High Performance Technical Computing

Assignment

Author:
Wiktor Bednarek

Supervisor:
Dr Irene Moulitsas

January 31, 2017

Assignment part: 2450 Words

Abstract

Modern computers allows to calculate very complex problems in short amount of time. However it can be noticed that modern processors changed way of increasing their computational power. For last 10 years instead of increasing clock frequency in processors, engineers decided to increase number of cores in each processor.

This approach brings huge amount of new possibilities but, also there are many new aspects that programmers who are making multi-core software need to remember about,

In this paper will be presented potential of parallel programing based on following Computational fluid dynamics schemes:

- Explicit Upwind
- Crank-Nicolson
- Implicit Upwind

Results given from parallel software will be compared with the same schemes, but written in sequential style. Appropriate charts will be given and parallel approach used in those three schemes is going to be covered and explained.

Contents

1. Introduction

Computational fluid dynamics is a branch of science with really practical usage when it comes to analyze and solve problems related to fluid flows. Many mathematical solutions could be found in many publications and books like *On the partial difference equations of mathematical* [1].

Nevertheless when calculations are needed it is noticeable that Computational fluid dynamics (CFD) schemes are really complex. In spite of those problems for modern computers, it seems to be relatively easy to solve those schemes, they could be computed in reasonable time (in seconds for 10000 points), still using modern technology and new approach of software development, problems like those can be solved much faster.

High performance computing based on MPI will be covered in this paper. Results of calculating three schemes will be given. For this report is it important to indicate growth in performance (decrease in calculation time) for each scheme.

Parallel approach for each scheme will be explained. It is very important to write parallel code properly. That kind of software is much more sensitive for even small mistakes comparing to single core programming. Idea of solution is crucial, wrong conception can even reduce performance comparing to single core software.

Dependency between number of processors and performance growth is going to be shown. At the end in conclusions I will try to explain output results.

2. Used methods

2.1. Initial conditions

In this solution following one dimensional advection equation is used:

$$\frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} = 0 \tag{1}$$

Above equation is used for motion of one-dimesional description, u stands for speed.

To solve all schemes it is needed to have initial boundaries conditions. Two sets of conditions are considered:

Space domain for differential equation:

$$x \in [-50,50] \tag{2}$$

Initialization function is space is based on exponential function, and initial boundary conditions are:

$$f(x,0) = \frac{1}{2}e^{-x^2} \quad (3)$$

$$f(-50,t) = 0 \quad (4)$$

$$f(50,t) = 0 \quad (5)$$

In this report following analytical solution is given:

$$f(x,0) = \frac{1}{2}e^{-(x-1.75t)^2} \quad (6)$$

Speed condition for this solution:

$$u = 1.75 \quad (7)$$

2.2. Explicit Upwind Scheme

Equation below describes first order upwind scheme:

$$\frac{f_i^{n+1} - f_i^n}{\Delta t} + u\frac{f_i^n - f_{i-1}^n}{\Delta x} = 0 \quad if \quad u > 0 \quad (8)$$

After transformation equation (8) is:

$$f_i^{n+1} = (f_i^n - C(f_i^n - f_{i-1}^n) \quad (9)$$

This is how implementation equation looks in my program.

Courant number used in software described in this report was chosen according to stability conditions [1] described in equation (12):

$$C \leq 1 \quad (10)$$

Where

$$C = u\frac{\Delta t}{\Delta x} \quad (11)$$

In real program Courant number was set to 0.999 and 0.5 value has been tested as well.

Below is description of parallel approach used in software created for this paper reasons.

Initialization

In this program to store matrix proper Matrix class based on vector of vector has been crated.
For initialization functions following approach was done:

1. Splitting number of points by number of processors that are in use (demanded)
2. Each processor initializes its own chunk of Matrix with function (3). It is done parallel.
3. Processor with rank 0 is filling time points with “0” values for first space value – condition (4)
4. Processor with the highest rank is filling time points with “0” values for last space value – condition (5)

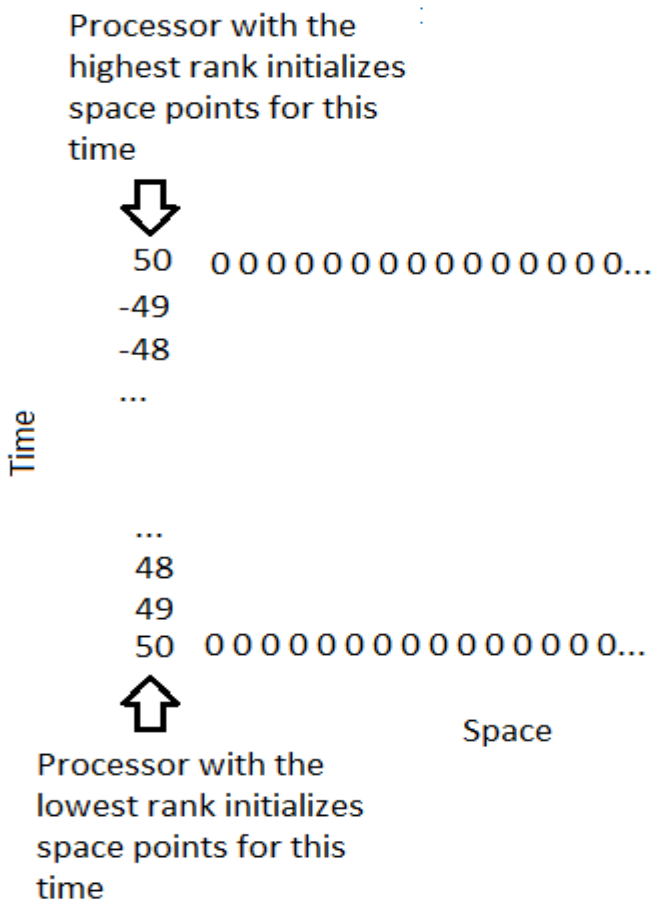

Figure 1 Visualization of initialization of two equations (4) and (5)

Computation

1. In loop of time points 3 steps are performed:
    a. When rank of processor ≠ 0 receive f_{i-1}^{n} which is send when last processs ≠ rank of processor

    b. Calculate in this loop Explicit Upwind Scheme using revived value

    c. In separate loop which will be performed for each processor – every one of them will calculate its own matrix chunk calculate Explicit Upwind Scheme as for single processor

2.3. Implicit Upwind Scheme

Second scheme is Implicit Upwind Scheme which has characteristic unconditional stability [2]. What is important thanks to good code scalability, most of Implicit Upwind Scheme implementation is similar to Explicit Upwind scheme. This is possible mostly because of using two dimensional matrix in this paper code. On the other hand it has bad influence for performance, comparing to solution when I would decide to use and perform the calculations on one dimensional data.

Below equation describes Implicit Upwind Scheme:

$$\frac{f_i^{n+1}-f_i^n}{\Delta t} + u\frac{f_i^{n+1}-f_{i-1}^{n+1}}{\Delta x} = 0 \qquad (12)$$

After transformations we get:

$$f_i^{n+1} = -C(f_i^{n+1} - f_{i-1}^{n+1}) + f_i^n \qquad (13)$$

Finally following equation was used in solution code:

$$f_i^{n+1} = \frac{(-f_i^n - C f_{i-1}^n)}{-(1+C)} \qquad (14)$$

Initialization

Initialization for this scheme remains the same as in Explicit Upwind Scheme. Boundaries are the same.

Computation

2. As it was mentioned before implementation in Implicit Upwind Scheme is slightly different form Explicit scheme. Only steps modification in this case are changes in equation. Changes from previous scheme are highlighted.

    a. When rank of processor ≠ 0 receive f_{i-1}^{n} which is send when last process ≠ rank of processor

    b. Calculate in this loop Implicit Upwind Scheme using revived value

    c. In separate loop which will be performed for each processor – every one of them will calculate its own matrix chunk calculate Implicit Upwind Scheme as for single processor

2.4. Crank-Nicolson

Crank-Nicolson is the second order trapezoidal method.

One dimensional heat equation is as follow:

$$\frac{f_i^{n+1}-f_i^n}{\Delta t} = \frac{f_{i+1}^n-2f_i^n+f_{i-1}^n}{2(\Delta x)^2} + \frac{f_{i+1}^{n+1}-2f_i^{n+1}+f_{i-1}^{n+1}}{2(\Delta x)^2} \qquad (15)$$

Finally equation is:

$$\frac{C}{4} f_{i+1}^{n+1} + f_i^{n+1} - \frac{C}{4} f_{i-1}^{n+1} = \frac{C}{4} f_{i+1}^{n} + f_i^{n} - \frac{C}{4} f_{i-1}^{n} \qquad (16)$$

3. Results

3.1. Explicit Upwind Scheme

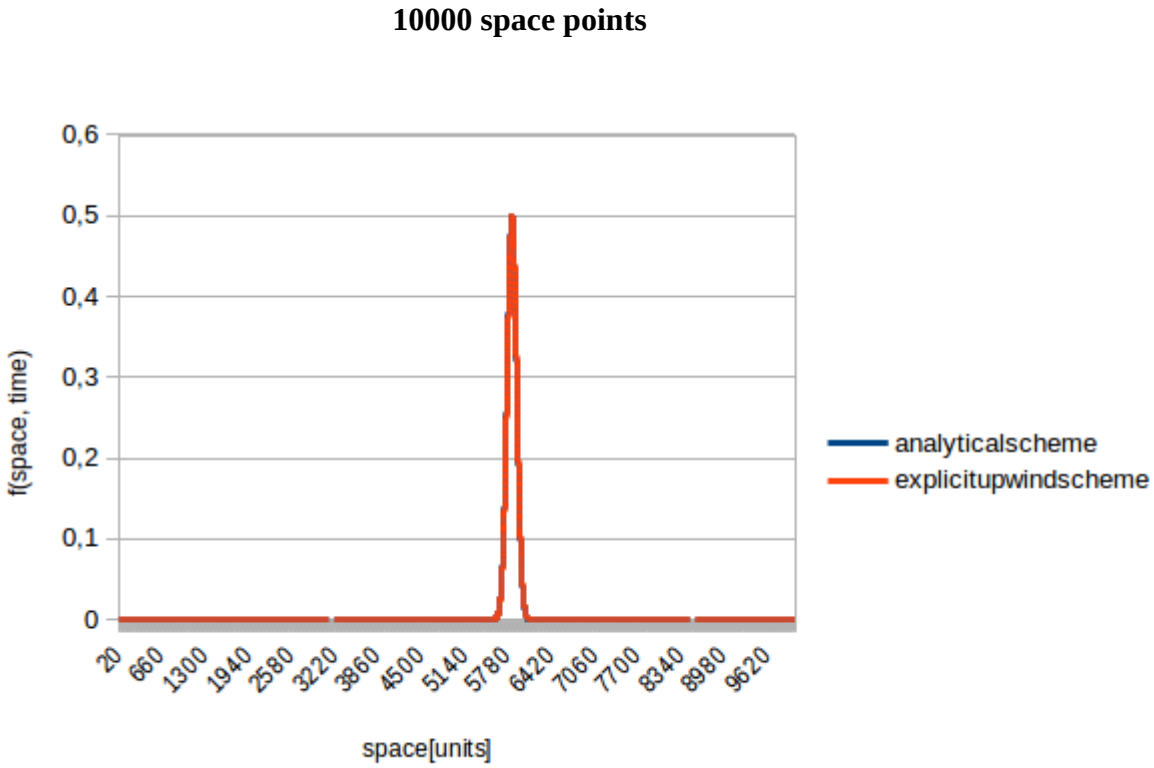

Figure 2 Comparing Analytical solution and Explicit Upwind Scheme parallel results for exp type of initial boundary t=5, CFL = 0.999, number of points = 10000

Because chart for that huge amount of points is unreadable it is necessary to check norms for this case.

Table 1 Norms values depending on Courant number in Explicit Upwind Scheme. Data results for t=5, CFL = { 0.25 ,0.5, 0.75, 0.999}, number of points = 10000.

Courant Number	Infinite norm	Norm one	Norm two
0,999	0,1132	0,0037407	0,0178942

Norm values are really small. That's why on the chart we are not able to observe anything. One chart is covered by another.

The most important thing to check is performance. In order to measure it MPI_time() function has been used. Below comparison between single processor performance and parallel.

Table 2 Performance depending on number of processors in Explicit Upwind Scheme. Data results for t=5, CFL = 0.999, number of points = 10000.

Number of cores	Solution paralell	Solution single
4	0,448562	0,718439
8	0,345987	0,718439
16	0,304498	0,718439

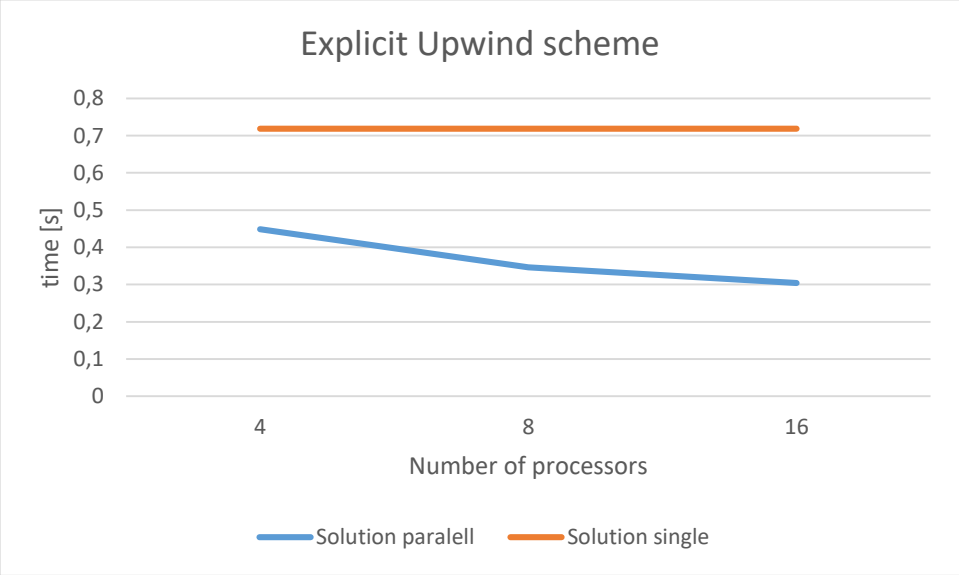

Table 3 Explicit Upwind Scheme and Explicit Upwind Scheme parallel performance results depending on number of processors results for exp type of initial boundary t=5, CFL = 0.999, number of points = 10000

There can be observed noticeable growth in performance for bigger number of processors in Explicit Upwind Scheme, but the most significant difference occurs for small number of processors. Growth in performance is not increase linearly. Probably number of communications is vary in even that relatively small amount of processors. Additionally despite having more processors and more “truncated” parts of matrix it doesn’t mean automatically that program will work faster times numbers of processors. Even if in truncated matrix there is less calculation for one processor, communication may delay program and not only calculations are processors work, but also initialization, checking condition etc. Sometimes when variables are assigned to many times it could consume more computing resources than calculations itself.

3.2. Implicit Upwind Scheme

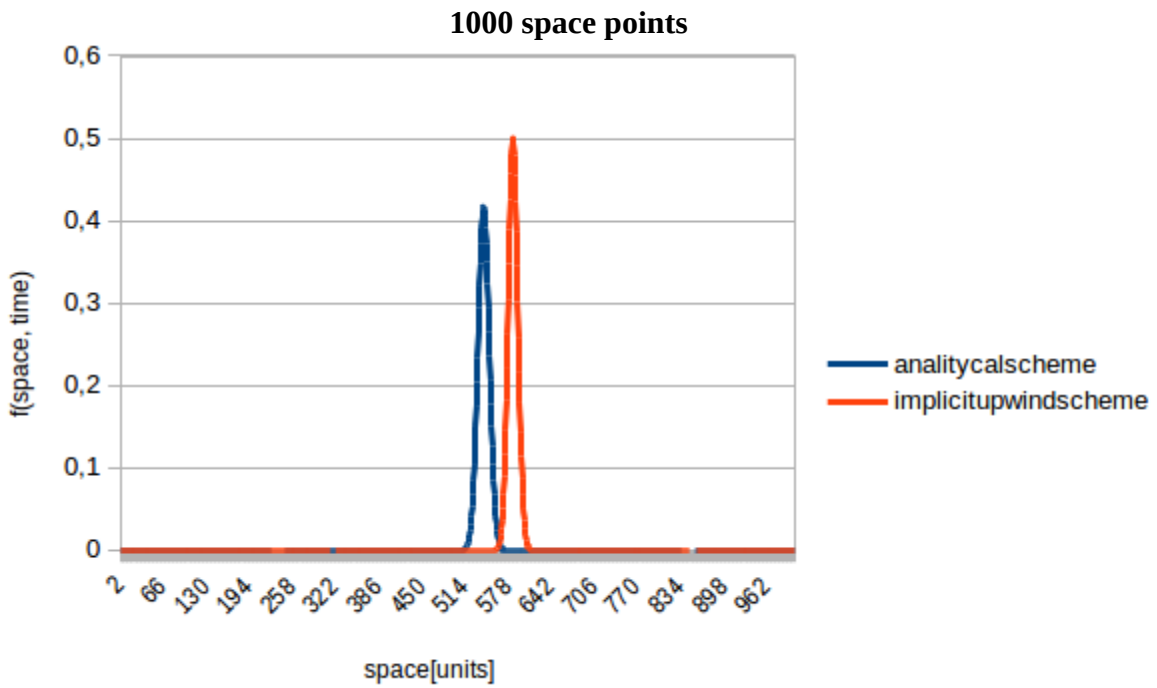

Figure 3 Comparing Analytical solution and Implicit Upwind Scheme parallel results for exp type of initial boundary t=5, CFL = 0.999, number of points = 1000

This time for 1000 space points it is more possible to observe change. For better details look at table.

Table 4 Norms values depending on Courant number in Implicit Upwind Scheme. Data results for t=5, CFL =0.999, number of points = 10000.

Courant Number	Infinite norm	Norm one	Norm two
0,999	0,2351	0,0056913	0,025987

Table 5 Performance depending on number of processors in Implicit Upwind Scheme. Data results for t=5, CFL = 0.999, number of points = 1000.

Number of cores	Solution paralell	Solution single
2	0.0019597	0.007095
4	0.00107694	0.007095
8	0.000781059	0.007095
16	0.000674498	0.007095

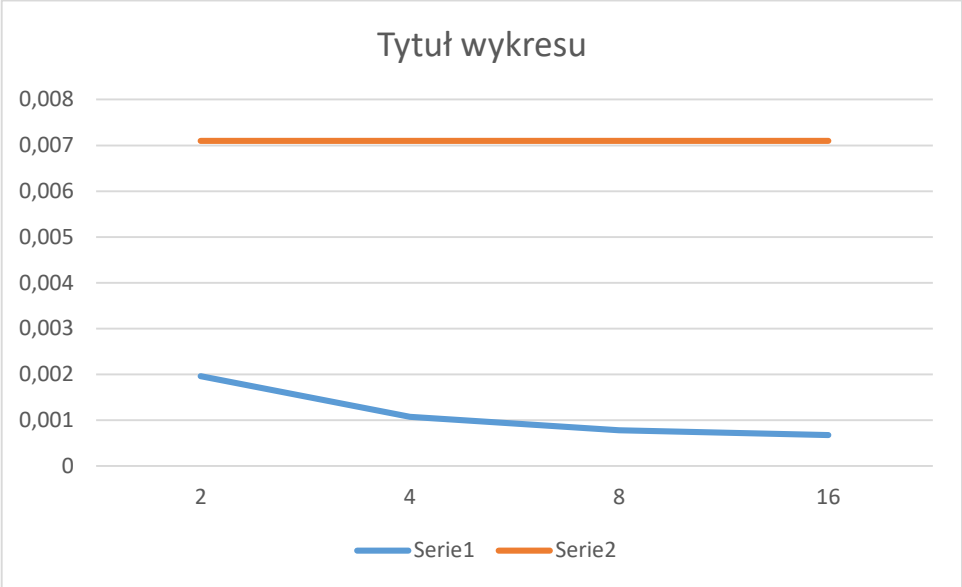

Table 6 Implicit Upwind Scheme and Implicit Upwind Scheme parallel performance results depending on number of processors results for exp type of initial boundary t=5, CFL = 0.999, number of points = 1000

10000 space points

Table 7 Performance depending on number of processors in Implicit Upwind Scheme. Data results for t=5, CFL = 0.999, number of points = 10000.

Number of cores	Solution paralell	Solution single
2	0.37183	0.866349
4	0.10365	0.866349
8	0.06613	0.866349
16	0.04624	0.866349

Obviously more space points consumes more computational resources and time of execution is longer than in 1000 points case. Increase in performance depending on number of processors is similar to 1000 points case. Is not linear but acceleration is evidently noticeable comparing to single core solution.

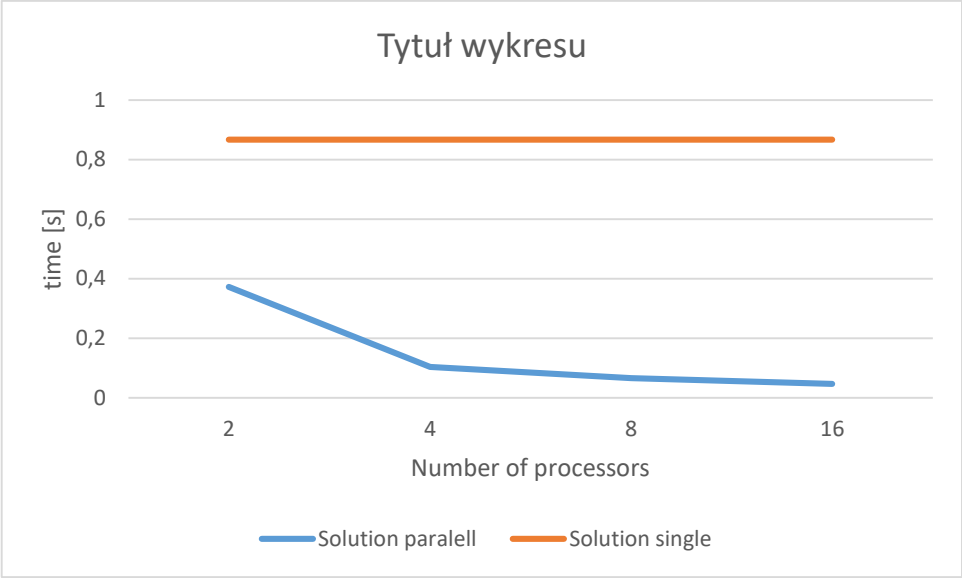

Figure 4 Implicit Upwind Scheme and Implicit Upwind Scheme parallel performance results depending on number of processors results for exp type of initial boundary t=5, CFL = 0.999, number of points = 10000

3.3. Crank-Nicolson Scheme

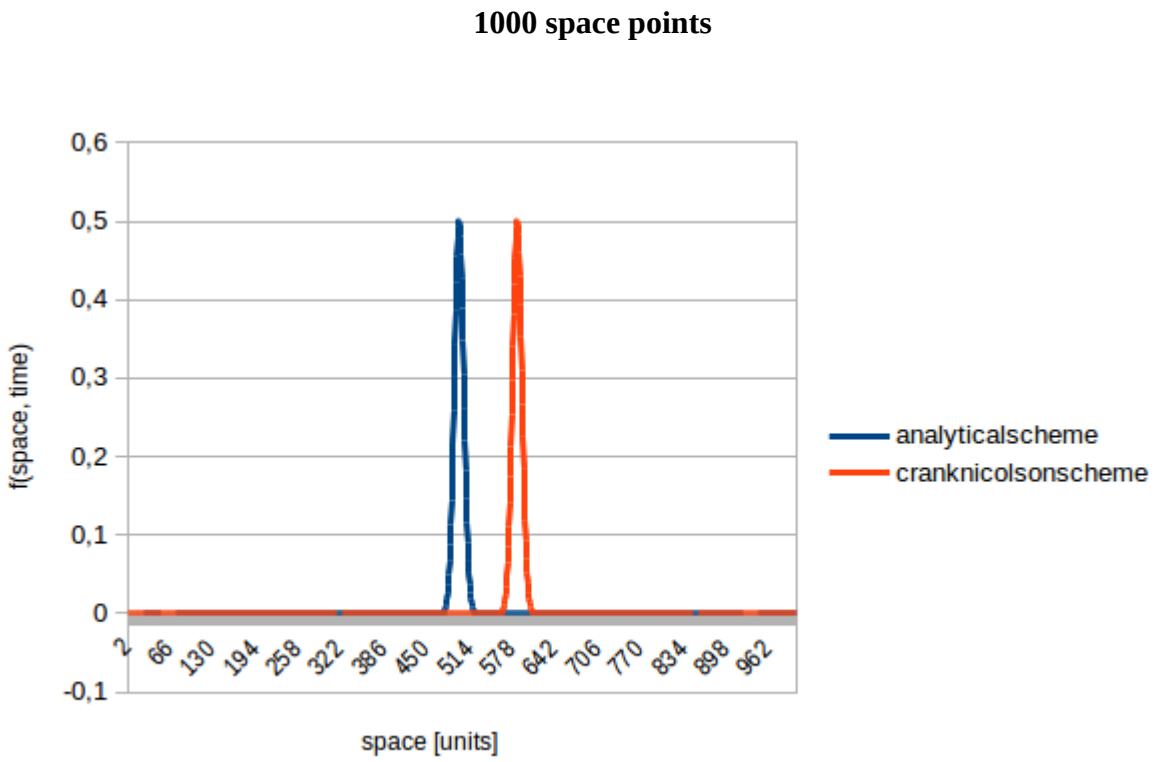

Figure 5 Comparing Analytical solution and Crank-Nicolson scheme parallel results for exp type of initial boundary t=5, CFL = 0.999, number of points = 1000

Table 8 Performance depending on number of processors in Crank-Nicolson Scheme. Data results for t=5, CFL = 0.999, number of points = 1000

Number of cores	Solution paralell	Solution single
2	0.01240	0.013201
4	0.00912	0.013201
8	0.00890	0.013201
16	0.00875	0.013201

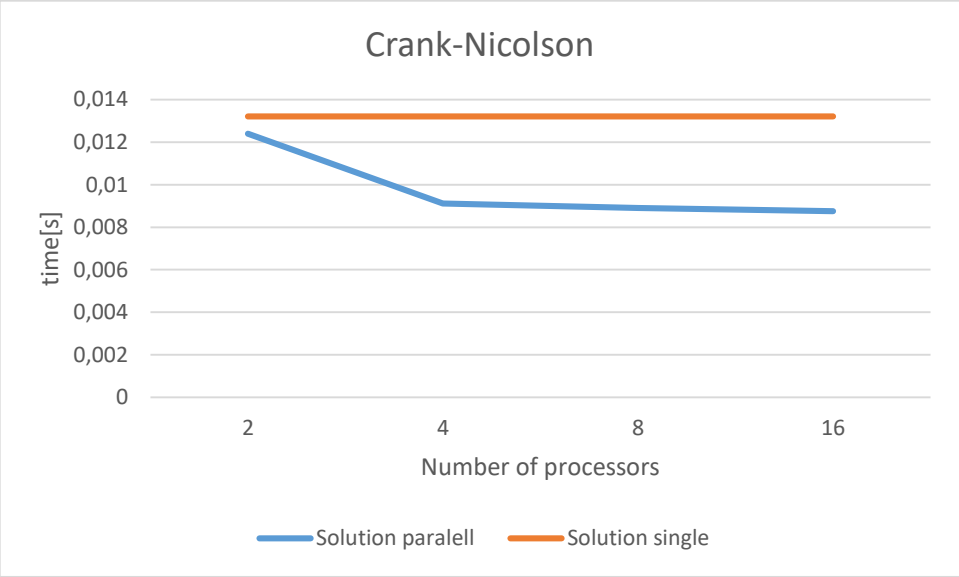

Figure 6 and Crank-Nicolson Scheme and and Crank-Nicolson Scheme parallel performance results depending on number of processors results for exp type of initial boundary t=5, CFL = 0.999, number of points = 1000

For 1000 with Crank-Nicolson Scheme we can observe big performance improvement for small number of processors (2 and 4). For 8 and 16 processors growth is really slow, most likely, because complicated communication in Crank-Nicolson Scheme.

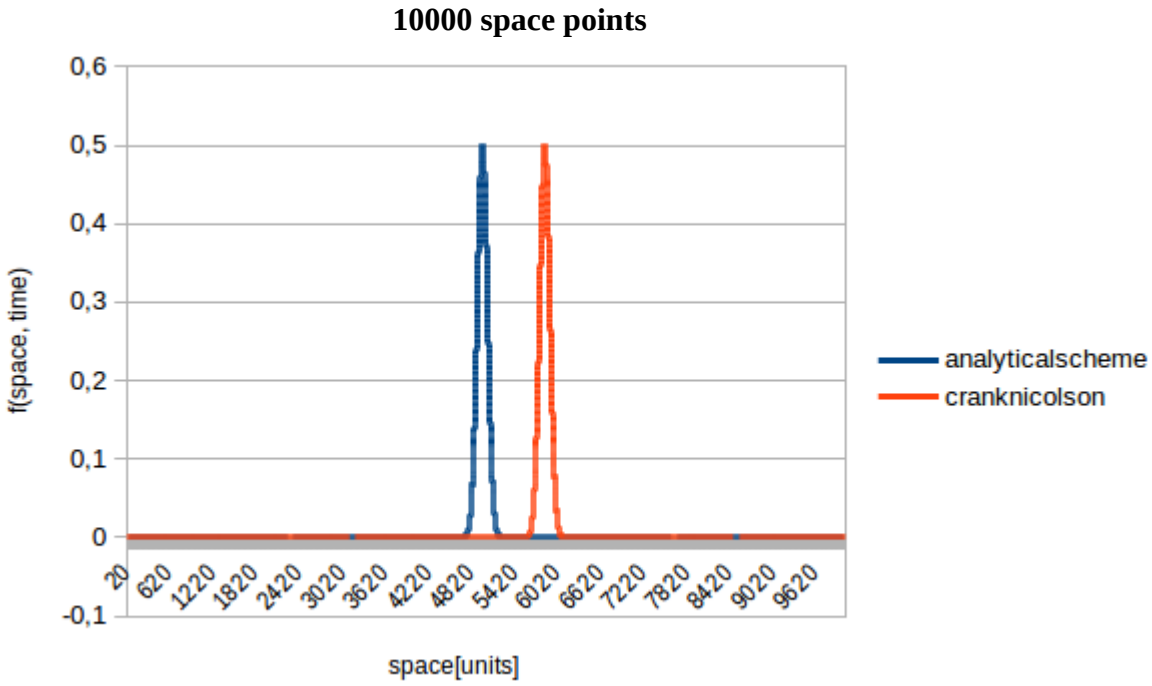

*Figure 7 Comparing Analytical solution and **Crank-Nicolson scheme** parallel results for exp type of initial boundary t=5, CFL = 0.999, number of points = 1000*

Table 9 Performance depending on number of processors in Crank-Nicolson Scheme. Data results for t=5, CFL = 0.999, number of points = 10000

Number of cores	Solution paralell	Solution single
2	0.80163	0.812644
4	0.81354	0.812644
8	0.84240	0.812644
16	0.91880	0.812644

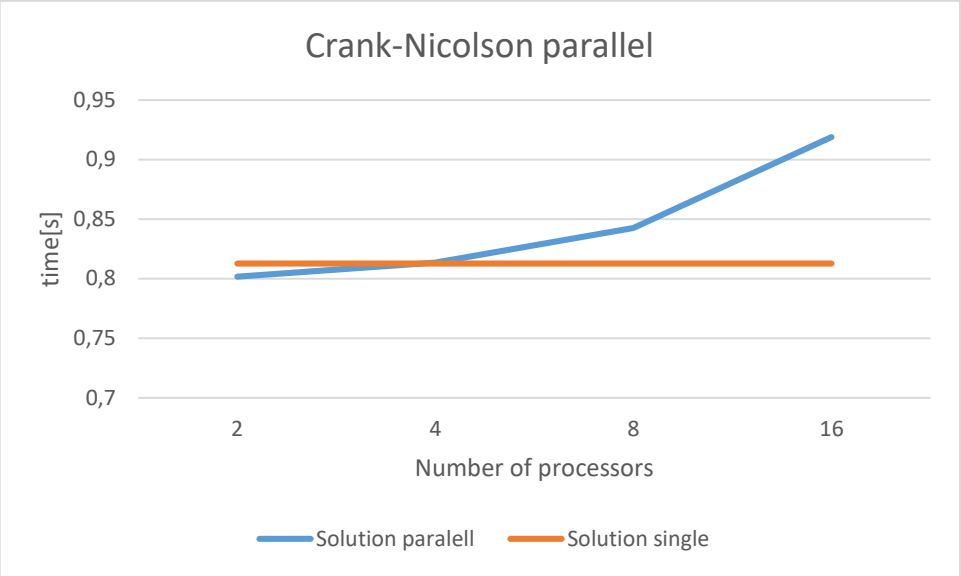

Figure 8 Crank-Nicolson and Crank-Nicolson parallel performance results depending on number of processors results for exp type of initial boundary t=5, CFL = 0.999, number of points = 10000

When it comes to Crank-Nicolson Scheme with 10000 points, bigger number of processors apparently slowing down the program. Tendency of time executing is growing when number processors amount is rising. So when previously with 1000 points Crank-Nicolson Scheme was slightly growth with performance when number of processors increased, but in this case performance even goes down.
That confirm my previous assumption, because of complicated communication not always parallelize Crank-Nicolson scheme will be profitable.

3.4. Third party library

For this report purpose I decided to choose third party library Intel® Math Kernel Library ScaLAPACK[1]. First test showed that for small amount of processors performance is similar to that you can find in software which was made for this paper purposes. ScaLAPACK library is more universal and my solutions applies mostly for given problem. This library is really solid one and we can expect good performance from its function. Besides it has also official intel support.
The fact why results are not so good as mine is that my solution is focused on one particular problem.

[1] https://software.intel.com/en-us/node/521455

4. Conclusions

In all cases parallelization of given scheme gave huge acceleration for program execution. Programs work much faster when they are optimized/written for multiprocessors purposes. Of course increase in performance is not linear and measurements done for this paper purpose confirm that assumption.

In Explicit Upwind scheme observations give conclusions that for small amount of processors like 2 or 4 we get almost expected increase in performance, which means program runs 2 or 4 faster. For higher amount of processors like 8 or 16 communication issues start to have influence on performance. It is worth to notice that for big amount of matrix chunks calculation itself may not consume the biggest amount of computing power. Sometimes initializations, conditions checking or assigning variables from one to another may consume more computing power than mathematical calculations itself.

Implicit Upwind Scheme gave better performance results than Explicit Upwind Scheme. Modifications in code in order to get Implicit Scheme are not complicated, logic of communication works exactly the same.

About communication, this is one of the biggest disadvantages of MPI. Naturally it is necessary and without it there is no parallel code, but when it comes to debugging it is really inconvenient. MPI library and compiler are constructed to show in its errors, address of error, but no line of code where potential error could be found. That makes process of finding errors much longer comparing with sequential debugging process.

In Crank-Nicolson Scheme case for big number of points, performance even decreased. It happened because that scheme has really complicated communication and each components communicates a lot of time with others, that brings delay and lower performance.

Nevertheless this paper measurements and experiments show how much, efforts of making code parallel increases performance and confirms that parallel solutions will still gain in value in the future.

References

[1] R. Courant, K. Friedrichs, H. Lewy, *On the partial difference equations of mathematical physics*, IBM Journal, March 1967, pp. 215-234.

[2] G. Cohen, Higher-Order Numerical Methods for Transient Wave equation, Springer-Verlag, 2001

[3] R. M. M. Mattheij et al., *The design and implementation of the parallel out-of-core scalapack LU QR and Cholesky Factorizationj Routines*

[4] George Em Karniadakis, Robert M. Kirby II Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and |Their Implementation

APPENDIX

main.cpp

```
#include <mpi.h>
#include <iostream>
#include <vector>
#include <stdio.h>
#include <iterator>
#include <sstream>
#include <memory>
#include <unistd.h>
#include <map>
#include <iomanip>
#include "MathFunctions.h"
#include "GeneralScheme.h"
#include "ExplicitUpwindScheme.h"
#include "Display.h"
#include "ExplicitUpwindParallel.h"
#include "ImplicitParallel.h"
#include "ImplicitUpwindScheme.h"
#include "CrankParallel.h"


using std::vector;
using std::cin;
using std::cout;
using std::endl;
//Own display namespace with user friendly displaying functions
using namespace display;

//Decimal separator Template
template<typename CharT>
class DecimalSeparator : public std::numpunct<CharT> {
public:
    DecimalSeparator(CharT Separator)
            : m_Separator(Separator) {}

protected:
    CharT do_decimal_point() const {
        return m_Separator;
    }

private:
    CharT m_Separator;
};


//Displaying vector
template<typename T>
std::ostream &operator<<(std::ostream &out, const std::vector<T> &v) {
    std::copy(v.begin(), v.end(), std::ostream_iterator<T>(out, "\n"));
    return out;
}

std::string getCurrentPath() {
    char charCurrentPath[1024];
    std::string path;
    if (getcwd(charCurrentPath, sizeof(charCurrentPath)) != NULL) {
```

```
        path = std::string(charCurrentPath);
    } else
        perror("getcwd() error");


    return path;

}


void runSchemes(int numberOfBoundaryConditionSet, vector<double> initialSettings,
std::string typeOfExtension) {

    //Create Results folder first in your project directory!
    std::string path = getCurrentPath() + "/Results/";


    GeneralScheme general = GeneralScheme(initialSettings[0], initialSettings[1],
initialSettings[2],
                                          initialSettings[3], initialSettings[4]);
    general.solve(numberOfBoundaryConditionSet);

    ExplicitUpwindScheme explicitScheme(initialSettings[0], initialSettings[1],
initialSettings[2], initialSettings[3],
                                        initialSettings[4]);
    explicitScheme.solve(numberOfBoundaryConditionSet);

    ExplicitUpwindParallel parallel(initialSettings[0], initialSettings[1],
initialSettings[2], initialSettings[3],
                                    initialSettings[4]);
    parallel.solve(numberOfBoundaryConditionSet);


    ImplicitUpwindScheme implicitUpwindScheme(initialSettings[0], initialSettings[1],
initialSettings[2],
                                              initialSettings[3], initialSettings[4]);
    implicitUpwindScheme.solve(numberOfBoundaryConditionSet);

    ImplicitParallel implicitParallel(initialSettings[0], initialSettings[1],
initialSettings[2], initialSettings[3],
                                      initialSettings[4]);
    implicitParallel.solve(numberOfBoundaryConditionSet);

    CrankParallel crankParallel(initialSettings[0], initialSettings[1],
initialSettings[2], initialSettings[3],
                                initialSettings[4]);
    crankParallel.solve(numberOfBoundaryConditionSet);




    //Generating open files
    std::ofstream osGeneralScheme;
    std::ofstream osExplicitScheme;
    std::ofstream osImplicitScheme;


    std::ofstream osParallel;
    std::ofstream osImplicitParallel;
    std::ofstream osCrankParallel;
```

```
    //Operation helps to plot charts in programs such as Exel. Setting type of decimal
separator depending on current geographical location. In some countries comma in
default separator in numbers in others dot
    osGeneralScheme.imbue(std::locale(std::cout.getloc(), new
DecimalSeparator<char>(',')));
    osExplicitScheme.imbue(std::locale(std::cout.getloc(), new
DecimalSeparator<char>(',')));
    osImplicitScheme.imbue(std::locale(std::cout.getloc(), new
DecimalSeparator<char>(',')));


    osParallel.imbue(std::locale(std::cout.getloc(), new
DecimalSeparator<char>(',')));
    osImplicitParallel.imbue(std::locale(std::cout.getloc(), new
DecimalSeparator<char>(',')));
    osCrankParallel.imbue(std::locale(std::cout.getloc(), new
DecimalSeparator<char>(',')));



    std::stringstream streams[initialSettings.size()];

    for (unsigned int i = 0; i < initialSettings.size(); ++i) {
        streams[i] << (int) initialSettings[i];
    }

    //Because I use c++98 I need to convert strings to chars
    std::string generalSchemeFileName =
            path + getInitialBoundaryConditionName(numberOfBoundaryConditionSet) + "_"
+ general.getName() +
            "Results_t=" + streams[2].str() + "_points=" + streams[3].str() + "_CFL="
+ streams[4].str() +
            typeOfExtension;
    const char *CharGeneralSchemeFileName = generalSchemeFileName.c_str();

    std::string UpwindSchemeFileName =
            path + getInitialBoundaryConditionName(numberOfBoundaryConditionSet) + "_"
+ explicitScheme.getName() +
            "Results_t=" + streams[2].str() + "_points=" + streams[3].str() + "_CFL="
+ streams[4].str() +
            typeOfExtension;
    const char *CharUpwindSchemeFileName = UpwindSchemeFileName.c_str();

    std::string ExplicitParallel =
            path + getInitialBoundaryConditionName(numberOfBoundaryConditionSet) + "_"
+ parallel.getName() +
            "Results_t=" + streams[2].str() + "_points=" + streams[3].str() + "_CFL="
+ streams[4].str() +
            typeOfExtension;
    const char *CharExplicitParallelFileName = ExplicitParallel.c_str();

    std::string implicitParallelFileName =
            path + getInitialBoundaryConditionName(numberOfBoundaryConditionSet) + "_"
+ implicitParallel.getName() +
            "Results_t=" + streams[2].str() + "_points=" + streams[3].str() + "_CFL="
+ streams[4].str() +
            typeOfExtension;
    const char *charImplicitParallelFileName = implicitParallelFileName.c_str();

    std::string implicitFileName =
```

```
            path + getInitialBoundaryConditionName(numberOfBoundaryConditionSet) + "_"
+
            implicitUpwindScheme.getName() +
            "Results_t=" + streams[2].str() + "_points=" + streams[3].str() + "_CFL="
+ streams[4].str() +
            typeOfExtension;
    const char *charImplicitFileName = implicitFileName.c_str();

    std::string crankFileName =
            path + getInitialBoundaryConditionName(numberOfBoundaryConditionSet) + "_"
+
            crankParallel.getMethodName() +
            "Results_t=" + streams[2].str() + "_points=" + streams[3].str() + "_CFL="
+ streams[4].str() +
            typeOfExtension;
    const char *charcrankFileName = crankFileName.c_str();




    //Single
    osGeneralScheme.open(CharGeneralSchemeFileName);
    osExplicitScheme.open(CharUpwindSchemeFileName);
    osImplicitScheme.open(charImplicitFileName);

    //Parallel
    osParallel.open(CharExplicitParallelFileName);
    osImplicitParallel.open(charImplicitParallelFileName);
    osCrankParallel.open(charcrankFileName);



    //Saving schemes calculated results, only one processor can do that

    int myRank;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    if (myRank == 0) {
        osGeneralScheme << general.getResults();

        //Explicit
        osExplicitScheme << explicitScheme.getLastExplicitMatrixColumn();
        osParallel << parallel.getGatherResults();

        //Implicit
        osImplicitParallel << implicitParallel.getImplicitParallelResults();
        osImplicitScheme << implicitUpwindScheme.getResults();


        osCrankParallel << crankParallel.getCrankResutls();


        //Closing all opened streams at the end
        osGeneralScheme.close();
        osExplicitScheme.close();
        osImplicitScheme.close();
        osImplicitParallel.close();
        osParallel.close();
        osCrankParallel.close();

    }
```

```
}


int main(int argc, char *argv[]) {

    MPI_Init(&argc, &argv);

    double timeOfStart;
    timeOfStart = MPI_Wtime();
    double timeOfEnd;


    //Number of boundary condition set. 1 for sign boundary set type ; 2 for exp
boundary set type

    int setNumber = 2;


    //Initial setings values are respectively: xMin, xMax, time, number of
spacePoints, CFL value

    cout << "Actual parameters are: ";

    //Extension type of file which storing results of schemes computation. It could be
Exel (.xls; .xlsx) file type for instance.
    std::string typeOfExtension = ".xls";

    //Running program using above settings for all initial boundary types


    double courantNumber = 0.999;


    double numOfPoints = 10000;


    double simulationTime = 5;


    vector<double> initialSettings;
    initialSettings.push_back(-50);
    initialSettings.push_back(50);
    initialSettings.push_back(simulationTime);
    initialSettings.push_back(numOfPoints);
    initialSettings.push_back(courantNumber);

    runSchemes(setNumber, initialSettings, typeOfExtension);
    timeOfEnd = MPI_Wtime();
    double programExecutionTime = timeOfEnd - timeOfStart;

    cout << "Total time of program execution is: " << programExecutionTime << endl;
    MPI_Finalize();
    return 0;

}
```

GeneralScheme.h

```
#pragma once

#include <memory>
#include <cmath>
#include <exception>
#include <iostream>
#include <algorithm>
#include "Matrix.h"
#include "MathFunctions.h"

/**
@brief This class gives number of functions used in numerical alalysis.
It initializes the Matrix, compte errors and calculating norms.
This class provides method to solve Analytical solution

*/

class GeneralScheme {


protected:

    static const double u = 1.75; //Const variable to of u value
    double xMin; //Lower boundary of space domain
    double xMax; //Upper boundary of space domain
    double time; // Time of simulation
    int numberOfSpacePoints; //Variable stores number of Space points for initializing
Matrix size
    double CFL; //Courant number
    Matrix matrixOfResults;

protected:
    //Matrix which stores Analytical solution results
    double dt; // Time step for solution
    double dx; // Space step

    int numberOfTimePoints; // Number of time points for initializing vetor
    bool isSetInitialised; //Checking if initial boundary conditions is initialized
    bool isAnaliticalSolutionSolved; ////Checking if Analitical Solution is Solved
    std::vector<double> errorVector;// Vector to store erroes

    double normInfiniteValue;
    double normOneValue;
    double normTwoValue;
    std::string name;

public:

    GeneralScheme();

    GeneralScheme(
            double xMin,
            double xMax,
            double time,
            double numberOfSpacePoints,
            double CFL);


    virtual ~GeneralScheme();
```

```
    /**
    @brief Calculating initial time point value

    @return Initial time point value
    */
    double calculateDtValue();

    /**
    @brief Calculatinginitialspace point value

    @return Initial pace point value
    */
    double calculateDxValue();

    /**
    @brief Returns actual time step value

    @return Actual time step value
    */

//    double getDx();


    /**
    @brief Method initializes classes Matrix with selected initial boundary type.
After this operation matrix if initialized with selected boundary type

    @param Variable provides boundary initialization set type number, 1 or 2
respectively for:  sign type and exponential type

    */
    void initializeSet(int setNumber);


    /**
    @brief Method allows to choose on of boundary function: 1 for sign type; 2 for
exponential type.

    @param Variable provides boundary initialization set type number, 1 or 2
respectively for:  sign type and exponential type
    @param Value of current time step to compute

    @return Calculated sign or exponential fucntion for given fuctionValue

    */
    double initializationFunction(int numberOfSet, double functionValue);

    /**
    @brief Method solves Analytical scheme. Results are stored in Matrix and returns
it.

    @param Variable provides boundary initialization set type number, 1 or 2
respectively for:  sign type and exponential type
    @param Value of actaul space point
    @param Value of current time point

    @return Matrix with calculated Analytical scheme

    */

    double solutionFunctionAnalytical(int numberOfSet, double actualSpace, double
actualTime);
```

```
    /**
    @brief Method returns matrix from class Matrix where Analitical solution is stored

    @return Matrix with calculated Analytical scheme

    */
    virtual const Matrix &getMatrix() const;


    /**
    @brief Method calcutales three norms in given matrix for last time step. Norm
infinite, One norm and Two norm

    @param Matrix for which last time point all 3 norms will be calculated


    @return Matrix with calculated Analytical scheme

    */


    void calculateNorms(Matrix &toCalculateError);

    void put_timeValues();


    /**
    @brief Virtual method to returns name of class

    @return name of class
    */
    virtual std::string getName();


    /**
    @brief Virtual method to solve analytical scheme

    @param Variable for inidicating boundary initialization set type, 1 or 2
respectively for:  sign type and exponential type


    @return Matrix with calculated Analytical scheme

    */
    virtual void solve(int setNumber);


    /**
  @brief Getting last matrix column which is General scheme solution

  @return Last Column of calculated Matrix

  */
    virtual std::vector<double> getResults();



};
```

GeneralScheme.cpp

```
#include <mpi.h>
#include "GeneralScheme.h"

/**
Default constructor
*/

GeneralScheme::GeneralScheme() {

}

GeneralScheme::~GeneralScheme() {

}


GeneralScheme::GeneralScheme(double xMin, double xMax, double time, double
numberOfSpacePoints, double CFL)
        : xMin(xMin), xMax(xMax), time(time),
numberOfSpacePoints(numberOfSpacePoints), CFL(CFL),
          isSetInitialised(false), isAnaliticalSolutionSolved(false),
name("GeneralScheme") {

    (*this).dt = (*this).calculateDtValue();
    (*this).dx = (*this).calculateDxValue();
    (*this).numberOfTimePoints = std::ceil((time) / (((*this).CFL * (*this).dx) / u));

    matrixOfResults = Matrix(numberOfSpacePoints, numberOfTimePoints);
    (*this).calculateDtValue();

}

double GeneralScheme::calculateDtValue() {
    return (*this).dt = ((*this).CFL * (*this).dx) / u;
}


double GeneralScheme::calculateDxValue() {
    return (*this).dx = (std::abs((*this).xMin) + std::abs((*this).xMax) + 1) /
(*this).numberOfSpacePoints;
}


//double GeneralScheme::getDx() {
//    return dx;
//}

double GeneralScheme::initializationFunction(int numberOfSet, double functionValue) {
    switch (numberOfSet) {
        case 1:
            return (MathFunctions::sign(functionValue) + 1);
            break;
        case 2:
            return std::exp((-1.0) * std::pow(functionValue, 2));
            break;
        default:
            std::cout << "There is no such choice using first condition set" <<
std::endl;
            return (MathFunctions::sign(functionValue) + 1);
    }
```

```
}

double GeneralScheme::solutionFunctionAnalytical(int numberOfSet, double
actualSpaceValue, double actualTimeValue) {
    switch (numberOfSet) {
        case 1:
            return 0.5 * (MathFunctions::sign(actualSpaceValue - 1.75 *
actualTimeValue) + 1);
            break;
        case 2:
            return 0.5 * std::exp((-1.0) * std::pow(actualSpaceValue - 1.75 *
actualTimeValue, 2));
            break;
        default:
            return 0.5 * (MathFunctions::sign(actualSpaceValue - 1.75 *
actualTimeValue) + 1);
            break;
    }
}


//Error and norms calculation
void GeneralScheme::calculateNorms(Matrix &toCalculateError) {
    errorVector.resize(toCalculateError.getNumOfRows());
    for (int i = 0; i < numberOfSpacePoints; ++i) {
        errorVector[i] =
                toCalculateError[i][numberOfTimePoints - 1] -
(*this).matrixOfResults[i][numberOfTimePoints - 1];
    }
    /*
    Second approach - quite complicated
    (*this).normInfiniteValue = std::max_element(errorVector.begin(),
errorVector.end(), MathFunctions::compareTwoAbsElements);
    normInfiniteValue1 = std::distance(errorVector.begin(), normInfiniteValue);
    double normInf = errorVector.at(normInfiniteValue1);
    */
    normInfiniteValue = 0;
    for (unsigned int i = 0; i < errorVector.size(); ++i) {
        if (abs(normInfiniteValue) < abs(errorVector.at(i))) {
            normInfiniteValue = abs(errorVector.at(i));
        }

        (*this).normOneValue += std::abs(errorVector[i]);
        (*this).normTwoValue += std::pow(std::abs(errorVector[i]), 2);
    }


    normTwoValue = std::sqrt(normTwoValue);
    normOneValue = normOneValue / numberOfSpacePoints;
    normTwoValue = normTwoValue / numberOfSpacePoints;

    std::vector<double> norms;
    norms.push_back(normInfiniteValue);
    norms.push_back(normOneValue);
    norms.push_back(normTwoValue);
    norms.push_back(dx);

    toCalculateError.resizeMat(numberOfSpacePoints, numberOfTimePoints + 1);
    int myRank;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
```

```
     if(myRank == 0)
    {
        for (unsigned int i = 0; i < norms.size()-1; ++i) {
            //Adding norms results to last matrix colunm
            std::cout<<"Norm "<< i+1 <<": " <<norms.at(i)<<std::endl;
            toCalculateError[i][numberOfTimePoints] = norms.at(i);
        }
    }


}


void GeneralScheme::put_timeValues() {

    double actualValue = 0;
    for (int i = 0; i < numberOfTimePoints; ++i) {
        matrixOfResults[0][i] = actualValue;
        actualValue += (*this).dt;
    }

}

std::string GeneralScheme::getName() {
    return name;
}

void GeneralScheme::initializeSet(int setNumber) {


    try {

        double actualValue = xMin;
        for (int i = 0; i < numberOfSpacePoints; ++i) {
            matrixOfResults[i][0] = (1.0 / 2.0) *
(*this).initializationFunction(setNumber, actualValue);
            actualValue += (*this).dx;
        }


        if (setNumber == 1) {
            for (int i = 0; i < numberOfTimePoints; ++i) {
                matrixOfResults[0][i] = 0;
                matrixOfResults[numberOfSpacePoints - 1][i] = 1;
            }
        } else {
            for (int i = 0; i < numberOfTimePoints; ++i) {
                matrixOfResults[0][i] = 0;
                matrixOfResults[numberOfSpacePoints - 1][i] = 0;
            }
        }

        (*this).isSetInitialised = true;

    }
    catch (std::exception &e) {
        std::cout << "Standard exception: " << e.what() << std::endl;
    }


}
```

```
//Analytical scheme solving
void GeneralScheme::solve(int numberOfBoundaryConditionSet) {
    (*this).initializeSet(numberOfBoundaryConditionSet);
    try {
        if ((*this).isSetInitialised == true) {
            std::cout << "Analytical solution runs and matrix is initialised\n";

            //Variables hold values below 0. Thanks to that negative values could be
passed to sign function, it makes loop iteration easier.
            double actualSpaceValue = xMin;
            //Variable assinged to dt because time at 0 point is initialised in
function initializeSet()
            double actualTimeValue = dt;
            for (int i = 1; i < numberOfSpacePoints; ++i) {
                for (int j = 1; j < numberOfTimePoints; ++j) {
                    matrixOfResults[i][j] =
solutionFunctionAnalytical(numberOfBoundaryConditionSet, actualSpaceValue,

actualTimeValue);
                    actualTimeValue += dt;

                }
                actualTimeValue = dt;
                actualSpaceValue += dx;
            }

            isAnaliticalSolutionSolved = true;
        } else {
            std::cout << "Matrix is not initialised\n";
        }

    }

    catch (std::exception &e) {
        std::cout << "Standard exception: " << e.what() << std::endl;
    }


}


std::vector<double> GeneralScheme::getResults()
{
    return matrixOfResults.getColumn(numberOfTimePoints - 1);
}

const Matrix &GeneralScheme::getMatrix() const {
    return matrixOfResults;
}
```

ExplicitUpwindScheme.h

```
#pragma once

#include <mpi.h>
#include "GeneralScheme.h"


/**
* Class created for calculating Explicit Upwind Scheme
* Class inherits methods from GeneralScheme class

*/

class ExplicitUpwindScheme : public GeneralScheme {

    std::string methodName;
    Matrix explicitResutls;

public:

    ExplicitUpwindScheme(double xMin,
                         double xMax,
                         double time,
                         double numberOfSpacePoints,
                         double CFL);

    ~ExplicitUpwindScheme();

    /**
    @brief Virtual method which solves Explicit Upwind Scheme

    @param Variable for inidicating boundary initialization set type, 1 or 2
respectively for:  sign type and exponential type

    */
    virtual void solve(int setNumber);

    /**
    @brief Method returns Matrix where Explicit Upwind Scheme solution is stored

    @return Matrix with calculated  Explicit Upwind Scheme

    */
    /* Matrix getUpwindMatrix();*/

    /**
    @brief Virtual method to returns name of ExplicitUpwindScheme class

    @return name of ExplicitUpwindScheme class
    */
    virtual std::string getName();

    std::vector<double> getLastExplicitMatrixColumn();

};
```

ExplicitUpwindScheme.cpp

```
#include "ExplicitUpwindScheme.h"


ExplicitUpwindScheme::ExplicitUpwindScheme(double xMin,
                                           double xMax,
                                           double time,
                                           double numberOfSpacePoints,
                                           double CFL) :
GeneralScheme::GeneralScheme(xMin, xMax, time,

numberOfSpacePoints, CFL),

methodName("ExplicitUpwindScheme") {

}


ExplicitUpwindScheme::~ExplicitUpwindScheme() {
}


void ExplicitUpwindScheme::solve(int setNumber) {
    try {

        double timeOfStart = MPI_Wtime();
        std::cout << "Explicit upwind scheme solution runs and matrix is
initialised\n";

        explicitResutls = Matrix(numberOfSpacePoints, numberOfTimePoints);
        double actualValue = xMin;

        for (int i = 0; i < numberOfSpacePoints; ++i)
        {
            actualValue = (i * (*this).dx) + xMin;
            explicitResutls[i][0] = (1.0 / 2.0) *
(*this).initializationFunction(setNumber, actualValue);

        }


        for (int i = 0; i < numberOfTimePoints; ++i)
        {
            explicitResutls[0][i] = 0;
            explicitResutls[numberOfSpacePoints - 1][i] = 2;

        }

//        (*this).isSetInitialised = true;
//        (*this).initializeSet(setNumber);
//        explicitResutls = Matrix((*this).getMatrix());

        for (int j = 0; j < numberOfTimePoints - 1; ++j) {
            for (int i = 1; i < numberOfSpacePoints; ++i) {
                explicitResutls[i][j + 1] = (explicitResutls[i][j] -
                                             CFL * (explicitResutls[i][j] -
explicitResutls[i - 1][j]));
            }

        }
        double timeOfSolutionsEnd = MPI_Wtime();
```

```
        double solutionTime = timeOfSolutionsEnd - timeOfStart;
        std::cout << "Single Explicit upwind scheme solution is: " << solutionTime <<
std::endl;

        //GeneralScheme::solve(setNumber);
        //calculateNorms((*this).explicitResutls);

    }

    catch (std::exception &e) {
        std::cout << "Standard exception: " << e.what() << std::endl;
    }
}

/*Matrix ExplicitUpwindScheme::getUpwindMatrix() {
    return explicitResutls;
}*/

std::string ExplicitUpwindScheme::getName() {
    return methodName;
}

std::vector<double> ExplicitUpwindScheme::getLastExplicitMatrixColumn() {
    return explicitResutls.getColumn(numberOfTimePoints - 1);
}
```

ExplicitUpwindParallel.h

```
#pragma once
#include <mpi.h>
#include "GeneralScheme.h"


/**
* Class created for calculating Explicit Upwind Scheme
* Class inherits methods from GeneralScheme class
*/

class ExplicitUpwindParallel : public GeneralScheme {

    std::string methodName;
    Matrix explicitParallelResults;
    int myRank;
    int numOfProc;
    double lastProc;
    double workAdditional;
    double tmp;
    double localLimit;
    std::vector<double> gatherResults;
public:
    const std::vector<double> &getGatherResults() const;


private:
    MPI_Status status;


public:

    ExplicitUpwindParallel(double xMin,
```

```
                            double xMax,
                            double time,
                            double numberOfSpacePoints,
                            double CFL);

    ~ExplicitUpwindParallel();

    /**
    @brief Virtual method which solves Explicit Upwind Scheme
    @param Variable for inidicating boundary initialization set type, 1 or 2
respectively for:  sign type and exponential type
    */
    virtual void solve(int setNumber);

    /**
    @brief Method returns Matrix where Explicit Upwind Scheme solution is stored
    @return Matrix with calculated  Explicit Upwind Scheme
    */
    /* Matrix getUpwindMatrix();*/

    /**
    @brief Virtual method to returns name of ExplicitUpwindScheme class
    @return name of ExplicitUpwindScheme class
    */
    virtual std::string getName();

    std::vector<double> getLastExplicitParallelMatrixColumn();

};
```

ExplicitUpwindParallel.cpp

```
#include "ExplicitUpwindParallel.h"


ExplicitUpwindParallel::ExplicitUpwindParallel(double xMin,
                                               double xMax,
                                               double time,
                                               double numberOfSpacePoints,
                                               double CFL) :
GeneralScheme::GeneralScheme(xMin, xMax, time,

numberOfSpacePoints, CFL),

methodName("ExplicitUpwindParallelScheme")
{

}

ExplicitUpwindParallel::~ExplicitUpwindParallel()
{
}


void ExplicitUpwindParallel::solve(int setNumber)
{
    try
    {
```

```
        std::cout << methodName << " solution runs and matrix is initialised\n";
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numOfProc);

        double timeOfStart = MPI_Wtime();
        explicitParallelResults = Matrix(numberOfSpacePoints, numberOfTimePoints);
        lastProc = numOfProc - 1;
        if (numberOfSpacePoints % numOfProc != 0)
        {
            throw "error";
        }

        localLimit = numberOfSpacePoints / numOfProc;


        int limitLow = myRank * localLimit;
        int limitHigh = (myRank + 1) * localLimit;

        double actualValue = xMin;


        for (int i = limitLow; i < limitHigh; ++i)
        {

            actualValue = (i * (*this).dx) + xMin;
            explicitParallelResults[i][0] = (1.0 / 2.0) *
(*this).initializationFunction(2, actualValue);

        }

        if (myRank == 0)
        {
            for (int i = 0; i < numberOfTimePoints; ++i)
            {
                explicitParallelResults[0][i] = 0;
            }
        }

        if (myRank == lastProc)
        {

            for (int i = 0; i < numberOfTimePoints; ++i)
            {

                explicitParallelResults[numberOfSpacePoints - 1][i] = 0;
            }

        }


        //explicitResutls = Matrix((*this).getMatrix());


        for (int j = 0; j < numberOfTimePoints - 1; ++j)
        {

            if (myRank != 0)
            {
                double localTmp;
                MPI_Recv(&localTmp, 1, MPI_DOUBLE, myRank - 1, 1, MPI_COMM_WORLD,
&status);
```

```
                explicitParallelResults[limitLow][j + 1] =
(explicitParallelResults[limitLow][j] -
                                                        CFL *
(explicitParallelResults[limitLow][j] - localTmp));
            }


            for (int i = limitLow + 1; i < limitHigh; ++i)
            {

                explicitParallelResults[i][j + 1] = (explicitParallelResults[i][j] -
                                                     CFL *
(explicitParallelResults[i][j] -
                                                            explicitParallelResults[i
- 1][j]));
            }

            if (lastProc != myRank)
            {
                MPI_Send(&explicitParallelResults[limitHigh - 1][j], 1, MPI_DOUBLE,
myRank + 1, 1, MPI_COMM_WORLD);

            }

        }


        gatherResults.resize(numberOfSpacePoints);

        std::vector<double> tempForReduce(numberOfSpacePoints);

        for (int i = 0; i < numberOfSpacePoints; ++i)
        {
            tempForReduce[i] = explicitParallelResults[i][numberOfTimePoints - 1];
        }


        MPI_Reduce(&tempForReduce[0], &gatherResults[0], numberOfSpacePoints,
MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);



        //GeneralScheme::solve(setNumber);
        //calculateNorms((*this).explicitResutls);
        double timeOfEnd = MPI_Wtime();
        double totalTime = timeOfEnd - timeOfStart;
        std::cout << methodName << " time is: " << totalTime << std::endl;


    }

    catch (std::exception &e)
    {
        std::cout << "Standard exception: " << e.what() << std::endl;
        return;
    }
}


/*Matrix ExplicitUpwindScheme::getUpwindMatrix() {
    return explicitResutls;
}*/
```

```
std::string ExplicitUpwindParallel::getName()
{
    return methodName;
}

std::vector<double> ExplicitUpwindParallel::getLastExplicitParallelMatrixColumn()
{
    return explicitParallelResults.getColumn(numberOfTimePoints - 1);
}

const std::vector<double> &ExplicitUpwindParallel::getGatherResults() const
{
    return gatherResults;
}
```

ImplicitUpwindScheme.h

```
#pragma once

#include <mpi.h>
#include "GeneralScheme.h"

class ImplicitUpwindScheme :
        public GeneralScheme {

    std::string methodName;
    Matrix implicitResults;
public:
    ImplicitUpwindScheme();

    ImplicitUpwindScheme(double xMin,
                         double xMax,
                         double time,
                         double numberOfSpacePoints,
                         double CFL);

    virtual ~ImplicitUpwindScheme();

    /**
    @brief Virtual method which solves Implicit Upwind Scheme

    @param Variable for inidicating boundary initialization set type, 1 or 2
respectively for:  sign type and exponential type

    */
    virtual void solve(int setNumber);

    /**
    @brief Method returns Matrix where Implicit Upwind Scheme solution is stored

    @return Matrix with calculated Implicit Upwind Schem

    */
    Matrix getImplicitUpwindMatrix();

    /**
    @brief Virtual method to returns name of  Implicit Upwind Scheme class

    @return name of  Implicit Upwind Scheme class
```

```
    */
    virtual std::string getName();


    virtual std::vector<double> getResults();
};
```

ImplicitUpwindScheme.cpp

```
#include "ImplicitUpwindScheme.h"


ImplicitUpwindScheme::ImplicitUpwindScheme() {
}

ImplicitUpwindScheme::ImplicitUpwindScheme(double xMin,
                                           double xMax,
                                           double time,
                                           double numberOfSpacePoints,
                                           double CFL) :
GeneralScheme::GeneralScheme(xMin, xMax, time,

numberOfSpacePoints, CFL),

methodName("ImplicitUpwindScheme") {
}


ImplicitUpwindScheme::~ImplicitUpwindScheme() {
}

void ImplicitUpwindScheme::solve(int setNumber) {
    try {

        std::cout << methodName << " scheme solution runs and matrix is
initialised\n";


        double timeOfStart = MPI_Wtime();
        std::cout << "Explicit upwind scheme solution runs and matrix is
initialised\n";

        implicitResults = Matrix(numberOfSpacePoints, numberOfTimePoints);
        double actualValue = xMin;

        for (int i = 0; i < numberOfSpacePoints; ++i)
        {
            actualValue = (i * (*this).dx) + xMin;
            implicitResults[i][0] = (1.0 / 2.0) *
(*this).initializationFunction(setNumber, actualValue);

        }


        for (int i = 0; i < numberOfTimePoints; ++i)
        {
            implicitResults[0][i] = 0;
            implicitResults[numberOfSpacePoints - 1][i] = 2;
```

```
        }

        //(*this).initializeSet(setNumber);
        //implicitResults = Matrix((*this).getMatrix());


        for (int j = 0; j < numberOfTimePoints - 1; ++j) {
            for (int i = 1; i < numberOfSpacePoints; ++i) {

                implicitResults[i][j + 1] = (-implicitResults[i][j] -
                                             CFL * implicitResults[i - 1][j]) / -(1 +
CFL);

            }

        }

        double timeOfEnd = MPI_Wtime();
        double totalTime = timeOfEnd - timeOfStart;
        std::cout << methodName << " time is: " << totalTime << std::endl;
        //GeneralScheme::solve(setNumber);
        //calculateNorms((*this).implicitResults);

    }

    catch (std::exception &e) {
        std::cout << "Standard exception: " << e.what() << std::endl;
    }
}

Matrix ImplicitUpwindScheme::getImplicitUpwindMatrix() {
    return implicitResults;
}

std::string ImplicitUpwindScheme::getName() {
    return methodName;
}

std::vector<double> ImplicitUpwindScheme::getResults()
{
    return implicitResults.getColumn(numberOfTimePoints - 1);
}
```

ImplicitParallel.h

```
#pragma once
#include <mpi.h>
#include "GeneralScheme.h"


/**
* Class created for calculating Explicit Upwind Scheme
* Class inherits methods from GeneralScheme class

*/

class ImplicitParallel : public GeneralScheme
{
```

```
    std::string methodName;
    Matrix implicitResultsParallel;
    int myRank;
    int numOfProc;
    double lastNode;
    double localLimit;
    std::vector<double> gatherResults;


private:
    MPI_Status status;


public:

    ImplicitParallel(double xMin,
                     double xMax,
                     double time,
                     double numberOfSpacePoints,
                     double CFL);

    ~ImplicitParallel();

    /**
    @brief Virtual method which solves Explicit Upwind Scheme

    @param Variable for inidicating boundary initialization set type, 1 or 2
respectively for:  sign type and exponential type

    */
    virtual void solve(int setNumber);

    /**
    @brief Method returns Matrix where Explicit Upwind Scheme solution is stored

    @return Matrix with calculated  Explicit Upwind Scheme

    */
    /* Matrix getUpwindMatrix();*/

    /**
    @brief Virtual method to returns name of ExplicitUpwindScheme class

    @return name of ExplicitUpwindScheme class
    */
    virtual std::string getName();

    std::vector<double> getLastExplicitParallelMatrixColumn();

    const std::vector<double> &getImplicitParallelResults() const;


};
```

ImplicitParallel.cpp

```
#include "ImplicitParallel.h"


ImplicitParallel::ImplicitParallel(double xMin,
                                   double xMax,
```

```
                                    double time,
                                    double numberOfSpacePoints,
                                    double CFL) : GeneralScheme::GeneralScheme(xMin,
xMax, time,

numberOfSpacePoints, CFL),

methodName("ImplicitUpwindParallelScheme")
{

}

ImplicitParallel::~ImplicitParallel()
{
}


void ImplicitParallel::solve(int setNumber)
{
    try
    {

        std::cout << methodName << " solution runs and matrix is initialised\n";
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numOfProc);
        implicitResultsParallel = Matrix(numberOfSpacePoints, numberOfTimePoints);
        double timeOfStart = MPI_Wtime();

        lastNode = numOfProc - 1;
        if (numberOfSpacePoints % numOfProc != 0)
        {
            throw "error";
        }

        localLimit = numberOfSpacePoints / numOfProc;


        int limitLow = myRank * localLimit;
        int limitHigh = (myRank + 1) * localLimit;

        double actualValue = xMin;


        for (int i = limitLow; i < limitHigh; ++i)
        {

            actualValue = (i * (*this).dx) + xMin;
            implicitResultsParallel[i][0] = (1.0 / 2.0) *
(*this).initializationFunction(setNumber, actualValue);

        }

        if (myRank == 0)
        {
            for (int i = 0; i < numberOfTimePoints; ++i)
            {
                implicitResultsParallel[0][i] = 0;
            }
        }

        if (myRank == lastNode)
        {
```

```
            for (int i = 0; i < numberOfTimePoints; ++i)
            {

                implicitResultsParallel[numberOfSpacePoints - 1][i] = 0;
            }

        }


        //explicitResutls = Matrix((*this).getMatrix());


        for (int j = 0; j < numberOfTimePoints - 1; ++j)
        {

            if (myRank != 0)
            {
                double localTmp;
                MPI_Recv(&localTmp, 1, MPI_DOUBLE, myRank - 1, 1, MPI_COMM_WORLD,
&status);
                implicitResultsParallel[limitLow][j + 1] = (-
implicitResultsParallel[limitLow][j] -
                                                        CFL * localTmp) / -(1 + CFL);
            }


            for (int i = limitLow + 1; i < limitHigh; ++i)
            {

                implicitResultsParallel[i][j + 1] = (-implicitResultsParallel[i][j] -
                                                     CFL * implicitResultsParallel[i -
1][j]) / -(1 + CFL);
            }

            if (lastNode != myRank)
            {
                MPI_Send(&implicitResultsParallel[limitHigh - 1][j], 1, MPI_DOUBLE,
myRank + 1, 1, MPI_COMM_WORLD);

            }

        }


        gatherResults.resize(numberOfSpacePoints);

        std::vector<double> tempForReduce(numberOfSpacePoints);

        for (int i = 0; i < numberOfSpacePoints; ++i)
        {
            tempForReduce[i] = implicitResultsParallel[i][numberOfTimePoints - 1];
        }


        MPI_Reduce(&tempForReduce[0], &gatherResults[0], numberOfSpacePoints,
MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);



        //GeneralScheme::solve(setNumber);
        //calculateNorms((*this).implicitResultsParallel);
```

```
        double timeOfEnd = MPI_Wtime();
        double totalTime = timeOfEnd - timeOfStart;
        std::cout << methodName << " time is: " << totalTime << std::endl;


    }

    catch (std::exception &e)
    {
        std::cout << "Standard exception: " << e.what() << std::endl;
        return;
    }
}


/*Matrix ExplicitUpwindScheme::getUpwindMatrix() {
    return explicitResutls;
}*/

std::string ImplicitParallel::getName()
{
    return methodName;
}

std::vector<double> ImplicitParallel::getLastExplicitParallelMatrixColumn()
{
    return implicitResultsParallel.getColumn(numberOfTimePoints - 1);
}

const std::vector<double> &ImplicitParallel::getImplicitParallelResults() const
{
    return gatherResults;
}
```

CrankParallel.h

```
#include <mpi.h>
#include <math.h>
#include "GeneralScheme.h"

#ifndef HIGH_PERFORMANCE_TECHNICAL_COMPUTING_CRANKPARALLEL_H
#define HIGH_PERFORMANCE_TECHNICAL_COMPUTING_CRANKPARALLEL_H


class CrankParallel : public GeneralScheme
{


    Matrix crankResutls;
    std::string methodName;
public:
    const std::string &getMethodName() const;

private:
    int myRank;
    int numOfProc;
    double lastNode;
    double localLimit;
    double A, AR;
    double B, BR;
```

```
    double C, CR;

public:

    CrankParallel(double xMin,
                  double xMax,
                  double time,
                  double numberOfSpacePoints,
                  double CFL);

    ~CrankParallel();

    void solve(int setNumber);

    void ThomasAlgorithm_P_LUDecomposition(int myCurrNode, int numberOfNodes, int N,
double *b,
                                           double *a, double *c, double *l, double
*d);

    void ThomasAlgorithm_P_solve(int N, double *l, double *d, double *c, double *x,
double *q);

    void CrankNicholson_RS(int N, double AR, double BR, double CR, double *y, double
*q);


    std::vector<double> getCrankResutls();


};


#endif //HIGH_PERFORMANCE_TECHNICAL_COMPUTING_CRANKPARALLEL_H
```

CrankParallel.cpp

```
#include "CrankParallel.h"


CrankParallel::CrankParallel(double xMin,
                             double xMax,
                             double time,
                             double numberOfSpacePoints,
                             double CFL) : GeneralScheme::GeneralScheme(xMin, xMax,
time,

numberOfSpacePoints, CFL),
                                           methodName("CrankParallelScheme")
{


    /**
       * from CN:
       * C * f(n+1)(i+1) + A * f(n+1)(i) + B * f(n+1)(i-1) = ...
       */
    A = 1;
    C = CFL / 4;
    B = -CFL / 4;

    BR = 1;
```

```
    CR = CFL / 4;
    AR = -CFL / 4;

}

CrankParallel::~CrankParallel()
{
}


void CrankParallel::ThomasAlgorithm_P_LUDecomposition(int myCurrNode, int
numberOfNodes, int N, double *b,
                                                      double *a, double *c, double *l,
double *d)
{
    int i;
    int rowsOfLocal, offsetLocal;
    double S[2][2], T[2][2], s1TMPVal, s2TMPVal;
    MPI_Status myStatus;

    for (i = 0; i < N; i++)
        l[i] = d[i] = 0.0;

    S[0][0] = S[1][1] = 1.0;
    S[1][0] = S[0][1] = 0.0;
    rowsOfLocal = (int) floor(N / numberOfNodes);
    offsetLocal = myCurrNode * rowsOfLocal;
// Form local products of the martriies R_k
    if (myCurrNode == 0)
    {
        s1TMPVal = a[offsetLocal] * S[0][0];
        S[1][0] = S[0][0];
        S[1][1] = S[0][1];
        S[0][1] = a[offsetLocal] * S[0][1];
        S[0][0] = s1TMPVal;
        for (i = 1; i < rowsOfLocal; i++)
        {
            s1TMPVal = a[i + offsetLocal] * S[0][0] -
                    b[i + offsetLocal - 1] * c[i + offsetLocal - 1] * S[1][0];
            s2TMPVal = a[i + offsetLocal] * S[0][1] -
                    b[i + offsetLocal - 1] * c[i + offsetLocal - 1] * S[1][1];
            S[1][0] = S[0][0];
            S[1][1] = S[0][1];
            S[0][0] = s1TMPVal;

            S[0][1] = s2TMPVal;
        }
    } else
    {
        for (i = 0; i < rowsOfLocal; i++)
        {
            s1TMPVal = a[i + offsetLocal] * S[0][0] -
                    b[i + offsetLocal - 1] * c[i + offsetLocal - 1] * S[1][0];
            s2TMPVal = a[i + offsetLocal] * S[0][1] -
                    b[i + offsetLocal - 1] * c[i + offsetLocal - 1] * S[1][1];
            S[1][0] = S[0][0];
            S[1][1] = S[0][1];
            S[0][0] = s1TMPVal;
            S[0][1] = s2TMPVal;
        }
    }
```

```
for (i = 0; i <= log2(numberOfNodes); i++)
{
    if (myCurrNode + pow(2, i) < numberOfNodes)
        MPI_Send(S, 4, MPI_DOUBLE, int(myCurrNode + pow(2, i)), 0,
                 MPI_COMM_WORLD);
    if (myCurrNode - pow(2, i) >= 0)
    {
        MPI_Recv(T, 4, MPI_DOUBLE, int(myCurrNode - pow(2, i)), 0,
                 MPI_COMM_WORLD, &myStatus);
        s1TMPVal = S[0][0] * T[0][0] + S[0][1] * T[1][0];
        S[0][1] = S[0][0] * T[0][1] + S[0][1] * T[1][1];
        S[0][0] = s1TMPVal;
        s1TMPVal = S[1][0] * T[0][0] + S[1][1] * T[1][0];
        S[1][1] = S[1][0] * T[0][1] + S[1][1] * T[1][1];
        S[1][0] = s1TMPVal;
    }
}

d[offsetLocal + rowsOfLocal - 1] = (S[0][0] + S[0][1]) /
                                   (S[1][0] + S[1][1]);
if (myCurrNode == 0)
{
    MPI_Send(&d[offsetLocal + rowsOfLocal - 1], 1, MPI_DOUBLE,
             1, 0, MPI_COMM_WORLD);
} else
{
    MPI_Recv(&d[offsetLocal - 1], 1, MPI_DOUBLE, myCurrNode - 1, 0,
             MPI_COMM_WORLD, &myStatus);
    if (myCurrNode != numberOfNodes - 1)

        MPI_Send(&d[offsetLocal + rowsOfLocal - 1], 1, MPI_DOUBLE,
                 myCurrNode + 1, 0, MPI_COMM_WORLD);
}


if (myCurrNode == 0)
{
    l[0] = 0;
    d[0] = a[0];
    for (i = 1; i < rowsOfLocal - 1; i++)
    {
        l[offsetLocal + i] = b[offsetLocal + i - 1] /
                             d[offsetLocal + i - 1];
        d[offsetLocal + i] = a[offsetLocal + i] -
                             l[offsetLocal + i] * c[offsetLocal + i - 1];
    }
    l[offsetLocal + rowsOfLocal - 1] = b[offsetLocal + rowsOfLocal - 2] /
                                       d[offsetLocal + rowsOfLocal - 2];
} else
{
    for (i = 0; i < rowsOfLocal - 1; i++)
    {
        l[offsetLocal + i] = b[offsetLocal + i - 1] /
                             d[offsetLocal + i - 1];
        d[offsetLocal + i] = a[offsetLocal + i] -
                             l[offsetLocal + i] * c[offsetLocal + i - 1];
    }
    l[offsetLocal + rowsOfLocal - 1] = b[offsetLocal + rowsOfLocal - 2] /
                                       d[offsetLocal + rowsOfLocal - 2];
}
```

```
    if (myCurrNode > 0)
        d[offsetLocal - 1] = 0;
    // Distribute d_k and l_k to all processes
    double *tmp = new double[N];
    for (i = 0; i < N; i++)
        tmp[i] = d[i];
    MPI_Allreduce(tmp, d, N, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    for (i = 0; i < N; i++)
        tmp[i] = l[i];
    MPI_Allreduce(tmp, l, N, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    delete[] tmp;

}

/**
 *
 * @param N
 * @param lStore
 * @param dStore
 * @param c
 * @param x output
 * @param q
 */
void CrankParallel::ThomasAlgorithm_P_solve(int N, double *lStore, double *dStore,
double *c, double *q, double *x)
{
    int i;
    double *y = new double[N];

    for (i = 0; i < N; i++)
        y[i] = 0.0;

    /* Forward Substitution [L][y] = [q] */
    y[0] = q[0];
    for (i = 1; i < N; i++)
        y[i] = q[i] - lStore[i] * y[i - 1];
    /* Backward Substitution [U][x] = [y] */
    x[N - 1] = y[N - 1] / dStore[N - 1];
    for (i = N - 2; i >= 0; i--)
        x[i] = (y[i] - c[i] * x[i + 1]) / dStore[i];

    delete[] y;
    return;
}

/*
 * from CN:
 * ... = AR * f(n)(i+1) + BR * f(n)(i) + CR * f(n)(i-1)
 *
 * @param y input (n)
 * @param q output (n+1)
 */
void CrankParallel::CrankNicholson_RS(int N, double AR, double BR, double CR, double
*y, double *q)
{
    for (int i = 1; i < N - 1; ++i)
    {
        q[i] = AR * y[i - 1] + BR * y[i] + CR * y[i + 1];
    }
}

void CrankParallel::solve(int setNumber)
```

```
{
    try
    {

        std::cout << methodName << " solution runs and matrix is initialised\n";
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
        MPI_Comm_size(MPI_COMM_WORLD, &numOfProc);
        crankResutls = Matrix(numberOfSpacePoints, numberOfTimePoints);
        double timeOfStart = MPI_Wtime();

        lastNode = numOfProc - 1;
        if (numberOfSpacePoints % numOfProc != 0)
        {
            throw "error";
        }

        //only for numOfProc = 2^n; n = 1, 2, ...


        if (myRank == 0)
        {
            double actualValue = xMin;

            for (int i = 0; i < numberOfSpacePoints; ++i)
            {

                actualValue = (i * (*this).dx) + xMin;
                crankResutls[i][0] = (1.0 / 2.0) *
(*this).initializationFunction(setNumber, actualValue);

            }

            for (int i = 0; i < numberOfTimePoints; ++i)
            {
                crankResutls[0][i] = 0;
            }

            for (int i = 0; i < numberOfTimePoints; ++i)
            {

                crankResutls[numberOfSpacePoints - 1][i] = 0;
            }

        }

        double *a = new double[numberOfSpacePoints];
        double *b = new double[numberOfSpacePoints];
        double *c = new double[numberOfSpacePoints];
        double *d = new double[numberOfSpacePoints];
        double *l = new double[numberOfSpacePoints];
        double *tmpData = new double[numberOfSpacePoints];
        double *tmpPrev = new double[numberOfSpacePoints];
        double *tmpNext = new double[numberOfSpacePoints];


        b[0] = 0.0;
        c[numberOfSpacePoints - 1] = 0.0;

        /**
         * from CN:
         * C * f(n+1)(i+1) + A * f(n+1)(i) + B * f(n+1)(i-1) = ...
         */
```

```
        for (int i = 0; i < numberOfSpacePoints; ++i)
        {
            a[i] = A;
            if (i != 0) b[i] = B;
            if (i != numberOfSpacePoints - 1) c[i] = C;
            l[i] = 0.0;
            d[i] = 0.0;
        }

        ThomasAlgorithm_P_LUDecomposition(myRank, numOfProc, numberOfSpacePoints, b,
a, c, l, d);

        for (int j = 0; j < numberOfTimePoints - 1; ++j)
        {
            for (int i = 0; i < numberOfSpacePoints; ++i)
                tmpPrev[i] = crankResutls[i][j];

            CrankNicholson_RS(numberOfSpacePoints, AR, BR, CR, tmpPrev, tmpData);
            ThomasAlgorithm_P_solve(numberOfSpacePoints, l, d, c, tmpData, tmpNext);

            for (int i = 0; i < numberOfSpacePoints; ++i)
                crankResutls[i][j + 1] = tmpNext[i];
        }

        delete (a);
        delete (b);
        delete (c);
        delete (d);
        delete (l);
        delete (tmpData);
        delete (tmpPrev);
        delete (tmpNext);

        //GeneralScheme::solve(setNumber);
        calculateNorms((*this).crankResutls);
        double timeOfEnd = MPI_Wtime();
        double totalTime = timeOfEnd - timeOfStart;
        std::cout << methodName << " time is: " << totalTime << std::endl;


    }

    catch (std::exception &e)
    {
        std::cout << "Standard exception: " << e.what() << std::endl;
        return;
    }
}

std::vector<double> CrankParallel::getCrankResutls()
{
    return crankResutls.getColumn(numberOfTimePoints - 1);
}

const std::string &CrankParallel::getMethodName() const
{
    return methodName;
}
```

Display.h

```
#pragma once

#include <vector>
#include <iostream>


/**
@brief Own namespace to display vectors and names
*/

namespace display {

    /**
    @brief Fucntion to display given input vector to console

    @param Vector to display
    */
    void displayVector(std::vector<double> vector);

    /**
    @brief Display name of setected boundary condition. This function contains two
sets of boundary condition: exponential and sign
    @param Number of Boundary condition to dsplay. Takes values 1 or 2
    */
    std::string getInitialBoundaryConditionName(int &numberOfBoundaryCondition);
}
```

Display.cpp

```
#include <iterator>
#include "Display.h"


/*
void  display::displayVector(std::vector<double> vector)
{
       for (auto v : vector)
       {
              std::cout << "\n" << v;
       }
       std::cout << std::endl;

}
*/
std::string display::getInitialBoundaryConditionName(int &numberOfBoundaryCondition) {
    switch (numberOfBoundaryCondition) {
        case 1:
            return "Sign";
            break;
        case 2:
            return "Exp";
            break;
        default:
            return "Sign";
            break;
    }
```

```
}

//Overloaded operator << for easy loading vector to file
```

Matrix.h

```
#include <vector>
#include <ostream>
#include <iostream>
#include <fstream>
#include <memory>


class Matrix : private std::vector<std::vector<double> > {


public:
    std::vector<double> column;
    using std::vector<std::vector<double> >::operator[];

    Matrix();

    //std::shared_ptr < std::vector<double> > tmpVec;
    Matrix(int numOfRows, int numOfColumns);

    Matrix(const Matrix &m);

    /**
    @brief Method returns number of rows in Matrix

    @return Number of rows in Matrix
    */
    int getNumOfRows() const;

    /**
    @brief Method returns number of columns in Matrix

    @return Number of columns in Matrix
    */
    int getNumOfColumns() const;

    /**
    @brief Method returns selected row as vector

    @return Returns selected row as vector
    */
    std::vector<double> &getRow(int rowNumber);

    /**
    @brief Method returns selected column as vector

    @return Selected column as vector
    */
    std::vector<double> &getColumn(int columnnumber);

    friend std::ostream &operator<<(std::ostream &os, Matrix &mat);

    friend std::ofstream &operator<<(std::ofstream &ofs,
```

```
                                    const Matrix &m);

    Matrix &operator=(const Matrix &m);

    /**
    @brief Method returns selected column as vector

    @param New rows quantity for Matrix resize

    @param New columns quantity for Matrix resize
    @return Selected column as vector
    */
    void resizeMat(int numOfRows, int numOfColumns);

};
```

Matrix.cpp

```
#include "Matrix.h"

/**
Matrix class allows to  inspired by Dr Peter Sherar's Matrix class
*/

typedef std::vector<std::vector<double> > mat;

/**
Default Constructor
*/
Matrix::Matrix() : mat() {}

/**
Constructor
*/
Matrix::Matrix(int numOfRows, int numOfColumns) : mat() {


    (*this).resize(numOfRows);

    for (int i = 0; i < numOfRows; ++i) {
        (*this)[i].resize(numOfColumns);
    }

}

//Copy constructor
Matrix::Matrix(const Matrix &m) : std::vector<std::vector<double> >() {
    // set the size of the rows
    (*this).resize(m.size());
    // set the size of the columns
    std::size_t i;
    for (i = 0; i < m.size(); i++) (*this)[i].resize(m[0].size());

    // copy the elements
    for (int i = 0; i < m.getNumOfRows(); i++)
        for (int j = 0; j < m.getNumOfColumns(); j++)
            (*this)[i][j] = m[i][j];
}

int Matrix::getNumOfRows() const {
```

```
    return (*this).size();
}

int Matrix::getNumOfColumns() const {
    return (*this)[0].size();
}



std::vector<double> &Matrix::getRow(int rowNumber) {
    static std::vector<double> tmp;

    for (int i = 0; i < (*this).getNumOfColumns(); ++i) {
        tmp.push_back((*this)[rowNumber][i]);
    }

    return tmp;
}


std::vector<double> &Matrix::getColumn(int columnNumber) {


    for (int i = 0; i < (*this).getNumOfRows(); ++i) {
        column.push_back((*this)[i][columnNumber]);
    }

    return column;
}


std::ostream &operator<<(std::ostream &os, Matrix &mat) {

    for (int i = 0; i < mat.getNumOfRows(); ++i) {
        for (int j = 0; j < mat.getNumOfColumns(); ++j) {
            os << mat[i][j] << " ";
        }
        os << std::endl;
    }

    return os;
}


std::ofstream &operator<<(std::ofstream &ofs, const Matrix &m) {
    //put matrix rownumber in first line (even if it is zero)
    //ofs << "dt" << std::endl;
    //put matrix columnnumber in second line (even if it is zero)
    //ofs << m.getNumOfColumns() << std::endl;
    //put data in third line (if size==zero nothing will be put)
    for (int i = 0; i < m.getNumOfRows(); i++) {
        for (int j = 0; j < m.getNumOfColumns(); j++) ofs << m[i][j] << "\t";
        ofs << std::endl;
    }
    return ofs;
}


Matrix &Matrix::operator=(const Matrix &m) {
    (*this).resize(m.size());
    std::size_t i;
    std::size_t j;
```

```
    for (i = 0; i < m.size(); i++) (*this)[i].resize(m[0].size());

    for (i = 0; i < m.size(); i++)
        for (j = 0; j < m[0].size(); j++)
            (*this)[i][j] = m[i][j];
    return *this;
}


void Matrix::resizeMat(int numOfRows, int numOfColumns) {


    (*this).resize(numOfRows);

    for (int i = 0; i < numOfRows; ++i) {
        (*this)[i].resize(numOfColumns);
    }

}
```

MathFunctions.h

```
#pragma once

#include <cmath>

/**
@brief Class

*/
class MathFunctions {
public:
    MathFunctions();

    ~MathFunctions();


    static int sign(double x);

    static bool compareTwoAbsElements(double first, double second);

};
```

MathFunctions.cpp

```
#include "MathFunctions.h"


MathFunctions::MathFunctions() {
}


MathFunctions::~MathFunctions() {
}

/**
```

```
@brief Static function sign

Fucntion gives an output of sign function
1. -1 When input walue < 0
2.  0 when input value = 0
3.  1 When input value > 0

@param Input value to sign function
@return result of sign function
*/
int MathFunctions::sign(double x) {
    if (x < 0) {
        return -1;
    } else if (x == 0) {
        return 0;
    } else {
        return 1;
    }

}

bool MathFunctions::compareTwoAbsElements(double first, double second) {
    return (std::abs(first) < std::abs(second));
}
```