Parallel Fast Tetrahedral Mesh Generation (in the Wild)

Alex Havrilla, Ariel Uy

December 2020

1 Summary

We added parallelism to the triangle insertion section of the fast Tetrahedral Meshing Algorithm, using shared-memory parallelism on high-core CPU machines. We present an analysis of our decomposition methods and granularities on various processor counts.

2 Background

The meshing problem is an old one, having deep roots in geometry, computer graphics, and numerical simulation. Given a representation for a domain, how can we transform this into a well structured mesh which well approximates and discretizes the domain, leading to accurate and stable computation? The Fast Tet Meshing Algorithm (FTMA) presented in 2020 at SIGGRAPH gives major speedup to the classic Tet Meshing Algorithm: Given a surface in 3D space represented as a triangle soup, i.e. an arbitrary collection of triangles, how can we best mesh the interior volume with tetrahedra? At a high level, the approach can be broken into four parts:

1. Simplify the input triangle soup within an epsilon bound of the surface
2. Generate a background mesh and insert triangles iteratively
3. Improve the quality of the mesh using local operations
4. Filter mesh elements to remove outside surface

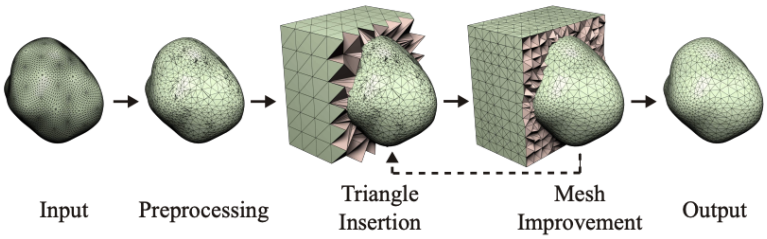


Figure 1: An overview of the algorithm [2]. In the third and fourth pictures, we see the background tetrahedral mesh.

2.1 Data Structures

The input to the algorithm is an STL file, which is used to represent triangular meshes. The output is a MSH file, which is used to represent tetrahedral meshes.

The main data structures used in this algorithm are:

- MeshVertex
  - Scalar3 positions
  - more...
- The triangular input mesh:
  - a vector of 3D vertices used in the triangles
  - a vector of triangles, which are each a Vector3 of indices into the above vector
- The tetrahedral background mesh:
  - a vector of 3D vertices used in the tetrahedra
  - a vector of tetrahedra, which are each a Vector4 of indices into the above vector of vertices
- Partitions
  - Varies depending on implementation, usually stored as additional parameters to MeshVertices

The input mesh is simplified, where certain edges and triangles are contracted if they are small enough. After this step, the input mesh is not changed.

The background mesh is subject to many operations. First, it is generated based on parameters from the input. Then, triangles from the input mesh are inserted into the background mesh by subdividing the tetrahedra which the triangles intersect with. Subdividing the tetrahedra is difficult to work with in parallel, since dividing one tetrahedron also requires dividing its neighbors in order to maintain the mesh properties. Some triangles will fail to be inserted on the first try, so the mesh quality is iteratively improved until all triangles are able to be inserted. Finally, the outside surface of the background mesh is removed, leaving only the tetrahedra on the interior.

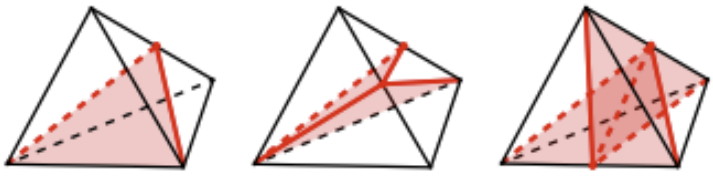


Figure 2: Here we can see a few possible subdivisions of a tetrahedron when inserting a triangle [2].

Inserting triangles into the mesh is the most expensive part of the computation, due to the number of iterations which must take place in order for all triangles to be successfully inserted.

2.2 Parallelism

Steps 1 and 4 already have naive forms of task level parallelism implemented. In both cases, independent components of a mesh are identified via a graph coloring procedure and then operated on in parallel.

We attempted to parallelize the iterative addition of triangles to the background mesh, as this is a bottleneck of the computation. As will be addressed below, this task is nontrivial as the addition of a triangle locally affects the structure of the mesh, hence leading to dependencies.

Adding a triangle to the background mesh changes the local tetrahedral connectivity. Furthermore, there is no clear locality, since the triangles in the triangle soup are not ordered in any way. In order to implement a good parallelization, we can map partitions of the polygon soup to neighborhoods of the tetrahedral mesh, parallelizing over each partition.

However, in any partition, there will be overlaps across the corresponding borders of the tetrahedral mesh neighborhoods. Thus, we can only parallelize over triangles which are fully contained in a neighborhood, and other triangles are inserted sequentially.

3 Approach

We started with the existing code base from [2], modifying files involving the sequential triangle insertion, mesh data structure, Delaunay meshing, and adding a new file implementing partitioning of the space.

3.1 Algorithm

Here we present our algorithm for triangle insertion at a high level. For more detail on the specifics of the implementation refer to [2].

1. Partition background mesh into partitions for parallelism
2. Localize vertices in soup to partition (in parallel)
3. Localize triangles to be inserted in partitions (in parallel)
4. Parallelize insertion over localized triangles
5. Sequentially insert triangles unable to be localized or insert in parallel

The code for sequential insertion does not deviate much from [2], only leveraging some additional optimizations afforded by computing localizations of vertices. Here we present code for parallel insertion:

```
parallel_inserting ← TRUE
for over partition components P do
  for over triangles T ∈ P do
    original_vert_length ← get_num_vertices(mesh)
    intersected_tets ← compute_intersecting_tets(T)
    new_tets ← snap_to_plane(intersected_tets)
```

```
    {ATOMIC
    new_points, new_tets ← subdivide_tets(new_tets)}
    if parallel_inserting && intersecting_tets_localized(intersected_tets) then
      {ATOMIC
      push_new_tets(points,new_tets,mesh,original_vert_length)}

      return TRUE
    else

      return FALSE
    end if
  end for
end for
```

Here we iterate over partition components in parallel and only commit an insertion if all intersecting tets are localized to the partition component. We say a tet is localized if all its vertices lie in the same component. This is also how we localize triangles. The main operation to the mesh at this point involves a subdivision of existing tetrahedra, creating new tets and points. It is useful to observe the subdivision operation preserves the locality of existing vertices and tets. Further it maintains the locality of the new tets and vertices, as they lie in the convex hull of the original vertices. Then as an optimization we memoize the locality of all vertices beforehand. Further we memoize the locality of all tets as they get computed on the fly (precomputing would be a waste of time as some tets in the background mesh will never intersect any triangles). The localization of point computation varies depending on the partitioning approach, and will be discussed in subsequent sections.

3.1.1 Uniform Partitioning

The first partitioning scheme we tried uniformly divides the bounding box of the background mesh into cubes, dividing along each dimension. This allows us to easily compute the cube that each vertex resides in. For each of the x, y, and z dimensions, we can determine which segment of the the vertex is in simply by dividing.

The disadvantage of uniform partitioning is that it is very unbalanced. The input mesh is a 2D surface of a 3D object, and it has more detail in some places than others, so the triangles are not distributed evenly throughout the mesh.

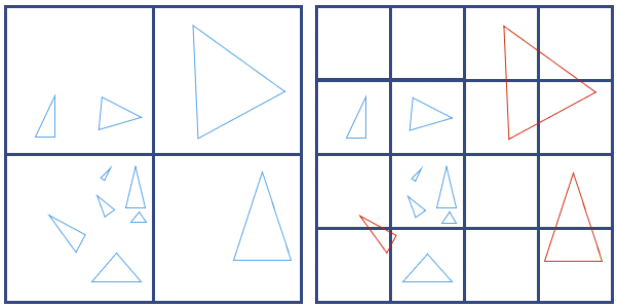


Figure 3: Here we can see an example of a uniform partition in 2D. At a coarser granularity, all triangles are partitioned properly, but the workload is unbalanced. At a finer granularity, the balance is somewhat better, but we also have triangles which overlap boxes.

3.1.2 Octrees

The second partitioning scheme we tried was motivated by wanting a better workload balance. Since we have a large number of unevenly distributed vertices in 3D space, we attempted to use octrees to partition the space. An octree allows us to recursively partition the space by subdividing only the octants which have "too many" vertices, aka the number of vertices is greater than some threshold.

The disadvantages of octrees are that it requires more precomputation and that it is more expensive to compute the cube that the each vertex is in. The octree must be precomputed in order to divide the space. Then, while running the algorithm, the mesh gains vertices as tetrahedra are subdivided. For each additional vertex, we must search the octree to determine its location.

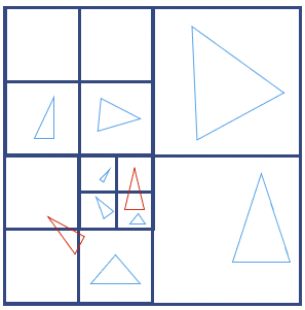


Figure 4: Here we can see the 2D analogue of octrees, a quadtree partitioning in 2D. This allows for a better workload balance between boxes, but it has the potential to create triangles which overlap boxes.

3.2 Hardware and Implementation

We used the 40-core Andrew Unix machines, because we used shared-memory parallelism and wanted to test our code on various thread counts up to 32. The starter code that we used is written in C++ and uses Intel Thread Building Blocks (TBB) for parallelism. We used TBB mainly for its parallel for construct, which dynamically schedules each for loop iteration.

3.2.1 Work Decomposition and Scheduling

Work decomposition is controlled by the partitioning scheme, ie. either a uniform or oct-tree approach. The ideal case is to have a uniform split of triangles among all partition components. We will discuss the effective of each scheme in our results.

Since we used TBB for parallelism, thread scheduling is done dynamically over the for loop. Thus each task is a partition

3.2.2 Synchronization and Locking

The main form of synchronization we use is mutual exclusion locking. Via our spatial decomposition we know no two threads should modify local mesh data, ie. mutate an intersecting set of tets and vertices at the same time. However for memory efficiency each tet is stored as a collection of four indices indexing into a vertex vector. Further when a triangle is successfully inserted, this often results in the creation of new tets and vertices which get appended to the mesh vectors. This leads to several complications. Most

immediately we do not want two threads trying to insert new points or tets at exactly the same time, as this could lead to one or both not being inserted. So we lock insertion of new tets or points into the global mesh.

We also we need to atomically lock computations involving creation of new tets and points to preserve the correct indexing ie. certain reads from the global datastructure. For example we do not want two threads interleaving insertion of points from two new sets. What is more locking these computations is not quite enough. Consider the following. Thread A creates new tets TA to commit with corresponding points PA, we could then have thread B commit its own new tets TB and vertices PB, interfering with the indexing of TA into the global vertex vector. This interferes since the points in PB will be appended globally after PA, whereas the indexing in TA did not account for extra length from PA. Locking the entire pipeline between computing new tets/points and updating the mesh is infeasible. Thus we need to compute an offset to add the indices of each new point in the new tets to preserve indexing. This is done using `original_vert_length` which we find works well.

3.2.3 Locality

The locality in this problem should be pretty clear. As a thread is inserting triangles in a component expect the tets being intersected, subdivided, and updated to be roughly the same working set since they are spatially localized. This leads to better cache performance and lower visible memory latency.

With this being said, the arithmetic intensity for the problem is very high, so exploiting locality (for higher memory performance) is less of a concern than in other more sensitive problems.

3.3 Iterations

We were motivated to parallelize this problem with the notion triangle is a local operation and thus a good spatial decomposition should work well. We began by implementing a uniform partitioning method, but soon found the data structures used to efficiently store the mesh presented global challenges which would require a much more thorough understanding of the codebase to properly address. We spent much time implementing this and trying to reduce the locked scope as much as possible while maintaining correct behavior. We also noticed the uniform scheme led to particularly poor performance and low speedup on poorly behaved meshes, and hence we implemented Octrees to compensate. After we spent time attempting to effectively parameter tune for different meshes and processor counts.

4 Results

We start with a general discussion of the difficulties we found in parallelizing this problem and then present results. We considered three inputs in our results. Animal Skull is a large mesh, consisting of 158,000 triangles, Yoda is a smaller mesh, consisting of 31,000 triangles, and Chess Horse has a very uneven distribution of triangles.

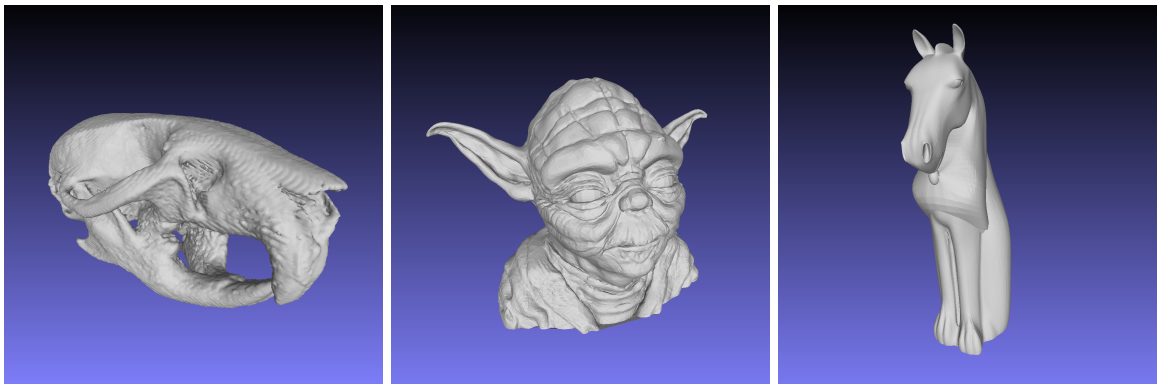


Figure 5: Animal Skull, Yoda, and Chess Horse input triangular meshes

4.1 Impediments to Parallelism

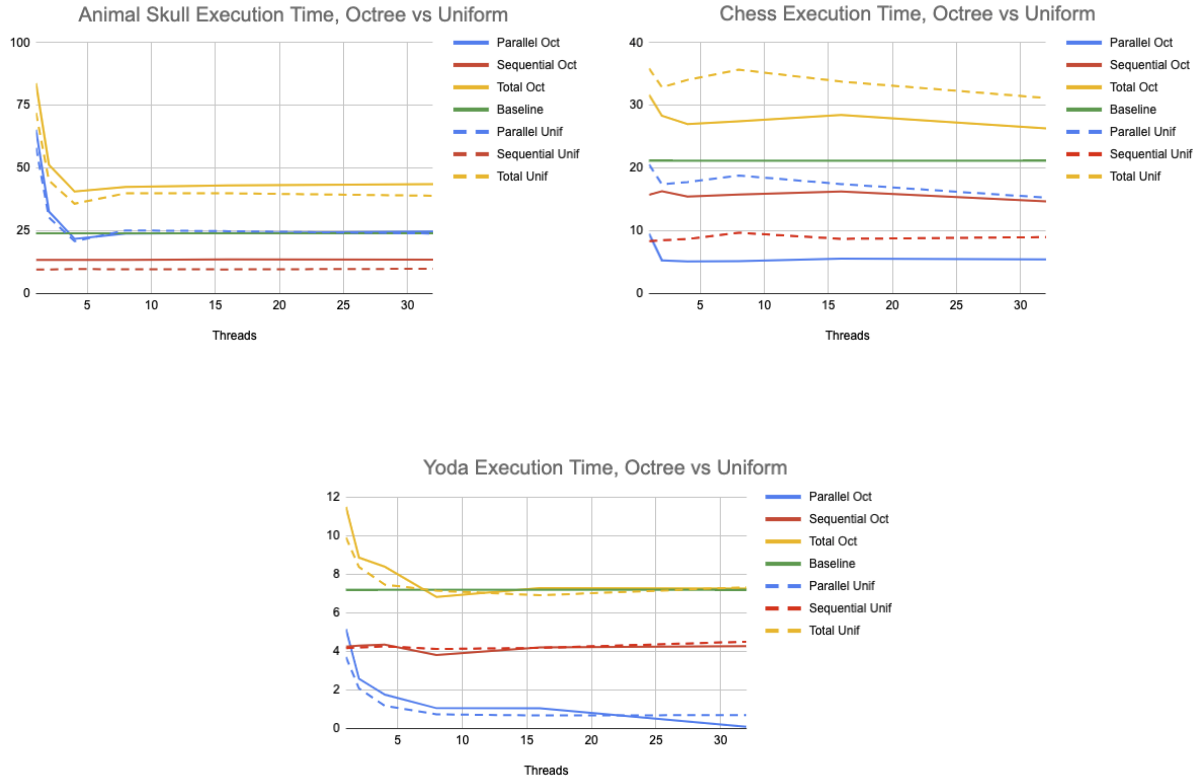
We had two main impediments to parallelism. First, the shared global data structures required locking in order to modify, which effectively sequentialized some otherwise parallel sections of our code. Second, we found a strong tradeoff between workload balance and the amount of sequential computation needed.

The background mesh is a global data structure that is shared between all threads. Each thread is attempting to insert triangles in parallel. Since the triangles are in non-overlapping regions, this computation is theoretically thread-safe. However, we run into issues due to the way that the mesh is stored, since inserting triangles modifies the length of the vertex array, which changes the indexing of tetrahedra. There were several regions of the triangle insertion code that we had to lock in order to make the code run safely. Over time, we decreased the size of the locked regions in order to allow for more parallel execution.

Finer spatial partitioning may result in better workload balance, but it decreases the number of triangles which can be inserted in parallel. When we have a finer spatial partition, i.e. increased number of cubes, more triangles reside on the edges between two or more cubes. Triangles on or near the edges cannot be inserted in parallel, and are left for the end of the computation to insert sequentially. However, a coarser partition may not divide the space into enough chunks to effectively balance the workload. The scheduling is done dynamically by TBB, so we want to split the work into enough chunks for each thread to keep busy. This tradeoff is difficult to work with and often resulted in zero or negative speedup on higher processor counts.

And of course in addition to these problem specific issues we have the usual problems involving larger overhead via scheduling as we increase thread count(though this was not super visible due to aforementioned factors).

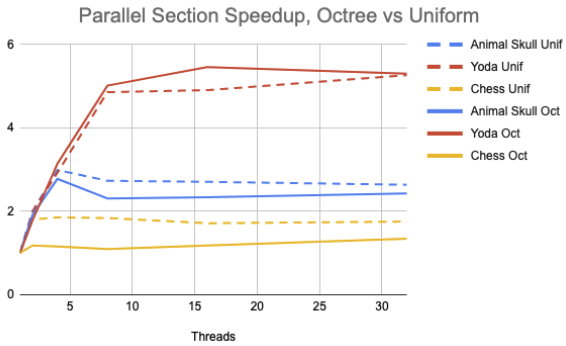
4.2 Execution Time and Speedup



The above graphs time the total execution time in seconds of the total time it takes for the insert triangles call to complete on first pass. Ie. no triangles have yet attempted to be inserted beforehand. The yellow curves give the total time our implementation gives. Green the time taken by the sequential insertion baseline, which has some trivial but largely negligible parallelism already implemented. Further we also breakdown the times for parallel and sequential insertion seconds. Note these will not sum to the total time since there are subsequent operations done in the insert triangles call(which we did not attempt to parallelize or optimize).

First some high level comments about the overall performance. We found the baseline provided by [2] is very competitive, and in most cases found it difficult to beat for a number of factors. However we do narrowly in some cases. Most notably the extra computation required to localize all relevant geometry in an insertion computation is highly nontrivial, leading to more necessary work. Compare 1 thread execution time on total to the baseline for a sense of this. Further our insertion scheme first attempts to insert triangles in parallel. A parallel insertion is rejected if the computation cannot be localized, ie. all relevant tets are in the same partition, and reattempts the same triangle sequentially. This ensures comparable mesh quality but in same cases doubles the amount of work otherwise done. However we think much of can be optimized for a more competitive time, which we will discuss in our future work section.

We now discuss speedup achieved in the parallel insertion section.



We focus on the parallel insertion time instead of total insertion time to illustrate effective workload balance can be achieved in some cases. Ensuring good work decomposition even with dynamic for loop scheduling is dependent on both the partitioning technique used and quality of the input triangle soup. For example, look at the graphs for Chess vs. Yoda. These are similarly sized meshes, both of around 15000 vertices and 30000 triangles. However the majority of the triangles in Chess are tiny and concentrated in areas of high curvature. These triangles are difficult to separate, and as a result we consistently saw a component with around 10000 of the 30000 triangles regardless of the partition strategy. This acts as a bottleneck to the parallel computation and is why we do not see much speedup for Chess. The Yoda mesh is much better behaved, allowing for a easier partitioning much better approaching a equal split of triangles across a large number of components. This then allows for much better parallel speedup. Animal Skull is a much larger mesh, of around 150000 triangles, and seems to split the difference.

In most cases we see a stop in speedup between 4-8 threads. We attribute this mostly to a relatively poor workload balance, which we discuss below. Further we usually see a slight drop-off in performance with 16-32 threads, likely due to higher overhead.

4.2.1 Uniform Partitioning

We found the uniform partitioning method performs remarkably well for simplicity. In several cases, in the particular the well behaved ones, it outperforms the Octree in execution time and speedup. This is likely due to less required localization computation. In practice we selected 10 blocks in each dimension, leading to 1000 total partitions, to present the above results.

For example, on Animal Skull we create 1001 total components, 346 of which are nonempty, with an average count of 457 triangles in a nonempty component and a maximum of 2835 in one component. The Chess piece is more pathological, with 1001 components created and 261 nonempty, but with an average occupancy of 97.24 and a maximum occupancy of 11901.

4.2.2 Octree

In contrast the Octree shines on the harder mehses, such as Chess. Whereas the uniform partitoning scheme has a component with around 12000 triangles, Octree gets us down to 10000. Further this was achieved using only a depth of 9 with a threshold 2000(these were the default parameters for the above data). Increasing the depth even more should result in an even better triangle split.

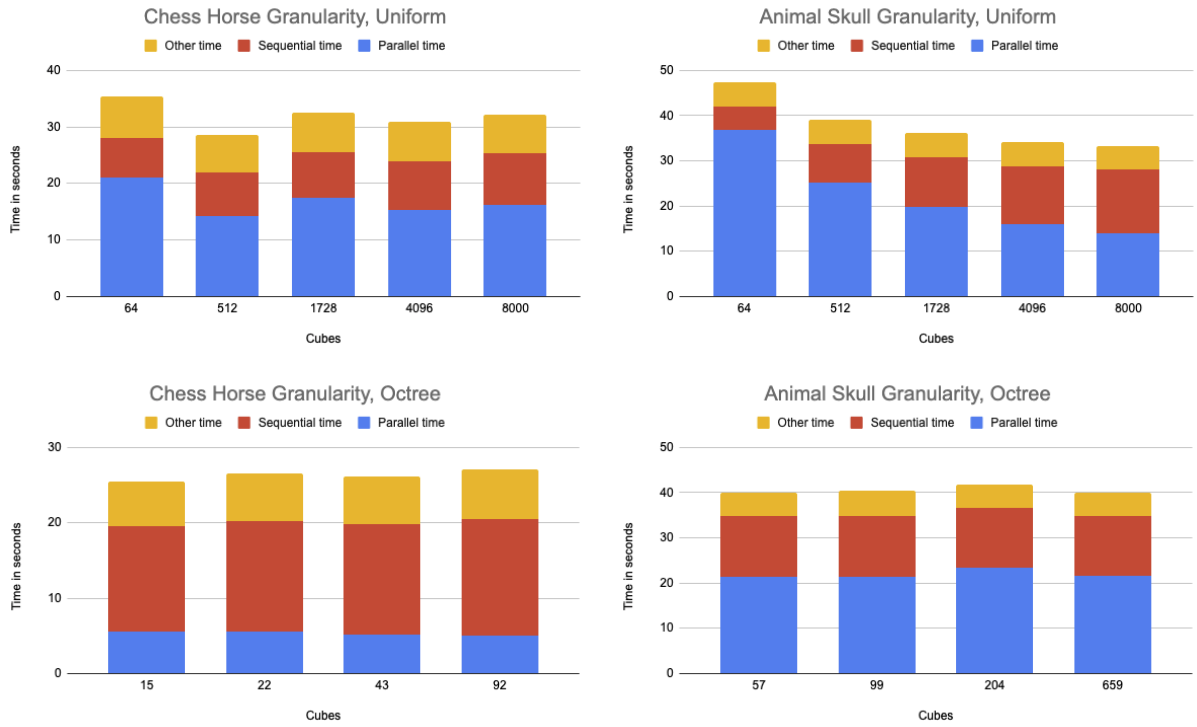
To be concrete, on Animal Skull we create 212 total components, 31 of which are nonempty, with an average count of 4600.27 triangles in a nonempty component and a maximum of 28848 in one component. The Chess piece is still pathological but less so, with 93 components created and 20 nonempty, but with an average occupancy of 1142 and a maximum occupancy of 10044.

It is also interesting to note the triangles rejected while inserted in parallel are largely(though not entirely) invariant under thread count, meaning the sequential execution time does not significantly change as we increase thread count. Note this only true for fixed granularity parameters.

4.3 Granularity

It is tempting at first glance to think in general sufficiently refining the partition, via the uniform approach or Octree, would lead to a better workload decomposition and thus a better overall time. However as previously mentioned this is not the case. Once a component gets too small it becomes harder and harder to localize an insertion computation, leading to a higher parallel rejection rate. This holds true in both partitioning algorithms.

The following graphs show the effect of granularity on the performance on both the uniform and octree decompositions. The number of threads was kept constant at 8. Animal Skull has fairly uniformly distributed triangles, while Chess Horse has large amounts of triangles concentrated in a small area.



In the uniform case, Animal Skull sees increasing overall performance as we increase the granularity. This is due to the fact that it has a large amount of triangles (over 100,000) and they are mostly uniformly distributed. However, we can see that while the parallel execution time decreases, the sequential execution

time increases. When we split the mesh into higher amounts of cubes, more triangles are forced to be computed sequentially. Looking at the octree case, we can see that the octree implementation on 57 cubes far outperforms the uniform implementation on 64 cubes, since the octree creates a better workload balance. However, at higher cube counts, the octree and uniform versions perform similarly, since uniform version is granular enough to create a good workload balance.

Chess Horse has an extremely uneven distribution of triangles, so neither the uniform breakdown nor the octree were able to break it down far enough to properly parallelize it. Thus, we didn't see much change when increasing granularity. Our octree implementation was unable to support the necessary granularity, but future work could explore the optimal solution in this case.

4.3.1 Tuning

We found tuning the parameters in either approach to be difficult, especially given the dependency on the mesh. The ideal decomposition yields a small set of components which is some constant multiple of the number of processors, with a uniform split of vertices. As we have said, achieving this is hard given the pathological nature of some meshes such as the chess piece. Further even given such a partitioning meeting these specifications, we must impose the additional constraint preventing components from being too small and leading to higher localized computation rejection rate.

It is our empirical opinion a uniform mesh of size 10*10*10 generally works well for most reasonable inputs on an arbitrary processor count. Further we found a Octrees with depth 9 and threshold 1000 perform reasonably well, as this encourages splitting. One key improvement to our algorithm is an implementation of Octrees supporting depths greater than 9. A much lower threshold seemed to lead to higher localized rejection rates.

4.4 Mesh Quality

Our algorithm changes the insertion order, eliminates some mesh simplification calls, and thus theoretically changes the resulting mesh of the input when compared to the Fast Tet Mesh algorithm. However in practice the difference is quantitatively and qualitatively negligible. Therefore for a thorough discussion of the quality of the resulting meshes we refer the reader to section 4 of [2]. For the rest of this section we present some pretty pictures.

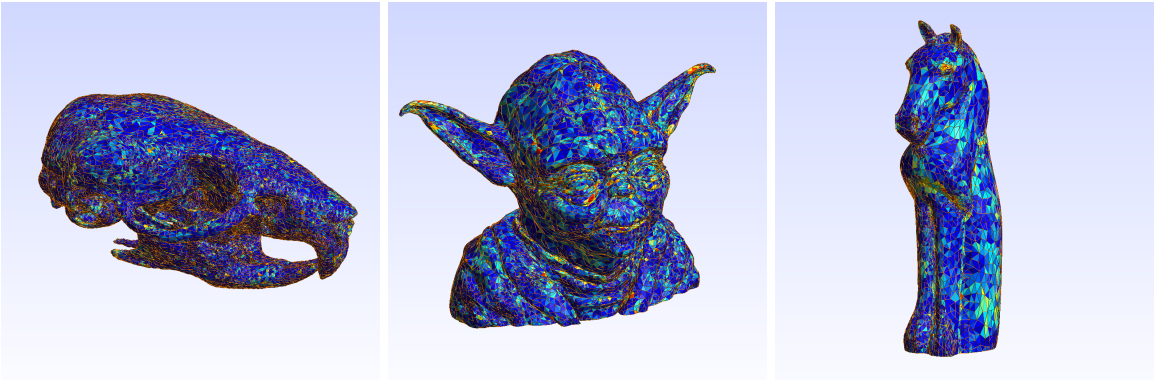


Figure 6: Animal Skull, Yoda, and Chess Horse output tetrahedral meshes, uniform partitioning on 16 threads.

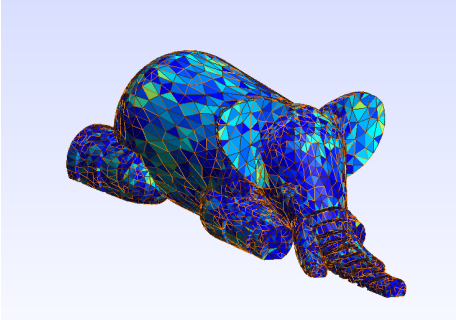


Figure 7: Mr. Trunks, output tetrahedral meshes, uniform partitioning on 16 threads.

5 Future Work

It turns out this is a much harder problem than we first anticipated and there is much more work to be done in order to capitalize on potential optimizations/parallelism, particularly if we want to be competitive with the Fast Tet Meshing algorithm. The researchers at NYU did a very good job optimizing.

However there are a number of improvements we can make which would make us more competitive. With a better understanding of the relevant codebase (which is hundreds of thousands of lines and thus infeasible to learn in total) we believe we can implement finer locking allowing for more parallelism. Further there are more optimizations we can make to do better triangle insertion in parallel and reduce the amount of necessary computation to localize tets. Also we expect implementing Octrees with depth greater than 9 would lead to much improved partitioning on pathological meshes, giving a better workload balance. Finally general code review would likely be useful, to ensure trivial optimizations such passing data structures by reference instead of value and so forth (we made sure to do this everywhere but are still suspicious of silly inefficiencies somewhere).

References

[1] Geuzaine. C, Remacle. J. **GMSH**. http://gmsh.info/

[2] Hu. Y, Schneider. T, Wang. B, Zorin. D, Panozzo. D. *Fast Tetrahedral Meshing in the Wild*. https://cs.nyu.edu/ yixinhu/ftetwild.pdf

[3] Hu. Y. Repository for *Fast Tetrahedral Meshing in the Wild*. https://github.com/wildmeshing/fTetWild

6 Work Breakdown

We had a 50-50 work breakdown:

- Getting existing starter code to compile and run on Gates machines - Ariel
- Implementation of uniform partitioning - Alex
- Implementation of octree - Both
- Implementation of synchronization and locking - Both
- Granularity tuning - Both
- Data collection and image generation for speedup section - Alex
- Data collection and image generation for granularity section - Ariel
- Analysis and report writing - Both

We faced significant challenges working with the starter code, as it is research code and not well documented. We also spent lots of time getting it to compile and run on the Gates machines, as we had to install new versions of CMake, GCC, and other dependencies.