, pages 3–23. Springer, 2007.

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

[36] F. Labelle and J. R. Shewchuk. Isosurface stuffing: fast tetrahedral meshes with good dihedral angles. *ACM Transactions on Graphics*, 26(3):57, 2007.

[37] L. Linardakis and N. Chrisochoides. Graded Delaunay decoupling method for parallel guaranteed quality planar mesh generation. *SIAM Journal on Scientific Computing*, 30(4):1875–1891, March 2008.

[38] Y. Liu, C. Yao, L. Zhou, and N. Chrisochoides. A point based non-rigid registration for tumor resection using imri. In *IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 1217–1220, April 2010.

[39] R. Löhner. A 2nd generation parallel advancing front grid generator. In X. Jiao and J.-C. Weill, editors, *Proceedings of the 21st International Meshing Roundtable*, pages 457–474, 2013.

[40] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.

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

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

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

[44] L. Oliker and R. Biswas. Parallelization of a dynamic unstructured algorithm using three leading programming paradigms. *IEEE Trans. Parallel Distrib. Syst.*, 11(9):931–940, Sept. 2000.

[45] S. Oudot, L. Rineau, and M. Yvinec. Meshing volumes bounded by smooth surfaces. In *Proceedings of the International Meshing Roundtable*, pages 203–219. Springer-Verlag, September 2005.

[46] J.-P. Pons, F. Ségonne, J.-D. Boissonnat, L. Rineau, M. Yvinec, and R. Keriven. High-Quality Consistent Meshing of Multi-label Datasets. In *Information Processing in Medical Imaging*, pages 198–210, 2007.

[47] T. C. S. Rendall, C. B. Allen, and 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(12):1518–1542, 2012.

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

[49] R. Said, N. Weatherill, K. Morgan, and N. Verhoeven. Distributed parallel Delaunay mesh generation. *Computer Methods in Applied Mechanics and Engineering*, 177(1-2):109–125, 1999.

[50] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the* 24th *annual ACM symposium on Principles of distributed computing*, PODC '05, pages 240–248. ACM, 2005.

[51] J. R. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. In *Proceedings of the* 14th *ACM Symposium on Computational Geometry*, pages 86–95, Minneapolis, MN, 1998.

[52] J. R. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1-3):21–74, May 2002.

[53] H. Si. Constrained Delaunay tetrahedral mesh generation and refinement. *Finite Elements in Analysis and Design*, 46:33–46, 2010.

[54] R. Staubs, A. Fedorov, L. Linardakis, B. Dunton, and N. Chrisochoides. Parallel n-dimensional exact signed euclidean distance transform. September 2006.

[55] I. Talos, M. Jakab, R. Kikinis, and M. Shenton. SPL-PNL Brain Atlas. March 2008.

[56] T. Tu, D. R. O. Hallaron, and O. Ghattas. Scalable parallel octree meshing for terascale applications. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, 2005.

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

[58] M. Zhou, O. Sahni, T. Xie, M. S. Shephard, and K. E. Jansen. Unstructured mesh partition improvement for implicit finite element at extreme scale. *J. Supercomput.*, 59(3):1218–1228, Mar. 2012.