15-418 Final Project Report

Alan Lee (soohyun3), Shilin Ma (shilinm)

https://alan7996.github.io/15418-project/

1 Summary

We implemented various parallelization strategies of triangulation and analyzed their performances on Scotty3D, a custom 3D graphics software package used in CMU's computer graphics course. Triangulation is a global mesh operation that subdivides every face on a mesh into triangles. We designed our solutions using halfedge mesh data structure, which defines a mesh through local connectivity descriptions between faces, edges, vertices, and halfedges. We experimented with OpenMP and OpenMPI to analyze the performance differences that varying parameter setups and implementation strategies introduced when ran on M1 Mac OSX setup versus unix setup on Gates-Hillaman cluster machines. We demonstrated the technical considerations and difficulties necessary to successful parallelization of mesh operations and achieved partial success of near-linear speedup with respect to number of cores in some setups.

2 Background

Manipulating meshes is integral to any modern 3D graphics and modelling software. The range of supported mesh manipulating operations varies greatly from implementation to implementation, often depending on the canonical mesh representation used by the software. Scotty3D is both a modelling and rendering package capable of rasterization, raytracing, animation, and particle simulation. The modelling aspect of this software was built with underlying design philosophy respecting the halfedge mesh data structure. Naturally, our goal for this project is to design, implement, and analyze various parallelization schemes for mesh operations on halfedge meshes. We define local operations to be any mesh-altering operation that is applied to a select subset of faces, edges, vertices, or halfedges of a mesh. Similarly, we define global operations to be any mesh-altering operation that gets applied to every element of faces, edges, vertices, or halfedges of a mesh.

The halfedge mesh structure is a local connectivity description which allows for fast local topology changes and has clear storage locations for data associated with vertices, edges, faces, and face-corners. The fundamental building block of halfedge mesh structure is a halfedge. Let us define an edge as a segment connecting two endpoint vertices. We define an halfedge as exactly one side of an edge. This means that a halfedge has a unique edge and face the halfedge belongs to. We complete the local connectivity description by connecting this halfedge with pointers to connected elements such as a starting vertex defining which side of the edge the halfedge belongs to, a twin halfedge pointing to the other half of the edge, and a next halfedge pointing towards the next edge of the face the halfedge belongs to. Figure 1 shows a visualization of this structure. A loop of twin halfedges traverses an edge, a loop of next halfedges traverses a face, and a loop of twin of next halfedges traverses a vertex. We internally maintain a counterclockwise ordering of halfedges per face with respect to the up-direction of the face normal to ensure consistency and the ability to complete above traversals. Refer to Scotty3D documentation and 15462/662 Computer Graphics lecture slides for more detailed explanation on implementation specifications and characteristics.

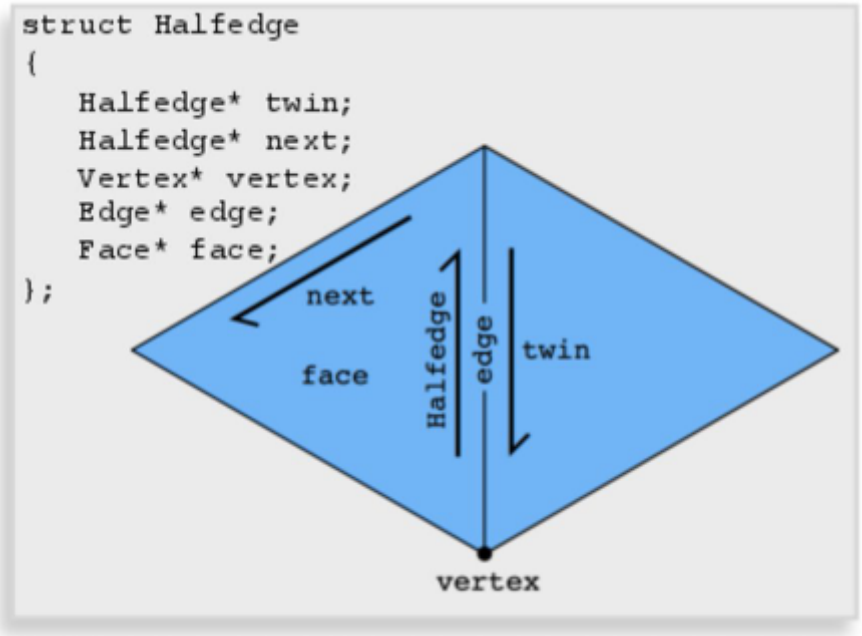

Figure 1: Visualization of halfedge data structure

As such, the halfedge mesh representation easily supports local modifications of any given manifold mesh - just update halfedge connections accordingly. However, global operations on halfedge meshes cannot be performed with a couple pointer reassignments like in local operations. We must loop through appropriate lists of faces, edges, vertices, or halfedges to identify relevant elements to modify and apply the changes. The current expectation and implementation of Scotty3D are that these loops and inner logics work in a completely serial manner in order to ease the difficulty of student assignments. This is exactly the area that parallelism can significantly improve performance by spreading the original serial workload across multiple processors / tasks / threads.

The global operation we tackle for this analysis is triangulation. This operation takes a not necessarily triangular mesh and outputs a mesh consisting only of triangular faces. This is a per-face operation where we inspect each polygonal face and check if it is already a triangle or not. If it is not a triangle, we subdivide the face into new triangular faces using the fan triangulation method. Since the fan triangulation approach on any given face does not affect any mesh connectivity of neighboring faces, it may seem trivially parallelizable by for example writing a GPU shader code for fan triangulation and generating a task calling this helper function per face. Unfortunately, the action of subdividing faces in halfedge data structure means creating new elements in global lists of faces, edge, and halfedges where the elements are connected via pointers. The parallelization for this operation therefore requires either a thread-safe scheme of creating new elements or a post-completion merging strategy to ensure no element reference conflicts or incorrect entries.

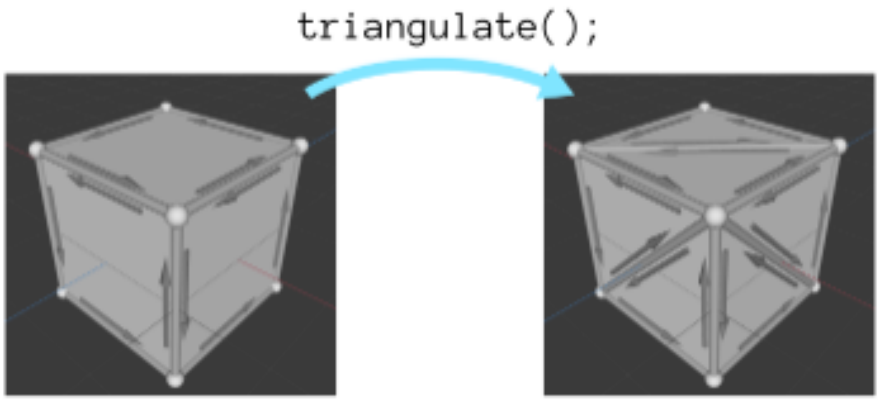

Figure 2: Triangulation on a cube mesh

The workload is hard to predict and impossible to benefit from memory coherency. Theoretically speaking, there is no guarantee on the quality of an input mesh to triangulation. The algorithm does not know for example the distribution of face degrees prior to accessing each face to judge the optimal scheduling strategy. The pointers between each element is also most certainly not contiguous in memory (for example a face may be defined by vertices in the order of indices 1, 10, 500, 2 and be perfectly valid), meaning the algorithm will constantly suffer from cache misses. The locality of triangulation lies in temporal closeness where we will keep reusing elements of a face during the process of face subdivisions. The goal of our parallelization is to distribute this series of expensive memory lookups to multiple threads or cores to hide the latency and therefore significantly improve the performance.

3 Approach

3.1 General Setup

The codebase we built our analysis on is Scotty3D. Scotty3D is a cross-platform application written mostly in C++. Since OpenMP is fully cross-platform and the primary targets of Open MPI are MacOS and Linux, our primary machine targets are Mac computers and Linux machines. More specifically, the testings were done using an M1-chip MacBook Pro and Gates-Hillman cluster's ghc unix machines. Both of these machine types run with 8 cores, we naturally mapped our problem up to 8 core or threads of parallelization.

The timings were measured using OpenMP library's omp_get_wtime() function to get wall times before and after calling triangulate function and finding their difference. This is because we encountered problems using traditional timer codes as some functions behaved differently depending on the platform. For example, using clock() in C++'s timer library measured the cumulative CPU time of all threads that have ever existed in the process since it was started for MacOS and Linux but it measured wall time on Windows. This cross-platform incompatibility hindered our collaboration efforts so we chose to use the aforementioned OpenMP's library clock function for improved consistency.

3.2 Sequential

A serial solution needs to be briefly discussed before diving into parallelization strategies we tried in order to better see why we made certain changes. A sequential polygon triangulation consists of a for loop over all faces within a mesh. For each given face, we extract a halfedge associated to the face and use it as a starting point. Following the fan triangulation approach, we take (1) the starting vertex of the extracted halfedge and (2) the vertex of its next halfedge as our first two points of our triangle. We then take the next→next of the original halfedge to get the (3) third vertex to enclose our triangle. At this enclosing part, we create new two halfedges, one edge, and one face, We then adjust the pointers we store across loop iterations to use vertices (1), (3), and the starting vertex of next of third halfedge. We repeat this process until we reach the halfedge which has a next pointer pointing to our original face halfedge, at which case we terminate. A pseudocode of this idea is presented below:

```
void Halfedge_Mesh::triangulate_seqeuential() {
    // iterate over all faces belonging to current Halfedge_Mesh
    for (Face f = faces.begin(); f != faces.end(); f++) {
        Halfedge start = f->halfedge;
        Halfedge prev = start;
        Halfedge next = start->next;

        while (next->next->next != start) {
            // create new elements

            // connect these new elements with each other and nearby elements

            // update original elements to enclose new face. the new face
            // consists of start->vertex, next->vertex, and next->next->vertex
```

```
            // iterate counterclockwise to the next face
            prev = next;
            next = next->next;
        }

        // update the elements corresponding to last triangle fan
    }
}
```

3.3 OpenMP

The most intuitive method for parallelizing triangulation is to parallelize the loop that iterates through all faces. From the beginning, it was clear that naively inserting an omp parallel for line is not enough to get our sequential version to become properly parallelized. The main problem is that appending new elements to global lists of a mesh's elements is not thread-safe. We therefore modified our halfedge mesh data structure to include a mutex lock guard and each of our mesh creation method to utilize the lock to emplace new elements. An example of this is shown in the pseudocode below:

```
Halfedge_Mesh::Halfedge Halfedge_Mesh::emplace_halfedge() {
    Halfedge halfedge;

    // std::unique_ptr<std::mutex> belonging to Halfedge_Mesh class
    mesh_lock->lock();

    if ( /* we have a freed halfedge */ ) {
        /* use the first entry in the freed halfedge list */
    } else {
        // allocate a new halfedge using the global highest element id
        // incremented by one
        halfedge = halfedges.insert(halfedges.end(), Halfedge(next_id++));
    }

    /* set halfedge pointers to default values */

    mesh_lock->unlock();

    return halfedge;
}
```

With this, we were able to modify the for loop to loop over face indices instead of face references and use an omp parallel for loop to achieve immediate parallelization gains. Since despite changing internals of some functions to become thread-safe, the interface of every function remained the same, and therefore the pseudocode from sequential version remained largely unchanged. The loop guard change to use face indices is captured by a short code snippet below:

```
void Halfedge_Mesh::triangulate_omp() {
    int f_size = (int)faces.size();
    #pragma omp parallel for
```

```
    for (int f_ind = 0; f_ind < f_size; f_ind++) {
        FaceRef f = faces.begin();
        std::advance(f, f_ind);

        /* same as in sequential */
    }
}
```

Once we had this framework support for omp parallelization, we tested with various scheduling strategies and conditions to compare and analyze the degrees of performance gains. This include comparisons between dynamic, static, and guided scheduling methods as well as parallelizing only if the size of faces list is larger than some threshold. This helped us better understand at what point the overhead of parallelization became greater than the gains from parallelization.

3.4 Open MPI

Another method for parallelizing triangulation that we tried was with OpenMPI. This method faced many challenges, which we will explain in more details below, and we ultimately conclude that it is not a good choice for this task.

We initially believed that MPI could be a suitable method for this task because triangulation can be trivially divided into independent pieces, which works well with MPI's isolated memory spaces. However, the first problem we encountered was that in the existing code base the mesh elements are stored in lists, and when they are referred to by other mesh elements using an iterator of those lists. This is incompatible with MPI because with MPI's isolated memory space, direct memory references, such as an iterator, in one process's memory space will not be valid in another process's memory space. So, it would be very difficult to merge the results after the computation on each processor. Therefore, we needed to convert the mesh into a form that is more suitable for message passing parallelism, and we decided to use a vector based implementation. The main idea is to use vectors to store in the mesh elements and use the index to refer to other elements. Below we show part of this implementation to better illustrate our idea:

```
class vHalfedge_Mesh {
    public:
        class vVertex;
        class vEdge;
        class vFace;
        class vHalfedge;

        class vHalfedge {
            public:
                //connectivity
                uint32_t twin; //halfedge on the other side of edge
                uint32_t next; //next halfedge ccw around face
                uint32_t vertex; //vertex this halfedge is leaving
                uint32_t edge; //edge this halfedge is half of
                uint32_t face; //face this halfedge borders

                uint32_t id; //unique-in-this-mesh id
                //uv coordinate for this corner of the face
```

```
            Vec2 corner_uv = Vec2(0.0f, 0.0f);
            //shading normal for this corner of the face
            Vec3 corner_normal = Vec3(0.0f, 0.0f, 0.0f);

            vHalfedge(uint32_t id_) : id(id_) { }
            vHalfedge() : id(0) { }
            friend class vHalfedge_Mesh;
    };

    class vFace {
        public:
            uint32_t halfedge; // used for connectivity
            uint32_t id;
            bool boundary = false;
            vFace(uint32_t id_, bool boundary_) : id(id_), boundary(boundary_) { }
            vFace() : id(0) { }
            friend class vHalfedge_Mesh;
    };

    class vVertex {
        public:
            ...
    };

    class vEdge {
        public:
            ...
    };

    //elements are held in these vectors:
    std::vector<vVertex> vertices;
    std::vector<vEdge> edges;
    std::vector<vFace> faces;
    std::vector<vHalfedge> halfedges;
};
```

After implementing this data structure, we were able to implement a correct version of MPI. We divided the algorithm into roughly three stages: distribution, computation and collection. In distribution, the master process sends data to all processes; in computation, each process performs the triangulation on the assigned workload; in collection, each process prepares the data and sends the updated results back to the master process. In the distribution stage, we use MPI_Bcast to send the entire mesh structure to all processes. The workload is assigned by block. During triangulation, each process would create new halfedeges, edges and faces. These new mesh elements are appended to the end of the respective vector, and in the collection stage we can easily send them back to the master process. However, there remains two other problems that we need to take care of: (1) as new faces and edges are added, the connectivity of existing mesh elements will be modified as well, and we need to take care of that; (2) After the mesh elements are merged together, we still need to ensure the correspondence between the position of an element in the vector and the index that

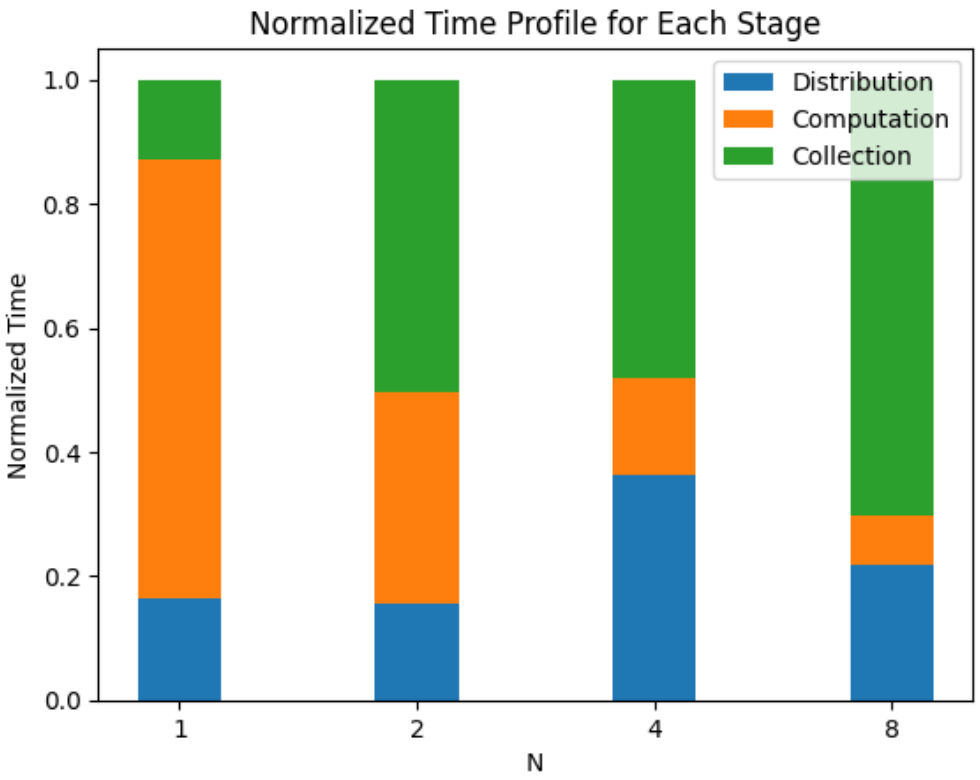

Figure 3: initial MPI implementation for bunny mesh (the behavior is similar for the other meshes)

it is being referred to by. To resolve the second issue, during collection, we send the size of the already merged elements to the child process. The child process uses this information to change the reference id of its newly created elements before send them to the master process. To resolve the first issue, we keep a vector of the modified element. When we modify the reference id of the newly created elements, we also go through this modified vector and make sure that the connectivities are correct. When the master process receives the modified vector, it uses the id as the index and replaces the element stored by the modified version. In the first version, the collection happens process by process. Figure 3 is the result of profiling this version of the code. As the number of processors used (N) increases, more and more time is spent on distributing and collecting the data.

One might be tempted to optimize the algorithm by only sending the relevant portion of the data to each child process. While this is easy to do for the faces with block assignment, because the connectivity between the halfedges, faces, vertices and edges are irrelevant of their position in vector as illustrated below in Figure 4, we cannot do easily do this for the other mesh elements. To do so would require finding all the other mesh elements that are connected to a face and reorder them to be consecutive in the vectors. Therefore, we focused on changing the collection stage. The new implementation asks each process to broadcast the number of new elements created in each category, so that the processes can modify the indices in parallel and send the data back to the master processor. The time profile for the new implementation is shown in 5. As can be seen, the proportion of the time spent on the collection stage has greatly improved.

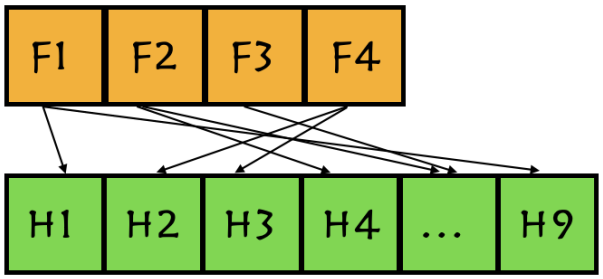

Figure 4: The faces may connect to the halfedges in any order

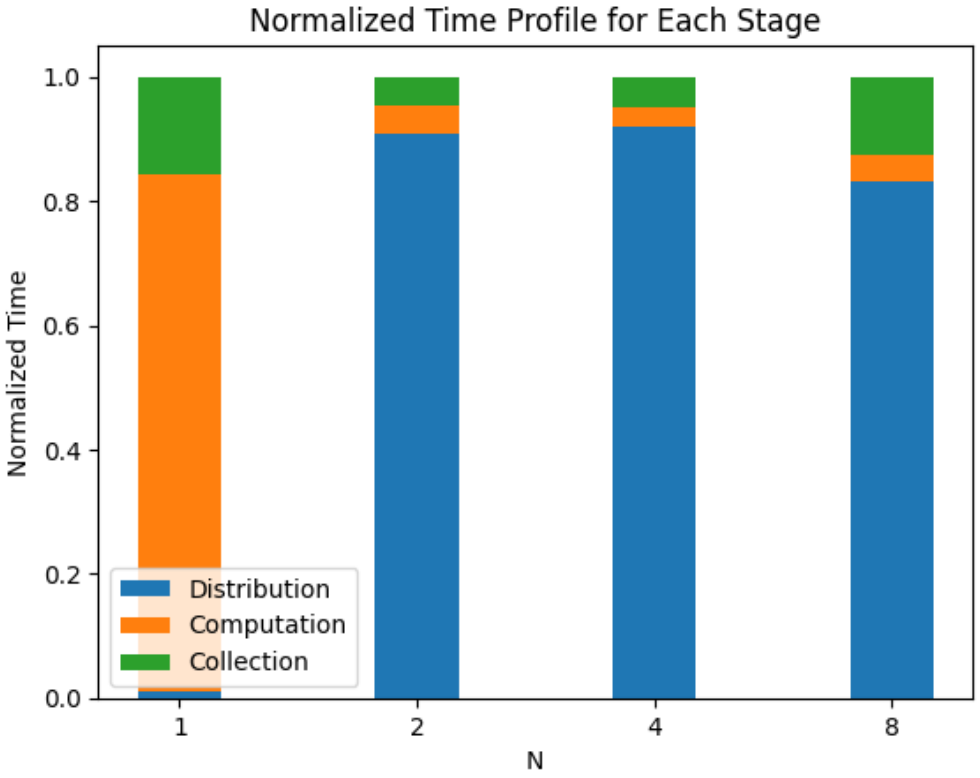

Figure 5: improved MPI implementation for bunny mesh (the behavior is similar for the other meshes)

4 Results and Discussions

4.1 Benchmarks

We evaluate the performance on three large quad meshes and three meshes created from subdividing the sphere. The large meshes are the result of linear subdividing the Stanford bunny, Keenan Crane's cow and the Stanford dragon. Because the large meshes are composed of quad-faces, the workload is trivially balanced. We have created the sphere-uneven mesh manually to test the workload balancing features, though we recognize that its size is quite small compared to the other benchmarks.

name	number of faces	number of vertices	number of edges	number of halfedges
bunny	14908	14944	29850	59700
cow	17568	17570	35136	70272
dragon	1082832	1082834	2165664	4331328
sphere-quad	960	962	1920	3840
sphere-triangulated	320	162	480	960
sphere-uneven	756	904	1658	3316

Table 1: benchmark information

4.2 OMP

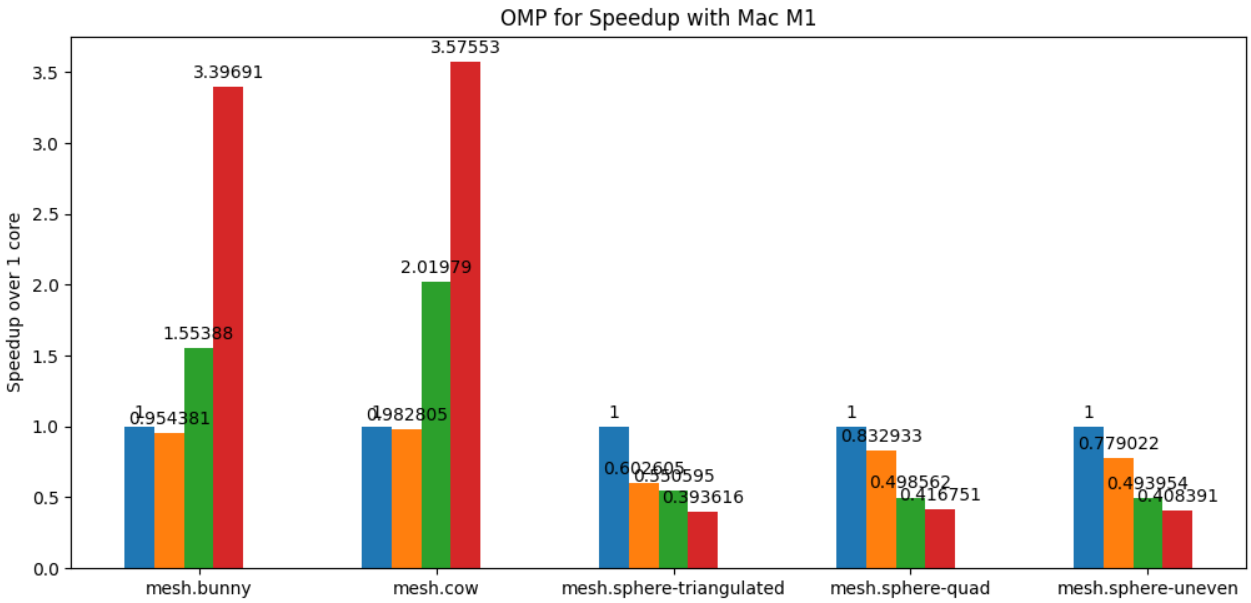

Figure 6: for

```
-- OMP Time Table ---
 benchmark       | mesh.bunny      | mesh.cow        | mesh.sphere-triangulated | mesh.sphere-quad | mesh.sphere-uneven
-----------------------------------------------------------------------------------------------------------------------------
 1               | 0.301401        | 0.436158        | 4.41074e-05              | 0.00057888       | 0.000447989
 2               | 0.315808        | 0.443789        | 7.31945e-05              | 0.00069499       | 0.000575066
 4               | 0.193967        | 0.215942        | 8.01086e-05              | 0.0011611        | 0.000906944
 8               | 0.088728        | 0.121984        | 0.000112057              | 0.00138903       | 0.00109696

-- OMP Speedup Table ---
 benchmark       | mesh.bunny      | mesh.cow        | mesh.sphere-triangulated | mesh.sphere-quad | mesh.sphere-uneven
-----------------------------------------------------------------------------------------------------------------------------
 1               | 1.0             | 1.0             | 1.0                      | 1.0              | 1.0
 2               | 0.95438051      | 0.98280489      | 0.60260539               | 0.83293285       | 0.77902189
 4               | 1.55387772      | 2.01979235      | 0.55059507               | 0.49856171       | 0.49395442
 8               | 3.39690966      | 3.5755345       | 0.39361575               | 0.41675126       | 0.40839137
```

Figure 7: for

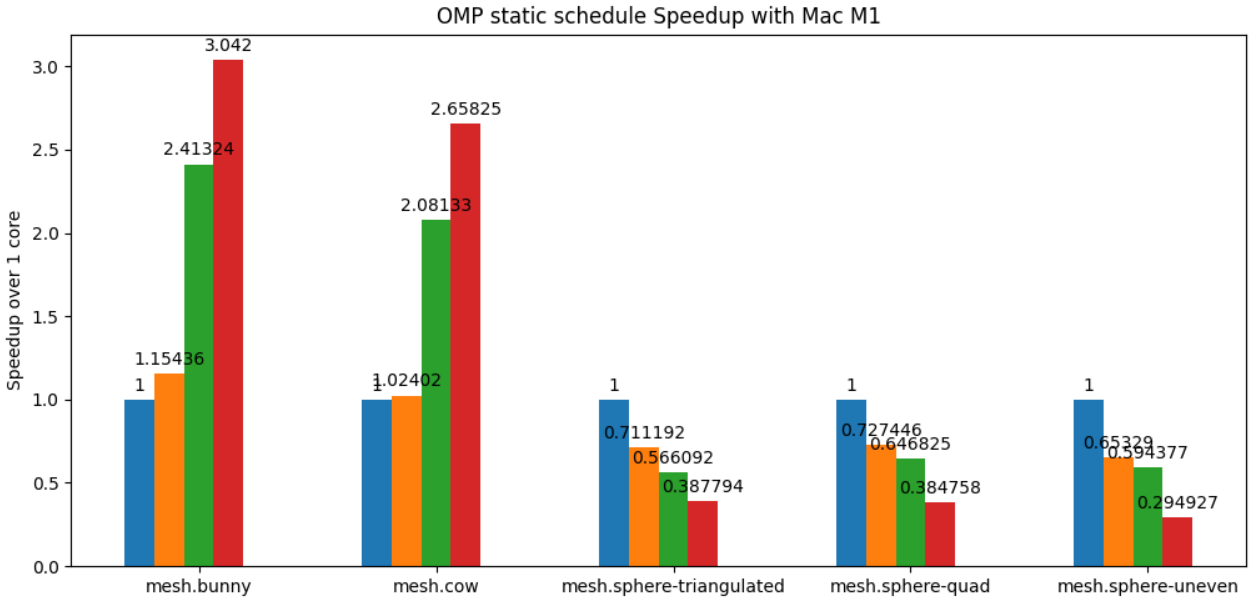

Figure 8: static schedule

```
-- OMP Time Table ---
 benchmark       | mesh.bunny      | mesh.cow        | mesh.sphere-triangulated | mesh.sphere-quad | mesh.sphere-uneven
-----------------------------------------------------------------------------------------------------------------------------
 1               | 0.301401        | 0.436158        | 4.41074e-05              | 0.00057888       | 0.000447989
 2               | 0.315808        | 0.443789        | 7.31945e-05              | 0.00069499       | 0.000575066
 4               | 0.193967        | 0.215942        | 8.01086e-05              | 0.0011611        | 0.000906944
 8               | 0.088728        | 0.121984        | 0.000112057              | 0.00138903       | 0.00109696

-- OMP Speedup Table ---
 benchmark       | mesh.bunny      | mesh.cow        | mesh.sphere-triangulated | mesh.sphere-quad | mesh.sphere-uneven
-----------------------------------------------------------------------------------------------------------------------------
 1               | 1.0             | 1.0             | 1.0                      | 1.0              | 1.0
 2               | 0.95438051      | 0.98280489      | 0.60260539               | 0.83293285       | 0.77902189
 4               | 1.55387772      | 2.01979235      | 0.55059507               | 0.49856171       | 0.49395442
 8               | 3.39690966      | 3.5755345       | 0.39361575               | 0.41675126       | 0.40839137
```

Figure 9: static schedule

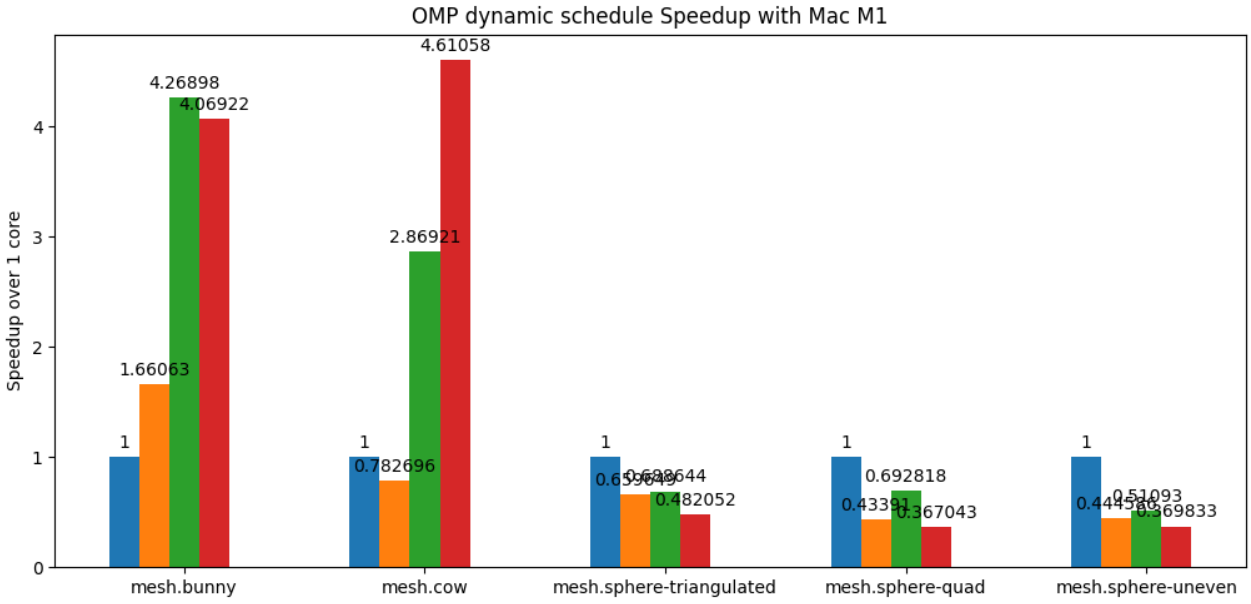

Figure 10: dynamic schedule

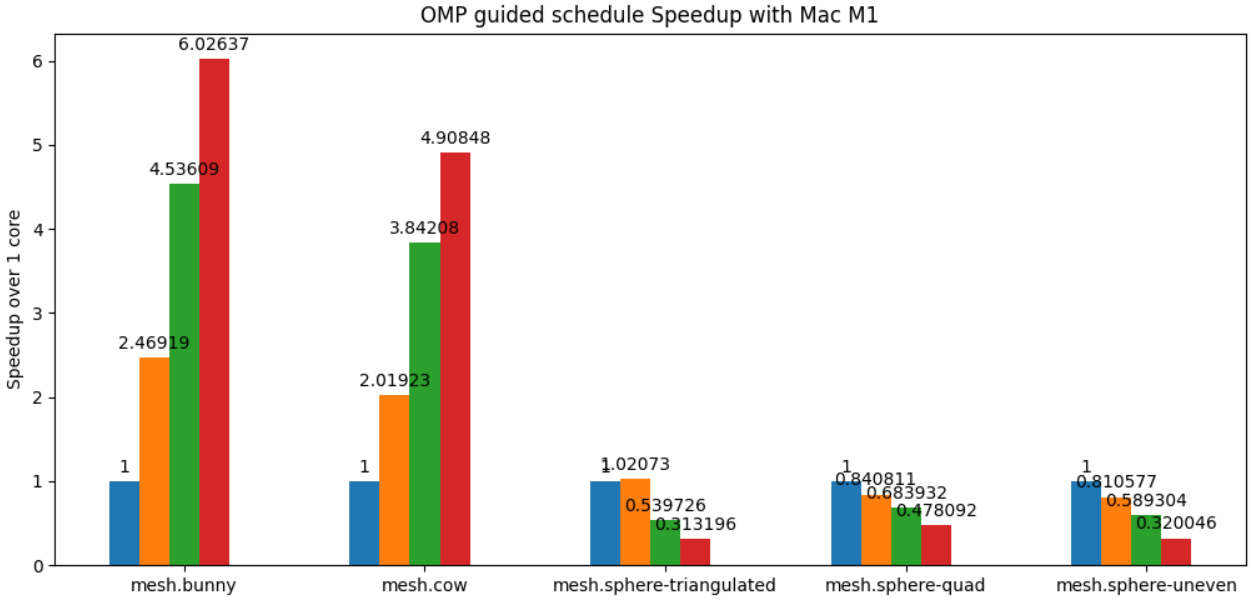

Figure 11: guided schedule

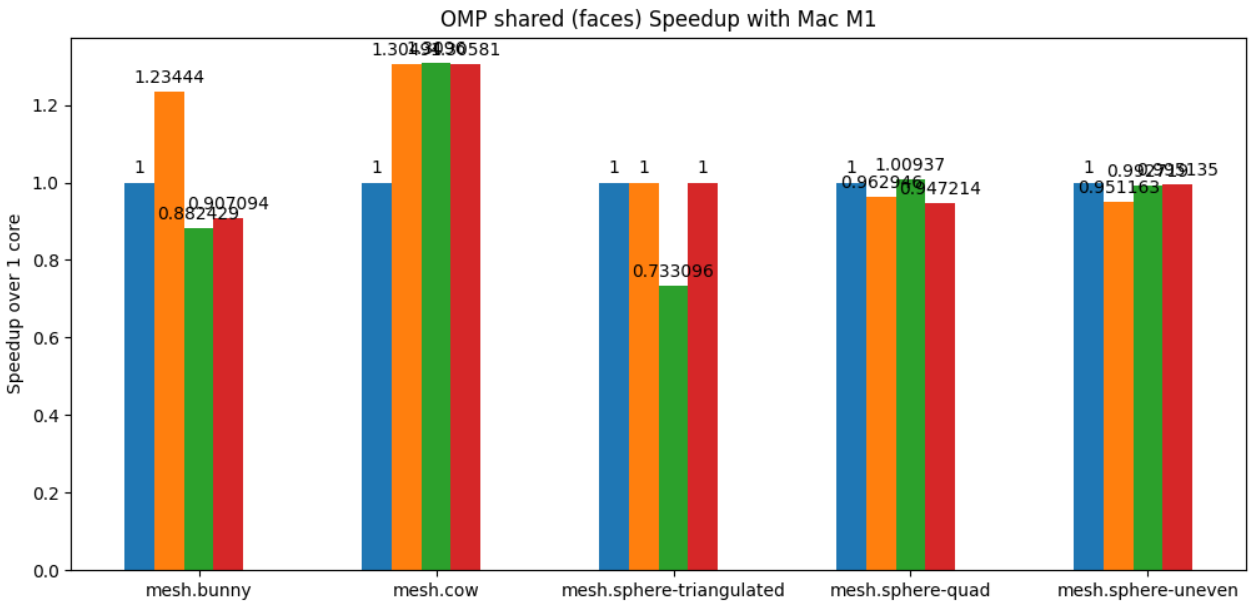

Figure 12: task shared (faces)

```
-- OMP Time Table ---
 benchmark       | mesh.bunny      | mesh.cow        | mesh.sphere-triangulated | mesh.sphere-quad | mesh.sphere-uneven
-----------------------------------------------------------------------------------------------------------------------
 1               | 0.444899        | 0.642631        | 9.8455e-05               | 0.000949383      | 0.00130997
 2               | 0.382854        | 0.53734         | 0.000148571              | 0.00112353       | 0.000960921
 4               | 0.253271        | 0.355731        | 0.000160143              | 0.00143004       | 0.00101812
 8               | 0.135276        | 0.188273        | 0.000236651              | 0.00148555       | 0.0011528
 16              | 0.0793005       | 0.119626        | 0.000437191              | 0.00179665       | 0.00149099

-- OMP Speedup Table ---
 benchmark       | mesh.bunny      | mesh.cow        | mesh.sphere-triangulated | mesh.sphere-quad | mesh.sphere-uneven
-----------------------------------------------------------------------------------------------------------------------
 1               | 1.0             | 1.0             | 1.0                      | 1.0              | 1.0
 2               | 1.16205917      | 1.19594856      | 0.6626798                | 0.84500013       | 1.36324422
 4               | 1.75661248      | 1.80650829      | 0.61479428               | 0.66388563       | 1.2866558
 8               | 3.28882433      | 3.41329346      | 0.41603458               | 0.63907846       | 1.13633761
 16              | 5.6102925       | 5.37200107      | 0.22519905               | 0.52841845       | 0.87859074
```

Figure 13: Time and speedup table for testing on GHC

4.3 MPI

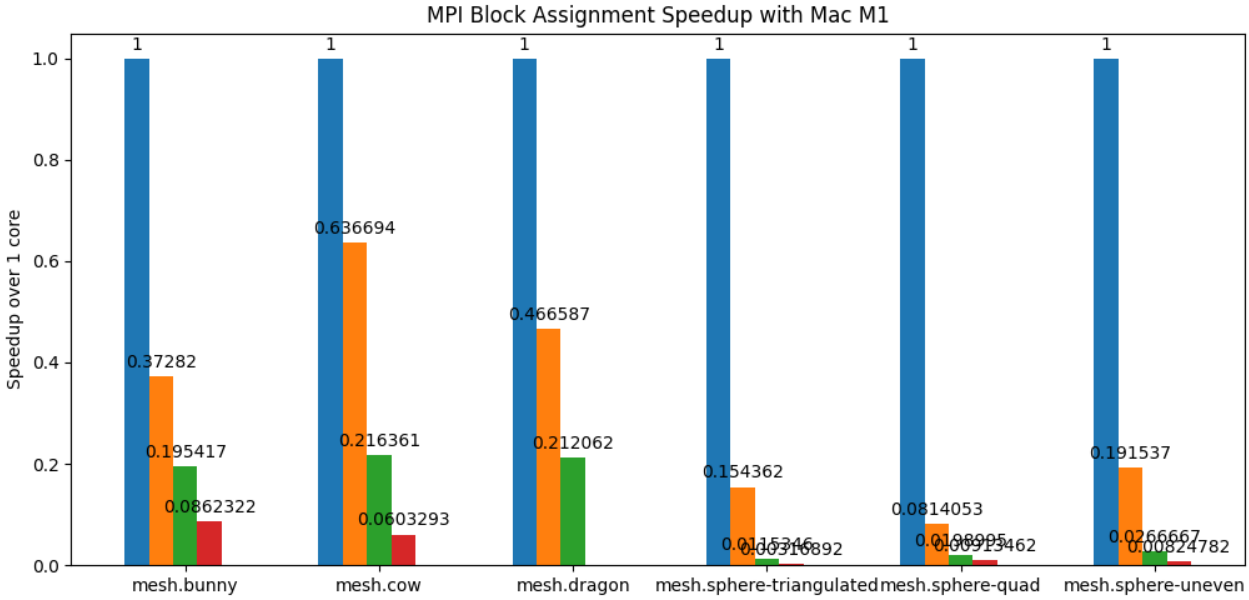

Figure 14: MPI Block Assignment on Mac. Despite the optimization efforts, the overhead of parallelism was still too great for the MPI approach. The operation has low arithmetic intensity and low cache locality. The amount of data that needs to be transferred also grows linearly with the number of processor used.

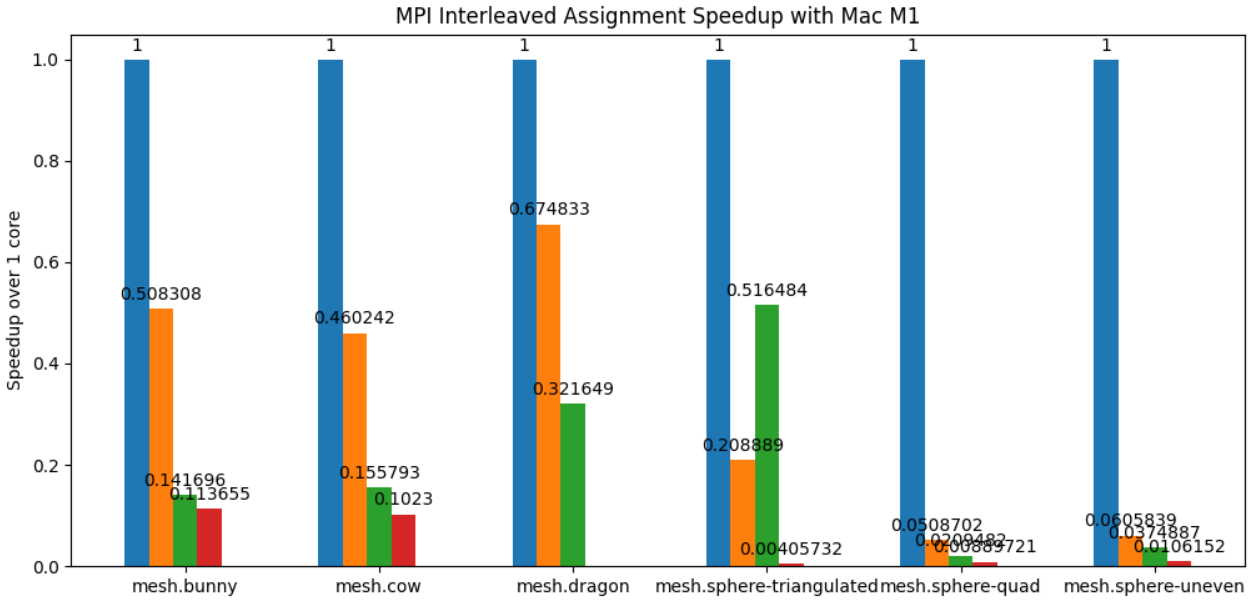

Figure 15: MPI Interleaved Assignment on Mac. Overall, interleaved assignment and block assginments produced similar results.

```
-- MPI Time Table ---
 benchmark       | mesh.bunny      | mesh.cow        | mesh.dragon     | mesh.sphere-triangulated | mesh.sphere-quad | mesh.sphere-uneven
-----------------------------------------------------------------------------------------------------------------------------------------------
 1               | 0.001499        | 0.001557        | 0.31824         | 4.7e-05                  | 7.6e-05          | 8.3e-05
 2               | 0.002949        | 0.003383        | 0.471583        | 0.000225                 | 0.001494         | 0.00137
 4               | 0.010579        | 0.009994        | 0.989403        | 9.1e-05                  | 0.003628         | 0.002214
 8               | 0.013189        | 0.01522         | nan             | 0.011584                 | 0.008542         | 0.007819

-- MPI Speedup Table ---
 benchmark       | mesh.bunny      | mesh.cow        | mesh.dragon     | mesh.sphere-triangulated | mesh.sphere-quad | mesh.sphere-uneven
-----------------------------------------------------------------------------------------------------------------------------------------------
 1               | 1.0             | 1.0             | 1.0             | 1.0                      | 1.0              | 1.0
 2               | 0.5083079       | 0.46024239      | 0.67483349      | 0.20888889               | 0.05087015       | 0.06058394
 4               | 0.14169581      | 0.15579348      | 0.32164851      | 0.51648352               | 0.02094818       | 0.03748871
 8               | 0.11365532      | 0.10229961      | nan             | 0.00405732               | 0.00889721       | 0.01061517
```

Figure 16: Time and speedup table for testing on mac (interleaved assignment)

```
-- MPI Time Table ---
 benchmark       | mesh.bunny      | mesh.cow        | mesh.dragon     | mesh.sphere-triangulated | mesh.sphere-quad | mesh.sphere-uneven
-----------------------------------------------------------------------------------------------------------------------------------------------
 1               | 0.00232806      | 0.00305535      | 0.217137        | 0.000226544      | 0.000848728      | 0.000882671
 2               | 0.0380011       | 0.0764626       | 4.64814         | 0.00161457       | 0.00192954       | 0.00168835
 4               | 0.0372521       | 0.0464165       | 4.74104         | 0.00207675       | 0.00256427       | 0.00597296
 8               | 0.122291        | 0.139534        | nan             | 0.00296511       | 0.00219154       | 0.00385913

-- MPI Speedup Table ---
 benchmark       | mesh.bunny      | mesh.cow        | mesh.dragon     | mesh.sphere-triangulated | mesh.sphere-quad | mesh.sphere-uneven
-----------------------------------------------------------------------------------------------------------------------------------------------
 1               | 1.0             | 1.0             | 1.0             | 1.0              | 1.0              | 1.0
 2               | 0.06126296      | 0.03995875      | 0.04671481      | 0.14031228       | 0.43986028       | 0.52280096
 4               | 0.06249473      | 0.06582465      | 0.04579944      | 0.10908583       | 0.33098231       | 0.14777782
 8               | 0.01903705      | 0.02189681      | nan             | 0.07640324       | 0.3872747        | 0.2287228
```

Figure 17: Time and speedup table for testing on GHC (block assignment). The performance degrades on GHC for bigger meshes.

```
Performance counter stats for 'mpirun -np 2 ./Scotty3D --run-tests a2.g1.mpi.triangulate.mesh.bunny' (20 runs):

          877.43 msec task-clock                #    1.029 CPUs utilized            ( +-  0.64% )
            3285      context-switches          #    3.777 K/sec                    ( +-  0.46% )
              24      cpu-migrations            #   27.593 /sec                     ( +-  1.26% )
           33317      page-faults               #   38.305 K/sec                    ( +-  0.01% )
      3501146801      cycles                    #    4.025 GHz                      ( +-  0.36% )  (62.42%)
      3854856292      instructions              #    1.08  insn per cycle           ( +-  0.61% )  (72.48%)
       862829741      branches                  #  991.996 M/sec                    ( +-  0.37% )  (72.53%)
        10077366      branch-misses             #    1.17% of all branches          ( +-  1.67% )  (72.13%)
      1179625844      L1-dcache-loads           #    1.356 G/sec                    ( +-  0.44% )  (71.93%)
        74698036      L1-dcache-load-misses     #    6.47% of all L1-dcache accesses  ( +-  0.47% )  (71.13%)
        18463828      LLC-loads                 #   21.228 M/sec                    ( +-  0.36% )  (70.19%)
         5590263      LLC-load-misses           #   30.35% of all LL-cache accesses  ( +-  0.90% )  (70.70%)
  <not supported>     L1-icache-loads
        10990290      L1-icache-load-misses                                         ( +-  1.13% )  (70.29%)
      1068389970      dTLB-loads                #    1.228 G/sec                    ( +-  0.59% )  (69.83%)
         1796606      dTLB-load-misses          #    0.17% of all dTLB cache accesses  ( +-  1.06% )  (59.29%)
           84488      iTLB-loads                #   97.136 K/sec                    ( +-  3.29% )  (60.55%)
          113974      iTLB-load-misses          #  145.49% of all iTLB cache accesses  ( +-  6.07% )  (60.55%)
  <not supported>     L1-dcache-prefetches
  <not supported>     L1-dcache-prefetch-misses

          0.85283 +- 0.00376 seconds time elapsed  ( +-  0.44% )

Performance counter stats for 'mpirun -np 4 ./Scotty3D --run-tests a2.g1.mpi.triangulate.mesh.bunny' (20 runs):

          1499.79 msec task-clock                #    1.663 CPUs utilized            ( +-  0.21% )
             6994      context-switches          #    4.623 K/sec                    ( +-  0.21% )
               53      cpu-migrations            #   35.031 /sec                     ( +-  3.85% )
            54597      page-faults               #   36.087 K/sec                    ( +-  0.01% )
       6543214383      cycles                    #    4.325 GHz                      ( +-  0.25% )  (64.17%)
       7217897015      instructions              #    1.11  insn per cycle           ( +-  0.57% )  (71.14%)
       1473207661      branches                  #  973.739 M/sec                    ( +-  0.66% )  (71.12%)
         14190017      branch-misses             #    0.94% of all branches          ( +-  0.74% )  (70.82%)
       1927391434      L1-dcache-loads           #    1.274 G/sec                    ( +-  0.66% )  (70.68%)
        110754591      L1-dcache-load-misses     #    5.45% of all L1-dcache accesses  ( +-  0.61% )  (70.07%)
         31725524      LLC-loads                 #   20.969 M/sec                    ( +-  0.38% )  (69.56%)
         13242577      LLC-load-misses           #   42.36% of all LL-cache accesses  ( +-  0.70% )  (67.97%)
  <not supported>     L1-icache-loads
         13023341      L1-icache-load-misses                                         ( +-  2.03% )  (66.77%)
       2029177508      dTLB-loads                #    1.341 G/sec                    ( +-  0.84% )  (67.77%)
          2285344      dTLB-load-misses          #    0.12% of all dTLB cache accesses  ( +-  0.79% )  (61.60%)
           121602      iTLB-loads                #   80.375 K/sec                    ( +-  4.00% )  (62.40%)
           375976      iTLB-load-misses          #  273.16% of all iTLB cache accesses  ( +-  7.20% )  (63.95%)
  <not supported>     L1-dcache-prefetches
  <not supported>     L1-dcache-prefetch-misses

          0.90204 +- 0.00389 seconds time elapsed  ( +-  0.43% )

Performance counter stats for 'mpirun -np 8 ./Scotty3D --run-tests a2.g1.mpi.triangulate.mesh.bunny' (20 runs):

          4011.15 msec task-clock                #    3.681 CPUs utilized            ( +-  0.29% )
            16681      context-switches          #    4.083 K/sec                    ( +-  0.25% )
              582      cpu-migrations            #  142.444 /sec                     ( +-  1.74% )
            97236      page-faults               #   23.798 K/sec                    ( +-  0.01% )
      14651647235      cycles                    #    3.586 GHz                      ( +-  0.17% )  (62.96%)
      12528901141      instructions              #    0.86  insn per cycle           ( +-  0.37% )  (70.71%)
       2632642215      branches                  #  644.335 M/sec                    ( +-  0.23% )  (71.03%)
         21520746      branch-misses             #    0.81% of all branches          ( +-  0.62% )  (71.10%)
       3707856892      L1-dcache-loads           #  907.493 M/sec                    ( +-  0.30% )  (71.24%)
        186499801      L1-dcache-load-misses     #    4.91% of all L1-dcache accesses  ( +-  1.12% )  (70.88%)
         55505640      LLC-loads                 #   13.585 M/sec                    ( +-  0.82% )  (70.01%)
         29875168      LLC-load-misses           #   54.02% of all LL-cache accesses  ( +-  0.23% )  (69.27%)
  <not supported>     L1-icache-loads
         20261142      L1-icache-load-misses                                         ( +-  3.40% )  (67.98%)
       3653327974      dTLB-loads                #  894.147 M/sec                    ( +-  0.53% )  (67.66%)
          4289576      dTLB-load-misses          #    0.12% of all dTLB cache accesses  ( +-  0.49% )  (60.41%)
           255151      iTLB-loads                #   62.448 K/sec                    ( +-  3.78% )  (61.33%)
           677184      iTLB-load-misses          #  230.46% of all iTLB cache accesses  ( +-  3.15% )  (62.50%)
  <not supported>     L1-dcache-prefetches
  <not supported>     L1-dcache-prefetch-misses

          1.08956 +- 0.00403 seconds time elapsed  ( +-  0.37% )
```

Figure 18: performance analysis on GHC (block assignment). As N increases, LLC miss increases. This is likely due to the bad access pattern inherent in the mesh structure, and explains the bad performance overall.

4.4 Discussions

To summarize the findings in above results sections, we found out that OpenMP is the most effective parallelization scheme for polygon triangulation under halfedge mesh datastructure in Scotty3D for both MacOS and Linux. We can see that once the tested mesh is past some threshold complexity (i.e. number of faces), we get near-linearly increasing speedup with respect to number of cores. Figures 6 through 11 best capture this idea. For the most simple "basic" cases using a sphere, we get in fact worse speedup compared to single core (sequential) execution when we increase the number of cores. We suspect this is likely due to the overhead of parellelization costing significantly more than sequentially going through the faces themselves.

Beyond being comparatively worse, parallelization using Open MPI proved to be objectively inefficient option. Under no test case did the Open MPI implementation achieve a positive speedup compared to sequential code, showcasing the very high cost of communication that is required for per-face parallelization as we have to copy mesh elements every time. As we increase the number of cores, the amount of communication needed increases linearly with it, leading to this seemingly exponential decrease in speedup with respect to sequential performance.

Despite our findings and conclusions, there are still many areas of further exploration. One major area of exploration is to identify specifically where the threshold for performance gain lies for OpenMP implementation. One way of trying this would be to start with a complex mesh that we experience performance gain due to parallelization, simplify (another global operation) the mesh so that it has less faces, check if we still get positive speedups, and repeat until the speedup stably turns negative. We were not able to conduct this part of the investigation ourselves due to time constraints, but the figures shown that the threshold should at least lie in between basic and mesh test cases.

5 References

https://en.wikipedia.org/wiki/Polygon_triangulation
https://en.wikipedia.org/wiki/Fan_triangulation
http://15462.courses.cs.cmu.edu/fall2023/lecture/lecture-06/slide_039
https://www.inf.ufsc.br/ bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf

links to meshes:
https://www.cs.cmu.edu/ kmcrane/Projects/ModelRepository/
https://graphics.stanford.edu/data/3Dscanrep/

6 List of Work and Distribution of Total Credit

Alan Lee :

- Created repositories for the project
- Set up, deployed, and updated project website on Github
- Implemented OpenMP implementation of triangulate
- Implemented graph visualization code for test case results
- Constructed sphere test cases for workload balance analysis
- Wrote majority of proposal, milestone report, and final report
- Constructed the slides for the poster session

Shilin Ma :

- Modified the code base to integrate OpenMP and OpenMPI
- Implemented Open MPI versions of triangulate, including the vector based mesh data structure and a mesh (de)serializer and a standalone triangulation program for testing outside of the code base
- Implemented locking modification for creating new mesh elements with list based (original) data structure used in a version of the OpenMP implementation
- Constructed testing framework and script
- Constructed basic and mesh test cases for general performance analysis
- Generated all testing results on personal M1 MacBook and connected GHC machines
- Wrote the Goals and Deliverables and Timeline sections of the proposal, most of the Goals and Deliverables Update and the Preliminary Results sections of the milestone report, and the OpenMPI related portions of the final report