# Guaranteed–Quality Parallel Delaunay Refinement for Restricted Polyhedral Domains

Démian Nave[a,*,1], Nikos Chrisochoides[b,*,2] L. Paul Chew[c,3]

[a] *Pittsburgh Supercomputing Center, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

[b] *Department of Computer Science, College of William and Mary, Williamsburg, VA 23187, USA*

[c] *Department of Computer Science, Cornell University, Ithaca, NY 14853, USA*

**Abstract**

We describe a distributed memory parallel Delaunay refinement algorithm for simple polyhedral domains whose constituent bounding edges and surfaces are separated by angles between $90^o$ to $270^o$ inclusive. With these constraints, our algorithm can generate meshes containing tetrahedra with circumradius to shortest edge ratio less than 2, and can tolerate more than 80% of the communication latency caused by unpredictable and variable remote gather operations.

Our experiments show that the algorithm is efficient in practice, even for certain domains whose boundaries do not conform to the theoretical limits imposed by the algorithm. The algorithm we describe is the first step in the development of much more sophisticated guaranteed–quality parallel mesh generation algorithms.

*Key words:* Delaunay triangulation, guaranteed–quality mesh generation, parallel mesh generation, distributed mesh data structures

* Corresponding author.
*Email addresses:* `dnave@psc.edu` (Démian Nave), `nikos@cs.wm.edu` (Nikos Chrisochoides), `chew@cs.cornell.edu` (L. Paul Chew).

[1] Supported by the NSF CISE Challenge #EIA–9726388, NSF Career Award #CCR–0049086, and NIH National Center for Research Resources #P41 RR06009

[2] Supported by the NSF Career Award #CCR–0049086, and by NSF grants CISE Challenge #EIA–9726388, Research Infrastructure #EIA–9972853, and ITR grant #ACI–0085969.

[3] Supported by NSF ITR #ACI–0085969, NSF Challenges in CISE #EIA–9726388, NSF CISE Research Infrastructure #EIA–9972853, and NSF #CCR–9988519.

## 1 Introduction

The generation, distribution, and refinement of good quality tetrahedral meshes of 3D domains is a necessary procedure in the numerical solution of partial differential equations (PDEs) on parallel machines. A traditional approach to refining and distributing a mesh for a parallel field solver might proceed as follows:

(1) Sequentially generate a sufficiently dense mesh.
(2) Use a graph partitioning program (e.g. Metis [1]) to partition the mesh into $N = P$ submeshes, where $P$ is the number of processing elements (PEs) to be used in the simulation.
(3) Distribute the submeshes to the $P$ PEs of a parallel machine and execute the parallel field solver.
(4) Sequentially refine the mesh to satisfy solver error bounds.

Repeating these steps to refine the mesh based upon input from an iterative field solver is much less efficient than refining the mesh in parallel to preserve locality and eliminate expensive I/O.

We consider only the generation and refinement of *tetrahedral* meshes. Many of the concepts are readily applicable to unstructured triangular meshes, as well. Furthermore, we only consider the case that the mesh is constructed by *iterative point insertion*, since this is the most convenient type of Delaunay mesh generation. An iterative point insertion algorithm generates a series of meshes starting from an initial mesh by adding new points within the interior of the domain to enforce tetrahedron size and *quality* bounds. Tetrahedron quality varies depending upon the needs of the field solver and the capabilities of the mesh generation algorithm.

Alternatives to iterative point insertion exist, like sphere packing [2] or *a priori* conforming Delaunay tetrahedralization [3], both of which generate mesh vertices inside the domain before computing the mesh connectivity. Such algorithms, including the related algorithm due to Cohen–Steiner et al. [4], do not appear to be flexible enough to efficiently and adaptively refine a mesh in parallel. Note, though, that any of these algorithms could be used to generate an initial mesh as input to our sequential and parallel meshing algorithms if the result adheres to the constraints described in Section 1.2.

Our approach consists of two steps: (i) *sequential mesh initialization* (Section 2.2), and (ii) *parallel mesh refinement* (Section 3.3). In the initialization step, a sequential mesh generator computes a *conforming Delaunay mesh* of an input domain in which the angle between any two incident components is between $90^o$ to $270^o$ inclusive. Then, in the mesh refinement step, our parallel refinement algorithm is executed to refine a distributed, element–wise partitioning of the initial mesh. Note that, although the initial mesh may be uniform–density, the density of tetrahedra in the final mesh is controlled by the user (Section 2).

The most important feature of our algorithm is that the submesh interfaces are allowed to change as new vertices are inserted concurrently into the distributed mesh. After each vertex insertion, the Delaunay property of the distributed mesh is restored within each submesh and, when necessary, across submesh interfaces by a parallel Bowyer–Watson algorithm [5]. Consequently, we can prove (Section 3) that our parallel refinement algorithm terminates, and generates a new distributed Delaunay mesh containing tetrahedra whose circumradius to shortest edge (radius–edge) ratio is less than 2.

Our experimental data (Section 5) suggest that our parallel refinement algorithm is efficient in practice, even for certain domains whose boundaries are not theoretically admissible. Our experiments also show that 80% or more of the time spent blocking on communication is overlapped with useful computation, but at the cost of increased communication overhead of up to 20% of the total execution time. We show that even with this additional overhead, our experimental implementation (Section 4) can create and place large meshes up to 6 times faster than the traditional approach to generating and refining a distributed mesh. Further, experimental evidence suggests that our algorithm is also practically efficient and latency tolerant (Section 6). Even so, a great deal more must be done to develop a truly practical parallel mesh generator; we discuss theoretical and practical improvements we are exploring for deployment in future implementations (Section 7).

### *1.1 Related Work*

Domain decomposition (DD) is the most frequently used technique for parallelizing unstructured mesh generation computations. Löhner et al. [6] present a two–phase approach in which the submesh interiors are independently refined, then the inter–submesh (*interface*) regions, composed of elements that intersect vertices and edges shared with other submeshes, are refined. Said et al. [7] describe a similar approach in which the submeshes are refined independently using a highly optimized and well–tested sequential mesh generation algorithm, and then the submeshes are passed to a post–processor to smooth mesh nodes in the submesh interfaces. deCougny et al. [8] describe a parallel mesh generation and refinement method which uses a distributed octree data structure to help partition a domain and a corresponding surface mesh for distribution to the PEs for further processing. All of these approaches are practical, and have been used extensively in the literature.

Tetrahedron subdivision is one popular method for mesh refinement [9–11]. Typically, tetrahedra are subdivided into eight self–similar tetrahedra (when possible) by adding nodes to the edges of the tetrahedron. Oliker et al. [10] describe a tetrahedron subdivision algorithm in which the tetrahedra are subdivided into 2, 4, or 8 new tetrahedra, depending upon an error estimator for each edge of the original tetrahedron. The tetrahedron subdivision approach is very efficient with small communication overhead, particularly when a stable graph partitioning program (e.g. Metis [1]) is used to compute the initial mesh partitioning.

The first efforts on the parallelization of existing sequential mesh generation algorithms were reported by Chrisochoides et al. [12] and Okusanya et al. [13]. Both papers present parallelization methods for mesh generation based on Delaunay triangulation. Specifically, both papers present a parallelization of the Bowyer–Watson [14,15] kernel. The parallel Bowyer–Watson (PBW) algorithm of Chrisochoides et al. [12] is based on a combination of the data–parallel and the task–parallel models. The algorithm is data–parallel because the triangulation (or mesh) is treated as a distributed object built upon a hierarchy of distributed objects (edges and triangles); vertices are entirely "owned" by a PE. Task–parallelism is introduced by considering the computation as a collection of individual tasks that can be *created*, *deleted*, *suspended* and *scheduled* dynamically at run–time.

Input to the algorithm is a boundary conforming mesh distributed over $P$ PEs. Unlike the domain decomposition approaches described above, element creation is decomposed into several tasks: (i) locate, lock, and remove the elements which are non–Delaunay with respect to a new point (the Delaunay *cavity*), (ii) collect and lock the local and remote data required to compute the new elements, (iii) and connect the new point to the edges bounding the cavity to form new elements. Tasks like the cavity computation might abort because some elements are locked by other tasks. Upon completion, newly created objects are assigned to PEs by following a set of distribution rules (D–rules) which (uniquely) assign the new objects that access to non–local data can be minimized for future tasks.

Okusanya et al. [13,16] describe a parallelization of the Bowyer–Watson kernel using a digital tree [17] structure maintained in each PE containing a submesh. Shared boundaries between submeshes are composed of uniquely identified mesh edges (2D) or faces (3D) to allow the search for a Delaunay cavity to enter possibly many PEs. Elements are locked to prevent concurrent cavity searches from invalidly sharing triangles or tetrahedra. Their algorithm also locks shared submesh boundaries to prevent simultaneous invalid updating of shared edges or faces appearing in a cavity.

None of the methods above mathematically guarantee the quality of the final mesh, although the quality is usually sufficient for many applications. The first provably good parallel algorithm was described (without proof) by Chew et al. [18], who propose an efficient 2D parallel constrained Delaunay mesh generation algorithm which ensures quality of the refined mesh. The shared boundary of each submesh is fixed by the partitioning of the initial mesh, but can be refined by inserting new points then consistently updating the shared edges in the neighbor submesh. In general, however, constrained Delaunay triangulations do not exist in higher dimensions [19]; a straightforward generalization of their algorithm has yet to be discovered.

George et al. [20] present a (nearly) guaranteed–quality domain–decomposition based algorithm which heuristically partitions a surface mesh of the domain boundary. When successful, their algorithm places Delaunay surfaces to partition the domain such that the surfaces will appear in a Delaunay triangulation of the resulting subdomains. The subdomains can then be distributed to the PEs and refined independently in each PE by a guaranteed–quality algorithm like that proposed by Shewchuk [21,22]. However, if the chosen partitioning surfaces introduce new small angles into the domain boundary, the quality may be substantially worse than that guaranteed by the sequential algorithm.

Freitag et al. [23] describe and prove the algorithmic complexity of a framework for 2D mesh refinement and improvement using several local element operations, including Delaunay flipping. In each iteration of their algorithm, an independent set of nodes of the *interference graph* of conflicting operations is computed in parallel to yield a set of operations which can be completed concurrently and result in a correct distributed mesh. It is straightforward to apply their framework to the parallelization of guaranteed–quality 2D mesh refinement. The most intuitive extension of this framework to 3D Delaunay meshing may be to replace the 2D element–level operations with 3D Delaunay cavity computations [24].

## *1.2 Definitions*

Define a polyhedral domain $\Omega \subset R^3$ as a closed, bounded volume having arbitrary genus and connectivity so that passages, voids, and multiple (disjoint) regions may be represented. The boundary $\partial\Omega$ of $\Omega$ is a piecewise–linear 2–manifold without boundary composed of vertices, linear vertex–bounded segments, and segment–bounded polygonal facets (which need not be convex). $\partial\Omega$ must be closed under intersection, such that the intersection of any two components in $\partial\Omega$ is also in $\partial\Omega$.

If any two components in $\partial\Omega$ share one or more vertices, then they are *incident*; otherwise, they are *non–incident*. For each component $\omega \in \partial\Omega$ and any point $p \in \omega$, the *local feature size* $\texttt{lfs}(p)$ is the minimum of the straight–line distances from $p$ to points in the components of $\Omega$ not incident to $\omega$. Similarly, define $\texttt{lfs}_{min}(\Omega)$ as the minimum distance between any two non–incident components of $\partial\Omega$. In addition, the correctness proofs of our algorithms require that the angle separating any two incident components in $\partial\Omega$ is between $90^o$ and $270^o$ inclusive. Examples of valid and invalid domains appear in Figure 1.

Let $\mathcal{T} \subset R^3$ be a triangulation composed of vertices, edges, faces, and tetrahedra. A sphere in $R^3$ whose boundary passes through the vertices of an edge, face, or tetrahedron in $\mathcal{T}$ is called a *circumsphere*. Infinitely many spheres circumscribe a vertex, an edge, or a face; exactly one sphere circumscribes the vertices of a tetrahedron. An edge, face, or tetrahedron $\phi \in \mathcal{T}$ is *Delaunay* if some circumsphere of $\phi$ encloses no vertex of $\mathcal{T}$ other than those forming $\phi$; $\phi$ is said to have the *Delaunay property* (each vertex of $\mathcal{T}$ trivially has the Delaunay property). $\mathcal{T}$ is a Delaunay triangulation only if every tetrahedron (or every face) in $\mathcal{T}$ has the Delaunay property.

Let a mesh $\mathcal{M} = \mathcal{M}(\mathcal{K}, \mathcal{D}, \mathcal{V})$ be a set of the following three triangulations:

(1) $\mathcal{K}$, a skeleton triangulation consisting of the vertices and edges (*subsegments*) created to refine the segments of $\partial\Omega$,
(2) $\mathcal{D}$, a Delaunay surface triangulation consisting of the vertices in $\mathcal{K}$, and the vertices and triangular faces (*subfacets*) created to refine the facets of $\partial\Omega$, and
(3) $\mathcal{V}$, a Delaunay triangulation containing the vertices in $\mathcal{K}$ and $\mathcal{D}$, and the vertices and tetrahedra created to refine the volume of $\Omega$.

Call the *diametral* sphere of a subsegment $s$ the unique smallest sphere circumscribing its vertices, with center and diameter as the midpoint and length of $s$, respectively. $s$ is *encroached* if some vertex $v$ in $\mathcal{V}$ lies inside or on the diametral sphere of $s$. Similarly, the *equatorial* sphere of a subfacet $f$ is the unique smallest sphere circumscribing its vertices, with center and radius as the planar circumcenter and circumradius of $f$. $f$ is *encroached* if a vertex $v$ in $\mathcal{V}$ lies inside or on the equatorial sphere of $f$. If the subsegments in $\mathcal{K}$ and the subfacets in $\mathcal{D}$ are unencroached by the vertices in $\mathcal{V}$, then the Delaunay property ensures that each subsegment and subfacet appears as an edge or a face (respectively) of some tetrahedron in $\mathcal{V}$. In this case, $\mathcal{M}$ is called a *conforming Delaunay triangulation* of $\Omega$.

If the circumradius–to–shortest–edge ratio, $ratio(t)$, of every tetrahedron $t \in \mathcal{V}$ is less than 2, then $\mathcal{M}$ is a *good–quality* mesh composed of *well–shaped* tetrahedra. Note that well–shaped tetrahedra whose vertices are coplanar (or nearly so) may appear in the mesh; such tetrahedra are called *slivers*.

## 2 Sequential Delaunay Refinement

In general, provable–quality Delaunay meshing algorithms are *iterative point insertion* algorithms, in that they begin with empty surface and volume Delaunay triangulations, and construct a refined mesh $\mathcal{M}$ by inserting new vertices into both the surface and/or the volume. After a new vertex $v$ is added into $\mathcal{M}$, the Delaunay property of the mesh is restored by first removing tetrahedra whose circumspheres enclose (or *conflict* with) $v$, then replacing them with tetrahedra which are Delaunay with respect to $v$ and to all existing vertices in $\mathcal{M}$.

The two most common algorithms for maintaining the Delaunay property of an initial Delaunay triangulation after inserting a new point are the *flip* [25,26] and Bowyer–Watson [14,15] algorithms. We have chosen the Bowyer–Watson algorithm as the basis of our parallel refinement algorithm [5], since it is fairly straightforward to parallelize [5].

Here, we describe both the sequential Bowyer–Watson algorithm (Section 2.1) and a sequential mesh initialization procedure (Section 2.2) which generates input suitable for our parallel Delaunay refinement algorithm. We then prove that the sequential algorithm from which our parallel algorithm is derived both terminates and outputs a mesh containing only well–shaped tetrahedra (Section 2.3).

### *2.1 The Bowyer–Watson Algorithm*

Let $\mathcal{T}_i \subset R^3$ be an initial Delaunay triangulation enclosing all points to be inserted, and let $p_{i+1} \notin \mathcal{T}_i$ be a new point to be added into $\mathcal{T}_i$. The sequential Bowyer–Watson algorithm generates a new Delaunay triangulation $\mathcal{T}_{i+1}$ by replacing the tetrahedra in $\mathcal{T}_i$ conflicting with $p_{i+1}$ with Delaunay tetrahedra whose apex is $p_{i+1}$. This process is normally implemented in four steps (see Figure 2 for a 2D example):

(1) *Point location*: Find an initial tetrahedron $t \in \mathcal{T}_i$ that conflicts with $p_{i+1}$.
(2) *Cavity search*: Perform a depth–first search from $t$ over the faces of $\mathcal{T}_i$ to compute $\mathcal{C}_{i+1} \subseteq \mathcal{T}_i$, the set of all tetrahedra which conflict with $p_{i+1}$ ($\mathcal{C}_{i+1}$ is called the *cavity* of $p_{i+1}$).
(3) *Cavity deletion*: $\mathcal{T}_i = \mathcal{T}_i - \mathcal{C}_{i+1}$, leaving a face bounded polyhedral "hole" in $\mathcal{T}_i$.
(4) *Cavity retriangulation*: Let $\mathcal{B}(\mathcal{C}_{i+1})$ be the set of faces in the boundary of $\mathcal{C}_{i+1}$; then $\mathcal{T}_{i+1} = \mathcal{T}_i \cup \{t \mid t = f \cup p_{i+1} \text{ and } f \in \mathcal{B}(\mathcal{C}_{i+1})\}$.

For our Delaunay mesh refinement algorithm, point location is not required to find $t$ since the algorithm places new vertices within the circumsphere of a known tetrahedron; point location may be required to generate an initial mesh, depending upon the algorithm chosen to generate the initial Delaunay triangulation. Note that a flip–based retriangulation algorithm would require point location over the tetrahedra in $\mathcal{C}_{i+1}$. This search could be implemented as *jump–and–walk* [27], or even brute–force search, since the size of $\mathcal{C}_{i+1}$ is expected to be small for most point insertions.

Because of the Delaunay property, each tetrahedron in $\mathcal{C}_{i+1}$ must share a face with at least one other tetrahedron in $\mathcal{C}_{i+1}$. Tetrahedra not in $\mathcal{C}_{i+1}$ which share at least one face with a tetrahedron in $\mathcal{C}_{i+1}$ are called *closure* tetrahedra (Figure 3 shows a 2D example). The union of these closure tetrahedra and $\mathcal{C}_{i+1}$ is called the cavity *closure*, $\bar{\mathcal{C}}_{i+1}$. The concept of the cavity closure is an integral part the correctness proof for our parallel algorithm (Section 3).

### *2.2 Generating a Mesh for Delaunay Refinement*

Input to our sequential and parallel algorithms is a conforming Delaunay mesh $\mathcal{M}_o = \mathcal{M}_o(\mathcal{K}_o, \mathcal{D}_o, \mathcal{V}_o)$, where $\mathcal{K}_o$, $\mathcal{D}_o$, and $\mathcal{V}_o$ are initial skeleton, surface and volume Delaunay triangulations, respectively. Let $d = \texttt{lfs}_{min}(\partial\Omega)/\sqrt{2}$. $\mathcal{M}$ must have the following properties:

(1) Each segment of $\partial\Omega$ appears as a union of subsegments in $\mathcal{K}_o$.
(2) Each facet of $\partial\Omega$ appears as a union of subfacets in $\mathcal{D}_o$.
(3) The subsegments in $\mathcal{K}_o$ appear as edges of subfacets in $\mathcal{D}_o$ and as edges of tetrahedra in $\mathcal{V}_o$.
(4) The subfacets in $\mathcal{D}_o$ appear as faces of tetrahedra in $\mathcal{V}_o$.
(5) For each $s \in \mathcal{K}_o$, $d \leq |s| < 2d$.
(6) For each $f \in \mathcal{D}_o$, $radius(f) < d\sqrt{2}$, where $radius(f)$ is the radius of the equatorial sphere of $f$.

$\mathcal{M}_o$ can be constructed with Algorithm 1. We have chosen $d$ such that no vertex in $\mathcal{T}_o$ lies inside or on the diametral or equatorial sphere of any subsegment or subfacet, respectively. Because of the Delaunay property, this is enough to ensure that $\mathcal{M}_o$ is a conforming Delaunay mesh (i.e. $\mathcal{M}_o$ contains all of the subsegments in $\mathcal{K}_o$ and all of the subfacets in $\mathcal{D}_o$).

It is clear from the definition of $d$ that $\mathcal{M}_o$ may contain many more tetrahedra than necessary to tile $\partial\Omega$ (see Figure 4 for an example). However, because the length of an edge in the mesh is proportional to the local feature sizes at its endpoints, a mesh consisting solely of tetrahedra with bounded radius–edge ratio may anyway require a large number of tetrahedra. Furthermore, it is reasonable[4] to expect that at least an order of magnitude more tetrahedra will be generated by our parallel refinement algorithm, thus lessening the effects of an overly dense initial mesh (Section 5).

## *2.3 A Sequential Delaunay Refinement Algorithm*

The correctness proof of our sequential refinement algorithm (Algorithm 2) depends upon a user–provided constant $h$, where $0 < h < d$, to specify the minimum length of any edge (and thus the size of the smallest tetrahedron) allowed in the final mesh. Note that the constant $d$ is used only to generate a suitable initial coarse mesh to be partitioned and distributed to the parallel PEs for refinement; for domains without very small features, we expect $h \ll d$.

A user can specify $h$ in one of two ways: (i) explicitly, for example by specifying a density function over the domain, or (ii) implicitly, by requesting the refinement of tetrahedra near an "interesting" feature of a solution computed by a field solver. In the implicit case, $h$ does not take on a specific value in the algorithm or the proof; rather, the *existence* of $h$ is more important, as this allows us to prove both the termination and the quality guarantees.

Our algorithm can potentially introduce *slivers* into the mesh. Several strategies exist [28–31] to prevent or eliminate slivers in a Delaunay mesh with bounded radius–edge ratio, although the best strategy is a subject of ongoing research.

To show that this algorithm is correct, we need to prove that each edge inserted into the mesh has length greater than $h$. We prove by induction that (i) the algorithm terminates with all mesh edges longer than $h$, and that (ii) upon termination, $ratio(t) < 2$ for every tetrahedron in the mesh.

Consider the following invariants:

(1) Each refined subsegment has length greater than $2h$.
(2) Each refined subfacet has circumradius greater than $h\sqrt{2}$.
(3) Each refined tetrahedron has circumradius greater than $2h$.

We will prove *SeqRefinement* is correct by showing that these invariants hold throughout the course of the algorithm.

**Lemma 1.** *In algorithm* SeqRefinement*, the above invariants hold after each iteration through the main loop.*

[4] Parallel meshing is justified for generating meshes containing tens to hundreds of millions of elements or more.

*Proof.* Assume that the initial has the properties guaranteed by *SeqInitialization* (Algorithm 1); each invariant therefore holds before the start of *SeqRefinement* (Algorithm 2).

*Invariant 1.* Assume that the first failure occurs because a subsegment $s$ in segment $S$ is encroached upon by some vertex $v$, and $|s| \leq 2h$. $v$ cannot lie on a segment different from $S$ due to the angle and distance restrictions; hence, $v$ cannot be a subsegment midpoint or input vertex. $v$ must therefore be the circumcenter of either a subfacet or a tetrahedron. $\mathcal{V}$ is Delaunay, so the sphere centered at $v$ cannot enclose the endpoints of $s$. Further, this sphere has radius more than $h\sqrt{2}$, otherwise one of the other two invariants would have already failed to hold. As a result, the diametral sphere of $s$ cannot contain $v$, which contradicts the assumption.

*Invariant 2.* Assume that the first failure occurs because a subfacet $f$ in facet $F$ is encroached upon by some vertex $v$, and $radius(f) \leq h\sqrt{2}$. The angle and distance restrictions ensure that vertices inserted onto any segment not bounding $F$ cannot encroach upon subfacets in $f$, so $v$ cannot be a subsegment midpoint. Further, due to the angle and distance restrictions, $v$ can lie on a facet $G$ only if $F$ and $G$ share a segment (say $S$). $f$ is Delaunay, so the intersection of $G$ and the equatorial sphere of $f$ is a circle in the plane of $G$ enclosed by the diametral sphere of some subsegment $s$ on $S$. However, this is a contradiction since $v$ would also encroach upon $s$, and the algorithm would have refined $s$ instead of $f$.

The only remaining case is that $f$ is encroached upon by a tetrahedron circumcenter. $f$ is the first failure, so $radius(t) > 2h$. Because $\mathcal{V}$ is Delaunay, the sphere centered at $v$ cannot enclose the vertices of $f$, so the vertices of $f$ are farther than $2h$ from $v$. Hence, $v$ can lie in the equatorial sphere of $f$ only if $f$ is an obtuse triangle. If this occurs, then the subfacet incident on the obtuse edge of $f$ is a larger–circumradius subfacet which is also encroached by $v$. But, this is a contradiction, since the algorithm refines the largest–circumradius subfacet encroached by $v$.

*Invariant 3.* Finally, assume the first failure occurs when some tetrahedron $t$ is refined, and $radius(t) \leq 2h$. $ratio(t) \geq 2$—otherwise $t$ would not have been refined—so the length of the shortest edge $e$ of $t$ is $h$ or less. Since this is assumed to be the first failure of the invariants, every edge in the current Delaunay triangulation is the result of refining a subsegment of length greater than $2h$, a subfacet of circumradius greater than $h\sqrt{2}$, or a tetrahedron of circumradius greater than $2h$. Thus the edge $e$ must have length greater than $h$. This is a contradiction.

Since any failure of the invariants leads to a contradiction, the invariants must hold throughout the execution of the algorithm. □

**Theorem 2.** *Algorithm* SeqRefinement *terminates, and $\mathcal{M}$ has the following properties:*

*(1) The length of every edge in $\mathcal{M}$ is greater than $h$.*
*(2) The radius–edge ratio of every tetrahedron in $\mathcal{M}$ is less than* $2$.

*Proof.* From Lemma 1, we know that no two vertices of the mesh are ever closer than $h$. Since only finitely many such vertices can be placed within a finite volume, the algorithm must terminate. It is clear from the algorithm that termination implies that the radius–edge ratio of each tetrahedron in $\mathcal{M}$ is less than 2. □

**Remark 3.** Subsegments and subfacets in $\mathcal{D}$ are unencroached in $\mathcal{V}$ before each new vertex is inserted.

This follows from Lemma 1. If a subsegment or subfacet becomes encroached by an existing mesh vertex, the distance between the encroaching vertex and a subsequently inserted vertex could be less than $h$, contradicting the proof of Lemma 1.

**Remark 4.** The correctness of *SeqRefinement* does not depend on the order in which any two vertices are inserted into $\mathcal{M}$.

This follows from the algorithm, Remark 3, and Lemma 1. Note that the proof of Lemma 1 defines an order only within a single iteration of *SeqRefinement*, and that the algorithm inserts exactly one vertex into the mesh per iteration. Up to two other vertices may be tested for insertion in the same iteration, but these temporary vertices do not cause a change in the mesh. With Remark 3, this implies that a vertex inserted in one iteration of the algorithm only influences the choice of tetrahedra to refine in subsequent iterations. Further, no order is specified by the algorithm for refining tetrahedra, so Lemma 1 holds regardless of the order in which any two iterations of *SeqRefinement* (and thus any two new vertex insertions or the corresponding cavity retriangulations) are processed.

These facts are the cornerstones of our correctness proof of the *ParRefinement* algorithm, which we describe next.

## 3 Parallel Delaunay Refinement

Our parallel refinement algorithm is designed for practical use, so we impose few restrictions on the programming model adopted in an implementation. We assume *non–blocking* message passing, in which each processing element (PE) of the parallel system operates independently on distributed data structures to accomplish the common goal of refining a guaranteed–quality Delaunay mesh. We also assume that an initial mesh, $\mathcal{M}_o = \mathcal{M}_o(\mathcal{K}, \mathcal{D}, \mathcal{V})$, has been generated with the properties described in Section 2.2. Finally, $\mathcal{M}_o$ must be partitioned and distributed to the PEs in the parallel system as described below:

(1) *Initial domain decomposition*: $\mathcal{M}_o$ has been element–wise partitioned into $N$ submeshes $S_0 \ldots S_{N-1}$, and distributed to $P$ PEs, where submesh $S_k = S(\mathcal{K}_k, \mathcal{D}_k, \mathcal{V}_k)$ is refined in the memory of PE$_k$. Note that *overdecomposing* $\mathcal{M}_o$ such that $N \gg P$ can both improve memory utilization and simplify parallel programming for dynamic load–balancing [32]. Subfacets and tetrahedra in $\mathcal{M}_o$ are assumed to be owned by a single PE during the computation, although ownership may change as the mesh is refined. Each subfacet and the tetrahedron that contains it are always owned by the same PE.

(2) *Submesh connectivity*: if $S_j \cap S_k \neq \emptyset$, then $S_j$ and $S_k$ are *adjacent* and $\mathcal{I}_{j,k} = S_j \cap S_k$. If $\mathcal{I}_{j,k}$ contains faces, then $S_j$ and $S_k$ are *neighbors*, $\mathcal{I}_{j,k}$ is an *interface* surface, and faces in $\mathcal{I}_{j,k}$ are *interface* faces. $\mathcal{I}_j$ is the set of all interface faces in $S_j$. We assume that vertices, edges, and faces in $\mathcal{I}_{j,k}$ are replicated in any submeshes that contain them.

The interface surfaces (or just *interfaces*) induced by partitioning $\mathcal{M}_o$ may be discontiguous, and are allowed to change as new vertices are added into the mesh. Further, interfaces do not constrain the submesh or surface triangulations; new vertices inserted into one submesh can affect the submesh or surface triangulation in another submesh. Our algorithm allows the cavities for new vertices to be computed and retriangulated concurrently (Section 3.1), but only if their closures do not share tetrahedra (Section 3.2). When two or more cavity closures share tetrahedra, a *setback* occurs, introducing additional overhead not present in a sequential algorithm. Setbacks hinder the progress of the algorithm, and cause computation to be restarted. Even so, that interface surfaces can change dynamically is both a fundamental basis for the quality guarantees of our parallel refinement algorithm (Section 3.3), and a useful tool to help load balance the refinement process [33].

## *3.1 Computing a distributed cavity*

Because $\mathcal{M}_o$ is a distributed mesh, tetrahedra which conflict with a new vertex may be contained in many submeshes. Let $v$ be a new vertex, and let $\mathcal{C}$ be the cavity of $v$. If $\mathcal{C}$ contains no interface faces, then $\mathcal{C}$ is a *local* cavity; otherwise, $\mathcal{C}$ is *distributed* cavity, and two or more submeshes contain the subsets (or *subcavities*) of $\mathcal{C}$. From the properties of the cavity, $\mathcal{C}$ is face–connected, so at least two subcavities $c_j \subseteq S_j$ and $c_k \subseteq S_k$, with boundaries $b(c_j)$ and

$b(c_k)$, share an interface face. A parallel depth– or breadth–first graph search algorithm can therefore be used to find the subcavities of $\mathcal{C}$ by visiting the interface faces which connect them.

We have chosen to implement parallel breadth–first search (Figure 5) to compute a distributed cavity. Let $v$ be a vertex chosen for insertion into $S_0$, and let $c_0$ be the initial subcavity computed by sequential depth–first search in $S_0$:

**Parallel breadth–first search**: Execute the following in $\text{PE}_0$:

(1) For each $S_k$, where $\mathcal{I}_k \cap b(c_0)$ contains unmarked interface faces, let $c \subseteq c_k$ be the subset of the subcavity $c_k$ found by depth–first search from each tetrahedron in $S_k$ containing an unmarked interface face in $\mathcal{I}_k \cap b(c_0)$.
(2) For each set of tetrahedra $c$ found in (1), mark the unmarked interface faces in $b(c_0) \cap b(c)$, add any remaining unmarked interface faces in $b(c)$ to $b(c_0)$, and copy $c$ into $c_0$.
(3) If no new sets were found in (1), then the search is terminated; let $\mathcal{C} = c_0$. Otherwise, continue at (1) in $\text{PE}_0$.

Upon termination, $\mathcal{C}$ is the union of all subcavities containing tetrahedra conflicting with $v$. Note that the subcavities of $\mathcal{C}$ may contain tetrahedra which are connected to $\mathcal{C}$ by a single interface face; multiple searches may be required to locate all tetrahedra in the subcavities $c_k$ (see Step 1). In Step 2, conflicting tetrahedra are copied into $c_0$ to find interface faces in the distributed cavity which have not yet been marked. If a subcavity search is initiated in $\text{PE}_0$, no additional copying is needed. If $\bar{\mathcal{C}}$ does not share tetrahedra with any other cavity closure, the tetrahedra in $\mathcal{C}$ can be added to $S_0$ and sequentially retriangulated. The conflicting tetrahedra in participating submeshes must also be removed, and the interfaces which changed due to migrating tetrahedra must be updated.

## *3.2 Problems Due to Concurrency*

Retriangulating any local or distributed cavity which intersects another cavity can result in a non–simplicial or non–Delaunay mesh. Consider the two following scenarios arising from inserting the new vertices $p$ and $q$; $t$ is a tetrahedron and $f$ is a face:

(1) *Overlapping cavities*: $\exists t \in \mathcal{C}_p \cap \mathcal{C}_q$; if the cavities are retriangulated, the result will not be simplicial, since some new tetrahedra will be pierced by edges (a 2D example appears in Figure 6).
(2) *Neighboring cavities*: $\exists f \in \mathcal{B}_p \cap \mathcal{B}_q$; if the cavities are retriangulated, one or more faces in the boundaries of the cavities may conflict with $p$ and $q$.

(a 2D example appears in Figure 7).

In both cases, the closures of the two cavities must also overlap. We can therefore avoid both problems and ensure a correct mesh if we ensure that the closures of any two concurrently computed cavities do not overlap.

**Remark 5.** Let $\mathcal{T} \subset R^3$ be a Delaunay triangulation, and let $p$ and $q$ be two vertices, such that $p \neq q$ and $p, q \notin \mathcal{T}$. If $\bar{\mathcal{C}}_p$ and $\bar{\mathcal{C}}_q$ do not overlap, then the triangulation $\mathcal{T}''$ resulting from retriangulating $\mathcal{C}_p$ and $\mathcal{C}_q$ is Delaunay.

This follows from the Delaunay property, and from the properties of the cavity.

**Remark 6.** If two cavities $\mathcal{C}_p$ and $\mathcal{C}_q$ are retriangulated by the Bowyer–Watson algorithm, and $\bar{\mathcal{C}}_p$ and $\bar{\mathcal{C}}_q$ do not overlap, the same Delaunay triangulation results regardless of the order in which the cavities are retriangulated. [5]

This follows from the Bowyer–Watson algorithm [14,15] and Remark 5.

A straightforward solution to prevent two concurrently computed cavities from overlapping is to *lock* tetrahedra in the closure of a cavity as the cavity is being computed. Then, any other cavity search which tries to acquire a locked tetrahedron should be terminated, the offending cavity should be discarded, and the PE initiating the search should consider a new vertex to insert. This simple scheme is essentially a dynamic parallel (not necessarily maximal) independent set algorithm [34] which ensures that the closures of any two retriangulated cavities do not overlap. Therefore, from Remark 5, retriangulating a cavity concurrently with any other non–overlapping cavities results in a Delaunay triangulation. Since the closure can be computed as a side–effect of the depth–first cavity search (Section 4), this strategy introduces little additional work into our parallel algorithm.

Note, though, that this simple algorithm can *livelock*. Two PEs could initiate overlapping distributed cavity searches, neither of which is able to complete because each PE locks tetrahedra required to complete the search initiated by the other PE. In this case, both cavity searches will be terminated, and each PE will attempt to reinsert the corresponding vertices. The two PEs may then repeatedly compete for the same locked tetrahedra, thus preventing termination. It may be worthwhile to note that, in our experiments, we have never encountered an occurrence of livelock. This is not surprising; the conditions required to initiate livelock are unlikely to occur due to the inherent "randomness" of *ParRefinement*.

Because the state of a cavity can be distributed across many PEs, designing a deterministic algorithm to prevent livelock is not as straightforward as it might

[5] Note that this result does not require the assumption of general position.

seem, and such a discussion is outside the scope of this paper. Instead, a simple heuristic can be used to reduce the likelihood that livelock will occur. If a PE has only a single new vertex to insert, and there are no outstanding remote cavity requests to service, then the PE attempts to insert the vertex. If the attempt fails because of locked tetrahedra, and there are still no outstanding remote cavity requests to process, then the PE busy–waits for a short random time before trying to insert the vertex again.

### *3.3 A Parallel Delaunay Refinement Algorithm*

Algorithm 3 presents our parallel refinement algorithm, *ParRefinement*. We assume that *parallel-breadth-first-search* returns a possibly distributed cavity $\mathcal{C}_p$. If $\bar{\mathcal{C}}_p$ overlaps any other closure, then this procedure returns an empty cavity. Otherwise, *parallel-breadth-first-search* ensures that $\mathcal{C}_p$ is contained in $S_k$, and that any copies of tetrahedra in $\mathcal{C}_p$ have been removed from other submeshes. Note that closure tetrahedra do not need to be removed, and can be used to help update the submesh interfaces.

To show that *ParRefinement* is correct, it is enough to prove that Theorem 2 holds for *ParRefinement*:

**Theorem 7.** *Algorithm* ParRefinement *terminates, and $\mathcal{M}$ has the following properties:*

*(1) The length of every edge in $\mathcal{M}$ is greater than $h$.*
*(2) The radius–edge ratio of every tetrahedron in $\mathcal{M}$ is less than* 2*.*

*(i.e. Theorem 2 is also correct* ParRefinement*).*

*Proof.* By Remark 4, if no two cavities are retriangulated concurrently, then $\mathcal{M}$ has the desired properties. For any two concurrently computed cavities which do not overlap, such as those returned from *breadth–first–parallel–search*, Remark 6 ensures that the resulting mesh is the same as if they were retriangulated sequentially. This implies that Remark 4 is valid for concurrently retriangulated cavities; therefore, by construction, *ParRefinement* is correct. □

## 4 Implementation

Implementing robust, efficient, and correct mesh generation algorithms is extremely challenging, even for sequential algorithms. For example, Shewchuk describes data structures in the context of his guaranteed–quality mesh generation algorithms [35]. Similarly, Bourachaki and George [36] describe data structures specifically designed for fast computation of Delaunay triangulations by the Bowyer–Watson algorithm. We have instead implemented data structures which represent the full connectivity of the mesh, for the benefits of both ease of use and clarity (see, for example, Remacle et al. [37]). Lessons learned from this first implementation of the parallel algorithm will help us fine–tune future implementations.

The run–time system for our implementation consists of the *Data Movement and Control Substrate* (DMCS) [38] and the *Mobile Object Layer* (MOL) [32] libraries. The remote data gathering required by the parallel Bowyer–Watson algorithm is both unstructured and unpredictable, making implementation with a binary message–passing paradigm, like that supported by MPI, both tedious and error–prone. The implementation is more natural and thus substantially less complex with the one–sided message passing support of DMCS. The MOL adds a small layer above DMCS to provide a global namespace and automatic message forwarding in support of dynamic load balancing. Our implementation supports dynamic load balancing [32,33], but this important topic is outside the scope of the current discussion.

We generate an initial Delaunay mesh consisting of the skeleton, surface, and volume triangulations described in Section 1.2 using the sequential initialization algorithm described in Section 2.2. The mesh is next partitioned into $N$ submeshes with, for example, Parallel Metis [1]. The resulting partitions are stored into separate files in a file system visible from the processing elements (*PEs*) of the parallel system. Once the parallel mesher is started, each of the $N$ instances of the mesher loads a single submesh from the appropriate file into a local submesh data structure (Section 4.1) in preparation for parallel refinement. Once the PEs are synchronized, each PE begins executing the main loop of *ParRefinement*.

The implementation of the Bowyer–Watson kernel [5] deals with both local and distributed cavities. The computation and retriangulation of local cavities require no communication, so a sequential Bowyer–Watson implementation suffices (Section 4.2). In the case of distributed cavities, we have implemented a straightforward parallel breadth–first search algorithm (Section 4.2) to compute a distributed cavity. A parallel depth–first search algorithm could also be implemented, although at the expense of some complexity and possibly reduced performance due to (i) the more complex maintenance of the distributed state of a cavity, and (ii) the additional messages required to determine cavity completion when two subcavities share unvisited interface faces (e.g. the shaded face between subcavities D and E in Figure 5 would require an additional message exchange between the PEs containing D and E to determine if the face has been visited).

The parallel breadth–first search algorithm collects the tetrahedra in a distributed cavity into a single submesh for retriangulation and redistribution using a *Simultaneous Mesh Generation and Partitioning* (SMGP) [33] algorithm. SMGP is a fine–grain dynamic load balancing scheme designed to improve performance of parallel Delaunay mesh refinement. SMGP incrementally optimizes an objective function with parameters modeling both estimated and real characteristics of the submeshes during refinement. SMGP redistributes tetrahedra resulting from retriangulating a distributed cavity; coupling the SMGP implementation with the parallel Bowyer–Watson kernel [5] implementation is fairly straightforward.

In the following sections, we describe the two primary constructs of the parallel mesh generator: (i) distributed mesh data structures, and (ii) parallel breadth–first search.

## *4.1 Distributed Mesh Data Structures*

Our mesh data structures model the "theoretical" mesh structure described in Section 1.2, although we combine the skeleton and surface triangulations to simplify retriangulating a Delaunay cavity when it intersects the surface mesh. The surface triangulation consists of vertices, edges, and faces which are connected by pointers to tetrahedra in the volume. Similarly, the volume triangulation contains vertices, edges, faces, and tetrahedra, some of which are shared with the surface triangulation data structure. In each submesh, we maintain the full connectivity of the volume and surface triangulations via lists and pointers. In particular, this type of data structure simplifies operations which query the structure of the mesh, such as listing the faces incident on an interior mesh edge.

We represent a submesh as four hash tables to record vertices, edges, faces, and tetrahedra. Each vertex is assigned a globally unique integer identifier, which makes it possible to uniquely identify each edge, face, and tetrahedron by sorting its vertices. We use a hash key which is simply an arithmetic combination of the vertex identifiers of the entity to be inserted into a hash table. For example, a hash key $K$ for an edge with vertices identified by integers $i_1$ and $i_2$ (assume $i_2 < i_1$) could be computed as $K = ((k \cdot i_1) \oplus i_2)$, where $\oplus$ represents arithmetic exclusive–or, and $k$ is some prime number, say less than 31 (we have used 7, 11, and 31 with no apparent difference in efficiency).

Edges and faces may be "flagged" to distinguish between subsegments and interior edges, and between subfacets, interface faces, and interior faces. Subsegments are the only edges which have pointers to incident subfacets, in addition to the normal list of incident faces. This facilitates searching the surface triangulation for conflicting triangles after a subsegment is refined. Similarly, a subfacet is the only face with one assigned pointer to an adjacent tetrahedron, and the only face which records an index into the array of facets comprising the boundary of the input domain.

We use interface faces to demarcate the shared surface between two submeshes. Each interface face in one submesh contains a "mobile pointer" referring to the submesh containing a copy of the face. This reference is used during distributed cavity expansion as the target of a *mol_message* [32]. Other than this additional field, interface faces are treated as if they are interior faces for the purposes of computing and retriangulating a distributed cavity.

On each PE, we maintain a list of local tetrahedra which have been marked for refinement. The list is processed by the main loop of *ParRefinement* executing on the PE. Due to the Delaunay property, subsegments and subfacets which would become encroached upon by a new vertex can be detected by searching the boundary of the corresponding Delaunay cavity. Since neither encroached subsegments nor encroached subfacets are introduced into the mesh when a new vertex is inserted (Remark 3), no other lists are required.

## *4.2 Implementing Parallel Breadth–First Search*

With the data structures described above, the sequential Bowyer–Watson algorithm (Section 2.1) is straightforward to implement. We are concerned only for the implementation of cavity search and cavity retriangulation, since point location is not needed to insert new vertices during refinement. The cavity search can be implemented as a simple depth–first search over the topological dual graph of the mesh, in which a node represents a tetrahedron, and an edge between two nodes represents the face–adjacency of the corresponding tetrahedra.

The search is initialized with a dual–graph node $r$ whose corresponding tetrahedron conflicts with a new point $p$ to be inserted into the mesh. $r$ is marked at the start; the search is initiated by recursively examining the nodes adjacent to $r$ whose corresponding tetrahedra conflict with $p$. The following two rules apply for these and subsequently examined nodes:

(1) A node $n$ of the mesh dual graph is marked only if (i) $p$ is enclosed by the circumsphere of the corresponding tetrahedron, and (ii) there is some edge connecting $n$ to some previously marked or visited node.
(2) A node is visited only if all of its adjacent nodes have been marked or visited (i.e. nodes adjacent to $n$ must be examined before terminating the cavity search through $n$).

The leaves of the search tree thus generated correspond to closure tetrahedra (which would remain Delaunay if $p$ were inserted) and/or tetrahedra containing subfacets in the surface mesh. Furthermore, it is clear from the constraints on marked nodes that the cavity for $p$ must be an edge–connected subgraph of the mesh dual graph, implying that the cavity is a face–connected subset of the mesh. Termination is ensured because of the finite size of the mesh. Further, assuming infinite–precision arithmetic, the cavity contains all tetrahedra which conflict with $p$; to ensure that this claim holds under floating–point arithmetic, we use the adaptive–precision predicates designed and implemented by Shewchuk [39].

Our parallel breadth–first search algorithm has been implemented as a "wrapper" around the sequential depth–first search function described above. A locally expanded cavity may intersect one or more interfaces of the submesh, in which case a distributed cavity expansion process is initiated. We detect this condition by searching the bounding faces of the local cavity; if the cavity contains no interface faces or locked tetrahedra, it is retriangulated immediately. If locked cavity or closure tetrahedra were found, then the cavity is discarded and the refined tetrahedron which (directly or indirectly) gave rise to the new vertex is added again into the local list of tetrahedra marked for refinement.

Otherwise, a distributed cavity search is initiated by calling *parallel-breadth-first-search*. The following steps are executed by the PE trying to insert a new vertex into its local submesh (the *root* submesh for the cavity search):

(1) Initialize two hash tables to record the distributed cavity:
   - (a) Add local cavity and closure tetrahedra into a hash table which will contain the currently expanding cavity, and lock them.
   - (b) Add interface faces which are faces of cavity tetrahedra into a hash table containing new interface faces.

(2) While the cavity is incomplete, execute the following loop:
   - (a) For each submesh sharing new interface faces, send a *mol_message* containing the coordinates of the vertex to be inserted and the vertex indices of the shared interface faces.
   - (b) Block the cavity search and wait for all reply messages, which will contain either a set of zero or more cavity tetrahedra, or a failure flag.
   - (c) If all replies contain a set of zero or more cavity tetrahedra:
      - (i) Update the new interface face table:
         - (A) Remove interface faces contained in two new cavity tetrahedra, in an existing and a new cavity tetrahedron, or in an existing closure and a new cavity tetrahedron.
         - (B) Remove interface faces contained only in an existing cavity tetrahedron.
         - (C) Add each interface face contained only in a new cavity

tetrahedron, and sets its mobile pointer to that of the submesh which sent the new tetrahedron.
    - (ii) Add copies of new tetrahedra into the expanding cavity hash table, and lock them.
  - (d) Otherwise, if one or more replies contain a failure flag, exit from the while loop.
- (3) If the distributed cavity search completed successfully:
  - (a) For each submesh except the root for which an expansion message was sent, send a *mol_message* containing a cavity completion flag.
  - (b) Insert the vertex by retriangulating the distributed cavity and adding the new tetrahedra into the root submesh.
- (4) Otherwise, if the distributed cavity search was unsuccessful:
  - (a) For each submesh except the root for which a *mol_message* was sent, send a *mol_message* containing a cavity termination flag.
  - (b) Discard local cavity data structures and any copied tetrahedra and interface faces.

When a PE—including the initial PE—receives a distributed cavity search request, a local handler is called on the PE to execute the following steps:

- (1) Retrieve the appropriate subcavity data structures:
  - (a) Extract the coordinates of the new vertex from the message.
  - (b) If the request is received by the initial PE, retrieve the corresponding distributed cavity hash tables.
  - (c) On any other PE:
    - (i) If the message is a request for a new subcavity expansion, initialize and record a new hash table to contain the currently expanding subcavity.
    - (ii) Otherwise, retrieve the existing subcavity hash table.
- (2) For each interface face in the message:
  - (a) Look up the face in the local submesh face hash table by its vertex indices.
  - (b) Initiate a depth–first search from the tetrahedron incident on the face to find cavity and closure tetrahedra which are not already in the local subcavity.
- (3) If no locked tetrahedra were found:
  - (a) Add new cavity and closure tetrahedra into the subcavity hash table and lock them.
  - (b) Reply with a *mol_message* containing the coordinates and indices of the vertices of the new cavity tetrahedra, along with the corresponding data for any subsegments, subfacets, or interface faces contained in the tetrahedra.
- (4) Otherwise, if locked tetrahedra were found, reply with a *mol_message* containing a failure flag.
- (5) Continue local processing.

When a PE receives a message containing a completion or failure flag, another handler is executed:

(1) Retrieve the appropriate subcavity hash table.
(2) If a completion flag was received:
    (a) Remove subcavity tetrahedra from the submesh, leaving zero or more interior faces with only one incident closure tetrahedron.
    (b) For each interior face with one incident closure tetrahedron, replace it with an interface face whose mobile pointer is set to that of the root submesh.
(3) If a failure flag was received:
    (a) Unlock all subcavity and closure tetrahedra.
    (b) Clear the subcavity data structures and continue processing.

User– or system–level threads can be used to avoid blocking the PE when waiting for reply messages, but only if one local cavity (or subcavity) search cannot be interrupted by another cavity search. In our experimental implementation, we simulate the effects of user–level threads by explicitly storing the local portion of distributed cavity state and returning from *parallel-breadth-first-search* when remote subcavity searches from the initial PE are initiated (i.e. the blocking step in the procedure above). When all reply messages have been received by the initial PE, the local cavity state is "restored," and *parallel-breadth-first-search* is called to continue processing to handle the set of messages.

## 5 Performance

We have chosen two model problems (Figure 8): (i) a simple cube with a suspended cubical void (*cube-in-cube*), and (ii) a half–tee brace with theoretically inadmissible angles in its boundary (*tee*). Our parallel data were collected on a cluster of 16 SPARC Ultra2 333MHz machines, each with 512MB RAM, and connected by a 100MB/s fast ethernet switching fabric. Sequentially generated meshes were created on a 450MHz SPARC Ultra-80 with 4GB RAM, connected to a remote file server via 100MB/s fast ethernet. More detailed and specialized data pertaining to the parallel Bowyer–Watson kernel can be found in our companion paper [5].

The initial meshes passed as input to our implementation of *ParRefinement* (Section 3) contained about $100,000$ tetrahedra, and were partitioned with sequential Metis [1]. In each experiment, we also specified a maximum volume criterion to control the number of tetrahedra in the refined mesh. The meshes output by *ParRefinement* were uniform–density meshes (due to the volume criterion) containing 1 to 4 million tetrahedra with radius–edge ratio less than 2.

We consider latency to be the total time spent by a PE in broadcasting distributed cavity expansion messages (*scatter*) and receiving the corresponding replies (*gather*). In the absence of concurrency, scatter-gather operations block the originating PE, and the PE incurs the total overhead of all scatter-gather operations it initiates. Concurrency allows this otherwise wasted time to be partially hidden (or *tolerated*) – a PE can continue to initiate scatter-gather operations while blocked scatter-gather operations await completion.

Although it is difficult to compute the actual latency of a single scatter-gather operation, we can calculate in aggregate how effectively each PE overlaps the latency of scatter-gather operations with other useful work. Table 1 [6] demonstrates how well our algorithm tolerates the long, variable, and unpredictable latencies due to the communication overheads arising from concurrently inserting new vertices into various sizes of distributed meshes

Note that setbacks on a PE can only occur in the presence of incomplete scatter-gather operations (i.e. due to concurrency). The total time spent by a PE in partially completing and subsequently cleaning up setbacks (*Setback Time*, $T_S$) is therefore a good estimate of the time that could not be hidden by concurrency. Conversely, the total time spent successfully completing both local ($T_{M,L}$) and distributed ($T_{M,D}$) cavities in the presence of other incomplete gather-scatter operations is a good estimate of the time spent overlapping latency with computation (*Masked Time*, $T_M = T_{M,L} + T_{M,D}$). Finally, the percentage $L = T_M/(T_M+T_S)$ represents how effectively our algorithm can use the time spent waiting for incomplete distributed cavities to perform useful work (i.e. the latency tolerance). This data shows that our algorithm can tolerate more than 80% of the communication latency in our experiments.

The arrival of distributed cavity expansion messages is unpredictable, so the network must be polled frequently to reduce the lifetime of cavity search threads blocked on communication. This in turn reduces the likelihood of incurring a setback due to locked tetrahedra, thus reducing the time spent handling setbacks. However, 12% to 20% of the computation time (*% No Message*) is wasted due to polling when no message is available (or, *overpolling*). Even with this additional overhead, our experimental implementation can generate and place meshes 6 times faster than traditional techniques. Table 2 shows traditional versus parallel performance for 1 to 4 million tetrahedron meshes of the *cube-in-cube* domain on 16 PEs of a SPARC CoW.

[6] The time spent completing cavities when no other incomplete cavities were present has no effect on latency tolerance, so it is not shown here.

## 6 Conclusions

We have described a simple, provably correct parallel Delaunay mesh refinement algorithm for generating meshes of restricted polyhedral domains. Experiments show that our algorithm is both practically efficient and latency tolerant due to the fine–grain concurrency afforded by decomposing an initial mesh and independently refining the submeshes. Our algorithm is one of only a handful of available techniques for generating, placing, and refining a distributed mesh. Other methods include parallel advancing front [6], distributed octree [8], domain–decomposition [40], and master–worker domain–decomposition [7].

One of the main differences of our algorithm is that the distributed mesh triangulation is unaffected by the triangulation of the submesh interfaces. Our algorithm can therefore guarantee that the mesh is globally Delaunay, and that the elements are of the same quality as those generated by the sequential algorithm. Furthermore, our algorithm can be easily augmented with a fine-grain dynamic load-balancing strategy like SMGP [33] without sacrificing mesh quality.

An experimental implementation (Section 4) of our algorithm has been shown to mask more than 80% of the time spent in blocking on communication requests, although the cost is increased communication overhead due to polling the network to ensure timely reception of messages. Even so, our implementation has been shown to create and place large meshes up to 6 times faster than a typical approach to generating a distributed mesh.

## 7 Future Work

Several practical questions result from this work:

(1) *Is it possible to prove termination and guaranteed quality for parallel mesh refinement of inputs with small angles?*

We have positive results for the sequential case of meshing piecewise–linear complexes [2] with angles as small $70.5^o$ at the segment joining two facets, and we are exploring ways to generate meshes in the presence of even smaller angles. However, it is difficult to devise comparable parallel algorithms with a simple proof structure like that we describe here; more work is needed in this area.

(2) *Can the computational complexity of this or a similar algorithm be determined?*

We will explore ways to modify our algorithm to make it deterministic to enable a simpler complexity analysis. One such method includes iteratively computing and retriangulating maximal independent sets of Delaunay cavities [24]. Such a strategy should allow a fairly straightforward derivation of the computational complexity.

(3) *Does our proof structure work for sliver–preventing algorithms like the randomized algorithms proposed by Chew [28] and Li [29], or the weighted Delaunay algorithm proposed by Cheng and Dey [31]?*

The answer is certainly positive, and our proofs and parallel implementation will appear in future work.

(4) *How can we reduce the overhead due to overpolling without increasing setbacks and decreasing latency tolerance?*

Reducing overpolling may be as simple as polling every $n^{th}$ iteration through the main loop (Algorithm 3), for some $n > 1$. Another more complicated option is to use asynchronous interrupts rather than polling. This would require supporting either (1) local message queues (a simple change, but requires polling into local memory) or (2) multithreading the mesh generator (a complex change, but more natural for interrupt-based message reception). Clearly, more work is necessary to determine the "optimal" polling strategy.

## Acknowledgements

This work was performed using computational facilities at the College of William and Mary, which were enabled by grants from the National Science Foundation (EIA-9977030) and Sun Microsystems (SAR # EDU00-03-793, EDU-02-Q1-197). We would also like to thank the reviewers for their helpful suggestions for improving the original version of this paper.

## References

[1] G. Karypis, K. Schloegel, V. Kumar, PARMETIS—parallel graph partitioning and sparse matrix ordering library, Version 2.0, University of Minnesota,

Minneapolis, MN (1998).

[2] G. L. Miller, D. Talmor, S.-H. Teng, N. Walkington, H. Wang, Control volume meshes using sphere packing: generation, refinement, and coarsening, Fifth International Meshing Roundtable (1996) 47–61.

[3] M. Murphy, D. M. Mount, C. W. Gable, A point placement strategy for conforming Delaunay tetrahedralization, in: 11th Annual ACM–SIAM Symposium on Discrete Algorithms, ACM, SIAM, 2000, pp. 67–74.

[4] D. Cohen-Steiner, E. C. de Verdiere, M. Yvinec, Conforming delaunay triangulations in 3d, in: Eighteenth Annual Symposium on Computational Geometry, ACM, 2002.

[5] N. Chrisochoides, D. Nave, Parallel Delaunay mesh generation kernel, Accepted to International Journal for Numerical Methods in Engineering .

[6] R. Löhner, J. Cebral, Parallel advancing front grid generation, in: Eighth International Meshing Roundtable, Sandia National Labs, 1999, pp. 67–74.

[7] R. Said, N. Weatherill, K. Morgan, N. Verhoeven, Distributed parallel delaunay mesh generation., Comp. Methods Appl. Mech. Engrg. 177 (1999) 109–125.

[8] H. de Cougny, M. Shephard, Parallel volume meshing using face removals and hierarchical repartitioning, Computational Methods in Applied Mechanical Engineering 174 (3–4) (1999) 275–298.

[9] D. Mavriplis, S. Pirzadeh, Large–scale parallel unstructured mesh computations for 3d high–lift analysis, Tech. Rep. NASA/CR–1999–208999 and ICASE Report No. 99–9, Institude for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA (1999).

[10] L. Oliker, R. Biswas, R. C. Strawn, Parallel implementation of an adaptive scheme for 3d unstructured grids on the sp2, in: Workshop on Parallel Algorithms for Irregularly Structured Problems, 1996, pp. 35–47.

[11] H. L. deCougny, K. D. Devine, R. M. L. J. E. Flaherty, C. Ozturan, M. S. Shephard, Load balancing of parallel adaptive solutions of partial differential equations, Tech. Rep. TR94–8, Rensselaer Polytechnic Institute, Department of Computer Science (1994).

[12] N. Chrisochoides, F. Sukup, Task parallel implementation of the BOWYER–WATSON algorithm, in: Proceedings of Fifth International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, 1996.

[13] T. Okusanya, J. Peraire, Parallel unstructured mesh generation, in: Proceedings of Fifth International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, 1996.

[14] A. Bowyer, Computing Dirichlet tessellations, The Computer Journal 24 (2) (1981) 162–166.

[15] D. F. Watson, Computing the n–dimensional Delaunay tessellation with application to Voronoi polytopes, The Computer Journal 24 (2) (1981) 167–172.

[16] T. Okusanya, J. Peraire, 3D parallel unstructured mesh generation, in: S. A. Canann, S. Saigal (Eds.), Trends in Unstructured Mesh Generation, 1997, pp. 109–116.

[17] J. Bonet, J. Peraire, An alternating digital tree (ADT) algorithm for 3d geometric searching and intersection problems, International Journal for Numerical Methods in Engineering 31 (1991) 1–17.

[18] P. Chew, N. Chrisochoides, F. Sukup, Parallel constrained Delaunay meshing, in: Proceedings of 1997 Joint ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Generation, 1997.

[19] J. R. Shewchuk, A condition guaranteeing the existence of higher–dimensional constrained Delaunay triangulations, in: Fourteenth Annual Symposium on Computational Geometry, ACM, 1998, pp. 76–85.

[20] 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.

[21] J. R. Shewchuk, Tetrahedral mesh generation by Delaunay refinement, in: Fourteenth Annual Symposium on Computational Geometry, 1998, pp. 86–95.

[22] J. R. Shewchuk, Mesh generation for domains with small angles, in: Sixteenth Annual Symposium on Computational Geometry, ACM, 2000, pp. 111–112.

[23] L. A. Freitag, M. T. Jones, P. E. Plassmann, The scalability of mesh improvement algorithms, in: M. T. Heath, A. Ranade, R. S. Schreibner (Eds.), Algorithms for Parallel Processing, Vol. 105 of IMA Volumes in Mathematics and Its Applications, Springer–Verlag, 1998, pp. 185–212.

[24] N. Chrisochoides, L. Linardakis, Parallel Delaunay mesh generation using decoupling zone, to be submitted.

[25] C. Lawson, Software for C1 surface interpolation, in: J. R. Rice (Ed.), Mathematical Software III, Academic Press, New York, 1977, pp. 161–194.

[26] B. Joe, Construction of three–dimensional Delaunay triangulations using local transformations, Computer Aided Geometric Design 10 (1989) 123–142.

[27] E. P. Mücke, I. Saias, B. Zhu, Fast randomized point location without preprocessing in two– and three–dimensional delaunay triangulations, in: Twelfth Annual Symposium on Computational Geometry, 1996, pp. 274–283.

[28] L. P. Chew, Guaranteed–quality Delaunay meshing in 3–D (short version), in: Fourteenth Annual Symposium on Computational Geometry, ACM, 1997, pp. 391–393.

[29] X.-Y. Li, Spacing control and sliver–free Delaunay mesh, in: Ninth International Meshing Roundtable, Sandia National Labs, 2000, pp. 295–306.

[30] H. Edelsbrunner, Geometry and Topology for Mesh Generation, Cambridge University Press, Cambridge, England, 2001.

[31] S.-W. Cheng, T. K. Dey, Quality meshing with weighted delaunay refinement, in: 13th ACM–SIAM Symposium on Discrete Algorithms, SIAM and ACM, 2002, pp. 137–146.

[32] N. Chrisochoides, K. Barker, D. Nave, C. Hawblitzel, Mobile object layer: A runtime substrate for parallel adaptive and irregular computations, Advances in Engineering Software 31 (8–9) (2000) 621–637.

[33] N. Chrisochoides, D. Nave, Simultaneous mesh generation and partitioning, Mathematics and Computers in Simulation 54 (4–5) (2000) 321–339.

[34] M. Luby, A simple parallel algorithm for the maximal independent set, Siam Journal on Computing 15 (1986) 1036–1053.

[35] J. R. Shewchuk, Delaunay Refinement Mesh Generation, Ph.D. thesis, Carnegie Mellon University, School of Computer Science, available as Technical Report CMU–CS–97–137 (May 1997).

[36] H. Borouchaki, P. L. George, S. H. Lo, Optimal Delaunay point insertion, International Journal for Numerical Methods in Engineering 39.

[37] J.-F. Remacle, B. K. Karamete, M. S. Shephard, Algorithm oriented mesh database, in: *Ninth* International Meshing Roundtable, Sandia National Labs, 2000, pp. 349–359.

[38] K. Barker, N. Chrisochoides, J. Dobbelaere, D. Nave, K. Pingali, Data Movement and Control Substrate for parallel, adaptive applications, Concurrency and Computation Practice and Experience 14 (2002) 1–15.

[39] J. R. Shewchuk, Robust adaptive floating-point geometric predicates, in: Twelfth Annual Symposium on Computational Geometry, 1998, pp. 141–150.

[40] R. Löhner, J. Camberos, M. Marshal, Parallel unstructured grid generation, Unstructured Scientific Computation on Scalable Multiprocessors (1990) 31–64.

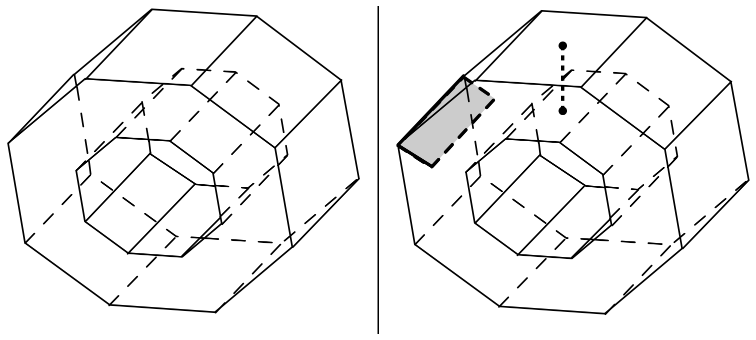
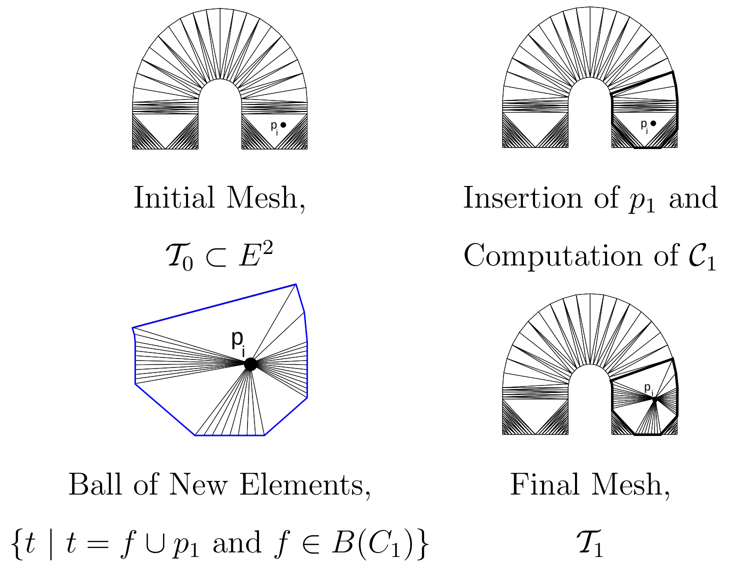


Fig. 1. A valid (left) and an invalid (right) domain. The invalid domain encloses a segment and a facet which intersect the boundary of the domain, but are not contained in the boundary.



$p_i$
Initial Mesh,
$\mathcal{T}_0 \subset E^2$
$p_i$
Insertion of $p_1$ and
Computation of $\mathcal{C}_1$
$p_i$
Ball of New Elements,
$\{t \mid t = f \cup p_1 \text{ and } f \in B(C_1)\}$
$p_i$
Final Mesh,
$\mathcal{T}_1$

Fig. 2. Steps of the Bowyer–Watson algorithm in $\Re^2$. $\mathcal{T}_{i+1} = (\mathcal{T}_i - \mathcal{C}_{i+1}) \cup \{t \mid t = f \cup p_{i+1} \text{ and } f \in \mathcal{B}(\mathcal{C}_{i+1})\}$.

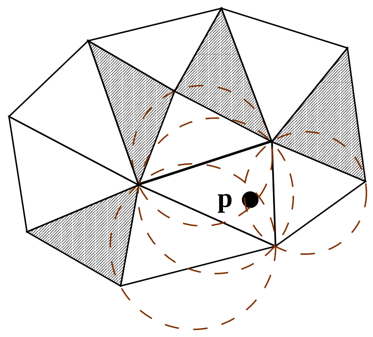


Fig. 3. The shaded triangles are the *closure triangles* adjacent to the cavity of $p$. This extends to tetrahedral cavities in the obvious way.

**Algorithm 1** Initialize a Delaunay mesh, $\mathcal{M}_o = \mathcal{M}_o(\mathcal{K}_o, \mathcal{D}_o, \mathcal{V}_o)$, conforming to $\Omega$ by refining the segments and facets of $\partial\Omega$.

```
SeqInitialization($\Omega$)
Input: $\Omega$, domain with boundary angles from $90^o$ to $270^o$.
Output: $\mathcal{M}_o(\mathcal{K}_o, \mathcal{D}_o, \mathcal{V}_o)$, 3D Delaunay mesh conforming to $\Omega$.
  Let $\mathcal{K}_o$, $\mathcal{D}_o$, and $\mathcal{V}_o$ be empty skeleton, surface, and volume triangulations.
  Insert subsegment into $\mathcal{K}_o$ for each segment in $\partial\Omega$.
  while $s \in \mathcal{K}_o$ and $|s| \geq 2d$ do
    Split $s$ in $\mathcal{K}_o$ at its midpoint $v$.
    Add $v$ onto each facet containing $s$.
  end while
  for each facet $F \in \partial\Omega$ do
    Let $\mathcal{T}$ = 2D Delaunay triangulation of subsegment endpoints on $F$.
    Remove from $\mathcal{T}$ all subfacets lying outside of $F$.
    while $f \in \mathcal{T}$ and $radius(f) \geq d\sqrt{2}$ do
      Insert circumcenter of $f$ into $\mathcal{T}$.
    end while
    Let $\mathcal{D}_o = \mathcal{D}_o \cup \mathcal{T}$
  end for
  Insert all vertices of $\mathcal{D}_o$ into $\mathcal{V}_o$.
  Remove from $\mathcal{V}_o$ all tetrahedra lying outside of $\partial\Omega$.
  Return $\mathcal{M}_o(\mathcal{K}_o, \mathcal{D}_o, \mathcal{V}_o)$.
```

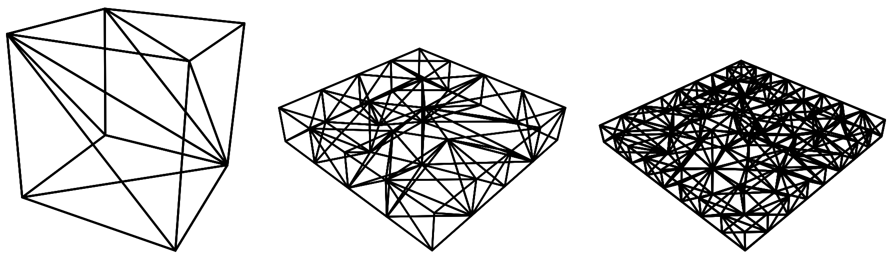


Fig. 4. The base of each domain is a unit square. The heights of the domains are, from left to right, 1, .2, and .1. Clearly, the mesh density produced by our initialization algorithm increases as $\texttt{lfs}_{min}(\partial\Omega)$ decreases.

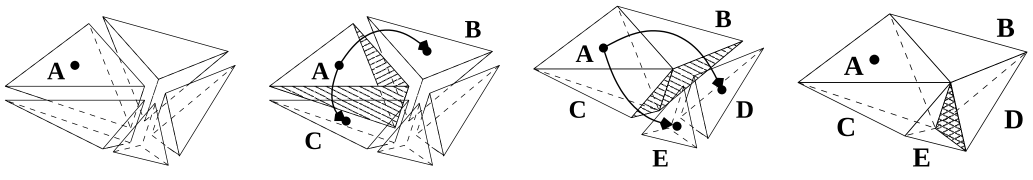


Fig. 5. The phases of a parallel breadth–first search for the subcavities A–E. Shaded faces are the marked interface faces. The shaded interface face between D and E is interior to the cavity, so no additional messages are required to complete the cavity.

**Algorithm 2** *SeqRefinement* is the sequential counterpart to our parallel mesh refinement algorithm. Parallelization is straightforward, since tetrahedra can be refined in arbitrary order.

```
*SeqRefinement*($\mathcal{M}_o(\mathcal{K}_o, \mathcal{D}_o, \mathcal{V}_o)$)
**Input**: $\mathcal{M}_o(\mathcal{K}_o, \mathcal{D}_o, \mathcal{V}_o)$, Delaunay mesh generated by initialization algorithm
**Output**: $\mathcal{M}(\mathcal{K}, \mathcal{D}, \mathcal{V})$, refined Delaunay mesh such that $\forall t \in \mathcal{V}, ratio(t) < 2$
  Let $\mathcal{M} = \mathcal{M}_o$.
  **while** $t \in \mathcal{V}$ and $ratio(t) \geq 2$ **do**
    Refine($t$)
  **end while**

*Refine*($\phi$)
  $p \leftarrow circumcenter(\phi)$
  $\mathcal{C}_p \leftarrow \{t \mid t \in \mathcal{V}$ and $t$ conflicts with $p\}$
  **if** ($\phi \in \mathcal{D}$ or $\phi \in \mathcal{V}$) and $p$ encroaches some $s \in (\mathcal{C}_p \cap \mathcal{K})$ **then**
    Refine($s$); Return.
  **else if** $\phi \in \mathcal{V}$ and $p$ encroaches some $f \in (\mathcal{C}_p \cap \mathcal{D})$ **then**
    $g \leftarrow$ largest circumradius subfacet in $(\mathcal{C}_p \cap \mathcal{D})$ that $p$ encroaches
    Refine($g$); Return.
  **end if**
  Insert $p$ into $\mathcal{V}$ by retriangulating $\mathcal{C}_p$ in $\mathcal{V}$
  **if** $\phi \in (\mathcal{D} \cup \mathcal{K})$ **then** Retriangulate $\mathcal{C}_p \cap (\mathcal{D} \cup \mathcal{K})$.
```

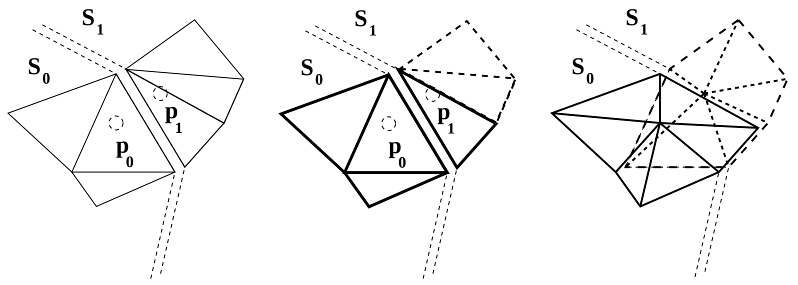


Fig. 6. Two overlapping cavities are retriangulated, which results in a *non–simplicial mesh*.

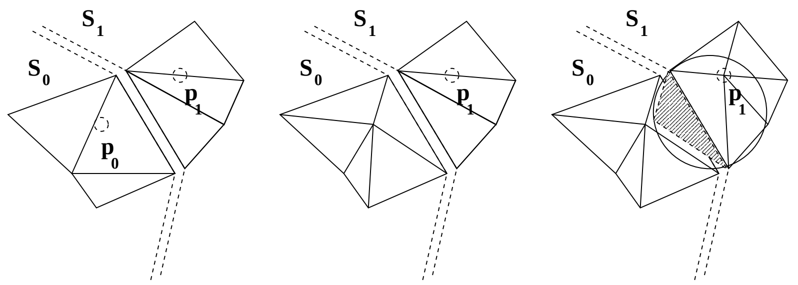


Fig. 7. Two neighboring cavities are retriangulated, which results in a *non–Delaunay mesh*.

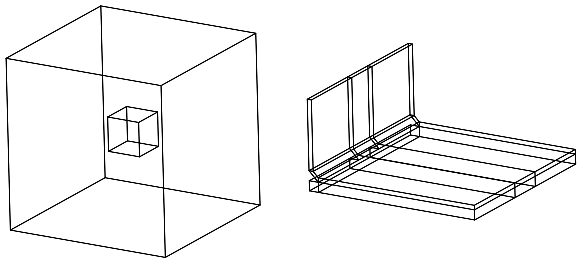


Fig. 8. Two of our experimental domains; *cube-in-cube* is on the left, and *tee* is on the right.

Table 1
Average latency tolerance ($L$), and unsuccessful poll time (*No Message*) statistics for *cube-in-cube* and *tee*, on 16 PEs of a SPARC CoW (see text for explanation of columns). Data is from the PE with the worst latency tolerance. All times are in seconds.

| Mesh Size | Execution Time | Masked Time $T_{M,L}$ ($T_{M,D}$) | Setback Time | $L$ | Poll Time (% No Message) |
|---|---|---|---|---|---|
| *cube-in-cube* | | | | | |
| 1M | 53.9 | 16.1 (6.0) | 2.7 | 89.1% | 17.4 (17.6%) |
| 2M | 92.9 | 29.1 (11.1) | 4.8 | 89.4% | 22.2 (14.4%) |
| 4M | 184.4 | 53.0 (20.9) | 10.7 | 87.3% | 46.8 (11.8%) |
| *tee* | | | | | |
| 1M | 45.3 | 11.9 (2.9) | 2.5 | 85.6% | 13.4 (19.6%) |
| 2M | 82.4 | 19.8 (4.5) | 4.6 | 84.2% | 20.1 (13.6%) |
| 4M | 159.3 | 30.9 (8.5) | 9.6 | 80.4% | 40.3 (16.1%) |

Table 2
Parallel mesh generation vs. traditional approach for *cube-in-cube*. The total parallel mesh refinement and I/O time (*Par. Time*) is much less than the total sequential mesh refinement time (*Seq. Time*). I/O (*I/O*) comprises more than 40% of the total time to sequentially generate and distribute a mesh. The total speedup (*Improvement*) is (*Seq. Time*)/(*Par. Refine*). All times are in seconds.

| Mesh Size | Mesh Gen. (I/O) | Partition (I/O) | Par. Refine | Seq. Time | Improvement |
|---|---|---|---|---|---|
| 1M | 147 (37) | 14 (75) | 81 | 273 | 3x |
| 2M | 305 (82) | 33 (158) | 120 | 578 | 4x |
| 4M | 650 (175) | 75 (379) | 211 | 1279 | 6x |

**Algorithm 3** This algorithm is guaranteed to terminate, where $\forall t \in \mathcal{M}$, $ratio(t) < 2$. $thread(F)$ instantiates a thread in which to execute function $F$.

```
$ParRefinement(\mathcal{M}_o(\mathcal{K}_o, \mathcal{D}_o, \mathcal{V}_o))$
**Input**: $\mathcal{M}_o(\mathcal{K}_o, \mathcal{D}_o, \mathcal{V}_o)$, Delaunay mesh generated by initialization algorithm
**Output**: $\mathcal{M}(\mathcal{K}, \mathcal{D}, \mathcal{V})$, refined Delaunay mesh such that $\forall t \in \mathcal{V}, ratio(t) < 2$
  Let $S_0 \cup S_1 \cup \ldots \cup S_{N-1} = \mathcal{M}_o(\mathcal{K}_o, \mathcal{D}_o, \mathcal{V}_o)$.
  **On** each PE$_k$, $0 \le k \le N-1$ **do**
    Let $\mathcal{L}_k$ be a list of to–be–refined tetrahedra in $\mathcal{V}_k$.
    Let $Q_k$ be a set of outstanding refinement threads in PE$_k$.
    **loop**
      Poll network for new messages.
      Yield to threads in $Q_k$.
      $t \leftarrow \mathcal{L}_k.remove\_head()$
      $Q_k \leftarrow Q_k \cup thread(\text{RefineThread}(t))$
      **if** $Q_k = \emptyset$ and $\mathcal{L}_k = \emptyset$, $0 \le k \le N-1$ **then** Exit.
    **end loop**


$RefineThread(t)$
  Refine($t$)
  **if** $t \in \mathcal{V}_k$ **then** $\mathcal{L}_k.append(t)$.

$Refine(\phi)$
  $p \leftarrow circumcenter(\phi)$
  $\mathcal{C}_p \leftarrow$ parallel-breadth-first-search($p$, $\phi$)
  **if** $\mathcal{C}_p = \emptyset$ **then** Return.
  **if** ($\phi \in \mathcal{D}_k$ or $\phi \in \mathcal{V}_k$) and $p$ encroaches some $s \in (\mathcal{C}_p \cap \mathcal{K}_k)$ **then**
    Refine($s$); Return;
  **else if** $\phi \in \mathcal{V}_k$ and $p$ encroaches some $f \in (\mathcal{C}_p \cap \mathcal{D}_k)$ **then**
    $g \leftarrow$ largest circumradius subfacet in $(\mathcal{C}_p \cap \mathcal{D}_k)$ that $p$ encroaches
    Refine($g$); Return.
  **end if**
  Insert $p$ into $\mathcal{V}_k$ by retriangulating $\mathcal{C}_p$ in $\mathcal{V}_k$
  **if** $\phi \in (\mathcal{D}_k \cup \mathcal{K}_k)$ **then** Retriangulate $\mathcal{C}_p \cap (\mathcal{D}_k \cup \mathcal{K}_k)$.
```