Parallelize Mesh Simplification Algorithm with Pthread and OpenMP

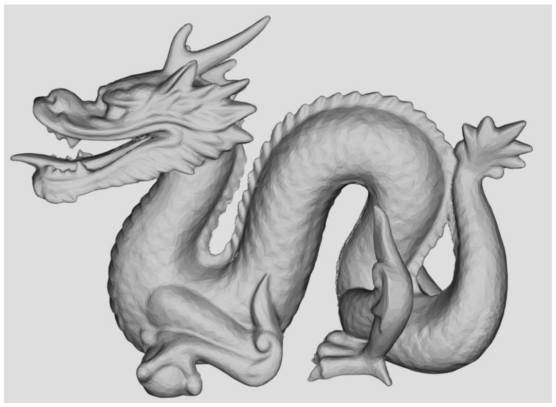
Bole Chen (bolec@andrew.cmu.edu)
Haixin Liu (haixinl@andrew.cmu.edu)

Summary

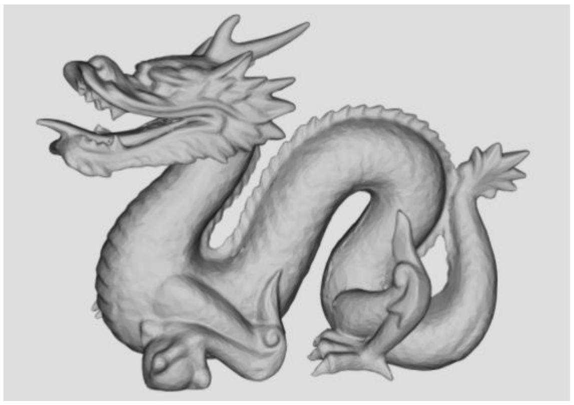
Surface mesh simplification is the process of reducing the number of faces used in a surface mesh while keeping the overall shape, volume and boundaries preserved as much as possible. There are lots of mesh simplification algorithms, and all of them are iterative, greedy algorithms that might cost really long time. One big problem is that all these algorithms have really strong dependency, which makes it hard to parallelize them. For this project, we proposed our new lazy-update implementation of mesh simplification algorithm. This implementation gives us some spaces for parallelism. We explored different methods to parallelize this our serial implementation with pThread and OpenMP. We run experiments on a 8-core Linux machine from AWS and found that our final solution can achieve nearly linear speedup without losing too much qualities.

Background

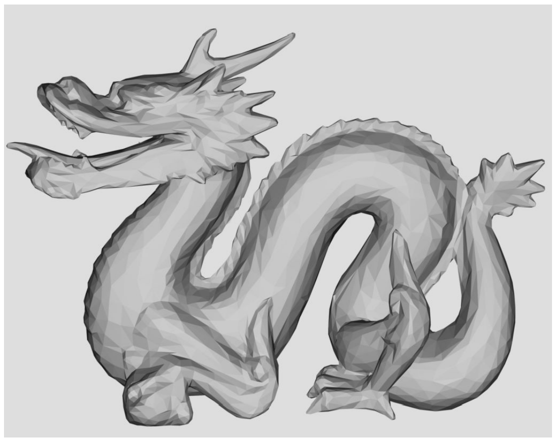
The problem of approximating a given input mesh with a less complex but geometrically faithful representation is well-established in computer graphics. Given the visual complexity required to create realistic-looking scenes, simplification efforts can be essential to efficient rendering. And different simplification ratios are need for different precision requirements. The figures below show simplifying Stanford Dragon from 100% to 1%.



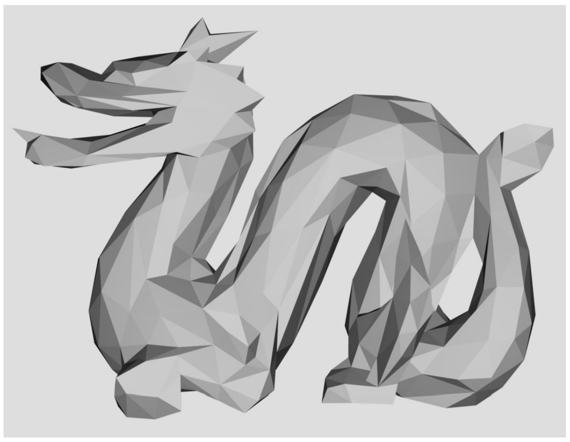
100%



50%



10%

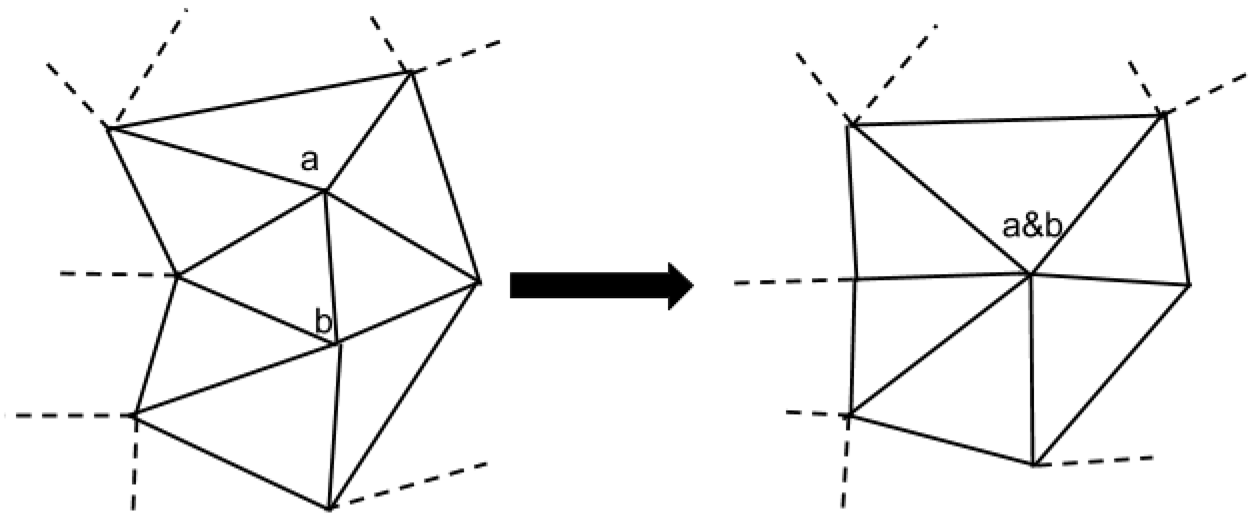


1%

In this work, we only handle the 3D OBJ files. A model object consists of thousands to millions of triangles. Each triangle has its direction for illumination and it shares 3 vertexes in 3D space with some other triangles. For the input file with this format, the first line gives the number of vertex and triangle. Then follows by the coordinate of each vertex and the vertex ids for each triangle. The output files of mesh simplification algorithm should follow the same format.

Serial Version Algorithm Introduction

The basic idea of mesh simplification algorithm is to maintain the error cost of each edge from all triangles in a model object. Each time an edge with the lowest error cost will be selected and collapsed. That two vertexes will be merged into one single vertex. Then firstly, the topological structure of nearby area need to be updated, and also the error costs of all related edge need to be re-calculated. The following figure shows how collapsing a single edge would influence nearby topological structure and other edges' error costs.



Although the basic idea of mesh simplification algorithm is simple and straightforward, the implementation is complex. On the one hand, we have to maintain a lot of data structure such as vertex, edge, triangle, and almost all of these data have relationship and dependency, which means even the change of one single vertex will lead to the update of a lot of other data. On the other hand, because the algorithm selects an edge with the lowest error cost to collapse for each iteration, always minimum heap is used to maintain the error costs of all edges. But the problem is that for each iteration, the top edge is popped out and at the same time, some other edges also need to be updated. As a result, the structure of the minimum heap changes frequently. The following table shows the pseudo-code of common mesh simplification algorithm.

Pseudo Code

```
1.  Read Vertex Data and Triangle Data, Initialize a Minimum Heap Q
2.  Foreach Edge: Calculate Error Cost, Push in Q
3.  While not Achieve Simplification Ratio Requirement:
4:      Pop Top Edge from Q
5:      Collapse the Edge
6:      Update Topological Structure in Nearby Area
7:      Calculate and Update Error Cost for Related Edges in Q
8:  Generate Output OBJ File
```

Potential Parallelisms Inside

Because in the algorithm there is really strong dependency inside each iteration and between different iterations, we believe we have mainly two directions to parallelize the mesh simplification algorithm. Firstly, we can try to find the independent part inside each iteration. These independent tasks can be done concurrently. And we can benchmark the fraction of the algorithm that is parallelizable in the overall algorithm. According to Amdahl’s Law, we can know the upper-bound of speedup we can achieve. Secondly, the basic idea is that actually we can collapse edges from different parts of an object as long as they don’t influence each other. It’s obvious that the collapse of edge in the left side doesn’t have any impact on the topological structure as well as error cost of right side. If this approach works, the speedup we can gain just based on how much resources we have to run the program.

Challenges for Parallelisms

The challenge for approach one is to find the parallelizable part in original algorithm. And our concern is that the fraction of the parallelizable part can be not enough to gain high speedup. And also the cost for each iteration is limited, hence the synchronization can be a huge overhead compared to computation time.

The challenge for approach two is how to partition the mesh and join together after processing. Or in other words, how do resolve the border problem. One of the papers we studied uses greedy BFS approach to produce even partitions. It uses MPI to parallel the algorithm, and manages the border problem with communications. However, the details of this algorithm is not clearly described in this paper. We need to find out how to partition input mesh and rejoin the partitions by ourselves. Also because the number of vertex that along the border margin could be huge, synchronizations would be needed frequently during the algorithm, which would prevent us from achieving high speedup.

Approach

Lazy-update Serial Implementation of Mesh Simplification

According our study, we found that some original mesh simplification algorithms are really hard to parallelize with our partition approach because of the border problem. The reason is that for each collapse operation, besides one edge needs to be popped out, a lot of other related edges inside the minimum heap need to be updated. Also a lot of meta datas need to be updated during one single collapse. All of these factors make the algorithm has really strong dependency. To make our way to parallelism easier, we proposed a lazy update implementation.

Instead of changing the value of edge and vertex, we just give an order id to each vertex and edge. If a vertex's value change, we just set that vertex as stale, and insert a new one. Also for a edge in the minimum heap, if the order id of that two vertexes not equal to the vertexes' lastest id, the edge should be seemed as invalid, hence we just pop that edge out without collapsing it. We call it lazy update because rather than change the value of edge in the minimum heap, we just set that element as invalid, and push in a new one. In our new implementation, the operations to the minimum heap reduce a lot, and therefore it's easier for us to parallelize it. The following table shows the pseudo-code of our lazy update mesh simplification algorithm.

Pseudo Code

```
1.    Read Vertex Data and Triangle Data, Initialize a Minimum Heap Q
2.    Foreach Edge: Calculate Error Cost, Push in Q
3.    While not Achieve Simplification Ratio Requirement:
4:        Pop Top Edge from Q
5:        If Edge is Invalid: Continue
6:        Collapse the Edge
7:        Update Topological Structure in Nearby Area
8:        Calculate and Update Error Cost for Related Edges in Q
9:        Push New Edges into Q & Set Old Edges as Invalid
10:   Generate Output OBJ File
```

Preliminary results

In this section, we show our benchmark on our new serial lazy update algorithm. We show the running time, the percentage of the potential parallel parts in the serial algorithm and how many independent tasks we can generate to parallelize for the algorithm for models of different size and with different simplification ratio. We run all these benchmarks in a 4-core Mac Pro machine with GCC compiler.

Stanford Bunny with 35,292 vertices and 70,580 triangles

Ratio	Running Time	Percentage of Parallelism	Concurrent tasks number
0.5	3.63s	62.81%	7.23
0.1	7.80s	53.57%	7.13
0.01	8.96s	53.12%	7.11

Stanford Dragon with 54,855 vertices and 109,227 triangles

Ratio	Running Time	Percentage of Parallelism	Concurrent tasks number
0.5	4.81s	65.25%	6.85
0.1	10.81s	53.82%	6.81
0.01	12.49s	52.66%	6.73

Stanford Lucy with 1,002,540 vertices and 2,005,076 triangles

Ratio	Running Time	Percentage of Parallelism	Concurrent tasks number
0.2	125.32s	48.46%	7.06
0.1	315.74s	45.21%	7.05
0.01	375.13s	43.26%	7.03

We can see from the experiments that when the size of model is large, it takes longer time for the mesh simplification algorithm to complete. That's why we need parallelism to speedup it. Another thing to notice is that the potential parallel part in this algorithm is just about 40% and the concurrent task number is no more than 8. We explored our first try based on these results.

First Try to Parallelism

Our first try to parallelize the algorithm is to identify the independent loops within the serial algorithm. These loops can be distributed to many workers in a shared address space fashion. Common implementations include OpenMp and Pthread library. We implemented this parallel algorithm in both of these two ways. The first step is to benchmark the serial algorithm and find out the time-consuming independent loops. Unfortunately, this mesh simplification algorithm is highly serial, since each modification is based on the previous one. Finally we decided to parallelize the loop within one modification, which updates the error costs of all neighbor edges of a merged vertex.

We firstly implemented the parallel program with Pthreads and then with OpenMP, and tested them on Macbook Pro machine. However, we observed a slow down of the running time instead of speedup. We then tested them on 8-core AWS Linux machine, and got the similar results. We found out 3 reasons why this approach does not work:

1. In each iteration of this loop, a shared queue is maintained. The concurrent update of this queue need to be protected by lock (or atomic operations).
2. The task for each worker thread is not computation intensive. It only involves a few calculations, and then updates the shared data structure. You can see some related pre-result from last section in **Preliminary results**.
3. The overhead of creating/destroying threads.

To overcome these issues, we tried many ways to optimize it. We kept buffer to accumulate local updates, and tried to use as few locks as possible by batch updating. We maintained a pool of threads to avoid creating threads on the fly. However, the improvement is not satisfying.

Partition for Parallelism

After our first (but not successful) attempt in parallelizing independent loops within serial code, we tried several other approaches to do mesh simplification in parallel. A straightforward idea is to partition the original mesh into several blocks, assign each block to a worker thread, and then merge the blocks into the output mesh. However, there are several difficulties in implementing this parallel algorithm. The first challenge is how to partition the input mesh into blocks. These blocks should be in roughly same size, so that the workload is balanced among worker threads. Another challenge is how to deal with the block border vertices and edges. If these elements on or near border change significantly after processing, partitioned blocks will be hard or even impossible to form a new complete mesh.

We began our exploration with simple assumptions and implementations, and then refined the algorithm for better performance (speed & quality) step by step.

Margin Unmodified

Easy Partition

We start with the simple way to perform the partition. For the input mesh, we first calculate the center point, which has the average value of maximum and minimum x, y and z value of vertices. Based the this center point, the input mesh is partitioned into 8 (2x2x2) blocks. We start exactly 8 worker threads to process these 8 blocks independently. For border vertices and edges (which we call margin), we just keep them unmodified in the algorithm. In this way, there is no synchronization among worker threads needed. Also, no explicit merge process after dealing with each block is needed, since the margin is left unchanged.

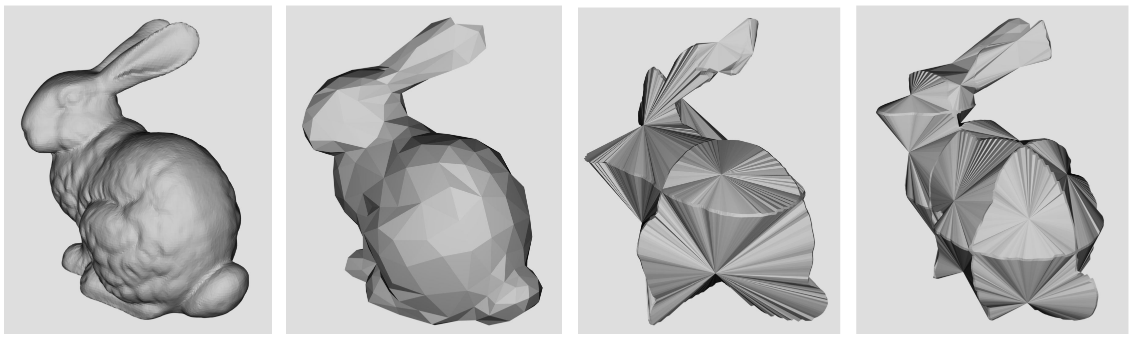
Partition with Task Queue

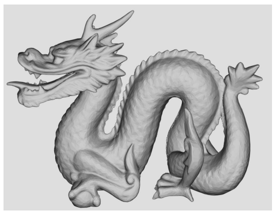
One of the major drawback of the easy partition algorithm is load balance problem. It is very likely that the 8 blocks contain different number of triangles, so that the workload of each worker thread can vary significantly. The slowest thread will determine the performance of whole program.

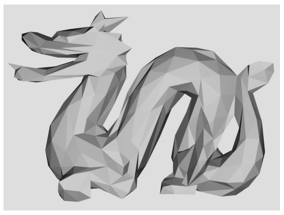
In order to improve the load balance, we partition the input mesh with finer granularity. With similar method, we partition the input mesh evenly into 2x2x2, 3x3x3 and 4x4x4 spatial blocks. We set up a task queue which contains workload of all these blocks at the beginning. All worker threads will pull work (block of mesh) from this queue and then process that block. The algorithm will finish once the task queue is empty. We observed much better load balance among worker threads with this approach. All threads take nearly same amount of time to do tasks. As a result, the overall speedup of this algorithm is better than the former naive approach.

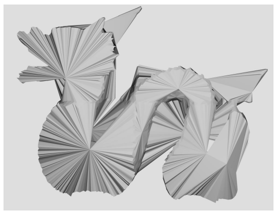
Margin Protected

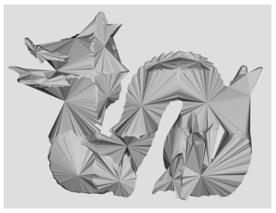
For the two approaches in the last section, we just make the margins between different blocks as unchangeable. This choice comes with two-side effects. On one hand, there is no synchronization on the border so that the contention is not a problem in this case. On the other hand, the quality of output mesh is not satisfying with low compression ratio. In this case, there are few vertices left in each block, and the border lines can be clearly seen. The quality is much worse compared with the output generated by serial algorithm. The figures below show the problem for the two methods before.




Original mesh


0.02 with serial


0.02 with easy partition


0.02 with task queue (3x3x3)

You can see that simplification quality for the previous two methods is poor, the density of the margin area is really high while the density inside each partition is low. Therefore we must cope with the border problem for acceptable simplification quality. We then tried two different ways to resolve the border problem.

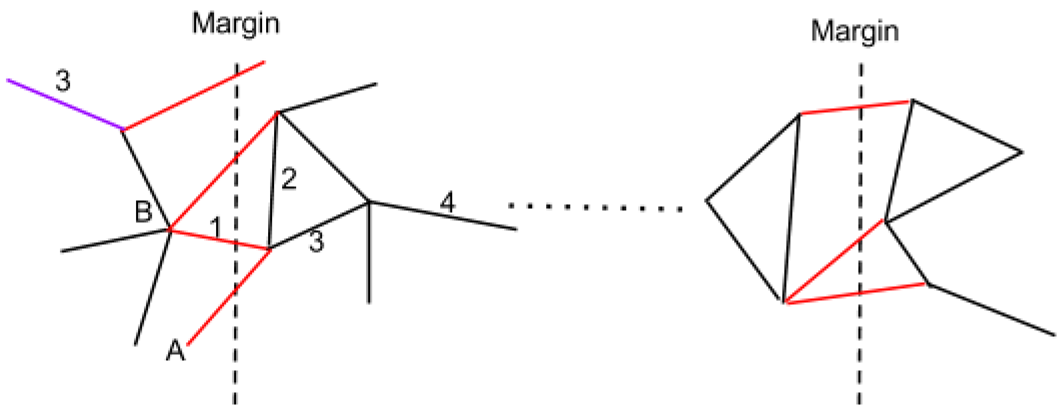
<u>Rough-Grained Lock</u>

The border problem is that when a thread is collapsing an edge near the margin area in a partition, it might change the topological structure as well as some values in another partition. If there is another thread being copes with that partition at the same time, there must be race condition problem. So when a thread is collapsing the edge in the margin area, all the other edges in margin area must be protected and cannot be coped with. Because the collapse of the edge in margin area doesn't influence the data inside each partition, we only need to protect the vertexes and edges in the margin area.

The basic idea of this implementation is that we have a global Margin-Lock. When a thread tries to collapse an edge inside a partition, it just does whatever it wants. But when a thread tries to collapse an edge in the margin area, it must acquire the Margin-Lock first. It must release the lock after all the data related to other partitions have been updated.

<u>Fine-Grained Lock</u>

The problem for the rough-grained lock implementation is that only one thread can cope with the edge in margin area, as a result the synchronization overhead is really high when the fraction of margin vertexes is high. But actually it's unnecessary to set a single Margin-Lock. We found there are 4 types of edge for an object. They are marked with numbers in the following figure.



We still set a constain that edge of type 1 cannot be modified. When edge of type 4 collapses, we just collapse it without caring about anything. Even when the edge of 2 or 3 collapse, some other edges in margin area can still collapse. For example, when the edge marked with 1 collapses, the edge in the margin area of right side can still work normally. Also when the edge marked with 1 collapses, even the purple edge in the same margin area can collapse. The reason is that the collapse of that edge only influence vertex A and B.

Based on all the findings, we proposed a fine-grained lock implementation. In this implementation, we maintain a vector for all the margin vertexes. And we use a single lock a protect that vector. Whenever a vertex in margin tries to merge with other vertex, it must access to the shared vector and find if the margin vertexes it would influence have already been in the shared vector. If not, it puts itself in the shared vector, otherwise it must wait until it can do the merge operation.

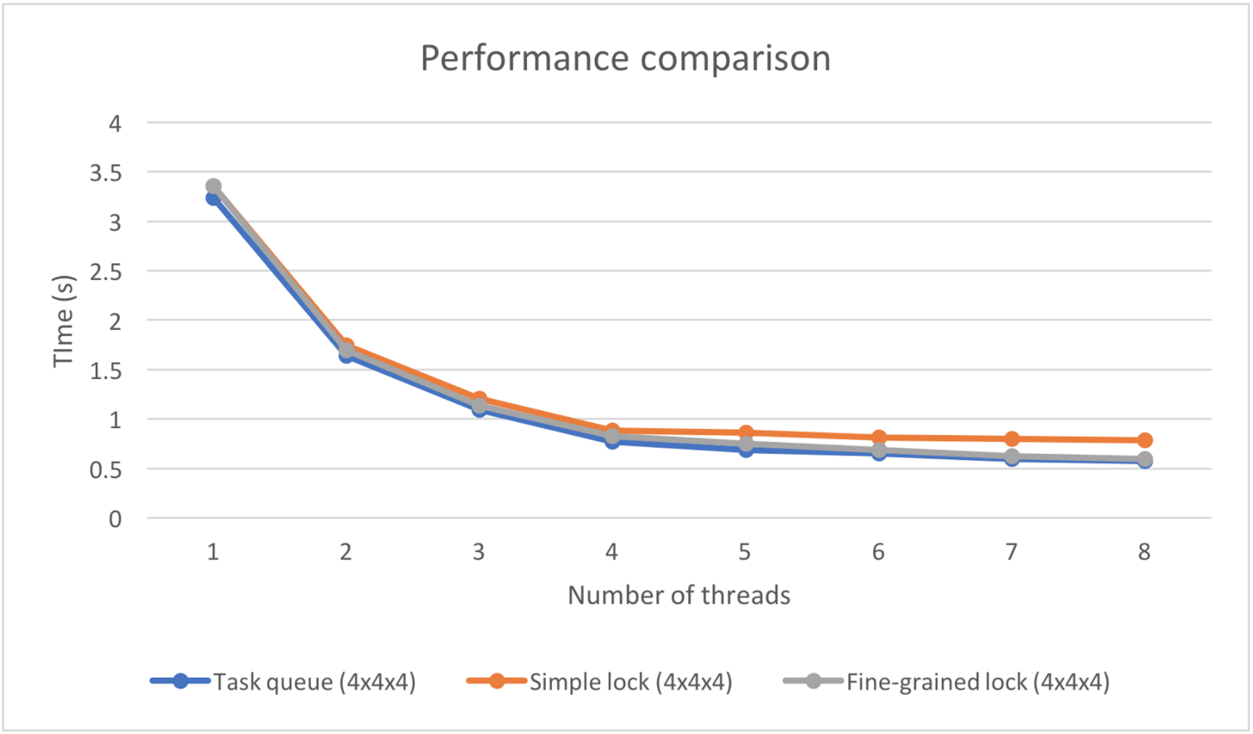
Results

We implemented our 4 parallel mesh simplification algorithms with both pthread library and OpenMP. Then we ran experiments to compare and evaluate their performance.

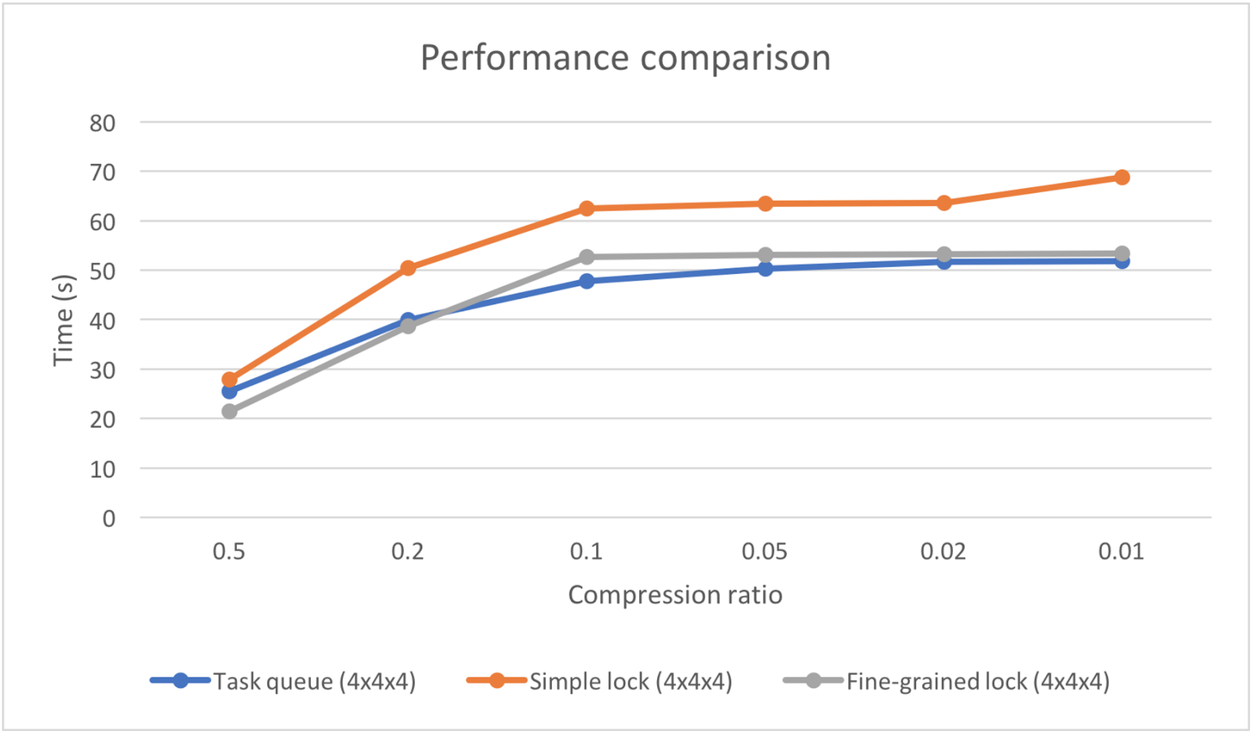
Experiments Setup

We ran our experiments on both 4-core MacBook Pro machine and AWS 8-core Linux machine. For pthread implementation, we compile the code with standard g++ compiler. For OpenMP implementation, we compile the code with Intel C++ Compiler (icpc).

PThread Speedup



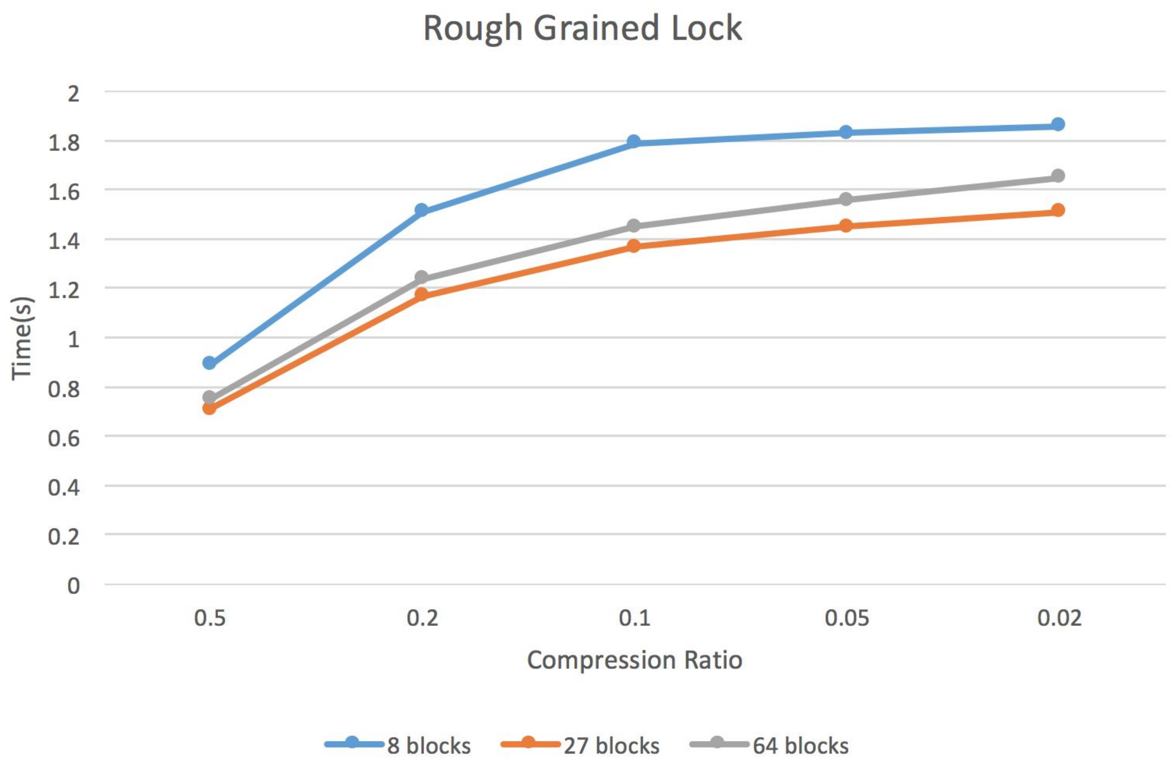
First we compare the performance (speedup) of our parallel algorithms with thread count ranging from 1 to 8. The performance with 1 thread is the serial result and can be used as baseline. In this experiment, we use Stanford dragon as input mesh file, with 0.1 compression ratio and 4x4x4 partition. All of our parallel algorithms can achieve nearly linear speedup over serial version. Task queue based algorithm has best performance since it does not synchronize margins. Simple (coarse grained) lock approach has worst performance. Fine-grained lock approach has performance similar to task queue based approach, but much better output quality. For input mesh with small size (e.g. 5 MB), this pattern may change. 4 thread performance may beat 8 thread performance, since lock contention cost becomes more obvious compared with computation cost.

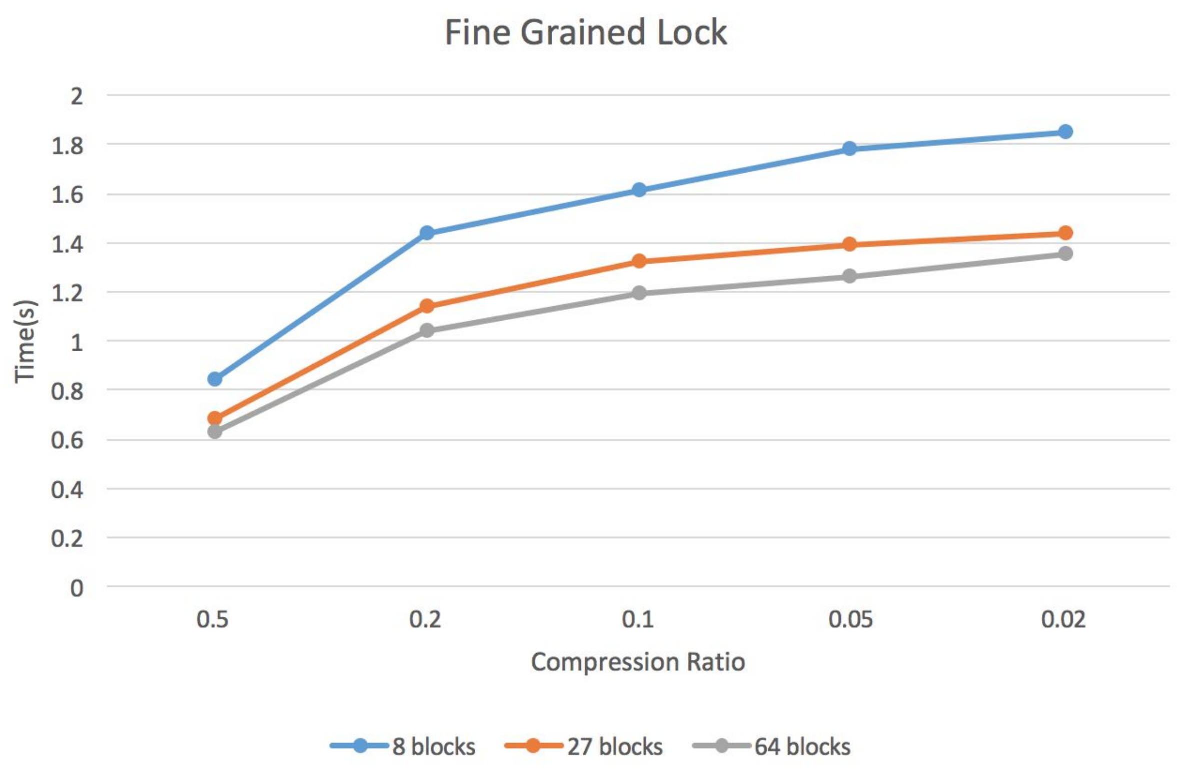


Higher compression ratio results in more time consumed, since more computation is needed. This experiment is run on Stanford Lucy mesh file. Fine-grained lock approach has significant improvement over simple lock approach.

Rough-Grained Lock vs. Fine-Grained Lock

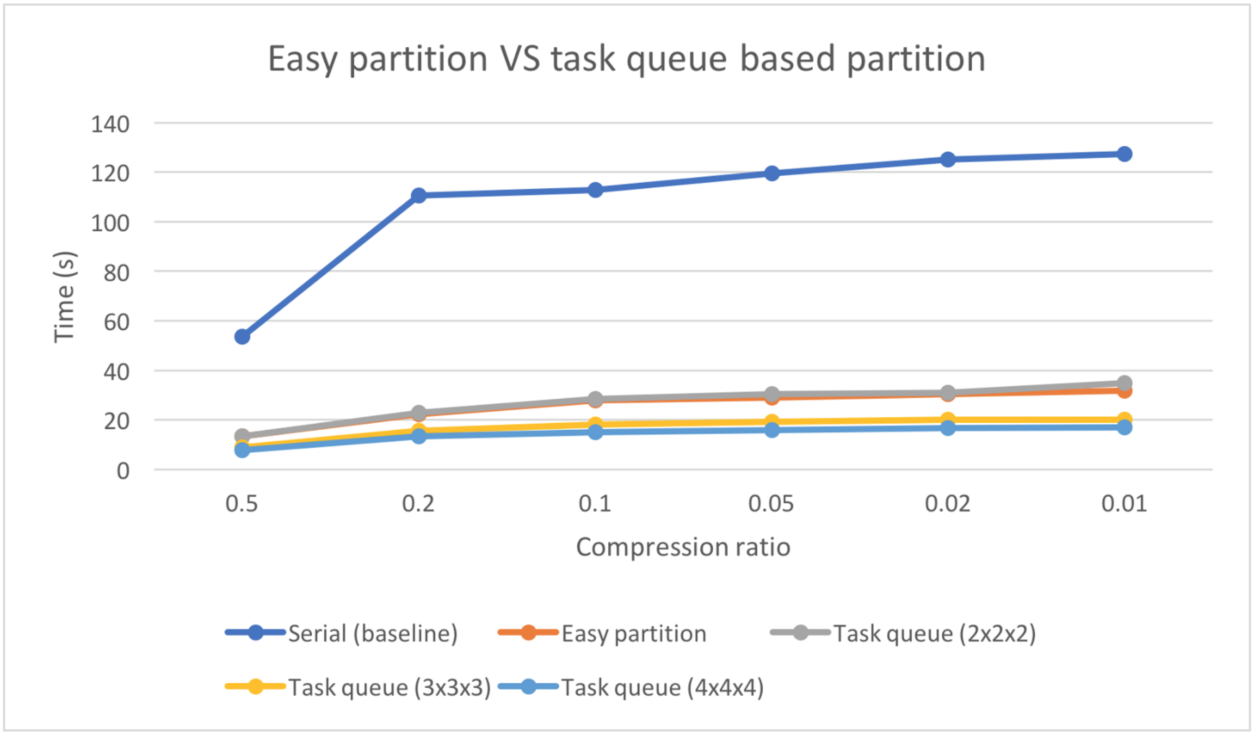
We found something abnormal for small size object. We run the Stanford Bunny model for Rough Grained Lock implementation and Fine Grained Lock implementation with different block numbers. As we discussed above, the more blocks we have, the workloads are more balanced for all the threads, and hence the faster should the program run. But the result shows that, for rough grained lock implementation, it ran faster with 27 blocks rather than 64 blocks. And there is no such phenomenon for fine grained lock implementation. We think the reason is that there is a trade-off for rough grained lock implementation. Although the more blocks bring good load balance, it also introduce more synchronizations. But our fine grained lock implementation doesn't have this problem, which ensure us the fine grained lock help reduce the synchronizations a lot.

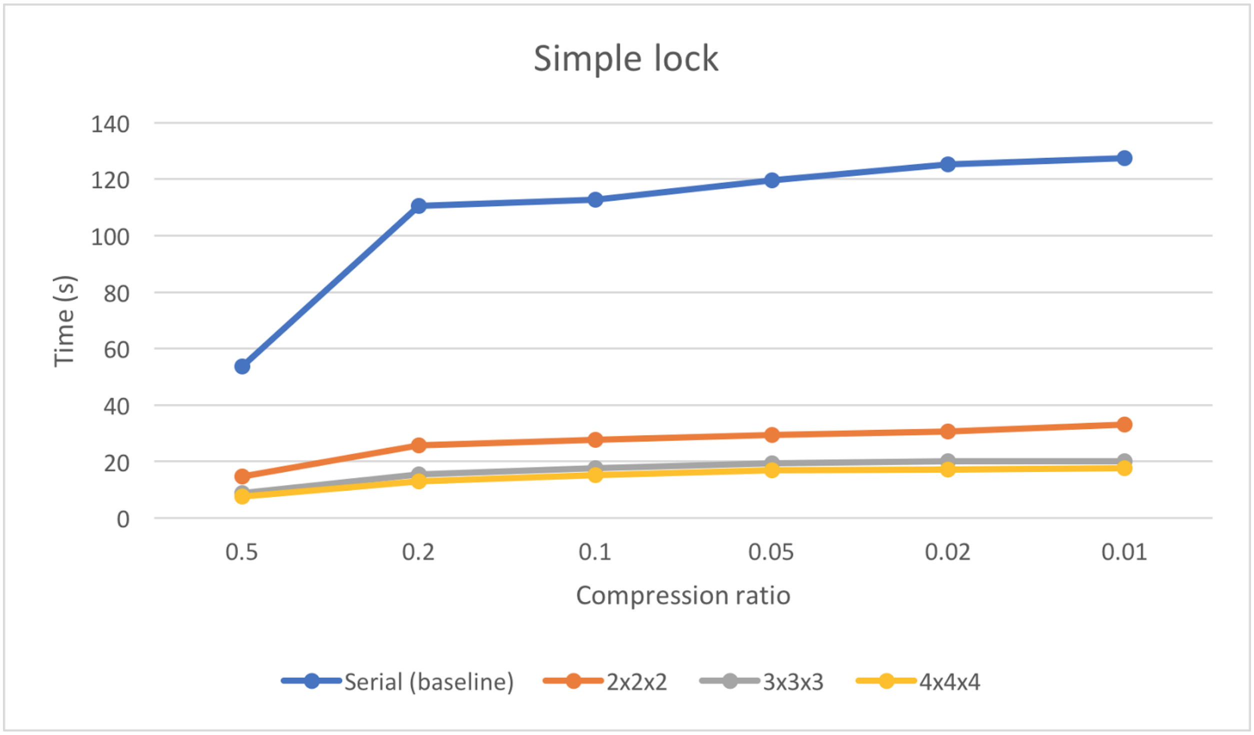


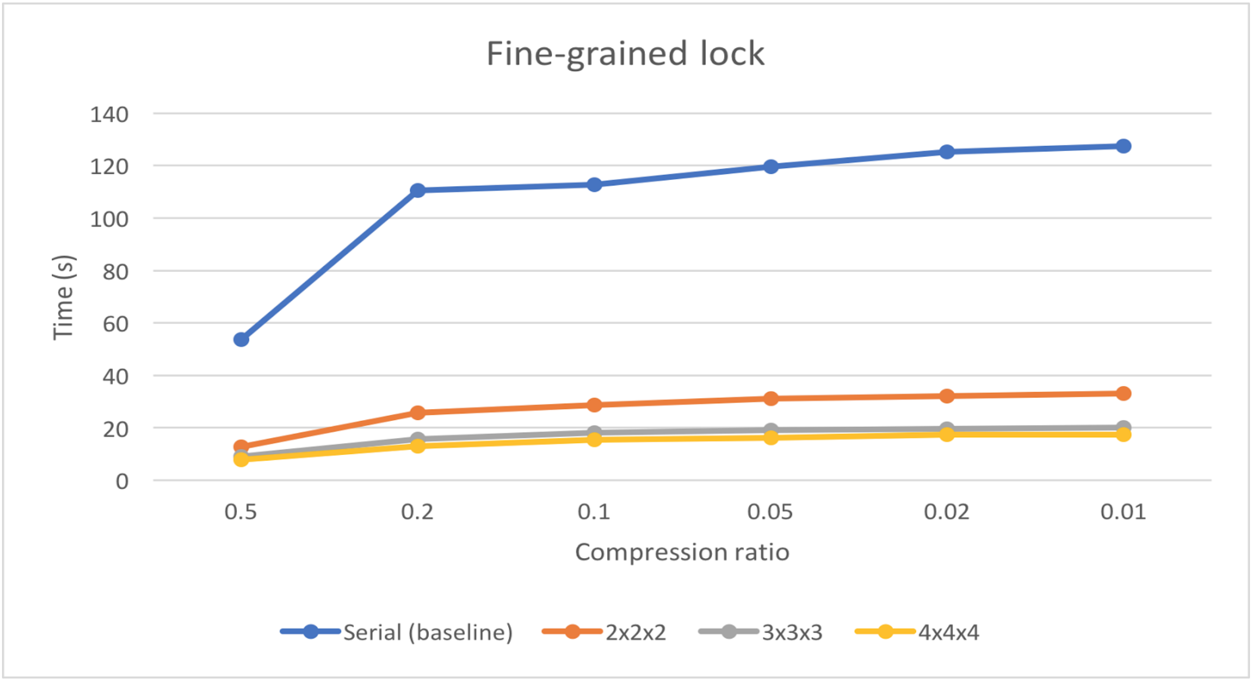


OpenMP Speedup

First we measured the running time of our various parallel mesh simplification algorithms, and compare with the baseline serial algorithm. With 8 worker threads running on 8-core machine, our parallel approaches can achieve ~6x speedup over the initial serial version of algorithm. These results come from experiments with Stanford lucy obj. The overall performance is similar with pthread implementation. This is because under the hood, OpenMp is implemented with thread model.

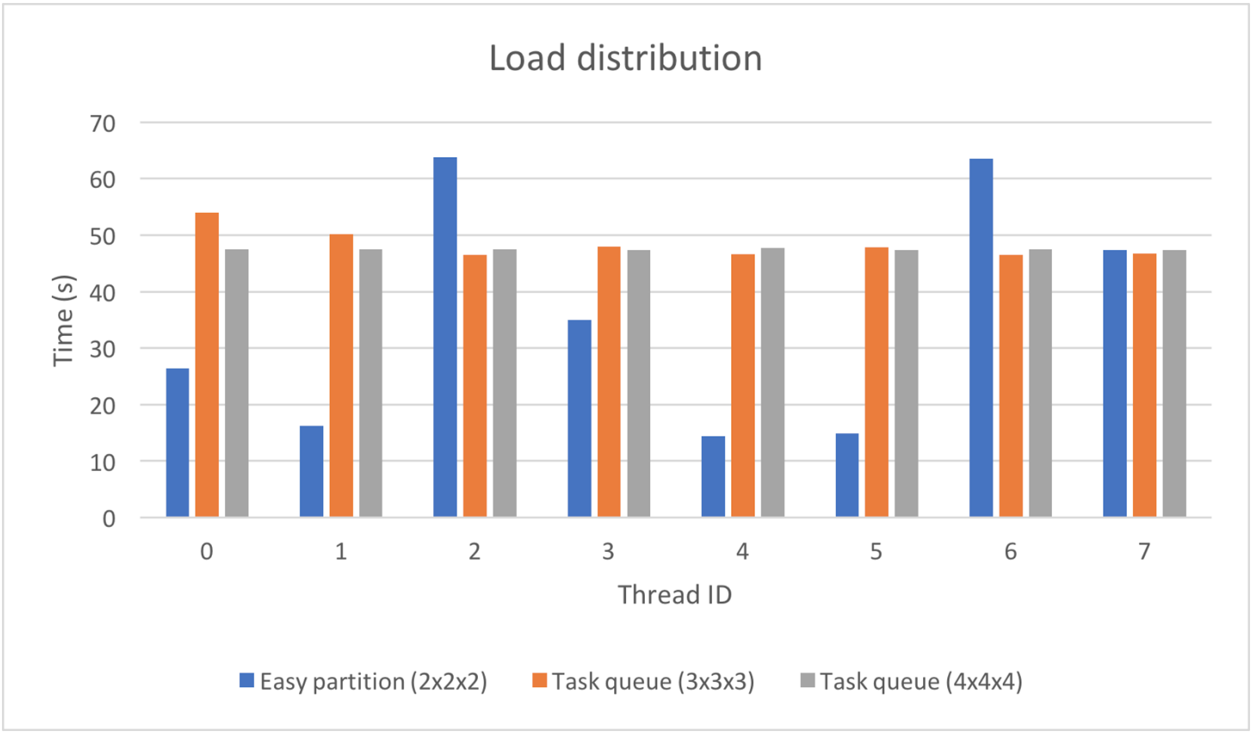

Easy partition VS task queue based partition
Time (s)
140
120
100
80
60
40
20
0
0.5
0.2
0.1
0.05
0.02
0.01
Compression ratio
Serial (baseline)
Easy partition
Task queue (2x2x2)
Task queue (3x3x3)
Task queue (4x4x4)


Simple lock
Time (s)
140
120
100
80
60
40
20
0
0.5
0.2
0.1
0.05
0.02
0.01
Compression ratio
Serial (baseline)
2x2x2
3x3x3
4x4x4



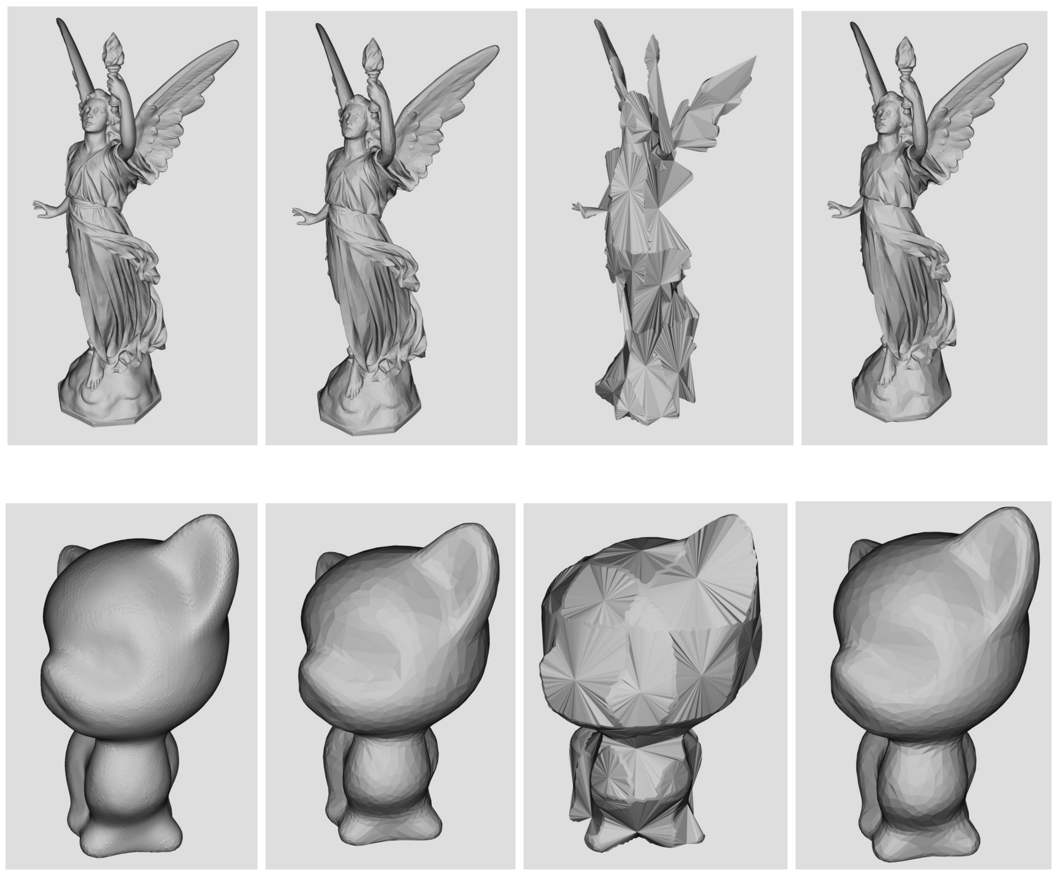
Easy Partition vs. Task Queue

Load balance issue



For these two ways to partition the input mesh, we measured the time consumed by each thread. The load is highly imbalanced in easy partition approach, because each thread can only access its own block of mesh. After processing the assigned block, that thread will be idle and the resource will be wasted. With adjustable partition granularity, task queue based partition algorithm can assign multiple blocks to a single worker thread. The work queue is pull based, which means the worker thread can pull new work from the queue if the current work is done. As a result, it achieves better load balance and shorter overall consumed time. It typically achieves better performance with finer granularity.

Simplification Quality Comparison



Original mesh Serial Task queue (4x4x4) Fine-grained lock

In order to improve the quality of output mesh when the compression ratio is low, we designed and implemented algorithms with locking to synchronize updates on margin vertices and edges. This is a trade off between speed and quality, since locking will incur extra overhead and contention problem. The quality of output mesh is nearly the same as output generated by original serial algorithm.

Reference

- Surface Simplification Using Quadric Error Metrics
- A Simple, Fast, and Effective Polygon Reduction Algorithm
- Mesh simplification in parallel
- Real-time mesh simplification using the GPU
- OpenMP Tutorial
- PThread Tutorial
- Thanks to Professor Todd's several discussions with us

Work By Each Student

In general, we spend almost the same time on this project. We discussed together to the final solutions, and we wrote the reports together. Therefore we would say as a team we both contribute 50% to this project. More specifically:

Work By Bole Chen:

- Proposed lazy update serial version mesh simplification algorithm
- Implemented the serial version algorithm
- Proposed final solution to resolve the border problem
- Parallelized our algorithm with pThread
- Report writing

Work By Haixin Liu:

- Benchmark our serial implementation and explored parallelism inside
- Tried our first approach to parallelize the algorithm
- Parallelized our algorithm with OpenMP
- Benchmarked performance of different algorithms
- Report writing