Optimal-time dynamic mesh refinement: preliminary results

Umut A. Acar* Benoît Hudson†

Abstract

We present early results on a dynamic mesh refinement algorithm. Using a variant of the Sparse Voronoi Refinement algorithm and applying the technique of Self-Adjusting Computation, we find that we expect to run in $O(\text{polylog}\, n)$ time per update on points sets in arbitrary dimension. This is based on some theoretical results, along with experimental results from an implementation.

1 Sparse Voronoi Refinement

Mesh refinement solves the following problem: given a point set $\mathcal{P} \in R^d$, add additional Steiner points such that the Delaunay tessellation of the augmented set creates only simplices of good quality. The quality metric usually used, because it is tractable to theoretical analysis, is the radius-edge condition: given the radius of the circumscribing ball around s, and the length of its shortest edge. If the ratio is less than $\sqrt{2}$, the simplex is said to be of good quality.

The problem has seen significant study in the past 20 years, driven by graphics and scientific computing applications. The current state of the theoretical art is the Sparse Voronoi Refinement (SVR) algorithm of Hudson, Miller, and Phillips [HMP06], which solves the problem outputting only a constant factor more vertices m than is optimal, in output-sensitive time $O(n \lg n + m)$ assuming polynomial spread[1].

The basic operation of SVR is to try to insert the circumcenter of a simplex s. Before actually performing the insertion and incrementally retriangulating using the Bowyer-Watson method, SVR checks whether there is any uninserted point "near" the circumcenter being inserted, where near is defined being within as a constant fraction $0 < k < 1$ of the circumradius of s. If not, SVR inserts the circumcenter; but if there is such a point, it instead *yields* to the input point. This yielding ensures that the input points are all eventually recovered, but not so early that the input points would cause any arbitrarily bad-quality simplices.

By controlling the quality, SVR controls the runtime. Each point insertion takes constant time (hence the $O(m)$ term), while checking for yielding takes $O(\lg n)$ time per vertex.

2 Self-Adjusting Computation

Acar [Aca05] described the self-adjusting computation (SAC) model as a method to automatically dynamize static algorithms. In this model, we first perform an *initial run* by running the static algorithm on a fixed input. During the initial run, an underlying run-time system generates an execution *trace* of the computation. The trace contains information about the memory locations read, the operations executed on the values read, writes to memory, and the control dependences between executed operations.

After the initial run, we can change the input by mutating the contents of memory locations, then update the output and the trace by running a *change-propagation algorithm*. The change propagation algorithm starts by re-executing the first operation that depends on the value of a changed memory location. When re-executed, the operation may change the contents of other memory locations and invoke a different next operation. The change-propagation algorithm recursively propagates the changes, stopping a thread of propagation when it writes the same result as it had written in the previous run, or when it finds a previously computed result via memoization. Change propagation simulates rerunning from scratch, producing the same output and the trace as would have been produced by a from-scratch execution with the changed input.

For certain computations, change propagation takes time linear in the size of the symmetric difference between initial and final traces. More precisely, define an algorithm $O(f(n))$-*stable* for a class of (input) changes, if the size of the symmetric set difference between the traces of that algorithm on inputs related by the changes is bound by $O(f(n))$. It is shown that if a static algorithm is $O(f(n))$-*stable*, then self-adjusting computation can respond in $O(f(n))$ time [Aca05].

*Toyota Technological Inst., Chicago, IL. umut@tti-c.org

†Carnegie Mellon University, Pittsburgh, PA. bhudson@cs.cmu.edu. This work was performed while the author was at Toyota Technological Inst.

[1]The spread is defined as the ratio between the diameter of the input space and the distance of the nearest pair of input points.

3 Stability of SVR

Three basic properties of SVR indicate that it is $O(\text{polylog}\, n)$-stable:

1. At every step of the algorithm, the mesh has constant degree per vertex. Thus insertions always take $O(1)$ time to perform (that is, insertions perform $O(1)$ operations).
2. Second, a corollary of Ruppert's proof of optimal output size implies that at most, a newly-added vertex requires $O(\lg n)$ Steiner vertices to achieve a quality mesh.
3. Hudson *et al.* prove that the point location charges (to check for yielding in tryInsert) sum to $O(\lg n)$ per vertex.

The remaining barrier to a full stability proof for SVR is that an attempted insertion may yield differently after a new input point is inserted. We can (and, in the implementation, do) somewhat mitigate this by picking a random vertex to yield to when given the option. However, eventually tryInsert *must* change its result: after all, the new input must ultimately appear in the mesh.

4 Experimental Validation

Given the high likelihood of SVR being dynamically stable, but in the absence of hard proof, we produced an SML implementation using the self-adjusting-computation library developed by Acar, Blelloch, Blume, Harper, Tangwongsan [ABBT06, ABB+06]. The experiment we ran was the following: we initialized SVR with an empty input. Then we added to the initially empty input, one by one, 200 points around a circle. During each addition to the input list, we counted the number of line-side primitives performed during point location, and the number of Steiner points inserted into the mesh.

The results for line-side tests are summarized in Figure 4. First, notice that the number of line-side tests grows only slightly faster than linearly; approximately $O(\lg^3 n)$ per incremental addition to the input. This confirms our suspicions that we should see polylogarithmic behaviour. The number of point insertions appear to scale linearly on this point set, as it does in the static case.

5 Conclusions

Our experiments indicate that SVR, a time-optimal static algorithm, likely retains its properties in the dynamic case up to logarithmic factors. This remains to be formally proved, but appears tractable.

The real problem we want to solve with a dynamic mesh refiner is that of producing a fast moving mesh code. Acar, Blelloch, Tangwongsan, and Vittes [ABTV06] have shown how to use the self-adjusting-computation framework to automatically *kinetize* self-adjusting code. A fast and correct kinetic mesher would be of huge benefit to Lagrangian finite element simulations, and to the graphics community.

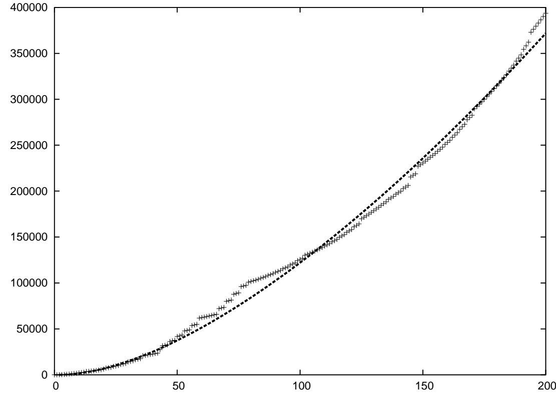

Figure 1: Plot showing the number of line-side tests performed against the number of input points added during the incremental insertion of 200 points around a circle. For comparison, the plot shows the curve $40n\lg^3(n)$, which tracks the experimental data well.

References

[ABB+06] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A library for self-adjusting computation. *Electronic Notes in Theoretical Computer Science*, 148(2), 2006.

[ABBT06] Umut A. Acar, Matthias Blume, Guy E. Blelloch, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[ABTV06] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge Vittes. Kinetic algorithms via self-adjusting computation. In *European Symposium on Algorithms (ESA)*, 2006.

[Aca05] Umut A. Acar. *Self-Adjusting Computation.* PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[HMP06] Benoît Hudson, Gary Miller, and Todd Phillips. Sparse Voronoi Refinement. In *IMR*, 2006.