Parallel Quadric Error Simplification

Hyojae Park, Eugene Lee

March 2026

Website

`https://hyojp13.github.io/15418-parallelQES/`

Summary

We are going to implement a parallel version of Quadric Error Metric (QEM) mesh simplification using OpenMP and SIMD vectorization. Our project will focus on effective mesh partitioning, robust handling of boundary conflicts, efficient recombination of mesh partitions, and the design of multiple priority queue strategies to enable scalable parallel edge collapses while preserving mesh quality. We will analyze parallel performance, scalability, and its tradeoffs with mesh quality.

Background

Quadric Error Metric (QEM) mesh simplification is a widely used technique in computer graphics for reducing the complexity of polygonal meshes while preserving visual fidelity. It is commonly applied in level-of-detail (LOD) generation, real-time rendering, and large-scale asset processing, where reducing the number of vertices and faces is essential for improving performance without significantly degrading appearance. The core idea of QEM is to iteratively collapse edges in a mesh while minimizing a geometric error metric derived from quadric matrices associated with each vertex.

At a high level, the algorithm proceeds by computing an error metric for each vertex based on the planes of its adjacent faces. For every edge in the mesh, a collapse cost is evaluated using the combined quadric of its two endpoints. The algorithm repeatedly selects the edge with the lowest cost, collapses it into a new vertex, and updates the affected regions of the mesh. This process continues until a target level of simplification is reached. While this greedy approach produces high-quality results, it introduces strong sequential dependencies because each edge collapse modifies the mesh topology and affects subsequent cost evaluations.

Algorithm 1 Quadric Error Metric (QEM) Simplification

```
for all vertices v do
    compute quadric Q[v]
end for
for all edges (v1, v2) do
    compute collapse cost C[v1, v2]
end for
build priority queue of edges
while target not reached do
    (v1, v2) ← extract minimum cost edge
    v_new ← optimal position minimizing Q[v1] + Q[v2]
    collapse (v1, v2) into v_new
    update affected vertices and edges
    recompute quadrics and edge costs
end while
```

The parallelizable portion of our QEM implementation focuses on two key components: local quadric error computations and parallel edge processing using multiple priority queues.

First, quadric error evaluation for vertices and edges involves repeated small matrix operations, such as quadric accumulation and error computation. These operations are independent across vertices and edges, making them well-suited for data-level parallelism. We will apply SIMD vectorization to accelerate these matrix computations by evaluating multiple quadric operations simultaneously within each CPU core.

Second, instead of relying on a single global priority queue, we will introduce multiple thread-local priority queues using OpenMP. The mesh will be partitioned so that each thread processes a subset of edges and maintains its own priority queue of collapse candidates. This allows multiple low-cost edge reductions to be selected and processed in parallel across threads, enabling concurrent simplification steps. Importantly, since the quadric error cost of an edge depends only on its local neighborhood (i.e., the quadrics of its incident vertices), evaluating and selecting edges within local regions preserves the accuracy of the cost metric. As a result, parallel processing with local priority queues maintains consistency with the original QEM formulation while improving performance.

The Challenge

Challenges

QEM mesh simplification is challenging to parallelize due to its strong data dependencies and irregular workload structure. The algorithm is inherently sequential in nature because each edge collapse modifies the mesh topology and affects subsequent computations. As a result, designing an efficient parallel version requires careful handling of dependencies, data structures, and load balancing.

One major challenge lies in how to partition the mesh for parallel processing. There are multiple areas for exploration, such as whether we divide the mesh based on spatial regions, the number of vertices, the number of edges, or by identifying regions with dense/sparse connectivity. Even with a chosen strategy, the partitioning is likely to be semi-static and may not adapt well to dynamic changes in mesh topology during simplification. This can lead to load imbalance across threads, where some threads have significantly more work than others. We will therefore investigate different methods to detect these imbalances and determine how and when to combine different regions of the mesh.

Another challenge arises from replacing a global priority queue with multiple thread-local priority queues. While this enables parallelization, it introduces complications in maintaining consistency across regions. In particular, when edges lie on the boundary between partitions, updates must be reflected across multiple queues, requiring careful coordination. Additionally, since each thread only extracts the top element from its own queue, the algorithm may miss globally optimal edge collapses. For example, if the two lowest-cost edges reside in the same local queue, only one will be selected in a given iteration, potentially affecting both performance and simplification quality.

Updating the priority queues after each collapse is also nontrivial. Since edge costs depend on local neighborhood changes, updates must propagate to neighboring edges, which may belong to different partitions. This creates irregular memory access patterns and introduces communication between threads. The workload is therefore not purely data-parallel and may exhibit limited locality, as well as a relatively high communication-to-computation ratio in certain cases.

Workload Characteristics

The workload consists of a mix of compute-intensive and irregular operations. Quadric and edge cost computations involve small, repeated matrix operations that exhibit good data locality and are well-suited for SIMD and parallel execution. However, the mesh connectivity updates and priority queue operations involve pointer-based data structures with irregular memory access patterns and limited spatial locality.

There are also dynamic dependencies between iterations, as each edge collapse changes the structure of the mesh and affects future computations. This leads to divergent execution behavior, where different threads may perform different amounts of work depending on the local mesh structure.

System Constraints and Mapping Challenges

Mapping this workload efficiently onto a shared-memory multi-core system introduces several challenges. First, synchronization must be minimized to avoid performance bottlenecks, especially when managing priority queues and updating shared mesh data. Second, ensuring correctness while performing parallel edge collapses requires avoiding conflicts between threads operating on adjacent regions. Third, achieving good scalability depends on maintaining load balance and minimizing cross-thread communication, both of which are difficult due to the dynamic and irregular nature of the algorithm.

What We Hope to Learn
Through this project, we aim to understand how to effectively parallelize an algorithm with inherent sequential dependencies by identifying and exploiting its partially parallel components. We are particularly interested in exploring how design choices, such as mesh partitioning strategies and the use of multiple priority queues, affect both performance and result quality. Additionally, we hope to gain insight into the trade-offs between parallel efficiency, synchronization overhead, and approximation of the original algorithm.

Resources

Our algorithm comes from Garland and Heckert's "Surface Simplification Using Quadric Error Metrics" paper [1]. There are already pre-existing implementations of this well-known algorithm, which we plan to use as a codebase (we provide multiple options as we may run into dependency issues with some of them):

- https://github.com/mjrister/mesh-simplification
- https://github.com/samukallio/mesh-simplifier
- https://github.com/jinzihao/MeshSimplification

Because our implementation is focused on using OpenMP and SIMD, we will only depend on the CPU. We will likely not need to use any special hardware, and the GHC machines should be sufficient.

We will use the Stanford 3D Scanning Repository as our primary mesh dataset for testing and evaluation. In particular, we plan to use several standard triangle-mesh models from the repository, such as the Bunny and Dragon, because they are widely used benchmarks for mesh processing and simplification. These models provide a good range of mesh sizes and geometric complexity, which makes them suitable for comparing the sequential and parallel versions of our QEM implementation in terms of both runtime and output quality.

Goals and Deliverables

What We Plan to Achieve
Our primary goal is to implement a parallel version of the Quadric Error Metric (QEM) mesh simplification algorithm using OpenMP and SIMD on a multi-core CPU system. The implementation will include SIMD-accelerated quadric computations and parallel edge processing using multiple thread-local priority queues.

We plan to compare the parallel implementation against a trivial sequential baseline. We will first compare with the parallel version with SIMD for matrix multiplications. Then, we will adapt multiple priority queues using OpenMP

and compare this version to the sequential version in terms of execution time, scalability, and output quality.

We aim to investigate how key components of our parallelization pipeline affects performance vs mesh quality. Specifically, our goals are to devise effective methods for: performing mesh partitioning, updating the partition over time, dealing with contention on edges/vertices on boundaries, and efficiently selecting edges to collapse.

We believe that because choosing which edge to collapse is not a strictly sequential process (we do not necessarily have to collapse the edge with the greatest "cost" to get a high quality output), there may be interesting optimizations that we can make to enable fast parallelization without sacrificing noticable output fidelity.

More concretely, our goal is to achieve at least half-linear speedup with barely worse output fidelity. This would enable rapid simplification of extremely large meshes with billions of vertices.

What We Hope to Achieve

If the project progresses well, we aim to further optimize performance and deepen our analysis. We may performance analysis on the PSC machine to see how scaling is maintained as the thread count increases drastically. In particular, we hope to achieve near-linear speedup in the compute-heavy stages of the algorithm. We believe that such a result may be possible due to the number of optimizations possible such as cleverly partitioning the mesh.

If we cannot improve speedup, we will work on maintaining efficient scaling while improving the output fidelity (to be as close to the original sequential algorithm).

Fallback Goals

If full parallel edge collapse is more challenging than expected, we will focus on analyzing just one or two specific bottlenecks in the parallelization pipeline, such as creating an effective semi-static assignment approach for dividing up the mesh. Even under this reduced scope, we expect measurable performance improvement over the sequential and SIMD-only methods as devising a decent parallelization scheme with some bottlenecks is still faster than a sequential implementation.

Live Demo Plan

We plan to present a short live demo that visually compares the sequential and parallel implementations. The demo will show a mesh being simplified in real time (or near real time), with side-by-side comparison of execution time and resulting mesh quality.

Evaluation and Analysis

We aim to answer the following key questions:

How much speedup can be achieved by parallelizing QEM on a multi-core CPU?

Which components (quadric computation, edge cost evaluation, and collapse processing) benefit most from parallelism?

How does the use of multiple priority queues affect performance and result quality?

What is the trade-off between parallel efficiency and global optimality?
We will evaluate performance using execution time, speedup, and scalability across different thread counts. In addition, we will compare mesh quality between sequential and parallel implementations to ensure that the simplification remains visually and geometrically consistent.

Platform Choice

We chose C++ on the GHC machine because our project depends on efficient mesh data structures, OpenMP-based multi-threading, and SIMD vectorization, all of which are well supported in C++. GHC is a good match for our workload because QEM simplification in our design is a shared-memory, multi-core CPU problem: local quadric matrix computations benefit from SIMD, and parallel edge processing with multiple priority queues benefits from OpenMP on a multi-core machine. If the implementation achieves a strong speedup on GHC, we may additionally test it on the PSC machine to study scalability on a larger CPU system.

Schedule

Week	Dates	Planned Tasks
Week 1	(3.23 – 3.30)	• Understand the QEM algorithm and submit the proposal • Set up development environment on GHC (C++, OpenMP, SIMD) • Implement baseline sequential QEM
Week 2	(3.31 – 4.6)	• Implement SIMD optimization • Begin OpenMP parallelization
Week 3	(4.7 – 4.13)	• Design and implement multiple thread-local priority queues • Implement mesh partitioning strategy • Handle boundary edge cases • Submit Project Milestone Report
Week 4	(4.14 – 4.20)	• Improve priority queue update logic • Try a different partitioning strategy
Week 5	(4.21 – 4.27)	• Measure speedup across threads • Compare sequential vs parallel runtime • Evaluate mesh quality differences
Week 6	(4.28 – 5.1)	• Final optimization • Submit final project report • Prepare for poster session

References

[1] Michael Garland and Paul S. Heckbert, *Surface Simplification Using Quadric Error Metrics*, in Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 209–216, 1997.