Guaranteed–Quality Parallel Delaunay Refinement for Restricted Polyhedral Domains

[Extended Abstract]

Démian Nave*
Pittsburgh Supercomputing Center
Carnegie Mellon University
Pittsburgh, PA 15213
dnave@psc.edu

Nikos Chrisochoides†
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187
nikos@cs.wm.edu

Paul Chew‡
Department of Computer Science
Cornell University
Ithaca, NY 14853
chew@cs.cornell.edu

ABSTRACT

We describe a parallel Delaunay refinement algorithm for polyhedral domains which can generate meshes of tetrahedra with circumradius to shortest edge ratio less than 2, as long as the angle separating any two incident segments and/or facets is between 90^o and 270^o degrees.

Our algorithm allows the submesh interfaces induced by an element–wise partitioning of an initial mesh of the domain to change as the mesh is refined. Concurrently inserted mesh vertices can change the tetrahedralization in many submeshes. This flexibility is crucial to ensure mesh quality, but it introduces unpredictable and variable latencies due to long delays in gathering remote data required for updating mesh data structures. In our experiments, more than 80% of this latency was masked with computation due to the fine–grained concurrency of our algorithm.

Our experiments also 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 arguably a simple first step in the development of much more sophisticated guaranteed–quality parallel mesh generation algorithms.

*This work was supported by the NSF CISE Challenge #EIA-9726388 and NIH National Center for Research Resources #P41 RR06009

†This work was 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.

‡This work was 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 adaptive solution of partial differential equations on parallel machines. A traditional approach to refining and distributing a mesh for a parallel field solver might first sequentially generate a refined mesh, then partition and distribute this mesh to the processing elements of a parallel machine with graph partitioning software like parallel Metis [13]. 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.

Algorithms for creating distributed Delaunay triangulations of known point sets have existed for some time, such as the parallel gift–wrapping algorithm of Teng et. al. [19], or the worst–case optimal recursive parallel projective Delaunay triangulation algorithm of Blelloch et. al. [2]. Assuming that an initial point set is known is inconvenient for mesh refinement, however, as a mesh refinement algorithm must be able to insert new vertices to meet the desired solution accuracy of the field solver.

In contrast, Chrisochoides et. al. [9] present a combined task– and data–parallel approach for iteratively generating a distributed 2D Delaunay triangulation with a parallel Bowyer–Watson [4, 21] algorithm. Their approach does not consider the placement of new points to guarantee quality, although the submesh interfaces are allowed to change due to a newly inserted point. Using a different technique to ensure quality, Chew et. al. [5] proposed an efficient 2D parallel constrained Delaunay mesh generation algorithm which could ensure quality of the refined mesh. The boundary of each submesh is fixed in their algorithm by the partitioning of the initial mesh.

George et. al. [11] present a domain–decomposition based algorithm which heuristically partitions a surface mesh, rather than a volume mesh, of the domain. When successful, their algorithm places Delaunay surfaces to partition the domain such that the surfaces will appear in a Delaunay tetrahedralization of each subdomain. The subdomains which result

can then be distributed to the processing elements (PEs) of a parallel machine, and independently meshed by a sequential Delaunay refinement algorithm.

Okusanya et. al. [15] present a parallel Bowyer–Watson algorithm for 3D mesh refinement which inserts new vertices into a distributed Delaunay mesh based upon the distribution of a background mesh. Much like the algorithm proposed by Chrisochoides et. al. [9], their algorithm allows the submesh interfaces to change as new vertices are inserted. Their algorithm does not explicitly handle the surface of the domain, however.

Of the techniques presented, only the parallel constrained Delaunay meshing algorithm of Chew et. al. can be used to ensure quality of the resulting mesh, although constrained Delaunay triangulations do not necessarily exist in higher dimensions [16].

Our approach consists of two steps: 1) *sequential mesh initialization* (§2), and 2) *parallel mesh refinement* (§4). In the initialization step, a *conforming Delaunay mesh* $\mathcal{M}_o$ is constructed which fills the input domain Ω, as long as the angle separating incident entities in $\partial\Omega$ is between 90^o and 270^o. In the second step, $\mathcal{M}_o$ is passed as input to our guaranteed–quality parallel Delaunay refinement algorithm, which refines a distributed, element–wise decomposition of the initial mesh by concurrently and asynchronously adding new vertices into and restoring the Delaunay property of each submesh in the distributed mesh.

The most important feature of our algorithm is that the separators induced by partitioning the initial mesh are allowed to change with the tetrahedralization of the submeshes. Consequently, we can prove (§4) that our parallel refinement algorithm terminates, and generates a new distributed Delaunay mesh containing tetrahedra whose radius–edge ratio is less than 2.

Our experimental data (§5) suggest that our parallel Delaunay 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 46%. We show that even with this additional overhead, our algorithm can create and place large meshes up to 6 times faster than a typical approach to generating and refining a distributed mesh. Our algorithm appears to be the first provably–good parallel mesh refinement algorithm which is also practically efficient and latency tolerant (§6). We discuss theoretical and practical improvements we are exploring for deployment in future algorithms (§7).

1.1 Definitions

Let Ω be a closed polyhedral domain, possibly having holes and voids, and let $\partial\Omega$ be the boundary of Ω, whose components are vertices, linear vertex–bounded segments, and (not necessarily convex) planar, segment–bounded, polygonal facets. If two segments share a vertex, two facets share a segment, or a facet and a segment (or another facet) share a vertex, then they are *related*; otherwise, they are unrelated. We require that the angle separating any two related components in $\partial\Omega$ be at least 90^o, but no more than 270^o.

We allow new mesh vertices to be inserted on segments and facets in $\partial\Omega$ to *refine* the domain boundary. A refined segment is the union of one or more *subsegments* whose endpoints are vertices on the segment. The *diametral* sphere of a subsegment s is the unique smallest sphere circumscribing its vertices, with center and diameter as the midpoint and length of s, respectively.

Similarly, a refined facet is the union of one or more triangular *subfacets* in a simplicial triangulation of the vertices on the facet. The *equatorial* sphere of a subfacet f is the unique smallest sphere circumscribing its vertices, with center and radius in the plane of f (referred to as the circumcenter and circumradius of f, respectively). A subsegment or subfacet is *encroached* if its smallest closed circumscribing sphere encloses a vertex in $\mathcal{T}(\Omega)$ other than its vertices.

A Delaunay mesh $\mathcal{M}(\Omega)$ of Ω is the combination of two meshes: 1) a Delaunay surface triangulation $\mathcal{D}(\partial\Omega)$ consisting of the refined segments and facets of $\partial\Omega$, and 2) a tetrahedralization $\mathcal{T}(\Omega)$ consisting of Delaunay tetrahedra Ω, such that the subsegments and subfacets in $\mathcal{D}(\partial\Omega)$ appear in $\mathcal{T}(\Omega)$. $\mathcal{M}(\Omega)$ is a *good–quality* mesh if the circumradius to shortest edge ratio (radius–edge ratio, or $ratio(t)$) of every tetrahedron t in $\mathcal{T}(\Omega)$ is less than 2.

1.2 The Bowyer–Watson Algorithm

Typical Delaunay meshing algorithms are *incremental point insertion* algorithms, in that they begin with empty surface and volume Delaunay triangulations, and construct a refined mesh $\mathcal{M}(\Omega)$ of a domain Ω by inserting new vertices one-at-a-time into both the surface and/or the volume. After a new vertex v is added into $\mathcal{M}(\Omega)$, the Delaunay property of the mesh is restored by first removing tetrahedra whose open circumspheres contain (or *conflict* with) v, and then replacing them with tetrahedra which are Delaunay with respect to v and to all existing vertices in $\mathcal{M}(\Omega)$.

The two most common algorithms for maintaining the Delaunay property of an initial Delaunay tetrahedralization after inserting a new point are the *flip* [14, 12] and Bowyer-Watson [4, 21] algorithms. We have chosen the Bowyer-Watson algorithm as the basis of our parallel refinement algorithm, as it is particularly amenable to parallelization—more so than flip–based algorithms. A careful implementation of the Bowyer-Watson algorithm can be more efficient than a comparable implementation of a flip–based algorithm, because fewer floating–point calculations are required [3].

The sequential Bowyer-Watson algorithm generates a new Delaunay tetrahedralization $\mathcal{T}_{i+1}$, given an initial Delaunay tetrahedralization $\mathcal{T}_i$ and a new point $p_{i+1} \notin \mathcal{T}_i$ to add to $\mathcal{T}_i$. This process is normally implemented in four steps (see fig. 1 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

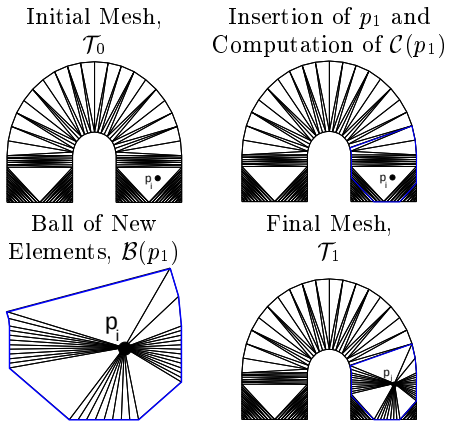

Figure 1: ***The Bowyer-Watson algorithm in*** $\Re^2$**:** $\mathcal{T}_{i+1} = (\mathcal{T}_i - \mathcal{C}_{p_{i+1}}) \cup \mathcal{B}_{p_{i+1}}$.

the faces of $\mathcal{T}_i$ to compute $\mathcal{C}_{p_{i+1}} \subseteq \mathcal{T}_i$, the set of all tetrahedra which conflict with p_{i+1} ($\mathcal{C}_{p_{i+1}}$ is the *insertion polyhedron* or *cavity* of p_{i+1}).

3. *Cavity deletion*: $\mathcal{T}_i = \mathcal{T}_i - \mathcal{C}_{p_{i+1}}$, leaving a face–bounded polyhedral "hole" in $\mathcal{T}_i$. Let $\mathcal{H}_{p_{i+1}}$ be the set of faces in the boundary of the hole.

4. *Cavity retriangulation*: $\mathcal{B}_{p_{i+1}} = \{t, \forall f \in \mathcal{H}_{p_{i+1}}, t = p_{i+1} \cup f\}$; then $\mathcal{T}_{i+1} = \mathcal{T}_i \cup \mathcal{B}_{p_{i+1}}$.

A depth–first search of $\mathcal{T}_i$ can be used to find the tetrahedra in $\mathcal{C}_{p_{i+1}}$ which conflict with p_{i+1}, given an initial tetrahedron, t, as the first tetrahedron in the search. For our Delaunay mesh refinement algorithm, point location is not required to find t, since we place all new vertices on (or near) the circumcenter of a known tetrahedron.[1]

Because of the Delaunay property, each tetrahedron in $\mathcal{C}_{p_{i+1}}$ must share a face with at least one other tetrahedron in $\mathcal{C}_{p_{i+1}}$. Tetrahedra not in $\mathcal{C}_{p_{i+1}}$ which share at least one face with a tetrahedron in $\mathcal{C}_{p_{i+1}}$ are called *closure* tetrahedra. The union of these closure tetrahedra and $\mathcal{C}_{p_{i+1}}$ is called the cavity *closure*, $\bar{\mathcal{C}}_{p_{i+1}}$.

Generating the new tetrahedra in $\mathcal{B}_{p_{i+1}}$ is data structure dependent; a straightforward algorithm can be used to walk the faces of $\mathcal{H}_{p_{i+1}}$ and connect each to p_{i+1} [3]. Note that it is this bulk update property of the Bowyer-Watson algorithm that makes it attractive for parallel Delaunay mesh generation (see §4).

2. MESH INITIALIZATION AND SEQUENTIAL DELAUNAY REFINEMENT

We describe next a sequential procedure which can generate an initial mesh suitable for input to our parallel Delaunay refinement algorithm. Then, before moving on to develop our algorithm, we prove that the base sequential algorithm, *SeqRefinement* (fig. 2), from which our parallel algorithm is derived, both terminates and outputs tetrahedra with the required bounds on radius–edge ratio. Shewchuk [17] has developed a sequential algorithm similar to the one presented here.

[1] This is also a property shared by existing Delaunay refinement algorithms. With a flip–based algorithm point location would still be necessary, although the search is bounded by the size of $\mathcal{C}_{p_{i+1}}$.

2.1 Initializing a Mesh for Parallel Delaunay Refinement

Let d be the minimum distance between any two unrelated components in $\partial\Omega$, times $1/\sqrt{2}$. Then, input to our parallel algorithm is a Delaunay mesh, $\mathcal{M}_o(\Omega)$, consisting of the surface triangulation $\mathcal{D}_o$ and the tetrahedralization $\mathcal{T}_o$, with the following properties:

1. The segments and facets in $\partial\Omega$ appear as a union of subsegments and subfacets in $\mathcal{D}_o$ and $\mathcal{T}_o$.

2. The length of each subsegment in $\mathcal{D}_o$ is less than $2d$.

3. The circumradius of each subfacet in $\mathcal{D}_o$ is less than $d\sqrt{2}$.

$\mathcal{M}_o = \mathcal{M}_o(\Omega)$ can be constructed with the following procedure:

1. *Segment refinement*: refine each segment $S \in \partial\Omega$ by subdividing subsegments on S until the length of each subsegment on S is less than $2d$.

2. *Facet refinement*: for each facet $F \in \partial\Omega$, let D be a Delaunay triangulation of the vertices of F and the vertices added onto the segments bounding F. Then, insert the circumcenters of subfacets until the circumradius of each subfacet on F is less than $d\sqrt{2}$. Remove all subfacets in D which lie outside of F, and let $\mathcal{D}_o = \mathcal{D}_o \cup D$.

3. *Mesh initialization*: add the vertices in $\mathcal{D}_o$ by Delaunay insertion into an empty Delaunay tetrahedralization, $\mathcal{T}_o$. Then, remove all tetrahedra in $\mathcal{T}_o$ which lie outside of Ω.

The output of this initialization algorithm is $\mathcal{M}_o$, a uniform–density conforming Delaunay mesh. It is clear from the definition of d that $\mathcal{M}_o$ may contain many more tetrahedra than necessary to fill $\partial\Omega$. However, we expect that an order of magnitude more tetrahedra will be generated by our parallel refinement algorithm, thus lessening the effects of an overly dense initial mesh (see §5).

3. A SEQUENTIAL DELAUNAY REFINEMENT ALGORITHM

Our sequential refinement algorithm (fig. 2) depends upon a constant h, $0 < h < d$. The idea here is that d is a relatively large value used during initialization to build a coarse mesh suitable for the initial partitioning of the mesh among parallel processors, while the constant h is a smaller value specifying the minimum element-size that the user desires in the final mesh.

SeqRefinement($\mathcal{M}_o$)

Input: $\mathcal{M}_o$, Delaunay mesh generated by initialization algorithm

Output: $\mathcal{M}(\Omega)$, refined Delaunay mesh such that $\forall t \in \mathcal{T}(\Omega), ratio(t) \leq 2$

Let $\mathcal{M} = \mathcal{M}_o$, $\mathcal{T} = \mathcal{T}_o$, and $\mathcal{D} = \mathcal{D}_o$
while $\exists t \in \mathcal{T} \ni: ratio(t) \geq 2$ **do**
Refine(t)
end while

Refine(ϕ)
$p \leftarrow circumcenter(\phi)$
$\mathcal{C}_p \leftarrow \{t, t \in \mathcal{T}$ and t conflicts with $p\}$
Retriangulate(ϕ,p,$\mathcal{C}_p$)

Retriangulate(ϕ,p,$\mathcal{C}_p$)
if (ϕ is a subfacet or tetrahedron) and p encroaches subsegment $s \in \mathcal{C}_p$ **then**
Refine(s); Return.
else if ϕ is a tetrahedron and p encroaches a subfacet $f \in \mathcal{C}_p$ **then**
$f \leftarrow \max_{radius(f)} (f \in \mathcal{D})$ that p encroaches
Refine(f); Return.
end if
Retriangulate $\mathcal{C}_p$ in $\mathcal{T}$
if $\phi \in \mathcal{D}$ **then** Retriangulate $\mathcal{C}_p \cap \mathcal{D}$ in $\mathcal{D}$

Figure 2: *A parallelizable sequential Delaunay meshing algorithm*: no subsegment or subfacet is encroached upon before or after a new vertex is inserted; hence, tetrahedra can be refined in arbitrary order.

The value of the constant h is determined either by adding additional vertices into $\partial\Omega$ to decrease the distance between unrelated components, or by specifying a size function over Ω to control the maximum size of tetrahedra. A slight modification to the algorithm is required to make sure that tetrahedra larger than the value of the size function are refined.

Our algorithm can potentially introduce *slivers* (flat elements of near-zero volume) into the mesh. The best strategy to produce 3D Delaunay meshes with bounded aspect ratio and without slivers is a subject of ongoing research (see, for instance, [10] where one such strategy is discussed).

To show that this algorithm is correct, we prove by induction that the algorithm has the following properties: 1) the algorithm terminates with the length of the smallest edge greater than h, and 2) when the algorithm terminates, $ratio(t) \leq 2$ for every tetrahedron in the mesh.

Consider the following invariants:

1. Subsegments and subfacets in $\mathcal{D}$ are unencroached in $\mathcal{T}$ before each new vertex is added to $\mathcal{M}$.
2. Each refined subsegment has length greater than $2h$.
3. Each refined subfacet has circumradius greater than $h\sqrt{2}$.
4. 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. First, note that each of these invariants holds at the start of the algorithm.

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

PROOF. Each time a new vertex other than a subsegment midpoint is inserted, a check is performed to see if it encroaches upon a subsegment or a subfacet (in the case of a tetrahedron circumcenter). The angle bounds and the value of d have been chosen to ensure that a new subsegment midpoint can never encroach upon an existing subsegment or subfacet. Further, when a subfacet circumcenter is added, the new vertex can never encroach upon an existing subfacet without also encroaching upon a subsegment. This would cause the subsegment to be refined instead of adding the circumcenter to the mesh. Therefore, it is impossible to add a new vertex that encroaches upon a subsegment or subfacet, so the first invariant holds.

For the remaining invariants, consider what happens at the first occurrence of a failure.

Assume the first failure occurs when some subsegment (say s) with length $2h$ or less is refined. Because of the angle and distance restrictions, this can occur only if there is an encroaching vertex v that is the circumcenter of a subfacet or a tetrahedron. $\mathcal{T}$ is Delaunay, so the open sphere centered at v cannot contain the endpoints of s. But, the radius of this sphere is greater than $h\sqrt{2}$, so the closed diametral sphere of s cannot contain v. This is a contradiction.

Next, assume the first failure occurs when some subfacet (say f) with circumradius $h\sqrt{2}$ or less is refined. Note that f must be encroached upon by the circumcenter (say v) of some tetrahedron with circumradius greater than $2h$ to have been refined. Because $\mathcal{T}$ is Delaunay, the open sphere centered at v cannot contain the vertices of f, so the vertices of f are farther than $2h$ from v. Hence, v can be within the closed equatorial sphere of f only if f is an obtuse triangle. If this occurs, then the subfacet over the obtuse edge of f is a larger–radius subfacet which is also encroached upon by v. But, this is a contradiction, since the algorithm refines the subfacet with the largest circumradius encroached upon by v.

Finally, assume the first failure occurs when some tetrahedron (say t) with circumradius $2h$ or less is refined. The radius–edge ratio of t must be 2 or more to have been refined, so the length of the shortest edge (call it e) of t is less than h. 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 1. *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 SKETCH. 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's clear from the algorithm that termination implies that the radius–edge ratio of each tetrahedron in $\mathcal{M}$ is less than 2. □

REMARK 1. *The correctness of* SeqRefinement *does not depend on the order in which newly inserted vertices are added into the mesh.*

This property makes Algorithm *SeqRefinement* relatively easy to parallelize, since any two cavities which do not intersect can be retriangulated concurrently without invalidating the proof of correctness of *SeqRefinement*. We next develop our parallel refinement algorithm, which is straightforward both in theory and in practice.

4. 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 a relaxed, *asynchronous* programming model, in which each processing element (PE) of the parallel system operates independently on distributed data structures to accomplish the common goal of generating a refined, guaranteed–quality Delaunay mesh.

We assume the following properties of the application run–time environment:

1. The message passing model is that of one-sided, asynchronous remote procedure calls as supported by Active Messages [20], or DMCS [1], for example.
2. Application data structures in the memory of a single PE are modified sequentially by any preemptive or non–preemptive [2] thread executing in that PE.

We will also assume that an initial mesh, $\mathcal{M}_o = \mathcal{M}(\Omega)$, has been generated, and it has the properties described in §2.1. We also require that $\mathcal{M}_o$ be partitioned and distributed to the PEs in the parallel system, as described below:

[2]If non–preemptive threads are not available, then the required state of suspended operations can be stored explicitly, and the functionality split into several procedures to emulate the process of saving and restoring thread state.

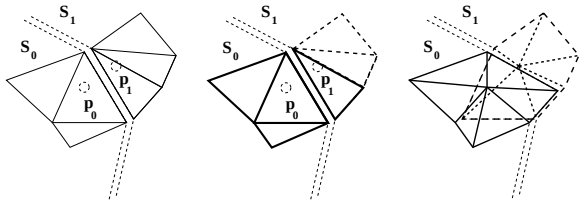

Figure 3: Two overlapping cavities are retriangulated, which results in a *non–simplical mesh.*

1. *Initial domain decomposition*: $\mathcal{M}_o$ has been element–wise partitioned into N submeshes $\bigcup_{k=1}^{N} S_k = \mathcal{M}_o$, and distributed to P[3] PEs, where submesh S_k is refined in the memory of PE_k. 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 $\mathcal{I}_{j,k} = S_j \cap S_k \neq \emptyset$, then S_j and S_k are *adjacent*. 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. Also, if S_j is a neighbor of S_k, then each face $f \in \mathcal{I}_j \cap \mathcal{I}_{j,k}$ contains a reference to S_k.

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 tetrahedralizations or surface triangulations; new vertices inserted into one submesh can affect the tetrahedralization or surface triangulation in another submesh. This flexibility is both a fundamental basis for the quality guarantees of our parallel refinement algorithm, and a useful tool to help load balance the refinement process [7].

We also allow new vertices to be concurrently inserted into the distributed mesh by the PEs in the computation. More importantly, we allow the cavities for these new vertices to be computed and retriangulated concurrently, as long as their closures (§1.2) do not share tetrahedra. A *setback* occurs if two or more cavities share tetrahedra. Setbacks introduce additional overhead not present in a sequential algorithm (§5), since they hinder the progress of the algorithm, and cause computation to be restarted.

Concurrency introduces additional theoretical and practical problems, which we describe in the following sections.

4.1 Theoretical Problems Due to Concurrency

A *distributed* cavity may contain conflicting tetrahedra from many submeshes, while a cavity which contains tetrahedra in one submesh is *local*. Retriangulating any local or distributed cavity which intersects another cavity can result in

[3]It is also possible to *over-decompose* $\mathcal{M}_o$ by distributing $N \gg P$ submeshes to P PEs without invalidating our algorithm.

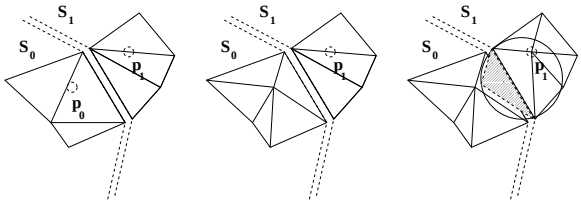

Figure 4: Two neighboring cavities are retriangulated, which results in a *non–Delaunay mesh.*

a non–simplicial or non–Delaunay mesh. Consider the two following scenarios, in which 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 fig. 3).
2. *Neighboring cavities*: $\exists f \in \mathcal{H}_p \cap \mathcal{H}_q$; 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 fig. 4).

From the definition of the closure, it is clear that the closures of two overlapping or neighboring cavities must also overlap. We can therefore avoid both problems and ensure a correct mesh by preventing the closures of concurrently computed cavities from overlapping, as the following lemma demonstrates.

LEMMA 2. *Let $\mathcal{T}$ be a Delaunay tetrahedralization, and let p and q be two vertices, such that $p \neq q$ and $p, q \notin \mathcal{T}$. If $\mathcal{C}_p$ and $\mathcal{C}_q$ do not overlap, then the tetrahedralization $\mathcal{T}''$ resulting from retriangulating $\mathcal{C}_p$ and $\mathcal{C}_q$ is Delaunay.*

PROOF. Clear from the Delaunay property of $\mathcal{T}$. □

REMARK 2. *If the Bowyer–Watson algorithm is used to retriangulate $\mathcal{C}_p$ and $\mathcal{C}_q$, then retriangulating $\mathcal{C}_p$ and $\mathcal{C}_q$ results in the same tetrahedralization regardless of the order of retriangulation.*

This follows from Lemma 2.

All that remains is to devise a strategy to prevent the closures of two concurrently computed cavities from overlapping. A straightforward solution 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 locking scheme ensures that the closures of any two retriangulated cavities do not overlap. Therefore, from Lemma 2, it is clear that retriangulating this cavity concurrently with any other non–overlapping cavities results in a Delaunay tetrahedralization.

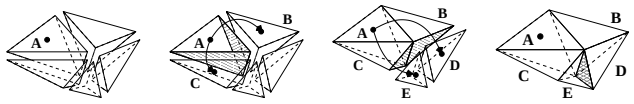

Figure 5: *Breadth–first parallel search*: scatter–gather phases of a breadth–first parallel search for the subcavities A-E. Shaded faces are the marked interface faces. The shaded face between D and E is interior to the cavity, and does not cause an additional scatter–gather phase to be executed.

Since the closure is computed as a side–effect of the depth–first cavity search, this strategy introduces little additional work into our parallel algorithm. It is also straightforward to compute a distributed cavity by parallelizing the depth–first search used in the sequential Bowyer-Watson algorithm, which we describe next.

4.2 Computing a distributed cavity

Due to the decomposition of $\mathcal{M}_o$, tetrahedra which conflict with a new vertex inserted into the mesh may be contained in many submeshes. Let v be a new vertex inserted into submesh S_j. If $\mathcal{H}_v \cap \mathcal{I}_j = \emptyset$, then the cavity $\mathcal{C}_v \subseteq S_j$ is local. Otherwise, $\mathcal{C}_v$ is distributed, and each *subcavity* $\bar{c}^k \subset \bar{\mathcal{C}}_v$ with boundary $\mathcal{H}^k$ is contained in some submesh S_k. Note that $\bar{c}^k$ may contain only conflicting tetrahedra if $\mathcal{H}^k \subseteq \mathcal{I}_k$.

From the properties of the cavity, $\bar{\mathcal{C}}_v$ and its subcavities are face–connected, so at least two subcavities $\bar{c}^j$ and $\bar{c}^k$ in $\bar{\mathcal{C}}_v$ share one or more interface faces (i.e. $\mathcal{H}^j \cap \mathcal{H}^k \subseteq \mathcal{I}_{j,k}$). A parallel graph search algorithm can therefore be used to compute $\bar{\mathcal{C}}_v$ by visiting the interface faces which connect the subcavities of $\bar{\mathcal{C}}_v$.

The first computed subcavity $\bar{c}^j$ in a parallel graph search to compute a distributed cavity is the *root* subcavity, which can be computed by sequential depth–first search in S_j. Each other subcavity $\bar{c}^k$ is a *child* of the root subcavity, and can be computed by sequential depth–first search from each tetrahedron in S_k containing an interface face in $\mathcal{I}_k \cap \mathcal{H}^k$.

For each child $\bar{c}^k \subset \bar{\mathcal{C}}_v$, $\mathcal{H}^j \cap \mathcal{H}^k \subseteq \mathcal{I}_{j,k}$ for some subcavity $\bar{c}^j \subset \bar{\mathcal{C}}_v$, so a message containing the faces in $\mathcal{I}_j \cap \mathcal{H}^k$ can be sent from PE_j to PE_k to prime the search for $\bar{c}^k$. S_k may contain several subcavities, each of which must be computed to ensure the Delaunay property of the mesh after retriangulating $\bar{\mathcal{C}}_v$.

Let v be a new vertex inserted into submesh S_0, and assume that $\mathcal{C}_v$ is distributed. The parallel search to compute $\mathcal{C}_v$ can be either depth–first or breadth–first over the subcavities to be located. We have chosen to implement the following breadth–first parallel search (fig. 5); assume $j = 0$ to start, and let $\bar{c}^0$ be the root subcavity in S_0:

Breadth–first parallel search: Execute the following in

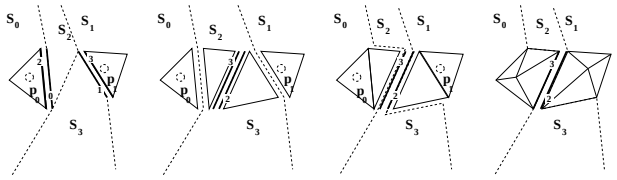

Figure 6: *Invalid submesh connectivity*: triangles in two neighboring cavities are migrated, which invalidates the connectivity information in the shared edge.

PE_0:

1. For each $S_k \ni: \mathcal{I}_k \cap \mathcal{H}^0 \neq \emptyset$, compute in PE_k each subcavity $\bar{c}^k \subseteq S_k$ sharing an unmarked interface face in $\mathcal{I}_k \cap \mathcal{H}^0$.
2. Mark in $\mathcal{H}^0$ and $\mathcal{I}_k$ the interface faces in $\mathcal{H}^0 \cap \mathcal{H}^k$. Then, if $S_k \neq S_0$, copy the conflicting tetrahedra $\mathcal{C}_v^k$ into $\bar{c}^0$, and update $\mathcal{H}^0$.
3. If no new subcavities were found, then the search is terminated. Otherwise, continue at (1) in PE_0.

Upon termination, $\bar{\mathcal{C}}_v = \bar{c}^0$, and contains the union of all subcavities containing tetrahedra conflicting with v. The conflicting tetrahedra in $\mathcal{C}_v$ are copied into $\bar{c}^0$ to find interface faces in the distributed cavity which have not yet been marked. If $\bar{\mathcal{C}}_v$ does not overlap any other cavity closure, $\mathcal{C}_v$ can be inserted into S_0 and sequentially retriangulated. Then, the conflicting tetrahedra in participating submeshes can be removed, and the interfaces which changed by migrating tetrahedra can be updated.

4.3 Practical Problems Due to Concurrency

There are two practical problems which still remain to be solved:

1. *Maintaining submesh connectivity*: if the boundaries of two cavities intersect at one or more interface faces, reference information in the faces could be updated incorrectly when tetrahedra in the cavities are migrated by the breadth–first parallel search algorithm (see fig. 6).
2. *Preventing live-lock*: our simple strategy of locking tetrahedra to prevent invalid sharing can result in live-lock, if two PEs repeatedly attempt to insert new vertices whose cavities overlap close to the submesh interfaces.

The submesh connectivity problem is really a special case of the neighboring cavities problem, which we have already solved. Live-lock can be prevented by setting a limit on the number of times a vertex insertion with the same vertex is attempted. If this threshold is reached, then the PE attempting to insert the vertex can be paused for some random length of time, which should be longer than the round-trip time of a message, before attempting to insert the vertex again. Alternatively, another vertex could also be inserted,

ParRefinement($\mathcal{M}_o$)

Input: $\bigcup_{k=1}^{N} S_k$, submeshes of $\mathcal{M}_o$.

Output: $\mathcal{M}(\Omega)$, refined Delaunay mesh such that $\forall t \in \mathcal{T}(\Omega), ratio(t) < 2$

On each PE_k:
Let $\mathcal{L}$ be list of to-be-refined tetrahedra.
Let Q be list of outstanding refinement threads.
loop
  Poll network for new messages.
  Yield to threads in Q.
  $t \leftarrow \mathcal{L}.remove\_head()$
  $Q \leftarrow thread(\text{Refine}(t))$
  if $|Q| = 0$, $|\mathcal{L}| = 0$, and all PEs are done **then** Exit.
end loop

Refine(ϕ)
  $p \leftarrow circumcenter(\phi)$
  $\mathcal{C}_p \leftarrow$ breadth-first-parallel-search(p, ϕ)
  if $\mathcal{C}_p = \emptyset$ **then**
    if ϕ is a tetrahedron **then** $\mathcal{L}.append(\phi)$
    Return.
  end if
  Retriangulate(ϕ,p,$\mathcal{C}_p$)
  for all new tetrahedra $t \ni: ratio(t) \geq 2$ **do** $\mathcal{L}.append(t)$

Figure 7: *Guaranteed–quality parallel Delaunay refinement*: this algorithm terminates, and $\forall t \in \mathcal{M}(\Omega)$, $ratio(t) < 2$.

if it has not violated the threshold. Since it is improbable that two PEs (or more) will repeatedly choose the same time to pause, live-lock is prevented.[4]

4.4 A Parallel Delaunay Refinement Algorithm

Figure 7 presents our parallel refinement algorithm, *ParRefinement*. *breadth-first-parallel-search*, as described in §4.2, returns the possibly distributed cavity $\mathcal{C}_p$. If $\bar{\mathcal{C}}_p$ overlaps any other closure, then this procedure returns an empty cavity. Otherwise, *breadth-first-parallel-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.

Assuming that a live-lock prevention mechanism has been installed into the algorithm, the following theorem shows that *ParRefinement* is correct.

THEOREM 2. *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.*

PROOF. By Remark 1, if no two cavities are retriangulated concurrently, then $\mathcal{M}$ has the desired properties. For

[4] It may be worthwhile to note that we have not encountered live-lock during the development of our code.

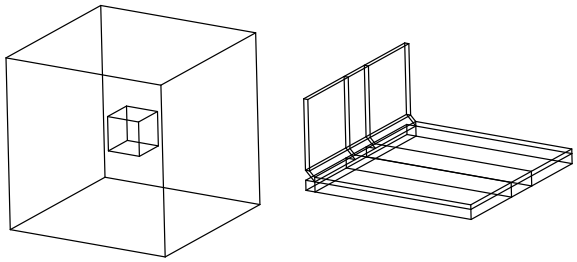

Figure 8: ***Experimental domains*****: (a)** ***cube-in-cube*****, (b)** ***tee***

any two concurrently computed cavities which do not overlap, such as those returned from *breadth–first–parallel–search*, Remark 2 ensures that the resulting mesh is the same as if they were retriangulated sequentially. This implies that Remark 1 is valid for concurrently retriangulated cavities; therefore, by construction, *ParRefinement* is correct. □

5. OVERVIEW OF PERFORMANCE

We have chosen two model problems (see fig 8): (1) a simple cube with a suspended cubical void (*cube-in-cube*), and (2) 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 512MiB RAM, and connected by a 100MiB/s fast ethernet switching fabric. Sequentially generated meshes were created on a 450MHz SPARC Ultra-80 with 4GiB RAM, connected to a remote file server via 100MiB/s fast ethernet. More detailed and specialized data pertaining to our parallel Bowyer-Watson kernel can be found in our companion paper [8].

The initial meshes passed as input to our implementation of *ParRefinement* (§4) contained about $100,000$ tetrahedra, and were partitioned with sequential Metis [13]. In each experiment, we also specified a constant size function which allowed us to control the number of tetrahedra in the refined mesh. The mesh output by *ParRefinement* were uniform–density meshes containing 1 to 4 million tetrahedra with circumradius to shortest edge ratio less than 2.

Dihedral angle data for elements in a 1 million element mesh of the *tee* appears in fig. 9. In this particular experiment, the radius–edge ratio of every tetrahedron ranged from .61 to 1.5. This plot shows dihedral angles of tetrahedra both interior to and near the boundary of the domain. Near the boundary, element dihedral angles are between 1^o and 179^o, while in the interior, element dihedral angles are between 0^o and 179^o (slivers appear even in a mesh for this relatively simple geometry).

Note that our experimental implementation is less efficient than possible, as it generates only about 1000 tetrahedra per second per processor. An optimized implementation could improve this by an order of magnitude or more (see Bourachaki et. al [3], for example).

Table 1 demonstrates how well our algorithm tolerates the long, variable, and unpredictable latencies due to the communication overheads arising from concurrently inserting new vertices into the mesh. It is very difficult to compute the actual latency of each message incurred during distributed cavity searches. However, we can measure the amount of time spent by each PE to compute and retriangulate cavities while the PE contains distributed cavity search threads which are blocked on communication.

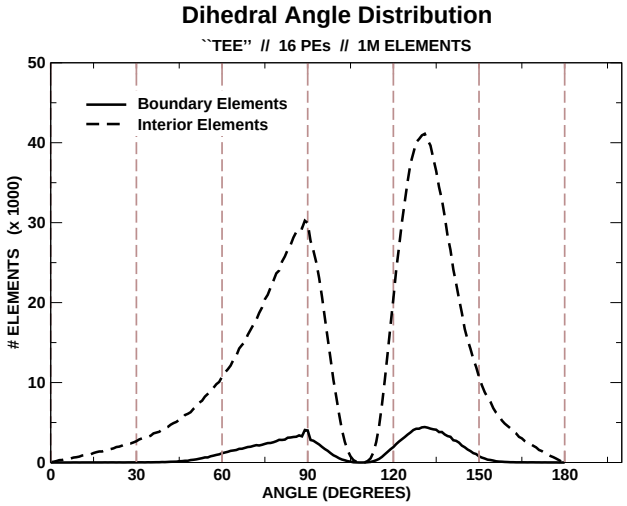

Figure 9: ***Dihedral angle distributions*****: dihedral angle distributions of elements in a 1 million element mesh of the *tee* geometry on 16 PEs. Dihedral angles near the boundary range from 1^o to 179^o. In the interior, slivers appear. As a result, dihedrals range from 0^o to 179^o. $.61 \leq ratio(t) \leq 1.5$ for all tetrahedra in the mesh.**

Further, setbacks on a PE can only occur in the presence of blocked distributed cavity search threads. We can therefore calculate how effectively each PE utilizes otherwise wasted time by computing the ratio of useful time to total work time (the sum of useful time and setback time). In our table, this value appears as the average percent of overlap, or latency tolerance. We consider the useful work to be the total time spent by a PE in computing and retriangulating cavities, including those cavity searches which complete, but result in encroached subsegments and/or subfacets. Setback time is the total time spent by a PE in computing and destroying cavities which overlap other (distributed) cavities.

It is clear from this data that our algorithm tolerates 80% or more of the communication latency in our experiments, but not without the significant additional cost of frequent network polling indicated in the final column of tab. 1. Because the arrival of messages generated by the breadth–first parallel cavity search algorithm is unpredictable, the network must be polled frequently to reduce the lifetime of cavity search threads blocked on communication. In turn, this reduces the likelihood of incurring a setback due to locked tetrahedra, and hence reduces the time spent handling setbacks, at the expense of excessive polling.[5] The last column

[5]Executing remote service requests asynchronously requires fine–grain locking of mesh data structures, which can have non–obvious and detrimental effects on performance.

Ω	Mesh Size	Execution Time	Completed Cav Time	Masked Time Local (Dist)	Setback Time	% Latency Tolerance	Poll Time (Excess)
cube	1M	53.9	26.6	16.1 (6.0)	2.7	89.1%	17.4 (9.5)
in	2M	92.9	52.6	29.1 (11.1)	4.8	89.4%	22.2 (8.6)
cube	4M	184.4	105.7	53.0 (20.9)	10.7	87.3%	46.8 (21.8)
	1M	45.3	24.6	11.9 (2.9)	2.5	85.6%	13.4 (8.9)
tee	2M	82.4	48.7	19.8 (4.5)	4.6	84.2%	20.1 (11.2)
	4M	159.3	91.0	30.9 (8.5)	9.6	80.4%	40.3 (25.7)

Table 1: *Average latency tolerance for* cube-in-cube *and* tee*:* run–time data for the worst latency–tolerant PEs refining the *cube-in-cube* (top) and *tee* (bottom) domains on 16 processors of a CoW. Time spent computing local and (in parenthesis) distributed cavities in the presence of distributed cavities is the masked time. Percent latency tolerance is the ratio of the masked time over the sum of the masked and setback times. The last column shows that excessive polling is a significant source of wasted time. For the 4 million element *tee*, $21.8s$ out of $46.8s$ (46%) was wasted. All times are in seconds.

of tab. 1 shows the total amount of time spent polling on the PE, along with the time wasted in excessive polling.

Even with this additional overhead, our experimental implementation can generate and place meshes 6 times faster than traditional techniques, as shown in tab. 2. This table depicts traditional versus parallel performance data for 1 to 4 million element meshes of the *cube-in-cube* domain on 16 PEs. For consistency, the sequential meshes were generated with *SeqRefinement* The parallel mesher was initialized with a $100,000$ element mesh, also generated by *SeqRefinement*, which was partitioned by Metis [13], and distributed via NFS to the compute nodes.

6. CONCLUSIONS

We have described a provably correct (albeit very restricted) parallel Delaunay mesh refinement algorithm, which is 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.

An experimental implementation 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, primarily 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

In order to weaken the angle restrictions on the domain boundary (and maintain a similar proof structure), we need only design an algorithm which preserves the correctness of Remark 1 and Lemma 2. By ensuring that two cavities can be retriangulated in arbitrary order (Remark 1), our sequential proof of correctness can be applied without change to a parallel refinement algorithm which only retriangulates non–overlapping cavities (Lemma 2).

For example, it is possible to modify our algorithm to weaken the angle restriction for incident segments. The refinement of an encroached subsegment can trigger the refinement of possibly (but not practically) many other subsegments which would directly or indirectly become encroached due to inserting the subsegment midpoint. It is straightforward to show that no encroached subsegments will remain in the mesh by retriangulating *en masse* all cavities resulting from the midpoints of these encroached subsegments. If any cavity could not be completed due to locked tetrahedra, then all of the cavities must be destroyed (*en masse*).[6] This procedure would allow arbitrarily ordered subsegment refinement while preserving the invariant of unencroached subsegments. Preserving the invariant of unencroached subfacets is not so straightforward, however; this is the subject of current investigation.

Data not presented here, but which will appear in the final version of this paper, shows that our algorithm offers fixed speedup proportional to $\log P$, and scalable speedup proportional to P, where P is the number of PEs. As expected, the performance of our algorithm depends heavily upon the volume of messages exchanged by the PEs. We are therefore actively studying the difficult problem of tolerating long, variable, and unpredictable latencies due to concurrency, without introducing the large communication overheads currently incurred by the algorithm. These studies include methods to schedule vertex insertions so as to prevent overlapping cavities [6].

8. ACKNOWLEDGMENTS

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).

9. REFERENCES

[1] K. Barker, N. Chrisochoides, J. Dobbelaere, D. Nave, and K. Pingali. Data Movement and Control Substrate for parallel, adaptive applications. *Accepted to Concurrency Practice and Experience*, 2001.

[2] G. Blelloch, J. Hardwick, G. Miller, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24:243–269, 1999.

[6] This is a variant of an idea Shewchuk proposed [18] in the context of refining a domain with small angles.

Mesh Size	Mesh Gen. (I/O)	Part. (I/O)	Total	Par. Time	Improvement
1M	147 (37)	14 (75)	273	81	3x
2M	305 (82)	33 (158)	578	120	4x
4M	650 (175)	75 (379)	1278	211	6x

Table 2: *Parallel mesh generation vs. traditional approach for* cube-in-cube: 16 processor CoW experiments indicate up to $6x$ speedup for a 4 million element mesh. The time measured for the traditional approach includes the sequential mesh generation, partitioning, and I/O (read/write) times. The time measured for parallel mesh generation includes $100,000$ element sequential mesh initialization, partitioning, and I/O time, and the time spent loading and generating the mesh. All times are in seconds.

[3] H. Borouchaki, P. L. George, and S. H. Lo. Optimal Delaunay point insertion. *International Journal for Numerical Methods in Engineering*, 39, 1996.

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

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

[6] N. Chrisochoides and L. Linardakis. Parallel Delaunay mesh generation using decoupling zone. To be submitted.

[7] N. Chrisochoides and D. Nave. Simultaneous mesh generation and partitioning. *Mathematics and Computers in Simulation*, 2000.

[8] N. Chrisochoides and D. Nave. Parallel Delaunay mesh generation kernel. *Submitted to IJNME*, 2001.

[9] N. Chrisochoides and 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.

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

[11] J. Galtier and 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.

[12] B. Joe. Construction of three-dimensional Delaunay triangulations using local transformations. *Computer Aided Geometric Design*, 10:123–142, 1989.

[13] G. Karypis, K. Schloegel, and V. Kumar. *PARMETIS – parallel graph partitioning and sparse matrix ordering library, Version 2.0*. University of Minnesota, Minneapolis, MN, 1998.

[14] C. Lawson. Software for C1 surface interpolation. In J. R. Rice, editor, *Mathematical Software III*, pages 161–194. Academic Press, New York, 1977.

[15] T. Okusanya and J. Peraire. 3D parallel unstructured mesh generation. In S. A. Canann and S. Saigal, editors, *Trends in Unstructured Mesh Generation*, pages 109–116, 1997.

[16] J. R. Shewchuk. A condition guaranteeing the existence of higher-dimensional constrained Delaunay triangulations. In *Fourteenth Symposium on Computational Geometry*, pages 76–85. ACM, 1998.

[17] J. R. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. In *Symposium on Computational Geometry*, pages 86–95, 1998.

[18] J. R. Shewchuk. Mesh generation for domains with small angles. In *Sixteenth Symposium on Computational Geometry*, pages 111–112. ACM, 2000.

[19] Y. A. Teng, F. Sullivan, I. Beichl, and E. Puppo. A data-parallel algorithm for three-dimensional Delaunay triangulation and its implementation. In *SuperComputing*, pages 112–121. ACM, 1993.

[20] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A mechanism for integrated communication and computation. In *Nineteenth International Symposium on Computer Architecture*. ACM, 1992.

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