



**BERGISCHE  
UNIVERSITÄT  
WUPPERTAL**

---

**Term Paper**

**Hydrodynamics  
with python/MPI**

---

by

**Zeliha Kanat and Nitin Sethi**

**Supervisor:** Korzec, Dr. Tomasz

Bergische Universität Wuppertal

# Abstract

This project aims to explore the divergence-free nature of Incompressible flow, the loss of this property due to approximative derivatives or inexact treatment of boundary conditions, and the application of its correction based on the approach of Helmholtz decomposition of Velocity [1].

The velocity profile (Equation 3.1) has been used to create a square grid with uniform spacing  $h$  and the derivatives are discretized using the symmetric finite difference method with errors  $O(h^2)$ . On Boundary points,  $\vec{v}$  is not corrected and  $\phi$  is zero. The decomposition algorithm and geometry have been achieved using 1-D parallelization in MPI python.

In the end, the result is shown in terms of the plot of divergence of velocity before and after the correction is applied. The parallelization of the problem has also given us the opportunity to explore the efficiencies of 1-D parallelization using a strong scaling plot and the relationship of time with respect to the number of sites.

As a result , the implemented algorithm has been decreased the norm of the divergence of initial velocity profile 22.09 to 0.706 for  $N=128$  and 44.98 to 0.366 for  $N = 256$  uniformly defined grid. This results shows the correctness of the implementation and  $O(h^2)$  error effect for finite difference method. Additionally, strong scaling result has showed that the algorithm has good parallelization performance.

# Copywrite Statement

We hereby confirm that the content of this work was written entirely by us and no other sources than the cited ones were used in the process.

21.07.2022 Wuppertal

Date, Place

21.07.2022 Wuppertal

Date, Place



Signature: Kanat, Zeliha



Signature: Sethi, Nitin

# Contents

<b>Abstract</b>	<b>I</b>
<b>Copywrite Statement</b>	<b>II</b>
<b>List of Figures</b>	<b>IV</b>
<b>List of Tables</b>	<b>V</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Body</b>	<b>2</b>
2.1 Incompressible flows . . . . .	2
2.2 Finite difference methods . . . . .	3
2.3 Poisson equation . . . . .	6
2.4 Divergence . . . . .	6
2.5 Jacobi Iteration . . . . .	7
2.6 Gradient . . . . .	7
2.7 Task Definition . . . . .	8
2.8 Parallelization . . . . .	8
<b>3 Result and Discussion</b>	<b>10</b>
<b>Bibliography</b>	<b>13</b>
<b>A The code</b>	<b>14</b>
<b>B Some program outputs</b>	<b>18</b>

# List of Figures

2.2.1	An example of a 1D (above) and 2D (below) Cartesian grid for FD methods (full symbols denote boundary nodes and open symbols denote internal computational nodes). . . . .	3
2.2.2	On the definition of a derivative and its approximations . . . . .	4
2.8.1	Illustration of 1D parallelization process. Each color shows the different processors(4 in total) and red covered nodes shows neighnoring communication in the boundaries. . . . .	9
3.0.1	Final norms versus error results . . . . .	11
3.0.2	Strong Scaling results . . . . .	12
B.0.1	Results for $N = 256$ , processor = 4 . . . . .	18
B.0.2	Results for $N = 256$ , processor = 16 . . . . .	18
B.0.3	Results for $N = 256$ , processor = 64 . . . . .	19
B.0.4	Results for $N = 128$ , processor = 4 . . . . .	19
B.0.5	Results for $N = 128$ , processor = 16 . . . . .	19
B.0.6	Results for $N = 128$ , processor = 64 . . . . .	20

# List of Tables

3.0.1 Algorithm results for different problem sizes . . . . .	11
---	----

# Chapter 1

## Introduction

A flow is classified as being compressible or incompressible, depending on the level of variation of density during flow. Incompressibility is an approximation, in which the flow is said to be incompressible if the density remains nearly constant throughout. Therefore, the volume of every portion of fluid remains unchanged over the course of its motion when the flow is approximated as incompressible.

Incompressible flows are divergence free, i.e. the velocity field has the property:

$$\text{div}(\vec{v}) = 0$$

In terms of these properties we investigated incompressible flows and made the calculation of the correction is based on the Helmholtz decomposition of  $\vec{v}$  and it has been done starting with computing  $\text{div}(\vec{v})$  after that solving the Poisson equation for  $\phi$  and computing gradient  $\text{grad}(\phi)$  followed to reach divergence free result.

# Chapter 2

## Body

### 2.1 Incompressible flows

In this section theoretical background about the equations for incompressible flows and finite difference approach for the equations are represented.

Navier-Stokes equations (NSE) in a general:

$$\frac{\partial \rho}{\partial t} + \vec{\nabla} \cdot (\rho \vec{u}) = 0 \quad (2.1)$$

$$\frac{\partial(\rho \vec{u})}{\partial t} + \vec{\nabla} \cdot [\rho \overline{\vec{u} \otimes \vec{u}}] = -\vec{\nabla} p + \vec{\nabla} \cdot \vec{\tau} + \rho \vec{f} \quad (2.2)$$

$$\frac{\partial(\rho e)}{\partial t} + \vec{\nabla} \cdot ((\rho e + p) \vec{u}) = \vec{\nabla} \cdot (\vec{\tau} \cdot \vec{V}) + \rho \vec{f} \cdot \vec{u} + \vec{\nabla} \cdot (\vec{q}) + r \quad (2.3)$$

Here  $\otimes$  denotes the tensorial product, forming a tensor from the constituent vectors. A double bar denotes a tensor.

Incompressible flows allow for a simplification of the Navier-Stokes equations. After those simplifications(velocity and kinematic pressure i.e., pressure divided by density) [2] , we can present:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u + \nabla P = \nu \nabla^2 u \quad (2.4)$$

$$\nabla \cdot u = 0 \quad (2.5)$$



## 2.2 Finite difference methods

A finite difference method proceeds by replacing the derivatives in the differential equations by finite difference approximations. This gives a large algebraic system of equations to be solved in place of the differential equation, something that is easily solved on a computer.

The first step in obtaining a numerical solution is to discretize the geometric domain— i.e., a numerical grid must be defined. In finite difference (FD) discretization methods the grid is usually locally structured, i.e., each grid node may be considered the origin of a local coordinate system, whose axes coincide with grid lines.

Figure 2.1 shows examples of one-dimensional (1D) and two-dimensional (2D) Cartesian grids used in FD methods.[3]

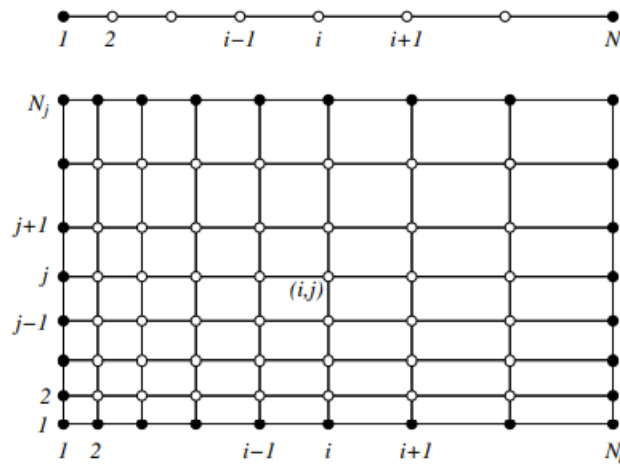


Figure 2.2.1: An example of a 1D (above) and 2D (below) Cartesian grid for FD methods (full symbols denote boundary nodes and open symbols denote internal computational nodes).

Each node thus has one unknown variable value associated with it and must provide one algebraic equation. The latter is a relation between the variable value at that node and those at some of the neighboring nodes. It is obtained by replacing each term of the PDE at the particular node by a finite-difference approximation. Of course, the numbers of equations and unknowns must be equal. At boundary nodes where variable values are given (Dirichlet conditions), no equation is needed. When the boundary conditions involve derivatives (as in Neumann conditions), the boundary condition must be discretized to contribute an equation to the set that must be solved.[?]

The idea behind finite-difference approximations is borrowed directly from the definition of a derivative:

$$\left(\frac{\partial\phi}{\partial x}\right)_{x_i} = \lim_{h \rightarrow 0} \frac{\phi(x_i + h) - \phi(x_i)}{h} \quad (2.6)$$

A geometrical interpretation is shown in Figure 2.2.2 to which we shall refer frequently. The first derivative at a point is the slope of the tangent to the curve at that point, the line marked 'Exact' in the figure. Its slope can be approximated by the slope of a line passing through two nearby points on the curve. The dotted line shows approximation by a forward difference; the derivative at  $x_i$  is approximated by the slope of a line passing through the point  $x_i$  and another point at  $x_i + \delta x$ . The dashed line illustrates approximation by backward difference for which the second point is  $x_i - \delta x$ . The line labeled 'Central' represents approximation by a central difference: it uses the slope of a line passing through two points lying on opposite sides of the point at which the derivative is approximated.[?]

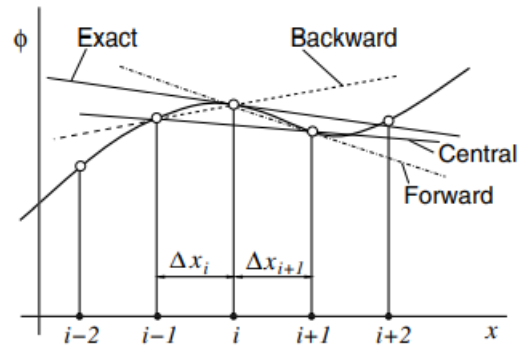


Figure 2.2.2: On the definition of a derivative and its approximations

It is obvious from Figure 2.2.2 that some approximations are better than others. The line for the central-difference approximation has a slope very close to the slope of the exact line; if the function were a second-order polynomial and the points were equally spaced in x-direction, the slopes would match exactly. It is also obvious from Figure 2.2.2 that the quality of the approximation improves when the additional points are close to  $x_i$ , i.e., as the grid is refined, the approximation improves. The approximations shown in Figure 2.2.2 are a few of many possibilities; Now we will outline the principal approaches to deriving approximations for the first and second derivatives.[?]

$$\left(\frac{\partial\phi}{\partial x}\right)_i \approx \frac{\phi_{i+1} - \phi_i}{h} \quad (2.7)$$

$$\left(\frac{\partial\phi}{\partial x}\right)_i \approx \frac{\phi_i - \phi_{i-1}}{h} \quad (2.8)$$

$$\left(\frac{\partial\phi}{\partial x}\right)_i \approx \frac{\phi_{i+1} - \phi_{i-1}}{2h} \quad (2.9)$$

These are the forward- (FDS), backward- (BDS), and central-difference (CDS) schemes, respectively. Here  $h$  is the distance between the grid points i.e.,  $x_i - x_{(i-1)}$  and  $x_{(i+1)} - x_i$ .

To estimate the second derivative at a point, one may use the approximation for the first derivative twice. We will end up with the following equation.[?]

For equidistant spacing of the points:

$$\left(\frac{\partial^2\phi}{\partial^2x}\right)_i \approx \frac{\phi_{i+1} + \phi_{i-1} - 2\phi_i}{h^2} \quad (2.10)$$

## 2.3 Poisson equation

We have the Poisson equation  $-\Delta\phi = \text{div}(\vec{v})$  and if we present following approach ,

$$-\Delta\phi = \text{div}(\vec{v}) \quad (2.11)$$

$$\Delta\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} \quad (2.12)$$

If we apply Finite Differences to a square grid with uniform spacing  $h$ ,

$$\phi_{xx}(x_i, y_j) \approx \frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{h^2}, \phi_{yy}(x_i, y_j) \approx \frac{\phi_{i,j-1} - 2\phi_{i,j} + \phi_{i,j+1}}{h^2}, \quad (2.13)$$

Finally we will reach ,

$$\Delta\phi = \frac{4\phi_{i,j} - \phi_{i-1,j} - \phi_{i+1,j} - \phi_{i,j-1} - \phi_{i,j+1}}{h^2} \quad (2.14)$$

## 2.4 Divergence

The general definition of the divergence of a vector field,

$$\begin{aligned} \nabla \cdot \vec{F} \\ \nabla \cdot \vec{F} = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z} \end{aligned}$$

In our case in 2D velocity vector field,

$$\text{div}(\vec{v}) = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}$$

If we apply Finite Differences method to a square grid with uniform spacing  $h$ ,

$$\text{div}(\vec{v}) \approx \frac{\vec{v}_{i-1,j} - \vec{v}_{i+1,j}}{2h} + \frac{\vec{v}_{i,j+1} - \vec{v}_{i,j-1}}{2h}$$

## 2.5 Jacobi Iteration

The Jacobi iteration for solving the Poisson equation by definition works as follow[1];

$$-\Delta\phi = \rho \quad (2.15)$$

- Set  $\phi = 0$  everywhere
- Set  $\phi_{new} = \phi$
- While not converged

1. For all internal points  $(i, j)$ , set

$$\phi_{new}(i, j) = \frac{\rho(i, j)h^2 + \phi(i + a, j) + \phi(i - a, j) + \phi(i, j + a) + \phi(i, j - a)}{4}$$

2. Set  $\phi_{new} = \phi$

When we apply this process to our system;

$$-\Delta\phi = \text{div}(\vec{v}) \quad (2.16)$$

We will have ;

$$-\left(\frac{4\phi_{i,j} - \phi_{i-1,j} - \phi_{i+1,j} - \phi_{i,j-1} - \phi_{i,j+1}}{h^2}\right) = \frac{\vec{v}_{i-1,j} - \vec{v}_{i+1,j}}{2h} + \frac{\vec{v}_{i,j+h} - \vec{v}_{i,j-h}}{2h} \quad (2.17)$$

$$\phi_{newi,j} = \frac{1}{4}(\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1}) + \frac{h}{8}(\vec{v}_{i-1,j} - \vec{v}_{i+1,j} + \vec{v}_{i,j+h} - \vec{v}_{i,j-h}) \quad (2.18)$$

## 2.6 Gradient

The general definition of the gradient of a vector field,

$$\nabla f(x_1, x_2, \dots, x_n) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x_1, x_2, \dots, x_n) \\ \frac{\partial f}{\partial x_2}(x_1, x_2, \dots, x_n) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x_1, x_2, \dots, x_n) \end{bmatrix}$$

In our case ,

$$grad(\phi) = \begin{bmatrix} \frac{\partial \phi}{\partial x} \\ \frac{\partial \phi}{\partial y} \end{bmatrix}$$

If we apply Finite Differences method to a square grid with uniform spacing  $h$ ,

$$grad(\phi) \approx \begin{bmatrix} \frac{\phi_{i+1} - \phi_{i-1}}{2h} \\ \frac{\phi_{i+1} - \phi_{i-1}}{2h} \end{bmatrix}$$

## 2.7 Task Definition

In this paper the correction of velocity profile for incompressible flows has been investigated. The calculation of the correction is based on the Helmholtz decomposition of  $\vec{v}$  and works as follows[1]:

- Compute  $div(\vec{v})$
- Solve the Poisson equation  $-\Delta\phi = div(\vec{v})$  for  $\phi$
- Compute  $grad(\phi)$
- Correct  $\vec{v} \rightarrow \vec{v} + grad(\phi)$

The defined process above has implemented using Python and MPI has used for 1D parallel computing. For the original code for the implementation please check appendix A.

## 2.8 Parallelization

The package for MPI (Message Passing Interface) in Python, called *mpi4py*, has been used for completing the parallelization task. In our project 1D parallelization has been applied. According to number of processors, the problem size(N) has been divided and local size has been found for each processors. The communication has been done for the boundary points to get the necessary information for the next iteration. The Figure 2.8.1 illustrates the parallelization process when N=16 and the number of processors is 4. In this case every processors has local size 4(16/4). Each color presents different processor and communication happens for the red highlighted boundary points.

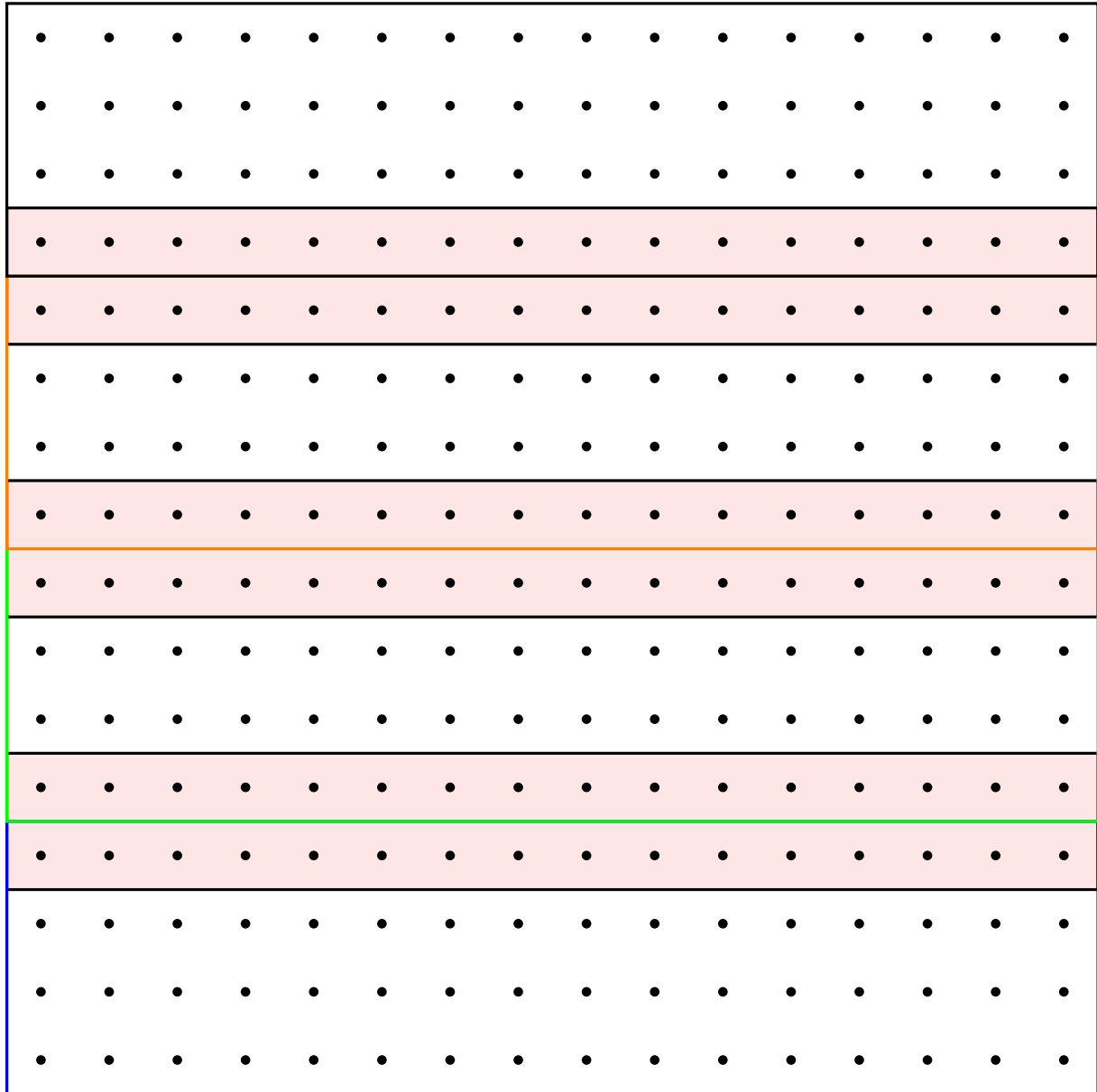


Figure 2.8.1: Illustration of 1D parallelization process. Each color shows the different processors(4 in total) and red covered nodes shows neighboring communication in the boundaries.

## Chapter 3

# Result and Discussion

In this paper application for velocity profile correction by using Helmholtz decomposition has been studied. In the beginning the initial velocity profile has been defined by following a smooth function;

$$v = \begin{bmatrix} x_2 \\ -x_1 \end{bmatrix} * \frac{1}{(x_1^2 + x_2^2)} + 2 * \begin{bmatrix} x_1 \\ -x_2 \end{bmatrix} * \exp(-x_1^2 - x_2^2) \quad (3.1)$$

It is significant to have a smooth definition for velocity profile in the beginning, random selections increase the complexity by contrast with our aim. In order to set the initial velocity profile every grid points has been defined accordingly their coordinates using this smooth function. Another useful package of Python, numpy, has been used to define corresponding arrays for x-coordinates and y-coordinates into the one array. Setting up the velocity profile with this way is advantages for paralyzing with ready buffer for memory also calculation tasks for arrays are straightforward and fast with well designed numpy package.

After definition of 2D velocity profile, we have implemented the finite difference methods on the divergence , jacobi iteration, and gradient as we explained in the body part of the term paper. As in the task definition , we have corrected the internal velocity and calculated the divergence again for this corrected velocity.



Problem size N	Norm of Divergence Before	Norm of the Divergence After
64	10.36	1.21
128	22.09	0.706
256	44.98	0.366

Table 3.0.1: Algorithm results for different problem sizes

Table 3.0.1 shows us the results of our algorithms for different problem sizes. The problem size  $N$  is important for the final result because it directly affects the  $h$  (the step size),  $h = L/N$ , and the error depends on  $h$  with  $O(h^2)$ . This shows the correctness of our algorithm.

It clearly seems that our implementation for Helmholtz decomposition works appropriately. As we can see also our plot for final norms versus error results: we have increasing final norm of the divergence of the velocity profile with increasing error. The algorithm of correction is getting better with the increasing size of the problem as expected.

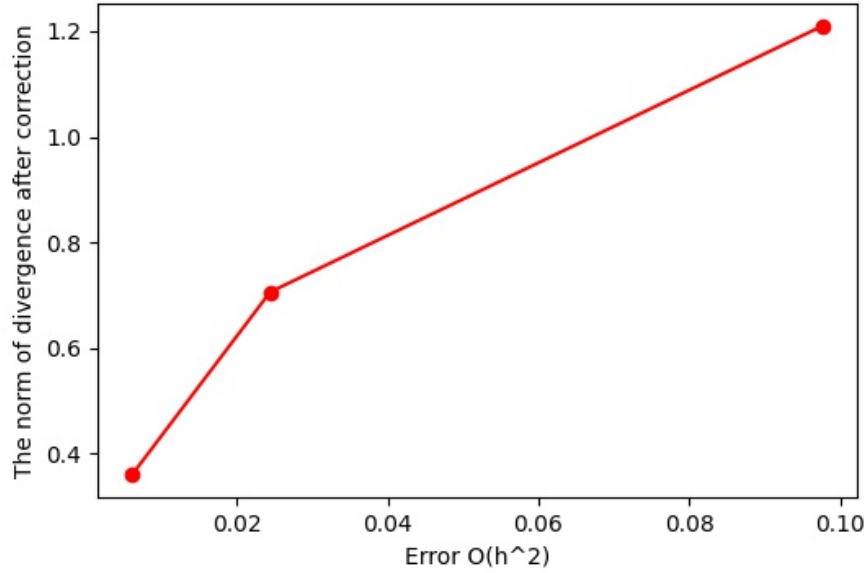


Figure 3.0.1: Final norms versus error results

After investigating the correctness of the implementation, we have investigated the parallelization performance of our project. To achieve this, the implementation has been done with the fixed size  $N = 128$  for the different numbers of processors 1, 2, 4, 8, 16, 32, and 64 to see the results and to show the strong scaling for our parallelization process.

As we know, in the case of strong scaling, the number of processors is increased while the problem size remains constant. The speedup achieved by increasing the number of processes usually decreases more or less continuously. In the following, you will see similar behavior in Figure 3.0.2 which is the strong scaling graph for our implementation. After 32 processors, the scaling starts to level off and that means we are about to reach the maximum speed-up. To see this level-off behavior better and decrease the speedup a larger problem size, for example,  $N=1024$  can be defined, in future implementations.

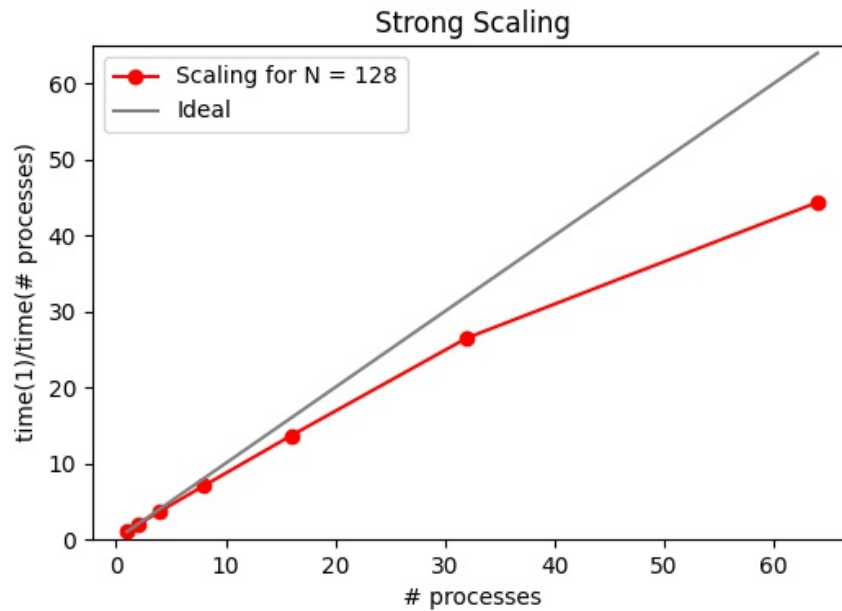


Figure 3.0.2: Strong Scaling results

# Bibliography

- [1] Task Definition Link. <https://moodle.uni-wuppertal.de/course/view.php?id=22366>. Accessed: 2022-07-15.
- [2] R.L. Sani P.M. Gresho. On pressure boundary conditions for incompressible navier-stokes equation. *International Journal for Numerical Methods in Fluids*, 7(2):1111–1145, 1987.
- [3] Robert L. Street Joel H. Ferziger, Milovan Perić. *Computational Methods for Fluid Dynamics*. Springer Nature Switzerland AG, Massachusetts, 2 edition, 2020.

# Appendix A

## The code

```
1 import numpy as np
2 import copy
3 import math
4 from mpi4py import MPI
5
6 def global_index(x):
7     return int(rank * (N/size) + x)
8
9 def communication_above(theone, above):
10     if rank < size-1:
11         comm.Send(theone, dest = rank+1, tag=13)
12
13     if rank > 0 and rank < size:
14         comm.Recv(above, source=rank-1, tag=13)
15
16
17 def communication_below(theone, below):
18     if rank > 0 and rank < size:
19         comm.Send(theone, dest = rank-1, tag=14)
20
21     if rank < size-1:
22         comm.Recv(below, source=rank+1, tag=14)
23
24 def divergence(ghost_below, vel_local, ghost_above):
25     for i in range(local_size):
26         if global_index(i) > 0 and global_index(i) < N-1:
27             for j in range(1, N-1):
28                 if i+1 > local_size-1 :
29                     div_vel_local[i][j] = (ghost_below[0][j] - vel_local[0][i-1, j] +
30                     vel_local[1][i, j+1] - vel_local[1][i, j-1])/(2*h)
31                 elif i-1 < 0:
```

```

31         div_vel_local[i][j] = (vel_local[0][i+1,j] - ghost_above[0][j] +
    vel_local[1][i,j+1] - vel_local[1][i,j-1])/(2*h)
32         else:
33             div_vel_local[i][j] = (vel_local[0][i+1,j] - vel_local[0][i-1,j] +
    vel_local[1][i,j+1] - vel_local[1][i,j-1])/(2*h)
34     return div_vel_local
35
36 #basics for parallelisations
37 comm = MPI.COMM_WORLD
38 size = comm.Get_size()
39 rank = comm.Get_rank()
40 status = MPI.Status()
41
42 L=20
43 n=127
44 N=n+1
45 local_size = int(N/size)
46 h=L/n
47
48
49 #Velocity definition
50 vel_local=np.array(((np.zeros((local_size,N))), (np.zeros((local_size,N))))))
51
52 x0 = -10
53 y0 = -10
54
55 for i in range(local_size):
56     for j in range(0,N):
57         vel_local[0][i,j] = (y0+h*j)/((x0+h*global_index(i))**2+(y0+h*j)**2+1)
    + 2*(x0+h*global_index(i)) * math.exp(-(x0+h*global_index(i))**2-(y0+h*j)**2)
58         vel_local[1][i,j] = (-x0-h*global_index(i))/((x0+h*global_index(i))
    **2+(y0+h*j)**2+1) + 2*(y0+h*j) * math.exp(-(x0+h*global_index(i))**2-(y0+h*j)**2)
59
60 comm.Barrier()
61 start = MPI.Wtime()
62 #Calculation of divergence of the initial velocity
63 ghost_below = np.zeros((1,N))
64 ghost_above = np.zeros((1,N))
65 sum1=np.array([[0.0]])
66 sumall=np.array([[0.0]])
67
68 communication_below(vel_local[0][0,:],ghost_below)
69 communication_above(vel_local[0][local_size-1,:],ghost_above)
70
71 div_vel_local = np.zeros((local_size,N))

```

```

72
73 divergence(ghost_below,vel_local, ghost_above)
74 sum1[0] = np.linalg.norm(div_vel_local)**2
75 comm.Allreduce(sum1, sumall, op=MPI.SUM)
76
77 if rank == 0:
78     print("Before = ",sumall**(1/2))
79
80 #Calculation of phi
81 phi_local = np.zeros((local_size,N))
82 phi_ghost_below = np.zeros((1,N))
83 phi_ghost_above = np.zeros((1,N))
84 phi_new_local = np.zeros((local_size,N))
85 c=1
86
87 tol = np.array([[1.0]])
88 while (1):
89     sum=np.array([[0.0]])
90     communication_below(phi_local[0][:],phi_ghost_below)
91     communication_above(phi_local[local_size-1][:],phi_ghost_above)
92     for i in range(local_size):
93         if global_index(i) > 0 and global_index(i) < N-1:
94             for j in range(1,N-1):
95                 if i+1 > local_size-1:
96                     phi_new_local[i][j] = h*h/4*(div_vel_local[i][j]) + 1/4*(phi_ghost_below[0][
97 j] + phi_local[i-1,j] + phi_local[i,j+1] + phi_local[i,j-1])
98                 elif i-1 < 0:
99                     phi_new_local[i][j] = h*h/4*(div_vel_local[i][j]) + 1/4*(phi_local[i+1,j] +
100 phi_ghost_above[0][j] + phi_local[i,j+1] + phi_local[i,j-1])
101                 else:
102                     phi_new_local[i][j] = h*h/4*(div_vel_local[i][j]) + 1/4*(phi_local[i+1,j] +
103 phi_local[i-1,j] + phi_local[i,j+1] + phi_local[i,j-1])
104                 sum[0] = sum[0] + ((phi_new_local[i][j]-phi_local[i][j])/(phi_local[i][j]+1e-40)
105 )**2
106
107 c=c+1
108 comm.Allreduce(sum, tol, op=MPI.SUM)
109 phi_local = copy.copy(phi_new_local)
110 if tol**(0.5)<1e-40:
111     break
112
113 #Calculation of gradient
114
115 grad_phi_local = np.array(((np.zeros((local_size,N))), (np.zeros((local_size,N)))))

```

```

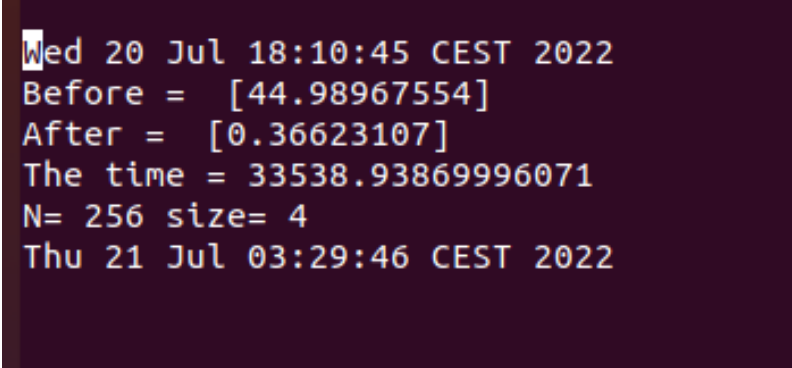
113 for i in range(local_size):
114     if global_index(i) > 0 and global_index(i) < N-1:
115         for j in range(1,N-1):
116             if i+1 > local_size-1 :
117                 grad_phi_local[1][i,j]=(phi_new_local[i,j+1]-phi_new_local[i,j-1])/(2*h)
118                 grad_phi_local[0][i,j]=(phi_ghost_below[0][j]-phi_new_local[i-1,j])/(2*h)
119             elif i-1 < 0 :
120                 grad_phi_local[1][i,j]=(phi_new_local[i,j+1]-phi_new_local[i,j-1])/(2*h)
121                 grad_phi_local[0][i,j]=(phi_new_local[i+1,j]-phi_ghost_above[0][j])/(2*h)
122             else :
123                 grad_phi_local[1][i,j]=(phi_new_local[i,j+1]-phi_new_local[i,j-1])/(2*h)
124                 grad_phi_local[0][i,j]=(phi_new_local[i+1,j]-phi_new_local[i-1,j])/(2*h)
125
126
127 #Calculation of new velocity
128 vel_new_local = np.array(((np.zeros((local_size,N))), (np.zeros((local_size,N))))))
129 vel_new_local[0]=np.add(vel_local[0], grad_phi_local[0])
130 vel_new_local[1]=np.add(vel_local[1], grad_phi_local[1])
131
132 div_vel_new_local = np.zeros((local_size,N))
133 vel_new_ghost_below = np.zeros((1,N))
134 vel_new_ghost_above = np.zeros((1,N))
135
136 sum2=np.array([[0.0]])
137 sumall=np.array([[0.0]])
138
139 communication_below(vel_new_local[0][0,:], vel_new_ghost_below)
140 communication_above(vel_new_local[0][local_size-1,:], vel_new_ghost_above)
141
142 div_vel_new = np.zeros((N,N))
143 div_vel_new_local = divergence(vel_new_ghost_below, vel_new_local, vel_new_ghost_above)
144
145 sum2[0] = np.linalg.norm(div_vel_new_local)**2
146 comm.Allreduce(sum2, sumall, op=MPI.SUM)
147
148 comm.Barrier()
149 end = MPI.Wtime()
150
151 if rank == 0:
152     print("After = ", sumall**(1/2))
153     print("The time =", end-start)
154     print("N=", N, "size=", size)

```

Listing A.1: Parallel code

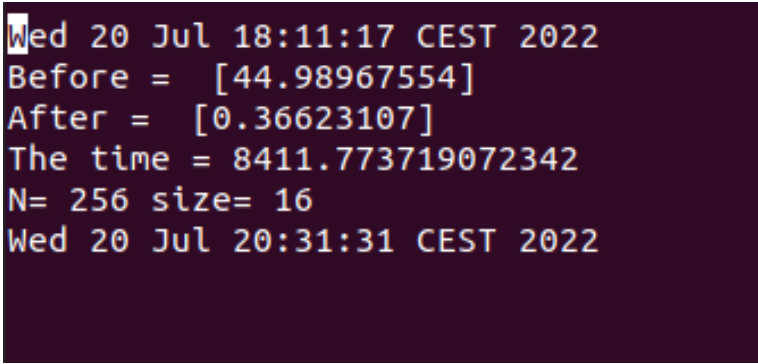
## Appendix B

### Some program outputs



```
Wed 20 Jul 18:10:45 CEST 2022
Before = [44.98967554]
After = [0.36623107]
The time = 33538.93869996071
N= 256 size= 4
Thu 21 Jul 03:29:46 CEST 2022
```

Figure B.0.1: Results for  $N = 256$ , processor = 4



```
Wed 20 Jul 18:11:17 CEST 2022
Before = [44.98967554]
After = [0.36623107]
The time = 8411.773719072342
N= 256 size= 16
Wed 20 Jul 20:31:31 CEST 2022
```

Figure B.0.2: Results for  $N = 256$ , processor = 16



```
Wed 20 Jul 18:11:43 CEST 2022  
Before = [44.98967554]  
After = [0.36623107]  
The time = 2251.2824420928955  
N= 256 size= 64  
Wed 20 Jul 18:49:18 CEST 2022
```

Figure B.0.3: Results for  $N = 256$ , processor = 64

```
Wed 20 Jul 17:25:14 CEST 2022  
Before = [22.0965123]  
After = [0.70684018]  
The time = 2008.607125043869  
N= 128 size= 4  
Wed 20 Jul 17:58:44 CEST 2022
```

Figure B.0.4: Results for  $N = 128$ , processor = 4

```
Wed 20 Jul 17:28:56 CEST 2022  
Before = [22.0965123]  
After = [0.70684018]  
The time = 540.0523068904877  
N= 128 size= 16  
Wed 20 Jul 17:37:59 CEST 2022
```

Figure B.0.5: Results for  $N = 128$ , processor = 16

```
Wed 20 Jul 17:30:52 CEST 2022  
Before = [22.0965123]  
After = [0.70684018]  
The time = 166.3463749885559  
N= 128 size= 64  
Wed 20 Jul 17:33:41 CEST 2022
```

Figure B.0.6: Results for  $N = 128$ , processor = 64