ICP3029 – HPC Assignment

Parallelization of the Implicit Surface Program

Jakub Lukáč*

April 3, 2017

This report describes the process of parallelization of a single-threaded code for generating Implicit Surfaces. This assignment was part of the distributed systems module and together with laboratories introduce the concept of Message Passing Interface (MPI).

Main goals of this assignment were to understand the single-threaded code, implement parallelization using the MPI, experiment with resulting implementation and do comparison. The parallelization approach and experimentation will be described in more details in the sections below.

Parallelization

The `voxelise` function was the one mostly affected by the parallelization. Firstly, I divided an implementation of this function to simple version which could process the certain range of slides from the final volume. For this purpose, a `simple_voxelise` function was implemented and it is a base for splitting the workload to multiple cores in cluster.

Then, the original function interface is used to check for available cluster cores. Next, a core with the lowest ID, a master core, splits workload to other slave cores. This process is controlled and organized by using the MPI to pass messages between cores in cluster.

The final volume is then reconstructed by the master core which receives partial volume parts represented by slide ranges from slave cores. The master core is receiving these slides in specific order and move these data to final data set.

Experimentation

The initial comparison was computation of the simple byte to byte difference between parallelized and single-threaded version raw output of the program. This test was successful for each density function and it proofs correctness of the parallelized implementation. For further experimentation, the `META_BALL` density function was used.

More importantly, I did the efficiency comparison for the selected implicit surface. The parallelized implementation was run ten times for each number of cores and data are shown in Figure 1 and 2. Each figure contains a standard box plot and an average value for each run. Besides the different number of cores, the parallelized implementation was compiled by two standard compiles with two different MPI implementations. Figure 1 is showing results of the program which was compiled by GCC compiler using the Open MPI. Figure 2 is showing results of the program which was compiled by Intel ICC

*eeu893@bangor.ac.uk

compiler using the Intel MPI. These testing runs were run on Intel chip-sets and resulted in the better performance using the proprietary Intel software.

Furthermore, the difference between the two of those compilers can be seen even better in Figure 3. The blue speedup curve represents version, which was compiled by Intel ICC compiler, and this version was faster especially with the increasing number of cores.

Another essential point is that those curves in Figure 3 also show the implications of Amdhal's Law. The theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.[1] The limitation can be seen also in the Figure 3 where both speedup curves are bounded above. The Intel ICC version is bounded by 11 and GCC version is bounded by 8. There is also another implication, the part which can't be parallelized is taking more execution time with the increasing number of cores. Therefore, these two aspects are causing reduction of speedup for the larger number of cores. According to this experimentation, the maximal speedup for the parallelized implementation can be achieved with about 40 cores for Intel ICC version and 32 cores for GCC version.

Conclusion

It has been shown that it is possible to parallelize the single-threaded implementation and achieve significant speedup. However, the current version has its upper speedup limit, which is caused by non-parallelized sections.

Next, experimentation could add more cores or what could be more important compute larger volumes to see how will parallelization change according to change of the workload. Also, better results could be accomplished by dividing slices to smaller pieces which will result to more uniformly distributed workload between the cores.

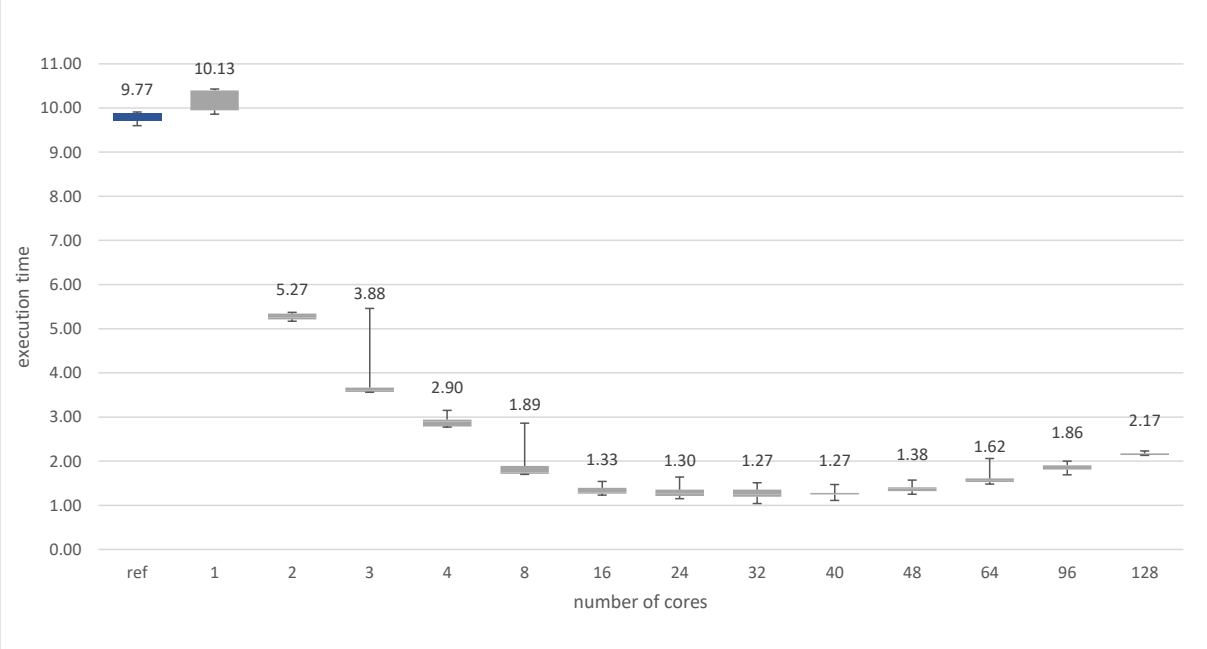

Figure 1: Box plot for the implementation compiled by GCC compiler using Open MPI

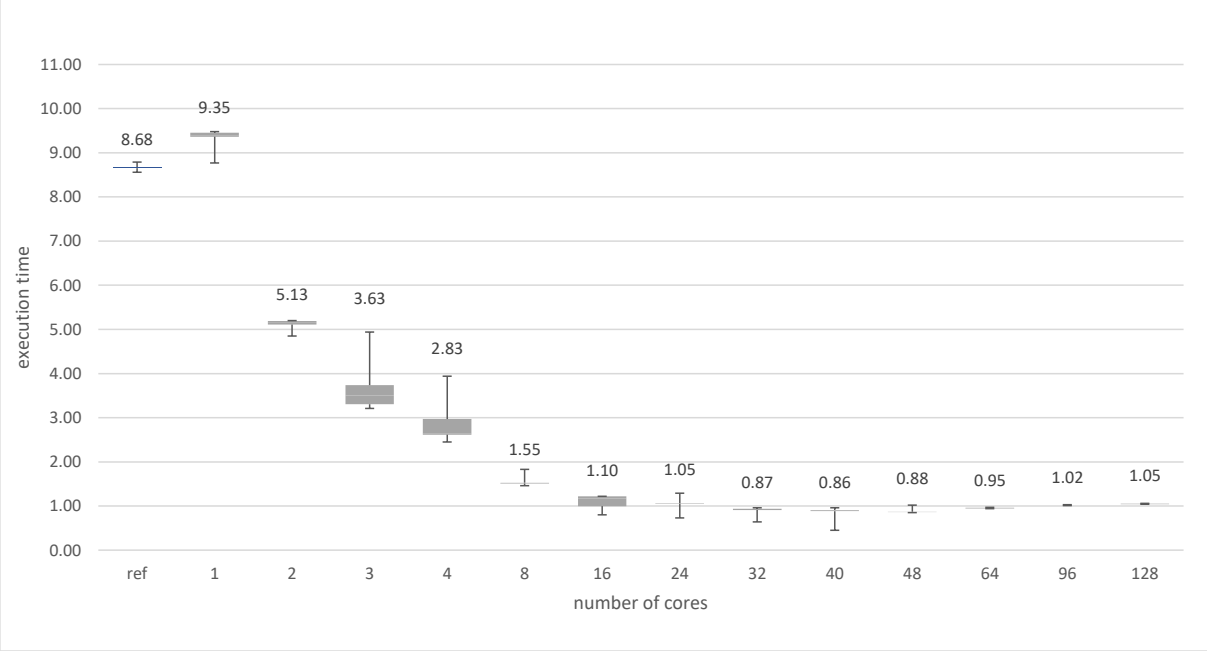

Figure 2: Box plot for the implementation compiled by Intel ICC compiler using Intel MPI

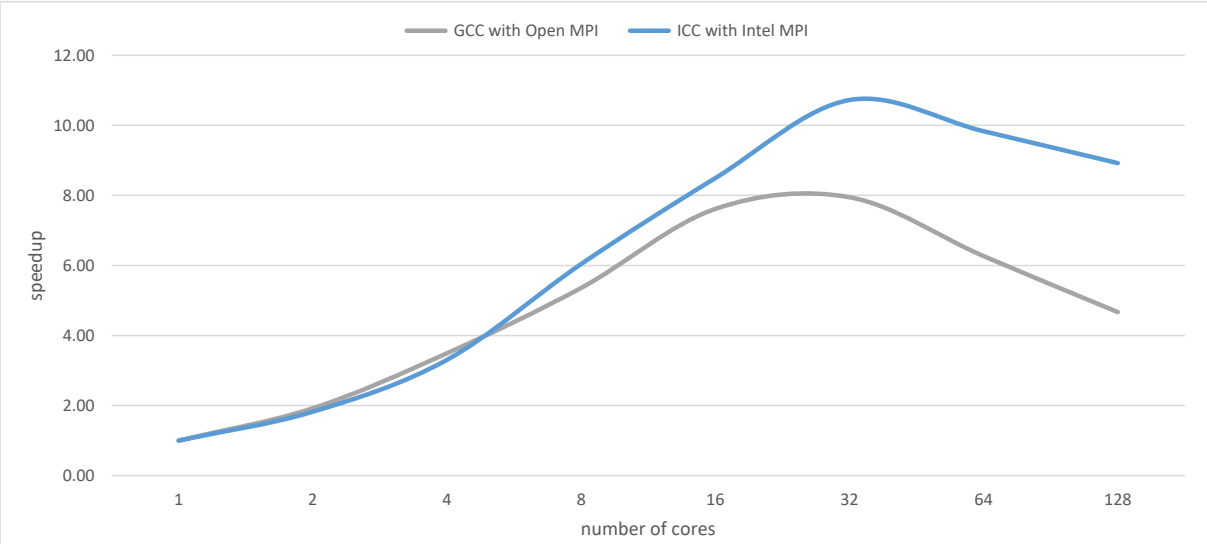

Figure 3: Speedup comparison

References

[1] Wikipedia, "Amdahl's law — Wikipedia, The Free Encyclopedia," 2017, [Online; accessed 2-April-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=763349931

Appendix

test.cxx

```
/**
*****************************************************************************
*
*  @file      test.cxx
*  @brief     Main program.
*  @version   1.0
*  @date      08/03/2014
*  @author    Dr Franck P. Vidal
*  @author    Jakub Lukac
*
*****************************************************************************
*/


//*****************************************************************************
//  Include
//*****************************************************************************
#include <mpi.h>
#include <iostream>
#include <string>
#include <cstdlib>
#include <time.h>
#include <getopt.h>

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


//------------------------------
int main(int argc, char** argv)
//------------------------------
{
  MPI_Init(&argc, &argv);

  // ID of actual process
  int core_id;
  MPI_Comm_rank(MPI_COMM_WORLD, &core_id);

  // Choose a density function
  DensityFunctionType density_function(META_BALL);
  float a(2);
  float b(3);

  // Set the volume data set
  std::vector<float> p_voxel_data;

  unsigned int p_number_of_voxel[3] = {512, 512, 512};
  float p_voxel_size[3] = {0.025, 0.025, 0.025};
  float p_centre[3] = {0.0, 0.0, 0.0};

  // Set the control points
  std::vector<float> p_control_point_set;

  // Add the control points
  float control_point_1[3] = {-2.0, 0.0, 1.0};
  p_control_point_set.push_back(control_point_1[0]);
  p_control_point_set.push_back(control_point_1[1]);
  p_control_point_set.push_back(control_point_1[2]);

  float control_point_2[3] = { 2.0, 1.0, 0.0};
  p_control_point_set.push_back(control_point_2[0]);
  p_control_point_set.push_back(control_point_2[1]);
  p_control_point_set.push_back(control_point_2[2]);

  float control_point_3[3] = { 1.0, 2.0, -2.0};
  p_control_point_set.push_back(control_point_3[0]);
  p_control_point_set.push_back(control_point_3[1]);
  p_control_point_set.push_back(control_point_3[2]);

  /*float control_point_4[3] = { 0.0, -2.0, 2.0};
  p_control_point_set.push_back(control_point_4[0]);
  p_control_point_set.push_back(control_point_4[1]);
  p_control_point_set.push_back(control_point_4[2]);
  */

  clock_t start;
  if (core_id == MASTER_ID)
  {
    // Start the timer
    start = clock();
  }

  ////////////////////////////////////////////////////////////////////////////
  // Paralel job...

  // Generate the density field
  voxelise(density_function,
```

```
          a,
          b,
          p_control_point_set,
          p_voxel_data,
          p_number_of_voxel,
          p_voxel_size,
          p_centre);

  ////////////////////////////////////////////////////////////////////////////

  if (core_id == MASTER_ID)
  {
    // Stop the timer
    clock_t end(clock());

    // Get the duration of the computations
    float number_of_seconds(float(end - start) / CLOCKS_PER_SEC);

    // Display the computing time
    std::cout << "Computing time:\t" << number_of_seconds << " seconds." << std::endl;

    // Save the data in a file
    writeVoxelData("output.raw", p_voxel_data);
  }

  MPI_Finalize();

  return (EXIT_SUCCESS);
}
```

ImplicitSurface.h

```
/**
*******************************************************************************
*
*  @file      ImplicitSurface.h
*  @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
*
*******************************************************************************
*/


#ifndef IMPLICIT_SURFACE_H
#define IMPLICIT_SURFACE_H


//*****************************************************************************
//  Include
//*****************************************************************************
#include <vector>

#define MASTER_ID 0

//-----------------------------------------------------------------------------
///  Type of density function.
//-----------------------------------------------------------------------------
typedef enum DensityFunctionTypeTag
{
  SPHERE,
  BLOBBY_MOLECULE,  ///< Blobby molecule, first created by Jim Blinn and modelled after electron density fields
  META_BALL,        ///< Meta balls
  SOFT_OBJECT       ///< Soft Objects, first created by the Wyvill brothers
} DensityFunctionType;


//-----------------------------------------------------------------------------
///  Use the density function that creates spheres.
/**
*  @param r:  the distance of the voxel to the center of the control point
*  @return the density value corresponding to distance r
*/
//-----------------------------------------------------------------------------
float evaluateSphere(float r);


//-----------------------------------------------------------------------------
///  Use the density function that creates Blobby Molecules.
///  It was first created by Jim Blinn.
///  It is modelled after electron density fields.
/**
*  @param r:  the distance of the voxel to the center of the control point
*  @param a:  the height
*  @param b:  the standard deviation of the curve
*  @return the density value corresponding to distance r
*/
```

```
//---------------------------------------------------------------------------
float evaluateBlobbyMolecule(float r, float a, float b);


//---------------------------------------------------------------------------
///  Use the density function that creates Meta Balls.
/**
*  @param r:  the distance of the voxel to the center of the control point
*  @param a:  the scaling factor
*  @param b:  the maximum distance a control primitive contributes to the field
*  @return the density value corresponding to distance r
*/
//---------------------------------------------------------------------------
float evaluateMetaBall(float r, float a, float b);


//---------------------------------------------------------------------------
///  Use the density function that creates Soft Objects.
///  It was first created by the Wyvill brothers.
/**
*  @param r:  the distance of the voxel to the center of the control point
*  @param a:  the scaling factor
*  @param b:  the maximum distance a control primitive contributes to the field
*  @return the density value corresponding to distance r
*/
//---------------------------------------------------------------------------
float evaluateSoftObject(float r, float a, float b);


//---------------------------------------------------------------------------
///  Evaluate the density function.
/**
*  @param aDensityFunction:  the type of density function
*  @param r:  the distance of the voxel to the center of the control point
*  @param a:  the first parameter of the density function (default value: 0)
*  @param b:  the second parameter of the density function (default value: 0)
*  @return the density value corresponding to distance r
*/
//---------------------------------------------------------------------------
float evaluate(DensityFunctionType aDensityFunction,
        float r,
        float a = 0,
        float b = 0);


//---------------------------------------------------------------------------
///  The distance between two 3D points.
/**
*  @param aPoint1:  the first point
*  @param aPoint2:  the second point
*  @return the distance between aPoint1 and aPoint2
*/
//---------------------------------------------------------------------------
float distance(const float aPoint1[3], const float aPoint2[3]);


//---------------------------------------------------------------------------
///  Write a volume dataset into a file (binary RAW format).
/**
*  @param aFileName:  the name of the file
*  @param apVoxelDataSet:  the volume dataset to write
*/
//---------------------------------------------------------------------------
void writeVoxelData(const char* aFileName, const std::vector<float>& apVoxelDataSet);


//---------------------------------------------------------------------------
///  Write a volume dataset into a file (binary RAW format).
/**
*  @param aFileName:  the name of the file
*  @param apVoxelDataSet:  the volume dataset to write
*/
//---------------------------------------------------------------------------
void writeVoxelData(const std::string& aFileName, const std::vector<float>& apVoxelDataSet);


//---------------------------------------------------------------------------
///  Write a volume dataset into a file (binary RAW format).
/**
*  @param aDensityFunction:  the type of density function
*  @param a:  the first parameter of the density function
*  @param b:  the second parameter of the density function
*  @param apControlPointSet:  the set of control points
*  @param apVoxelDataSet:  the volume dataset to generate
*  @param aNumberOfVoxels:  the number of voxels in the volume dataset
*  @param aVoxelSize:  the size of voxels
*  @param aVolumeCentre:  the position in 3D of the centre of the volume dataset
*/
//---------------------------------------------------------------------------
void voxelise(DensityFunctionType aDensityFunction,
        float a,
        float b,
        const std::vector<float>& apControlPointSet,
```

```
        std::vector<float>& apVoxelDataSet,
        unsigned int aNumberOfVoxels[3],
        float aVoxelSize[3],
        float aVolumeCentre[3]);


//*****************************************************************************
#include "ImplicitSurface.inl"


#endif // IMPLICIT_SURFACE_H
```

ImplicitSurface.cxx

```
/**
*******************************************************************************
*
*  @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
*  @author    Jakub Lukac
*
*******************************************************************************
*/

//*****************************************************************************
//  Include
//*****************************************************************************
#include <mpi.h>
#include <fstream>
#include <iostream>
#include <cstdlib>
#include <algorithm>

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


//*****************************************************************************
//  Name space
//*****************************************************************************
//using namespace Graphics;


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


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


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

//-----------------------------------------------------------------------------
///  Write a volume dataset into a vector.
/**
*  @param slidesRange:         the range of z coordinates to process
*  @param aFirstVoxelOffset:   the centre of the first voxel
*  @param aDensityFunction:    the type of density function
*  @param a:                   the first parameter of the density function
*  @param b:                   the second parameter of the density function
*  @param apControlPointSet:   the set of control points
*  @param apVoxelDataSet:      the volume dataset to generate
*  @param aNumberOfVoxels:     the number of voxels in the volume dataset
*  @param aVoxelSize:          the size of voxels
*/
//-----------------------------------------------------------------------------
void simple_voxelise(
        std::vector<unsigned int> slidesRange,
        float aFirstVoxelOffset[3],
        DensityFunctionType aDensityFunction,
        float a,
        float b,
        const std::vector<float>& apControlPointSet,
        std::vector<float>& apVoxelDataSet,
        unsigned int aVolumeSize[3],
        float aVoxelSize[3])
//-----------------------------------------------------------------------------
{
  // Clear the voxel data
  apVoxelDataSet.clear();
```

```
  // Set the size of the buffer is necessary
  unsigned int number_of_voxels(aVolumeSize[0] * aVolumeSize[1] * (slidesRange[1] - slidesRange[0]));
  if (apVoxelDataSet.size() != number_of_voxels)
  {
    // Resize the buffer
    apVoxelDataSet.resize(number_of_voxels, 0.0);
  }

  // Process all the slides
  for (unsigned int z(slidesRange[0]), z_i(0); z < slidesRange[1]; ++z, ++z_i)
  {
    // Store the voxel centre
    float voxel_center[3];

    // Update the centre of the current voxel
    voxel_center[2] = aFirstVoxelOffset[2] + z * aVoxelSize[2];

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

      // 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] = aFirstVoxelOffset[0] + x * aVoxelSize[0];

        // 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_i * aVolumeSize[0] * aVolumeSize[1] + y * aVolumeSize[0] + x);

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

//-------------------------------------------------------
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])
//-------------------------------------------------------
{
  // 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, 0.0);
  }

  // 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]
  };

  std::vector<float> core_result;

  // ID of actual process
  int core_id;
  MPI_Comm_rank(MPI_COMM_WORLD, &core_id);

  if (core_id == MASTER_ID)
  {
    //// Master core ////

    // Check how many cores are running
    int number_of_cores;
    MPI_Comm_size(MPI_COMM_WORLD, &number_of_cores);

    unsigned int chunk_size(aVolumeSize[2] / number_of_cores);
    unsigned int chunk_remider(aVolumeSize[2] % number_of_cores);

    // Initialize the ranges
    std::vector< std::vector<unsigned int> > slidesRanges(number_of_cores, std::vector<unsigned int>());
    unsigned int previous_upper_limit(0);
    for (std::vector< std::vector<unsigned int> >::iterator range(slidesRanges.begin()); range != slidesRanges.end(); ++range)
    {
      (*range).resize(2, 0);
      (*range)[0] = previous_upper_limit;
      (*range)[1] = (*range)[0] + chunk_size + (chunk_remider != 0 ? --chunk_remider, 1 : 0);
      previous_upper_limit = (*range)[1];
    }

    // Send the ranges to slaves cores
    for (unsigned int i(1); i < number_of_cores; ++i)
    {
      MPI_Send(&(slidesRanges[i][0]), slidesRanges[i].size(), MPI_UNSIGNED, i, 0, MPI_COMM_WORLD);
    }

    // Generate the density field for the first range
    simple_voxelise(slidesRanges[0], offset, aDensityFunction, a, b, apControlPointSet, core_result, aVolumeSize, aVoxelSize);

    // Summarize all results
    unsigned int data_set_offset(0);
    std::move(core_result.begin(), core_result.end(), apVoxelDataSet.begin() + data_set_offset);  // slides from the master
    data_set_offset += core_result.size();
    for (unsigned int i(1); i < number_of_cores; ++i)
    {
      core_result.resize(aVolumeSize[0] * aVolumeSize[1] * (slidesRanges[i][1] - slidesRanges[i][0]), 0.0);

      // Receive slides from slave cores
      MPI_Recv(&(core_result[0]), core_result.size(), MPI_FLOAT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

      // Move a partial result to data set
      std::move(core_result.begin(), core_result.end(), apVoxelDataSet.begin() + data_set_offset);
      data_set_offset += core_result.size();
    }
  }
  else
  {
    //// Slave core ////

    // Receive the range from the master core
    std::vector<unsigned int> slidesRange(2, 0);
    MPI_Recv(&(slidesRange[0]), slidesRange.size(), MPI_UNSIGNED, MASTER_ID, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // Generate the density field for the particular range
    simple_voxelise(slidesRange, offset, aDensityFunction, a, b, apControlPointSet, core_result, aVolumeSize, aVoxelSize);

    // Send the field back to master
    MPI_Send(&(core_result[0]), core_result.size(), MPI_FLOAT, MASTER_ID, 0, MPI_COMM_WORLD);
  }
}

//-----------------------------------------------------------------------------
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();
}
```

Makefile

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

CXX=mpicxx
CXXFLAGS+=-std=c++0x
CXXFLAGS+=-I../include -O3 -Wall -g
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
```

test.slurm

```
#!/bin/bash --login
#SBATCH --job-name=mpi-test-distributed_systems
#SBATCH -o mpi_out/test.out
#SBATCH -e mpi_out/test.err
#SBATCH -t 0-12:00
#SBATCH --account=hpcw0284
#SBATCH --ntasks=4

module purge
module load compiler/gnu mpi/openmpi/1.6.4

mpirun -np $SLURM_NTASKS ./test >& mpi_out/test.log.np_$SLURM_NTASKS.$SLURM_JOBID
```