Sparse Parallel Delaunay Mesh Refinement *

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

December 18, 2006

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.

Keywords: Computational geometry, Shared-memory parallelism

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

1 Introduction

The meshing problem is very old, going at least back to the 1950's – the early days of the finite element method [30]. The goal is to decompose the input domain up into simple pieces. These simple pieces are then used to represent functions over the domain such as temperature or velocity. By requiring that the mesh resolve features we can allow the represented 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 spaces to represent these functions for 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 [21]. 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 should compute a **constant-approximation** to the optimal mesh size. Finally the algorithm should be **work efficient** and fast.

The 2D meshing problem as we have stated was first posed by Bern, Eppstein, and Gilbert[2] who proposed a quadtree algorithm. Ruppert[25] gave an $O(n^2)$ time constant size approximation algorithm for the meshing problem using Delaunay refinement. Mitchell and Vavasis[22] extended the quadtree algorithm to 3D and proved that the Bern, Eppstein, and Gilbert algorithm was in fact also a constant size approximation algorithm. 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 angles are at least 90°. In 2D, one can obtain constant approximation algorithms for input angles of as small as roughly 20° [19]. Without the simplified input assumption in 3D, design and analysis of a practical algorithm are still very open.

The second issue has to do with the how we define the quality of the tetrahedra that are output. The most commonly desired shape quality guarantee for a tetrahedron is a bound on the **aspect ratio**: the volume ratio of the circumscribing sphere to the largest inscribed sphere. The **radius-edge ratio** of a simplex the radius of the circumscribing sphere and the length of its shortest edge. For Delaunay/Voronoi algorithms, generating tetrahedra with good radius-edge is the most natural shape condition for simple algorithms. In two dimensions, these criteria are equivalent, but in 3D and higher, the corner case of slivers arises. Our algorithm generates good radius-edge meshes, in Section 6 we discuss an extension that performs sliver elimination for obtaining good aspect ratio.

Given the simplified input assumption and using bounded radius-edge to define shape quality, we can now state our **main results**: We introduce a new parallel algorithm for refining a mesh in any fixed dimension. For all reasonable inputs, it is asymptotically work optimal. The parallel time is also close to the best possible. In 3D in particular, it generates a mesh with at most a constant factor more vertices than an optimal good aspect ratio mesh. For meshing in four or more dimensions we still generate a mesh with bounded radius-edge whose tetrahedron size is bounded below by the local feature size of the input.

Our parallel algorithm is based on our sequential algorithm **Sparse Voronoi Refinement** (SVR) [12]. In that work we presented an algorithm running in output-sensitive time $O(n\log(L/s)+m)$, with constants depending only on the dimension and prescribed element shape quality bounds. 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 $\Theta(n\log n+m)$.

The SVR sequential algorithm at a very high level alternates between two types of moves "break" and "clean". As designed, 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. 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 maintain a balanced tree and thus have been parallelized in a similar fashion [3].

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 meshers report to higher dimensional meshers about balls that must remain empty or nearly empty. Conversely, a higher dimensional mesher will report to a lower dimensional mesher 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 forced us to look much more closely at the sequential algorithm, making it simpler by removing unnecessary dependencies. We feel there are still many unnecessary dependencies. We also feel that it should be now possible to develop a provably work efficient distributed memory parallel algorithm.

2 Related Work

Finding parallel unstructured meshing algorithms has been a research topic since the early 1990's [26] and is still an important research topic today [6]. 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 only handle point sets, most oct-tree methods are fairly easily parallelize for point input, but higher dimensional features have proven difficult to handle efficiently [11, 29]. In [28], an $O(\log m)$ parallel time bound with $O(m \log m)$ 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. [7, 8, 24, 23, 4, 15]. The state of the art for parallel meshing features is the algorithm of Spielman et al. in [27]. They present a very parallel 3D meshing algorithm that handles 1D features with a $O(\log^2(L/s)$ polylog$(n))$ time bound, 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). It also maintains a mesh for the entire mesh domain – we refer to this mesh as the d-mesh because it is in dimension d. At the end, the algorithm returns the final d-mesh as its output.

The meshes do not necessarily coincide at any time midstream, and so the algorithm maintains the geometric correspondance between meshes of different dimension: a Voronoi cell in a mesh in dimension i has a pointer to every intersecting Voronoi cell in any intersecting mesh of dimension $j < i$.

Finally, the algorithm maintains a work set of Voronoi nodes to be inserted. Each item is tagged with a type, the dimension, a handle into the appropriate mesh, and the geometric point desiring to be inserted. There are three types:

items represent poorly shaped Voronoi cells that must be remedied by refinement.

items represent balls (d-spheres) that contain and protect Voronoi cells whose destruction has been ordered by some higher-dimensional mesh desiring refinement in the region.

Finally, items 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.

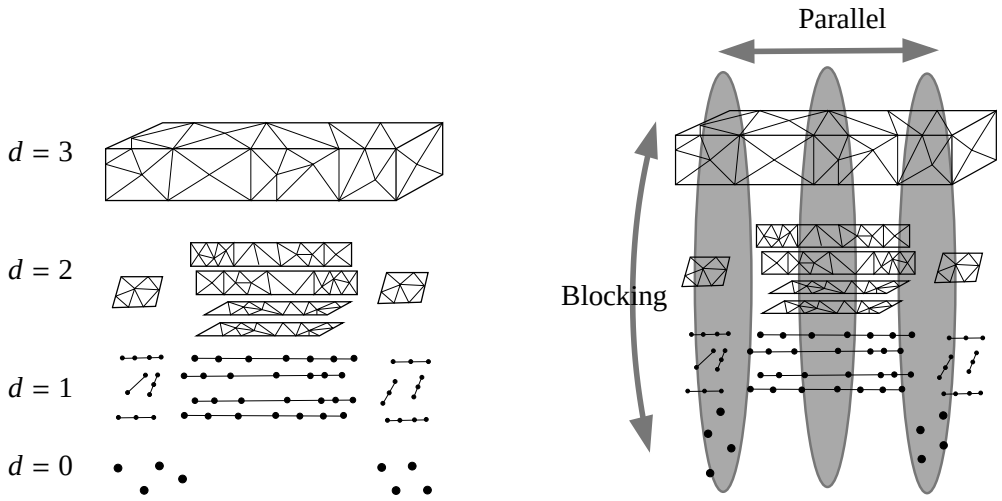

Figure 1: *Left:* SVR works with a mesh of each feature at each dimension. We have the 3D spatial mesh, a 2D mesh for each constraint surface, a 1D mesh for each of the edges of those surfaces, and zero-dimensional meshes for the corners. The meshes are naturally nested by containment of features. All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

The work set is ordered with moves first, in increasing order of dimension; then moves in increasing dimension; then the moves. 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 appear in the final output mesh.

For further details on this sequential algorithm, see [12]. The rest of this paper will work 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 the workset quite independent of the meshing problem at hand; the algorithm is somewhat standard. Then, 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.

P W S (S: a work set)
1: **return if** S is empty
2: $S' \leftarrow \emptyset$ {set of items to defer}
3: $G \leftarrow$ C (S)
{Defer Items Blocked by Lower Dimensional Work}
4: **for** each work item $w \in$ B (S) in parallel **do**
5: remove B (w) from C
6: add B (w) to S'
7: **end for**
{Process all the Unblocked Work Items}
8: Colour the conflict graph G using $k \in O(\Delta)$ colours.
9: Let G_i be the set of work items of colour i.
10: $P \leftarrow \emptyset$ {the set of processed items}
11: **for** $i = 1$ to k **do**
12: **for** each work item $w \in G_i$ in parallel **do**
13: **if** $w \in G$ **then**
14: remove M (w) from G
15: add w to P
16: process w similarly as the sequential algorithm
17: **end if**
18: **end for**
19: **end for**
{New Work Set is the Deferred Events and the New Events}
20: **return** $S' \cup$ N W (P)

Figure 2: The generic algorithm. Given a workset, we compute the conflict graph, defering blocked items until the next round. Then we colour the graph, which gives us a safe ordering for the work items. Finally, we iterate over each colour and perform the work. Items that have been made moot by a work item in a prior iteration of the loop are simply ignored.

4.1 Generic algorithm

As input to the function, we assume we are given a set S of items 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 items on the workset. The black box takes an item, processes it, and returns a set of additional items to work on. Finally, the fourth oracle updates the work set given the set of work items 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.
- 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 items 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 have 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 is direct from C and B being constant-degree graphs.
- Colouring the graph is $O(\lg|S|)$ parallel depth and $O(|S|)$ work. We can also trade depth and work using an algorithm of Goldberg, Plotkin, and Shannon [10], and get $O(\lg^*|S|)$ depth for $O(|S|\lg^*|S|)$ work.
- The main loop sees each work item at most once, does $O(1)$ work per item looking up w to see whether it has become moot, and if not, does $O(1)$ more work removing from S the items 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 item on the workset is either processed, mooted, or blocked. Blocked items will reappear next round unless they were mooted, alongside new items generated by the processed items.

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. What is left is discover the number of rounds.

4.2 Processing

The items on the workset are tuples listing: (1) a feature F; (2) a point p to attempt to insert; (3) a ball (c, r) corresponding to a Voronoi node; (4) a cell $V_{M_F}(v)$ associated with the item; (5) the reason, defined below, that this item is on the workset.

Processing a work item means that we insert p into M_F using a variant of the F I call described in the sequential algorithm to update the mesh and the correspondence between meshes. The conflict and mooting rules will be written so that this is safe to do in parallel.

F 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 shall show that the mesh is of constant degree, and that this implies that the first and third steps are constant time. Therefore, we can do those two steps sequentially. The second step, however, is bounded only by $O(n)$, so we must parallelize that step. Thankfully, we can do it in the trivial fashion, taking $O(1)$ depth and linear work on a CRCW.

4.3 Populating the work set

Given a cell $V_M(v)$ in some mesh M that corresponds to a feature F, we can determine whether that cell and feature define a work item, using the rules that follow. After processing, then, we can examine every cell created by F 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.

Rule 1 (clean move) *If $R_M(v)/r_M(v) > \tau$ then the work item inserts a point* $\mathrm{far}_M(v)$ *with reason , where* $\mathrm{far_M(v)}$ *is the farthest Voronoi node from v.*

Rule 2 (break move on a Steiner) *If v has containment dimension d, and $V_M(v)$ includes a pointer to at least one uninserted point p, then the work item tries to insert the point p with reason - .*

Rule 3 (break move on an input) *If v has containment dimension $i < d$, and $V_M(v)$ includes a pointer to at least one uninserted point p, and $F = \Omega$, then the work item inserts* $\mathrm{far}_M(v)$ *with reason - .*

Definition 4.1 *A cell $V_M(v)$ is **partially resolved** if v and at least one of its neighbours in M both appear in the d-mesh.*

Rule 4 (weak encroachment move) *If $V_M(v)$ intersects two lower-dimensional cells $V_{M_F}(u_1)$ and $V_{M_{F'}}(u_2)$, both of which are partially resolved, and F and F' are distinct (the features do not meet at any point), then we consider defining a set of work items. For each Voronoi node (a, r_a) of $V_{M_F}(u_1)$, check whether it intersects any Voronoi node (b, r_b) of $V_{M_{F'}}(u_2)$. If so, add the two work items $\langle F, a, r_a, V_{M_F}(u_1),$ - $\rangle$ and symmetrically $\langle F', b, r_b, V_{M_{F'}}(u_2),$ - $\rangle$.*

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 [27].

Having defined these rules, we will modify them in the following way:

Rule 5 (warping rule) *Consider a work item $w = \langle F, p, (c, r), V_{M_F}(v)\rangle$. If w has not already warped, then for every uninserted input vertex u that intersects the ball $B(c, r)$, check whether $|pu| < (1-\epsilon)r$. If so, modify w: Replace p with u.*

We say *p warps* to *u* when this happens. The goal is to ensure that no Steiner point is ever inserted too near an input vertex, which would break our sizing guarantees. A similar rule ensures that no Steiner point is ever inserted too near a higher-dimensional input feature.

Rule 6 (yielding rule) *Consider a work item $w = \langle F, p, (c, r), V_{M_F}(v)\rangle$, with $\dim(F) = i$. For every cell $V_{M^-}(u)$ on a feature F^- in dimension $j < i$, check: if p does not encroach on $V_{M^-}(u)$, do nothing. Otherwise, if either $p = c$ (that is, p is a Steiner point) or if there is some other point $u \in M_F$ that encroaches on $V_{M^-}(u)$, create the work item $\langle F^-, q, (q, r_q), V_{M^-}(u),$ $\rangle$.*

We check the clean, break, and weak-encroachment rules on all cells affected during F I . Having done so, we check the warping rule once. Then we check the yielding rule. Finally, we check the warping rule again (to warp the new items created by the yielding).

Each of the first few rules are very cheap to check: the clean move is a simple division operation. The break move requires noting, in the the parallel redistribution step of F 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.

The warping and yielding rules need to check a potentially large number of lower-dimensional cells. However, each of those items is only checked by a constant number of cells, all of which were processed. Therefore, we can parallelize the checks in constant depth and linear work on a CRCW machine. Although these rules need to check a large number of cells, they can only spawn $O(1)$ new work items.

4.4 Conflict graph

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

With each cell, we associate a *protected zone* that describes the area where a point being inserted can possibly modify the cell's geometry (and topology). If two work items both modify a cell concurrently, this might require some special coding. Thus, we will put a edge between any two such work items. More formally:

Definition 4.2 *Given a mesh vertex v, the **protected zone** of v is the union of all empty balls that have v on their surface.*

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:

Fact 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 work items a and b if their respective protective zones intersect, and a is on a subfeature or the same feature as b.

Lemma 4.4 *The conflict graph has degree $O(1)$.*

Proof A radius-edge quality Delaunay triangulation has constant ply [20]: that is, any point intersects only a constant number of Delaunay circumballs. 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 work items that both encroach upon the same cell. Clearly, there can only be a constant number of such. ■

The results in this section showed that (a) the conflict graph allows parallel processing of work items 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 BLOCKS graph

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

While we could afford to do this in our parallel code, 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 blocks on any clean move with which it has a conflict edge.
- Any move blocks on 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 work item 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 [12].

We can sketch the proof as follows: The algorithm quickly makes progress. Any event 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 event enters the work set within $O(\log L/s)$ rounds.

First we show that within $O(\log L/s)$ 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 events, 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 $O(\log L/s)$ 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.

Lemma 5.1 ([13, Theorem 8.5]) *At all times during the algorithm, every mesh has quality $\tau' \in O(\tau)$.*

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.

Lemma 5.2 (Packing Lemma: [13, Lemma 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.

Lemma 5.3 (α-Lemma: [13, Lemma 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 an item is always either processed, mooted, or blocked every round. We want to make sure that an item is not blocked for too long – in fact, we will require that it be blocked only $O(1)$ rounds.

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

Proof If the two items are of the same dimension, then by the quality condition they have the same size. Otherwise, we can instead invoke the α-proof for the same result. ■

Lemma 5.5 *Given a work item a blocking on items in a lower-dimensional mesh F, at most $O(1)$ insertions can be performed in F before a is no longer blocked by F.*

Proof By the prior lemma, we know that all items in F that are blocking a have about the same size as a. By the Packing Lemma, then, we can only insert $O(1)$ points into F such that the work items that insert them will block a. ∎

Theorem 5.6 *An event on the work set blocks $O(1)$ rounds before it is either processed or mooted.*

Proof 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.

Lemma 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 $|pq| \leq \text{cfs}_M(p)$ and $NN_{M'}(q) \in \Omega(\text{cfs}_M(p))$.*

Proof 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. ∎

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

Proof 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 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. ∎

Theorem 5.9 *After $O(\lg L/s)$ rounds, the* P R *algorithm has produced a mesh of quality $\tau' \in O(\tau)$ that conforms to local feature size.*

Proof 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.

Lemma 5.10 (Clean moves grow) *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 work item w that inserted u: $R_M(v) \geq \frac{\tau\epsilon}{2} r(w)$.*

Proof We know that $R_M(v) \geq \tau r_M(v)$ 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 work item 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 $NN_{M'}(u) > \epsilon r(w)$. ■

Lemma 5.11 *Consider an work item b. The work item was spawned by some higher-dimensional work item 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 [13, proof of Lemma 7.4]. Thus, so long as $\tau\epsilon > 2^{d-3/2}$, 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.

Theorem 5.12 *Given a strongly conformal mesh of quality τ', the* P R *algorithm takes $O(\lg L/s)$ rounds before reaching quality τ.*

6 Parallel Sliver Removal

So far, we have shown how to produce a good aspect ratio Voronoi mesh. This dualizes to a good radius-edge Delaunay mesh. In 3D, bounded radius-edge meshes are unsuitable for some 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 [5, 9, 17, 18, 16].

In the Li and Teng paper [18] they give an algorithm that inserts at most a constant fraction more points in order to eliminate all slivers. In this section, we overview how to parallelize the sliver elimination algorithm of Li and Teng as an extension of Parallel SVR. As input, their algorithm takes a conforming, bounded radius-edge mesh such as the one output by P R . Iteratively, they remove a sliver T by inserting a point p into the circumball of T. Adding this new point may introduce a new sliver with p as a vertex, may introduce a new bad radius-edge cell, or may encroach on a protective ball. They made the following powerful observation:

Theorem 6.1 ([18, Theorem 4.1]) *Given a bounded radius-edge mesh then there is a point in a suitably defined picking region near the circumcenter, whose insertion generates no small slivers.*

Thus, by carefully selecting points from the picking region we can ensure that destroying slivers only generates geometrically larger slivers, guaranteeing swift termination.

We can now populate the work set as in P R with one new rule:

Rule 7 (sliver) *If a Delaunay tetrahedron is skinny, then the work item tries to insert a point p in the picking region that does not generate a smaller sliver.*

This rule presents obvious modification to the blocks graph from subsection 4.5. A sliver move simply is blocked by any conflicting non-sliver move.

The run time follows by methods analogous to Lemma 5.10. Thus we still get at most $O(\log L/s)$ 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. Under such an assumption, the bounds we've proven are that we can achieve the optimal work bound of $O(n \lg n + m)$, and our algorithm 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.

A more practically-minded way to look at these results is to say that our algorithm can accomodate up to about $O(m/\lg^2 m)$ processors. But if we are running the algorithm in parallel, most likely that is because m is very large (millions or billions), and it is unphysical to actually construct this many processors in a shared-memory machine.

Instead, we can view the results as predicting 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 [6], out-of-core and streaming computation [14], and dynamic mesh refinement [1].

Furthermore, this algorithm should be easy to fit in a distributed-memory framework like that of Chernikov *et al.* [6, 24]. Their initial work partitioned the meshing domain across a large number of distributed-memory nodes, then meshed each part of the domain sequentially on each node, and stitched the results together. Each node, however, will have four or eight processing cores. Their more recent work has started to examine how to exploit this fine-grained parallelism but noted that their results paid a large penalty compared to the current best sequential code (namely, Shewchuk's *Triangle*). We can hope that our results here will fill in this gap, especially in three dimensions.

References

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

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

[3] Marshall W. Bern, David Eppstein, and Shang-Hua Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry and Applications*, 9(6):517–532, 1999.

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

[5] Siu-Wing Cheng, Tamal Krishna Dey, Herbert Edelsbrunner, Michael A. Facello, and Shang-Hua Teng. Sliver Exudation. *Journal of the ACM*, 47(5):883–904, September 2000.

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

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

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

[9] Herbert Edelsbrunner, Xiang-Yang Li, Gary L. Miller, Andreas Stathopoulos, Dafna Talmor, Shang-Hua Teng, Alper Üngör, and Noel Walkington. Smoothing and cleaning up slivers. In *Proceedings of the 32th Annual ACM Symposium on Theory of Computing*, pages 273–277, Portland, Oregon, 2000.

[10] 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.

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

[12] Benoît Hudson, Gary Miller, and Todd Phillips. Sparse Voronoi Refinement. In *Proceedings of the 15th International Meshing Roundtable*, Birmingham, Alabama, 2006.

[13] Benoît Hudson, Gary Miller, and Todd Phillips. Sparse Voronoi Refinement. Technical Report CMU-CS-06-132, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 2006.

[14] Martin Isenburg, Yuanxin Liu, Jonathan Richard Shewchuk, and Jack Snoeyink. Streaming computation of Delaunay triangulations. *ACM Trans. Graph.*, 25(3):1049–1056, 2006.

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

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

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

[18] Xiang-Yang Li and Shang-Hua 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.

[19] Gary L. Miller, Steven E. Pav, and Noel 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.

[20] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel 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.

[21] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. On the radius–edge condition in the control volume method. *SIAM J. Numer. Anal.*, 36(6):1690–1708, 1999.

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

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

[24] Démian Nave, Nikos Chrisochoides, and L. Paul Chew. Guaranteed-quality parallel delaunay refinement for restricted polyhedral domains. In *SoCG'02*, 2002.

[25] Jim 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).

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

[27] Daniel Spielman, Shang-Hua Teng, and Alper 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.

[28] Daniel A. Spielman, Shang-Hua Teng, and Alper 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.

[29] Tiankai Tu, David O'Hallaron, and Omar Ghattas. Scalable parallel octree meshing for terascale applications. In *ACM/IEEE Super Computing Conference*, Seattle, WA, 2005.

[30] 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.