ECE 4530 - Parallel Programming
Graduate Project
Conway's Game of Life on a Tetrahedral Mesh

Max Hughson

December 17, 2018

1 Project Description

For my project, I implemented a 3D version of Conway's Game of Life on an unstructured, finite, tetrahedral mesh. This project builds on the ORB code of Lab 3, with some significant changes to facilitate the Game of Life.

For my project, I generated tetrahedral meshes using gmsh. A tetrahedral mesh is a partitioning of a physical volume into a set of non-intersecting tetrahedra whose union approximately fills the physical volume. Such a mesh can be described by a set of 4-tuples in 3D space – each of which describes the 4 corners of a tetrahedron. This is implemented in gmsh as a list of points in 3D space, and a list of integer 4-tuples which index list of points.

Conway's Game of Life is normally played on an infinite square grid. Such a structure is nicely ordered, and Conway's simple set of rules can produce very interesting emergent properties on the grid. For example, one can build a glider gun that continuously shoots out structures that travel indefinitely. Some people choose to draw meaningful conclusions about human life from Conway's game. I think that's ridiculous. Real life doesn't happen on a nice uniform grid. Not only that, we don't live in planes any more, everybody lives among and on top of one another. We now need three co-ordinates to describe our relationships. People today live like vertices in a mesh. We are suspended, tenuously, barely help up by our friends, yearning to avoid the abyss below. For a 3D tetrahedral Game of Life, I could have chosen to have either the tetrahedra or the vertices be the elements of interest. For reference, a simple, in Fig. 1, I have included a simple, coarse mesh. I chose to explore the Game of Life on vertices, for 2 reasons:

1. The degree of the vertices in a mesh is not uniform, whereas each tetrahedral element has 4 face-neighbours
2. In a tetrahedral setup, we would have to deal with the exterior faces. They would need to be treated as dead neighbours, and that's too morbid.

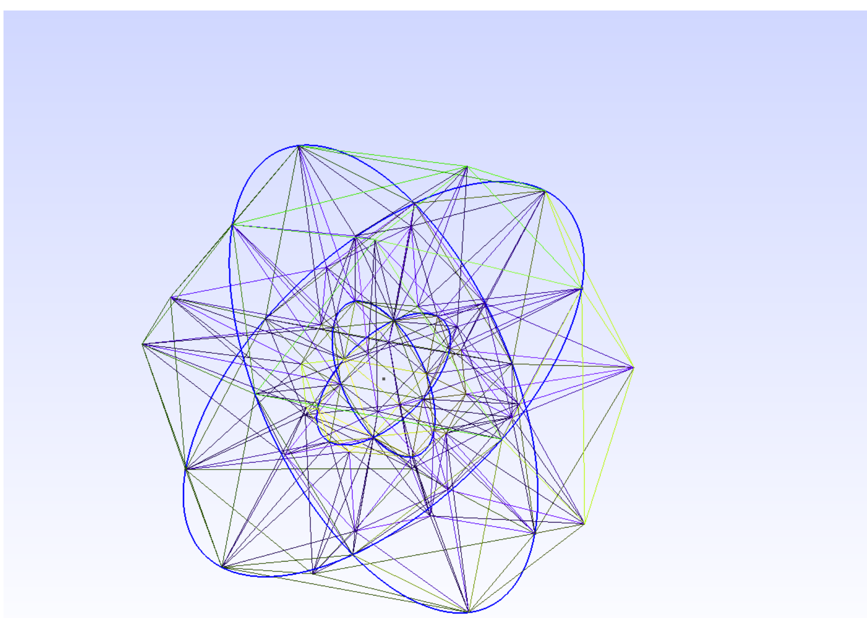

Figure 1: Sample Mesh

In Conway's Game of Life, the grid is assumed to be infinite. This is not the case on my vertex mesh. The mesh is finite. Despite this fact, we don't need to worry about treating fictitious boundary neighbours as 'dead'. Rather, the nature of the vertex-based mesh means that each of a vertex's neighbours exist in the playing area. This Game of Life is also inherently cyclic. There is a finite number of vertices, each of which can either be alive or dead. If we have v vertices, then there are 2^v possible states of the mesh. Therefore the game will repeat in as many as 2^v steps. Bear in mind that (everybody is dead) $\rightarrow$ (everybody is dead) is a valid cycle.

2 Implementation Details

My Game of Life code built off of the mesh partitioning code that was developed for Lab 3. In order to run a Game of Life simulation on an unstructured tetrahedral mesh, the following steps needed to be taken:

1. Modify certain mesh data types to include information about connectivity and life state.
2. Decide how to distribute vertices among the processors.
3. Decide on a method for initializing the mesh's life states.
4. Develop an efficient method to update the mesh's life states when they are shared among many processors.
5. Formulate a way to visualize the game.

2.1 Vertex Distribution

I chose to divide the vertices by using the ORB code that was developed for Lab 3. I figured that it would be wise to minimize the interface area between processors. ORB should do a good job of minimizing that interface area. If I had chosen to, say, apply `parallelRange` to the vertices x-position, then I would end up with many thin slices of my mesh, which would probably lead to a whole lot of communication overhead.

The ORB code from Lab 3 perform ORB on the centroids of the tetrahedral mesh elements. I left this behaviour intact, instead of modifying the code to work on vertices. This means that each processor hold a unique list of elements, but its vertices may also exist on other processors. Since some vertices are shared, it is necessary to pick one of the processors who shares the vertex to be the owner. I explored two methods for deciding who owns a shared vertex: making the lowest rank the owner, and choosing the owner randomly. Ultimately, I decided to go with minimum-rank ownership. Reasons for this decision will be discussed later.

2.2 Vertex Connectivity

In order to capture the connectivity of the vertices in the mesh, I modified contents of the `vertex_t` data type. The new data type is shown below:

```
typedef struct vertex_t
{
    double r[3];                        // Position in 3-space
    int mid;                            // Mesh ID
    bool current_state;                 // Alive or dead
    bool next_state;                    // Alive or dead
    int owner;                          // Rank who owns vertex
    bool is_shared;                     // Tell me whether this is shared
    int alive_dead[2];                  // Number of alive/dead neighbours
    std::vector<int> family;            // List of ranks which share vertex
    std::vector<vertex_t*> neighbours;  // List of pointers to local neighbouring vertices
    std::vector<int> global_nbr_mids;   // List of global connections
    std::vector<bool> nbr_was_counted;  // Used during vertex update
}
vertex_t;
```

The vector `neighbours` stores pointers to local neighbouring vertices, making tabulation of neighbours' alive/dead counts simple. The vector `family` is a list of the ranks which share the vertex. The determination of this connectivity information is non-trivial. This is handled by the function `Mesh::calculateVertexConnectivity`

The first step toward determining the connectivity of the mesh is for each processor to loop over its local elements, and make a list of which local vertices are connected to each other, indexed by their mesh ID. The code for this is shown below:

```
    std::map<int, std::vector<int> > mid_connections_by_mid;

    for(int ie = 0; ie < elements_3d_.size(); ie++)
    {
        element_t * cur_ele = &(elements_3d_[ie]);
        for(int iv = 0; iv < (cur_ele->nvert-1); iv++)
        {
```

```
            int midi = cur_ele->vertex_mids[iv];
            for(int jv = iv+1; jv < (cur_ele->nvert); jv++)
            {
                int midj = cur_ele->vertex_mids[jv];
                mid_connections_by_mid[midi].push_back(midj);
                mid_connections_by_mid[midj].push_back(midi);
            }
        }
    }
```

This leads to plenty of over-counting, so these lists are sorted and shrunk after all the elements are processed. At the same time, a map from vertex mesh ID to local linear index is generated. This map is makes the update step flow smoothly.

```
    // You probably double-counted a bunch. Make sure each list
    // contains no copies.
    int my_highest_mid = 0;
    for(int iv = 0; iv < vertices_.size(); iv++)
    {
        int midi = vertices_[iv].mid;
        mid2lindx_[midi] = iv;
        if(midi > my_highest_mid) my_highest_mid = midi;
        sort(mid_connections_by_mid[midi].begin(), mid_connections_by_mid[midi].end());
        mid_connections_by_mid[midi].erase(unique(mid_connections_by_mid[midi].begin(), mid_connections_by_mid[midi].end()), mid_connections_by_mid[midi].end());
    }
```

Once all the local connectivity information is determined, the processors need to figure out the connectivity of vertices between partitions. This is done in a loop over the global mesh IDs. For each global mesh ID, if a processor has that vertex locally, it attempts to attain ownership of the vertex. The lowest rank processor is granted ownership of the vertex. The owner vertex determines the global list of mesh IDs which are connected to this vertex. This list is useful during the update step, when the owner may need to process redundant messages about this vertex's neighbours.

```
    // Determine ownership and families
    for(int midi = 0; midi <= global_highest_mid; midi++)
    {
        // Do I have a vertex with this mid?
        std::map<int,int>::iterator it;
        it = mid2lindx_.find(midi);

        bool vertex_exists_locally = (it != mid2lindx_.end());

        std::vector<int> send_existence(nproc, vertex_exists_locally);
        std::vector<int> recv_existence(nproc);
        // Tell everyone else whether I have this mid locally.
        // Find out who else has this mid locally.
        MPI_Alltoall(
            &(send_existence[0]),
            1,
            MPI_INT,
            &(recv_existence[0]),
            1,
            MPI_INT,
            MPI_COMM_WORLD
        );

        int vertex_owner = nproc;
        // Pick the lowest rank with a local copy of this mid as then
        // owner.
        for(int iproc = 0; iproc < nproc; iproc++)
        {
            if(recv_existence[iproc])
            {
                vertex_owner = iproc;
                break;
            }
        }

        if(vertex_exists_locally)
        {
            int lindx = mid2lindx_[midi];
            vertex_t * cur_v = &(vertices_[lindx]);
            cur_v->owner = vertex_owner;
            // Fill the family for this vertex
            cur_v->family.resize(0);
            for(int iproc = 0; iproc < nproc; iproc++)
            {
                if(recv_existence[iproc]) cur_v->family.push_back(iproc);
            }
        }
        if(vertex_owner < nproc)
        {
            // If this mid was actually claimed by someone, then
            // make sure the owner knows all of the global mids that
            // are connected to this mid.
            std::vector<std::vector<int> > send_mid_connections(nproc);
            std::vector<std::vector<int> > recv_mid_connections(nproc);
            if(vertex_exists_locally)
            {
                // Tell the owner about all the connections I am aware
                // of.
                send_mid_connections[vertex_owner] = mid_connections_by_mid[midi];
            }
            MPI_Alltoall_vecvecT(send_mid_connections,recv_mid_connections);
            if(vertex_owner == rank)
            {
                for(int iproc = 0; iproc < nproc; iproc++)
```

```
                {
                    for(int icon = 0; icon < recv_mid_connections[iproc].size();icon++)
                    {
                        // Add this connection to my list of global connections.
                        vertices_[mid2lindx_[midi]].global_nbr_mids.push_back(recv_mid_connections[iproc][icon]);
                    }
                }
                // Delete any copies from the list of global connections.
                sort(vertices_[mid2lindx_[midi]].global_nbr_mids.begin(), vertices_[mid2lindx_[midi]].global_nbr_mids.end());
                vertices_[mid2lindx_[midi]].global_nbr_mids.erase(unique(vertices_[mid2lindx_[midi]].global_nbr_mids.begin(), vertices_[mid2lindx_[midi]].global_nbr_mids.end()), vertices_[mid2lindx_[midi]].global_nbr_mids.end());
                vertices_[mid2lindx_[midi]].nbr_was_counted.resize(vertices_[mid2lindx_[midi]].global_nbr_mids.size());
            }
        }
    }
```

2.3 Initialization of the Mesh

In order to run a Game of Life simulation, the mesh needs to have some initial state. This is the responsibility of the function `Mesh::populateMeshVertices`. Basically, each processor loops over its vertices and assigns an initial state to each vertex according to some rule. If a vertex is shared among processors, then the owner decides its initial state and communicates that state to the other family members. Currently, my code assigns initial states in a deterministic way, depending on the vertices' mesh IDs. Communication among processors is not strictly necessary in this case, but if I wish to assign states randomly, then communication would become necessary.

2.4 Vertex State Update

The distributed update step is handled in the function `Mesh::updateVertexStates`. In this function, there are six loops which serve to efficiently communicate vertex states between processors, and calculate and broadcast new states to families. This function takes advantage of the non-blocking nature of `MPI_Send` to communicate vertex states quickly.

In the first loop, each processor looks at each of its vertices, trying to find vertices which it does not own. These vertices' owners will eventually be responsible for calculating their next state, so they need to know the current state of all of the vertices in their neighbourhoods. If a processor has a local vertex neighbouring someone else's vertex, it sends the owner the neighbour vertex's mesh ID, the host vertex's mesh ID, and the neighbour's state, wrapped up in a `helper_vertex_t` structure.

```
typedef struct helper_vertex_t
{
    int host_mid;
    int nbr_mid;
    bool nbr_state;
}
```

Next, the processors collectively swap these helper vertices so that they have all they need to update their local vertices.

In the second loop, each processor tallies up the alive/dead counts for all of its local vertices which it owns. If any local vertices are owned by someone else, then they will be updated by someone else, so it is unnecessary to perform any work on them. When a vertex's neighbour's state is tallied, a flag is set in an accompanying array to indicate that the connection between the host and neighbour has already been considered.

In the third loop, the helper vertices are tabulated. This is where the map from mesh ID to local vertex index comes in handy. Each processor can look at its helper vertices' mesh IDs and quickly find out their corresponding local vertex. It is possible, at this point, that some of the helper vertices are supplying redundant information. The mesh IDs in each helper are checked against the `nbr_was_counted` array to prevent double-counting vertex states.

The fourth loop is where the vertex states actually get updated. The actual update calculation will be discussed later. A processor only updates the state of vertices which it owns. Once a vertex's state is updated, the processor uses `MPI_Send` to communicate the new state to the other members of the vertex's family. These messages are not immediately received – the corresponding `MPI_Recv` commands don't appear until the next loop. The fact the send command is non-blocking allows each processor to complete this loop without checking whether the message made it to its destination.

The fifth loop takes care of receiving all of the floating `MPI_Send` commands. Each processor loops through its vertices, looking for vertices which it does not own. If such a vertex is found, the processor grabs the message from MPI and updates its local copy of that vertex with its new information. Finally, in the sixth loop, each processor loops through its vertices and moves the value from `next_state` into `current_state`. Additionally, the processor zeros out the counts in `alive_dead`, and clears all the `nbr_was_counted` flags to prevent contamination in the next update.

2.4.1 State Update Rules

In the 2D Game of Life, cells are updated as follows:

IF Cell is currently alive
    IF Cell has 2 or 3 live neighbours
      Cell stays alive
    ELSE
      Cell dies
ELSE Cell is currently dead
    IF Cell has 3 live neighbours
      Cell becomes alive
    ELSE
      Cell stays dead

These rules are meant to emulate the effect of community on organisms' health. Organisms can die of loneliness or overpopulation, and cells can become alive if the population density is correct. These rules could be directly applied to the vertex mesh, but that would fail to capture the intention of the rules. The rules focus on the population *density* in a cell's neighbourhood. The size of a cell's neighbourhood is highly variable on the vertex mesh. For example, consider the histogram shown in Fig. 2, which shows the degrees of all the vertices in a sample mesh. There is a wide spread of vertex degrees, thus the state update rules need to consider the *proportion* of neighbouring vertices which are alive or dead.

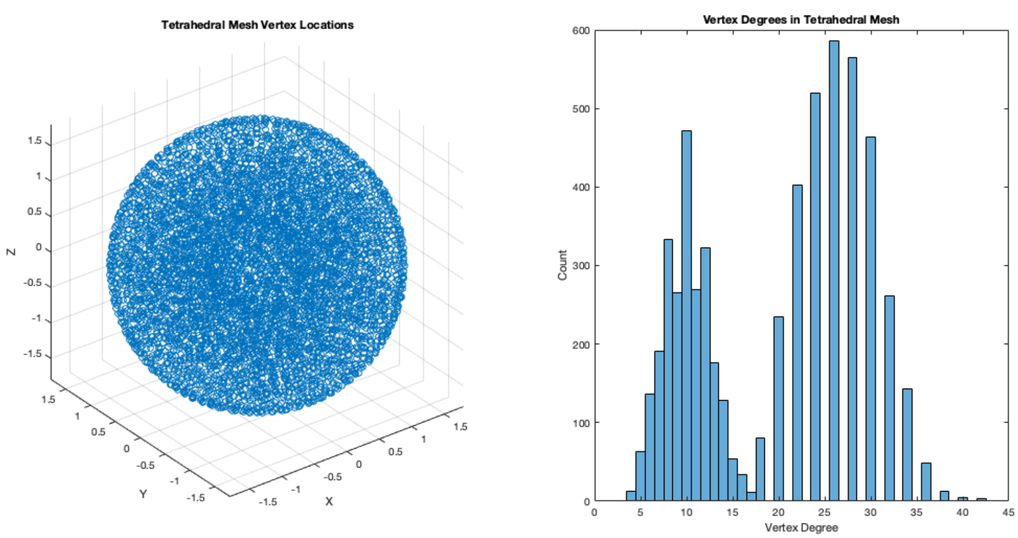

Figure 2: Vertex Degrees for a Fine Tetrahedral Mesh

My state update logic is found in `GOL_CalculateNextState`, shown below. These values seem to produce populations that don't immediately die out.

```cpp
bool GOL_CalculateNextState(bool current_state,int n_alive, int n_dead)
{
    double life_rate = (double)(n_alive + n_dead);
    life_rate = 1/life_rate;
    life_rate *= (double)(n_alive);
    bool next_state = current_state;
    if(current_state)
    {
        if(life_rate < 0.2999)
        {
            // Die of loneliness
            next_state = false;
        }
        else if(life_rate < 0.5111)
        {
            // Keep living because you're in a good community.
            next_state = true;
        }
        else
        {
            // Die from overpopulation
            next_state = false;
        }
    }
    else
    {
        if((0.2999 <life_rate) && (life_rate < 0.5111))
        {
            // Get born
            next_state = true;
        }
        else
        {
            // Stay dead
            next_state = false;
        }
    }
    return(next_state);
}
```

2.5 Visualization

I decided to use Matlab for my visualization because I didn't want to figure out how to make 3D plots in openCV. My c++ code can append its mesh's state to a text file at each iteration. When the log file is created, each processor writes down a list of its vertices' positions. For each subsequent write, each processor just loops through its vertices and writes down a 0 or 1, to mark its vertices' states. I included a command-line parameter to turn off this file I/O, since it is an inherently slow, serial process. If logging is disabled, then only the vertices' positions and their final states are recorded.

I then wrote a Matlab script which parses the log file and displays the vertices' states with a 3D scatter plot, pausing between each state. An animation of a game of life simulation can be found in the included `./Mod2HalfAliveAnimateGOL.mov` file. In this example, the vertices with positive x-position were initially alive, and the ones with negative x-position were initially dead. Red vertices are dead and blue vertices are alive.

3 Results and Analysis

3.1 Speedup Analysis

Processing speed was tested by running games of life on four different meshes of a sphere. There various meshes are summarized in Table. 1.

Table 1: Mesh Statistics

	Nodes	**Elements**
Coarse Mesh	62	270
Medium Mesh	217	734
Fine Mesh	1113	4745
Super Fine Mesh	5797	28874

Looking at the Tables 2, 4, and 3, it seems like all this work was for naught. For most of the meshes, the number of vertices processed per second decreases when more processors are added. The memory footprint does not seem to have any better results. For the meshes, parallelizing the problem generally leads to slower performance with meagre memory savings.

Table 2: Iterations Per Second

	Number of Processors					
	1	2	3	4	5	6
Coarse Mesh	20900	8630	9060	6120	6020	5120
Medium Mesh	6280	3960	4210	3030	3310	3160
Fine Mesh	1080	940	987	847	1020	978
Super Fine Mesh	168	198	216	200	247	247

Table 3: Vertices Updated Per Second [$\times 10^6$]

	Number of Processors					
	1	2	3	4	5	6
Coarse Mesh	1.3	0.535	0.562	0.379	0.373	0.317
Medium Mesh	1.37	0.856	0.914	0.658	0.718	0.686
Fine Mesh	1.2	1.05	1.01	1.14	1.14	1.09
Super Fine Mesh	0.974	1.15	1.25	1.16	1.43	1.43

The fine mesh, unlike the other, smaller meshes, does benefit from parallel computation with several processors. This suggests to me that the behaviour expressed in these tables is not the asymptotic behaviour of this algorithm. It is probably true that in order to see good speedup, it would be necessary to process a very, very large mesh file. I do not have the means to process such a file on my computer, but there bigger computers which would be capable of handling such a task. A bigger problem would see better speedup because the volume a mesh grows much faster than its surface area. During the update step, the processors need to communicate with each other to determine the states of vertices which are shared between processors. These shared vertices are only found on the edge of the mesh partitions on interface surfaces. As the problem size increase, the volume of a problem grows as a cubic function, while the interface surface only grows as a quadratic function. Therefore, for sufficiently large problems, the interface vertices would make up a vanishingly small portion of a processor's work. To illustrate this point, consider the impact of turning off the ORB partition. Without the ORB partition, the mesh's elements are distributed among the processors almost randomly. This leads to much more sharing of vertices, and much larger interface areas. Table 5 shows the number of iterations completed per second with and without ORB enabled. With 4 processors running, ORB leads to a 5x speedup.

4 Conclusion

For this report, a parallel program was developed which is capable of running a simulation in the style of Conway's Game of Life, on the vertices of a three-dimensional, unstructured mesh of arbitrary geometry. The geometrical differences between a standard grid and and unstructured mesh meant that it became necessary to determine complex connectivity information in order to iterate the Game of Life on a mesh. The notion of a single Game of Life update also had to be re-worked, in order to fit with this new problem. Furthermore, the act of parallelizing the problem introduced the problem of redundant neighbourhood calculations. In order to solve this problem, the parallel code had to not only tally up the states of a vertex's neighbours,

Table 4: Memory Per Processor in Megabytes

	Number of Processors					
	1	2	3	4	5	6
Coarse Mesh	2.2	2.4	2.4	2.4	2.5	2.5
Medium Mesh	2.4	2.7	2.6	2.6	2.8	2.8
Fine Mesh	4.0	5.0	4.6	4.6	4.4	4.3
Super Fine Mesh	14.8	15.1	13.7	11.1	9.8	10.6

Table 5: Impact of ORB on Iteration Rate

	Number of Processors					
	1	2	3	4	5	6
Fine Mesh with ORB	1080	940	987	847	1020	978
Fine Mesh without ORB	1080	1040	223	170	173	170

but also ensure that every state is counted exactly once. This redundancy-checking ultimately adds a large amount of overhead to the parallel code, meaning that this Game of Life implementation is best suited to run on big computers and with huge meshes.

A Code

A.1 Main

```cpp
#include <iostream>
#include <mpi.h>
#include <vector>
#include <stdlib.h>
#include <sstream>
#include "Mesh.h"




int main(int argc, char** argv)
{
    int rank,nproc;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    srand(MPI_Wtime() + rank);

    std::stringstream mesh_name;
    std::stringstream cell_filename;
    cell_filename << "CellMatrix_P" << nproc << ".txt";
    mesh_name << argv[1] << ".msh";

    int n_iter = atoi(argv[2]);
    double pop_percent = atof(argv[3]);
    double pop_rate = (pop_percent)/100.0;
    bool log_each_iteration = (atoi(argv[4])>0);
    std::cout << "pop rate = " << pop_rate << std::endl;
    Mesh BallMesh(mesh_name.str());

    BallMesh.partitionMesh();
    BallMesh.calculateVertexConnectivity();
    BallMesh.populateMeshVertices(pop_rate);
    BallMesh.writeCellMatrixFile(cell_filename.str(),false);
    //BallMesh.outputStatistics();
    BallMesh.updateVertexStates();
    double tstart = MPI_Wtime();
    double tnow;
    double iter_per_s;
    MPI_Barrier(MPI_COMM_WORLD);
    int disp_counter = 0;
    for(int iter = 0; iter < n_iter; iter++)
    {
        if(log_each_iteration)
        {
            BallMesh.writeCellMatrixFile(cell_filename.str(),true);
        }
        BallMesh.updateVertexStates();
        if((disp_counter++ > 200))
        {
            if(rank == 0)
            {
                disp_counter = 0;
                tnow = MPI_Wtime();
                iter_per_s = iter/(tnow-tstart);
                std::cout << "\r" << iter << " / " << n_iter << ", " << iter_per_s << std::flush;
            }
        }
    }
    if(rank == 0) std::cout << std::endl;
    BallMesh.writeCellMatrixFile(cell_filename.str(),true);
    MPI_Finalize();

    return(0);
}
```

A.2 Mesh

```cpp
#include <iostream>
#include <vector>
#include <fstream>
#include <math.h>
#include <cassert>
#include <mpi.h>
#include <sstream>
#include <map>
#include <iostream>
#include <iomanip>
#include "./Mesh.h"
#include "./MeshUtilities.h"

using namespace std;

//------------------------------------
// Constructor
//------------------------------------
Mesh::Mesh(string filename)
{
    vertices_.resize(0);
    elements_3d_.resize(0);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc_);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank_);
    readMesh(filename);
    getElementVertices();   //complete the vertices we have for local elements
    createElementCentroidsList();    //create the list of centroids as private member
}

//------------------------------------
```

```
// Destructor - free dynamically allocated
// memory here
//------------------------------------
Mesh::~Mesh()
{

}

//------------------------------------
// Get a pointer to a vertex from the
// mesh id (mid). Returns NULL if not
// found.
//------------------------------------
const vertex_t* Mesh::getVertexFromMID(int vertex_mid) const
{
    vertex_t search_vertex;
    search_vertex.mid = vertex_mid;
    void* result = bsearch(&search_vertex, &vertices_[0], vertices_.size(), sizeof(vertex_t), searchVertexByMIDPredicate);
    if (result != NULL) // found vertex
    {
        const vertex_t* found_vertex = (const vertex_t*)result;
        return found_vertex;
    }
    else
    {
        return NULL;
    }
}

//------------------------------------
// Read a Gmsh formatted (version 2+)
// mesh. Creates list of vertices and
// elements. Not guaranteed to have
// vertices for local elements.
//------------------------------------
void Mesh::readMesh(string filename)
{
    ifstream in_from_file(filename.c_str(),std::ios::in);

    if (!in_from_file.is_open())
    {
        if (rank_ == 0) std::cerr << "Mesh File " << filename << " Could Not Be Opened!" << std::endl;
        MPI_Finalize();
        exit(1);
    }

    std::string parser = "";
    in_from_file >> parser;

    if (parser != "$MeshFormat" )
    {
        if (rank_ == 0) std::cerr << "Invalid Mesh Format/File" << std::endl;
        MPI_Finalize();
        exit(1);
    }

    double dummy;
    in_from_file >> dummy >> dummy >> dummy;
    in_from_file >> parser >> parser;       //skip $EndMeshFormat and $Nodes

    //Vertex Read
    int n_vertices_in_file;
    in_from_file >> n_vertices_in_file;
    int local_vert_start;
    int local_vert_stop;
    int local_vert_count;
    parallelRange(0, n_vertices_in_file - 1, rank_, nproc_, local_vert_start, local_vert_stop, local_vert_count);


    vertices_.resize(0);
    vertex_t vertex;


    for (int ivert = 0; ivert < n_vertices_in_file; ivert++)
    {
        if (ivert >= local_vert_start && ivert <= local_vert_stop)
        {
            in_from_file >> vertex.mid >> vertex.r[0] >> vertex.r[1] >> vertex.r[2];
            vertices_.push_back(vertex);
        }
        else    //skip over the line
        {
            int dummy;
            in_from_file >> dummy;
            in_from_file.ignore(1000,'\n');
        }
    }

    //3D Tetrahedral Element Read

    elements_3d_.resize(0);

    in_from_file >> parser >> parser;   //skip $EndNodes and $Elements

    if (parser != "$Elements")
    {
        std::cerr << "Something has gone very wrong" << std::endl;
        assert(0==1);
    }

    int n_elements_in_file = 0;
    in_from_file >> n_elements_in_file;

    int local_ele_start;
    int local_ele_stop;
    int local_ele_count;
    parallelRange(0, n_elements_in_file - 1, rank_, nproc_, local_ele_start, local_ele_stop, local_ele_count);
```

```

    int element_num_tags;
    element_t element;

    int n_global_3d_elements = 0;

    for (int i_ele = 0; i_ele < n_elements_in_file; i_ele++)
    {
        int ele_in_range = (i_ele >= local_ele_start && i_ele <= local_ele_stop);

        in_from_file >> element.mid >> element.type >> element_num_tags >> dummy >> dummy;
        for (int itag = 0; itag < element_num_tags - 2; itag++)
        {
            in_from_file >> dummy;
        }

        //we are going to skip over anything that isn't a tetrahedral element so we just peel off the vertex mids
        if (element.type == FV_MESH_GMESH_ELEMENT_POINT)
        {
            in_from_file >> dummy;
        }
        else if (element.type == FV_MESH_GMESH_ELEMENT_FIRST_ORDER_LINE)
        {
            in_from_file >> dummy >> dummy;
        }
        else if (element.type == FV_MESH_GMESH_ELEMENT_FIRST_ORDER_TRIANGLE)
        {
            in_from_file >> dummy >> dummy >> dummy;
        }
        else if (element.type == FV_MESH_GMESH_ELEMENT_FIRST_ORDER_QUADRANGLE)
        {
            in_from_file >> dummy >> dummy >> dummy >> dummy;
        }
        else if (element.type == FV_MESH_GMESH_ELEMENT_FIRST_ORDER_TETRAHEDRAL)
        {
            in_from_file >> element.vertex_mids[0];
            in_from_file >> element.vertex_mids[1];
            in_from_file >> element.vertex_mids[2];
            in_from_file >> element.vertex_mids[3];
            element.nvert = 4;
            if (ele_in_range) elements_3d_.push_back(element);
        }
        else if (element.type == FV_MESH_GMESH_ELEMENT_FIRST_ORDER_HEXAHEDRAL)
        {
            in_from_file >> dummy >> dummy >> dummy >> dummy >> dummy >> dummy >> dummy >> dummy;
        }
        else
        {
            std::cerr << "Hit an unsupported element type" << std::endl;
            assert(0==1);
        }
    }//for each element
    in_from_file.close();
}


//-------------------------------------------------------------------------
// Construct the list of element centroids.
//-------------------------------------------------------------------------

void Mesh::createElementCentroidsList()
{
    element_3d_centroids_.resize(elements_3d_.size());
    const vertex_t* vertex_pointer;

    for (unsigned int iele = 0; iele < elements_3d_.size(); iele++)
    {
        element_3d_centroids_[iele].r[0] = 0.0;
        element_3d_centroids_[iele].r[1] = 0.0;
        element_3d_centroids_[iele].r[2] = 0.0;

        //The centroid of a simplex is just the average of the vertex coordinates
        for (unsigned int ivert = 0; ivert < elements_3d_[iele].nvert; ivert++)
        {
            vertex_pointer = getVertexFromMID(elements_3d_[iele].vertex_mids[ivert]);
            element_3d_centroids_[iele].r[0] += vertex_pointer->r[0];
            element_3d_centroids_[iele].r[1] += vertex_pointer->r[1];
            element_3d_centroids_[iele].r[2] += vertex_pointer->r[2];
        }

        element_3d_centroids_[iele].r[0] /= (double)elements_3d_[iele].nvert;
        element_3d_centroids_[iele].r[1] /= (double)elements_3d_[iele].nvert;
        element_3d_centroids_[iele].r[2] /= (double)elements_3d_[iele].nvert;
    }

    //Sanity check - the average of the centroids should be the "center" of the mesh.

    double3_t centroid_sum;
    centroid_sum.r[0] = 0.0;
    centroid_sum.r[1] = 0.0;
    centroid_sum.r[2] = 0.0;

    for (unsigned int iele = 0; iele < element_3d_centroids_.size(); iele++)
    {
        centroid_sum.r[0] += element_3d_centroids_[iele].r[0];
        centroid_sum.r[1] += element_3d_centroids_[iele].r[1];
        centroid_sum.r[2] += element_3d_centroids_[iele].r[2];
    }


    double3_t total_centroid_sum;
    MPI_Reduce(&centroid_sum.r[0], &total_centroid_sum.r[0], 3, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    int centroid_count = element_3d_centroids_.size();
    int total_centroid_count;

    MPI_Reduce(&centroid_count, &total_centroid_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
    if (rank_ == 0)
    {
        double3_t average_centroid = total_centroid_sum;
        average_centroid.r[0] /= (double)total_centroid_count;
        average_centroid.r[1] /= (double)total_centroid_count;
        average_centroid.r[2] /= (double)total_centroid_count;

        //std::cout << "The average centroid across all processors is (" << average_centroid.r[0] <<  ", " << average_centroid.r[1] << ", " << average_centroid.r[2] << ")" << std::endl;
    }
}


//-------------------------------------------------------------------------
// Obtain the list of unique vertices that completes our elements on
// every processor.
//-------------------------------------------------------------------------
void Mesh::getElementVertices()
{
    if (nproc_ == 1) return;

    //determine the vertices that we need to complete our local elements
    std::vector<int> vertex_mid_requests(0);
    for (unsigned int iele = 0; iele < elements_3d_.size(); iele++)
    {
        for (unsigned int ivert = 0; ivert < elements_3d_[iele].nvert; ivert++)
        {
            vertex_mid_requests.push_back(elements_3d_[iele].vertex_mids[ivert]);
        }
    }

    //make the list unique
    sort(vertex_mid_requests.begin(), vertex_mid_requests.end());
    vertex_mid_requests.erase(unique(vertex_mid_requests.begin(), vertex_mid_requests.end()), vertex_mid_requests.end());

    //now we know all the vertices we need to actually complete our elements

    //should already be sorted based on sequential read from mesh - but just to be safe
    sort(vertices_.begin(), vertices_.end(),sortVertexByMIDPredicate);


    //having collected the vertex set, we can now obtain the vertices we require
    //we send the vertices we need to every process (we could do better here)

    std::vector<std::vector<int> > outgoing_vertex_requests(0);
    for (unsigned int iproc = 0; iproc < nproc_; iproc++) outgoing_vertex_requests.push_back(vertex_mid_requests);

    std::vector<std::vector<int> > incoming_vertex_requests(0);
    MPI_Alltoall_vecvecT(outgoing_vertex_requests, incoming_vertex_requests);

    std::vector<std::vector<vertex_t> > outgoing_vertices(nproc_);
    vertex_t search_vertex;
    for (unsigned int iproc = 0; iproc < nproc_; iproc++)
    {
        outgoing_vertices[iproc].resize(0);
        for (unsigned int ivert = 0; ivert < incoming_vertex_requests[iproc].size(); ivert++)
        {
            int vertex_mid = incoming_vertex_requests[iproc][ivert];
            search_vertex.mid = vertex_mid;
            void* result = bsearch(&search_vertex, &vertices_[0], vertices_.size(), sizeof(vertex_t), searchVertexByMIDPredicate);
            if (result != NULL) // found vertex
            {
                const vertex_t* found_vertex = (const vertex_t*)result;
                search_vertex.r[0] = found_vertex->r[0];
                search_vertex.r[1] = found_vertex->r[1];
                search_vertex.r[2] = found_vertex->r[2];
                outgoing_vertices[iproc].push_back(search_vertex);
            }
        }
    }

    std::vector<std::vector<vertex_t> > incoming_vertices;
    MPI_Alltoall_vecvecT(outgoing_vertices, incoming_vertices);

    vertices_.resize(0);
    std::map<int,int> vertex_map;
    for (unsigned int iproc = 0; iproc < nproc_; iproc++)
    {
        for (unsigned int ivert = 0; ivert < incoming_vertices[iproc].size(); ivert++)
        {
            if (vertex_map.find(incoming_vertices[iproc][ivert].mid) == vertex_map.end())
            {
                vertices_.push_back(incoming_vertices[iproc][ivert]);
                vertex_map[incoming_vertices[iproc][ivert].mid] = 1;
            }
        }
    }

    sort(vertices_.begin(), vertices_.end(), sortVertexByMIDPredicate);

    //now we should be able to find the vertices for any element we have locally.
    //sanity check.
    for (unsigned int iele = 0; iele < elements_3d_.size(); iele++)
    {
        for (unsigned int ivert = 0; ivert < elements_3d_[iele].nvert; ivert++)
        {
            const vertex_t* vertex = getVertexFromMID(elements_3d_[iele].vertex_mids[ivert]);
            assert(vertex != NULL);
        }
    }
}

// This routine will run ORB on all of the mesh's element centroids
// to determine a block partition of the elements. Then, this function
// will swap global element.mids so that each processor owns one of
// the ORB partitions. Then, each processor will call a routine to
```

```cpp
// get all the vertices associated with that list of elements.
void Mesh::partitionMesh()
{
    if(nproc_ == 1) return; // Partition is already complete.

    // Perform ORB on element centroids
    std::vector<std::vector<int> > local_idcs_per_dom(nproc_);
    ORB(
        nproc_,
        element_3d_centroids_,
        local_idcs_per_dom
    );
    // Now you know which of your elments to send to the other procs,
    // by your local index. It would be more helpful to just give the
    // whole element to the other processor. Do that.
    std::vector<std::vector<element_t> > global_elements_per_dom(nproc_);
    for(int idom = 0;idom < local_idcs_per_dom.size();idom++)
    {
        global_elements_per_dom[idom].resize(local_idcs_per_dom[idom].size());
        for(int iloc = 0; iloc< local_idcs_per_dom[idom].size();iloc++)
        {

            global_elements_per_dom[idom][iloc] = elements_3d_[local_idcs_per_dom[idom][iloc]];
        }
    }
    // Do the big group swap of global elements.
    std::vector<std::vector<element_t> > recv_elements_per_dom(nproc_);
    MPI_Alltoall_vecvecT(global_elements_per_dom,recv_elements_per_dom);
    // Rewrite my elements_3d_ with what I just received.
    int new_ele_3d_count = 0;
    for(int iproc = 0;iproc<nproc_;iproc++) new_ele_3d_count += recv_elements_per_dom[iproc].size();
    elements_3d_.resize(new_ele_3d_count);
    int ele_3d_idx = 0;
    for(int iproc = 0;iproc<nproc_;iproc++)
    {
        for(int ipt = 0; ipt < recv_elements_per_dom[iproc].size();ipt++)
        {
            elements_3d_[ele_3d_idx++] = recv_elements_per_dom[iproc][ipt];
        }
    }
    // Get the vertices and centroids associated with my elements.
    getElementVertices();
    createElementCentroidsList();
}


//--------------------------------------------------------------------------
// Write unstructured mesh to Paraview XML format. This will create P+1 files
// on P processors. The .vtu files are pieces of the mesh. The .pvtu file is a
// single wrapper file that can be loaded in paraview such that every .vtu file with
// the corresponding names will be opened simultaneously.
//
// Inputs:
//
// The filename should be a complete path with NO extension (.vtu and .pvtu
// will be added.
//
// Value label is a string that will be written to the vtu file labeling the values
// that you are writing for each element (e.g. "rank").
//
// Values is a vector with length elements_3d_.size() corresponding to a single
// scalar value to be associated with each element in the mesh.
//--------------------------------------------------------------------------
void Mesh::writeMesh(string filename, std::string value_label, const vector<double>& values) const
{
    ofstream vtu_out, pvtu_out;

    std::ostringstream converter;
    converter << filename << "_P" << nproc_ << "_R" << rank_;
    std::string vtu_filename = converter.str() + ".vtu";

    //------------------------------------------------------------------------------
    // Open the VTU file (All ranks)
    //------------------------------------------------------------------------------

    vtu_out.open(vtu_filename.c_str());
    if (!vtu_out.is_open())
    {
        std::cerr << "Could not open vtu file" << std::endl;
        assert(0==1);
    }

    vtu_out << "<?xml version=\"1.0\"?>" << std::endl;
    vtu_out << "<VTKFile type=\"UnstructuredGrid\" version=\"0.1\" byte_order=\"LittleEndian\">" << std::endl;

    //------------------------------------------------------------------------------
    // Open the PVTU file (Rank 0)
    //------------------------------------------------------------------------------

    if (rank_==0)
    {
        std::ostringstream pconverter;
        pconverter << filename << "_P" << nproc_;
        std::string pvtu_filename = pconverter.str() + ".pvtu";
        pvtu_out.open(pvtu_filename.c_str());
        if (!pvtu_out.is_open())
        {
            std::cerr << "Could not open pvtu file" << std::endl;
            assert(0==1);
        }

        pvtu_out << "<?xml version=\"1.0\"?>" << std::endl;
        pvtu_out << "<VTKFile type=\"PUnstructuredGrid\" version=\"0.1\" byte_order=\"LittleEndian\">" << std::endl;
    }

    //------------------------------------------------------------------------------
    // Write the 3D Mesh Elements to File
```

```
    //-----------------------------------------------------------------------------

    int n_elements = elements_3d_.size();

    //-----------------------------------------------------------------------------
    // VTU Mesh
    //-----------------------------------------------------------------------------

    //Preamble
    vtu_out << "<UnstructuredGrid>" << std::endl;
    vtu_out << "<Piece NumberOfPoints=\"" << vertices_.size() << "\" NumberOfCells=\"" << n_elements << "\">" << std::endl;

    //Vertices
    vtu_out << "<Points>" << std::endl;
    vtu_out << "<DataArray type=\"Float32\" NumberOfComponents=\"3\" format=\"ascii\">" << std::endl;
    for (unsigned int ivert = 0; ivert < (int)vertices_.size(); ivert++)
    {
        vtu_out << vertices_[ivert].r[0] << " " << vertices_[ivert].r[1] << " " << vertices_[ivert].r[2] << " ";
    }
    vtu_out << std::endl;
    vtu_out << "</DataArray>" << std::endl;
    vtu_out << "</Points>" << std::endl;

    vtu_out << "<Cells>" << std::endl;

    //Element Connectivity
    vtu_out << "<DataArray type=\"Int32\" Name=\"connectivity\">" << std::endl;
    for (unsigned int iele = 0; iele < (int)elements_3d_.size(); iele++)
    {
        for (unsigned int ivert = 0; ivert < elements_3d_[iele].nvert; ivert++)
        {
            const vertex_t* vertex = getVertexFromMID(elements_3d_[iele].vertex_mids[ivert]);
            assert(vertex != NULL);
            int vertex_lid = (vertex - &vertices_[0]);
            assert(vertices_[vertex_lid].mid == elements_3d_[iele].vertex_mids[ivert]);
            assert(vertex_lid >= 0 && vertex_lid < vertices_.size());
            vtu_out << vertex_lid << " ";
        }
    }
    vtu_out << std::endl;
    vtu_out << "</DataArray>" << std::endl;

    //Offsets
    vtu_out << "<DataArray type=\"Int32\" Name=\"offsets\">" << std::endl;
    int vert_sum = 0;
    for (unsigned int iele = 0; iele < elements_3d_.size(); iele++)
    {
        vert_sum += elements_3d_[iele].nvert;
        vtu_out << vert_sum << " ";
    }
    vtu_out << std::endl;
    vtu_out << "</DataArray>" << std::endl;

    //Types
    vtu_out << "<DataArray type=\"UInt8\" Name=\"types\">" << std::endl;
    for (unsigned int iele = 0; iele < (int)elements_3d_.size(); iele++)
    {
        vtu_out << "10 ";
    }
    vtu_out << std::endl;

    vtu_out << "</DataArray>" << std::endl;
    vtu_out << "</Cells>" << std::endl;


    //-----------------------------------------------------------------------------
    // PVTU Mesh
    //-----------------------------------------------------------------------------

    if (rank_ == 0)
    {
        pvtu_out << "<PUnstructuredGrid GhostLevel=\"0\">" << std::endl;

        pvtu_out << "<PPoints>" << std::endl;
        pvtu_out << "<PDataArray type=\"Float32\" NumberOfComponents=\"3\" format=\"ascii\">" << std::endl;

        pvtu_out << "</PDataArray>" << std::endl;
        pvtu_out << "</PPoints>" << std::endl;

        pvtu_out << "<PCells>" << std::endl;

        //Connectivity
        pvtu_out << "<PDataArray type=\"Int32\" Name=\"connectivity\">" << std::endl;
        pvtu_out << "</PDataArray>" << std::endl;

        //Offsets
        pvtu_out << "<PDataArray type=\"Int32\" Name=\"offsets\">" << std::endl;
        pvtu_out << "</PDataArray>" << std::endl;

        //Types
        pvtu_out << "<PDataArray type=\"UInt8\" Name=\"types\">" << std::endl;
        pvtu_out << "</PDataArray>" << std::endl;
        pvtu_out << "</PCells>" << std::endl;
    }

    //-----------------------------------------------------------------------------
    // VTU Cell Data Open
    //-----------------------------------------------------------------------------

    vtu_out << "<CellData>" << std::endl;

    vtu_out << "<DataArray type=\"Float32\" format=\"ascii\" Name=\"" << value_label << "\">" << std::endl;
    assert(values.size() == elements_3d_.size());
    for (int iele = 0; iele < (int)elements_3d_.size(); iele++)
    {
        vtu_out << values[iele] << " ";
    }
```

```
    vtu_out << "</DataArray>" << std::endl;
    vtu_out << "</CellData>" << std::endl;

    //-----------------------------------------------------------------------------
    // PVTU Cell Data Open
    //-----------------------------------------------------------------------------

    if (rank_ == 0)
    {
        pvtu_out << "<PCellData>" << std::endl;
        pvtu_out << "<PDataArray type=\"Float32\" format=\"ascii\" Name=\"" << value_label << "\">" << std::endl;
        pvtu_out << "</PDataArray>" << std::endl;
        pvtu_out << "</PCellData>" << std::endl;
    }


    //-----------------------------------------------------------------------------
    // VTU Close
    //-----------------------------------------------------------------------------

    vtu_out << "</Piece>" << std::endl;
    vtu_out << "</UnstructuredGrid>" << std::endl;
    vtu_out << "</VTKFile>" << std::endl;
    vtu_out.close();

    //-----------------------------------------------------------------------------
    // PVTU Close
    //-----------------------------------------------------------------------------

    if (rank_ == 0)
    {
        for (int iproc = 0; iproc < nproc_; iproc++)
        {
            std::ostringstream vtu_converter;
            //vtu_converter << vtkfilename << "_Run" << iRun << "_N" << num_proc_ <<"_P" << iproc << ".vtu";
            //We always assume the pvtu file exists in the same directory as the other files so here vtkfilename must only be the relative name
            vtu_converter << filename << "_P" << nproc_ << "_R" << iproc << ".vtu";
            pvtu_out << "<Piece Source=\"" << vtu_converter.str() << "\"/>" << std::endl;
        }

        pvtu_out << "</PUnstructuredGrid>" << std::endl;
        pvtu_out << "</VTKFile>" << std::endl;
        pvtu_out.close();
    }
}

//------------------------------------
// Output some statistics including
// number of elements/vertices on each
// processor and global number of
// elements/vertices. Nicely formatted.
//------------------------------------
void Mesh::outputStatistics() const
{
    MPI_Barrier(MPI_COMM_WORLD);
    int local_n_eles = elements_3d_.size();
    int global_n_eles = 0;
    int local_n_vert = vertices_.size();
    int global_n_vert = 0;
    // Figure out number of global elements and vertices
    MPI_Reduce(
        &local_n_eles,
        &global_n_eles,
        1,
        MPI_INT,
        MPI_SUM,
        0,
        MPI_COMM_WORLD
    );

    MPI_Reduce(
        &local_n_vert,
        &global_n_vert,
        1,
        MPI_INT,
        MPI_SUM,
        0,
        MPI_COMM_WORLD
    );

    if(rank_ == 0)
    {
        std::cout << "\nGlobal Mesh Statistics:" << std::endl;
        std::cout << "\tGlobal Element Count: " << global_n_eles << std::endl;
        std::cout << "\tGlobal Vertex  Count: " << global_n_vert << "\n" <<std::endl;
        std::cout << "Element/Vertex Counts By Processor:\n" << std::endl;
        std::cout << "|\tProc\t";
        std::cout << "|\tEles\t";
        std::cout << "|\tVrts\t";
        std::cout << "|\tX Min\t";
        std::cout << "|\tX Max\t";
        std::cout << "|\tY Min\t";
        std::cout << "|\tY Max\t";
        std::cout << "|\tZ Min\t";
        std::cout << "|\tZ Max\t";
        std::cout << "| " << std::endl;
        std::cout << "================"
        "================"
        "================"
        "================"
        "================"
        "================"
        "================"
        "================"
        "================"
        "="<<std::endl;
    }
```

```

    // Calculate your local extent
    std::vector<std::vector<double> > extents(3,std::vector<double>(2,1));
    extents[0][0] = DBL_MAX;
    extents[1][0] = DBL_MAX;
    extents[2][0] = DBL_MAX;
    extents[0][1] = DBL_MIN;
    extents[1][1] = DBL_MIN;
    extents[2][1] = DBL_MIN;
    for(int ivert = 0; ivert < vertices_.size(); ivert++)
    {
        for (int idim = 0;idim<3;idim++)
        {
            if(vertices_[ivert].r[idim]<extents[idim][0]) extents[idim][0] = vertices_[ivert].r[idim];
            if(vertices_[ivert].r[idim]>extents[idim][1]) extents[idim][1] = vertices_[ivert].r[idim];
        }
    }

    for(int irank = 0;irank < nproc_; irank++)
    {
        MPI_Barrier(MPI_COMM_WORLD);
        if(irank == rank_)
        {
            std::cout << "|\t" << rank_ << "\t";
            std::cout << "|\t" << elements_3d_.size() << "\t";
            std::cout << "|\t" <<vertices_.size() << "\t";
            std::cout << "|" << std::setw(15) << extents[0][0];
            std::cout << "|" << std::setw(15) << extents[0][1];
            std::cout << "|" << std::setw(15) << extents[1][0];
            std::cout << "|" << std::setw(15) << extents[1][1];
            std::cout << "|" << std::setw(15) << extents[2][0];
            std::cout << "|" << std::setw(15) << extents[2][1];
            std::cout << "|" << std::endl;
            std::cout << "----------------"
            "----------------"
            "----------------"
            "----------------"
            "----------------"
            "----------------"
            "----------------"
            "----------------"
            "----------------"
            ""<<std::endl;
        }
    }
}

void Mesh::calculateVertexConnectivity()
{
    // Run through all of my elements and see which vertex mids are
    // connected to each other. This makes an mid -> mid list

    int rank,nproc;
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    std::stringstream msg;

    for(int iv = 0; iv < vertices_.size(); iv++)
    {
        vertices_[iv].neighbours.resize(0);
    }

    std::map<int, std::vector<int> > mid_connections_by_mid;

    for(int ie = 0; ie < elements_3d_.size(); ie++)
    {
        element_t * cur_ele = &(elements_3d_[ie]);
        for(int iv = 0; iv < (cur_ele->nvert-1); iv++)
        {
            int midi = cur_ele->vertex_mids[iv];
            for(int jv = iv+1; jv < (cur_ele->nvert); jv++)
            {
                int midj = cur_ele->vertex_mids[jv];
                mid_connections_by_mid[midi].push_back(midj);
                mid_connections_by_mid[midj].push_back(midi);
            }
        }
    }

    // You probably double-counted a bunch. Make sure each list
    // contains no copies.
    int my_highest_mid = 0;
    for(int iv = 0; iv < vertices_.size(); iv++)
    {
        int midi = vertices_[iv].mid;
        mid2lindx_[midi] = iv;
        if(midi > my_highest_mid) my_highest_mid = midi;
        sort(mid_connections_by_mid[midi].begin(), mid_connections_by_mid[midi].end());
        mid_connections_by_mid[midi].erase(unique(mid_connections_by_mid[midi].begin(), mid_connections_by_mid[midi].end()), mid_connections_by_mid[midi].end());
    }

    int global_highest_mid = my_highest_mid;

    MPI_Allreduce(
        &my_highest_mid,
        &global_highest_mid,
        1,
        MPI_INT,
        MPI_MAX,
        MPI_COMM_WORLD
    );

    // Determine ownership and families
    for(int midi = 0; midi <= global_highest_mid; midi++)
    {
```

```
        // Do I have a vertex with this mid?
        std::map<int,int>::iterator it;
        it = mid2lindx_.find(midi);

        bool vertex_exists_locally = (it != mid2lindx_.end());

        std::vector<int> send_existence(nproc, vertex_exists_locally);
        std::vector<int> recv_existence(nproc);
        // Tell everyone else whether I have this mid locally.
        // Find out who else has this mid locally.
        MPI_Alltoall(
            &(send_existence[0]),
            1,
            MPI_INT,
            &(recv_existence[0]),
            1,
            MPI_INT,
            MPI_COMM_WORLD
        );

        int vertex_owner = nproc;
        // Pick the lowest rank with a local copy of this mid as then
        // owner.
        for(int iproc = 0; iproc < nproc; iproc++)
        {
            if(recv_existence[iproc])
            {
                vertex_owner = iproc;
                break;
            }
        }

        if(vertex_exists_locally)
        {
            int lindx = mid2lindx_[midi];
            vertex_t * cur_v = &(vertices_[lindx]);
            cur_v->owner = vertex_owner;
            // Fill the family for this vertex
            cur_v->family.resize(0);
            for(int iproc = 0; iproc < nproc; iproc++)
            {
                if(recv_existence[iproc]) cur_v->family.push_back(iproc);
            }
        }
        if(vertex_owner < nproc)
        {
            // If this mid was actually claimed by someone, then
            // make sure the owner knows all of the global mids that
            // are connected to this mid.
            std::vector<std::vector<int> > send_mid_connections(nproc);
            std::vector<std::vector<int> > recv_mid_connections(nproc);
            if(vertex_exists_locally)
            {
                // Tell the owner about all the connections I am aware
                // of.
                send_mid_connections[vertex_owner] = mid_connections_by_mid[midi];
            }
            MPI_Alltoall_vecvecT(send_mid_connections,recv_mid_connections);
            if(vertex_owner == rank)
            {
                for(int iproc = 0; iproc < nproc; iproc++)
                {
                    for(int icon = 0; icon < recv_mid_connections[iproc].size();icon++)
                    {
                        // Add this connection to my list of global connections.
                        vertices_[mid2lindx_[midi]].global_nbr_mids.push_back(recv_mid_connections[iproc][icon]);
                    }
                }
                // Delete any copies from the list of global connections.
                sort(vertices_[mid2lindx_[midi]].global_nbr_mids.begin(), vertices_[mid2lindx_[midi]].global_nbr_mids.end());
                vertices_[mid2lindx_[midi]].global_nbr_mids.erase(unique(vertices_[mid2lindx_[midi]].global_nbr_mids.begin(), vertices_[mid2lindx_[
      midi]].global_nbr_mids.end()), vertices_[mid2lindx_[midi]].global_nbr_mids.end());
                vertices_[mid2lindx_[midi]].nbr_was_counted.resize(vertices_[mid2lindx_[midi]].global_nbr_mids.size());
            }
        }
    }
    // Fill neighbour pointers
    for(int iv = 0; iv < vertices_.size(); iv++)
    {
        vertex_t * cur_v = &(vertices_[iv]);
        int midi = cur_v->mid;
        cur_v->neighbours.resize(0);
        for(int jj = 0; jj < mid_connections_by_mid[midi].size();jj++)
        {
            int midj = mid_connections_by_mid[midi][jj];
            vertex_t * nbr_v = &(vertices_[mid2lindx_[midj]]);
            cur_v->neighbours.push_back(nbr_v);
        }
    }

}




void Mesh::populateMeshVertices(double alive_probability)
{
    int rank,nproc;
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

    if(alive_probability > 1) alive_probability = 1;
    if(alive_probability < 0) alive_probability = 0;
    int n_vert = vertices_.size();
    int n_alive_verts = 0;
    unsigned int alive_checker = (unsigned int)(((double)RAND_MAX)*alive_probability);
```

```
    bool initial_state;
    for(int ivert = 0; ivert<n_vert;ivert++)
    {
        if(vertices_[ivert].owner == rank)
        {
            initial_state = (rand() < alive_checker);
            //initial_state = (vertices_[ivert].mid%2);
            //initial_state = (vertices_[ivert].r[0] > 0);
            for(int ifam = 0; ifam < vertices_[ivert].family.size();ifam++)
            {
                if(vertices_[ivert].family[ifam] != rank)
                {
                    MPI_Send(
                        &(initial_state),
                        1,
                        MPI_LOGICAL,
                        vertices_[ivert].family[ifam],
                        vertices_[ivert].mid,
                        MPI_COMM_WORLD
                    );
                }
            }
        }
        else
        {
            // Receive initial state from owner.
            MPI_Recv(
                &initial_state,
                1,
                MPI_LOGICAL,
                vertices_[ivert].owner,
                vertices_[ivert].mid,
                MPI_COMM_WORLD,
                &status
            );
        }
        vertices_[ivert].current_state = initial_state;
        if(initial_state)n_alive_verts++;
    }
}

bool GOL_CalculateNextState(bool current_state,int n_alive, int n_dead)
{
    double life_rate = (double)(n_alive + n_dead);
    life_rate = 1/life_rate;
    life_rate *= (double)(n_alive);
    bool next_state = current_state;
    if(current_state)
    {
        if(life_rate < 0.2999)
        {
            // Die of loneliness
            next_state = false;
        }
        else if(life_rate < 0.5111)
        {
            // Keep living because you're in a good community.
            next_state = true;
        }
        else
        {
            // Die from overpopulation
            next_state = false;
        }
    }
    else
    {
        if((0.2999 <life_rate) && (life_rate < 0.5111))
        {
            // Get born
            next_state = true;
        }
        else
        {
            // Stay dead
            next_state = false;
        }
    }
    return(next_state);
}

void Mesh::updateVertexStates()
{
    int rank,nproc;
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    int n_vert = vertices_.size();
    int n_come_alive = 0;
    int n_come_dead = 0;
    int total_n_alive = 0;
    std::stringstream msg;

    std::vector<std::vector<helper_vertex_t> > send_helpers(nproc);
    std::vector<std::vector<helper_vertex_t> > recv_helpers(nproc);

    // Loop over my vertices:
    //  For each vertex where I'm not the owner, send a collection of
    //  helper vertices.
    int n_helpers_sent = 0;
    for(int iv = 0; iv < n_vert; iv++)
    {
        vertex_t * cur_v = &(vertices_[iv]);
        if((cur_v->owner != rank) &&(cur_v->family.size()>0))
        {
            for(int inb = 0; inb < cur_v->neighbours.size(); inb++)
            {
```

```
                vertex_t * nbr_v = cur_v->neighbours[inb];
                helper_vertex_t helper;
                helper.host_mid = cur_v->mid;
                helper.nbr_mid = nbr_v->mid;
                helper.nbr_state = nbr_v->current_state;
                send_helpers[cur_v->owner].push_back(helper);
                n_helpers_sent++;
            }
        }
    }
    MPI_Alltoall_vecvecT(send_helpers,recv_helpers);

    // For each of my vertices:
    // If you don't own this vertex, don't bother. You sent its
    // neighbours to its owner. You'll get its next state later on.
    // If you own it, then:
    //  For each neighbour:
    //      add its state to alive_dead
    //      check off that that connection was counted
    for(int iv = 0; iv < vertices_.size(); iv++)
    {
        vertex_t * cur_v = &(vertices_[iv]);
        vertices_[iv].alive_dead[0] = 0;
        vertices_[iv].alive_dead[1] = 0;
        if(cur_v->owner != rank) continue;
        for(int inb = 0; inb < cur_v->neighbours.size(); inb++)
        {
            vertex_t * nbr_v = cur_v->neighbours[inb];
            // Has this connection been counted already?
            int gm_idx=-1;
            for(int igm = 0; igm < cur_v->global_nbr_mids.size();igm++)
            {
                if(nbr_v->mid == cur_v->global_nbr_mids[igm])
                {
                    gm_idx = igm;
                    break;
                }
            }
            bool connection_already_counted = cur_v->nbr_was_counted[gm_idx];
            if(!connection_already_counted)
            {
                if(nbr_v->current_state)
                {
                    cur_v->alive_dead[0]++;
                }
                else
                {
                    cur_v->alive_dead[1]++;
                }
            }
            cur_v->nbr_was_counted[gm_idx] = true;

        }
    }

    // For each helper_vertex I received:
    //  I received it because I AM THE OWNER
    //  Find its host's index in my list using mid2lindx_
    //  Update that host's alive_dead using the helper's state.
    for(int iproc = 0; iproc < nproc; iproc++)
    {
        for(int ih = 0; ih < recv_helpers[iproc].size();ih++)
        {
            helper_vertex_t * helper = &(recv_helpers[iproc][ih]);
            // This helper is telling me that the host at host_mid
            // needs to know that its neighbour at nbr_mid has state
            // nbr_state.
            // Do I already know about this connection?
            vertex_t * host_v = &(vertices_[mid2lindx_[helper->host_mid]]);
            bool connection_was_counted = false;
            int gm_idx=-1;
            for(int igm = 0; igm < host_v->global_nbr_mids.size();igm++)
            {
                if(helper->nbr_mid == host_v->global_nbr_mids[igm])
                {
                    gm_idx = igm;
                    connection_was_counted = host_v->nbr_was_counted[igm];
                    break;
                }
            }
            if(!connection_was_counted)
            {
                if(helper->nbr_state)
                {
                    host_v->alive_dead[0]++;
                }
                else
                {
                    host_v->alive_dead[1]++;
                }
                host_v->nbr_was_counted[gm_idx] = true;
            }
        }
    }

    // For each of my vertices
    // If you own this vertex, update its state and communicate the
    // new state.
    for(int iv = 0; iv < vertices_.size(); iv++)
    {
        vertex_t * cur_v = &(vertices_[iv]);
        if(cur_v-> owner == rank)
        {
            cur_v->next_state = GOL_CalculateNextState(
                cur_v->current_state,
                cur_v->alive_dead[0],
                cur_v->alive_dead[1]
```

```
            );
            for(int ifam = 0; ifam < cur_v->family.size();ifam++)
            {
                // No need to talk to yourself, you already know the
                // next state
                if(cur_v->family[ifam] == rank) continue;
                // I am the owner. Only owners get to send out the
                // next_state of a vertex. Therefore, nobody else
                // will be trying to send this message.
                MPI_Send(
                    &(cur_v->next_state),
                    1,
                    MPI_LOGICAL,
                    cur_v->family[ifam],
                    cur_v->mid,
                    MPI_COMM_WORLD
                );
            }
        }
    }
    // For each of my vertices
    // If someone else owns it, then there should be a message waiting
    // for me telling me its new state.
    for(int iv = 0; iv < vertices_.size(); iv++)
    {
        vertex_t * cur_v = &(vertices_[iv]);
        if(cur_v->owner != rank)
        {
            MPI_Recv(
                &(cur_v->next_state),
                1,
                MPI_LOGICAL,
                cur_v->owner,
                cur_v->mid,
                MPI_COMM_WORLD,
                &status
            );
        }
    }
    // Finally, loop through all your vertices and make current state
    // become next state.
    for(int iv = 0; iv < vertices_.size(); iv++)
    {
        vertex_t * cur_v = &(vertices_[iv]);
        cur_v->current_state = cur_v->next_state;
        // Zero out alive_dead, just to be safe.
        cur_v->alive_dead[0] = 0;
        cur_v->alive_dead[1] = 0;
        for(int ig = 0; ig < cur_v->global_nbr_mids.size();ig++)
        {
            cur_v->nbr_was_counted[ig] = false;
        }
    }
}




void Mesh::writeCellMatrixFile(
    std::string filename,
    bool append=false
)
{
    int rank,nproc;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    std::ofstream matfilestream;
    if(append)
    {
        for(int irank = 0; irank < nproc; irank ++)
        {
            if(irank == rank)
            {
                matfilestream.open(filename,std::ofstream::out | std::ofstream::app);
                if(rank == 0) matfilestream << std::endl;
                for(int iv = 0; iv < vertices_.size();iv++)
                {
                    if(vertices_[iv].owner == rank)
                    {
                        matfilestream << vertices_[iv].current_state << " ";
                    }
                }
                if(rank == (nproc-1)) matfilestream << std::endl;
                matfilestream.close();
            }
            MPI_Barrier(MPI_COMM_WORLD);
        }
    }
    else
    {
        for(int irank = 0; irank < nproc; irank ++)
        {
            if(irank == rank)
            {
                if(rank == 0)
                {
                    matfilestream.open(filename,std::ofstream::out);
                    // matfilestream << global_n_vertices << std::endl;
                }
                else
                {
                    matfilestream.open(filename,std::ofstream::out | std::ofstream::app);
                }
                for(int iv = 0; iv < vertices_.size();iv++)
                {
                    //If you are the owner, output the mid and position
                    if(vertices_[iv].owner == rank)
```

```
                    {
                        matfilestream << vertices_[iv].mid << " " << vertices_[iv].r[0] << " " << vertices_[iv].r[1] << " " << vertices_[iv].r[2] << std::endl;
                    }
                }
                matfilestream.close();
            }
            MPI_Barrier(MPI_COMM_WORLD);
        }
        for(int irank = 0; irank < nproc; irank ++)
        {
            if(irank == rank)
            {
                matfilestream.open(filename,std::ofstream::out | std::ofstream::app);
                if(rank == 0) matfilestream << std::endl;
                for(int iv = 0; iv < vertices_.size();iv++)
                {
                    if(vertices_[iv].owner == rank)
                    {
                        matfilestream << vertices_[iv].current_state << " ";
                    }
                }
                if(rank == (nproc-1)) matfilestream << std::endl;
                matfilestream.close();
            }
            MPI_Barrier(MPI_COMM_WORLD);
        }
    }
}

double Mesh::calculateVertexLifeRate()
{
    int rank,nproc;
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

    int total_owned_alive;
    int total_owned_dead;
    for(int iv = 0; iv < vertices_.size();iv++)
    {
        vertex_t * cur_v = &(vertices_[iv]);
        if(cur_v->owner == rank)
        {
            if(cur_v->current_state)
            {
                total_owned_alive++;
            }
            else
            {
                total_owned_dead++;
            }
        }
    }
    int global_n_alive;
    int global_n_dead;
    MPI_Allreduce(
        &total_owned_alive,
        &global_n_alive,
        1,
        MPI_INT,
        MPI_SUM,
        MPI_COMM_WORLD
    );
    MPI_Allreduce(
        &total_owned_dead,
        &global_n_dead,
        1,
        MPI_INT,
        MPI_SUM,
        MPI_COMM_WORLD
    );
    double life_rate = (double)(global_n_alive+global_n_dead);
    if((global_n_alive + global_n_dead)>0)
    {
        life_rate = 1/life_rate;
        life_rate *= (double)(global_n_alive);
    }
    else
    {
        life_rate = 0;
    }
    return(life_rate);
}
```