3D Delaunay triangulation

Libin Lu, Zhuoheng Yang, Weijing Liu

December 22, 2012

Part I
Introduction

Our project project topic is parallel 3D Delaunay triangulation. In 2-D, a Delaunay triangulation for a set P of points in a plane is a triangulation $DT(P)$ such that no point in P is inside the circumcircle of any triangle in $DT(P)$. In 3-D, a Delaunay triangulation for a set P of points in space is a triangulation $DT(P)$ such that no point in P is inside the sphere of any tetrahedron in $DT(P)$. With this property, Delaunay triangulation can avoid skinny triangles. And the mesh generated by Delaunay triangulation can be used in many scientific computing areas, such as the finite element method and the finite volume method of physics simulation because the angle guarantee. Delaunay triangulation can also be used in modeling terrain or other objects. In our project, we try to parallel Bowyer-Watson algorithm for Delaunay triangulation using task based parallelism.

More formally, the input of the algorithm is an array of XYZ coordinates, which represents the points. The output of the algorithm is an array of tuples. Each tuple has 4 indices which represent a tetrahedron. The tetrahedrons should meet the Delaunay triangulation rule above. The program aims to solve an instance with $10,000$ to $1,000,000$ points. At the high end, the program is limited by available memory, or the high complexity of Delaunay triangulation depending on specific machine. At the low end, synchronization overhead overcomes the benefit of parallelism.

Part II
Bowyer-Watson algorithm

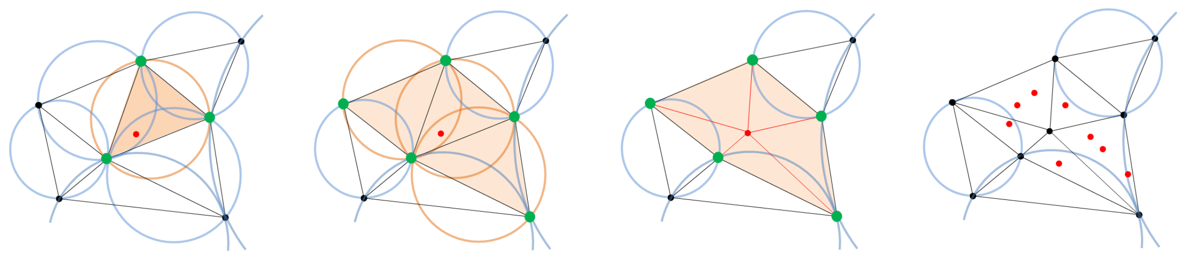
Bowyer-Watson algorithm is an incremental insertion algorithm for Delaunay triangulation. It is named after Adrian Bowyer and David Watson. They devised it independently of each other at the same time, and each published a paper on it in the same issue of *The Computer Journal.*

The Bowyer–Watson algorithm[2, 3] works by adding points, one at a time, to a valid Delaunay triangulation of a subset of the desired points. The description of this algorithm is simple.

Every iteration we try to insert one point to the Delaunay triangulation. We first find all existing triangles whose circumscribing circle contains the new point. This can be done to find the triangle which contains the new point first. Then the neighbors of this triangle are searched and then their neighbors, etc., until no more neighbors have the new point in their circumcircle. Then we delete these triangles, leaving a star-shaped polygonal cavity. At last we join the new point to all edges on the boundary of the cavity. Iterate over all points, when every point is inserted into the Delaunay triangulation we get the final valid Delaunay triangulation of all the points.

Part III
Parallelism

We try to parallel Bowyer-Watson algorithm, it is interesting to note that when we try to insert points into our Delaunay triangulation, the points in different area may be independent from each other during the insertion phase. Use this idea we can form a task based parallelism. The idea is similar to the paper[1] by Daniel K. Blandford. They



(a) Step one (b) Step two (c) Step three (d) Step four

Figure 1: Steps of insertion kernel

use a more complicate but more compact data structure in their implementation for engineering scale application. Our data structure is fairly simple but easy to implement.

The task based idea is the same. Each task represents a triangle in space containing some points. And each thread tries to get a task and insert points, during the insertion phase, triangles far from each other can perform insertion at the same time, thus threads can execute tasks simultaneously. And all the threads perform the same insertion kernel.

Insertion kernel

We will explain our insertion kernel in 2D. It can be easily generalized to 3D. We represent our Delaunay triangulation using a concurrent hashmap. The key of the hashmap is the three vertices of a triangle. The data is three neighbor triangles and a list P_list which contains all the points in this triangle not been inserted yet. We maintain a task queue, and all the triangles whose P_list are non-empty are in the task queue. Initially, we create a bounding triangle containing all the points we want to triangulate. And this bounding triangle is put into our task queue as initialization of the task queue. Then all the threads try to get a task from task queue randomly, each task associated with a triangle having a non-empty P_list. The thread performs the same insertion kernel in four steps.

In figure 1a, The shaded triangle say Tri_A represents a task obtained by some thread, the red point say P is the first point in the P_list, and we want to insert P into our Delaunay triangulation. We try to lock all the three vertices of triangle Tri_A first, and test if the Tri_A is still in the hashmap, if it's not in the hashmap, we abort the task, and unlock the vertices.

If triangle Tri_A is still in the hashmap we perform a tree search in the neighborhood of Tri_A as the second step shown in figure 1b. The tree search is performed recursively, we first test if the circumcircles of the three neighboring triangles of Tri_A contain the point P. Then for the triangles whose circumcircles contain P, we test their neighboring triangles to see if the circumcircles of the neighboring triangles contain P, do the search recursively. The return criteria is when the circumcircle of the testing triangle doesn't contain P, we return and don't search for the neighboring triangles further. Using this method we can find all the triangles in our current Delaunay triangles whose circumcircle contain P. The proof can be found in the original paper of Bowyer-Waston incremental algorithm for Delaunay triangulation[2, 3]. During the progress of searching the triangles whose circumcircles contain P, we try to lock the vertices of the triangle when we find its circumcircle contain P immediately. This may cause a deadlock when several threads trying to lock some points at the same time. So we try to lock the vertices, if fail to lock the vertices, we unlock all the vertices having been locked so far and put the task back into task queue to try again later, also we abort the current task.

If we successfully lock all the vertices we want to lock, in the third step we delete all the triangles whose circumcircle contain P from hashmap, and remain a star-shaped polygonal cavity. We connect the point P with the boundary edges forming several new triangles as shown in figure 1c.

In the last step as shown in figure 1d, we sort all the remaining points in the star-shaped polygonal cavity into the new triangles and insert new triangles into the hashmap. For those new triangles whose P_list are not empty, we put it into the task queue. Also we should update the neighbor triangles for those triangles whose neighborhood has been changed. By now the thread has finish executing the task, and we can release the locked vertices. After finishing this task, the thread continually try to get task from task queue whenever the queue is not empty.

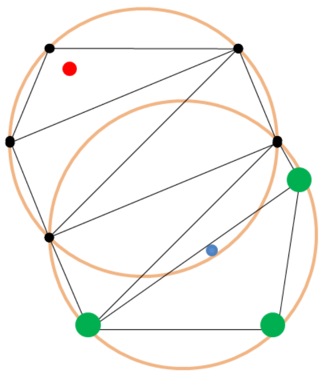


Figure 2: Fail situation with indexing order lock

The deadlock

We notice that there may be a lot of contentions due to the try lock fails when nearby triangles try to insert points at the same time. We may use the method of locking in indexing order. But this classical method of preventing deadlock may not work here. Although we are locking vertices, we are actually locking the triangle. If we want to perform locking in indexing order, we have to get all the vertices we want to lock first and then lock all the vertices in some order. But during the recursive searching of those triangles, if we don't lock the vertices immediately, some triangles may be deleted by other threads. When we find all the triangles we want lock, then we try to lock the vertices, although we may successfully lock the vertices, but it may not be the vertices of the triangles we found before, the triangles may already been deleted. Actually when we perform a task associate with some triangle say Tri_A, we first lock the vertices of Tri_A, then the neighbor triangles of Tri_A will not be deleted by other threads since we lock all the vertices of Tri_A, if we know that the triangles whose circumcircle contain the insertion point P are just the neighbor triangles, we can perform the lock of vertices in indexing order. But there may be some triangles whose circumcircles contain the insertion point P but they're not the neighboring triangles of Tri_A. As shown in figure 2, the triangle Tri_A with green vertices wants to insert the blue point, we lock the three green vertices and search for other triangles whose circumcircles contain the blue point. In the process, we find the upper most triangle Tri_B whose circumcircle contains the blue point, if we don't lock Tri_B immediately, we just store it in some list, and we continue searching for other triangles. During the searching, other threads may pick Tri_B as a task to insert the red point. Then when we finish searching and begin to lock the vertices of triangles containing the blue point, the triangle Tri_B may already been deleted. Although we could lock the three vertices of Tri_B, but the Tri_B has been deleted, then our triangulation will go wrong.

Another issue in insertion kernel is the geometry computation when we do test of whether a point is in circle or whether a point is in triangle. The numerical precision error may cause our computation go wrong same as the issue in ray tracing computation. For this project we simply trust the test functions in predicate.c library which has robust performance.

Part IV

TBB or Customized task queue.

Originally we try to use TBB(Thread Building Blocks) to perform the task scheduling. TBB could handle the task queue and distributes task to thread automatically. And TBB is fast because it's unfair task scheduling. It favors the task recently in queue. So for the tree relation tasks, TBB is a good choice. However, the relations between our tasks are not tree like. The parent task immediately finishes after it creates child tasks, so we won't benefit from TBB in this sense. Moreover, one of the characteristics of our tasks is that adjacent tasks are likely to have common points, which means that executing them at the same time will lead to failure locking, or more seriously, starvation. Experiment result supports the prediction above.

In order to avoid failure locking problem, we try to implement our customized task queue. We use a $std :: deque$ to implement the task queue, the improvement in performance seems not good enough due to the high complexity, thus we switch to use Fibonacci heap to implement the task queue. Fibonacci heap has amortized complexity $O(1)$ with insertion and $O(\log n)$ with delete min. To perform random distribution of task, we attach a random number to the task when it is inserted to the Fibonacci heap. When we fetch a task from the heap, we choose the one with smallest number. Use Fibonacci heap to implement task queue does improve the speed a lot. It reduces the cost of task insertions and task deletions. Another optimization we used is limiting the number of active tasks when there are too few tasks in task queue. This helps reduce the chance of starvation.

Part V

Memory Locality

For locality of memory access, we found it important to ensure that vertices that were close spatially stayed close in memory. To ensure this, as a preprocessing step we reordered the vertices using *x-y-z* cuts.

X-Y-Z Cuts

The procedure of $x - y - z$ cuts is described below:

1. Find which of the x, y, z axes has the greatest $diameter(D)$.
2. Find the approximate $median(M)$ of D.
3. Partitions the points into two sides using M.
4. Recursively apply $x - y - z$ cuts to one side of points first, and then the other.

And figure 3 shows a 2D example(x-y cuts). Figure 3 presents a bounding rectangle and several cuts it has. 1st cut is made because the diameter of X axis is the greatest. After the first cut, points are partitioned and also the rectangle. We next recursively process the left side of rectangle first . 2rd cut is made because the diameter of Y is the greatest. The recursion stops at a rectangle which contains less than 2 points.

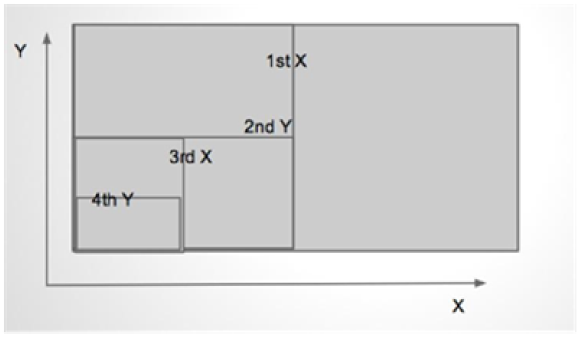
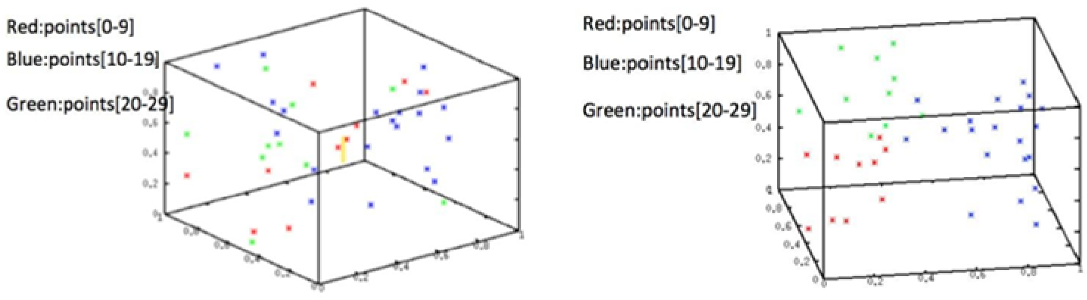


Figure 3: x-y cut

Figure 4 presents the effects of applying spatial sort to a set of random points. Notice that points in these figures are colored according to their memory locations. As figure 4a shows, the coloring tells points are generated randomly within the box. On the other hand, the coloring of figure 4b shows that memory locations of points are reordered according to their spatial relations.



(a) Before sorting

(b) After sorting

Figure 4: Compare index of non-sorting and x-y-z sorting

X-Y-Z Optimization

Conceptually, $X - Y - Z$ Cuts is similar to Quicksort, whose running time is $O(n \log n)$. As a preprocessing step, the cost of applying $x - y - z$ cuts should be light-weighted. So we did two optimizations on it: 1) Inplace sorting and 2) parallel $x - y - z$ cuts.

Because the number of points we were targeting for parallel 3D Delaunay triangulation is large-scale, i.e. 1 million, temporary memory allocation for processing sorting will be costly and so should be avoided. We solved this by using the inplace partitioning ideas from Quicksort. The temporary memory usage for sorting has been minimized as much as possible. Also we paralleled the algorithm by recursively applying sorting to both sides of points concurrently. As a result of these two optimizations, $x - y - z$ cuts has become a light-weight preprocessing step by which memory locality for triangulation can be improved.

Part VI

Result and Conclusion

The most time-consuming step varies in different stage. In the beginning, there are many tasks in the task queue. The chance that two threads are locking the same points is very small. In this stage, the bottle neck is geometry computing. When most of the tasks are processed, the remaining tasks in the task queue are highly relative. In this stage, many locking failure will happen, or tasks are forced to be processed serialized (to decrease the chance of starvation). The bottle neck is synchronization overhead. It is simple to parallel the algorithm, but it is difficult to achieve high efficiency due to the essential of this algorithm. Please refer to “TBB or Customized task queue” for our approach to gain high efficiency. Actually the way we use to get random task from a heap may cause a problem in the long run for a large problem size. The task attached with large random number has less probability to be executed, and in the long run we may accumulates a lot of tasks attached with large numbers, then the new created task may have high probability to be executed. This could be an optimization for future work. But equally probability random task queue may not have the least failure lock rate as test in the paper[1] by Daniel K. Blandford. It still remains a question what kind of task queue is optimal for this incremental parallel Delaunay triangulation algorithm.

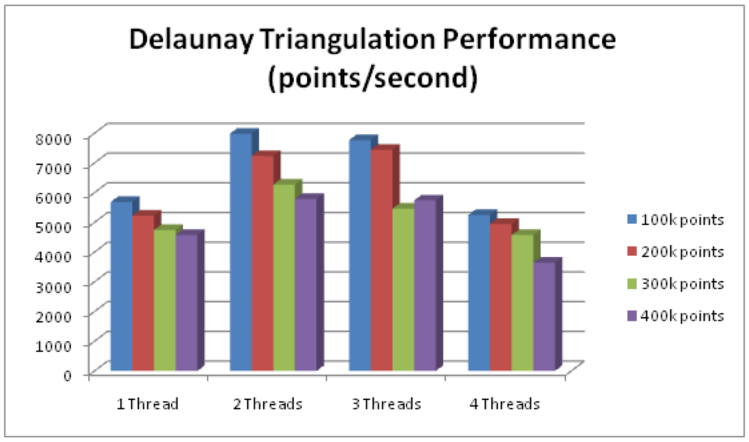


Figure 5: Performance using 1, 2, 3 and 4 threads

From the performance report in figure 5, we know that parallelism accelerates the program giving only two or three threads. Moreover, from the jobs rate we can conclude that failure locking is the most important factor which hurts the performance.

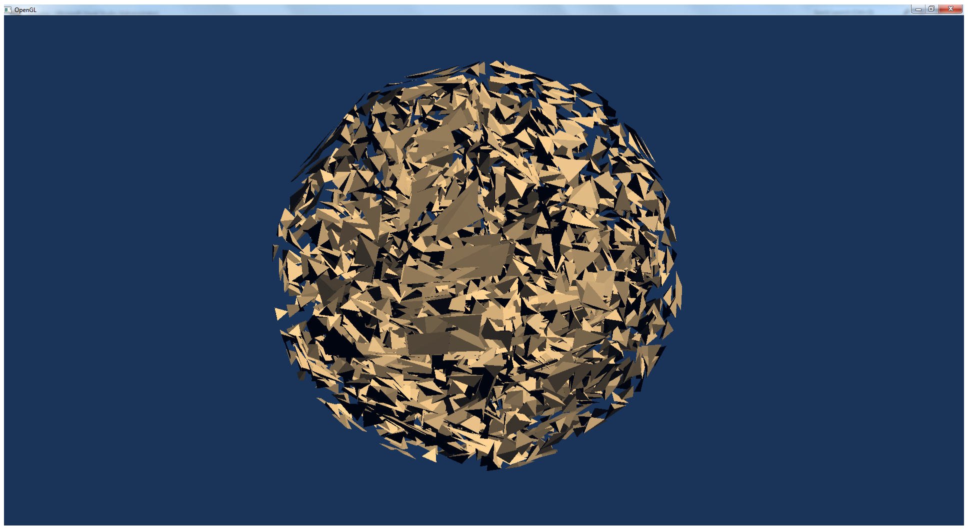


Figure 6: demo output of 1000 points within a sphere

References

[1] Daniel K. Blandford, Guy E. Blelloch, and Clemens Kadow. Engineering a compact parallel delaunay algorithm in 3d. In *Proceedings of the twenty-second annual symposium on Computational geometry*, SCG '06, pages 292–300, New York, NY, USA, 2006. ACM.

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

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