Sparse Parallel Delaunay Mesh Refinement *

Benoît Hudson Gary L. Miller Todd Phillips
Computer Science Department
Carnegie Mellon University
{bhudson,glmiller,tp517}@cs.cmu.edu

ABSTRACT

The authors recently introduced the technique of sparse mesh refinement to produce the first near-optimal sequential time bounds of $O(n \lg L/s + m)$ for inputs in any fixed dimension with piecewise-linear constraining (PLC) features. This paper extends that work to the parallel case, refining the same inputs in time $O(\lg(L/s) \lg m)$ on an EREW PRAM while maintaining the work bound; in practice, this means we expect linear speedup for any practical number of processors. This is faster than the best previously known parallel Delaunay mesh refinement algorithms in two dimensions. It is the first technique with work bounds equal to the sequential case. In higher dimension, it is the first provably fast parallel technique for any kind of quality mesh refinement with PLC inputs. Furthermore, the algorithm's implementation is straightforward enough that it is likely to be extremely fast in practice.

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algrithms and Problems

General Terms: Algorithms, Theory

Keywords: Shared-Memory Parallelism, Mesh Generation, Computational Geometry

1. INTRODUCTION

The meshing problem is very old, going at least back to the 1950's – the early days of the finite element method [34]. The goal is to partition the input domain up into simple pieces. These simple pieces are then used to embed functions over the domain such as temperature or velocity. By requiring that the mesh resolve features, we can allow the embedded function to be discontinuous at these features, thus giving better interpolation for the same number of pieces. By using good shaped pieces we ensure interpolation error guarantees. Finally, by minimizing the number of pieces we minimize the time and space to represent these functions with the same quality of interpolation error.

More formally, the meshing problem takes as input a domain containing a collection of features and returns a triangulation of the domain. The features are points, edges, and, in higher dimensions, polygonal faces; this is termed a Piecewise Linear Complex [25]. There are four fundamental properties we would like of the meshing algorithm. One, the mesh should **conform** to the input: all the vertices, edges, and faces should appear as a union of simplices in the mesh. Two, all the tetrahedra should have **good quality**. Three, the number of tetrahedra should not be much more than in an optimal triangulation that is conforming and good quality: our algorithm outputs a mesh that is **size-competitive** with an optimal mesh. Lastly, the algorithm should be **work efficient** and fast.

The meshing problem as we have stated was first posed in 2D by Bern, Eppstein, and Gilbert [3] who proposed a quadtree algorithm. Ruppert [29] gave an $O(n^2)$ time, size-competitive algorithm for the meshing problem using Delaunay refinement. Mitchell and Vavasis [26] extended the quadtree algorithm to 3D and proved that the Bern, Eppstein, and Gilbert algorithm was in fact also size-competitive. All of the above work required that there be no small angles formed between an two input features. The traditional assumption, which we too require, is that all input angles are at least 90°. Handling smaller input angles is an area of open research in 3D; in two dimensions, some techniques are known [23].

Most applications require meshes where all simplices (triangles, tetrahedra, ...) have **good aspect ratio**: the volume ratio of the circumscribing sphere to the largest inscribed sphere is not too large. Delaunay refinement algorithms more naturally produce simplices of good radius-edge ratio: the ratio of the circumcribing radius to the length of the shortest edge is not too large. In two dimensions, these criteria are equivalent, but in 3D and higher, the corner case of slivers arises.

We can now state our **main results**: We introduce a new parallel algorithm for refining a mesh in any fixed dimension. For a broad class of inputs, it is asymptotically work optimal, and the parallel depth is within a logarithmic factor of optimal. The algorithm outputs a mesh with at most a constant factor more vertices than an optimal good aspect ratio mesh.

Our parallel algorithm extends our prior sequential algorithm **Sparse Voronoi Refinement (SVR)** [15]. The SVR algorithm has output-sensitive runtime $O(n \lg(L/s) + m)$, with constants depending only on the dimension and a prescribed radius-edge quality bound. Here L is the size of the domain being meshed and s is the smallest input feature. Thus, for most meshing inputs in practice (including integer coordinates) this matches the optimal time bound of $\Theta(n \lg n + m)$.

The SVR sequential algorithm at a very high level alternates be-

*This work was supported in part by the National Science Foundation under grants ACI 0086093, CCR-0085982 and CCR-0122581.

SPAA'07, June 9–11, 2007, San Diego, California, USA.

tween two types of moves: "break" and "clean". In the sequential design, only one break move happens during a break phase, while during a clean phase only one clean move may happen at a time. The first problem this paper addresses is to show that many break and clean moves can be performed simultaneously, removing this control dependency. The beauty of Sparse Voronoi Refinement is that this is true and fairly easily shown. This is due to the fact that we maintain a good radius-edge mesh throughout the life of the algorithm. Note that quadtree based algorithms have been parallelized in a similar fashion [4].

The second issue is dealing with input features and showing that this part of the code can also be parallelized. This part of the analysis is the main contribution. SVR in parallel generates a mesh of each feature. These parallel meshing procedures interact by reporting balls: spheres that contain and protect mesh elements. Lower-dimensional meshes report to higher-dimensional meshes about balls that must be protected. Conversely, a higher-dimensional mesh will report to a lower-dimensional mesh the balls into which they would like to add a point. The higher-dimensional mesh may stall while the lower-dimensional mesher refines its mesh. To ensure sufficient amount of parallelism we show that the mesh will only stall a constant amount of time before it can proceed, and that a sufficient amount of work can be done in parallel in any round.

It is interesting to point out that study into parallelizing SVR has led to a more close analysis of the sequential algorithm, making it simpler by removing unnecessary dependencies. The study of these dependencies will enable improvements beyond parallelization; we discuss several avenues for extension in the Conclusion.

2. RELATED WORK

Finding parallel unstructured meshing algorithms has been a research topic since the early 1990's [30] and is still an important research topic today [8]. The work generally falls into one of two types, strong theoretical results for point sets, or engineered solutions to handle general input without good guarantees. In the realm of parallel algorithms that handle only point sets, most octtree methods parallelized fairly easily for point input, but higher-dimensional features have proven difficult to handle efficiently [33]. Spielman *et al.* present [32], an $O(\lg m)$ parallel time bound with $O(m \lg m)$ work is shown for simple implementations of Delaunay Refinement on point sets in parallel.

A variety of parallel algorithms have been engineeered to handle input feature constraints, but with few or no theoretical guarantees on output size [9, 11, 28, 27, 5, 18]. The current best theoretical results for parallel meshing with features in 3D is the algorithm of Spielman *et al.* [31]. They present a parallel 3D meshing algorithm that handles input segments in $O(\lg^2(L/s)\ \mathrm{polylog}(n))$ time, but their algorithm is not work efficient.

3. SEQUENTIAL SPARSE REFINEMENT

We will first give an abstract overview of the sequential SVR algorithm. SVR maintains a mesh for every input feature (See Figure 1). The features can be partially ordered by containment, yielding an obvious notion of a **subfeature**. We say that F is a subfeature of F' if F is of lower dimension than F' and F is spatially contained in F'. For instance, a boundary segment may be a subfeature of an input surface constraint. Superfeature may be defined symmetrically.

Midstream, the partition implied by a mesh does not necessarily coincide with all the meshes of its subfeatures, and so the algorithm

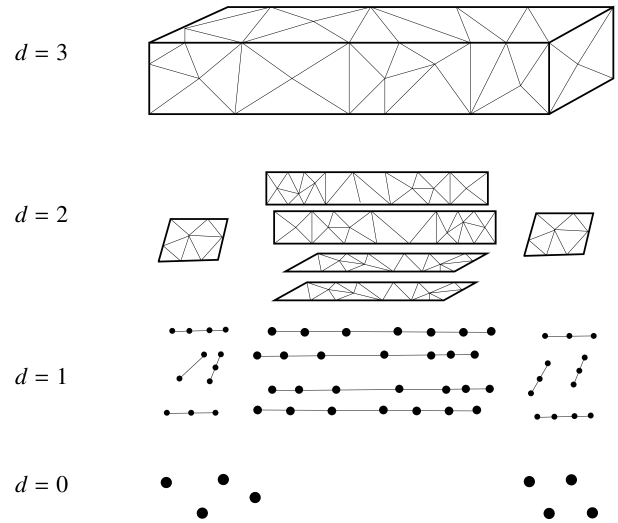

Figure 1. SVR works with a mesh of each feature at each dimension. We have the 3D volume mesh, a 2D mesh for each constraint surface, a 1D mesh for each of the edges of those surfaces, and zero-dimensional meshes for the corners. The meshes are naturally nested by containment of features. All the meshes are geometrically embedded together in $\mathbb{E}^3$, but their connectivity does not necessarily coincide midstream, so SVR uses careful communication between meshes.

maintains the geometric correspondance between meshes of different dimension: a Delaunay simplex in a higher-dimensional mesh has a pointer to every intersecting Delaunay simplex in any coincident lower-dimensional mesh. For simplicity in this work, we will consider any two simplices to be intersecting if their containing circumspheres intersect. In general, the geometric correspondance can be maintain more or less accurately, depending on implementation concerns.

SVR also maintains one mesh for the entire mesh domain; we refer to this highest-dimensional mesh as the D-mesh. At the end, the algorithm returns the final D-mesh as its output.

3.1 Initializing SVR

The first phase of the algorithm is to create coarse feature meshes from the input. We set up a constant size mesh for each of the input features and initialize the needed data structures.

The input features could be of many possible types. In 2D, features are only either vertices or edges. In either case, these are of constant size. When working in 3D, the 2D input features could be of unbounded size; consider the case of conforming to input polygons with an unbounded number of sides. Clearly, further complications can arise in higher dimension. Since a basic understanding of the algorithm can be achieved in the case of features of bounded size, we will restrict our input to such features. We first make more explicit the input.

Let $\mathcal{T}$ be the input PLC to be meshed with ambient dimension D. We will assume that $\mathcal{T}$ contains its own "bounding box".

The goal to to mesh the interior of this bounding box. In a more complicated version of the algorithm, we would introduce bounding boxes for each feature and later remove these lower-dimensional bounding boxes from the mesh. To simplify this discussion, we will assume that the input features do not need bounding boxes. That is, for each polytope F in $\mathcal{T}$ of dimension d we assume that F has bounded size, is convex, and has a Delaunay triangulation with good radius-edge in its d-dimensional space..

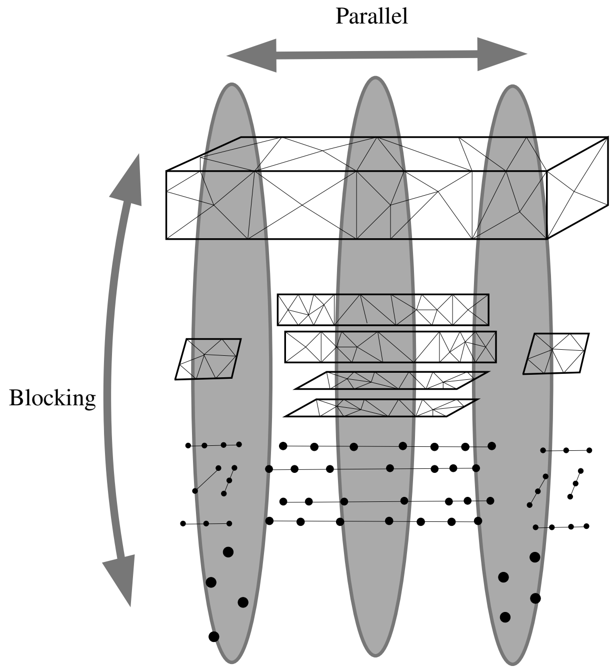

Figure 2. We show that lower-dimensional work only blocks spatially local work in the higher-dimensional meshes, so that any geometric areas of the mesh can work in parallel (circled areas). This ability to fully exploit spatial parallelism even in the presence of features is a key component of Parallel SVR.

We start by constructing the Delaunay triangulation $\mathcal{T}_P$ of each polytope P. Each triangulation $\mathcal{T}_P$ will be maintained as a mesh in it own right. We let d denote the dimension of this polytope.

More formally, a d**-simplex** s is the convex closure of a set of $d + 1$ affine independent points in a space of dimension D. The $d+1$ points are the **vertices** of s. The **circumball** of s is the smallest radius open D-ball containing the vertices of s on its boundary. The **circumcenter** and **circumradius** of s are the center and radius of its circumball. Each circumball will be reported to all superfeature meshes that intersect the simplex.

We say a ball B is **nearly empty** if 1) the interior is empty of points, or 2) there are no points on the boundary of B and at most one in the interior. We assume for simplicity that initially all the circumballs are nearly empty. This condition will be maintained throughout the algorithm.

We maintain the following information with each simplex s in each mesh $\mathcal{T}_{\mathcal{P}}$:

- A list of all uninserted points that are contained in s.
- A list of all simplices from all subfeature meshes whose circumball intersects the circumball of s.
- A list of all simplices from all superfeature meshes whose circumball intersects the circumball of s.

Throughout the algorithm we will maintain the set of meshes (one for each feature), partially ordered by containment. We will enforce that every circumball will be nearly empty with respect to all higher-dimensional vertices. Central to the runtime of the algorithm will be the invariant that every mesh is of bounded radius-edge, although the intermediate bound will be worse than in the final output.

3.2 SVR Work Set

Once we have finished this initialization, SVR progresses by refining these coarse feature meshes until all the features appear in the D-mesh and the final shape-quality bounds have been achieved. Suppose we are given a radius-edge bound ρ to be satisfied by the final output. The algorithm maintains a work set of Delaunay simplices to be destroyed by refinement. Each task is tagged with a type, the dimension, a handle into the appropriate mesh, and a geometric point we wish to insert. There are three types:

tasks represent poorly shaped Delaunay simplices that must be remedied by refinement.

tasks represent balls (D-spheres) that contain and protect lower-dimensional Delaunay simplices whose destruction has been ordered by some higher-dimensional entity desiring refinement in the region.

Finally, tasks are balls corresponding to the incongruences between lower-dimensional meshes and the D-mesh. These must be made to appear in the D-mesh by some refinement.

The work set is ordered with tasks first, in increasing order of dimension; then tasks in increasing dimension; then the tasks. The intuition for the ordering is as follows: cleaning first ensures that all the meshes maintain sufficiently good quality for basic operations to remain work efficient. Encroachment then enforces that low-dimensional Voronoi cells keep pace and do not remain oversize, ensuring that spatial locality is the same in all dimensions. Then as a default we work toward conformity, so that all the input features eventually appear in the final output mesh.

For further details on this sequential algorithm, see [15]. The goal of the rest of this paper is to expose as much parallelism as is possible in the work set.

4. ALGORITHM

To parallelize the sequential algorithm, we first discuss a very generic algorithmic framework for processing a work set with conflicting tasks. This generic version quite independent of the meshing problem at hand; the algorithm is somewhat standard. Then, beginning in we will show how to instantiate the unspecified parts of that algorithm for our geometric computation.

The analysis of our algorithm will be under a PRAM model: we have as many processors as we can use, all of them have uniform cost to access a shared memory pool, and there is a global clock. Most operations can be done with each memory cell being accessed exclusively by one processor per clock tick (the EREW model). Some require a CRCW model, where all processors can concurrently access memory. A standard result is that any reasonable shared-memory machine with p processors can simulate a PRAM with at most $O(\lg p)$ overhead.

4.1 Generic algorithm

As input to the function, we assume we are given a set S of tasks to process. We will also assume the existence of four oracles, and one black-box function. Three of the oracles tell us about three different graphs over the tasks on the work set. The black box takes a task, processes it, and returns a set of additional tasks to work on. Finally, the fourth oracle updates the work set given the set of tasks we processed. An instantiation of the algorithm is simply an implementation of the oracles and of the processing step.

The three graphs-oracles are:

- B : A directed edge $a \mapsto b$ means until a occurs, b cannot proceed.

```
P          W      S   (S : a work set)
return if S is empty
S′ ← ∅ {set of tasks to defer}
G ← C            (S)
{Defer Tasks Blocked by Lower-Dimensional Work}
for each task w ∈ B        (S) in parallel do
    remove B         (w) from C
    add B         (w) to S′
end for
{Process all the Unblocked Tasks}
Colour the conflict graph G using k ∈ O(Δ) colours.
Let G_i be the set of tasks of colour i.
P ← ∅ {the set of processed tasks}
for i = 1 to k do
    for each task w ∈ G_i in parallel do
        if w ∈ G then
            remove M      (w) from G
            add w to P
            process w similarly as the sequential algorithm
        end if
    end for
end for
{New Work Set is the Deferred Tasks and the New Tasks}
return S′ ∪ N     W      (P)
```

Figure 3. The generic algorithm. Given a work set, we compute the conflict graph, defering blocked tasks until the next round. Then we colour the graph, which gives us a safe ordering for the tasks. Finally, we iterate over each colour and perform the work. tasks that have been made moot by a task in a prior iteration of the loop are simply ignored.

- C   : An edge (a, b) means that a and b cannot be processed simultaneously, though they can be processed in either order.
- M   : A directed edge $a \mapsto b$ means that if a occurs, b will be moot. Unless the reverse edge exists, it is legal to perform b then a. However, it is not legal to perform a then b.

The B   graph must be acyclic. The M   graph need not be – in fact, it is common to have two tasks make each other moot. We assume that M   is a (directed) subgraph of C   : if a moots b, then a and b cannot both be processed simultaneously. Finally, the graphs must all be Δ-sparse; having bounded degree $\Delta \in O(1)$.

Since we have not yet defined the oracles, we cannot bound their cost. However, we can discuss the overhead of parallel processing.

- Computing G is linear work and constant depth. This follows directly from C   and B   being sparse graphs.
- Colouring the graph is $O(\lg |S|)$ parallel depth and $O(|S|)$ work. One could also trade depth and work using an algorithm of Goldberg, Plotkin, and Shannon [13], and get $O(\lg^* |S|)$ depth for $O(|S| \lg^* |S|)$ work.
- The main loop sees each task at most once, does $O(1)$ work per task looking up w to see whether it has become moot, and if not, does $O(1)$ more work removing from S the tasks that w makes moot. The remaining time is simply to iterate over the colours:

One iteration of P   W   S   is called a *round*. Every round, every task on the work set is either processed, mooted, or blocked. Blocked tasks will reappear next round unless they were mooted, alongside new tasks generated by the processed tasks.

What we've just shown is that the depth of each round is essentially $O(\lg |S|)$ per round, due to the colouring, and the optimal $O(|S|)$ work. It remains to discover the number of rounds.

```
S     (⟨T, reason⟩)
If T has been removed, return
Let B = B(c(T), kr(T)).
Compute S the set of neighbouring simplices that intersect B
for ∀ T′ ∈ S {in series} do
    for ∀ p ∈ T′ {in parallel} do
        if p ∈ kr(T), choose p
    end for
end for
if no p was chosen, let p = c(T).
for ∀ T′ ∈ S {in series} do
    for ∀ Y ∈ T′ {in parallel} do
        if p encroaches Y, and either p is a Steiner point, or p is
        the second point to encroach upon Y then
            E            (⟨Y,                    ⟩)
        end if
    end for
end for
if no Y was enqueued then
    I        (p, T)
else
    E           (⟨T,            ⟩)
end if
```

Figure 4. Processing a task. We first see if there is a lower-dimensional vertex to insert. If not, we try to insert the circumcenter. Even then, we check whether we must *yield* to lower-dimensional facets that we encroach. If we do yield, we try this task again by re-enqueueing it for next round. If not, we insert the point we chose.

4.2 Processing

Each task in the work set is a pair consisting of a simplex T and the reason, defined below, that this task is on the work set. To process a task, we use a variant of the S   call described in the sequential algorithm to find a point p to insert, and to check for encroachment on lower-dimensional features (see Figure 4). If there is no encroachment, we insert p using a variant of I   (see Figure 5).

I   has three phases: the first updates the mesh, the second updates the mapping to lower-dimensional cells, the third updates the mapping to higher-dimensional cells. We will show that the mesh is sparse; implying that the first and third phases are constant time and can therefore be done sequentially. The second phase, however, could cost as much as $O(n)$, so we must parallelize that phase. Thankfully, we can do it in the obvious trivial fashion, taking $O(1)$ depth and linear work on a CRCW.

4.3 Populating the work set

Given a cell T in some mesh M that corresponds to a feature F, we can determine whether that cell and feature define a task, using the rules that follow. After processing, then, we can examine every cell created by I   and see if it matches any or several of the rules. The necessary sequencing of rules is written at the end of this section.

R   1   (   ). *If $R_M(T)/r_M(T) > \rho$ then we add the task $\langle T,$ $\rangle$.*

R   2   (   ). *If T contains at least one uninserted point p, then we add the task $\langle T,$ $\rangle$.*

R   3   (   ). *During a* T   I   *in a mesh M_F in dimension d, if the query point q is a Steiner point (it has containment dimension d), and is inside the circumball of a lower-*

I (p, T)

```
Compute Delaunay cavity C of p in M(T), starting at T
Create a new vertex v for p
Compute the new star S around v
Relocate points and protective balls:
for ∀ T′ ∈ C {in series} do
    for ∀ Y ∈ T′ {in parallel} do
        for ∀ T″ ∈ S {in series} do
            if B(Y) intersects T″, assign Y to T″
        end for
    end for
end for
for ∀ T⁺ that contain members of C do
    replace C by {S, v}
end for
for ∀ T″ ∈ S {in series} do
    check Rules 1, 2, 4
end for
```

Figure 5. Performing the insertion of a point p. We compute the change to the current mesh $M(T)$, notify higher-dimensional meshes about the change, then reassign lower-dimensional uninserted points and protective balls. Loops over the cavity and star can be in series, as can loops over higher-dimensional meshes: these sets have bounded cardinality. Loops over the lower-dimensional features must be in parallel.

dimensional simplex T, then we add the task $\langle T,$ $\rangle$. Similarly, if q has containment dimension $d' < d$, but a vertex of the mesh M_F already encroaches upon T, we add the task $\langle T,$ $\rangle$.

D 4.1. *A simplex T (and its associated circumball) is said to be **partially resolved** if at least two of its vertices appear in the D-mesh.*

R 4 ( ). *Assume T holds two partially resolved lower-dimensional circumballs T' and T''. If T' and T'' intersect each other, then we add two tasks: $\langle F, T',$ - $\rangle$ and, symmetrically, $\langle F, T'',$ - $\rangle$.*

The weak-encroachment move is not in the original sequential code; however, we will see that it is required for achieving the full parallelism allowed by the problem. This was first noted by Spielman, Teng, and Üngör [31].

We check the clean, break, and weak-encroachment rules on all cells created during I . Encroachment is checked during T I .

Each of the first few rules are very cheap to check: the clean move is a few arithmetic operations. The break move requires noting, in the the parallel redistribution step of I , whether the cell has any uninserted points inside. The weak-encroachment rules requires checking any feature being redistributed for whether it has become partially resolved. Because of quality guarantees and the α-Lemma (Lemmas 5.1 and 5.3), there are only $O(1)$ partially-resolved features in any cell, so we can check all pairs in constant time.

4.4 Conflict graph

The goal of the conflict graph is to ensure that two tasks can be processed simultaneously without special processing. Here, we show how in a sparse mesh we can define a sparse conflict graph, which then allows us to stage insertions into the mesh in parallel without much modification to standard sequential codes for incrementally updating a mesh.

With each simplex, we associate a *protected zone* with respect to a given feature mesh; describing the area where a point being inserted could possibly modify the local geometry and/or topology. If two tasks being performed in parallel concurrently both modify the same simplex, this might require some special handling. Thus, we will put a edge between any two such tasks. More formally:

D 4.2. *Given a simplex T, the **protected zone** of T is the union of all circumballs who intersect T's circumball.*

Clearly, then, adding a new vertex u into the mesh only affects the cell $V(v)$ if u is inside the protected zone of v. Furthermore, it is important to note that the conflicts we want to maintain will not change during the processing of a round: any conflict not present before any colour has been processed will never become a conflict. The protected zone was defined in an continuous way; but it is easy to choose a discrete set of open balls that cover the protected zone: namely, the union of the Delaunay balls around the vertex:

F 4.3. *With each Voronoi node p of a cell $V_M(v)$, associate the ball $B(p, |vp|)$. The intersection of the protected zone of v with the mesh domain M is equal to the union of all those balls.*

Finally, we define the conflict graph. We put an edge between two tasks a and b if their respective protective zones intersect, and a is on a subfeature or the same feature as b.

L 4.4. *The conflict graph is of bounded degree.*

P . Observe that this graph is just the square of the intersection graph of all the circumballs, thus it suffices to prove that the intersection graph of the circumballs is of bounded degree. This is precisely true for a radius-edge quality Delaunay triangulation [24, 16] The protected zone of a vertex v is precisely the union of the circumballs of all Delaunay simplices incident on v. Therefore, any work item only encroaches on a constant number of cells in any mesh. A conflict edge goes between two tasks that both encroach upon the same cell. Clearly, there can only be a constant number of such edges. □

The results in this section showed that (a) the conflict graph allows parallel processing of tasks with minimal changes to standard codes; (b) the conflict graph is cheap to compute; (c) the conflict graph matches the requirements of the generic algorithm.

4.5 Blocking graph

In the previously-published sequential code, the algorithm very rigidly ordered the moves on its work set. Any task anywhere in space was blocked by any lower-dimensional clean task anywhere in space. Any break task was blocked by any clean task anywhere, in any dimension.

We can achieve rather more parallelism by loosening the blocking graph. We will generalize the proofs of the sequential code, and show that we can achieve the same algorithmic properties with the following blocking graph:

- A break or weak encroachment move is blocked by any clean move with which it has a conflict edge.
- Any move is blocked by a lower-dimensional yield move with which it has a conflict edge.

Since this blocking graph is a subset of the conflict graph, it is clear that it takes only $O(1)$ time to check any node for any blocks it may have.

4.6 Mooting graph

We put a mooting edge from a to any other move b if a and b are working on the same feature, and the point a inserts is inside the ball of the Voronoi node of b. The intuition is that the move b identified a Voronoi node it wanted to eliminate for some reason. The Voronoi node has now been eliminated, thus b is moot.

We also put a mooting edge from a task a to a break move b if the point inserted by a will change the cell of b. The intuition is that a break move indicates that a cell is too large; we break to whittle it down slowly to the local feature size. As long as some point modifies the cell, we have whittled at it and need to check if any further whittling is in order.

5. RUNTIME ANALYSIS

In this section, we present the runtime analysis of Parallel SVR. First, we will state some useful structural lemmas from the sequential algorithm analysis [15].

We can sketch the proof as follows: The algorithm quickly makes progress. Any task stays on the work set for only $O(1)$ rounds, even if blocked by clean moves or lower-dimensional work. Once this is established, we need only show that every task enters the work set within $O(\lg L/s)$ rounds.

First we show that within $O(\lg L/s)$ rounds, the mesh conforms to the input features. This proof will rely on packing results from the sequential algorithm that bound the spatial propagation of mesh tasks, thus removing the possibility of long chains of discovery. One the algorithm has conformed to the feature size, it only remains to improve the quality of the mesh before outputting.

As the algorithm draws toward completion, we show that only $O(\lg L/s)$ more rounds of cleaning are necessary. As in the second part, this will be due to a bound on the spatial propagation of cleaning moves, again eliminating the possibility of long chains of discovery.

5.1 Structural Lemmas

We use a few key facts repeatedly through our proofs. The first is that every mesh always has good quality, even at intermediate stages of the algorithm.

L 5.1 ([16, T 8.5]). *At all times during the algorithm, every mesh has quality $\rho' \in O(\rho)$.*

The most important corollary of this, that we use repeatedly throughout the runtime proofs is the following: because of the quality guarantees, we cannot pack more than a constant number of vertices around a point before the feature size of the mesh is forced to fall.

L 5.2 (P L : [16, L 6.7]). *Given a point p in a mesh M, the algorithm can only fit $O(1)$ more points around p until $\mathrm{cfs}_{M'}(p) \in o(\mathrm{cfs}_M(p))$.*

Finally, except at initialization, there is a correspondence in the size between all sub-meshes: essentially, if the algorithm performs any actions in a lower-dimensional mesh, it is because that mesh locally has about the same scale as the top-dimensional mesh.

L 5.3 (α-L : [16, L 7.5]). *Suppose p is considered for insertion into a mesh M. At that time, we know that $\mathrm{cfs}_{M_\Omega}(p) < \alpha\, \mathrm{cfs}_M(p)$*

5.2 Fast progress

Recall that a task is always either processed, mooted, or blocked every round. We want to make sure that a task is not blocked for too long – in fact, we will require that it be blocked only $O(1)$ rounds.

L 5.4. *If two taskss a and b conflict, then the Voronoi cells that define the protected zones of each have $r(a) \in \Theta(r(b))$.*

P . If the two tasks are of the same dimension, then by the quality condition they have the same size. Otherwise, we can instead invoke the α-lemma for the same result. □

L 5.5. *Given a task a, blocked by tasks in a lower-dimensional mesh F, at most $O(1)$ insertions can be performed in F before a is no longer blocked by F.*

P . By the α-lemma, we know that all tasks in F that are blocking a have about the same geometric size as a. By the Packing Lemma, then, we can only insert $O(1)$ points into F such that the tasks that insert them will block a. □

T 5.6. *A task on the work set blocks $O(1)$ rounds before it is either processed or mooted.*

P . Any move b that blocks a has about the same size as a, and the two moves must be geometrically near each other. By induction on the type and dimension of b, we can assume that b is processed within $O(1)$ rounds. Therefore, if we are to block a for many rounds, we need b to name a successor b'. However, we cannot do this more than $O(1)$ times by the Packing Lemma: if we tried, that would violate the condition that b' has the same size and is near a. The induction bottoms out at clean moves in dimension 1 and tops out at break moves in dimension D, so for constant D, the longest chain of blocks is $O(1)$. If b is mooted, that only speeds up the time at which it is removed from the work set. □

5.3 The mesh conforms in $O(\lg L/s)$

In this section we show that the mesh will conform to the input sizing within $O(\lg L/s)$ rounds. The general idea is that every round, everywhere that the mesh is too large for the local feature size, we will insert a point nearby. The packing proof guarantees that after inserting $O(1)$ points near an input point or protective ball, the mesh size locally has shrunk in scale by half. Given that the initial scale is $O(L)$ and the final scale is $\Omega(s)$, we can only perform this halving at most $O(\lg L/s)$ times in the finest part of the mesh.

L 5.7. *If a point p does not appear in the D-mesh M, then within $O(1)$ rounds some point q will appear in the new D-mesh M', such that $|pq| \le \mathrm{cfs}_M(p)$ and $NN_{M'}(q) \in \Omega(\mathrm{cfs}_M(p))$.*

P . If p is not in the mesh, then there is a break move associated with p on the work queue. From Theorem 5.6, we know that the break move will either occur or be removed from the queue in $O(1)$ rounds. If it occurs, then q is the point inserted by the break move. If it does not occur, then some q was inserted that modified the cell that contains p, obviating the break move. □

L 5.8. *If a lower-dimensional cell $V_{M^-}(v)$ does not conform to the local feature size – that is, it has $r_{M^-}(v) \in \omega(\mathrm{lfs}\, v)$, then within $O(1)$ rounds some point q will appear in the new D-mesh M', such that $|vq| \le \mathrm{cfs}_M(v)$, and $NN_{M'}(q) \in \Omega(\mathrm{cfs}_M(v))$.*

P . If the cell is resolved (it and its neighbours in M^- all appear in the D-mesh), then if it does not conform to lfs, it must weakly encroach on another feature. This will trigger the weak-encroachment rule, and a point will be inserted nearby soon. If instead the cell is not resolved, then it or one of its neighbours in M^- will trigger one of the two break rules, and a point will be inserted nearby soon. □

T 5.9. *After $O(\lg L/s)$ rounds, the* P R *algorithm has produced a mesh of quality $\rho' \in O(\rho)$ that conforms to local feature size.*

P . By the Packing Lemma, after $O(1)$ applications of either Lemma 5.7 or Lemma 5.8, everywhere the mesh size was larger than lfs, the mesh size will fall by half. There are $O(\lg L/s)$ length scales. □

5.4 The mesh is cleaned in $O(\lg L/s)$

The Theorem of the previous section showed that we achieved a conformal mesh of some constant quality. What's left is to show that the cleanup work afterwards takes only another $O(\lg L/s)$ rounds, at which point we will have produced the quality the user asked for. This result rests on two facts: first, that clean moves are always geometrically larger (in a sense) than the move that created the skinny cell being cleaned. Second, that if we split a cell due to a clean move, the split move is not much smaller than the clean move. Given that clean moves only spawn geometrically larger moves, a clean move can only have $O(\lg L/s)$ generations of descendents.

L 5.10 (C). *Consider a mesh vertex v whose cell $V_M(v)$ is skinny, and whose nearest neighbour u was inserted into a prior version M' of the mesh. The outradius of v is larger than the radius of the task w that inserted u: $R_M(v) \geq \frac{\rho\epsilon}{2} r(w)$.*

P . We know that $R_M(v) \geq \rho r_M(v)$ since $V_M(v)$ is skinny. Furthermore, $r_M(v) = |uv|/2$. Clearly, v is a neighbour of u, so $|uv| \geq NN_{M'}(v)$. The task w that inserted u was considering inserting some point p. It may have warped to u, up to a distance of $(1-\epsilon)r(w)$. Thus $NN_{M'}(u) > \epsilon r(w)$. □

L 5.11. *Consider an task b. The task was spawned by some higher-dimensional task a. Then $r(b) > 2^{d-1/2} r(a)$.*

This is a standard result which comes directly out of the spacing proof of SVR [16, proof of Lemma 7.4]. Thus, so long as $\rho\epsilon > 2^{d-3/2}$, a clean move a will only spawn moves – "children" of a – of size larger than the predecessor of a. What is left to show is that all the descendents of a are larger than the children. This is, in fact, false in general: until the mesh is conformal, sizes do shrink. However, we know from Theorem 5.9 that we need only wait $O(\lg L/s)$ rounds before reaching strong conformality. Afterwards, a yield move will never itself yield to another feature: it will only spawn clean moves, which we know grow.

T 5.12. *Given a strongly conformal mesh of quality ρ', the* P R *algorithm takes $O(\lg L/s)$ rounds before reaching quality ρ.*

5.5 Overall Analysis

T 5.13. *Given as input a Piecewise Linear Complex, a parameter $k \in (0,1)$, and a radius/edge quality ρ such that both $k\rho > 2^{d-3/2}$ and $(1-k)\rho > 2^{d-3/2}$, then the* P R *algorithm produces a Strongly Conforming output mesh of size m with every simplex having radius/edge at least ρ, in $O(\lg(L/s)\lg(m))$ parallel depth and work $O(n \lg L/s + m)$.*

P . Theorem 5.9 shows that after $O(\lg L/s)$ rounds, the mesh is strongly conforming. Theorem 5.12 shows that at most another $O(\lg L/s)$ rounds later, the mesh is both strongly conforming and has good radius/edge quality. According to the analysis in Section 4.1, each round takes $O(\lg|S|)$ time, where $|S|$ is the size of the workset in that round. A constant fraction of tasks end up inserting a point into the input, so we can bound the sum of all $|S|$ over time by $O(m)$. This establishes the parallel depth bound.

The work bound follows directly from the analysis of the static algorithm, and from the work-efficiency of the parallelization. □

6. PARALLEL SLIVER REMOVAL

So far, we have shown how to produce a good radius-edge Delaunay mesh. In 3D or higher dimension, bounded radius-edge meshes are unsuitable for many applications due to the presence of *slivers* – tetrahedra with good radius-edge ratio but poor aspect ratio. There have been several papers written on sliver-free meshing, or sliver exudation, i.e. modifying a mesh to remove slivers [7, 12, 20].

In this section, we overview how to parallelize the sliver elimination algorithm of Li and Teng [21, 22] as an easy extension of Parallel SVR. As input, their algorithm takes a strongly-conforming, bounded radius-edge mesh such as the one output by P R - , and parameters δ, b, ρ, and σ. Iteratively, they remove a sliver T by inserting a point p into the circumball of T. Instead of using the circumcenter of T, however, they define a *picking region* near $c(T)$ – a ball $B(c(T), \delta r(T))$. Li and Teng, following Chew [10], made the following powerful observation:

T 6.1 ([21, T 4.6]). *For appropriate δ, b, ρ, and σ, a constant fraction of points in $B(c(T), \delta r(T))$ induce only new slivers of size at least $br(T)$.*

Thus, by randomly selecting points from the picking region until we find such a safe point, we can ensure that destroying slivers only generates geometrically larger slivers, guaranteeing swift termination.

The specific changes needed to make P R produce a good aspect ratio mesh are the following. First, we must change the procedure for choosing a Steiner point as described here. Second, we add a new kind of task:

R 5 ( ). *If $\rho(T) < \rho$ but $\sigma(T) > \sigma$, or, in $D > 3$, if any subfacet of T satisfies those conditions, then add T to the work set with reason .*

Sliver elimination is only guaranteed to terminate if locally the mesh has good radius-edge radio. Therefore, tasks are blocked by tasks.

The run time follows by methods analogous to Lemma 5.10. Thus we still get at most $O(\lg L/s)$ rounds. Hence, we can remove slivers in only a constant increase in work and time over just generating a bounded radius-edge mesh.

7. CONCLUSIONS

A standard assumption is that the spread L/s of the input is polynomial in the input size. Indeed, on integer-valued input, the spread is at most $O(n^{1/d})$. Under the polynomial spread assumption, the bounds we have proven are that our algorithm achieves the optimal work bound of $O(n \lg n + m)$, and is only a logarithmic factor $O(\lg m)$ off-optimal in depth (the lower bound is from sorting). Theoretically, we could achieve $O(\lg(n)\lg^*(m))$ time but at the cost of a $O(\lg^* m)$ factor in work.

In more practical terms, we predict near-linear speedup as we add processors for as many processors as we know how to build. Furthermore, the data structures and analysis techniques are of obvious interest in analyzing several related problems: the distributed-memory case [8], out-of-core and streaming computation [17], and dynamic mesh refinement [1].

Several authors have experimented with shared-memory parallel Delaunay mesh refinement. A common technique is that of *optimistic* parallelization, where operations are performed speculatively, and backed out if a conflict is noticed [2, 19]. As noted by Kulkarni *et al.*, "optimistic parallelization is useful only if the risk of rollbacks is small." In Ruppert's algorithm, the conflict graph may have high degree (up to linear), and the risk of rollbacks may accordingly be large. Antonopoulos *et al.* note that in their code, they see a rollback rate of only about 6-10%; but they implement an algorithm that produces a near-uniform mesh. Kulkarni *et al.* do not report rates, but seem highly troubled by the number of rollbacks they see in their experiments. We suggest that our sparse parallel Delaunay algorithm, given that it has a small conflict graph, is likely to have a tiny rollback rate even in 3D. Furthermore, prior solutions need to produce a Delaunay triangulation in a preprocessing phase, likely by using a separate parallel triangulator [6]. Our preprocessing is far simpler, reducing software development time and probably also overall runtime.

References

[1] U. A. Acar and B. Hudson. Optimal-time dynamic mesh refinement: preliminary results. In *Proc. 16th Fall Workshop on Computational Geometry*, 2006.

[2] C. D. Antonopoulos, X. Ding, A. Chernikov, F. Blagojevic, D. S. Nikolopoulos, and N. Chrisochoides. Multigrain parallel delaunay mesh generation: challenges and opportunities for multithreaded architectures. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 367–376, New York, NY, USA, 2005. ACM Press.

[3] M. Bern, D. Eppstein, and J. R. Gilbert. Provably Good Mesh Generation. *Journal of Computer and System Sciences*, 48(3):384–409, June 1994.

[4] M. W. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry and Applications*, 9(6):517–532, 1999.

[5] D. K. Blandford, G. E. Blelloch, and C. Kadow. Engineering a compact parallel delaunay algorithm in 3d. In *Proceedings of the ACM Symposium on Computational Geometry*, 2006. To Appear.

[6] G. E. Blelloch, J. C. Hardwick, G. L. Miller, and D. Talmor. Design and Implementation of a Practical Parallel Delaunay Algorithm. *Algorithmica*, 24(3–4):243–269, Aug. 1999.

[7] S.-W. Cheng, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng. Sliver Exudation. *Journal of the ACM*, 47(5):883–904, Sept. 2000.

[8] A. Chernikov and N. Chrisochoides. Generalized delaunay mesh refinement: From scalar to parallel. In *15th International Meshing Roundtable*, pages 563–580, Birmingham, AL, Sept 2006.

[9] L. Chew, N. Paul, and F. Sukup. Parallel constrained delaunay meshing, 1997.

[10] L. P. Chew. Guaranteed-Quality Delaunay Meshing in 3D. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pages 391–393, Nice, France, June 1997. Association for Computing Machinery.

[11] N. Chrisochoides and D. Nave. Simultaneous mesh generation and partitioning for delaunay meshes, 1999.

[12] H. Edelsbrunner, X.-Y. Li, G. L. Miller, A. Stathopoulos, D. Talmor, S.-H. Teng, A. Üngör, and N. Walkington. Smoothing and cleaning up slivers. In *Proceedings of the 32th Annual ACM Symposium on Theory of Computing*, pages 273–277, Portland, Oregon, 2000.

[13] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proc. 19th ACM Symposium on Theory of Computing*, pages 315–324, 1987.

[14] S. Har-Peled and A. Üngör. A Time-Optimal Delaunay Refinement Algorithm in Two Dimensions. In *Symposium on Computational Geometry*, 2005.

[15] B. Hudson, G. Miller, and T. Phillips. Sparse Voronoi Refinement. In *Proceedings of the 15th International Meshing Roundtable*, pages 339–356, Birmingham, Alabama, 2006. Long version available as Carnegie Mellon University Technical Report CMU-CS-06-132.

[16] B. Hudson, G. Miller, and T. Phillips. Sparse Voronoi Refinement. Technical Report CMU-CS-06-132, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 2006.

[17] M. Isenburg, Y. Liu, J. R. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. *ACM Trans. Graph.*, 25(3):1049–1056, 2006.

[18] C. Kadow. *Parallel Delaunay Refinement Mesh Generation*. PhD thesis, Carnegie Mellon University, Pittsburgh, April 2004.

[19] M. Kulkarni, L. P. Chew, and K. Pingali. Using transactions in delaunay mesh generation. In *Workshop on Transactional Memory Workloads*, 2006.

[20] F. Labelle. Sliver Removal by Lattice Refinement. In *Proceedings of the Twenty-Second Annual Symposium on Computational Geometry*. Association for Computing Machinery, June 2006.

[21] X.-Y. Li. Generating well-shaped *d*-dimensional Delaunay meshes. *Theor. Comput. Sci.*, 296(1):145–165, 2003.

[22] X.-Y. Li and S.-H. Teng. Generating well-shaped Delaunay meshed in 3D. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 28–37. ACM Press, 2001.

[23] G. L. Miller, S. E. Pav, and N. J. Walkington. When and why ruppert's algorithm works. In *Proceedings, 12th International Meshing Roundtable*, pages 91–102. Sandia National Laboratories, September 14-17 2003.

[24] G. L. Miller, D. Talmor, S.-H. Teng, and N. Walkington. A Delaunay based numerical method for three dimensions: generation, formulation, and partition. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 683–692, Las Vegas, May 1995. ACM.

[25] G. L. Miller, D. Talmor, S.-H. Teng, and N. Walkington. On the radius–edge condition in the control volume method. *SIAM J. Numer. Anal.*, 36(6):1690–1708, 1999.

[26] S. Mitchell and S. Vavasis. Quality mesh generation in three dimensions. In *Proc. 8th ACM Symp. Comp. Geom.*, pages 212–221, 1992.

[27] D. Nave and N. Chrisochoides. Boundary refinement in delaunay mesh generation using arbitrarily ordered vertex insertion. In *CCCG*, pages 282–285, 2005.

[28] D. Nave, N. Chrisochoides, and L. P. Chew. Guaranteed-quality parallel delaunay refinement for restricted polyhedral domains. In *SoCG'02*, 2002.

[29] J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995. Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) (Austin, TX, 1993).

[30] M. Shephard, J. E. Flaherty, H. L. de Cougny, C. Ozturan, C. L. Bottasso, and M. Beall. Parallel automated adaptive procedures for unstructured meshes. Technical report, RPI, 1995. URL: http://www.scorec.rpi.edu/REPORTS/1995-11.pdf.

[31] D. Spielman, S.-H. Teng, and A. Üngör. Parallel Delaunay refinement: Algorithms and analyses. In *Proceedings, 11th International Meshing Roundtable*, pages 205–218. Sandia National Laboratories, September 15-18 2002. http://www.arxiv.org/abs/cs.CG/0207063.

[32] D. A. Spielman, S.-H. Teng, and A. Üngör. Time complexity of practical parallel steiner point insertion algorithms. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 267–268, New York, NY, USA, 2004. ACM Press.

[33] T. Tu, D. O'Hallaron, and O. Ghattas. Scalable parallel octree meshing for terascale applications. In *ACM/IEEE Super Computing Conference*, Seattle, WA, 2005.

[34] M. J. Turner, R. W. Clough, H. C. Martin, and L. P. Topp. Stiffness and deflection analysis of complex structures. *J. Aeronaut. Sci.*, 23:805–824, 1956.