MPI_Assessment

Generated by Doxygen 1.9.3

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Domain< T > . . . . . ??
MPI_Data< T > . . . . . ??
TimingTest . . . . . ??

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

AS/Domain.cpp . . . . . ??
AS/Domain.h . . . . . ??
AS/Main.cpp . . . . . ??
AS/MPIDatatype.cpp . . . . . ??
AS/MPIDatatype.h . . . . . ??
AS/Parameters.h . . . . . ??
AS/SerialWaveSolver.cpp . . . . . ??
AS/TimingTest.cpp . . . . . ??
AS/TimingTest.h . . . . . ??
AS/WaveSolver.cpp . . . . . ??
AS/WaveSolver.h . . . . . ??

Chapter 3

Class Documentation

3.1 Domain< T > Class Template Reference

Public Member Functions

- Domain (int n_i, int n_j)
- void allocatePointers ()
- void createSubDomains (int p)
- void setSubDomainSize ()
- void findNeighbours ()
- void id_to_index (int id, int &id_row, int &id_column)
- int id_from_index (int id_row, int id_column)
- void swapGridPointers (T **&subDomain2D_1, T **&subDomain2D_2, T **&subDomain2D_3, T *&sub↩ Domain1D_1, T *&subDomain1D_2, T *&subDomain1D_3)
- void printsubDomain (Domain< T > *Dom)

Public Attributes

- T * old_grid_1d
- T * current_grid_1d
- T * new_grid_1d
- T ** old_grid_2d
- T ** current_grid_2d
- T ** new_grid_2d
- int sub_imax
- int sub_jmax
- int sub_i
- int sub_j
- int sub_rows
- int sub_cols
- int global_imax
- int global_jmax
- int global_i
- int global_j
- int * start_row
- int * num_rows
- int * start_column
- int * num_columns
- int neighbours [4]

3.1.1 Detailed Description

template<class T>
class Domain< T >

Definition at line 11 of file Domain.h.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Domain() [1/2]

```
template<class T >
Domain< T >::Domain
```

Definition at line 17 of file Domain.cpp.

3.1.2.2 Domain() [2/2]

```
template<class T >
Domain< T >::Domain (
            int n_i,
            int n_j )
```

Definition at line 41 of file Domain.cpp.

3.1.2.3 ~Domain()

```
template<class T >
Domain< T >::~Domain
```

Definition at line 47 of file Domain.cpp.

3.1.3 Member Function Documentation

3.1.3.1 allocatePointers()

```
template<class T >
void Domain< T >::allocatePointers
```

Definition at line 62 of file Domain.cpp.

3.1.3.2 createSubDomains()

```
template<class T >
void Domain< T >::createSubDomains (
            int p )
```

Definition at line 112 of file Domain.cpp.

3.1.3.3 findNeighbours()

```
template<class T >
void Domain< T >::findNeighbours
```

Definition at line 194 of file Domain.cpp.

3.1.3.4 id_from_index()

```
template<class T >
int Domain< T >::id_from_index (
            int id_row,
            int id_column )
```

Definition at line 289 of file Domain.cpp.

3.1.3.5 id_to_index()

```
template<class T >
void Domain< T >::id_to_index (
            int id,
            int & id_row,
            int & id_column )
```

Definition at line 282 of file Domain.cpp.

3.1.3.6 printsubDomain()

```
template<class T >
void Domain< T >::printsubDomain (
            Domain< T > * Dom )
```

Definition at line 301 of file Domain.cpp.

3.1.3.7 setSubDomainSize()

```
template<class T >
void Domain< T >::setSubDomainSize
```

Definition at line 138 of file Domain.cpp.

3.1.3.8 swapGridPointers()

```
template<class T >
void Domain< T >::swapGridPointers (
            T **& subDomain2D_1,
            T **& subDomain2D_2,
            T **& subDomain2D_3,
            T *& subDomain1D_1,
            T *& subDomain1D_2,
            T *& subDomain1D_3 )
```

Definition at line 89 of file Domain.cpp.

3.1.4 Member Data Documentation

3.1.4.1 current_grid_1d

```
template<class T >
T* Domain< T >::current_grid_1d
```

Definition at line 32 of file Domain.h.

3.1.4.2 current_grid_2d

```
template<class T >
T** Domain< T >::current_grid_2d
```

Definition at line 36 of file Domain.h.

3.1.4.3 global_i

```
template<class T >
int Domain< T >::global_i
```

Definition at line 59 of file Domain.h.

3.1.4.4 global_imax

```
template<class T >
int Domain< T >::global_imax
```

Definition at line 56 of file Domain.h.

3.1.4.5 global_j

```
template<class T >
int Domain< T >::global_j
```

Definition at line 60 of file Domain.h.

3.1.4.6 global_jmax

```
template<class T >
int Domain< T >::global_jmax
```

Definition at line 57 of file Domain.h.

3.1.4.7 neighbours

```
template<class T >
int Domain< T >::neighbours[4]
```

Definition at line 85 of file Domain.h.

3.1.4.8 new_grid_1d

```
template<class T >
T* Domain< T >::new_grid_1d
```

Definition at line 33 of file Domain.h.

3.1.4.9 new_grid_2d

```
template<class T >
T** Domain< T >::new_grid_2d
```

Definition at line 37 of file Domain.h.

3.1.4.10 num_columns

```
template<class T >
int* Domain< T >::num_columns
```

Definition at line 66 of file Domain.h.

3.1.4.11 num_rows

```
template<class T >
int* Domain< T >::num_rows
```

Definition at line 64 of file Domain.h.

3.1.4.12 old_grid_1d

```
template<class T >
T* Domain< T >::old_grid_1d
```

Definition at line 31 of file Domain.h.

3.1.4.13 old_grid_2d

```
template<class T >
T** Domain< T >::old_grid_2d
```

Definition at line 35 of file Domain.h.

3.1.4.14 start_column

```
template<class T >
int* Domain< T >::start_column
```

Definition at line 65 of file Domain.h.

3.1.4.15 start_row

```
template<class T >
int* Domain< T >::start_row
```

Definition at line 63 of file Domain.h.

3.1.4.16 sub_cols

```
template<class T >
int Domain< T >::sub_cols
```

Definition at line 48 of file Domain.h.

3.1.4.17 sub_i

```
template<class T >
int Domain< T >::sub_i
```

Definition at line 44 of file Domain.h.

3.1.4.18 sub_imax

```
template<class T >
int Domain< T >::sub_imax
```

Definition at line 41 of file Domain.h.

3.1.4.19 sub_j

```
template<class T >
int Domain< T >::sub_j
```

Definition at line 45 of file Domain.h.

3.1.4.20 sub_jmax

```
template<class T >
int Domain< T >::sub_jmax
```

Definition at line 42 of file Domain.h.

3.1.4.21 sub_rows

```
template<class T >
int Domain< T >::sub_rows
```

Definition at line 47 of file Domain.h.

The documentation for this class was generated from the following files:

- AS/Domain.h
- AS/Domain.cpp

3.2 MPI_Data< T > Class Template Reference

Public Member Functions

- void CreateMPI_Types (Domain< T > &Dom)
- void send_data (Domain< T > &Dom, MPI_Datatype MDataType, int neigh_to_send, MPI_Request *request)
- void recv_data (Domain< T > &Dom, MPI_Datatype MDataType, int neigh_to_receive, MPI_Request *request)

Public Attributes

- MPI_Datatype Left_Type
- MPI_Datatype Right_Type
- MPI_Datatype Top_Type
- MPI_Datatype Bottom_Type
- MPI_Datatype Left_Ghost_Type
- MPI_Datatype Right_Ghost_Type
- MPI_Datatype Top_Ghost_Type
- MPI_Datatype Bottom_Ghost_Type

3.2.1 Detailed Description

template<class T>
class MPI_Data< T >

Definition at line 9 of file MPIDatatype.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 MPI_Data()

```
template<class T >
MPI_Data< T >::MPI_Data
```

Definition at line 12 of file MPIDatatype.cpp.

3.2.2.2 ~MPI_Data()

```
template<class T >
MPI_Data< T >::~MPI_Data
```

Definition at line 30 of file MPIDatatype.cpp.

3.2.3 Member Function Documentation

3.2.3.1 CreateMPI_Types()

```
template<class T >
void MPI_Data< T >::CreateMPI_Types (
            Domain< T > & Dom )
```

Definition at line 66 of file MPIDatatype.cpp.

3.2.3.2 recv_data()

```
template<class T >
void MPI_Data< T >::recv_data (
            Domain< T > & Dom,
            MPI_Datatype MDataType,
            int neigh_to_receive,
            MPI_Request * request )
```

Definition at line 56 of file MPIDatatype.cpp.

3.2.3.3 send_data()

```
template<class T >
void MPI_Data< T >::send_data (
            Domain< T > & Dom,
            MPI_Datatype MDataType,
            int neigh_to_send,
            MPI_Request * request )
```

Definition at line 48 of file MPIDatatype.cpp.

3.2.4 Member Data Documentation

3.2.4.1 Bottom_Ghost_Type

```
template<class T >
MPI_Datatype MPI_Data< T >::Bottom_Ghost_Type
```

Definition at line 27 of file MPIDatatype.h.

3.2.4.2 Bottom_Type

```
template<class T >
MPI_Datatype MPI_Data< T >::Bottom_Type
```

Definition at line 25 of file MPIDatatype.h.

3.2.4.3 Left_Ghost_Type

```
template<class T >
MPI_Datatype MPI_Data< T >::Left_Ghost_Type
```

Definition at line 27 of file MPIDatatype.h.

3.2.4.4 Left_Type

```
template<class T >
MPI_Datatype MPI_Data< T >::Left_Type
```

Definition at line 25 of file MPIDatatype.h.

3.2.4.5 Right_Ghost_Type

```
template<class T >
MPI_Datatype MPI_Data< T >::Right_Ghost_Type
```

Definition at line 27 of file MPIDatatype.h.

3.2.4.6 Right_Type

```
template<class T >
MPI_Datatype MPI_Data< T >::Right_Type
```

Definition at line 25 of file MPIDatatype.h.

3.2.4.7 Top_Ghost_Type

```
template<class T >
MPI_Datatype MPI_Data< T >::Top_Ghost_Type
```

Definition at line 27 of file MPIDatatype.h.

3.2.4.8 Top_Type

```
template<class T >
MPI_Datatype MPI_Data< T >::Top_Type
```

Definition at line 25 of file MPIDatatype.h.

The documentation for this class was generated from the following files:

- AS/MPIDatatype.h
- AS/MPIDatatype.cpp

3.3 TimingTest Class Reference

Public Member Functions

- void timing_domain_setup (int id, int p)
- void timing_MPI_Create_setup (int id, int p)
- void timing_MPI_comms (int id, int p)
- void timing_inner_iterations (int id, int p)
- void timing_edge_iterations (int id, int p)
- void timing_total (int id, int p, int imax, int jmax)

Public Attributes

- vector< double > domain_setup_time
- vector< double > total_time
- vector< double > mpi_create_time
- vector< double > mpi_comms_time
- vector< double > inner_iterations_time
- vector< double > edge_iterations_time
- chrono::duration< double > time_MPI_create
- chrono::duration< double > time_total
- chrono::duration< double > domain_setup
- chrono::duration< double > time_MPI_comms
- chrono::duration< double > time_inner_iterations
- chrono::duration< double > time_edge_iterations

3.3.1 Detailed Description

Definition at line 7 of file TimingTest.h.

3.3.2 Member Function Documentation

3.3.2.1 timing_domain_setup()

```
void TimingTest::timing_domain_setup (
            int id,
            int p )
```

Definition at line 10 of file TimingTest.cpp.

3.3.2.2 timing_edge_iterations()

```
void TimingTest::timing_edge_iterations (
            int id,
            int p )
```

Definition at line 63 of file TimingTest.cpp.

3.3.2.3 timing_inner_iterations()

```
void TimingTest::timing_inner_iterations (
            int id,
            int p )
```

Definition at line 81 of file TimingTest.cpp.

3.3.2.4 timing_MPI_comms()

```
void TimingTest::timing_MPI_comms (
            int id,
            int p )
```

Definition at line 45 of file TimingTest.cpp.

3.3.2.5 timing_MPI_Create_setup()

```
void TimingTest::timing_MPI_Create_setup (
            int id,
            int p )
```

Definition at line 27 of file TimingTest.cpp.

3.3.2.6 timing_total()

```
void TimingTest::timing_total (
            int id,
            int p,
            int imax,
            int jmax )
```

Definition at line 99 of file TimingTest.cpp.

3.3.3 Member Data Documentation

3.3.3.1 domain_setup

```
chrono::duration< double> TimingTest::domain_setup
```

Definition at line 31 of file TimingTest.h.

3.3.3.2 domain_setup_time

```
vector<double> TimingTest::domain_setup_time
```

Definition at line 11 of file TimingTest.h.

3.3.3.3 edge_iterations_time

```
vector<double> TimingTest::edge_iterations_time
```

Definition at line 27 of file TimingTest.h.

3.3.3.4 inner_iterations_time

```
vector<double> TimingTest::inner_iterations_time
```

Definition at line 23 of file TimingTest.h.

3.3.3.5 mpi_comms_time

```
vector<double> TimingTest::mpi_comms_time
```

Definition at line 20 of file TimingTest.h.

3.3.3.6 mpi_create_time

```
vector<double> TimingTest::mpi_create_time
```

Definition at line 17 of file TimingTest.h.

3.3.3.7 time_edge_iterations

```
chrono::duration< double> TimingTest::time_edge_iterations
```

Definition at line 34 of file TimingTest.h.

3.3.3.8 time_inner_iterations

```
chrono::duration< double> TimingTest::time_inner_iterations
```

Definition at line 33 of file TimingTest.h.

3.3.3.9 time_MPI_comms

```
chrono::duration<double> TimingTest::time_MPI_comms
```

Definition at line 32 of file TimingTest.h.

3.3.3.10 time_MPI_create

```
chrono::duration< double> TimingTest::time_MPI_create
```

Definition at line 29 of file TimingTest.h.

3.3.3.11 time_total

```
chrono::duration<double> TimingTest::time_total
```

Definition at line 30 of file TimingTest.h.

3.3.3.12 total_time

```
vector<double> TimingTest::total_time
```

Definition at line 14 of file TimingTest.h.

The documentation for this class was generated from the following files:

- AS/TimingTest.h
- AS/TimingTest.cpp

Chapter 4

File Documentation

4.1 Domain.cpp

```

#define _USE_MATH_DEFINES

#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <vector>
#include <cmath>

#include "Domain.h"

using namespace std;


template <class T>
Domain<T>::Domain()
{
    //-----------------------------------------------------------------------------|
    //          CREATE DOMAIN
    //-----------------------------------------------------------------------------|
    // assign pointers to null
    // 1-D Arrays
    T* old_grid_1d = nullptr;
    T* grid_1d = nullptr;
    T* new_grid_1d = nullptr;
    // 2-D Arrays
    T** old_2d = nullptr;
    T** grid_2d = nullptr;
    T** new_2d = nullptr;


    // Subdomain Divisions
    this->start_row = new int[p];
    this->num_rows = new int[p];
    this->start_column = new int[p];
    this->num_columns = new int[p];
}

template <class T>
Domain<T>::Domain(int n_i, int n_j)
{
    allocatePointers();
}

template <class T>
 Domain<T>::~Domain()
{
    // 1-D Arrays
    delete[] old_grid_1d;
    delete[] current_grid_1d;
    delete[] new_grid_1d;

    // 2-D Arrays
    delete[] old_grid_2d;
    delete[] current_grid_2d;
    delete[] new_grid_2d;
}
```

```
 template <class T>
void Domain<T>::allocatePointers()
{
    //cout «" process " « id « " sub_imax " « this->sub_imax « " sub_jmax " « sub_jmax « endl;

    // Allocate memory to the pointers and initialize to zero
    old_grid_1d = new T[sub_imax * sub_jmax]{0};     // set to total elements
    current_grid_1d = new T[sub_imax * sub_jmax]{0};
    new_grid_1d = new T[sub_imax * sub_jmax]{0};

    old_grid_2d = new T * [sub_imax];           // set to total number of rows
    current_grid_2d = new T * [sub_imax];
    new_grid_2d = new T * [sub_imax];



    // point the 2D array to the 1D array for the domain
    for (int i = 0; i < sub_imax; i++)
    {
        old_grid_2d[i] = &old_grid_1d[i * sub_jmax];            // set to total number of rows
        current_grid_2d[i] = &current_grid_1d[i * sub_jmax];
        new_grid_2d[i] = &new_grid_1d[i * sub_jmax];
    }

}


template <class T>
void Domain<T>::swapGridPointers(T**&subDomain2D_1, T**&subDomain2D_2, T**& subDomain2D_3, T * &
 subDomain1D_1, T*& subDomain1D_2, T*& subDomain1D_3)
{
    // swap the grid pointers of a 2-D grid using a swap variable
    T** swap1 = subDomain2D_1;
    subDomain2D_1 = subDomain2D_2;
    subDomain2D_2 = subDomain2D_3;
    subDomain2D_3 = swap1;

    // swap the grid pointers of a 1-D grid using a swap variable
    T* swap2 = subDomain1D_1;
    subDomain1D_1 = subDomain1D_2;
    subDomain1D_2 = subDomain1D_3;
    subDomain1D_3 = swap2;

}





 //-------------------------------------------------------------------------------------------------
//                              FIND SUBDOMAINS

 //-------------------------------------------------------------------------------------------------
template <class T>
void Domain<T>::createSubDomains(int p)
{
    int min_gap = p;
    int top = sqrt(p) + 1;
    for (int i = 1; i <= top; i++)
    {
        if (p % i == 0)
        {
            int gap = abs(p / i - i);
            if (gap < min_gap)
            {
                min_gap = gap;
                this->sub_rows = i;
                this->sub_cols = p / i;
            }
        }

    }


    /*if (id == 0)
        cout « "Divide " « p « " into " « this->sub_rows « " by " « this->sub_cols « " grid" « endl;*/

}

template <class T>
void Domain<T>::setSubDomainSize()
{
    // default processor variables
    num_rows = new int[sub_rows]{};
    start_row = new int[sub_rows]{};
```

```
    num_columns = new int[sub_cols]{};
    start_column = new int[sub_cols]{};

    int start_r = 0;
    for (int k = 0; k < this->sub_rows; k++)
    {
        // Dividing the rows among the processes
        num_rows[k] = (this->global_jmax - start_r) / (this->sub_rows-k);
        start_row[k] = start_r; // set the first start point as 0

        // incrementing start point to the next start
        start_r += num_rows[k];
    }
    //cout « "process " « id «  "  id / sub_cols" « id / sub_cols « endl;
    //cout « "process "« id « " num_rows " « num_rows[id /sub_cols] « " start_row " « start_row[id / sub_cols] « endl;

    int start_c = 0;
    for (int k = 0; k < this->sub_cols; k++)
    {
        // Dividing the columns among the processes
        num_columns[k] =(this->global_imax - start_c) / (this->sub_cols-k);
        start_column[k] = start_c;

        start_c += num_columns[k];
    }

    //cout « "process " « id « " num_columns " « num_columns[id%sub_cols ] « " start_column " « start_column[id % sub_cols] « endl;
    //cout « "process" « id « " num_rows " « num_rows[id /sub_cols] « " start_row " « start_row[id / sub_cols] « endl;

    // With padding the maximum value of the subdomain is incremented by 2
    this->sub_imax = this->num_rows[id / sub_cols] + 2;
    this->sub_jmax = this->num_columns[id % sub_cols] + 2;

    // Set subdomain variables to processor indices
    this->sub_i = id / sub_cols;
    this->sub_j = id % sub_cols;

    /*cout « "Process id : " « id « "sub_i " « this->sub_i « " this->sub_j " « this->sub_j« endl;*/

    // Set Global domain parameters
    this->global_i = start_row[id / sub_cols];
    this->global_j = start_column[id % sub_cols];

    //cout « " process " « id « " global_i " « this->global_i « " global_j " « this->global_j « endl;
    //cout « " For process " « id « " num_rows " « this->num_rows[id % sub_cols] « "num_columns " « this->num_columns[id / sub_cols]«" Local Start i, j " « this->sub_i «" "« this->sub_j « " Global Start i, j " « this->global_i « " " « this->global_j « endl;


}


template <class T>
void Domain<T>::findNeighbours()
{

    // set the neighbour array to -1
    for(int i = 0; i < 4; i++)
    {
        neighbours[i] = -1;
    }


        // Loop on left and right of the current processor position
        for (int i = -1; i <= 1; i++)
        {
            // Loop on top and bottom of the current processor position
            for (int j = -1; j <= 1; j++)
            {
                //-------------------------------------------------------------------------------|
                // PERIODIC NEIGHBOURS
                //-------------------------------------------------------------------------------|
                if (boundary == 2) // If the boubdary is periodic
                {
                    // so we dont go negative we add sub_cols and sub_rows to the numbers
                    int neigh_i = (this->sub_j + i + this->sub_cols) % this->sub_cols;
                    int neigh_j = (this->sub_i + j + this->sub_rows) % this->sub_rows;
                    int neigh_id = neigh_i + neigh_j * this->sub_cols;

                    if (neigh_id != id)
                    {
                        if (i == -1 && j == 0) // left neighbour
                        {
                            this->neighbours[0] = neigh_id;
```

```
                        }
                        else if (i == 0 && j == -1) // top neighbour
                        {
                            this->neighbours[1] = neigh_id;
                        }
                        else if (i == 1 && j == 0) // right neighbour
                        {
                            this->neighbours[2] = neigh_id;
                        }
                        else if (i == 0 && j == 1) // bottom neighbour
                        {
                            this->neighbours[3] = neigh_id;
                        }
                    }
                }
                else
                {
                    //-----------------------------------------------------------------------------|
                    // NON PERIODIC NEIGHBOURS
                    //-----------------------------------------------------------------------------|
                    // Find potential neighbours
                    int neigh_i = this->sub_j + i; // Find  left and right neighbours
                    int neigh_j = this->sub_i + j; // Find  top and bottom neighbours

                    // If found neighbours exist (ie they are non-negative), find the neightbour id
                    if (neigh_i >= 0 && neigh_i < this->sub_cols && neigh_j >= 0 && neigh_j < this->sub_rows)
                    {
                        int neigh_id = neigh_i + neigh_j * this->sub_cols;
                        if (neigh_id != id)
                        {
                            if (i == -1 && j == 0) // left neighbour
                            {
                                this->neighbours[0] = neigh_id;
                            }
                            else if (i == 0 && j == -1) // top neighbour
                            {
                                this->neighbours[1] = neigh_id;
                            }
                            else if (i == 1 && j == 0) // right neighbour
                            {
                                this->neighbours[2] = neigh_id;
                            }
                            else if (i == 0 && j == 1) // bottom neighbour
                            {
                                this->neighbours[3] = neigh_id;
                            }

                        }
                    }
                }
            }
        }

}


template <class T>
void Domain<T>:: id_to_index(int id, int& id_row, int& id_column)
{
    id_column = id % sub_cols;
    id_row = id / sub_cols;
}

template <class T>
int Domain<T>:: id_from_index(int id_row, int id_column)
{
    if (id_row >= sub_rows || id_row < 0)
        return -1;
    if (id_column >= sub_cols || id_column < 0)
        return -1;

    return id_row * sub_cols + id_column;
}


template <class T>
void Domain<T>::printsubDomain(Domain<T>* Dom)
{
    cout « " Dom->sub_imax" « Dom->sub_imax « " Dom->sub_jmax "« Dom->sub_jmax « endl;
    cout « endl;
    cout « " CURRENT GRID" « endl;
    for (int i = 0; i < Dom->sub_imax; i++)
    {
        for (int j = 0; j < Dom->sub_jmax; j++)
        {
            cout « Dom->current_grid_2d[i][j] « " ";
```

```
        }
        cout « endl;
    }
    cout « endl;

    cout « endl;
    //cout « " NEW GRID" « endl;
    //for (int i = 0; i < Dom->sub_imax; i++)
    //{
    //  for (int j = 0; j < Dom->sub_jmax; j++)
    //  {
    //      cout « Dom->new_grid_2d[i][j] « " ";
    //  }
    //  cout « endl;
    //}
    //cout « endl;
    //cout « endl;

    //cout « " OLD GRID" « endl;
    //for (int i = 0; i < Dom->sub_imax; i++)
    //{
    //  for (int j = 0; j < Dom->sub_jmax; j++)
    //  {
    //      cout « Dom->new_grid_2d[i][j] « " ";
    //  }
    //  cout « endl;
    //}
    //cout « endl;
}


```

4.2 Domain.h

```
#pragma once
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <vector>

#include "Parameters.h"


template <class T>
class Domain
{
public:

    //-----------------------------------------------------------------------------------------|
    // CONSTRUCTORS AND DESTRUCTORS                                                            |
    // ----------------------------------------------------------------------------------------|
    // constructor that assigns pointers
    Domain();
    // constructor that allocates data
    Domain(int n_i, int n_j);
    // destructor that prevents memory leaks in the matrix
    ~Domain();


    //-----------------------------------------------------------------------------------------|
    // LOCAL SUBDOMAIN GRID VARIABLES                                                          |
    // ----------------------------------------------------------------------------------------|
    // Class variables that are pointers to 1-D and 2_D arrays of 3 required sub domains
    // 1-D Arrays
    T* old_grid_1d;
    T* current_grid_1d;
    T* new_grid_1d;
    // 2-D Arrays
    T** old_grid_2d;
    T** current_grid_2d;
    T** new_grid_2d;

    // Sub Domain Parameters-------------------------------------------------------------------|
    // Variables that define maximum size of sub domain
    int sub_imax;
    int sub_jmax;
    // Varibales that define the indexing on the sub domain
    int sub_i;
    int sub_j;
    // Parametris that define the subdivision of the grid
    int sub_rows;
    int sub_cols;

```

```
    //-----------------------------------------------------------------------------------------------|
    // GLOBAL SUBDOMAIN GRID VARIABLES                                                               |
    // ----------------------------------------------------------------------------------------------|
    // Global Domain Parameters  -------------------------------
    // Variables that define maximum size of global domain
    int global_imax;
    int global_jmax;
    // Varibales that define the indexing on the global domain
    int global_i;
    int global_j;

    // Allocate the number of start rows and num-rows
    int* start_row;
    int* num_rows ;
    int* start_column ;
    int* num_columns;

    //-----------------------------------------------------------------------------------------------|
    // GRID FUNCTIONS TO CREATE AND SET DOMAIN PARAMERETS                                            |
    // ----------------------------------------------------------------------------------------------|

    // class method to allocate the data in the domain and sub domains
    void allocatePointers();

    // Create Sub Domains
    void createSubDomains(int p);

    // Class function that sets the subdomain sizes
    void setSubDomainSize();

    //-----------------------------------------------------------------------------------------------|
    // GLOBAL SUBDOMAIN GRID VARIABLES                                                               |
    // ----------------------------------------------------------------------------------------------|
    // Array to store the neighbours for this process
    int neighbours[4];

    // Class function that finds neighbours of the processes
    void findNeighbours();

    // Class functions to convert indices from local to global domains and vice-versa
    void id_to_index(int id, int& id_row, int& id_column);
    int id_from_index(int id_row, int id_column);

    // Function to swap pointers of grid - used after each time step
    void swapGridPointers(T**& subDomain2D_1, T**& subDomain2D_2, T**& subDomain2D_3, T*&
 subDomain1D_1, T*& subDomain1D_2, T*& subDomain1D_3);

    // Functions to print the indiviual process's domain
    void printsubDomain(Domain<T>* Dom);
};

template class Domain<double>;
template class Domain<int>;
```

4.3 Main.cpp

```

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

#include <string>
#include "Parameters.h"
#include "WaveSolver.h"

#include "MPIDatatype.h"
#include "TimingTest.h"

using namespace std;

 //-------------------------------------------------------------------------------------------------------------|
//  DECLARING SOFTWARE PARAMETERS FOR USE IN MAIN

 //-------------------------------------------------------------------------------------------------------------|
// Declaring processor parameters
int id, p;
int tag_num = 0;

```

```
// Declaring spatial grid parameters
int imax, jmax ;
double y_max, x_max, dx, dy;

// Declaring time stepping parameters
double t_max;
double t , dt;

// Declaring parameters to print output to file
double t_out, dt_out;

// Declaring wave equation parameters
double c;

// Declaring boundary condition parameters
int boundary;

//-----------------------------------------------------------------------------------------------------------------------|

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    srand(time(NULL) + id * 10);


//-----------------------------------------------------------------------------------------------------------------------|
    //  READ FROM PARAMETERS TEXT FILE
    //
-----------------------------------------------------------------------------------------------------------------------|

    fstream file;
    file.open("Parameters.txt", std::ios::in);
    // If file is open, read the text in the file and write it out
    if (file.is_open())
    {
        std::string line;
        // Skip the first line with instructions
        getline(file, line);

        // Store variables
        file » imax » jmax » x_max » y_max » t » t_max » dt_out » c » boundary ;
    }

    // Close file
    file.close();

//-----------------------------------------------------------------------------------------------------------------------|

//=======================================================================================================================|
    // START TIMING FOR WHOLE PROCESS

//=======================================================================================================================|
    auto time_total_start = chrono::high_resolution_clock::now();


//-----------------------------------------------------------------------------------------------------------------------|
    // Ceate a domain object
    auto* domain_instance = new Domain<double>();
    // Setup Timing Test object
    auto* Timer = new TimingTest;


//-----------------------------------------------------------------------------------------------------------------------|
    // Set global parameters to our domain
    domain_instance->global_imax = imax;
    domain_instance->global_jmax = jmax;


//=======================================================================================================================|
    // START TIMING FOR DOMAIN DECOMPOSITION

//=======================================================================================================================|
     auto time_domain_start = chrono::high_resolution_clock::now();

//=======================================================================================================================|

    // Find the dimeansions of the sub-domain
    domain_instance->createSubDomains(p);


//-----------------------------------------------------------------------------------------------------------------------|
    // Allocate the pointers for all the grids
    domain_instance->setSubDomainSize();


```

```
//-----------------------------------------------------------------------------------------------------------------------|
   // Allocate memory to the process domains
   domain_instance->allocatePointers();


//-----------------------------------------------------------------------------------------------------------------------|
   // Set Initial Disturbance
   setInitialDisturbance(*domain_instance);
   //fillSubDomainsWithProcessId(*domain_instance);



//-----------------------------------------------------------------------------------------------------------------------|
   // Find the neighbours of the process
   domain_instance->findNeighbours();


//=======================================================================================================================|
   auto time_domain_end = std::chrono::high_resolution_clock::now();
   Timer->domain_setup = time_domain_end - time_domain_start;
   auto count_domain_setup = Timer->domain_setup.count();
   Timer->domain_setup_time.push_back(count_domain_setup);
   //cout « "Domain Setup Time for process " «  id « " is " « count_domain_setup « "s " « endl;
   //Timer->timing_domain_setup(id, p);
//=======================================================================================================================|




//-----------------------------------------------------------------------------------------------------------------------|
   // Calculate time stepping parameters
   dt = 0.1 * min(dx, dy) / c;


//-----------------------------------------------------------------------------------------------------------------------|
   // Set output count and iteration paramters
   int out_cnt = 0, it = 0;


//-----------------------------------------------------------------------------------------------------------------------|
   // Output Initial Disturbance to file and update the time counters
   grid_to_file(*domain_instance, out_cnt, boundary);
   out_cnt++;
   t_out += dt_out;



//-----------------------------------------------------------------------------------------------------------------------|
   //  MPI DATATYPES FOR COMMUNICATIONS
   //
-----------------------------------------------------------------------------------------------------------------------|
//=======================================================================================================================|
   // START TIMING FOR MPI DATATYPES
//=======================================================================================================================|
   auto time_MPI_create_start = chrono::high_resolution_clock::now();
//=======================================================================================================================|

   // Create instance of MPI_Data class to access MPI Datatypes and send and receive functions
   MPI_Data<double>* MData = new MPI_Data<double>();

   // Create MPI Data Types
   MData->CreateMPI_Types(*domain_instance);

   // Setup requests for non-blocking communications
   MPI_Request* request_list;
   request_list = nullptr;



//=======================================================================================================================|
   auto time_MPI_create_end = std::chrono::high_resolution_clock::now();
   Timer->time_MPI_create = time_MPI_create_end - time_MPI_create_start;
   Timer->mpi_create_time.push_back(Timer->time_MPI_create.count());
   //cout « "MPI Create Time for process " «  id « " is " « time_MPI_create.count() « "s " « endl;
   //Timer->timing_MPI_Create_setup(id, p);
//=======================================================================================================================|



   while (t < t_max)
   {

```

```
//-----------------------------------------------------------------------------------------------------------|
        //      START MPI COMMUNICATIONS
//-----------------------------------------------------------------------------------------------------------|

        // Setup counter for requests
        int count = 0;

        // Initialize with 8 requests - 4 forr send and 4 for receive for each process
        request_list = new MPI_Request[8 * p];


//===========================================================================================================|
        // START TIMING FOR MPI COMMUNICATIONS
//===========================================================================================================|
        auto time_MPI_comms_start = chrono::high_resolution_clock::now();
//===========================================================================================================|

//-----------------------------------------------------------------------------------------------------------|
        // Do MPI Communications between the processes to send and receive boundary data
        doMPICommunications(*domain_instance, *MData, count, request_list);



//===========================================================================================================|
        auto time_MPI_comms_end = std::chrono::high_resolution_clock::now();
        Timer->time_MPI_comms = time_MPI_comms_end - time_MPI_comms_start;
        Timer->mpi_comms_time.push_back(Timer->time_MPI_comms.count());
        //cout « "MPI Ccommunication Time for process " «  id « " is " « time_MPI_comms.count() « "s " « endl;
        //Timer->timing_MPI_comms(id, p);
//===========================================================================================================|



//===========================================================================================================|
        // START TIMING FOR  INNER ITERATIONS
//===========================================================================================================|
        auto time_inner_iterations_start = chrono::high_resolution_clock::now();

        // Do Iterations for the Inner Domain of the process grid
        doInnerIterations(*domain_instance);


//===========================================================================================================|
        auto time_inner_iterations_end = std::chrono::high_resolution_clock::now();
        Timer->time_inner_iterations = time_inner_iterations_end - time_inner_iterations_start;
        Timer->inner_iterations_time.push_back(Timer->time_inner_iterations.count());
        //cout « "Average inner iteration time for all process " «  id « " is " « time_inner_iterations.count() « "s " « endl;
        //Timer->timing_inner_iterations(id, p);*/
//===========================================================================================================|


        // Wait for all MPI processes to finish communicating
        MPI_Waitall(count, request_list, MPI_STATUSES_IGNORE);



//===========================================================================================================|
        // START TIMING FOR  BOUNDARY SELECTION AND ITERATIONS
//===========================================================================================================|
        auto time_edge_iterations_start = chrono::high_resolution_clock::now();


//-----------------------------------------------------------------------------------------------------------|
        // If not periodic, set Boundary Conditions depening upon selected input(above)
        if(boundary != 2) // Used to avoid additional checks, however if its periodic, this is automatically negated
            selectBoundary(*domain_instance, boundary); // This is set to neumann, since boundary = 1 by default


        // Do Iterations on the Edgaes
        doBoundaryIterations(*domain_instance);


//===========================================================================================================|
        auto time_edge_iterations_end = std::chrono::high_resolution_clock::now();
        Timer->time_edge_iterations = time_edge_iterations_end - time_edge_iterations_start;
```

```
        Timer->edge_iterations_time.push_back(Timer->time_edge_iterations.count());
        //cout « "Edge iteration time for process " «  id « " is " « time_edge_iterations.count() « "s " « endl;
        //Timer->timing_edge_iterations(id, p);

//==========================================================================================================|



//----------------------------------------------------------------------------------------------------------|
        // Swap pointers of old grid with new grid and new grid with current grid
        domain_instance->swapGridPointers(domain_instance->old_grid_2d, domain_instance->current_grid_2d,
                                          domain_instance->new_grid_2d, domain_instance->old_grid_1d,
                                          domain_instance->current_grid_1d, domain_instance->new_grid_1d);

//----------------------------------------------------------------------------------------------------------|
        // Increment time step
        t += dt;


//----------------------------------------------------------------------------------------------------------|
        // Output the grid contents to files
        if (t_out <= t)
        {
            cout « "output: " « out_cnt « "\tt: " « t « "\titeration: " « it « endl;
            grid_to_file(*domain_instance, out_cnt, boundary);
            out_cnt++;
            t_out += dt_out;
        }


//----------------------------------------------------------------------------------------------------------|
        // Incerement the number of iterations
        it++;


//----------------------------------------------------------------------------------------------------------|
        // Delete pointer  the request list
        delete[] request_list;

    }



//==========================================================================================================|
    auto time_total_end = std::chrono::high_resolution_clock::now();
    Timer->time_total = time_total_end - time_total_start;
    Timer->total_time.push_back(Timer->time_total.count());

    //Timer->timing_total(id, p, imax, jmax);
    if (id == 0)
    {
        cout « " ------------------------------------------------------------------------------" « endl;
        cout « " SIMULATION PARAMETERS  " « endl;
        cout « " ------------------------------------------------------------------------------" « endl;
        cout « " Domain Size : " « imax « " X " « jmax « endl;
        cout « " Total number of processors " « p « endl;
        cout « " The domain divides into" « domain_instance->sub_rows « " and "  « domain_instance->sub_cols « " columns." « endl;
        cout « " Total Time : " « t_max « "s. Total Iterations : " « it « endl;
        cout « " Boundary Type ( 0 = Dirichlit, 1 = Neumann, 2 = Periodic) : " « boundary « endl;
        cout « "------------------------------------------------------------------------------" « endl;
        cout « endl;
        cout « endl;
        cout « " ------------------------------------------------------------------------------" « endl;
        cout « " TIMING ANALYSIS RESULTS " « endl;
        cout « " ------------------------------------------------------------------------------" « endl;
        cout « " Time spech by a process to setup the domain : " « count_domain_setup « endl;
        cout « " Time spent by a process to create MPI Data Types : " « Timer->time_MPI_create.count() « endl;
        cout « " Time spent by a process to carry out MPI communications : " « Timer->time_MPI_comms.count() « endl;
        cout « " Time spent by a process to finish a set of inner iterations : " « Timer->time_inner_iterations.count() « endl;
        cout « " Time spent by a process to finish a set of edge iterations :  " « Timer->time_edge_iterations.count() « endl;
        cout « "Total time for a processor is : " « Timer->time_total.count() « "s. " « endl;
    }

//==========================================================================================================|
```

```


    // End MPI Functions
    MPI_Finalize();
    return(0);
}
```

4.4 MPIDatatype.cpp

```

#include <vector>
#include <mpi.h>

#include "MPIDatatype.h"

using namespace std;

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

template <class T>
MPI_Data<T>::MPI_Data()
{
    // Set MPI Types for the  cells to null
    MPI_Datatype Left_Type = MPI_DATATYPE_NULL;
    MPI_Datatype Right_Type = MPI_DATATYPE_NULL;
    MPI_Datatype Top_Type = MPI_DATATYPE_NULL;
    MPI_Datatype Bottom_Type = MPI_DATATYPE_NULL;

    // Set MPI Types for the Ghost cells to null
    MPI_Datatype Left_Ghost_Type = MPI_DATATYPE_NULL;
    MPI_Datatype Right_Ghost_Type = MPI_DATATYPE_NULL;
    MPI_Datatype Top_Ghost_Type = MPI_DATATYPE_NULL;
    MPI_Datatype Bottom_Ghost_Type = MPI_DATATYPE_NULL;

}


template <class T>
MPI_Data<T>::~MPI_Data()
{
    // deleting regular datatypes
    MPI_Type_free(&Left_Type);
    MPI_Type_free(&Right_Type);
    MPI_Type_free(&Top_Type);
    MPI_Type_free(&Bottom_Type);

    // deleting ghost datatypes
    MPI_Type_free(&Left_Ghost_Type);
    MPI_Type_free(&Right_Ghost_Type);
    MPI_Type_free(&Top_Ghost_Type);
    MPI_Type_free(&Bottom_Ghost_Type);
}

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

template <class T>
void MPI_Data<T>::send_data(Domain<T> &Dom, MPI_Datatype MDataType, int send_to, MPI_Request* request)
{
    // Function that facilitates the ease of sends the MPI Data Type
    MPI_Isend(Dom.current_grid_1d, 1, MDataType, send_to, tag_num, MPI_COMM_WORLD, request);
}


template <class T>
void MPI_Data<T>::recv_data(Domain<T>& Dom, MPI_Datatype MDataType, int receive_from, MPI_Request*
 request)
{
    // Function that facilitates the ease of receives the MPI Data Type
    MPI_Irecv(Dom.current_grid_1d, 1, MDataType, receive_from, tag_num, MPI_COMM_WORLD, request);
}


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

template <class T>
void MPI_Data<T>::CreateMPI_Types(Domain<T>& Dom)
{

    // Use vectors because we dont explicitly know the lengths
```

```
    vector<int> block_lengths;
    vector<MPI_Aint> displacements;
    MPI_Aint add_start;
    vector<MPI_Datatype> typelist;


    // Resize all lengths of vector to Dom.sub_imax-2
    block_lengths.resize(Dom.sub_imax - 2);
    displacements.resize(Dom.sub_imax - 2);
    typelist.resize(Dom.sub_imax - 2);

    // Get the address and set the object to the start of the address
    // Explanation: This is done so that we can send the data by referencing the object
    MPI_Get_address(Dom.current_grid_1d, &add_start);



    //
---------------------------------------------------------------------------------------------------------------------
    //  MPI_DATATYPE 1 : LEFT_TYPE MPI - Regular
    //
---------------------------------------------------------------------------------------------------------------------
    // Set each of the dataType variables
    // to the correct addresses depending on the location
    // in the array

    // Initialize the variables
    for (int i = 0; i < Dom.sub_imax - 2; i++)
    {
        typelist[i] = MPI_DOUBLE;
        block_lengths[i] = 1;
        MPI_Get_address(&Dom.current_grid_2d[i + 1][1], &displacements[i]);
        displacements[i] = displacements[i] - add_start;

    }
    // Create the dataType
    MPI_Type_create_struct((Dom.sub_imax - 2), block_lengths.data(), displacements.data(),
typelist.data(), &Left_Type);
    // Commit to use
    MPI_Type_commit(&Left_Type);



    //
---------------------------------------------------------------------------------------------------------------------
    //  MPI_DATATYPE 2 : LEFT_TYPE MPI - GHOST
    //
---------------------------------------------------------------------------------------------------------------------
    for (int i = 0; i < Dom.sub_imax - 2; i++)
    {
        typelist[i] = MPI_DOUBLE;
        block_lengths[i] = 1;
        MPI_Get_address(&Dom.current_grid_2d[i + 1][0], &displacements[i]);
        displacements[i] = displacements[i] - add_start;
    }
    // Create the dataType
    MPI_Type_create_struct(Dom.sub_imax - 2, block_lengths.data(), displacements.data(),
typelist.data(), &Left_Ghost_Type);
    // Commit to use
    MPI_Type_commit(&Left_Ghost_Type);



    //
---------------------------------------------------------------------------------------------------------------------
    //  MPI_DATATYPE 3 :RIGHT_TYPE MPI - Regular
    //
---------------------------------------------------------------------------------------------------------------------
    // The only difference is that the address to store in the
    // 2D array is [i][Dom.sub_jmax-1]

    // In+ialize the variables for all the grids
    for (int i = 0; i < Dom.sub_imax - 2; i++)
    {
        typelist[i] = MPI_DOUBLE;
        block_lengths[i] = 1;
        MPI_Get_address(&Dom.current_grid_2d[i + 1][Dom.sub_jmax - 2], &displacements[i]);
        displacements[i] = displacements[i] - add_start;
    }
    // Create the dataType
    // Note: We use .data() because they are vectors
    MPI_Type_create_struct((Dom.sub_imax - 2), block_lengths.data(), displacements.data(),
typelist.data(), &Right_Type);
    // Commit to use
    MPI_Type_commit(&Right_Type);


```

```
    //
    //  MPI_DATATYPE 4 :RIGHT_TYPE MPI - Ghost
    //
    // Initialize the variables for the ghost right grids
    for (int i = 0; i < Dom.sub_imax - 2; i++)
    {
        typelist[i] = MPI_DOUBLE;
        block_lengths[i] = 1;
        MPI_Get_address(&Dom.current_grid_2d[i + 1][Dom.sub_jmax - 1], &displacements[i]);
        displacements[i] = displacements[i] - add_start;
    }
    // Create the dataType
    // Note: We use .data() because they are vectors
    MPI_Type_create_struct((Dom.sub_imax - 2), block_lengths.data(), displacements.data(), typelist.data(), &Right_Ghost_Type);
    // Commit to use
    MPI_Type_commit(&Right_Ghost_Type);



    //
    //  TOP AND BOTTOM
    //
    // Note: the top and bottom are stored in contiguous memory so dont need
    // to be sent as different types
    //



    //
    // MPI_DATATYPE 5 : TOP - REGULAR
    //

    // Initialize variables
    typelist[0] = MPI_DOUBLE;
    block_lengths[0] = Dom.sub_jmax - 2;
    MPI_Get_address(&Dom.current_grid_2d[1][1], &displacements[0]);
    displacements[0] = displacements[0] - add_start;
    // Create the dataType
    MPI_Type_create_struct(1, block_lengths.data(), displacements.data(), typelist.data(), &Top_Type);
    // Commit to use
    MPI_Type_commit(&Top_Type);


    //
    // MPI_DATATYPE 6 : TOP - GHOST
    //
    // Initialize variables
    typelist[0] = MPI_DOUBLE;
    block_lengths[0] = Dom.sub_jmax - 2;
    MPI_Get_address(&Dom.current_grid_2d[0][1], &displacements[0]);
    displacements[0] = displacements[0] - add_start;
    // Create the dataType
    MPI_Type_create_struct(1, block_lengths.data(), displacements.data(), typelist.data(), &Top_Ghost_Type);
    // Commit to use
    MPI_Type_commit(&Top_Ghost_Type);



    //
    // MPI_DATATYPE 7 : BOTTOM - REGULAR
    //
    // Initialize variables
    typelist[0] = MPI_DOUBLE;
    block_lengths[0] = Dom.sub_jmax - 2;
    MPI_Get_address(&Dom.current_grid_2d[Dom.sub_imax - 2][1], &displacements[0]);
    displacements[0] = displacements[0] - add_start;
    // Create the dataType
    MPI_Type_create_struct(1, block_lengths.data(), displacements.data(), typelist.data(), &Bottom_Type);
    // Commit to use
    MPI_Type_commit(&Bottom_Type);
```

```
    //
    // MPI_DATATYPE 8 : BOTTOM - GHOST
    //
    // Initialize variables
    typelist[0] = MPI_DOUBLE;
    block_lengths[0] = Dom.sub_jmax - 2;
    MPI_Get_address(&Dom.current_grid_2d[Dom.sub_imax - 1][1], &displacements[0]);
    displacements[0] = displacements[0] - add_start;
    // Create the dataType
    MPI_Type_create_struct(1, block_lengths.data(), displacements.data(), typelist.data(), &Bottom_Ghost_Type);
    // Commit to use
    MPI_Type_commit(&Bottom_Ghost_Type);

}
```

4.5 MPIDatatype.h

```

#pragma once
#include <mpi.h>
#include <vector>
#include "Domain.h"


template <class T>
class MPI_Data
{
public:
    //-----------------------------------------------------------------------------------------------|
    //  CONSTRUCTORS AND DESTRUCTORS                                                                 |
    // ----------------------------------------------------------------------------------------------|
    // Constructor
    MPI_Data();
    // Destructor to avoid explicitly deleting the memory
    ~MPI_Data();


    //-----------------------------------------------------------------------------------------------|
    //  MPI DATATYPES FOR COMMUNICATIONS                                                             |
    // ----------------------------------------------------------------------------------------------|
    // MPI TYPES for the regualr rows
    MPI_Datatype Left_Type, Right_Type, Top_Type, Bottom_Type;
    // MPI Types for the Ghost rows
    MPI_Datatype Left_Ghost_Type, Right_Ghost_Type, Top_Ghost_Type, Bottom_Ghost_Type;


    //-----------------------------------------------------------------------------------------------|
    //  CREATE MPI DATA TYPES                                                                        |
    // ----------------------------------------------------------------------------------------------|
    // function to create all data types
    void CreateMPI_Types(Domain<T>& Dom);

    //-----------------------------------------------------------------------------------------------|
    // FUNCTIONS FOR COMMUNICATIONS                                                                  |
    // ----------------------------------------------------------------------------------------------|
    void send_data(Domain<T>& Dom, MPI_Datatype MDataType, int neigh_to_send, MPI_Request* request);
    void recv_data(Domain<T>& Dom, MPI_Datatype MDataType, int neigh_to_receive, MPI_Request* request);

};

template class MPI_Data<double>;
template class MPI_Data<int>;
```

4.6 Parameters.h

```
#pragma once


// Processor Parameters
extern int id, p;
extern int tag_num;

// Domain Parameters - grid and spacing
```

```
extern int imax , jmax ;
extern double y_max , x_max , dx, dy;

// Time Stepping Parameters
extern double t_max;
extern double t, t_out, dt_out , dt;

// Wave Equation parameters
extern double c;

// Boundary Conditions
extern int boundary;

// Initial Conditions
extern int boundary;
```

4.7 SerialWaveSolver.cpp

```
//#define _USE_MATH_DEFINES
//
//#include <iostream>
//#include <sstream>
//#include <fstream>
//#include <vector>
//#include <cmath>
//
//using namespace std;
//
//vector<vector<double» grid, new_grid, old_grid;
//int imax = 100, jmax = 100;
//double t_max = 30.0;
//double t, t_out = 0.0, dt_out = 0.04, dt;
//double y_max = 10.0, x_max = 10.0, dx, dy;
//double c = 1;
//
//void grid_to_file(int out)
//{
//  stringstream fname;
//  fstream f1;
//  fname « "./output/serial/output" « "_" « out « ".dat";
//  f1.open(fname.str().c_str(), ios_base::out);
//  for (int i = 0; i < imax; i++)
//  {
//      for (int j = 0; j < jmax; j++)
//          f1 « grid[i][j] « "\t";
//      f1 « endl;
//  }
//  f1.close();
//}
//
//void do_iteration(void)
//{
//  //  Updating inner part of the domain
//  for (int i = 1; i < imax - 1; i++)
//      for (int j = 1; j < jmax - 1; j++)
//          new_grid[i][j] = pow(dt * c, 2.0) * ((grid[i + 1][j] - 2.0 * grid[i][j] + grid[i - 1][j])
 / pow(dx, 2.0) + (grid[i][j + 1] - 2.0 * grid[i][j] + grid[i][j - 1]) / pow(dy, 2.0)) + 2.0 *
 grid[i][j] - old_grid[i][j];
//
//  // Updating the outer columns of the domain
//  for (int i = 0; i < imax; i++)
//  {
//      new_grid[i][0] = new_grid[i][1];
//      new_grid[i][jmax - 1] = new_grid[i][jmax - 2];
//  }
//
//  // Updating the rows of the domain
//  for (int j = 0; j < jmax; j++)
//  {
//      new_grid[0][j] = new_grid[1][j];
//      new_grid[imax - 1][j] = new_grid[imax - 2][j];
//  }
//
//  // Updating the time-step
//  t += dt;
//
//  old_grid.swap(new_grid); // swap elements of old grid with new grid
//  old_grid.swap(grid); //
//}
//
//int main(int argc, char* argv[])
//{
//  old_grid.resize(imax, vector<double>(jmax));
```

```
//  grid.resize(imax, vector<double>(jmax));
//  new_grid.resize(imax, vector<double>(jmax));
//
//  dx = x_max / ((double)imax - 1);
//  dy = y_max / ((double)imax - 1);
//
//  t = 0.0;
//
//  dt = 0.1 * min(dx, dy) / c;
//
//  int out_cnt = 0, it = 0;
//
//  grid_to_file(out_cnt);
//  out_cnt++;
//  t_out += dt_out;
//
//
//  //sets half sinusoidal intitial disturbance - this is brute force - it can be done more elegantly
//  double r_splash = 1.0;
//  double x_splash = 3.0;
//  double y_splash = 3.0;
//  for (int i = 1; i < imax - 1; i++)
//      for (int j = 1; j < jmax - 1; j++)
//      {
//          double x = dx * i;
//          double y = dy * j;
//
//          double dist = sqrt(pow(x - x_splash, 2.0) + pow(y - y_splash, 2.0));
//
//          if (dist < r_splash)
//          {
//              double h = 5.0 * (cos(dist / r_splash * M_PI) + 1.0);
//
//              grid[i][j] = h;
//              old_grid[i][j] = h;
//          }
//      }
//
//  while (t < t_max)
//  {
//      do_iteration();
//
//      if (t_out <= t)
//      {
//          cout « "output: " « out_cnt « "\tt: " « t « "\titeration: " « it « endl;
//          grid_to_file(out_cnt);
//          out_cnt++;
//          t_out += dt_out;
//      }
//
//      it++;
//  }
//  return 0;
//}
```

4.8 TimingTest.cpp

```
#include <vector>
#include<iostream>

#include "TimingTest.h"

using namespace std;

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

void TimingTest::timing_domain_setup(int id, int p)
{
    if (id == 0)
    {
        double avg_domain_time = 0;
        for (int i  = 0; i < domain_setup_time.size(); i++)
        {
            avg_domain_time += domain_setup_time[i];
        }
        avg_domain_time = avg_domain_time / p;

        cout « "The Average Domain Setup Time for all the processes is " « avg_domain_time « "s. " « endl;
    }
}

```

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

void TimingTest::timing_MPI_Create_setup(int id, int p)
{
    if (id == 0)
    {
        double avg_MPI_Create_time = 0;
        for (int i = 0; i < mpi_create_time.size(); i++)
        {
            avg_MPI_Create_time += mpi_create_time[i];
        }
        avg_MPI_Create_time = avg_MPI_Create_time / p;

        cout « "The Average MPI Create Time for all the processes is " « avg_MPI_Create_time « "s. " « endl;
    }

}

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

void TimingTest::timing_MPI_comms(int id, int p)
{
    if (id == 0)
    {
        double avg_MPI_comms_time = 0;
        for (int i = 0; i < mpi_comms_time.size(); i++)
        {
            avg_MPI_comms_time += mpi_comms_time[i];
        }
        avg_MPI_comms_time = avg_MPI_comms_time / p;

        cout « "The Average MPI Communication Time for all the processes is " « avg_MPI_comms_time « "s. " « endl;
    }

}

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

void TimingTest::timing_edge_iterations(int id, int p)
{
    if (id == 0)
    {
        double avg = 0;
        for (int i = 0; i < edge_iterations_time.size(); i++)
        {
            avg += edge_iterations_time[i];
        }
        avg = avg / p;

        cout « "The Average Edge Iteration Time for all the processes is " « avg « "s. " « endl;
    }

}

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

void TimingTest::timing_inner_iterations(int id, int p)
{
    if (id == 0)
    {
        double avg = 0;
        for (int i = 0; i < inner_iterations_time.size(); i++)
        {
            avg += inner_iterations_time[i];
        }
        avg = avg / p;

        cout « "The Average Inner Iteration for all the processes is " « avg « "s. " « endl;
    }

}

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

void TimingTest::timing_total(int id, int p, int imax, int jmax)
{
    if (id == 0)
    {
        double avg = 0;
        for (int i = 0; i < this->total_time.size(); i++)
```

```
        {
            avg += total_time[i];
        }
        avg = avg / p;
        cout « "The Average Total time for all the processes is " « avg « "s. " « endl;
    }

}

//
```

4.9 TimingTest.h

```
#pragma once
#include <vector>
#include <chrono>

using namespace std;

class TimingTest
{
public:
    // Vector to store time spent in setting up the domain for each process and creating MPI_DataTypes
    vector<double> domain_setup_time;

    // Vector to store total run time for all iterations
    vector<double> total_time;

    // Vector to store timings for a single iteration
    vector<double> mpi_create_time;

    // Timing to doMPICommunications
    vector<double> mpi_comms_time;

    // Timing to complete innerIterations
    vector<double> inner_iterations_time;


    // Timings to complete set boundary conditions and complete outerIterations
    vector<double> edge_iterations_time;

    chrono::duration < double> time_MPI_create;
    chrono::duration <double> time_total;
    chrono::duration < double> domain_setup;
    chrono::duration <double> time_MPI_comms;
    chrono::duration < double> time_inner_iterations;
    chrono::duration < double> time_edge_iterations;

    //-----------------------------------------------------------------------------|
    //  FUNCTION DEALING TIMING DOMAIN SETUP
    // ----------------------------------------------------------------------------|
    void timing_domain_setup(int id, int p);


    //-----------------------------------------------------------------------------|
    //  FUNCTIONS DEALING WITH MPI CREATION AND COMMUNICATIONS
    // ----------------------------------------------------------------------------|
    void timing_MPI_Create_setup(int id, int p);

    void timing_MPI_comms(int id, int p);

    //-----------------------------------------------------------------------------|
    //  FUNCTION DEALING TIMING TIME STEP ITERATIONS
    // ----------------------------------------------------------------------------|

    void timing_inner_iterations(int id, int p);

    void timing_edge_iterations(int id, int p);
    //-----------------------------------------------------------------------------|
    //  FUNCTION DEALING TIMING TOTAL TIME FOR ALL PROCESSES
    // ----------------------------------------------------------------------------|

    void timing_total(int id, int p, int imax, int jmax);

};
```

4.10 WaveSolver.cpp

```
```

```
#define _USE_MATH_DEFINES
#include<iostream>
#include<string>
#include<mpi.h>
#include <sstream>
#include <fstream>
#include <cmath>


#include "WaveSolver.h"
#include "Parameters.h"

using namespace std;

void readWaveSolverParameters(int &i , int &j , double &x_m, double &y_m, double &tee, double &t_m,double &dt_o , double &cee , int &bo,double &dist, double *disturbance_p)
{

    stringstream data;
    data « "Parameters.txt";
    ifstream file;

    // Open Parameters file
    file.open(data.str().c_str(), ios_base::in);

    // Read in value by value
    if (file.good())
    {
        file » i » j » x_m » y_m » t » t_m » dt_o » c » bo » dist;
    }

    // Initialize distrbance array with 3 parameters(radius, x loc, y loc)
    disturbance_p = new double[dist * 3];

    while (file.good())
    {
        // Fill te disturbance array with the 3 parameters
        for (int i = 0; i < 3 * dist; i++)
        {
            file » disturbance_p[i];
        }
    }
    file.close();
    cout « i « j;
}

// ---------------------------------------------------------------------------------------------------|
// Select Boundary
void selectBoundary(Domain<double> &Dom, int boundary)
{
    if (boundary == 0)
    {
        dirichlitBoundary(Dom);
    }
    if (boundary == 1)
    {
        neumannBouandry(Dom);
    }

}

// Set Dirichlit Boundry Conditions
void dirichlitBoundary(Domain<double>& Dom)
{
    // If the left neighbours dont exist
    if (Dom.neighbours[0] == -1)
    {
        // Set left boundaries
        for (int i = 1; i < Dom.sub_imax - 1; i++)
        {
            Dom.current_grid_2d[i][0] = 0;
        }
    }

    // If the right neighbours dont exist
    if (Dom.neighbours[2] == -1)
    {
        // Set right boundaries
        for (int i = 1; i < Dom.sub_imax - 1; i++)
        {
            Dom.current_grid_2d[i][Dom.sub_jmax - 1] = 0;
        }
    }

    //If the top neighbours dont exist
    if (Dom.neighbours[1] == -1)
    {
```

```
        // Set top boundaries
        for (int i = 1; i < Dom.sub_jmax - 1; i++)
        {
            Dom.current_grid_2d[1][i] = 0;
        }
    }

    // If the bottom neighbours dont exist
    if (Dom.neighbours[3] == -1)
    {
        // Set bottom boundaries
        for (int i = 1; i < Dom.sub_jmax - 1; i++)
        {
            Dom.current_grid_2d[Dom.sub_imax - 1][i] = 0;
        }
    }
}

// Set Neumann Boundary Conditions
void neumannBouandry(Domain<double>& Dom)
{

    // If the left neighbours dont exist
    if (Dom.neighbours[0] == -1)
    {
        // Set left boundaries
        for (int i = 1; i < Dom.sub_imax - 1; i++)
        {
            Dom.current_grid_2d[i][1] = Dom.current_grid_2d[i][2];
        }
    }

    // If the right neighbours dont exist
    if (Dom.neighbours[2] == -1)
    {
        // Set right boundaries
        for (int i = 1; i < Dom.sub_imax - 1; i++)
        {
            Dom.current_grid_2d[i][Dom.sub_jmax - 1] = Dom.current_grid_2d[i][Dom.sub_jmax - 2];
        }
    }

    //If the top neighbours dont exist
    if (Dom.neighbours[1] == -1)
    {
        // Set top boundaries
        for (int i = 1; i < Dom.sub_jmax - 1; i++)
        {
            Dom.current_grid_2d[1][i] = Dom.current_grid_2d[2][i];
        }
    }

    // If the bottom neighbours dont exist
    if (Dom.neighbours[3] == -1)
    {
        // Set bottom boundaries
        for (int i = 1; i < Dom.sub_jmax - 1; i++)
        {
            Dom.current_grid_2d[Dom.sub_imax-1][i] = Dom.current_grid_2d[Dom.sub_imax - 2][i];
        }
    }
}


// ------------------------------------------------------------------------------------------------|
// Create Disturbance
void setInitialDisturbance(Domain<double>& Dom)
{

        // Create an array to store x,y and radius for each disturbance
        // Calculate the grid spacing
        dx = x_max / (Dom.global_imax - 1);
        dy = y_max / (Dom.global_jmax - 1);

        double x_splash = 3;
        double y_splash = 3;
        double r_splash = 1;

        // Start looping on the non-padded region
        for (int i = 1; i < Dom.sub_imax - 1; i++)
        {
            for (int j = 1; j < Dom.sub_jmax - 1; j++)
            {

                // map global to local
                int I = (i - 1) + Dom.global_i;
                int J = (j - 1) + Dom.global_j;
```

```

                // set the grid jumps
                double x = dx * I;
                double y = dy * J;

                // find the distance of a point from the splash
                double dist = sqrt(pow(x - x_splash, 2.0) + pow(y - y_splash, 2.0));
                double h = 1;

                // if the distance is within splash range, set the vlaues to calculated h
                if (dist < r_splash)
                {
                    // calculate wave height
                    h = 5.0 * (cos(dist / r_splash * M_PI) + 1.0);

                    //cout « " h " « h « " i " « i « " j " « j « " Dom.current_grid_2d[i][j] " « Dom.current_grid_2d[i][j] « endl;

                    // set the grid values
                    Dom.current_grid_2d[i][j] = h;
                    Dom.old_grid_2d[i][j] = h;
                }

                /*  if (id == 5)
                    {
                        cout « " h " « h « " i " « i « " j " « j « " Dom.current_grid_2d[i][j] " « Dom.current_grid_2d[i][j] « endl;
                    }*/
            }
        }

    //if (id == 0)
    //{
    //  cout « " OLD GRID AFTER DISTURBANCE " « endl;
    //  for (int i = 1; i < Dom.sub_imax - 1; i++)
    //  {
    //      for (int j = 1; j < Dom.sub_jmax - 1; j++)
    //      {
    //          cout « Dom.old_grid_2d[i][j] « " ";
    //      }
    //      cout « endl;
    //  }
    //  cout « endl;
    //}
}


void fillSubDomainsWithProcessId(Domain<double>& Dom)
{
    for (int i = 0; i < Dom.sub_imax; i++)
    {
        for (int j = 0; j < Dom.sub_jmax; j++)
        {
            Dom.current_grid_2d[i][j] = id;
        }
    }
}




// ---------------------------------------------------------------------------------------------|
// Function to do the communications
void doMPICommunications(Domain<double> &domain_instance,  MPI_Data<double> &MData, int count, MPI_Request *request_list)
{

    //---------------------------------------------------------------------------------------------|
    //      SEND FROM REGULAR BOUNDARIES AND RECEIVE INTO GHOST BOUNDARIES
    // ---------------------------------------------------------------------------------------------|
    // Send the regualar cells of the sub-domain to the other processes

    if (domain_instance.neighbours[0] != -1)
    {
        // Receive left data in ghost cells
        MData.recv_data(domain_instance, MData.Left_Ghost_Type, domain_instance.neighbours[0], &request_list[count]);
        count++;

        // Send left Data
        MData.send_data(domain_instance, MData.Left_Type, domain_instance.neighbours[0], &request_list[count]);
        count++;
    }


    if (domain_instance.neighbours[2] != -1)
```

```
    {
        // Receive right data in ghost cells
        MData.recv_data(domain_instance, MData.Right_Ghost_Type, domain_instance.neighbours[2], &request_list[count]);
        count++;

        // Send right data
        MData.send_data(domain_instance, MData.Right_Type, domain_instance.neighbours[2], &request_list[count]);
        count++;
    }

        if (domain_instance.neighbours[1] != -1)
        {
            // Receive top data in ghost cells
            MData.recv_data(domain_instance, MData.Top_Ghost_Type, domain_instance.neighbours[1], &request_list[count]);
            count++;

            // Send top data
            MData.send_data(domain_instance, MData.Top_Type, domain_instance.neighbours[1], &request_list[count]);
            count++;
        }


        if (domain_instance.neighbours[3] != -1)
        {
            // Receive bottom data in ghost cells
            MData.recv_data(domain_instance, MData.Bottom_Ghost_Type, domain_instance.neighbours[3], &request_list[count]);
            count++;

            // Send bottom data
            MData.send_data(domain_instance, MData.Bottom_Type, domain_instance.neighbours[3], &request_list[count]);
            count++;
        }

    if(boundary == 2)
    {
            //  ADDITIONAL PERIODIC COMMUNICATIONS ON THE TOP AND BOTTOM
            // Send top data
            MData.send_data(domain_instance, MData.Top_Type, domain_instance.neighbours[1], &request_list[count]);
            count++;

            // Receive bottom data in ghost cells
            MData.recv_data(domain_instance, MData.Bottom_Ghost_Type, domain_instance.neighbours[1], &request_list[count]);
            count++;


            // Receive top data in ghost cells
            MData.recv_data(domain_instance, MData.Top_Ghost_Type, domain_instance.neighbours[1], &request_list[count]);
            count++;

            // Send bottom data
            MData.send_data(domain_instance, MData.Bottom_Type, domain_instance.neighbours[3], &request_list[count]);
            count++;

    }
    MPI_Waitall(count, request_list, MPI_STATUSES_IGNORE);

}


// ---------------------------------------------------------------------------------------------------------
// Solve inner domain
void doInnerIterations(Domain<double> &Dom)
{
    // Iterate over the innder domain in both i and j directions
    for (int i = 2; i < Dom.sub_imax - 1; i++)
    {
        for (int j = 2; j < Dom.sub_jmax - 1; j++)
        {
            // Descretization of equation of the wave for time stepping in the inner domain
            Dom.new_grid_2d[i][j] = pow(dt * c, 2.0)
                * ((Dom.current_grid_2d[i + 1][j]
```

```
                        - 2.0 * Dom.current_grid_2d[i][j]
                        + Dom.current_grid_2d[i - 1][j]) / pow(dx, 2.0)

                + (Dom.current_grid_2d[i][j + 1]
                    - 2.0 * Dom.current_grid_2d[i][j]
                    + Dom.current_grid_2d[i][j - 1]) / pow(dy, 2.0))

                + 2.0 * Dom.current_grid_2d[i][j] - Dom.old_grid_2d[i][j];
        }
    }
}

// Solve outer domain
void doBoundaryIterations(Domain<double>& Dom)
{
    // LEFT BOUNDARY
    if (Dom.neighbours[0] != -1)
    {
        for (int i = 1; i < Dom.sub_imax-1; i++)
        {
            Dom.new_grid_2d[i][1] = pow(dt * c, 2.0) *
                ((Dom.current_grid_2d[i + 1][1]
                    - 2.0 * Dom.current_grid_2d[i][1]
                    + Dom.current_grid_2d[i - 1][1]) / pow(dx, 2.0)

                    + (Dom.current_grid_2d[i][2]
                        - 2.0 * Dom.current_grid_2d[i][1]
                        + Dom.current_grid_2d[i][0]) / pow(dy, 2.0))

                + 2.0 * Dom.current_grid_2d[i][1] - Dom.old_grid_2d[i][1];
        }
    }

    // TOP BOUNDARY
    if (Dom.neighbours[1] != -1)
    {
        for (int j = 1; j < Dom.sub_jmax -1; j++)
        {
            Dom.new_grid_2d[1][j] = pow(dt * c, 2.0)
                * ((Dom.current_grid_2d[2][j]
                    - 2.0 * Dom.current_grid_2d[1][j]
                    + Dom.current_grid_2d[0][j]) / pow(dx, 2.0)

                + (Dom.current_grid_2d[1][j + 1]
                    - 2.0 * Dom.current_grid_2d[1][j]
                    + Dom.current_grid_2d[1][j - 1]) / pow(dy, 2.0))

                + 2.0 * Dom.current_grid_2d[1][j] - Dom.old_grid_2d[1][j];
        }
    }

    // RIGHT BOUNDARY
    if (Dom.neighbours[2] != -1)
    {
        for (int i = 1; i < Dom.sub_imax - 1; i++)
        {
            Dom.new_grid_2d[i][Dom.sub_jmax-1] = pow(dt * c, 2.0)
                * ((Dom.current_grid_2d[i + 1][Dom.sub_jmax-1]
                    - 2.0 * Dom.current_grid_2d[i][Dom.sub_jmax - 1]
                    + Dom.current_grid_2d[i - 1][Dom.sub_jmax - 1]) / pow(dx, 2.0)

                + (Dom.current_grid_2d[i][Dom.sub_jmax - 1]
                    - 2.0 * Dom.current_grid_2d[i][Dom.sub_jmax - 1]
                    + Dom.current_grid_2d[i][Dom.sub_jmax - 2]) / pow(dy, 2.0))

                + 2.0 * Dom.current_grid_2d[i][Dom.sub_jmax - 1] - Dom.old_grid_2d[i][Dom.sub_jmax -
    1];
        }

    }

    // BOTTOM BOUNDARY
    if (Dom.neighbours[3] != -1)
    {
        for (int j = 1; j < Dom.sub_jmax-1; j++)
        {
            Dom.new_grid_2d[Dom.sub_imax-1][j] = pow(dt * c, 2.0)
                * ((Dom.current_grid_2d[Dom.sub_imax - 1][j]
                    - 2.0 * Dom.current_grid_2d[Dom.sub_imax - 1][j]
                    + Dom.current_grid_2d[Dom.sub_imax - 2][j]) / pow(dx, 2.0)

                + (Dom.current_grid_2d[Dom.sub_imax - 1][j + 1]
                    - 2.0 * Dom.current_grid_2d[Dom.sub_imax - 1][j]
                    + Dom.current_grid_2d[Dom.sub_imax - 1][j - 1]) / pow(dy, 2.0))

                + 2.0 * Dom.current_grid_2d[Dom.sub_imax - 1][j] - Dom.old_grid_2d[Dom.sub_imax -
    1][j];
```

```
        }
    }

}


// Output files after iterations for postprocessing
void grid_to_file(Domain<double>& Dom, int out, int boundary)
{

    // Function that uses fstream to write variable to the file
    stringstream fname;
    fstream f1;

    // Outputting into folders according to the boundary condition and number
    if (boundary == 1)
    {
        fname « "./output/neumann/"« p «"_processors/"«"output_" « out « "_id_" « id  « ".dat";
    }
    else if (boundary == 0)
    {
        fname « "./output/dirichlit/" « p « "_processors/" « "output_" « out « "_id_" « id « ".dat";
    }
    else if (boundary == 2)
    {
        fname « "./output/periodic/" « p « "_processors/" « "output_" « out « "_id_" « id « ".dat";
    }

    // Open and write contents of each processor to the respective files
    f1.open(fname.str().c_str(), ios_base::out);

    for (int i = 1; i < Dom.sub_imax - 1; i++)
    {
        // Writing contents of processsor grid to file
        for (int j = 1; j < Dom.sub_jmax - 1; j++)
            f1 « Dom.current_grid_2d[i][j] « "\t";
        f1 « endl;
    }
    f1.close();
}
```

4.11 WaveSolver.h

```
#pragma once
#define _USE_MATH_DEFINES
#include "MPIDatatype.h"
#include "Domain.h"


//-----------------------------------------------------------------------------|
//  FUNCTIONS DEALING WITH BOUNDRY CONDITIONS
// ----------------------------------------------------------------------------|
void selectBoundary(Domain<double>& Dom, int boundary);

void dirichlitBoundary(Domain<double>& Dom);

void neumannBouandry(Domain<double>& Dom);


//-----------------------------------------------------------------------------|
//  FUNCTIONS DEALING WITH COMMUNICATIONS
// ----------------------------------------------------------------------------|

// Dirichlit and Neumann Communications-------------------------------------------|
void doMPICommunications(Domain<double>& domain_instance,
    MPI_Data<double>& MData, int count, MPI_Request* request_list);


//-----------------------------------------------------------------------------|
// FUNCTION TO SET INITIAL CONDITION
// ----------------------------------------------------------------------------|
void setInitialDisturbance(Domain<double> &Dom);
// Array pointer to store thenumber of splashes


//-----------------------------------------------------------------------------|
// FUNCTIONS DEALTING WITH TIME STEPPING
// ----------------------------------------------------------------------------|
// Solve inner domain
void doInnerIterations(Domain<double>&Dom);

```

```
// Solve outer domain
void doBoundaryIterations(Domain<double> &Dom);


//-----------------------------------------------------------------------------|
// TEST AND POSTPROCESSING FUNCTIONS
// ----------------------------------------------------------------------------|
void fillSubDomainsWithProcessId(Domain<double>& Dom);

void grid_to_file(Domain<double>& Dom, int out, int boundary);
```