1 Task One - Parallelise the Implicit Surface Program

I aimed to tackle my program by splitting the array up per it's dimension. Which means that thread one would do the X dimension, thread two would do the Y dimension and thread three would do the X dimension. I was then going to go further and allow the user to add more threads (as long as it was divisible by three). Meaning that if a user had six threads, thread one would do half of X, and thread four would do the second half of X. However I had difficulties understanding how to combine the data quickly afterwards.

I had difficulties grasping how to correctly loop through each dimension independently, in the end I just had each thread calculate the centre for every pixel in that dimension. I believe I would have needed to loop through two or three dimensions in each core, and use modulo and integer division and an if statement to only pull out and assign the index corresponding that position, but I couldn't grasp it or how efficient that method would have been considering it would still be iterating as much as the single threaded version.

I could only compute the centre in my threads as I had problems implementing the distance function per vertex due to the function requiring multiple functions, I think the way to do this would have been to just add placeholder 0's in for the Y and Z inputs and then just return a running total at the end. This lead to not being able to compute the voxel value and was explained earlier with me not grasping how to iterate through the array; it made me unable to store the value in the appropriate location.

Required files are in the appendix.

2 Task Two - HPC Experimentation

2.1 Compiler: GNU, MPI: Open

module purge module load compiler/gnu/auto module load mpi/openmpi/1.8.5

Test One = 3.64 Test Two = 3.67 Test Three = 3.64

Average = 3.65

Compiler: GNU, MPI: Intel module purge module load compiler/gnu/auto module load mpi/intel/auto

Test One = 3.64 Test Two = 3.65 Test Three = 3.65

Average = 3.645

2.2 Compiler: Intel, MPI: Open

module purge module load compiler/intel/auto module load mpi/openmpi/1.8.5

Test One = 3.66 Test Two = 3.64 Test Three = 3.64

Average = 3.646

2.3 Compiler: Intel, MPI: Intel

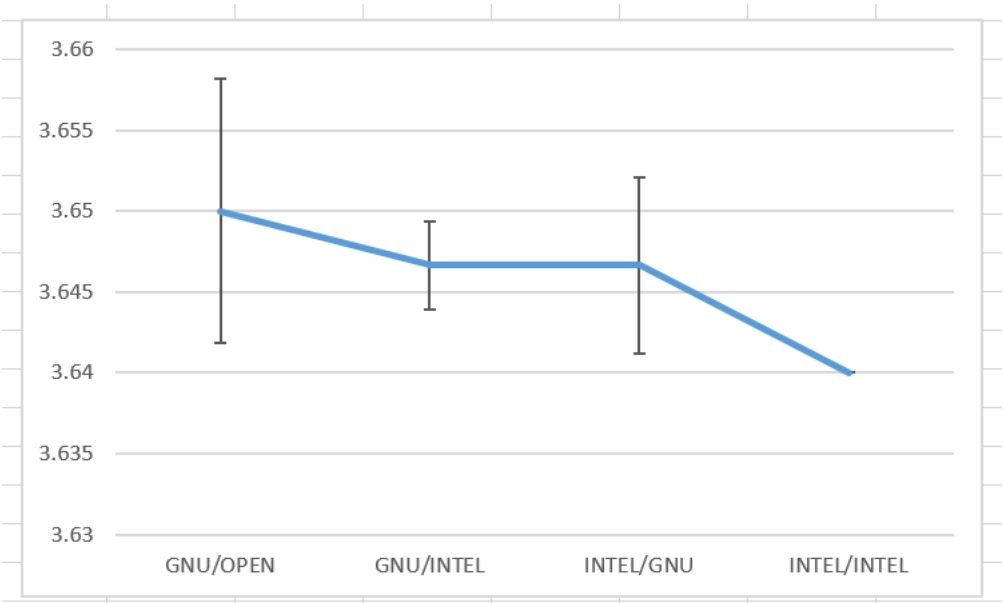
module purge module load compiler/intel/auto module load mpi/intel/auto

Test One = 3.64 Test Two = 3.64 Test Three = 3.64

Average = 3.64

2.3.1 Correlation

My graph shows that the Intel/Intel combination is both the fastest and the most reliable combination as it produced a standard deviation and standard error of zero.



	GNU/OPEI	GNU/INTE	INTEL/GNI	INTEL/INTEL
1	3.64	3.64	3.66	3.64
2	3.67	3.65	3.64	3.64
3	3.64	3.65	3.64	3.64
avg	3.65	3.646667	3.646667	3.64
stddev	0.014142	0.004714	0.009428	0
stderr	0.008165	0.002722	0.005443	0

Figure 1: Graph showing the averages of each compiler and mpi combo with error bars.

3 Appendix: Code

```
/**
*************************************************************************

*
*       @file           ImplicitSurface.cxx
*
*       @brief          Functions to build 3D implicit surfaces. To
    know what implicit
*                               surfaces are, please visit Paul Bourke'
    s tutorial available at
*                               http://paulbourke.net/geometry/
    implicitsurf/
*
*       @version        1.0
*
*       @date           08/03/2014
*
*       @author         Dr Franck P. Vidal
*
*************************************************************************

*/
```

```
//
    ********************************************************************************

//      Include
//
    ********************************************************************************

#include <fstream>
#include <iostream>
#include <cstdlib>
#include <mpi.h>

#ifndef IMPLICIT_SURFACE_H
#include "ImplicitSurface.h"
#endif


//
    ********************************************************************************

//      Name space
//
    ********************************************************************************

//using namespace Graphics;
using namespace std;


//
    ********************************************************************************

//      Constant variables
//
    ********************************************************************************



//
    ********************************************************************************

//      Global variables
//
    ********************************************************************************



//
    ********************************************************************************

//      Function declaration
//
    ********************************************************************************



//------------------------------------------------------
void voxelise(DensityFunctionType aDensityFunction,
              float a,
```

```
                float b,
                const std::vector<float>& apControlPointSet,
                std::vector<float>& apVoxelDataSet,
                unsigned int aVolumeSize[3],
                float aVoxelSize[3],
                float aVolumeCentre[3],
                int argc,
                char **argv)
//-------------------------------------------------------------
{
    // Clear the voxel data
    apVoxelDataSet.clear();

    // Set the size of the buffer is necessary
    unsigned int number_of_voxels(aVolumeSize[0] * aVolumeSize[1] *
        aVolumeSize[2]);
    if (apVoxelDataSet.size() != number_of_voxels)
    {
        // Resize the buffer
        apVoxelDataSet.resize(number_of_voxels, 0);
    }

    // Compute the half size of a voxel
    float half_voxel_size[3] = {
            aVoxelSize[0] / 2.0f,
                        aVoxelSize[1] / 2.0f,
                        aVoxelSize[2] / 2.0f
    };

    // Compute the half size of the volume
        float half_volume_size[3] = {
            half_voxel_size[0] * aVolumeSize[0],
                        half_voxel_size[1] * aVolumeSize[1],
                        half_voxel_size[2] * aVolumeSize[2]
    };

    // Compute the position of the centre of the first voxel
        float offset[3] = {
        aVolumeCentre[0] - half_volume_size[0] + half_voxel_size[0],
        aVolumeCentre[1] - half_volume_size[1] + half_voxel_size[1],
        aVolumeCentre[2] - half_volume_size[2] + half_voxel_size[2]
    };

    MPI::Init(argc, argv);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    int divisor = 1;
    if (world_size == 7)
    {
        divisor = 2;
    }

    if (world_size != 4 || world_size != 7)
    {
        return;
    }
```

```
if (world_rank == 0) {

    float centerX[512];
    float centerY[512];
    float centerZ[512];

    MPI_Recv(centerX, 512, MPI_FLOAT, 1, 1, MPI_COMM_WORLD, NULL);
    MPI_Recv(centerY, 512, MPI_FLOAT, 2, 1, MPI_COMM_WORLD, NULL);
    MPI_Recv(centerZ, 512, MPI_FLOAT, 3, 1, MPI_COMM_WORLD, NULL);

// Process all the slides
    for (unsigned int z(0); z < aVolumeSize[2]; ++z)
    {
    // Store the voxel centre
    float voxel_center[3];

    // Update the centre of the current voxel
            voxel_center[2] = centerZ[z];

    // Process all the rows
            for (unsigned int y(0); y < aVolumeSize[1]; ++y)
            {
        // Update the centre of the current voxel
                    voxel_center[1] = centerY[y];

        // Process all the colums
                    for (unsigned int x(0); x < aVolumeSize[0]; ++x
                        )
                    {
            // Value of the current voxel
                            float voxel_value(0);

                        // Update the centre of the current voxel
            voxel_center[0] = centerX[x];

                            // Process all the points
                            for (std::vector<float>::const_iterator
                                point_ite(apControlPointSet.begin()
                               );
                                            point_ite !=
                                               apControlPointSet.
                                               end();
                                            point_ite += 3)
                            {
                    // Compute the distance between the control point
                       and the centre of the current voxel
                        float r(distance(voxel_center, &(*point_ite)));
                                    voxel_value += evaluate(
                                       aDensityFunction, r, a, b);
                            }

            // Compute the voxel index
            unsigned int voxel_index(z * aVolumeSize[0] *
               aVolumeSize[1] + y * aVolumeSize[0] + x);

            // Set the value of the current voxel
            apVoxelDataSet[voxel_index] = voxel_value;
```

```
                        }
                }
        }
}


        else if (world_rank == 1)
        {
                float centre[512];

                for (unsigned int i(0); i < aVolumeSize[0]/divisor; ++i
                    )
                {
                        centre[i] = offset[0] + i * aVoxelSize[0];
                }
                MPI_Send(&centre, 512, MPI_FLOAT, 0, 1, MPI_COMM_WORLD)
                    ;
        }


        else if (world_rank == 2)
        {
                float centre[aVolumeSize[1]/divisor];

                for (unsigned int i(0); i < aVolumeSize[1]/divisor; ++i
                    )
                {
                        centre[i] = offset[1] + i * aVoxelSize[1];
                }
                MPI_Send(&centre, 512, MPI_FLOAT, 0, 1, MPI_COMM_WORLD)
                    ;
        }

        else if (world_rank == 3)
        {
                float centre[512];

                for (unsigned int i(0); i < aVolumeSize[2]; ++i)
                {
                        centre[i] = offset[2] + i * aVoxelSize[2];
                }
                MPI_Send(&centre, 512, MPI_FLOAT, 0, 1, MPI_COMM_WORLD)
                    ;
        }

        MPI::Finalize();
}


//
    ------------------------------------------------------------------------

void writeVoxelData(const char* aFileName, const std::vector<float>&
    apVoxelDataSet)
//
    ------------------------------------------------------------------------

{
```

```
    // Open the file
    std::ofstream output_file;
    output_file.open(aFileName, std::ios::out | std::ios::trunc | std::
        ios::binary);

    // The file is not open
    if (!output_file.is_open())
        {
                std::cerr << "Cannot␣write␣the␣file␣(" << aFileName <<
                    ".)" << std::endl;
                exit(EXIT_FAILURE);
        }

    // Process all the voxel
    for (std::vector<float>::const_iterator voxel_ite(apVoxelDataSet.
        begin());
             voxel_ite != apVoxelDataSet.end();
             ++voxel_ite)
    {
         output_file.write(reinterpret_cast<const char*>(&(*voxel_ite)),
             sizeof(float));
    }

    // Close the file
    output_file.close();
}
```

4 Makefile

```
CXX=mpicxx

BIN=test
LIB=libImplicitSurface.a
OBJECTS= ImplicitSurface.o test.o

CXXFLAGS+=-I../include -O3 -Wall
LDFLAGS+=-L. -lImplicitSurface


all: $(BIN)


$(BIN): $(LIB) test.o
        @echo Build $@
        @$(CXX) -o $@ test.o $(LDFLAGS)


$(LIB): ImplicitSurface.o
        @echo Build $@ from $<
        @$(AR) rcs $@ $<


# Default rule for creating OBJECT files from CXX files
%.o: ../src/%.cxx
        @echo Build $@ from $<
        @$(CXX) $(CXXFLAGS) -c $< -o $@
```

```
ImplicitSurface.o: ../include/ImplicitSurface.h ../include/
   ImplicitSurface.inl ../src/ImplicitSurface.cxx
test.o: ../include/ImplicitSurface.h $(LIB) ../src/test.cxx


clean:
        $(RM) $(OBJECTS)
        $(RM) $(LIB)
        $(RM) $(BIN)
        $(RM) -r ../doc/html
        $(RM) -r ../doc/tex
```

5 SlurmScript

```
#!/bin/bash --login
#SBATCH --job-name=ImplicitSurfaceCXX
#SBATCH -o ImplicitSurface.out
#SBATCH -e ImplicitSurface.err
#SBATCH -t 0-12:00
#SBATCH -n 4

module purge
module load compiler/gnu/auto
module load mpi/openmpi/1.8.5

mpirun -np $SLURM_NTASKS ./MPI_assignment/build/test
```