# TDT4200 - Task 2

Daniel Romanich

October 2018

## 1 Getting started with MPI

### 1.1 Parallel image rasterisation

Before doing this exercise I made the assumption that the parallel execution would scale very well up to the amount of cores on my processor (8). That is; the MPI-execution will have a pretty good scaling until it hits $\#ranks = \#cores$. This is the amount of ranks that can be run in actual parallel. If we have more ranks we won't be able to process them, which means we will see a slow-down in average completion time (the ranks that does not get to execute first will have a way slower run-time). When I ran 8 images in parallel I got about an 6x speedup from running 8 consecutively. My expectations were to get somewhere a little under 8x. I expected under 8x because I assumed that the object loading got cached, resulting in faster run-times after the first execution. As explained above, I have 8 cores, so running more ranks in parallel will slow the program down. To test this even further I assumed that running 16 ranks vs running 8 ranks would be about 2x slower, which was correct. The code below illustrates the changes made to the code.

```
MPI_Init(NULL, NULL);

int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

auto start = std::chrono::high_resolution_clock::now();

/**
 * Parameter loading here
**/

std::cout << "Loading '" << input << "' file... " << std::endl;
std::vector<Mesh> meshs = loadWavefront(input, false);
std::vector<unsigned char> frameBuffer = rasterise(meshs, width, height, depth);
std::cout << "Writing image to '" << output << "'..." << std::endl;
```

```
auto end = std::chrono::high_resolution_clock::now();
auto time = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
std::cout << "Time spent rendering the image for process" << rank <<
" took " << time << " ms" << std::endl;

MPI_Finalize();
```

## 1.2 MPI Communication

In this task I simply made node 0 the master, and made it send an angle to each of the remaining processes. Each angle is rank * 10. Meaning 36 processes gives a full rotation. The code below illustrates the master distributing angles. I also changed a bit of the code that calls rasterise (to take rotationAngle in as a parameter).

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);

float rotationAngle = 0;

/**
 * Master node is 0
 */
if (rank == 0) {
    for (int i = 1; i < size; i++) {
        float angle = i * 10;
        MPI_Send(&angle, 1, MPI_FLOAT, i, i, MPI_COMM_WORLD);
    }
} else {
    MPI_Recv(&rotationAngle, 1, MPI_FLOAT, 0, rank, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

## 1.3 Broadcasting vertices, textures & normals

I implemented this by first clearing the vertices/textures/normals of the non-master nodes. Then I created two new MPI_Datatypes for float3 & float4. To broadcast I made 3 new buffer-variables to be used in the broadcast (for vertices/textures/normals). Finally I made a for loop that iterated over the mesh and broadcast the values for all of the slave-nodes to copy. The code below is only the relevant snippets from the code.

```
/**
 * Clear out the provided vertices/normals/textures
 */
if (rank != 0) {
```

```
    for (unsigned int i = 0; i < meshs.size(); i++) {
        for (unsigned int j = 0; j < meshs.at(i).vertices.size(); j++) {
            meshs.at(i).vertices.at(j).w = 0;
            meshs.at(i).vertices.at(j).x = 0;
            meshs.at(i).vertices.at(j).y = 0;
            meshs.at(i).vertices.at(j).z = 0;
        }
        for (unsigned int j = 0; j < meshs.at(i).normals.size(); j++) {
            meshs.at(i).normals.at(j).x = 0;
            meshs.at(i).normals.at(j).y = 0;
            meshs.at(i).normals.at(j).z = 0;
        }
        for (unsigned int j = 0; j < meshs.at(i).textures.size(); j++) {
            meshs.at(i).textures.at(j).x = 0;
            meshs.at(i).textures.at(j).y = 0;
            meshs.at(i).textures.at(j).z = 0;
        }
    }
}

MPI_Datatype FLOAT_3 = generateFloat3();
MPI_Datatype FLOAT_4 = generateFloat4();

/**
 * Broadcast vertices/normals/textures
 */
for (unsigned int i = 0; i < meshs.size(); i++) {
    vertices = meshs.at(i).vertices;
    textures = meshs.at(i).textures;
    normals = meshs.at(i).normals;

    MPI_Bcast(&vertices.front(), vertices.size(), FLOAT_4, 0, MPI_COMM_WORLD);
    MPI_Bcast(&textures.front(), textures.size(), FLOAT_3, 0, MPI_COMM_WORLD);
    MPI_Bcast(&normals.front(), normals.size(), FLOAT_3, 0, MPI_COMM_WORLD);


    meshs.at(i).vertices = vertices;
    meshs.at(i).textures = textures;
    meshs.at(i).normals = normals;

}


/**
 * Method that generates and returns the FLOAT_4 MPI_Datatype
 */
```

```
MPI_Datatype generateFloat4() {
    int count = 4;
    MPI_Aint offset[4];
    offset[0] = offsetof(float4, w);
    offset[1] = offsetof(float4, x);
    offset[2] = offsetof(float4, y);
    offset[3] = offsetof(float4, z);
    int block_length[4] = {1, 1, 1, 1};
    MPI_Datatype types[4] = {MPI_FLOAT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT};
    MPI_Datatype FLOAT_4;
    MPI_Type_create_struct(count, block_length, offset, types, &FLOAT_4);
    MPI_Type_commit(&FLOAT_4);
    return FLOAT_4;
}

/**
 * Method that generates and returns the FLOAT_3 MPI_Datatype
 */
MPI_Datatype generateFloat3() {
    int count = 3;
    MPI_Aint offset[3];
    offset[0] = offsetof(float3, x);
    offset[1] = offsetof(float3, y);
    offset[2] = offsetof(float3, z);
    int block_length[3] = {1, 1, 1};
    MPI_Datatype types[3] = {MPI_FLOAT, MPI_FLOAT, MPI_FLOAT};
    MPI_Datatype FLOAT_3;
    MPI_Type_create_struct(count, block_length, offset, types, &FLOAT_3);
    MPI_Type_commit(&FLOAT_3);
    return FLOAT_3;
}
```

# 2 Collective MPI computation

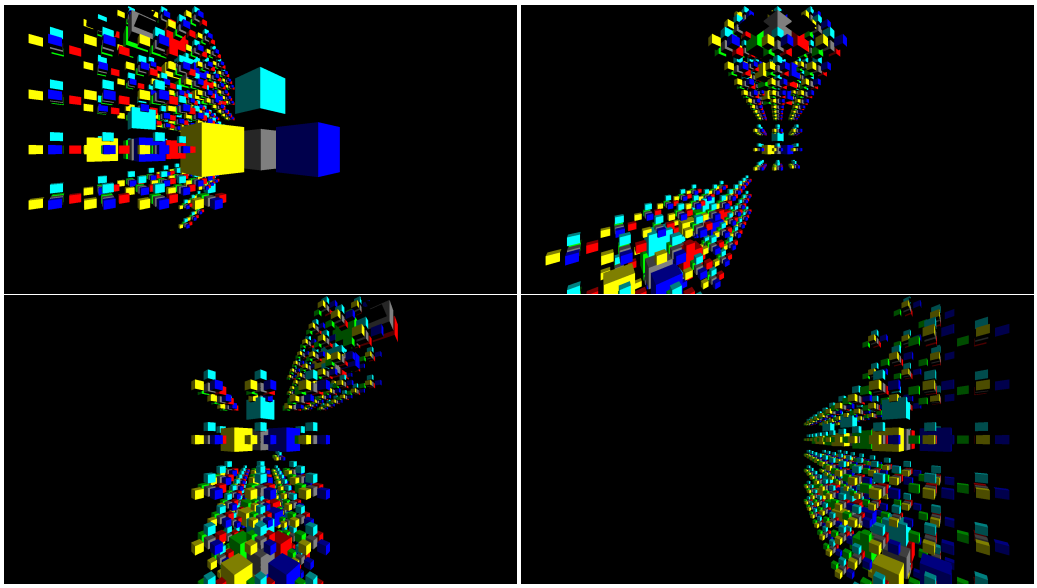
## 2.1 Collective Construction

I solved this problem by first calculating the amount of iterations required to render the whole picture. From studying the algorithm I quickly realized that every node expanded by 26 for every depth. Resulting in a total of

$$\sum_{i=0}^{depth} 26^i = total\_renders$$

Using this sum it was easy to divide the rendering amongst the ranks. Thus, every rank will only render "nodes" that are between its index range. The

drawback with this solution is that every rank still has to calculate position and scale of every object. An improvement to the algorithm would be to also divide the calculation of positions and scale between the processes. Below is the image 4 ranks produced before combining them.



## 2.2 MPI_Allreduce

In this task I only used *MPI_Allreduce* to reduce both the depth and the frame buffers. As specified in the task I first reduced the depthBuffer, then used this to create a new frameBuffer with only top-level pixels. After this, the new frambuffer was Bitwise-OR reduced to produce the final image.

## 2.3 Measurements

The solution scaled okay with more ranks. The biggest speedup is from 1 to 2 ranks, and after that the speedup is a little slower. At one point; from 7 to 8 it is actually slower. However, the biggest slow-down is when I hit 9+ ranks. This is because the computer only have 8 cores, meaning it will not be able to compute all 9+ ranks in parallel, resulting in a slow-down (ending up with some code running sequential).

The time spent computing a single image without MPI was about 450ms, while the parallel computing was about 180ms, a speedup of around **2,5x**. When rendering the Audi R8 with sequential code it spent around 115 seconds on average, while it spent around 25 seconds in parallel, giving a speedup of around **4,6x**. The reason to why it has more of a speedup in the audi case is that it

spends **A LOT** more time rendering than calculating position/scale (which gives a significant overhead in the cube case, since it has to render less).

From the two tests above we can clearly see that it is more efficient to render an image in parallel rather than sequential. This also makes a lot of sense, since dividing a task amongst two people is faster than having one person do the whole task. As seen from the Audi example it is clear that when the rendering gets pretty heavy, we benefit a lot from doing the work in parallel. With that being said, it is very important that the workload between the ranks are equal to see good parallel performance. This is because every rank is halted until every rank is finished before it can print the image. An unequal balance load would slow down some ranks, resulting in a total slow-down.