Parallelization Of An Experimental Multiphase Flow Algorithm

by

Ankit Mittal

in partial fulfilment of the requirements for the degree of

Master of Science

In Applied Mathematics

At the Delft University of Technology,
To be defended publicly on Friday August 12, 2016 at 03:00 PM.

Student number:	4503163	
Supervisors:	Dr. Martin van Gijzen,	TU Delft
	Dr. Aris Twerda	TNO-Netherlands
Thesis committee:	Prof. dr. Arnold Heemink,	TU Delft
	Dr. Duncan van der Heul	TU Delft

ACKNOWLEDGMENT

Firstly, I would like to thank my thesis supervisor Dr. Martin van Gijzen for the guidance, support, and encouragement that he offered me. I also thank Guido Oud and Dr. Aris Twerda for their guidance and for the numerous fruitful discussions we had. Without the above three people this thesis wouldn't have been possible. I would also like to thank Prof. dr. Kees Vuik and Dr. Duncan van der Heul for all the discussions and help that they kindly offered me.
I am grateful to Europe's Education, Audiovisual and Culture Executive Agency (EACEA) for providing me with the scholarship to pursue my master's education. Further, I am also grateful to TU Delft and TNO Netherlands for providing me with the resources to complete my master's thesis.
Above all, I would like to express my sincerest gratitude to God, my family and my friends for their love and belief in me. The moral support and love that I received from them made my journey easy and worthwhile.

Ankit Mittal

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Multiphase flows are widely occurring in nature. Such flows either have two or more immiscible fluids separated by an interface or one or more fluids, again separated by an interface, which are in different phases. They commonly occur in atmosphere (for example bubbly flows), chemical reactors, turbo-machines, fuel injectors, pipe flows, etc. Multiphase flows are also of particular interest to the petroleum industry, where such flows often occur in wells and pipelines during oil and gas production.
In multiphase flows, since the interface also gets advected with the flow, accurate calculation of the interface location is necessary for accurate prediction of the flow field. In order to calculate the interface at each time step one can use either the Level Set (LS) method [1] or the Volume of Fluid (VOF) method [2]. The LS method, though computationally cheap, is rather inaccurate, while the VOF method though more accurate, is computationally very expensive. In [3] van der Pijl studied bubbly flows, and to calculate the interface an approach based on the combination of the level set and volume of fluid methods (MCLS) was adopted. This approach gave more accurate results (as compared to the Level Set method) without incurring prohibitively large computational costs.
A project titled *CFD to study instabilities in 3d flows* is currently being carried out at TU Delft jointly with TNO Netherlands, Shell, and Deltares. The aim of the project is to study multiphase flows in pipes, where those are initially filled with oil, and water is pumped to flush the oil out or vice-versa. A somewhat similar approach as in [3] is employed here to predict the interface and a working code is available. The code extensively uses Krylov subspace methods to iteratively solve the obtained linear system of equations. The aim of this thesis is to improve the efficiency of the available code.
In multiphase flows, there is a jump in viscosity and density across the interface due to the difference in properties of the two fluids. This jump slows down the convergence of

the iterative solvers. The present code as well is plagued by the slow convergence and hence proper preconditioners to improve the convergence are analyzed in this endeavor. Also, we consider how to improve the performance of the given code by changing its structure, and by using less computation and memory intensive solvers.
As size of the system grows, even the best solvers with the most suitable preconditioners take prohibitively large amount of time to generate the numerical results on a single processor. The way-out is to use parallel programming and to split the whole big system into smaller more manageable pieces, and distribute them amongst the available processors. But parallelization is far from straightforward, as explained later, if the number of processors increases the convergence behavior of the preconditioner deteriorates. Hence, techniques like deflation have to be used to restore the performance of the preconditioner, and therefore in this endeavor we also study the feasibility of parallelization with the deflation technique.
Furthermore, we acknowledge that the decoupled time integration method used in the available code is not kinetic energy conserving, which might be of interest in turbulent flow cases. To preserve the kinetic energy, one has to integrate the flow field in a coupled manner. The coupled system suffers from a very poor convergence, and therefore we also consider a few preconditioners to improve the convergence of the coupled solvers.
In this report, Chapter 2 explores in detail the physics of incompressible multiphase flows, and Chapters 3 and 4 deal with solvers and preconditioners used in the available code or proposed to be used to improve the efficiency of the available code. Further, in Chapter 5 the structure of the available code is discussed, while Chapter 6 covers the parallelization study for the current system. In Chapter 7, we use the modified parallel code to simulate two physical problems to prove the accuracy of the new code and to discuss the obtained speedup and scaling results. This is followed by demonstarting the benefits of the modified parallel code by simulating a full scale engineering problem in Chapter 8. At the end, we present the conclusion in Chapter 9, and discuss the saddle point preconditioners in Appendix A.

Chapter 2

GOVERNING EQUATIONS & SOLUTION TECHNIQUES

In the present study, we deal with multiphase flows between two fluids which are separated by a sharp interface. The flow is characterized by velocity $\boldsymbol{v}(x, y, z, t)$ and pressure $p(x, y, z, t)$ (where bold indicates a vector). Due to the assumption of incompressibility, the fluids on either side have different but constant densities and viscosities, also we assume the flow to be isothermal and Newtonian.
In this section we present a very brief overview of the governing equations based on [3], for a more detailed discussion one is referred to [3, 4]. The physics of our problem is split into two distinct parts, the flow and the interface. In our approach they are treated separately, hence we present them one after the other. First we shall discuss the flow part.

2.1 FLOW

The flow is governed by the 3-d unsteady incompressible Navier-Stokes equations. In this study we are not interested in the thermal energy, and since in incompressible flows the energy equation is decoupled from the momentum and continuity equations we do not solve it. Also, since we are essentially dealing with pipe flows, the cylindrical coordinates system is an obvious choice for the coordinate system in which the Navier-Stokes equations are solved. The N-S equations in cylindrical coordinates are,

Continuity equation,

$$\frac{1}{r}\frac{\partial r u_r}{\partial r} + \frac{1}{r}\frac{\partial u_\theta}{\partial \theta} + \frac{\partial u_z}{\partial z} = 0$$

Radial momentum equation,

$$\frac{\partial u_r}{\partial t} + \frac{\partial (u_r u_r)}{\partial r} + \frac{1}{r}\frac{\partial (u_r u_\theta)}{\partial u_\theta} + \frac{\partial (u_r u_z)}{\partial z} + \frac{(u_r u_r - u_\theta u_\theta)}{r} = \\ -\frac{1}{\rho}\frac{\partial p}{\partial r} + g_r + \frac{\mu}{\rho}\left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u_r}{\partial r}\right) - \frac{u_r}{r^2} + \frac{1}{r^2}\frac{\partial^2 u_r}{\partial \theta^2} - \frac{2}{r^2}\frac{\partial u_\theta}{\partial \theta} + \frac{\partial^2 u_r}{\partial z^2}\right]$$

Angular momentum equation,

$$\frac{\partial u_\theta}{\partial t} + \frac{\partial (u_\theta u_r)}{\partial r} + \frac{1}{r}\frac{\partial (u_\theta u_\theta)}{\partial \theta} + 2\frac{u_r u_\theta}{r} + \frac{\partial (u_\theta u_z)}{\partial z} = \\ -\frac{1}{\rho r}\frac{\partial p}{\partial \theta} + g_\theta + \frac{\mu}{\rho}\left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u_\theta}{\partial r}\right) - \frac{u_\theta}{r^2} + \frac{1}{r^2}\frac{\partial^2 u_\theta}{\partial \theta^2} + \frac{2}{r^2}\frac{\partial u_r}{\partial \theta} + \frac{\partial^2 u_\theta}{\partial z^2}\right]$$

Axial momentum equation,

$$\frac{\partial u_z}{\partial t} + \frac{\partial (u_r u_z)}{\partial r} + \frac{1}{r}\frac{\partial (u_\theta u_z)}{\partial \theta} + \frac{\partial u_z u_z}{\partial z} + \frac{(u_r u_z)}{r} = \\ -\frac{1}{\rho}\frac{\partial p}{\partial z} + \rho g_z + \frac{\mu}{\rho}\left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u_z}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 u_z}{\partial \theta^2} + \frac{\partial^2 u_z}{\partial z^2}\right]$$

where ρ is the density, p is the pressure, g is the gravitational constant, r, θ, z are the space variables, and u_r, u_θ, u_z are the velocity components in respective r, θ, z directions. If we mark the two fluids as 0 and 1, then to separate the two fluid regimes, we introduce a so-called color function χ defined as
$\chi(\boldsymbol{x})) = \begin{cases} 0, & \boldsymbol{x} \in \text{fluid } 0 \\ 1, & \boldsymbol{x} \in \text{fluid } 1 \end{cases}$, where $\boldsymbol{x}$ is the position vector. Then the density and viscosity can be expressed as

$$\rho = \rho_0 + (\rho_1 - \rho_0)\Psi \qquad \& \qquad \mu = \mu_0 + (\mu_1 - \mu_0)\chi$$

where subscripts 0 and 1 indicate the respective fluids, and Ψ is the VOF function (discussed in the next subsection). To get a smooth pressure and gradient of velocity across the interface we regularize the color function, i.e., smear it out over a small but finite distance.

2.2 INTERFACE

For the interface convection we follow the Eulerian approach and only look at a fixed space with a fixed grid. For numerical treatment of the interface we use the Volume

Tracking methods. In these methods we assign a color to each of the fluid regimes, and the region where the color function changes implicitly defines the interface. The benefit of such a method is that the changes in the interface topology and coalescence are automatically taken care of.
The volume tracking methods can be sub-categorized into two subcategories viz. the Level Set (LS) method [1] and the Volume of Fluid (VOF) method [2]. In both the LS and VOF methods the fluid interface is identified by some coloring function, and that function is advected in an Eulerian way as,

$$\frac{\partial \Phi}{\partial t} + \boldsymbol{u} \cdot \nabla(\Phi) = 0. \tag{2.1}$$

We further discuss each of the above two techniques in some detail.

2.2.1 LEVEL SET METHOD

In the Level Set method [1], the interface is defined by a marker function Φ. The marker function is defined to be positive in one fluid and negative in the second fluid. Hence, the locations where the marker function is zero marks the location of the interface, i.e.,

$$\text{Interface} \;=\; \{\boldsymbol{x} \mid \Phi(\boldsymbol{x}, t) = 0\}.$$

The signed distance function $d(\boldsymbol{x}, t)$ is a well suited marker function. The level set function is advected according to

$$\frac{\partial \Phi}{\partial t} + \boldsymbol{u} \cdot \nabla \Phi = 0.$$

The level set function Φ is a smooth function that, unlike VOF, allows for a straightforward calculation of the interface curvature. The main disadvantage is that if the interface is advected using this approach, the mass is not conserved due to the viscous and convective smoothing. Another disadvantage of LS is that when it is advected through a non-uniform flow it may no longer correspond to a distance function. To remedy this one has to reinitialize the level set function after some, or every, numerical integration step(s) depending on some criterion.

2.2.2 VOLUME OF FLUID METHOD

In the VOF method [2], we use the volume of fluid function Ψ to implicitly define the location of the interface. This function measures the fractional volume of a certain fluid

in a computational cell. Ψ can be 0 or 1 or somewhere in between if the computational cell is filled with both the fluids, i.e., it contains an interface. The cells which contain the interface are called mixed cells. We mathematically define Ψ for each cell as

$$\Psi(\boldsymbol{x}_k) \;=\; \frac{1}{\Omega_k}\int_{\Omega_k} \chi d\Omega,$$

where χ is the color function which is 0 in one fluid and 1 in the second fluid, $\boldsymbol{x}_k$ is the node and Ω_k is the volume of the corresponding computational cell k. The interface is advected using

$$\frac{\partial \chi}{\partial t} + \boldsymbol{u}\cdot\nabla(\chi) = 0.$$

The advantage of the VOF technique is that it is mass conserving, unlike the LS method. The main disadvantage is that the evaluation of normals and curvatures, and the interface reconstruction are much more tedious and computationally expensive.
Because of the advantages and disadvantages of both the methods, there is no clear choice amongst them. In the given code a hybrid of both is used which, while being mass conserving, is relatively easy to evaluate. The details of this hybrid model are not discussed here and one is referred to [3] for more information.

2.3 DISCRETIZATION AND LINEARIZATION

In the present study, we use the Finite Difference Method (FDM) to discretise the continuous Navier-Stokes equations, and since the collocated storage of variables give rise to an odd-even decoupling which introduces spurious oscillations into the solution called the Checkerboard modes, the variables are stored in the Arakawa C grid [5], also known as staggered grid. In the Arakawa C grid, the pressure is stored at the center while various velocity components are stored at the respective cell faces. Figure 2.1 shows the locations where the variables are defined for each cell in a 2d domain.
In such a description we solve the continuity equation at the center of each cell, while the momentum equation is solved at the cell boundaries.
The methodology for the space discretization is similar to the one used by Morinishi et al. [4]. Discretization of the viscous terms is done using the standard second order scheme employing a three point stencil (in 1-d), similarly for the convective terms a second order central scheme is employed. If we require variables at locations where they are not defined, we approximate them by averaging the variables at the known locations, for example, if we require $u_{i,j}$ (Figure 2.1) we approximate it as

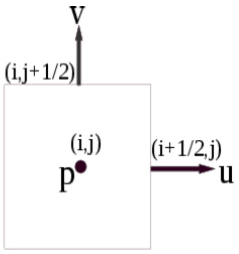


Figure 2.1: Arrangement of variables in Arakawa C grid.

$$u_{i,j} \;=\; \frac{u_{i-\frac{1}{2},j} \;+\; u_{i+\frac{1}{2},j}}{2}.$$

For the time integration of the momentum equation, the second order implicit mid-point method is used, while for the interface advection a first order explicit method is employed. One important aspect of the implicit time integration is the linearization of the convective terms. In the present scheme Newton linearization [6] is used, which can be described as follows.
Let u and v be any two variables which are function of some variable. Then by Tailor series

$$(uv)^{n+1} \;=\; (uv)^n + \left(\frac{\partial uv}{\partial s}\right)^n \Delta s + O(\Delta s^2) \;=\; (uv)^n + \left(u\frac{\partial v}{\partial s} + v\frac{\partial u}{\partial s}\right)^n \Delta s + O(\Delta s^2),$$

which can be further expressed as

$$(uv)^{n+1} \;=\; (uv)^n + \left(v^n\frac{(u^{n+1}-u^n)}{\Delta s} + u^n\frac{(v^{n+1}-v^n)}{\Delta s}\right)\Delta s + O(\Delta s^2).$$

Neglecting $O(\Delta s^2)$ terms gives the Newton linearization

$$(uv)^{n+1} \;=\; -\,(uv)^n + v^n u^{n+1} + u^n v^{n+1}.$$

After the space-time discretization of the Navier-Stokes equations, we get a system of non-linear equations, i.e., the discrete continuity and momentum equations which are defined as follows:

Discrete continuity equation

$$B\boldsymbol{u}^{n+1} \;=\; \boldsymbol{g},$$

where $\boldsymbol{g}$ contains the discrete velocity boundary conditions and B is defined as

$$B \;=\; \begin{bmatrix} B_r & B_\theta & B_z \end{bmatrix},$$

where $B_r,\ B_\theta,\ B_z$ the discrete divergence operators corresponding to the discrete velocities in the radial, angular and axial directions.

Discrete momentum equation

$$\frac{\boldsymbol{u}^{n+1}-\boldsymbol{u}^{n}}{\Delta t}\ +\ \bar{\hat{F}}\left(\boldsymbol{u}^{n+1}\right)\ =\ -\frac{1}{\rho^{n+\frac{1}{2}}}G\boldsymbol{p}^{n+\frac{1}{2}}+\left(\frac{1}{\rho}\boldsymbol{f}_s\right)^{n+\frac{1}{2}}+\boldsymbol{h}^{n+\frac{1}{2}} \tag{2.2}$$

where $\bar{\hat{F}}\left(\boldsymbol{u}^{n+1}\right)$ contains only the diffusion and non-linear convection terms.

After linearizing the convective terms as indicated above we get

$$\frac{\boldsymbol{u}^{n+1}-\boldsymbol{u}^{n}}{\Delta t}\ +\ \hat{F}\boldsymbol{u}^{n+1}\ =\ -\frac{1}{\rho^{n+\frac{1}{2}}}G\boldsymbol{p}^{n+\frac{1}{2}}+\boldsymbol{\tau}^{n}+\left(\frac{1}{\rho}\boldsymbol{f}_s\right)^{n+\frac{1}{2}}+\boldsymbol{h}^{n+\frac{1}{2}} \tag{2.3}$$

The matrix $\hat{F}$ is of the form

$$\hat{F}\ =\ \begin{bmatrix}\hat{F}_r & 0 & 0\\ 0 & \hat{F}_\theta & 0\\ 0 & 0 & \hat{F}_z\end{bmatrix},$$

where $\hat{F}_r,\ \hat{F}_\theta,\ \hat{F}_z$ are derived from the linearized implicit time discretization of the radial, angular and axial momentum equations respectively and contain only the convective and diffusive terms. G is the discrete gradient operator of the form

$$G\ =\ \begin{bmatrix}G_r\\ G_\theta\\ G_z\end{bmatrix}.$$

As above G_r, G_θ, G_z are the discrete gradient operators obtained from the discretization of the radial, angular and axial momentum equations respectively. Finally, $\rho^{n+\frac{1}{2}}$, $\boldsymbol{f}_s^{n+\frac{1}{2}}$, $\boldsymbol{h}^{n+\frac{1}{2}}$ in equations (2.2) and (2.3) are the discrete density, surface tension, and body force respectively calculated based on the known location of the interface at $n+\frac{1}{2}$ time level, and $\boldsymbol{\tau}^n$ in equation (2.3) contains all the terms which appear due to linearization. Moreover $\boldsymbol{p}^{n+\frac{1}{2}}=\boldsymbol{p}^{n+1}$, the reason for writing pressure at $n+\frac{1}{2}$ will be clear in the following section. Further, it is to be noted that for our discretization the discrete gradient operator G is equal to the transpose of the discrete divergence operator B, hence we can replace G by B^T in the above equations.

Clubbing the above mentioned linearized discrete momentum and discrete continuity equations together we get

$$A\boldsymbol{x}\ =\ \begin{bmatrix}F & B^T\\ B & 0\end{bmatrix}\begin{bmatrix}\boldsymbol{u}\\ \boldsymbol{p}\end{bmatrix}\ =\ \begin{bmatrix}\boldsymbol{f}\\ \boldsymbol{g}\end{bmatrix}\ =\ \boldsymbol{b}, \tag{2.4}$$

where F contains the contributions from $\hat{F}$ and $\frac{1}{\Delta t}I$. We must solve the above system (2.4) at each time step to obtain a time dependent solution.

2.4 TIME INTEGRATION METHODOLOGY

We now discuss the methodology for the time integration which until now we assumed to be given. The time integration is split into two decoupled parts, the flow integration and the interface advection. The calculations of flow and interface are staggered in time, i.e., we first calculate the flow field at the new time step based on the current interface position and then calculate the new position of the interface based on the current flow field; Chapter 5 discusses the algorithm in detail.
What is to be stressed at this junction is the time integration technique for the flow field used in the available code. Although an algorithm is presented later, here we present the mathematical motivation, disadvantages and remedies of the used integration methodology. As discussed above, the interface is assumed to be given for the flow field time integration. To obtain the velocity and pressure at the new time step one has to solve the discrete system (2.4). The system can be solved either in a decoupled manner (solve for the velocity and pressure separately) or in a coupled fashion. We further discuss the decoupled (pressure-correction/projection) scheme implemented in the available code.
In the decoupled solver we have to solve for velocity from equation (2.3)

$$\frac{\boldsymbol{u}^{n+1}-\boldsymbol{u}^{n}}{\Delta t} \; + \; \hat{F}\boldsymbol{u}^{n+1} \;=\; -\frac{1}{\rho^{n+\frac{1}{2}}}B^{T}\boldsymbol{p}^{n+\frac{1}{2}}+\boldsymbol{\tau}^{n}+\left(\frac{1}{\rho}\boldsymbol{f}_{s}\right)^{n+\frac{1}{2}}+\boldsymbol{h}^{n+\frac{1}{2}}. \tag{2.5}$$

We can not solve the above system directly since $\boldsymbol{p}^{n+\frac{1}{2}}$ is not known. Hence the following scheme is adopted:
Solve

$$\frac{\hat{\boldsymbol{u}}-\boldsymbol{u}^{n}}{\Delta t} \; + \; \hat{F}\hat{\boldsymbol{u}} \;=\; -\frac{1}{\rho^{n-\frac{1}{2}}}B^{T}\boldsymbol{p}^{n-\frac{1}{2}}+\boldsymbol{\tau}^{n}+\left(\frac{1}{\rho}\boldsymbol{f}_{s}\right)^{n-\frac{1}{2}}+\boldsymbol{h}^{n-\frac{1}{2}} \tag{2.6}$$

Since the above equation is solved with pressure at previous time step $\hat{u}$ is not solenoidal. But we want u^{n+1} to be divergence-free, hence we subtract equation (2.6) from equation (2.3) to get

$$\frac{\boldsymbol{u}^{n+1}-\hat{\boldsymbol{u}}}{\Delta t} \;=\; -\frac{1}{\rho^{n+\frac{1}{2}}}B^{T}\boldsymbol{p}^{n+\frac{1}{2}}+\left(\frac{1}{\rho}\boldsymbol{f}_{s}\right)^{n+\frac{1}{2}}+\frac{1}{\rho^{n-\frac{1}{2}}}B^{T}\boldsymbol{p}^{n-\frac{1}{2}}-\left(\frac{1}{\rho}\boldsymbol{f}_{s}\right)^{n-\frac{1}{2}}. \tag{2.7}$$

It can be shown [7] that if a second order time integration method is used, neglecting $\hat{F}\boldsymbol{u}^{n+1} - \hat{F}\hat{\boldsymbol{u}}$ in the above equation will still give a second order approximation if we use a good enough approximation of pressure in equation (2.6). Further, since we do not deal with time dependent body force $h^{n+\frac{1}{2}} = h^{n-\frac{1}{2}}$.
Now, all that remains is to find the pressure which will give a solenoidal velocity. From the continuity equation we have $B\boldsymbol{u}^{n+1} = \boldsymbol{g}$, hence we can derive an equation for the pressure by taking the discrete divergence of equation (2.7):

$$
\begin{aligned}
-B\frac{1}{\rho^{n+\frac{1}{2}}}B^T\boldsymbol{p}^{n+\frac{1}{2}} \;=\; & \\
& B\left(-\frac{1}{\Delta t}\hat{\boldsymbol{u}} - \left(\frac{1}{\rho}\boldsymbol{f}_s\right)^{n+\frac{1}{2}} - \frac{1}{\rho^{n-\frac{1}{2}}}B^T\boldsymbol{p}^{n-\frac{1}{2}} + \left(\frac{1}{\rho}\boldsymbol{f}_s\right)^{n-\frac{1}{2}} + \frac{1}{\Delta t}\boldsymbol{g}\right). \quad (2.8)
\end{aligned}
$$

The equations (2.6), (2.8) and (2.7) are known as the predictor, Poisson and corrector equations respectively. From the above discussion we get a time integration technique, i.e., first predict the velocity based on the pressure at the previous time step, then solve the Poisson equation to find the pressure at the current time step which will give a divergence-free velocity, and then finally update the velocity at the current time step by using the corrector equation. The reason for writing $\boldsymbol{p}^{n+\frac{1}{2}}$ is simply because the pressure is calculated *in between* the velocity updates at n^{th} and $n+1^{th}$ time step. This integration technique is known as the projection scheme or the pressure correction scheme, the algorithm used in the current code is presented in Chapter 5.
The current procedure of solving though is robust, computationally *cheap* and widely used, does not preserve kinetic energy of the fluid. Conservation of kinetic energy of the fluid becomes a topic of interest in turbulent flows where energy is transferred between different eddies. In a turbulent case if the kinetic energy is not conserved, the size of eddies will not be accurately predicted giving an overall inaccurate flow.
To preserve the kinetic energy, one has to solve the non-linear system (2.2) in a coupled manner i.e., simultaneously solve for both the velocity and pressure. Solving such a non-linear system is very expensive, and hence in this endeavor, we study techniques to solve the linearized system ((2.4)/(2.3)) in a coupled manner. This, though not kinetic energy conserving, is a step towards it and should give a more accurate result. System (2.4) gives rise to a saddle point problem being solved:

$$
A\boldsymbol{x} \;=\; \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix}\begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{p} \end{bmatrix} \;=\; \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix} \;=\; \boldsymbol{b}.
$$

The matrix A has a zero block on the diagonal and hence is a saddle point matrix. Solving such a coupled system is much more expensive than solving the pressure-correction scheme. Therefore in the last century, the computational power severely

limited the applications for which a coupled system could be solved. The present day computers, however, are much faster, making the solution of the coupled systems more practical to obtain.

To solve a coupled system we can either use the direct solvers or the iterative solvers. The direct solvers are very expensive for large systems and hence will not be discussed any further. The iterative solvers can be further classified into the segregated (not to be confused with the earlier discussed decoupled solver) and the coupled methods. In the segregated methods, velocity and pressure are solved separately one after the other, the order in which they are solved differs amongst different solvers. The idea is to solve two smaller problems one for each velocity and pressure. Coupled methods, on the other hand, solve the complete system simultaneously.

Iterative methods to solve the saddle point problem, especially the Krylov subspace methods, without a suitable preconditioner suffer from a terribly slow convergence due to the presence of a zero diagonal block in matrix A which makes it highly indefinite. Thus proper preconditioning must be used to get a higher efficiency from such solvers. The preconditioners for a saddle point problem tend to blur the distinction between the segregated and coupled methods since the preconditioners for the coupled methods are often based on the segregated methods. We shall discuss the appropriate preconditioners in Appendix A.

Chapter 3

SOLVERS

The choice of the solver plays a vital role in achieving the desired speedup. Improving the performance of the code is not just about parallel programming but also about using smart solvers that are more suitable for the problem in hand, i.e., a solver which converges quickly with relatively small computational and storage complexities. To solve a problem of the form $A\boldsymbol{x} = \boldsymbol{b}$, a whole plethora of solvers is available ranging from direct to iterative. Direct solvers generally decompose the matrix A into product of matrices which are easier to invert, for example, the Gaussian Elimination [8] method splits the matrix A into a product of lower (L) and upper (U) triangular matrices. These matrices are relatively easy to invert using the forward and backward substitution. The main drawback, however, is in the time complexity involved in forming these lower and upper triangular matrices, as a matrix of size $N \times N$ requires $O(N^3)$ floating point operations (flops). Although there exist solvers which work more efficiently than Gaussian Elimination, they still are too expensive for large N.
The alternative to the direct methods are the so-called iterative methods. In these methods, unlike the direct methods, one does not compute the solution exactly, rather the solution is updated iteratively and if the current solution satisfies the convergence criterion the iterations are stopped [8]. The definiteness of a matrix, plays a vital role in the performance of the iterative methods [8]. For the systems having highly indefinite matrices, the convergence of an iterative method deteriorates drastically. We can improve the convergence by preconditioning the matrix [9], this will be discussed later.
The Krylov subspace methods [8] are probably the most used subclass of iterative solvers. In these methods we look for the solution of $A\boldsymbol{x} = \boldsymbol{b}$ in the Krylov subspace $K_i(A, \boldsymbol{v})$ which is

$$K_i(A, \boldsymbol{v}) \;=\; span\{\boldsymbol{v}, A\boldsymbol{v}, A^2\boldsymbol{v}, ..., A^{(i-1)}\boldsymbol{v}\},$$

where $\boldsymbol{v} \in \mathbb{R}^n$ is a suitably chosen vector. A vector $\boldsymbol{v}$ is said to be of grade d with

respect to A if d is the smallest integer for which the set $\{\boldsymbol{v}, A\boldsymbol{v}, A^2\boldsymbol{v}, ..., A^d\boldsymbol{v}\}$ is linearly dependent. Since $d \leq n$ the Krylov subspace methods converge in at-most n iterations [8], but practically these methods converge much before n iterations. Further we discuss some of the solvers which are either used in the available code, or can be used to increase its performance.

3.1 RESTARTED GMRES

GMRES [10] is a Krylov subspace method which is widely used to iteratively solve $A\boldsymbol{x} = \boldsymbol{b}$ where A is a non hermitian matrix. In this section, we discuss the GMRES method in detail.
In the GMRES method, the solution $\boldsymbol{x}$ is approximated by $\boldsymbol{x}_i \in \boldsymbol{x}_0 + K_i(A, \boldsymbol{r}_0)$, where $\boldsymbol{x}_0$ is the initial guess, $K_i(A, \boldsymbol{r}_0)$ is the Krylov subspace, and $\boldsymbol{r}_0$ is the residual (defined by $\boldsymbol{r}_i = \boldsymbol{b} - A\boldsymbol{x}_i$) of the initial guess. The GMRES method reduces to finding $\boldsymbol{x}_i$ in each iteration such that it minimizes the residue $\boldsymbol{r}_i$ which lives in the subspace $AK_i(A, \boldsymbol{r}_0)$. This gives

$$\boldsymbol{x}_i = \boldsymbol{x}_0 + \boldsymbol{y},\ \boldsymbol{y} \in K_i(A, \boldsymbol{r}_0), \quad \text{where } K_i(A, \boldsymbol{r}_0) = span\{\boldsymbol{r}_0, A\boldsymbol{r}_0, ...A^{(i-1)}\boldsymbol{r}_0\}.$$

Choosing the Krylov subspace defined as above does not generally give a stable algorithm, therefore we use the Gram-Schmidt type vectors as the orthogonal basis of $K_i(A, \boldsymbol{r}_0)$. We form the orthogonal basis vectors using the Arnoldi decomposition:

$$\begin{gathered}\hat{\boldsymbol{v}}_{i+1} = A\boldsymbol{v}_i - \textstyle\sum_{j=1}^{i}(A\boldsymbol{v}_i, \boldsymbol{v}_j)\boldsymbol{v}_j, \\ \boldsymbol{v}_{i+1} = \hat{\boldsymbol{v}}_{i+1}/\|\hat{\boldsymbol{v}}_{i+1}\|, \quad h_{ji} = (A\boldsymbol{v}_i, \boldsymbol{v}_j), \quad \boldsymbol{v}_1 = \boldsymbol{r}_0.\end{gathered}$$

This gives,

$$A\left[\boldsymbol{v}_1\ \boldsymbol{v}_2\ \ldots\ \boldsymbol{v}_i\right] = \left[\boldsymbol{v}_1\ \boldsymbol{v}_2 \ldots \boldsymbol{v}_i\ \boldsymbol{v}_{i+1}\right] \begin{bmatrix} h_{11} & h_{12} & \cdots & \\ \|\hat{\boldsymbol{v}}_2\| & h_{22} & \cdots & \\ 0 & \|\hat{\boldsymbol{v}}_3\| & \cdots & \\ 0 & 0 & \ddots & \\ 0 & 0 & 0 & \|\hat{\boldsymbol{v}}_{i+1}\| \end{bmatrix}.$$

In matrix form we get $AV_i = V_{i+1}H_{i+1,i}$, where

$$H_{i+1,i} = \begin{bmatrix} h_{11} & h_{12} & \cdots & \\ \|\hat{\boldsymbol{v}}_2\| & h_{22} & \cdots & \\ 0 & \|\hat{\boldsymbol{v}}_3\| & \cdots & \\ 0 & 0 & \ddots & \\ 0 & 0 & 0 & \|\hat{\boldsymbol{v}}_{i+1}\| \end{bmatrix} \text{ and } V_i = [\boldsymbol{v}_1\ \boldsymbol{v}_2\ \ldots\ \boldsymbol{v}_i].$$

$H_{i+1,i} \in \mathbb{R}^n$ is the unreduced upper Hessenberg matrix. If $\boldsymbol{r}_0$ is of grade d with respect to A we get $AV_d = V_dH_{dd}$, where $H_{dd} \in \mathbb{R}^{d\times d}$ is given by

$$H_{d,d} = \begin{bmatrix} h_{11} & h_{12} & \cdots & & \\ \|\hat{\boldsymbol{v}}_2\| & h_{22} & \cdots & & \\ 0 & \|\hat{\boldsymbol{v}}_3\| & \ddots & & \\ 0 & 0 & \ddots & \ddots & \\ 0 & 0 & 0 & \|\hat{\boldsymbol{v}}_d\| & h_{dd} \end{bmatrix}.$$

GMRES finds an $\boldsymbol{x}_i \in \boldsymbol{x}_0 + K_i(A, \boldsymbol{r}_0)$ in each iteration i such that the residual at that iteration is minimized, i.e., find $\boldsymbol{x}_i$ such that $\|\boldsymbol{r}_i\|_2 = \min_{\boldsymbol{x}_i \in \boldsymbol{x}_0 + K_i(A,\boldsymbol{r}_0)} \|\boldsymbol{b} - A\boldsymbol{x}_i\|_2$. The above equation leads to finding a vector $\boldsymbol{t}_i \in \mathbb{R}^i$ such that it minimizes $\|V_{i+1}(\|r_0\|_2 \boldsymbol{e}_1 - H_{i+1,i}\boldsymbol{t})\|_2$. If we form the QR decomposition of the Hessenberg matrix as $H_{i+1,i} = QR$, where $Q \in \mathbb{R}^{(i+1)\times(i+1)}$ is an orthogonal matrix and $R \in \mathbb{R}^{(i+1)\times i}$ is an upper triangular matrix, we get the GMRES method as

$$\min_{\boldsymbol{t}_i \in \mathbb{R}^i} \|Q^H \boldsymbol{e}_1 \|\boldsymbol{r}_0\|_2 - \begin{bmatrix} R_i \\ 0 \end{bmatrix} \boldsymbol{t}_i \|_2.$$

Choosing $\boldsymbol{t}_i = \|r_0\|_2 R_i^{-1} \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_i \end{bmatrix}$ gives us the solution to the above problem, with the minimum residual equal to $|q_{i+1}|\ \|\boldsymbol{r}_0\|_2$ for each iteration. Now that we have discussed the GMRES method, it shall be prudent to discuss its convergence properties. By analyzing the GMRES method as a polynomial problem, one can derive a bound on the residue at each iteration:

$$\frac{\|\boldsymbol{r}_i\|_2}{\|\boldsymbol{r}_0\|_2} \leq \inf_{p_i \in P_i} \kappa(X) \max_{\lambda \in \sigma(A)} \|p_i(\lambda)\|_2$$

where P_i are the polynomials of degree i having $P_i(0) = 1$, and $\kappa(X)$ is the conditioning number of X where X is the matrix of eigenvectors of A and is obtained from the diagonalization of A (assuming that A is diagonalizable). The above bound shows that the reduction in residual per iteration is large if

1. Conditioning number $\kappa(X)$ is small, i.e., A is nearly normal.
2. Eigenvalues of A are clustered far away from the origin.

One drawback of GMRES is that all the Arnoldi vectors need to be stored for computation of the solution vector. If the dimension of the system is large, this results in an excessive computational and storage overhead. To avoid this, the Restarted GMRES method can be employed in which, after some predefined iterations, all the Arnoldi vectors are deleted and the iterations are restarted with the available solution as the initial guess. In the present code the GMRES method is used to solve the implicit predictor step, and the iterations are restarted after every 50 internal iterations.

3.2 CONJUGATE GRADIENT (CG)

The Conjugate Gradient (CG) method [11] is another Krylov subspace method, and is the most preferred solver for the systems having Symmetric Positive Definite (SPD) matrices. In the CG method, as in the GMRES method, we express the approximate solution as a member of the Krylov subspace and form the Krylov subspace by spanning the space generated by the orthogonal Gram-Schmidt style vectors. But because the matrix is symmetric, the Gram-Schmidt style vectors reduces to Lanczos vectors and we get an SPD Hessenberg matrix giving a short recursion formula, hence we need not store all the Lanczos vectors. A similar analysis as GMRES can be easily performed for the CG method and an elegant algorithm can be easily derived, for brevity the CG algorithm is not presented in this report.
We further discuss the convergence properties of the CG method. Useful bounds on the error for the CG method can be derived by approximating it as a polynomial problem. We get

$$\frac{\|\boldsymbol{x} - \boldsymbol{x}_i\|_A}{\|\boldsymbol{x} - \boldsymbol{x}_0\|_A} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i,$$

where $\kappa = \kappa(A)$ is the conditioning number of matrix A, and $\boldsymbol{x}$ is the actual solution. This convergence bound tells us that the CG method will converge much faster if the matrix is well conditioned. More details about CG can be found in [8, 11].

3.3 IDR(s)

IDR(s) is another method to solve $A\boldsymbol{x} = \boldsymbol{b}$, where A is a general matrix, which has certain advantages over GMRES. As discussed above, the GMRES method has an ever increasing depth of recursion with the number of iterations, i.e., the amount of computational work increase as the number of iterations increase, also the memory requirements scale with the number of iterations. Though restarting of GMRES helps to partially alleviate these two problems it also slows down the convergence of the GMRES method, which may lead to an increase in the total computational work.
The IDR(s) method due to Sonneveld, P. & van Gijzen, M.B. [12] has the same number of matrix vector products per iteration as GMRES (which is one) but has a fixed depth of recursion $(s + 1)$ which is smaller than GMRES, hence the overhead of IDR(s) is lower [12]. A short recurrence in GMRES (by restarting), as indicated earlier, seriously affects the convergence property of the method as the finite termination property is lost. While for IDR(s), as we will see shortly, a short recurrence is not achieved at the cost of finite termination. Hence, the IDR(s) method can be used instead of the

GMRES methods to reduce the computational cost. Below we discuss the IDR(s) algorithm.

IDR(s) Theorem (4.1): Consider any matrix $A \in \mathbb{R}^{N\times N}$ and non-zero vector $\boldsymbol{v}_0 \in \mathbb{R}^N$, and let $G_0 \in K^N(A, \boldsymbol{v}_0)$. Let S be a proper subspace of $\mathbb{R}^N$, such that S and G_0 does not share a nontrivial invariant subspace of A, and for nonzero $\omega_j' s$ define sequence $G_j = (I - \omega_j A)(G_{j-1} \cap S)$, $j = 1, 2, 3, 4, \ldots$. Then the following holds,
1. $G_j \subset G_{j-1} \forall j > 0$.
2. $G_j = 0$ for some $j \leq N$.
The proof of this theorem can be found in [12]. The above theorem can be applied by generating residuals $\boldsymbol{r}_k$ which are forced to live in the subspace G_j, here j is non-decreasing with k. This implies that the system will be solved in at-most N iterations. The residual $\boldsymbol{r}_{k+1}$ belongs to G_{j+1} if

$$\boldsymbol{r}_{k+1} = (I - w_{j+1}A)\boldsymbol{v}_k \text{ where } \boldsymbol{v}_k \in G_j \cap S.$$

Now if we choose,

$$\boldsymbol{v}_k = \boldsymbol{r}_k - \sum_{i=1}^{\hat{l}} \gamma_i \Delta \boldsymbol{r}_{k-i} \text{ we get}$$

$$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \omega_{j+1}A\boldsymbol{v}_k - \sum_{i=1}^{\hat{l}} \gamma_i \Delta \boldsymbol{r}_{k-i} = \boldsymbol{v}_k - \omega_{j+1}A\boldsymbol{v}_k$$

This is similar to the residual in the general Krylov subspace methods [12]. Let us assume S to be in the left nullspace of some $N \times s$ matrix T. Now, since $\boldsymbol{v}_k \in G_j \cap S \in S$ we have $T^H \boldsymbol{v}_k = 0$. It follows that we get a linear system of size $s \times \hat{l}$ for $\hat{l}$ coefficients γ_i, and the system is uniquely solvable if $\hat{l} = s$. Hence, the first vector in G_j requires s+1 vectors in G_{j-1} and $\boldsymbol{r}_k$ lies in G_j only if $k \geq j(s+1)$. Defining

$$\Delta R_k = (\Delta \boldsymbol{r}_{k-1}, \Delta \boldsymbol{r}_{k-2}, \ldots\ldots, \Delta \boldsymbol{r}_{k-s}) \text{ and}$$

$$\Delta X_k = (\Delta \boldsymbol{x}_{k-1}, \Delta \boldsymbol{x}_{k-2}, \ldots\ldots, \Delta \boldsymbol{x}_{k-s}) \text{ then}$$

$\boldsymbol{r}_{k+1} \in G_{j+1}$ can be computed in the following way. Calculate $\boldsymbol{\gamma} \in \mathbb{R}^s$ from

$$(T^H \Delta R_k)\boldsymbol{\gamma} = T^H \boldsymbol{r}_k \text{ then compute}$$

$$\boldsymbol{v} = \boldsymbol{r}_k - \Delta R_k \boldsymbol{\gamma} \text{ giving}$$

$$\boldsymbol{r}_{k+1} = \boldsymbol{v} - \omega_{j+1} A \boldsymbol{v}.$$

Since $G_{j+1} \subset G_j$, the new residuals $\boldsymbol{r}_{k+2}, \boldsymbol{r}_{k+3}, \ldots\ldots, \boldsymbol{r}_{k+s+1} \subset G_{j+1}$ can be produced by performing the above calculations repeatedly. The next residual, however, will belong to G_{j+2}. Also to be noted is the fact that for calculation of the first residual in G_j we can use any value for ω_j but this value must be same for calculation of the remaining residuals in G_j. Further the algorithm for IDR(s) is presented.

Data: $A, \boldsymbol{x}_0, \boldsymbol{b}, T$
//Initialize $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_0$, calculate first s residuals by
$\boldsymbol{v} = A\boldsymbol{r}_k$; $\omega = \frac{\boldsymbol{v}^H \boldsymbol{r}_k}{\boldsymbol{v}^H \boldsymbol{v}}$ and $\Delta\boldsymbol{x}_k = \omega\boldsymbol{r}_k$; $\Delta\boldsymbol{r}_k = -\omega\boldsymbol{v}$
$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k + \Delta\boldsymbol{r}_k$; $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta\boldsymbol{x}_k$ where $k \in [0, s-1]$ and form
$\Delta R_{k+1} = (\Delta\boldsymbol{r}_k, \Delta\boldsymbol{r}_{k-1}, \ldots\ldots, \Delta\boldsymbol{r}_0)$ and $\Delta X_{k+1} = (\Delta\boldsymbol{x}_k, \Delta\boldsymbol{x}_{k-1}, \ldots\ldots, \Delta\boldsymbol{x}_0)$
//Building G_j spaces for $j = 1, 2, 3, ...$
$n = s$
//loop over G_j spaces
while $((\|\boldsymbol{r}_k\| > Tol)$ & $(k < Max\_iter))$ **do**
  //loop inside G_j space-time
  for $k = 0, s$ **do**
    Solve for $\boldsymbol{\gamma}$ by $T^H \Delta R_k \boldsymbol{\gamma} = T^H \boldsymbol{r}_k$
    $\boldsymbol{v} = \boldsymbol{r}_k - \Delta R_k \boldsymbol{\gamma}$
    if $k = 0$ **then**
      //First vector in G_{j+1}
      $\boldsymbol{t} = A\boldsymbol{v}$
      $\omega = \frac{\boldsymbol{t}^H \boldsymbol{v}_k}{\boldsymbol{t}^H \boldsymbol{t}}$
      $\Delta\boldsymbol{r}_k = -\Delta R_k \boldsymbol{\gamma} - \omega\boldsymbol{t}$
      $\Delta\boldsymbol{x}_k = -\Delta X_k \boldsymbol{\gamma} - \omega\boldsymbol{v}$
    else
      //Subsequent vectors in G_{j+1}
      $\Delta\boldsymbol{x}_k = -\Delta X_k \gamma - \omega\boldsymbol{v}$
      $\Delta\boldsymbol{r}_k = -A\Delta\boldsymbol{x}_k$
    end
    $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k + \Delta\boldsymbol{r}_k$
    $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta\boldsymbol{x}_k$
    $k = k + 1$
    $\Delta R_k = (\Delta\boldsymbol{r}_{k-1}, \Delta\boldsymbol{r}_{k-2}, \ldots\ldots, \Delta\boldsymbol{r}_{k-s})$
    $\Delta X_k = (\Delta\boldsymbol{x}_{k-1}, \Delta\boldsymbol{x}_{k-2}, \ldots\ldots, \Delta\boldsymbol{x}_{k-s})$
  end
end

Algorithm 1: IDR(s) Solver (to solve $A\boldsymbol{x} = \boldsymbol{b}$)

As proven by Theorem 4.1, the IDR(s) method converges in at-most N outer steps, with each outer step having $s + 1$ inner steps. Hence we have at-most $N \times (s + 1)$ matrix vector multiplications. It can be proven, however, that the rate at which the dimension of $G_j$reduces is "almost always" equal to s[12], hence the total number of outer iterations reduces to $\frac{N}{s}$.

3.4 GCR METHOD

The GCR method [8] is yet another method to solve $A\boldsymbol{x} = \boldsymbol{b}$, where A is a general matrix. For a constant preconditioner, the GCR method and the GMRES method are mathematically the same. The GCR method, although may not be the fastest algorithm, is very simple to implement, minimizes the residual norm and has a very important property that it does not require a constant preconditioner. The usefulness of the last property will be shown in Appendix A where we discuss the preconditioners for the saddle point problem, here we just discuss the basic formulation.
Let $\{\boldsymbol{v}_1\boldsymbol{v}_2 \dots \boldsymbol{v}_k\}$ be the orthonormal basis of $K_k(A, \boldsymbol{r}_0)$, we construct $\boldsymbol{r}_k$ orthonormal to $K_k(A, \boldsymbol{r}_0)$. Then we have

$$\boldsymbol{r}_k = \boldsymbol{r}_0 - \sum_1^k \alpha_j \boldsymbol{v}_j \text{ where } \alpha_j = (\boldsymbol{r}_0, \boldsymbol{v}_j) = \left(\boldsymbol{r}_0 - \sum_{m=1}^{j-1} (\boldsymbol{r}_0, \boldsymbol{v}_m)\boldsymbol{v}_m, \boldsymbol{v}_j \right) = (\boldsymbol{r}_{j-1}, \boldsymbol{v}_j)$$

implying

$$\boldsymbol{r}_k = \boldsymbol{r}_{k-1} - (\boldsymbol{r}_{k-1}, \boldsymbol{v}_k)\boldsymbol{v}_k.$$

Which gives

$$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + (\boldsymbol{r}_{k-1}, \boldsymbol{v}_k)\boldsymbol{s}_k, \qquad \boldsymbol{v}_k = A\boldsymbol{s}_k$$

Now all that remains is to find $\{\boldsymbol{v}_1, \boldsymbol{v}_2 \dots \boldsymbol{v}_k\}$ and $\{\boldsymbol{s}_1, \boldsymbol{s}_2 \dots \boldsymbol{s}_k\}$. This can be done by using the Gram-Schmidt type orthogonalization processes. The GCR method with preconditioning is presented later.
Evidently the computational overhead of the GCR method is more than that of the GMRES method (nearly twice), since both $\{\boldsymbol{v}_1, \boldsymbol{v}_2 \dots \boldsymbol{v}_k\}$ and $\{\boldsymbol{s}_1, \boldsymbol{s}_2 \dots \boldsymbol{s}_k\}$ must be stored and computed. But an advantage of GCR is that we can truncate it easily, unlike the GMRES method, because of which the GCR method may converge faster than GMRES.

Chapter 4

PRECONDITIONERS

The previous discussion of convergence for the GMRES and CG methods motivates the use of preconditioning, i.e., improve the spectral properties of A to solve $A\boldsymbol{x} = \boldsymbol{b}$ efficiently. The main idea is to premultiply matrix A with another easily invertible matrix P^{-1}, which is close to A, such that the iterative solver for the system having $P^{-1}A$ as the system matrix converges faster than the iterative solver for the system having system matrix A. That is, instead of solving $A\boldsymbol{x} = \boldsymbol{b}$ we solve the system

$$P^{-1}A\boldsymbol{x} = P^{-1}\boldsymbol{b}.$$

Another way of forming a preconditioned system is by right preconditioning. That is, we solve $AP^{-1}P\boldsymbol{x} = \boldsymbol{b}$. In this system we first solve for $\boldsymbol{y}$ in $AP^{-1}\boldsymbol{y} = \boldsymbol{b}$ and then extract solution by solving $P\boldsymbol{x} = \boldsymbol{y}$. Which way to model a preconditioned system is a topic open for debate with no conclusive answers yet. However, for this study both left and right preconditioning is used.
As described above, in choosing a preconditioner we are faced with two requirements [9]
1. The preconditioner must be easily invertible, i.e., $P\boldsymbol{x} = \boldsymbol{y}$ is easily solvable.
2. It must improve the convergence of the iterative method.
If we look at the first condition the Identity matrix is a perfect choice, but it does not help us at all with the second requirement. While, if we look at the second condition A^{-1} is the perfect choice, but it does not help us at all with the first condition. Therefore, choosing a preconditioner is an optimization problem between the above two requirements. Further, we discuss some of the preconditioners used in the available code, or proposed to be used in this thesis.

4.1 JACOBI PRECONDITIONER

Jacobi is an easy to implement preconditioner, which and can improve the convergence in the cases having jumps in the diffusion coefficient [9] (which is also the case in this endeavor due to the presence of the interface). Jacobi preconditioning is also easy to parallelize. It is basically scaling the equation with the diagonal of system matrix, presenting us with very little extra calculations. Here,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & \cdots & & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \;\&\; P = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & & a_{nn} \end{bmatrix}$$

with P^{-1} simply

$$P^{-1} = \begin{bmatrix} 1/a_{11} & 0 & \cdots & 0 \\ 0 & 1/a_{22} & 0 & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & & 1/a_{nn} \end{bmatrix}$$

4.2 INCOMPLETE CHOLESKY PRECONDITIONER

For methods like CG, which are applicable for a Symmetric Positive Definite (SPD), we would like to have a preconditioner such that the resulting system is also SPD. Let LL^T be the Cholesky decomposition of system matrix A, then $P = LL^T$ preserves the the SPD properties of the system if used as a dual-sided preconditioner. The problem with this preconditioner is that it is not sparse, hence the resulting system may also not be sparse requiring more memory space and incurring more construction cost. To preserve the sparsity we can use an Incomplete Cholesky factorization [13] which is in some sense close to the Cholesky factorization. The Incomplete Cholesky factorization is constructed by setting the non-zero elements of L which are zero in A to zero.
Below, we describe the algorithm of the factorization used in the present code. For a 2-d case, we know that we get a five diagonal matrix from the Poisson equation (for which CG method is used) which requires storage of only 3 diagonal (due to symmetry). We form the preconditioner as $P = LD^{-1}L^T$ where, lower triangular matrix L and diagonal matrix D satisfies
1. $L_{ij} = 0$ if $A_{ij} = 0,\ i > j$
2. $L_{ii} = D_{ii}$
3. $(LD^{-1}L^T)_{ij} = A_{ij}\ \forall\ (i,j)$ where $A_{ij} \neq 0,\ i \geq j$

Therefore, if A and L are

$$A = \begin{bmatrix} a_1 & b_1 & 0 & & c_1 & & & \\ b_1 & a_2 & b_2 & \ddots & & c_2 & & 0 \\ \vdots & \ddots & \ddots & \ddots & & & \ddots & \\ c_1 & & b_m & a_{m+1} & b_{m+1} & 0 & & c_{m+1} \\ 0 & \ddots & 0 & \ddots & \ddots & \ddots & 0 & \ddots \end{bmatrix}$$

$$L = \begin{bmatrix} d_1^1 & & & & & \\ b_1^1 & d_2^1 & & & & \\ \vdots & \ddots & \ddots & & 0 & \\ c_1^1 & & b_m^1 & d_{m+1}^1 & & \\ 0 & \ddots & 0 & \ddots & \ddots & \end{bmatrix},$$

then

$$\left.\begin{array}{rl} d_i^1 = & a_i - \frac{b_{i-1}^2}{d_{i-1}} - \frac{c_{i-m}^2}{d_{i-m}} \\ b_i^1 = & b_i \\ c_i^1 = & c_i \end{array}\right\} i = 1, ..., n.$$

Very easily this method can be extended for a 3d case in which case we will get a seven diagonal matrix from the Poisson equation. More details about this method can be found in [13].

4.3 DEFLATION PRECONDITONER

As discussed above, the aim of preconditioning is to improve the spectral properties of matrix A so that the Krylov subspace method solves $A\boldsymbol{x} = \boldsymbol{b}$ efficiently. Presence of the small eigenvalues is one such spectral property which deteriorates the performance of the solver considerably. Therefore, it makes sense to have a preconditioner which deflates the small eigenvalues of A. This is the main motivation behind the deflation preconditioners [14, 15, 16, 17].

As will be discussed below, the deflation preconditioner requires construction of the deflation matrix which spans the approximate null space of matrix A, which points to the small eigenvalues that need to be deflated. Naturally, the construction of an

efficient deflation matrix is quite important as a bad choice of deflation matrix may not deflate the required eigenvalues. One obvious way to construct the deflation matrix is to run the Arnoldi iterations to find the smallest eigenvalues, this however is a very expensive task and would definitely mitigate the speedup achieved by deflation to a great extent, if not completely.
Another way is to find the eigenvalues based on the physics of the problem, for example, if an interface is present in the domain, the interface location may give information about the small eigenvalues and this information could be used to construct the deflation matrix. The third way, which pertains more to this research, is to construct the deflation matrix using the algebraic deflation vectors, as in the case of domain decomposition. We shall discuss this more in detail later, for now it suffices to say that since in this research the deflation preconditioner is only used to improve the performance of domain decomposition, from now on, we will not consider the first two methods of constructing the deflation matrix. Further, we discuss the deflation algorithm.
Suppose we have to solve $A\boldsymbol{x} = \boldsymbol{b}$ where $A \in \mathbb{R}^{n\times n}$. We consider a matrix $Z \in \mathbb{R}^{n\times m}$, where $m < n$ and rank of Z is m, i.e., columns of Z are all linearly independent. The columns of Z are called the deflation vectors and Z is called the deflation matrix. The columns of Z are spanned by the deflation subspace in which the *bad* (small) eigenvalues of A reside (which are to be projected out of the residual). For this we define two projectors

$$\Pi = I - AZE^{-1}Z^T \text{ and } Q = I - ZE^{-1}Z^TA, \text{ with}$$
$$\Pi^2 = \Pi,\ Q^2 = Q \text{ and } E = Z^TAZ,$$

where $E \in \mathbb{R}^{m\times m}$ and I is identity matrix of appropriate size. To solve for $\boldsymbol{x}$ we write

$$\boldsymbol{x} = (I - Q)\boldsymbol{x} + Q\boldsymbol{x} = ZE^{-1}Z^T\boldsymbol{b} + Q\boldsymbol{x}. \tag{4.1}$$

Here $ZE^{-1}Z^T\boldsymbol{b}$ is easily computed, hence we turn our attention towards the remaining expression $Q\boldsymbol{x}$. Since $\Pi A = AQ$ we compute $Q\boldsymbol{x} = \boldsymbol{x}_a$ by

$$\Pi A\boldsymbol{x}_a = \Pi AQ\boldsymbol{x} = \Pi^2 A\boldsymbol{x} = \Pi A\boldsymbol{x} = \Pi\boldsymbol{b}, \text{ i.e., we solve}$$
$$\Pi A\boldsymbol{x}_a = \Pi\boldsymbol{b}.$$

Then if we multiply $\boldsymbol{x}_a$ by Q we get $Q\boldsymbol{x}_a = Q^2\boldsymbol{x} = Q\boldsymbol{x}$ hence we replace $Q\boldsymbol{x} = Q\boldsymbol{x}_a$ in equation (4.1) to solve for $\boldsymbol{x}$. For further discussion on deflation [14, 15, 16, 17] are recommended. In the above method, E is presumed to be easily invertible. It can be shown that the deflation method works correctly if E^{-1} is computed with high accuracy. Also, we do not explicitly compute E^{-1} rather solve the system $E\boldsymbol{q} = \boldsymbol{t}$ with high accuracy using some direct method.
Now, we discuss the way to formulate the deflation matrix for the domain decomposition technique. The need for deflation for domain decomposition is discussed later

(in the section where we discuss domain decomposition), here we only present how to formulate the deflation vectors. We base our deflation vectors on the decomposition of domain Ω, i.e., if we decompose the domain into d non-overlapping sub-domains we define the deflation vector $\boldsymbol{z}_i$ as

$$(\boldsymbol{z}_i)_j = \left.\begin{array}{ll} 0 & if \quad \boldsymbol{x}_j \notin \Omega_i \\ 1 & if \quad \boldsymbol{x}_j \in \Omega_i \end{array}\right\} \; 1 \leq i \leq d.$$

Where, Ω_i is the i^{th} sub-domain, and $\boldsymbol{x}_j$ is the position where the j^{th} unknown is evaluated. That is, the total number of deflation vectors is the number of non-overlapping sub-domains, and each vector has a non homogeneous entry only when that index belongs to the domain to which the corresponding deflation vector belongs. The exact deflation vector in our case will be presented in the domain decomposition sub-section. Further, we present the algorithms of a few Krylov subspace methods with deflation technique.

//Select $\boldsymbol{x}_0$. Compute $\boldsymbol{r}_0 = (\boldsymbol{b} - A\boldsymbol{x}_0)$, set $\boldsymbol{r}_0^1 = \Pi \boldsymbol{r}_0$
//Solve $P\boldsymbol{x}_0 = \boldsymbol{r}_0^1$ and set $\boldsymbol{p}_0 = \boldsymbol{y}_0$
for $j = 0, 1, \ldots$ *until convergence* **do**
  $\boldsymbol{w}_j^1 = \Pi A \boldsymbol{p}_j$
  $\alpha = \frac{(\boldsymbol{r}_j^1, \boldsymbol{y}_j)}{(\boldsymbol{w}_j^1, \boldsymbol{p}_j)}$
  $\boldsymbol{x}_{j+1}^1 = \boldsymbol{x}_j^1 + \alpha_j \boldsymbol{p}_j$
  $\boldsymbol{r}_{j+1}^1 = \boldsymbol{r}_j^1 - \alpha_j \boldsymbol{w}_j^1$
  Solve $P\boldsymbol{y}_{j+1} = \boldsymbol{r}_{j+1}^1$
  $\beta = \frac{(\boldsymbol{r}_{j+1}^1, \boldsymbol{y}_{j+1})}{(\boldsymbol{r}_j^1, \boldsymbol{y}_j)}$
  $\boldsymbol{p}_{j+1} = \boldsymbol{y}_{j+1} + \beta_j \boldsymbol{p}_j$
end
$\boldsymbol{x} = ZE^{-1}Z^T\boldsymbol{b} + Q\boldsymbol{x}_{j+1}^1$

Algorithm 2: Deflated Preconditioned CG for solving $A\boldsymbol{x} = \boldsymbol{b}$ with preconditioner P and deflation matrix Z.

Similarly the deflation technique can be used for non-symmetric systems as well. In this case instead of solving $A\boldsymbol{x} = \boldsymbol{b}$ (A non-symmetric), we use the above described method and solve $\Pi A\boldsymbol{x} = \Pi\boldsymbol{b}$ using Krylov subspace methods like GMRES/IDR(s). Then we multiply $\boldsymbol{x}_a$ by Q and put it in the equation (4.1) to solve for $\boldsymbol{x}$. The first part ($ZE^{-1}Z^T\boldsymbol{b}$) is computed like before.
If the non-symmetric system is left preconditioned with preconditioner P, then we instead solve $P^{-1}\Pi A\boldsymbol{x} = P^{-1}\Pi\boldsymbol{b}$ and form $QP^{-1}\boldsymbol{x}_a$ to solve for $\boldsymbol{x}$. While if the system

is right preconditioned, we solve $\Pi A P^{-1} \boldsymbol{x} = \Pi \boldsymbol{b}$ and form $Q P^{-1} \boldsymbol{x}_a$ to solve for $\boldsymbol{x}$.

Chapter 5

DESCRIPTION OF THE AVAILABLE CODE

The FDM scheme is used in this approach with variables stored in the Arakawa C grid as discussed in the previous section. Second order accurate schemes are used for the space discretization, and a second order implicit time integration method with Newton's linearization is used in the predictor step. The code is written in Fortran 90, and uses various LAPACK subroutines.

5.1 OVERALL ALGORITHM

In the original available code, as pointed out in the previous section, the calculation of flow and interface variables are fully decoupled. The flow at new time step $n+1$ is calculated based on the interface position at the previous time step $n+\frac{1}{2}$, and then the interface is advected based on the just calculated flow variables. The values of density and viscosity only depend on the interface and hence are fixed for a given interface position. The overall flow of algorithm is shown in Figure 5.1.

5.2 FLOW SOLVER

The flow solver part is broadly divided into three steps as mentioned in section 2, viz. the predictor step for a first approximation of the velocity based on the pressure value at the previous time step, the Poisson step to update the pressure based on the velocity calculated in the predictor step, and the corrector step to obtain a divergence-free velocity. Restarted GMRES is used to solve the implicit predictor equation, while

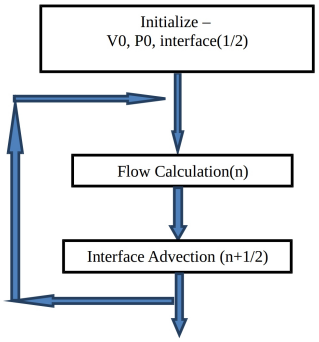


Figure 5.1: Overall flow chart of the algorithm.

for the Poisson equation the incomplete Cholesky preconditioned Conjugate Gradient method (ICCG) is used. A brief flowchart of the flow algorithm is given in Figure 5.2.

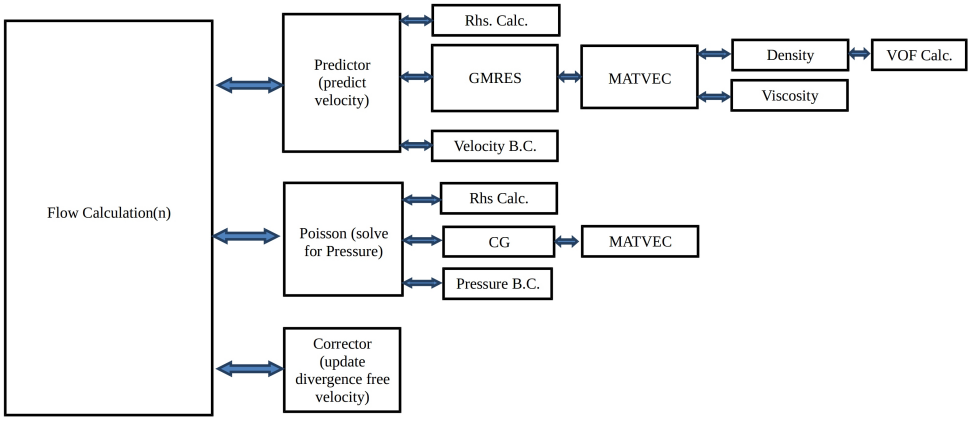


Figure 5.2: Brief flowchart of the flow algorithm.

5.3 INTERFACE SOLVER

The interface is advected as described in Chapter 2. For a stable advection of the interface the Courant number (defined as $u\frac{\Delta t}{\Delta x}$, where u is the velocity magnitude, t is time and x is the space variable) should be less than a half. It may so happen that the Courant number in flow calculation gets higher than one half, in which case the time step for the interface advection is reduced (so as to have a Courant number which

is less than a half) and multiple time step integrations are performed to increase the overall time step by the same amount as that in the flow calculation. How many steps need to be performed is calculated in the Subcycling module.
Figure 5.3 gives a brief flowchart of the Interface advection algorithm.

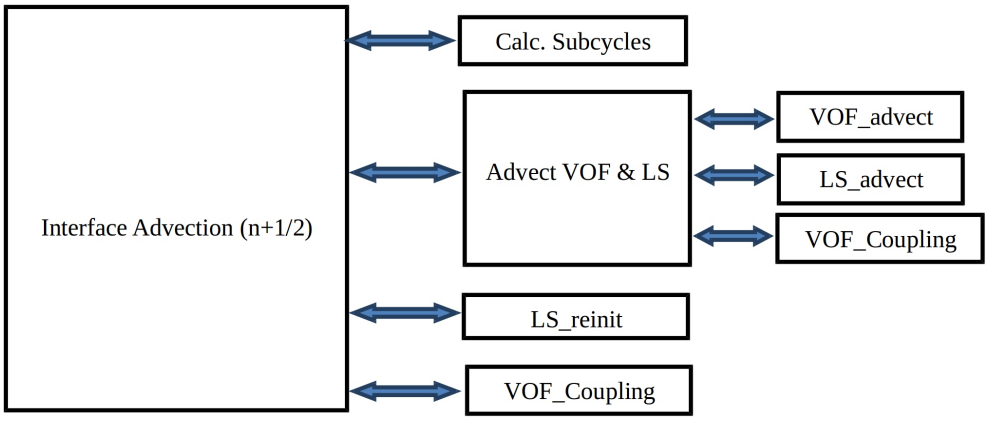


Figure 5.3: Brief flowchart of the interface advection algorithm.

Chapter 6

PARALLELIZATION OF THE MODIFIED CODE

The first step of parallelization involves making a choice regarding the architecture for which the parallelization of the code will be done. One has to choose between shared memory systems and distributed memory systems. The shared memory architecture has scalability limitations on both the memory and the processors. Additionally, parallelization on such an architectures, done using the Openmp library, is heavily plagued by synchronization problems. Due to these limitations of the shared memory architecture, in this thesis, the code is parallelized for distributed memory systems as described below.

6.1 DISTRIBUTED SYSTEMS

The distributed systems are those in which all the processors have their own memories and do not have a direct access to each others memory. A distributed system has all the nodes/processors connected via an interconnect and theoretically has no requirement of close proximity of all the participating nodes. In practice it may happen that the clusters and supercomputers have participating nodes which are several hundred miles away. Figure 6.1 shows a conceptual design of such a system.
Parallelization for the distributed memory systems has the following attributes,

- Access time depends on distance to data, i.e., distance between processors.
- Every processor has its own local address space.
- Non local data can be accessed through a request to the owning processor.

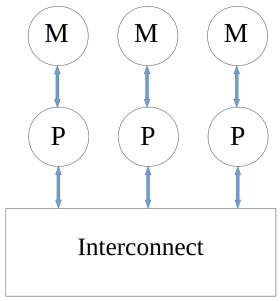


Figure 6.1: Conceptual design of a distributed memory system.

- Communication is through message passing.
- Data distribution is important and expensive.
- OpenMPI can be used to parallelize a software on the distributed memory architectures.

In distributed memory systems, synchronization problems are much less of an issue since the communication is through explicit messages and no processor has direct access to other processor's memory. Another very important advantage of distributed systems is that they are theoretically infinitely scalable, hence we can run our parallel application over a much large numbers of processors as compared to shared memory systems. Although the parallelization using OpenMPI is harder than Openmp (for shared memory systems), due to high scalability of distributed architectures we shall use OpenMPI to parallelize the available code.

6.2 DOMAIN DECOMPOSITION

After we have decided to choose the distributed architecture, we must now focus our attention towards the parallelization methodology. Since fine grain parallelization is not very suitable for the distributed architectures, we choose domain decomposition as a valid and efficient parallelization technique.
Now all that remains is to choose what type of domain decomposition we shall choose, i.e., shall we do column/row wise domain decomposition or 2d/3d block domain decomposition. Since in this case we essentially deal with pipe flows in which number of cells in axial direction is much large compared to the number of cells in any other direction, we decompose the domain axially. To implement domain decomposition axially in the available code, one should be careful about the following points:

1. We do not solve the axial (z)-momentum equation for the last boundary cell in the z direction (since a boundary conditions is specified there), but for the subdomains obtained after domain decomposition, we may have to solve the z-momentum equation for the local last boundary cell in the z direction (depending on the sub-domain).
2. Depending on the rank of the sub-domain, only the first, last, or both z-cells data need to be communicated for all the variables except for the level set function. For the level set function, the first/last 2 cells data must be communicated.
3. Another important point noted during the implementation of parallelization was the rounding errors. Since many arithmetic operations are non associative in machine precision, the rounding errors made by the parallel code will be different than the serial code. These rounding errors are inconsequential in most cases except if the problem is unstable. Then all codes behave differently since the rounding errors propagate differently in different codes because of the instabilities and non-linearity of the problem. This point will be demonstrated in the following Chapter.

Moreover, computing the solution in parallel on a decomposed domain reduces the computational time per iteration of a Krylov iterative solver on one hand. On the other hand, however, it may increase the total number of iterations to be performed [14, 15, 16, 17], thus reducing the total efficiency of the parallelized system. As reasoned in [17], though it is simple to implement a non-overlapping preconditioner in parallel, the convergence behavior of the preconditioner may deteriorate considerably when the domain is split into a high number of sub-domains. The loss of convergence is attributed to the small eigenvalues arising from domain decomposition. This motivates the use of deflation for improving the convergence of the incomplete Cholesky preconditioned CG method (used to solve the Poisson equation). The deflation technique can also be used as a coarse grid correction preconditioner [14], for this reason we shall use deflation in combination with diagonal scaling as a preconditioner for the predictor equation.
As previously discussed, for applying the deflation technique we must construct the deflation matrix whose construction is also discussed above. Here we present the deflation matrix for our case. Let us subdivide our domain into d sub-domains and let us have total n number of unknowns in the total domain and $m_{ri}, m_{\theta i}, m_{zi}, m_{pi}$ be the number of velocity and pressure unknowns in the i^{th} sub-domain $1 \leq i \leq d$. Further let $\{ri, \theta i, zi, pi\}$ be the index set (of velocity and pressure unknowns in the i^{th} sub-domain) giving the location of corresponding unknowns in $n \times 1$ vector, then deflation vectors $\boldsymbol{z}_i$ can be given by two ways:

$$z_{(r/\theta/z/p)i}(k) = \begin{cases} 1 & if\ k \in \{ri/\theta i/zi/pi\} \\ 0 & otherwise \end{cases} \textbf{ or } z_i(k) = \begin{cases} 1 & if\ k \in \{ri, \theta i, zi, pi\} \\ 0 & otherwise \end{cases}$$

Example: the deflation vectors for a 2-d channel divided into 4 sub-domains (as shown in Figure 6.2) can be constructed in two ways as indicated below:

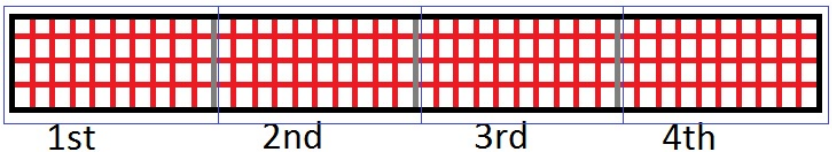


Figure 6.2: Domain decomposition for a 2-d channel.

first way

$$\boldsymbol{z}_{x1} = \begin{bmatrix} \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_{y1} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_{p1} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_{x2} = \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_{y2} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_{p2} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_{x3} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_{y3} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_{p3} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \end{bmatrix}$$

$$\boldsymbol{z}_{x4} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_{y4} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_{p4} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \end{bmatrix}, \boldsymbol{z}_{i} = \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \boldsymbol{u}_3 \\ \boldsymbol{u}_4 \\ \\ \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \boldsymbol{v}_3 \\ \boldsymbol{v}_4 \\ \\ \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \\ \boldsymbol{p}_4 \end{bmatrix} \text{ and}$$

$$Z = \begin{bmatrix} \boldsymbol{z}_{x1}, & \boldsymbol{z}_{y1}, & \boldsymbol{z}_{p1}, & \boldsymbol{z}_{x2}, & \boldsymbol{z}_{y2}, & \boldsymbol{z}_{p2}, & \boldsymbol{z}_{x3}, & \boldsymbol{z}_{y3}, & \boldsymbol{z}_{p3}, & \boldsymbol{z}_{x4}, & \boldsymbol{z}_{y4}, & \boldsymbol{z}_{p4} \end{bmatrix}.$$

second way

$$\boldsymbol{z}_1 = \begin{bmatrix} \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_2 = \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_3 = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{z}_4 = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \end{bmatrix}, \boldsymbol{z}_i = \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \boldsymbol{u}_3 \\ \boldsymbol{u}_4 \\ \\ \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \boldsymbol{v}_3 \\ \boldsymbol{v}_4 \\ \\ \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \\ \boldsymbol{p}_4 \end{bmatrix} \text{ and } Z = [\boldsymbol{z}_1,\ \boldsymbol{z}_2,\ \boldsymbol{z}_3,\ \boldsymbol{z}_4]$$

Where bold numerals denote the vectors of the size m_{xi}, m_{yi}, m_{pi} in each sub-domain i. It is not clear which of the above two ways of constructing the deflation vectors give a better performance. In this research, however, we implemented the first way as it *seems* to be more suitable for mechanical problems having discontinuous coefficients [18].

Chapter 7

RESULTS & DISCUSSION

In the subsequent sections, we first discuss the profiling results of the available code. Then, we show the speedup results (on a single processor) of the modified code obtained by changing the flow structure of the available code and by diagonal scaling of the GMRES method. Next, we shall demonstrate the accuracy of implementation of the different solvers, preconditioners, and parallelization by comparing the results with that of the available code for two physical test cases. This is followed by a discussion on the speedup obtained due to the implementation of parallelization, deflation, and IDR(s).

7.1 PROFILING & SERIAL IMPROVEMENT

To increase the speed of a code it is important to identify the parts which take the longest time per iteration. It is not very wise to improve the efficiency of those parts of the program which take little time in comparison to others, because it does not help in improving the overall speed by much. Hence, profiling of the available code was done to identify the components which take the maximum time. The profiling results were obtained for two different test cases which are explained in detail in the next section where we examine the physical results and the accuracy given by the different codes. Here, we only give the physical parameters for which the codes were run (Table 7.1). The profiling results of the original code for various density to viscosity ratios, which correspond to various Reynolds number, for the two test cases are presented in Table 7.2. Percentage denotes the percentage of total time taken by a particular module. As evident from the above profiling results, at least for lower Reynolds number flows, which are typical of the applications for which this code is developed, the predictor part takes the longest time for both the test cases. The Poisson solver, contrary to our expectations takes much less time.

Table 7.1: Physical properties used for two different test cases.

	case1 (Benjamin bubble)	case2 (Rising bubble)
Cylinder Radius [mm]	$4.2\text{x}10^{-2}$	$4.2\text{x}10^{-2}$
Cylinder Length [mm]	$1.18\text{x}10^{-1}$	$1.18\text{x}10^{-1}$
Bubble Initial Radius [mm]	-	$6.08\text{x}10^{-3}$
Bubble Initial Position [mm]	-	-0.354
Grid Size	20x30x54	20x30x50
time step [ms]	0.1	0.1
Surface Tension [N/m]	0	0.0322

In the available code, restarted GMRES is used to solve the predictor part. To speed it up, various modifications are performed on the original code. The first major improvement was motivated by the fact that, in the available code, the calculations of matrix entries are performed in each iteration of GMRES unnecessarily. Hence, if we could save the matrix in some diagonal form it would decrease the computational time. This modification, though quite complex and involved to perform, was rather fruitful as it gave an overall speedup of nearly 4 times in the predictor step (Table 7.3) irrespective of the Reynolds number. In Table 7.3 we present some speedup results for the two cases (for different density to viscosity ratios).
The second improvement was motivated by the knowledge that Krylov subspace solvers converge faster with preconditioning. Hence, the Jacobi preconditioner was used to speed up the predictor module. This preconditioner was chosen because it is easy to implement, delivers considerable speedups for our case [9], and also lends itself well to parallel programming which is a highly desirable property for the present endeavor. Table 7.4 compares the time taken by the non-preconditioned and preconditioned GMRES methods for the two test cases. As can be seen from Table 7.4, diagonal scaling gives a speedup of nearly 1.5-2 times for the low Reynolds number cases (which are typical of the target applications).
The above modifications improve the performance of the serial version substantially. For the low Reynolds number cases a speedup of approximately 8-9 times is obtained in the predictor step. For example in the rising bubble problem, the last case initially took 4.1s, but after the two improvements it took only 0.56s, hence a speedup of 8 times is obtained. While for the Benjamin bubble problem, the time was reduced from 9.5s to 1.1s (for the ratio $1e2\&1e3$) which is a 9 times speedup.
Similarly, for the high Reynolds number cases a speedup of approximately 4-5 times can be expected (in the predictor step) as a result of the above improvements.

Table 7.2: Profiling results of the original code for 20 time steps for two different cases.

Density to viscosity ratio (Fluid 1 & Fluid 2)	Flow Time[s] - percentage			Interface Time[s] - percentage		
	Predictor Time[s] / percentage	Poisson Time[s] / percentage	Corrector Time[s] / percentage	Advect VOF &LS Time[s] / percentage	LS reinit Time[s] / percentage	Coupling Time[s] / percentage
case1 (Benjamin bubble)						
1e4 & 1e6	17.4 - 91.7%			1.07 - 5.9%		
	10.89 - 60.2%	5.37 - 29.7%	0.29 - 1.6%	0.25 - 1.3%	0.45 - 2.5%	0.15 - 0.8%
1e3 & 1e5	40.2 - 96.1%			1.2 - 2.9%		
	33.69 - 81.8%	5.32 - 12.9%	0.31 - 0.75%	0.5 - 1.3%	0.45 - 1.1%	0.15 - 0.38%
1e2 & 1e4	129.32 - 97.8%			1.44 - 1.1%		
	123.91 - 93.3%	5.25 - 4%	0.26 - 0.2%	0.81 - 0.62%	0.44 - 0.3%	0.14 - 0.1%
1e2 & 1e3	146.45 - 98.1%			1.42 - 0.9%		
	141.2 - 94.2%	5.07 - 3.4%	0.31 - 0.21%	0.70 - 0.47%	0.45 - 0.3%	0.15 - 0.1%
case2 (Rising bubble)						
1e4 & 1e6	7.37 - 91.0%			0.6 - 8.0%		
	3.14 - 43.8%	3.18 - 44.4%	0.17 - 2.4%	0.23 - 3.13%	0.25 - 3.4%	0.08 - 1.1%
1e3 & 1e5	13.36 - 94.9%			0.53 - 4.3%		
	9.12 - 69.2%	3.21 - 24.4%	0.13 - 1.0%	0.17 - 1.3%	0.23 - 1.8%	0.07 - 0.5%
1e2 & 1e4	47.28 - 98.9%			0.45 - 1.0%		
	42.89 - 90.7%	3.33 - 7.1%	0.15 - 0.3%	0.2 - 0.4%	0.12 - 0.3%	0.09 - 0.2%
1e2 & 1e3	60.12 - 98.5%			0.48 - 1.0%		
	56.47 - 92.7%	3.18 - 5.2%	0.16 - 0.3%	0.15 - 0.3%	0.24 - 0.4%	0.05 - 0.1%

7.2 ACCURACY CHECK

To check the accuracy of the modified code, simulation results for the two test cases mentioned above are discussed. The following subsections describe the test cases in

Table 7.3: Number of iterations and time taken by GMRES method to solve a single predictor step.

	Benjamin bubble				**Rising bubble**			
Density to viscosity ratio	GMRES				GMRES			
	Original code		Storing Matrix		Original code		Storing Matrix	
	# Iter.	Time [s]	# Iter.	Time [s]	# Iter.	Time [s]	# Iter.	Time [s]
1e4 & 1e6	11	0.75	11	0.19	5	0.22	5	0.08
1e3 & 1e5	34	2.5	34	0.5	17	0.68	17	0.15
1e2 & 1e4	120	8.75	120	1.78	78	3.1	78	0.66
1e2 & 1e3	133	9.5	133	1.91	100	4.1	100	0.9

Table 7.4: Number of iterations and time taken by the improved GMRES method to solve a single predictor step with and without preconditioning.

	Benjamin bubble				**Rising bubble**			
Density to viscosity ratio	GMRES				GMRES			
	No precond.		precond.		No precond.		precond.	
	# Iter.	Time [s]	# Iter.	Time [s]	# Iter.	Time [s]	# Iter.	Time [s]
1e4 & 1e6	11	0.19	10	0.19	5	0.08	5	0.08
1e3 & 1e5	34	0.5	24	0.37	17	0.15	16	0.15
1e2 & 1e4	120	1.78	65	1.1	78	0.66	62	0.54
1e2 & 1e3	133	1.91	66	1.1	100	0.9	64	0.56

detail and compare the results obtained by the different codes. Before moving on, we refer to Table 7.5 to know some important terms used in further discussions.

7.2.1 Dam Break / Benjamin Bubble Problem

In the dam break problem, we consider two fluids with different densities, which are initially at rest, in a closed pipe. The fluids are separated by a vertical dam as shown in Figure 7.1. At t=0 the dam breaks, and the fluids start to move so as to attain an equilibrium where the heavier fluid is below the lighter one.
We impose a slip boundary condition at the walls, with the two fluids initially at rest. The geometry and the properties of oil and water are indicated in Table 7.6, and in this test case the surface tension is neglected. Figure 7.2 shows the movement of the interface with time. Initially, the interface falls as a single entity, but as the time

Table 7.5: Terminology and specifications of different codes.

Sl. No.	**Code name**	**Poisson Solver**	**Predictor solver**
1	original (available) code	ICCG	unpreconditioned GMRES (Restarted 50 unless stated otherwise)
2	modified serial code	ICCG	diagonally scaled GMRES (Restarted 50 unless stated otherwise)
3	parallel (modified) code	ICCG	diagonally scaled GMRES (Restarted 50 unless stated otherwise)
4	deflated code	Deflated ICCG	diagonally scaled GMRES (Restarted 50 unless stated otherwise)
5	IDR code	ICCG	diagonally scaled IDR(s) (s=5 unless stated otherwise)

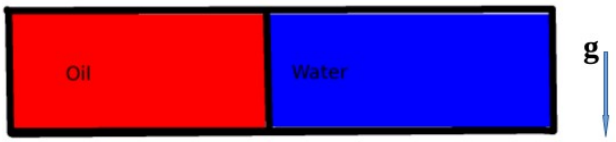


Figure 7.1: Initial condition for the Benjamin bubble problem.

progresses the interface hits the cylinder walls and due to the lack of surface tension and wall viscous effects the interface crumbles and disintegrates. As the interface crosses the center line, the inherent numerical errors in the code lead to instabilities in the solution.

Table 7.6: Benjamin bubble geometry and properties.

Cylinder Radius [mm]	$4.2\text{x}10^{-2}$
Cylinder Length [mm]	$1.18\text{x}10^{-1}$
Grid Size	50x60x84
time step [ms]	0.1
Density of Fluids [kg/m^3]	875.5 & 900.0
Viscosity of Fluids [kg/m/s]	0.118 & $1.77\text{x}10^{-3}$

To check the accuracy of the new codes, we shall compare the velocity of the center of mass of one fluid (Rise velocity) obtained by the modified serial/parallel code with that of the available code. Figure 7.3 shows three plots, i.e., difference between the Rise velocities predicted by the modified serial code and the original code, and the

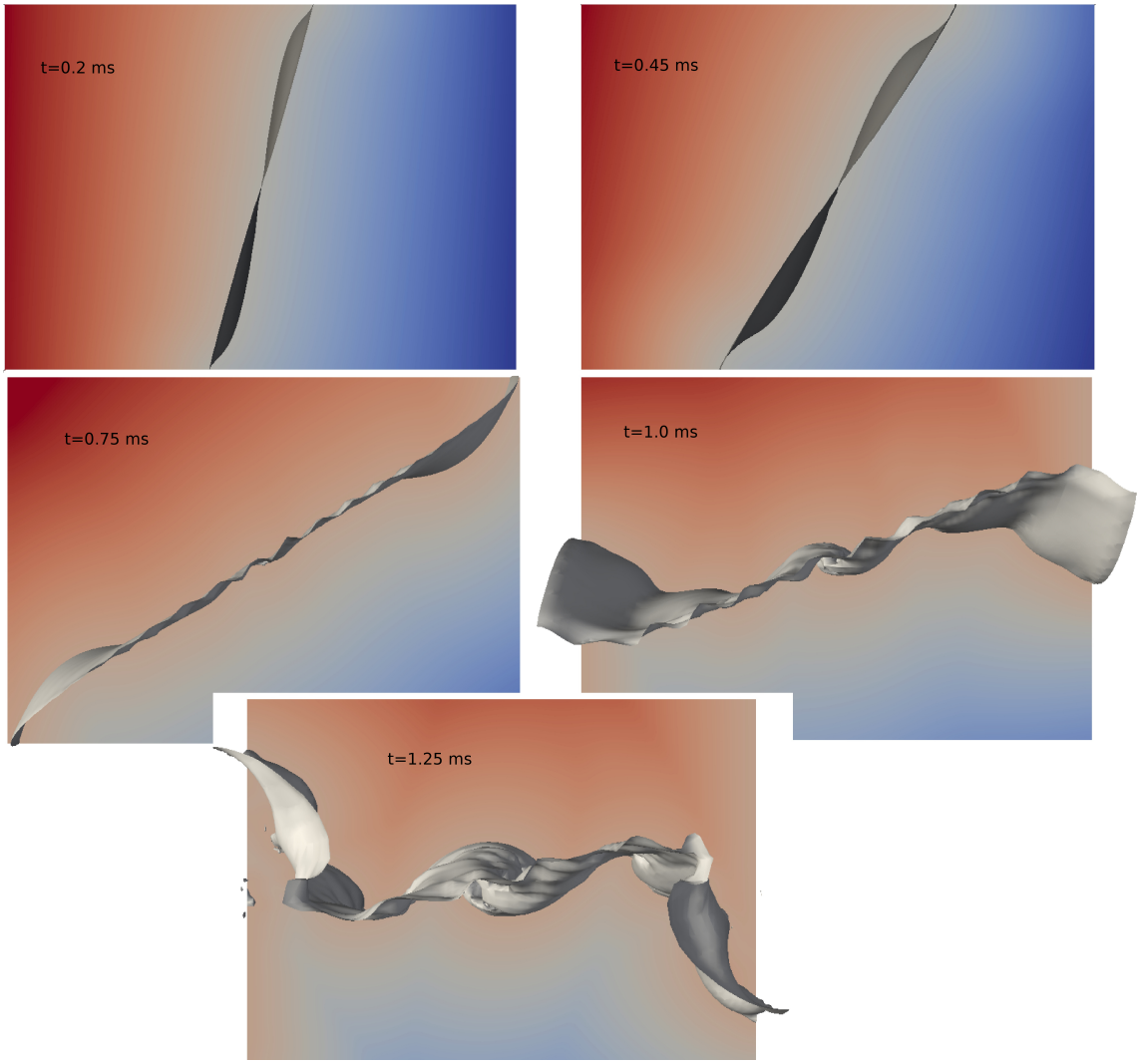


Figure 7.2: Movement of the interface.

difference between the Rise velocities obtained by the parallel modified code (with 7 and 12 processors) and the original code. As can be seen, up to around $0.75s$ we have a very good match for the Rise velocity. This means that the build-up in the machine precision differences, resulting due to the distinct sequence of floating point operations, for different codes are low. Later, due to the above explained instabilities and non-linearity of the problem, the machine precision differences lead to larger deviation in the solutions giving larger discrepancies in the Rise velocities.
Further, to check the accuracy of implementation of the deflated CG and IDR(s)

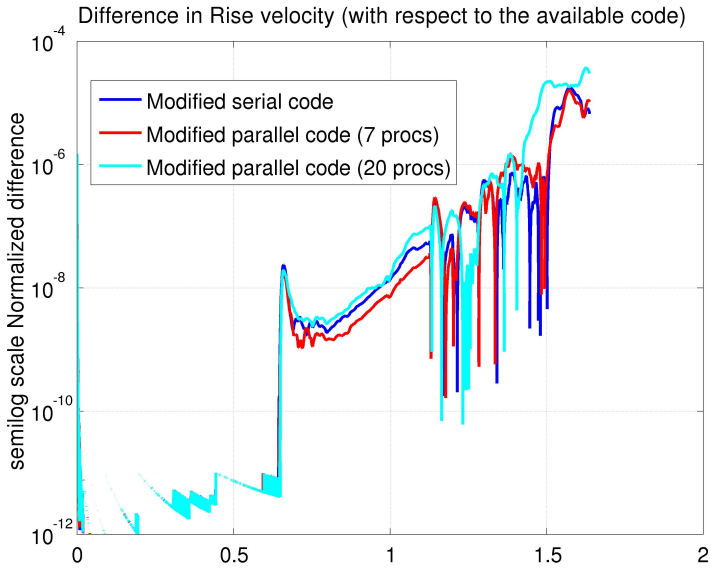


Figure 7.3: Difference in Rise velocity obtained by modified serial and parallel codes.

method, the deflated code and the IDR(s) code were run for the same test case in parallel (on 7 and 12 procs). Figure 7.4 shows the difference between the Rise velocities obtained by the two codes and the Rise velocity obtained by the modified parallel code (without deflation and IDR(s) solver) on respective number of processors. As evident, we have a good match, but again as before, due to the non-linearity of the problem and creeping instabilities the results differ at the end. Another observation can be made while comparing Figure 7.3 and Figure 7.4 i.e., there is an exact same trend in the growth of difference with time, even when in Figure 7.3 the comparison is with respect to the original available code and in Figure 7.4 the comparison is in respect to the modified parallel code. This points out to the fact that the difference in the Rise velocities is not due to some errors made during parallelization or solver implementation, but rather due to the inherent instabilities which affect all the codes.

7.2.2 The Axisymmetric Rising Bubble Problem

In the axisymmetric rising bubble problem, we initially have an axisymmetric bubble of one fluid in the other fluid. As the time progresses, the bubble rises due to the buoyancy effect. As in the previous case, the aim of this test case is to accurately predict (in comparison to the original code) the Rise velocity of the bubble.
If the bubble radius is comparable to the cylinder radius, then due to the wall effects the

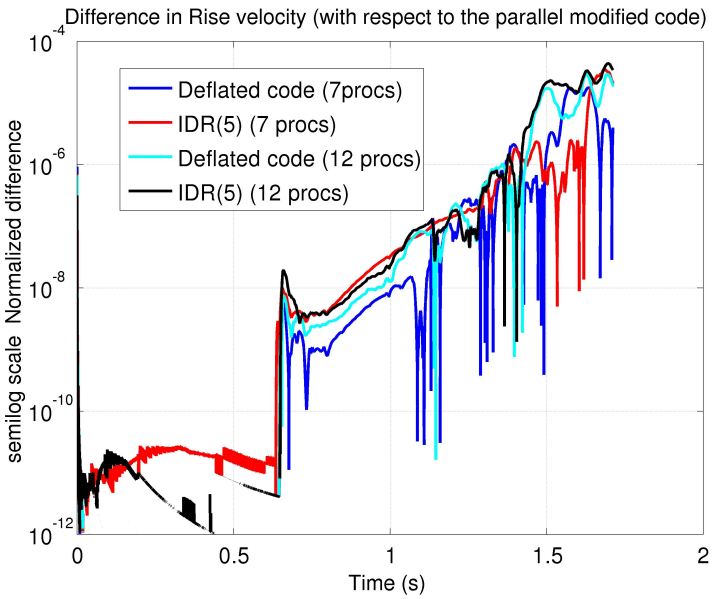


Figure 7.4: Difference in Rise velocity obtained by using deflation and IDR(s) solver.

bubble becomes unstable. Hence, a cylinder with a sufficiently large radius is chosen. Also due to the importance of surface tension on the shape and Rise velocity of the bubble, we have a non-zero surface tension for this case. Further, we impose a no-slip boundary condition on the walls. Table 7.7 gives the geometry and physical properties of the fluid.

Table 7.7: Rising bubble geometry and properties.

Cylinder Radius [mm]	$4.2\text{x}10^{-2}$
Cylinder Length [mm]	$1.18\text{x}10^{-1}$
Bubble Initial Radius	$6.08\text{x}10^{-3}$
Bubble Initial Position	-0.354
Grid Size	50x60x84
time step [μs]	10.0
Density of Fluids [kg/m^3]	875.5 & 1.225
Viscosity of Fluids [kg/m/s]	0.118 & $1.77\text{x}10^{-5}$
Surface Tension [N/m]	0.0322

Figure 7.5 shows the bubble captured at different time instances. As can be seen initially the bubble behaves fine, but after some time the bubble is not axisymmetric anymore. This is due to the numerical inaccuracies and insufficient grid size. These

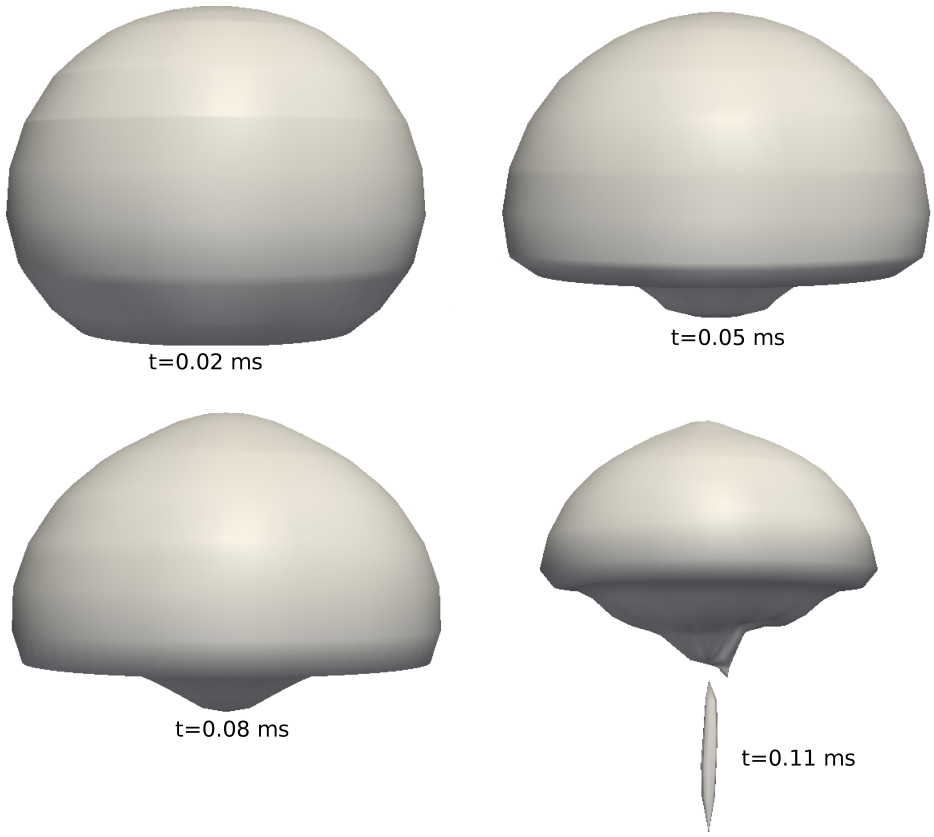
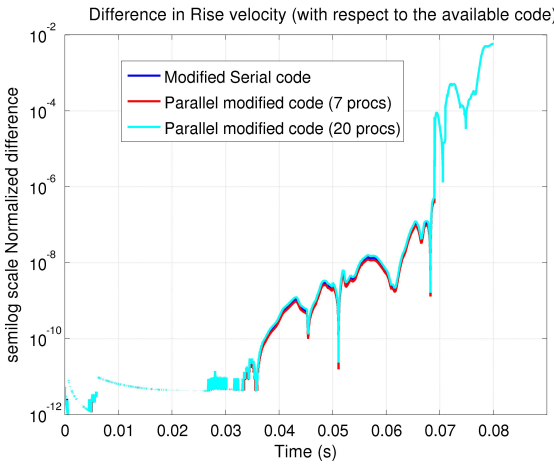


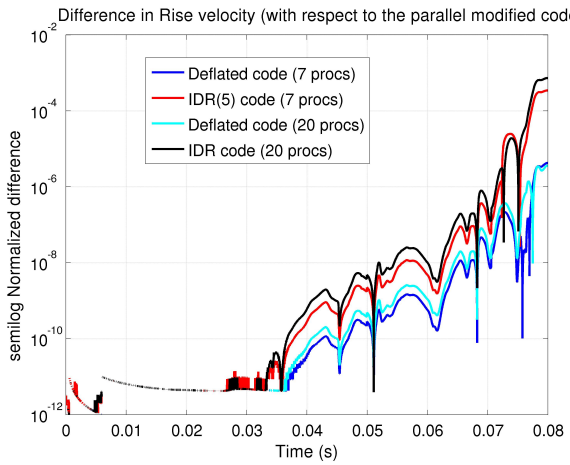
Figure 7.5: Movement of the bubble.

numerical inaccuracies give rise to instabilities, which causes different codes to behave differently. This is very well reflected in Figure 7.6a which, like in the previous test case, plots the difference between the Rise velocities predicted by the modified codes (serial and parallel on 7 and 20 processors) and the Rise velocity predicted by the original code. Initially the difference between the velocities is quite negligible, but as the time progresses and instabilities creep in, due to the non-linearity of the problem, the machine precision floating point differences in different codes result in a larger deviation in the solutions.

As in the previous case, same simulations are performed parallely (on 7 and 20 cores) with deflated CG and IDR(s) method turned on (separately) to further prove the accuracy of the implementation. Figure 7.6b shows the difference between the Rise velocities of the interfaces obtained by the two codes and the Rise velocity obtained by the parallel code (without deflation and IDR solver) on respective number of processors. As evident, at least initially, the difference between the Rise velocities is negligible for



(a) Difference in Rise velocity obtained by modified serial and parallel codes.



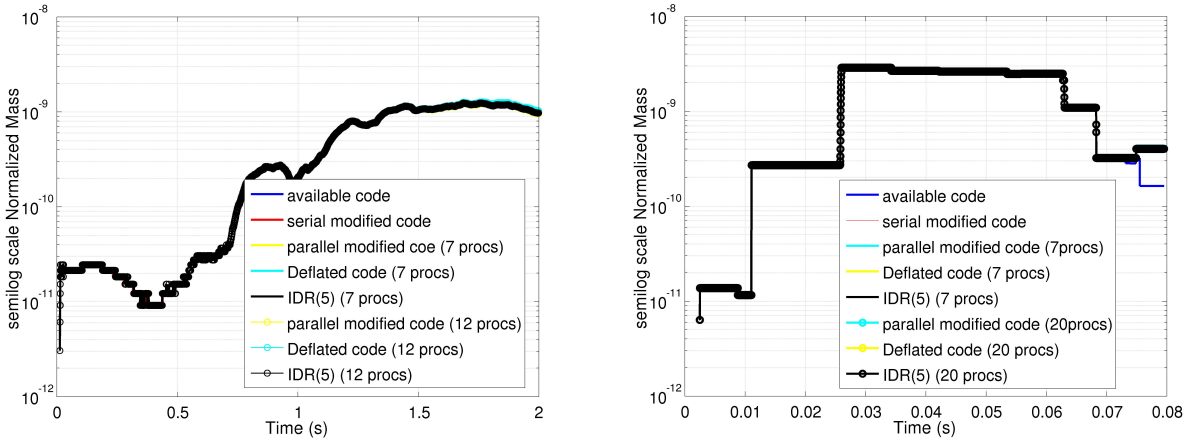
(b) Difference in Rise velocity obtained by using deflation and IDR(s) solver.

Figure 7.6: Difference in Rise velocity for the rising bubble.

all engineering purposes. At the later stage (after the onset of instabilities), however, due to the reasons explained earlier, different codes perform differently as the rounding errors propagate uniquely in each code. Further, for this test case as well, the similarity in the trends in Figure 7.6a and 7.6b shows that the differences are due to the inherent instabilities in the problem and not due to the implementation errors.
To prove the accuracy of the solution even further Figure 7.7a and Figure7.7b shows the normalized mass given by $\frac{mass(t)-mass(0)}{mass(0)}$ of one fluid in the domain (for both of the above test cases) for the original, modified serial, parallel GMRES, deflated, and parallel IDR(s) code. Due to the mass conservation the normalized mass should ideally be 0, this is reflected in Figure 7.7a and Figure7.7b where it is always less than the specified tolerance of $1x10^{-8}$.

7.3 SPEEDUP & SCALABILITY OF PARALLELIZATION

In this section, we shall present in detail the speedup results for the cases described above. Also, we shall demonstrate the scaling properties of the parallelized code for different problem sizes. In the next section, the effect of deflation preconditioner and performance of IDR(s) in comparison to GMRES is discussed in detail. All the speedup comparisons are based on first 100 time steps only (unless otherwise stated), as running



(a) Benjamin bubble

(b) Rising bubble

Figure 7.7: Normalized Mass obtained by different codes.

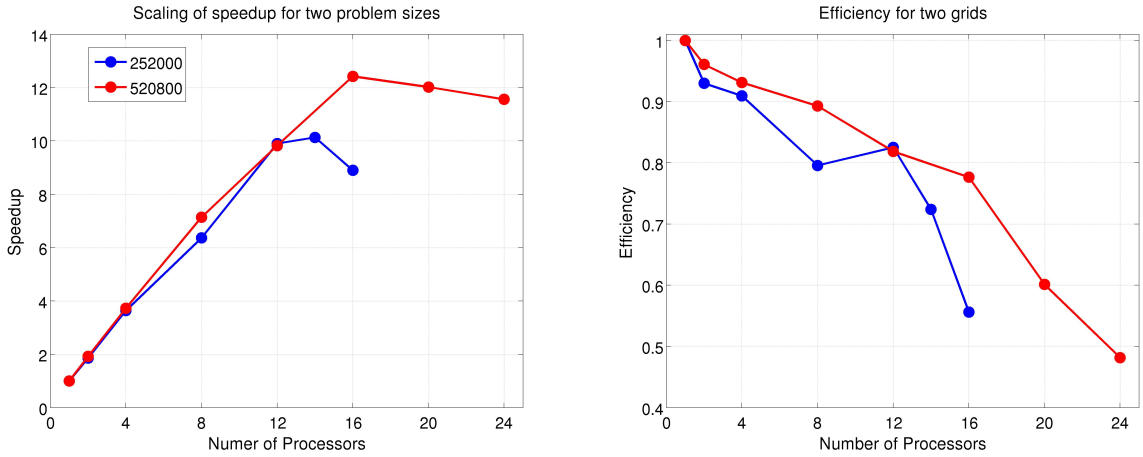
full simulations for all the grids and cases is not feasible.

7.3.1 Benjamin Bubble Problem

Table 7.8 gives the time taken to solve the Benjamin bubble problem (on different number of processors) for the above mentioned geometry and two different grids. As can be seen from Table 7.8, a big chunk of speedup is achieved due to the serial code modifications, this however is not 8-9 times as seen in the previous sections. This is due to the fact that we achieved 8-9 times speedup only for the predictor step, while for this test case the ratio of time taken by the predictor solver to the total time taken is much less than 1, hence the overall speedup gain is less.
The parallelization reduces the computational time even further, and behaves as expected i.e, the efficiency of parallelization reduces for a fixed grid as the number of processors increases. Figure 7.8a and Figure 7.8b shows the achieved speedup and efficiency of parallelization (for the two grids) with the number of processors. As evident from the plot, a nearly linear speedup is achieved for both the grids, and as the grid size increases the efficiency and speedup achieved at higher number of processors improves. Hence, as the problem (grid) size increases the number of processors at which the maximum speedup occurs increases. The non-monotonously decreasing efficiency indicates the better use of cache memory for the respective number of processors.

Table 7.8: Total computational time taken to solve Benjamin bubble for 2 grids on different number of processors.

Grid size	Time [s]								
	available code	modified serial	parallel modified (# cores)						
			2	4	8	12	16	20	24
50x60x84	1601	800.2	430.2	220.0	125.7	80.8	89.93	-	-
60x70x124	2995.2	2460.0	1280.1	660.4	344.4	250.4	198.0	204.6	212.8



(a) Scaling of Speedup

(b) Scaling of Efficiency

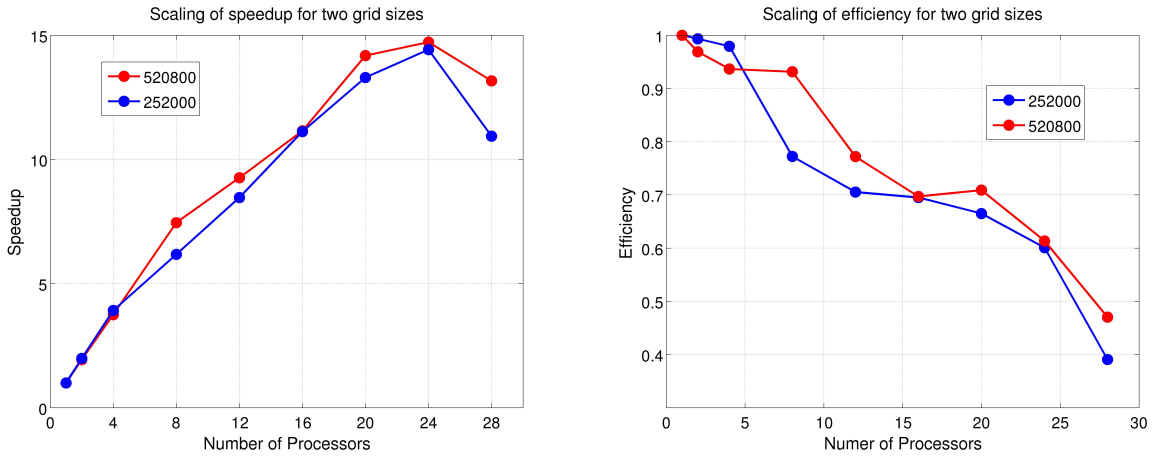
Figure 7.8: Scaling for the Benjamin bubble problem.

7.3.2 Rising Bubble

Table 7.9 shows the computational time taken for performing 100 time steps (on two different grids) for the rising bubble problem on different number of processors. As evident from the table, there exists a very large difference between the computational times taken by the serial available code and modified serial code. This speedup is much more than the speedup seen in the Benjamin bubble case, as for this problem, solving the predictor step takes the largest fraction of the computational time, and the modifications performed are quite efficient in obtaining a very high speedup.
As in the previous case, Figure 7.9a and Figure 7.9b show how the achieved speedup and efficiency of parallelization (for the two grids) vary with the number of processors. As can be seen from the plots a maximum speedup of around 15 times is obtained on 24 processors, and as expected, the efficiency of parallelization improves as the grid size increases. One important point to be noted here is that the scaling does not

Table 7.9: Total computational time taken to solve Rising bubble for 2 grids on different number of processors.

Grid size	Time [s]									
	avail. code	mod. serial	parallel modified (# cores)							
			2	4	8	12	16	20	24	28
50x60x84	13500	1186.6	597.3	302.9	192.1	140.2	106.7	89.3	82.4	108.5
60x70x124	27940	2606.1	1344.7	695.6	349.7	281.4	233.7	183.8	177.1	198.2



(a) Scaling of Speedup

(b) Scaling of Efficiency

Figure 7.9: Scaling for the rising bubble problem.

improve much as the problem size increases. This is due to the fact that as the grid is refined, the problem behavior changes due to which the solvers behave differently, leading to different parallelization characteristics. Further, the non-monotonicity in the parallelization efficiency comes from the fact that we have multiple levels of memory (cache) and the load on the cluster is different at different times.

7.4 SPEEDUP GIVEN BY SOLVERS AND PRECONDITIONERS

In this section, we first test how efficient the deflation preconditioner is in improving the convergence properties of the Krylov subspace solvers for the type of problems in hand. Further, as described before, the IDR(s) method can be more efficient than the restarted GMRES method since it does not need to store all the Arnoldi vectors,

resulting in lower a computational and storage overhead. Hence later in this section, we also test if the IDR(s) method is in fact more efficient than the restarted GMRES method, or the benefits of lower overhead is subsided by a loss in convergence. It is stressed at this juncture, that the accuracy of implementation of the deflation technique and the IDR(s) method were demonstrated in the previous sections.

7.4.1 Deflation

As described earlier, we implemented deflation as a preconditioner to improve the performance of the diagonalized predictor equation, and to improve the efficiency of the incomplete Cholesky preconditioned CG method over decomposed domains. The deflation method did not reduce the number of iterations required by the Krylov solver to solve the predictor step, as the system matrix has quite favorable spectral properties. This is due to the presence of $\frac{1}{timestep}$ term on the diagonal which makes the matrix highly diagonally dominant. Figure 7.10 shows the smallest 50 eigenvalues and the Gerschgorin circle of maximum radius, defined as $\{\max_i R_i \mid R_i = \sum_{j \neq i} |a_{ij}|,\ i, j \in 1, ..., n\}$, of a typical diagonally scaled system matrix obtained in one of the simulations. The diagonal scaling of the matrix helps to cluster the eigenvalues of the matrix around one as shown in Figure 7.10. Due to this clustering of the eigenvalues, deflating a few small ones does not help in improving convergence properties of the solver.

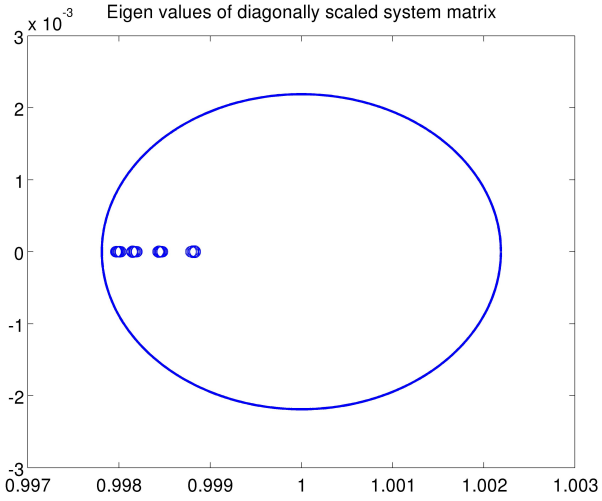


Figure 7.10: Smallest 50 eigenvalues of a typical diagonally scaled system matrix.

The Deflation method, however, proved to be quite useful in improving the convergence properties of the incomplete Cholesky preconditioner used to solve the Poisson

equation. Table 7.10 shows the number of iterations taken by the ICCG and deflated ICCG methods for the rising bubble case on different number of processors. As can be seen from Table 7.10, the number of iterations taken by the CG method reduces considerably, the total computational time however does not reduce. This is due to the fact that the system matrix of the ICCG method is *relatively* well conditioned for our case. Figure 7.11a shows the convergence history of the ICCG and deflated ICCG methods on different number of processors. The plot indicates that for the ICCG method, there exist a few eigenvalues which initially stall the convergence. These eigenvalues, however, are quickly overcome by the ICCG method, and after that, the convergence is more or less monotonic. The deflation method deflates these eigenvalues, and the convergence is nearly monotonic (Figure 7.11a) even initially. But because we specify a very small tolerance on the residue, the gain given by deflation is subsided by the large number of iterations required by both the methods to converge. Hence, though deflation reduces the number of iterations, it still requires sufficiently large number of iterations such that the extra time taken by the deflation process itself is more than the time saved by it due the improved convergence of the ICCG method.

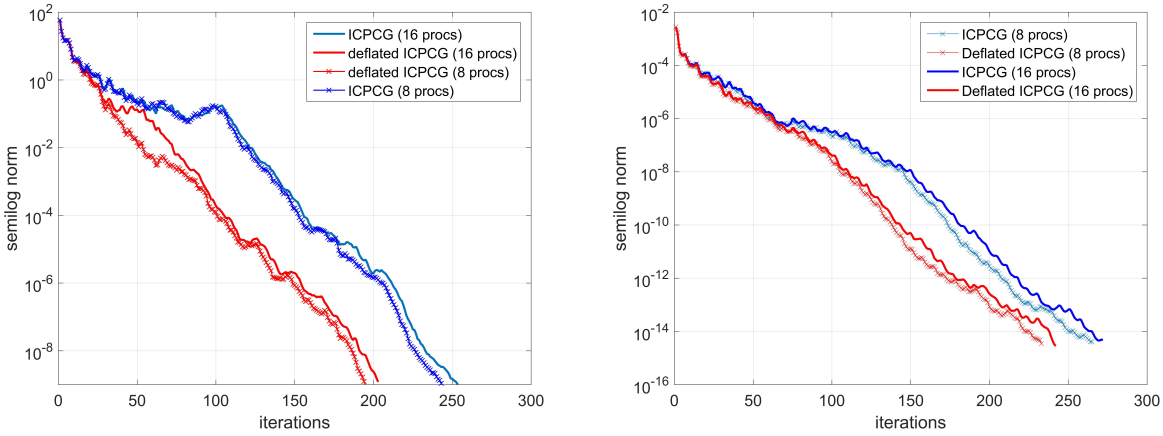
Table 7.10: Time and iterations taken by the Poisson solver for a single integration step (Rising bubble).

		# cores			
		8	12	16	20
ICCG	# iterations	274	278	285	295
Deflated ICCG	# iterations	225	229	233	237
ICCG	Time(s)	0.73	0.478	0.37	0.34
Deflated ICCG	Time(s)	0.76	0.57	0.5	0.48

To validate the above point, a similar simulation as above with a lower tolerance is performed and Table 7.11 gives the obtained results. As evident, for a smaller number of sub-domains (processors) we indeed got a speedup in terms of time as well, but as the number of subdomains (processors) increase the speedup due to the deflation technique decreases. This is due to the fact that the ICCG method does not perform particularly bad as the number of processors increase (Figure 7.11a), but the size of matrix $E = Z^T AZ$ increases and hence the time taken by the deflation module itself increases to high values. While at a relatively lower number of processors, the size of matrix E is small and deflation takes less time which leads to a reduced overall computational time.
Table 7.12 shows the number of iterations and time taken by the ICCG and deflated ICCG methods for the above discussed Benjamin bubble problem. As in the previous

Table 7.11: Time and iterations taken by the Poisson solver for a single integration step (Rising bubble), tolerance 1e-4.

		# cores			
		8	12	16	20
ICCG	# iterations	170	180	176	188
Deflated ICCG	# iterations	114	120	117	121
ICCG	Time(s)	0.48	0.34	0.29	0.24
Deflated ICCG	Time(s)	0.42	0.31	0.29	0.27



(a) Rising bubble

(b) Benjamin bubble

Figure 7.11: Convergence history of ICCG and deflated ICCG.

case, the deflation techniques helps to improve the convergence properties of the incomplete Cholesky preconditioner, but the time taken is more for the deflated case for all number of processors. The same reasoning as in the previous case is valid, i.e, since the matrix is relatively well conditioned, the effect of domain decomposition on the preconditioner does not affect the convergence by much (Figure 7.11b). Hence even though deflation helps to reduce the number of iterations initially, for converging to higher tolerances, the time taken by the deflation module itself is more dominant, and therefore the total time increases.

From the above discussion, the convergence benefits of the deflation technique for the problem at hand is quite clear. The time gain, however, varies from problem to problem and the user is advised to check the time benefits of deflation as per case.

Table 7.12: Time and iterations taken by the Poisson solver for a single integration step (Benjamin bubble).

		# cores			
		8	10	12	16
ICCG	# iterations	269	271	276	280
Deflated ICCG	# iterations	236	244	247	254
ICCG	Time(s)	0.71	0.47	0.41	0.36
Deflated ICCG	Time(s)	0.84	0.62	0.53	0.51

7.4.2 IDR(s)

The IDR(s) method described earlier has been tested as a replacement to the restarted GMRES method to solve the predictor equation. To extensively compare the two methods, computations have been performed using both the methods for both rising bubble and Benjamin bubble cases on different number of processors. We shall discuss them one by one.

Table 7.13 shows the computational time and number of iterations taken by the predictor step for the Benjamin bubble case. The table is for different number of processors, and different restarting points for GMRES and s values for IDR(s). As can be seen, the GMRES method converges quite quickly even as the frequency of restarting increases. This is because the system matrix has extremely well spectral conditions due to the presence of $\frac{1}{timestep}$ term on its diagonal, further the jump in the system coefficients (viscosity and density) is pretty minimal in this case which leads to the system matrix having even better spectral properties. The IDR(s) method on the other hand performs as expected, but due to the extremely few number of iterations required by the restarted GMRES method the overhead of storing and using the Arnoldi vectors is pretty minimal, hence the benefits of the IDR(s) method does not outweigh the ones of the GMRES method.

Table 7.13: Restarted GMRES versus IDR(s) for solving one predictor step on different number of processors (Benjamin bubble).

	# cores	Restarted GMRES		IDR(s)	
		Restarting iteration		s	
		10	5	10	5
#iterations (time(s))	4	8 (0.54)	8 (0.53)	8 (0.62)	8 (0.62)
	8	8 (0.36)	8 (0.37)	8 (0.43)	8 (0.41)
	16	8 (0.21)	8 (0.20)	8 (0.20)	8 (0.21)

A similar analysis is also performed for the rising bubble problem. For this case, the matrix spectral properties are slightly worse in comparison to the matrix properties for the Benjamin bubble problem because of higher differences in the physical coefficients (viscosity and density). Table 7.14 shows the computational time and number of iterations taken by the predictor step for different restarting points of restarted GMRES and different s values for IDR(s). The minimum achieved computational times are also highlighted. For this case, as the restarting frequency of GMRES increases, the computational time also increases as the number of iterations required to converge increases considerably, and the gain given by storing less Arnoldi vectors due to restarting is overpowered by the extra time taken because of the loss of convergence. The performance of IDR(s), on the other hand, is not much affected as s decreases because IDR(s) is not a truncating or restarting method, and s just indicates the number of previously computed residues based on which the current residue is calculated. Hence reducing s does not affect the convergence of IDR(s) as severely as restarting affects that of GMRES. This benefit of IDR(s) gives a speedup for this case as can be seen in Table 7.14.

Table 7.14: Restarted GMRES versus IDR(s) for solving one predictor step on different number of processors (Rising bubble).

	No. of cores	Restarted GMRES			IDR(s)		
		Restarting iteration			s		
		No restart	10	5	10	5	3
# iterations (time(s))	4	17 (0.85)	22 (1.0)	30 (1.3)	17 (1.14)	17 (0.96)	18 (0.82)
	8	17 (0.45)	22 (0.52)	30 (0.63)	17 (0.67)	18 (0.45)	18 (0.41)
	16	17 (0.35)	22 (0.39)	30 (0.49)	17 (0.42)	18 (0.34)	18 (0.32)

From the above table it is clear that IDR(s) can indeed save computational time if the GMRES method requires storage of sufficiently large number of Arnoldi vectors. To prove this point even further, for the same case, simulations were carried out with a larger time step (0.5 ms) which might be required depending on the problem. This results in a system matrix having slightly worse spectral properties due to the reduction of $\frac{1}{timestep}$ term on the diagonal. Table 7.15 shows the obtained results, and as can be seen the GMRES method requires much higher number of iterations to converge in comparison to the case having a smaller time step. This leads to the storage of large number of Arnoldi vectors on the part of GMRES, resulting in higher overhead cost. Restarting the GMRES method after 75 iterations helps (for this case) to reduce the overhead cost, but as the restarting is done more often the number of iterations required to converge increase so much that the benefits of lower overhead is subsided by the time required to perform extra iterations. The IDR(s) method, however, performs quite well

for this case in comparison to the (restarted) GMRES method. As can be seen from the table, as s decreases, the overhead cost reduces without a significant loss in the convergence resulting in a lower computational time.

Table 7.15: Restarted GMRES versus IDR(s) for solving one predictor step on different number of processors (Rising bubble, larger time step(0.5ms)).

	No. of cores	Restarted GMRES			IDR(s)		
		Restarting iteration			s		
		unrestarted	75	50	10	5	3
# iterations (time(s))	4	100 (7.9)	125 (7.6)	177 (8.8)	103 (4.8)	113 (4.5)	122 (4.2)
	8	100 (0.45)	125 (4.1)	176 (4.8)	106 (3.5)	111 (2.6)	117 (2.4)
	16	100 (3.0)	125 (2.7)	177 (3.6)	112 (2.9)	113 (1.9)	110 (1.6)

Figure 7.12 shows the overall simulation time taken, for the above case (higher time step), to perform 20 time integration steps on different number of processors if the predictor equation is solved using the GMRES/IDR(s) method. For this case, the fastest IDR solver is nearly 2 times faster than the fastest restarted GMRES solver (per predictor equation solve), while in terms of overall speedup a factor of nearly 1.5 is obtained (Figure 7.12). As is evident from Table 7.15 and Figure 7.12, the IDR(s) method can really help in reducing the computational cost of the predictor step if GMRES takes sufficiently long time to converge, that is, the spectral properties of the system matrix is such that it is not conducive to quick convergence. Such spectral properties may result from higher time steps or larger differences in the system coefficients.

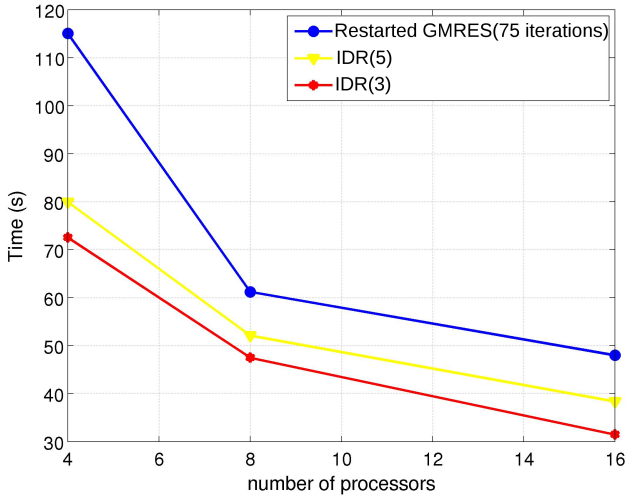


Figure 7.12: Overall simulation time after using Restarted GMRES/IDR(s) to solve the predictor step.

Chapter 8

PRACTICAL CAPABILITIES/USE OF THE MODIFIED PARALLEL CODE

In this section, we shall demonstrate the benefits of the parallel modified code by simulating a real world problem which otherwise takes a prohibitively large amount of time on the original available code. Below, we describe the test case first, and then analyze the results we obtain.

As mentioned in the introduction, TNO-Netherlands, Shell and Deltares are interested in multiphase flow happening in pipes. In such cases, pipes are initially filled with oil, and water is pumped to flush the oil out. Many experiments to visualize the shape and propagation velocity of the liquid-liquid interface during the flushing process have been performed in the past. We simulated one such problem provided by TNO (for two different geometries) to demonstrate the benefits of the newly developed code for solving the real world problems. Next, the details of the simulated test cases are provided.

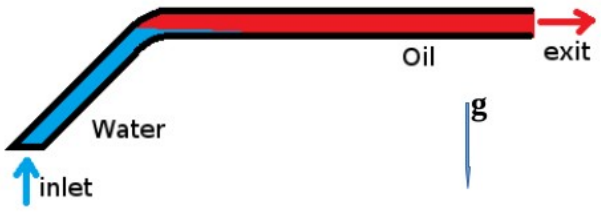


Figure 8.1: Geometry and configuration of the simulated pipe.

We simulate the flow field inside a pipe configured as indicated in Figure 8.1. Initially, the pipe is assumed to be filled with stagnant oil, and at t=0, water is provided at the inlet to flush the oil out. It was noted in the experiments that after water reaches the bend, it instead of rising further at the same speed, creeps horizontally displacing the oil. The resulting interface is difficult to capture numerically. It is seen in the previous simulations done at TNO, that a no-slip boundary condition gives an unphysical interface shape [19] (where an oil film is formed between water and the wall), while a slip boundary condition over-predicts the speed of the interface. For more details on the physics of the problem we refer to [19].
As mentioned earlier, we test the behavior of the interface for two different geometries. Table 8.1 gives the dimensions and test case properties for both the geometries. The first geometry is smaller than the full scale model, and the aim of this test case is to capture the physics correctly. Another point to be noted here is that we do not actually have a bend in our geometry, rather the bend is approximated by fixing an appropriate gravity vector.

Table 8.1: Geometry of the pipe and test case properties.

	subscale (first case)	full scale (second case)
Pipe Radius [mm]	30	30
Angle of Slant	26.4^o	26.4^o
Slant Pipe Length [mm]	250	500
Bend Circumference [mm]	200	1000
Horizontal Pipe Length [mm]	350	1700
Total Pipe Length [mm]	800	3200
Grid Size	30x40x200	30x40x800
Density of Fluids [kg/m^3]	1000 & 840	1000 & 840
Viscosity of Fluids [kg/m/s]	6.58x10^{-4} & 4.03x1e^{-2}	6.58x10^{-4} & 4.03x1e^{-2}
Surface Tension [N/m]	0.0	0.0
Flow Rate of Water [m^3/s]	0.00013005	0.00013005

As boundary conditions, we implement a no-slip boundary condition at the walls to see whether we get the same results as reported in [19]. Further, a parabolic velocity profile with a homogeneous pressure Neumann boundary condition is specified at the inlet. While at the outlet, a homogeneous Neumann boundary condition is implemented for both velocity and the pressure. The parabolic velocity profile at the inlet is given by $c(R^2 - r^2)$, where R is the radius of the pipe, r is the radial distance where the velocity is to be calculated, and c is calculated such that the total flow rate is equal to the specified flow rate. Moreover, so as to get a divergence free initial condition, the

simulation is run at a very small time step (1μs) for first 50000 iterations, and only later, the time step is increased to 0.5ms so as to get a quicker solution.
For running the simulation, a test was conducted to determine the number of processors which gives the maximum speedup for the current case. The results of the test are reported in Table 8.2 which gives the time taken to perform 100 time steps on different number of processors. As can be seen clearly, the minimum time was taken by 20 processors, and hence, this simulation was performed on the same number of processors.

Table 8.2: Time taken for 100 time steps by the first geometry on different number of processors.

Number of processors	8	12	16	20	24
time [s]	160.6	95.66	78.13	64.81	66.03

Further, it was noted that the ICCG method takes a high number of iterations to converge to the specified tolerance, the convergence history for the ICCG method is shown in Figure 8.2. As can be seen, the convergence of the ICCG method is stalled for long initially, this points to the existence of small eigenvalues in the spectrum of the system matrix which slows down the convergence. Hence, deflation was used to deflate these small eigenvalues and improve the efficiency of ICCG. Figure 8.2 gives the convergence history of the deflated ICCG method as well, and as evident, the deflation preconditioner drastically improves the convergence of the iterative solver. Table 8.3 gives the number of iteration and time taken by both the methods to solve the Poisson equation. The deflation method, unlike in the Benjamin bubble or rising bubble case, not only improves the convergence, but also reduces the computational time. A reduction of 1.5 times is obtained in the computational time, while the convergence is accelerated by nearly 3 times. The computational time is not reduced by the same factor as the convergence is improved because the deflation preconditioner itself takes quite some computational time.
Because of the achieved speedup, the simulation was performed on 20 processors by using deflated ICCG to solve the Poisson equation. The simulation took approximately 30 hours of computational time. Next, we discuss the obtained results.
Figure 8.3 shows the position and shape of the interface with time. Initially, the interface movement is nearly translational until it reaches the bend (the red line marks the starting of the bend and the black line marks the ending). After the interface reaches the bend, the water does not rise at the same rate as before, but creeps horizontally instead. This is completely in line with the physics explained earlier. Further, in our simulation, there is no unphysical oil film formed between the wall and the interface

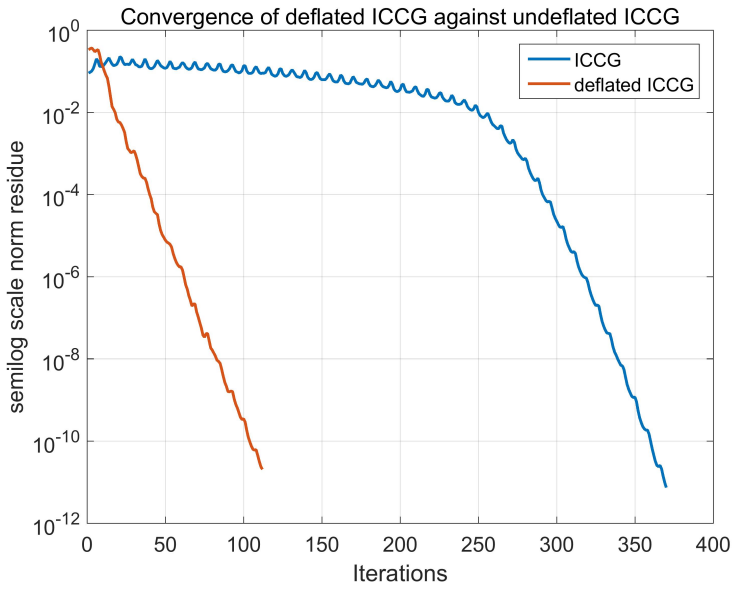


Figure 8.2: Convergence history for ICCG and deflated ICCG method.

Table 8.3: No. of Iterations and time taken by (deflated) ICCG to solve the Poisson equation.

ICCG		deflated ICCG	
Number of iterations	Time [s]	Number of iterations	Time [s]
370	0.36	112	0.242

as the water displaces the oil in the horizontal part (as reported in [19]). The obtained interface shape and its movement indicate that the code captures the physics of the problem at least qualitatively. More quantitative results are obtained from the full scale simulation discussed next.

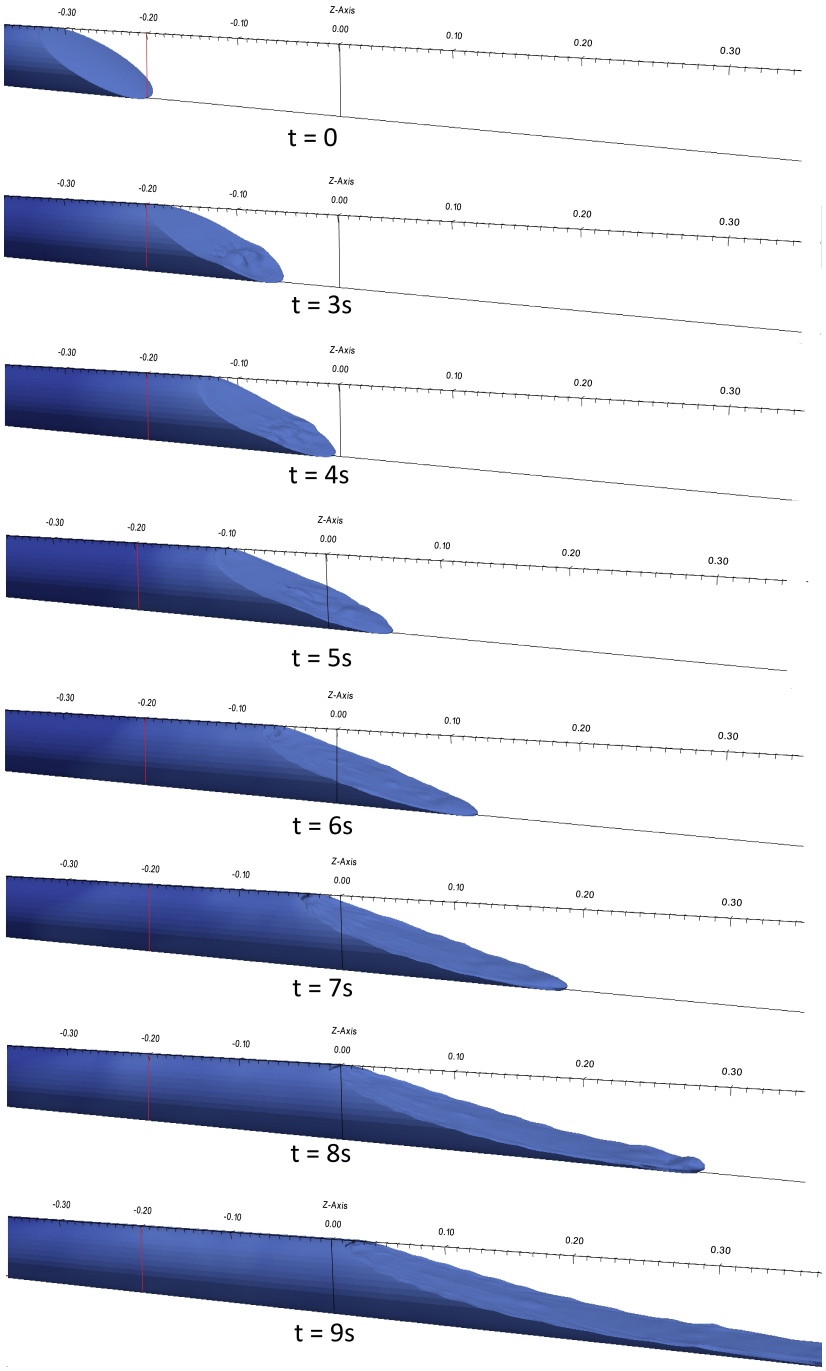


Figure 8.3: Movement of the interface with time for the subscale problem.

For the full scale geometry, the simulation was carried out for the same fluids and boundary conditions as the subscale geometry. We conducted tests to derive the optimal number of processors and concluded that running this problem parallely on 40 processors gives the highest speedup and efficiency factor. Further, as above, it was noticed that the ICCG method converges quite slowly while solving the Poisson equation and deflation improves the convergence of the ICCG method many folds. This is evident from Figure 8.4 which shows the convergence history of the deflated and undeflated ICCG methods for the full scale problem. Due to this improvement in convergence, we get a speedup of nearly 2 times in the Poisson step as can be seen from Table 8.4 which gives the number of iterations and time taken to solve one Poisson equation. At this point it should also be noted that deflation performed nearly 3 times better (in terms of convergence) as the size of the pipe increased from $0.8m$ (subscale geometry) to $3.2m$ (full scale geometry). This suggests that for bigger problems, bigger gains from deflation could be expected.
Furthermore, owing to the presence of $\frac{1}{timestep}$ term on the diagonal and lower jump in the coefficients across the interface, we noticed that the GMRES method converged quickly, hence the use of IDR(s) method was not required to solve the predictor problem in this case.

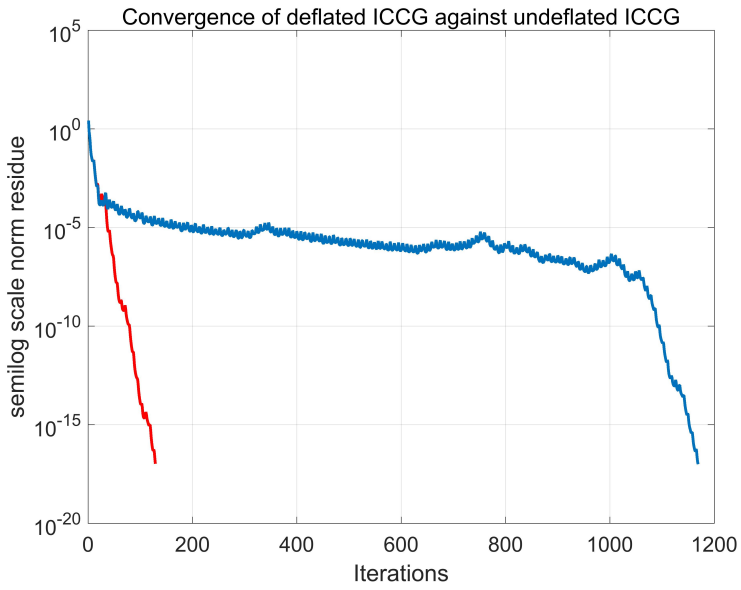


Figure 8.4: Convergence history for ICCG and deflated ICCG method for the full scale model.

Table 8.5 shows the total computational time taken by the deflated ICCG code on 40

Table 8.4: No. of iterations and time taken by (deflated) ICCG to solve the Poisson equation.

ICCG		deflated ICCG	
Number of iterations	Time [s]	Number of iterations	Time [s]
1167	3.3	129	1.7

cores and the original available code to perform 10 time integration steps (for the full scale problem). As evident, we gain an overall speedup of nearly 75 times. Due to these observations, the current simulation was carried out on 40 processors by using the diagonalized restarted GMRES method to solve the predictor equation and deflated ICCG method to solve the Poisson equation. This simulation took nearly 6 days to complete on 40 cores, and if we extrapolate the performance results of the original available code, we see that the simulation would have taken more than 12 months without the modifications performed in this endeavor. This huge time saving demonstrates the benefits of the new code and indicates that it can be efficiently used to further the research in the area of pipe flows.

Table 8.5: Total time taken by available code and deflated ICCG code on 40 processors to integrate 10 time steps.

	available code	deflated ICCG on 40 procs
Time [s]	2818.51	37.29

To see further how the modifications performed in this endeavor change the fraction of time taken by each module, we performed some profiling tests for the full scale problem simulated by the original available code and by the parallelized (40 processors) deflated ICCG code. Table 8.6 gives the obtained results. As can be seen, for the original code the predictor part is the heaviest and takes 65% of the total time. While for the parallelized deflated ICCG code, not only is the computational time for each module is reduced drastically, but also the bottleneck has changed, i.e., the predictor part is no more the heaviest step, but the Poisson step is (which takes 50% of the total time). This demonstrates that the efficiency of the predictor step is improved by much, and now to improve the speedup even more, one should modify the Poisson step.
Next, we discuss the physical results obtained by the present simulation and contrast it with the results presented in [19]. Figure 8.5 gives the movement of the interface with time for the full scale test case. As seen before, as the interface reaches the end of the bend, the water creeps horizontally instead of translating forward at the same rate as before. In this case however, we see the formation of a layer of oil between water and the wall as reported in [19]. We would like to stress at this juncture that

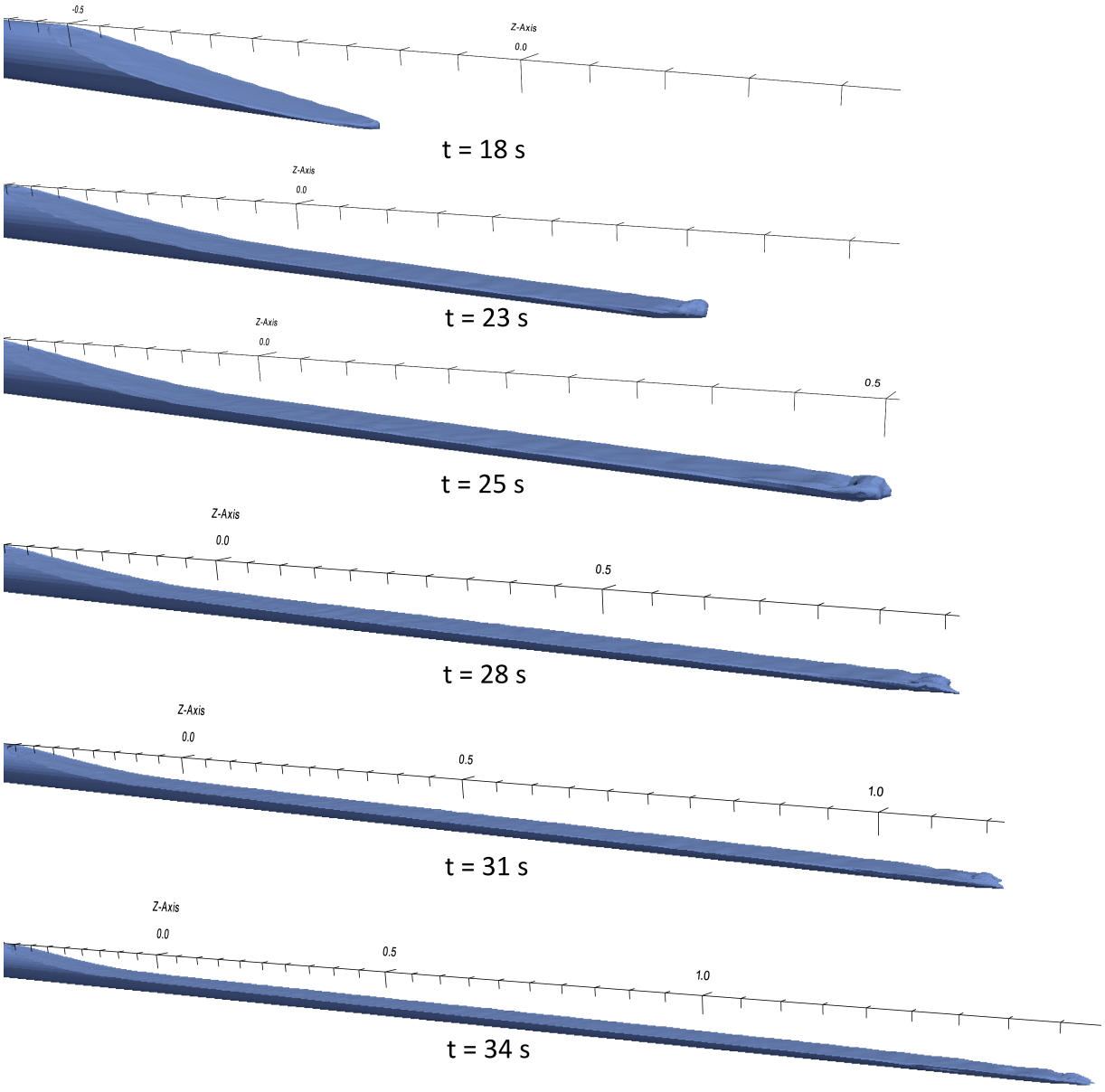


Figure 8.5: Movement of the interface with time for the full scale problem.

the comparison between the present results and the results reported in [19] can not be one-to-one because of the following reasons:

1. The reported results are for 2-d channel flow, while we have a 3-d pipe flow.
2. The reported results are obtained by implementing an adaptive grid refinement,

Table 8.6: Profiling results of the original and modified codes for one time step of the full scale problem.

Code	Flow Time[s] - percentage			Interface Time[s] - percentage
	Predictor Time[s] / percentage	Poisson Time[s] / percentage	Corrector Time[s] / percentage	
Original available code	237.02 - 99.24%			1.78 - 0.75%
	154.1 - 65.0%	80.2 - 33.8%	0.923 - 0.4%	
Parallel deflated ICCG code	3.14 - 90.2%			0.34 - 9.75%
	1.32 - 37.8%	1.7 - 50.48 %	0.046 - 1.3%	

while uniform grid is used in the current endeavor.

3. The initial conditions are different for our case, i.e., in [19] the pipe is assumed to be initially filled with oil, and at $t = 0$ water enters the pipe and the mass flow rate of water is increased until $t = 5s$. While for our case, we assume that the interface is already present in the slant portion of the pipe (the location is chosen arbitrarily) and we specify a constant in time parabolic velocity profile at the inlet.

4. The interface is captured using the VOF method in [19], while in the present study we use the MCLS method, which is a combination of the Level Set and VOF methods.

5. The length of the horizontal part for the current test case is 3.2 m, while in [19] it was 15 m.

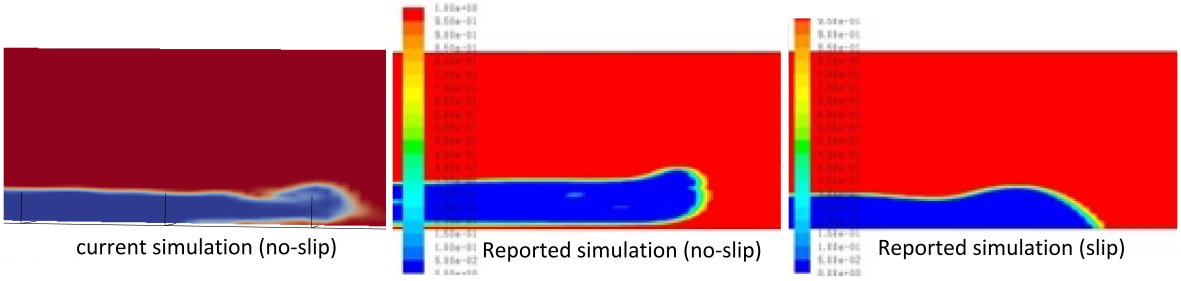


Figure 8.6: Comparison between the shape of the interface head captured by the current simulation and as reported in [19].

Even though we can not draw a one-to-one comparison, Figure 8.6 compares the interface profile reported in [19] (with and without no-slip boundary condition) with the interface profile at the symmetry plane obtained from the present simulation. There are few key points to be noted. Firstly, the shape of the head of the interface given by the present simulation is somewhere in between the shape of the heads obtained by using the no-slip and slip boundary conditions in the reported results. Secondly, even though there is a thin film of oil between water and the wall in the current simulation, it is smaller from the one reported in [19]. Further, we computed the speed of the interface calculated as:

$$speed = \frac{distance\ moved\ by\ the\ contact\ point\ of\ oil,\ water\ and\ the\ wall}{time}$$

Table 8.7 gives the interface velocity obtained from the experiments, simulations reported in [19], and the current simulation. It can be seen that the velocity obtained in the present endeavor is the closest match to the experimental results. However, this must be taken as an indicative result only as the grid independence studies have not been performed, and changing the refinement of the grid may result in a different interface velocity.

Table 8.7: Interface velocity obtained from experiments and simulations.

	experiments	current (no-slip)	reported (no-slip)	reported (slip)
velocity $[m/s]$	0.143	0.15	0.11	0.165

Furthermore, it is to be noted that for the smaller test case, we did not obtain any film of oil between water and the wall as seen in the full scale test case. This is quite intriguing and points to the fact that the length of the test case affects the presence of this unphysical layer of oil, and this phenomenon should be studied in more detail.
Moreover in the available code, a Navier-slip boundary condition is implemented in which the shear stress near the contact point of the interface and wall can be specified, i.e., we have a degree of freedom with which we can specify the slip near the interface-wall contact point and simulate the physics better. One can test this boundary condition to see if it captures the interface characteristics better.

This study of the full/sub scale test cases, though quite brief, indicates that the physical results given by the current code are close to the experimental results. Moreover, the results are obtained in much less computational time (as compared to the original available code) due to the modifications performed in this thesis. These benefits make clear that this code can be extensively used to study pipe flow problems.

Chapter 9

CONCLUSION

The aim of this study was to improve the efficiency of a CFD code which is used to simulate multiphase flows. To enhance the efficiency of the available code, it was proposed to check how the performance of the Krylov subspace solvers can be improved. Moreover, it was planned to parallelize the code based on domain decomposition, and check how much speedup is achieved by distributing the work amongst different processors. To improve the performance of parallelization, deflation was also aimed to be implemented. Further, it was also proposed to check if using the IDR(s) solver instead of GMRES to solve the predictor equation would reduce the computational time for our problems.

To answer the above questions, the available code was studied and profiled, and in addition, different solvers and preconditioners were studied to improve the performance of the available code. From the profiling results it was learnt that the predictor step took most of the computational time, hence the system matrix was diagonally scaled to improve the convergence of the Krylov subspace solver (GMRES method). The diagonal scaling improved the convergence of the GMRES method by a factor of 1.5-2 times depending on the case. Moreover, the structure of the code was modified so that the predictor matrix is not formed in each iteration of GMRES (as was the case originally). This modification further sped up the predictor module by nearly 4 times. Hence in total, depending on the case, the above modifications gave a speedup of nearly 4.5-9 times in the predictor module.

Further, parallelization was implemented to reduce the overall computational time by running the simulation on several processors simultaneously. It was seen that parallelization reduced the computational time for the tested cases many folds, for example, a maximum speedup of 10 times was achieved on 12 processors for the Benjamin bubble problem, while a speedup of nearly 15 times was achieved on 24 processors for the rising bubble problem. Moreover, the speedup achieved was nearly linear until

a certain number of processors (which varied from case to case), and if the number of processor was increased further, a drastic drop in the efficiency of parallelization was seen. Although as expected, as the problem (grid) size increased the efficiency of parallelization also improved.
Furthermore, the deflation preconditioner was implemented as a coarse grid corrector for the predictor equation, and to improve the performance of parallelization for the ICCG solver. Deflation did not help as a coarse grid corrector for the predictor equation owing to the presence of $\frac{1}{timestep}$ term on the diagonal of the system matrix, leading to it already having very good spectral properties. Therefore, the Krylov subspace solver itself converged quickly and deflation was not required to improve the spectral properties of the matrix further.
The deflation technique, however, improved the convergence of the ICCG solver by eliminating the small eigenvalues of the system matrix which were responsible for the slow convergence. Although, it was noticed that for few of the tested cases, the ICCG solver itself performed quite well and hence the deflation technique was not able to improve the performance of the solver significantly. This resulted in the deflation module itself taking more time (per Poisson equation solve) than it saved by improving the convergence of the ICCG method. For the TNO test cases, however, the deflation preconditioner improved the convergence of the ICCG method substantially. This substantial improvement in the convergence led to a noteworthy reduction in the computational time required per Poisson equation solve, giving an overall speedup.
Next, we tested how the IDR(s) solver performed in comparison to the GMRES solver. The IDR(s) method proved to be of great use in mitigating the computational overhead caused by the GMRES method (due to the storage of all the Arnoldi vectors). A factor of nearly 2 times was obtained for the cases where the predictor equation's system matrix had slightly poor spectral properties (owing to a higher time step or larger jumps in coefficients of the two fluids). However, if the matrix spectral properties were favorable to convergence and the GMRES method converged quickly, the IDR(s) method did not help in reducing the computational time further.
Later, we demonstrated the usefulness of the new parallelized code by simulating the real world sub/full scale test cases provided by TNO. As mentioned above, for these cases, the deflation preconditioner reduced the computational time required to solve the Poisson equation substantially. The IDR(s) method, however, was of no advantage for these cases since the GMRES method converged quickly. The parallelized deflated ICCG code took nearly 30 hours on 20 cores to simulate the subscale, and 6 days on 40 cores to simulate the full scale problem which could not be simulated by the available code due to the prohibitively large computational time it required per iteration. An overall speedup of nearly 75 times was obtained (for the full scale problem) by running the deflated ICCG code on 40 processors.

The speedup gained by using different solvers, preconditioners and parallelization makes it possible to simulate the real world problems in a reasonable amount of time, and the successful numerical simulation of the full scale TNO test case proves the value of the new code. From the results we achieved, it is evident that this code can definitely help a researcher carry out the numerical simulations for pipe flows in much less time.

In this endeavor we achieved considerable speedups, and improved the performance of the original code many folds. Using this new parallelized code, one could study the effect of the length of the pipe on the unphysical layer of oil formed between water and the pipe wall for the TNO test cases. Secondly, because of the achieved performance in the predictor step, the Poisson step has now become the bottleneck, and hence in a future study, one could look at the multigrid solvers to reduce the computational time of the Poisson module even further. Moreover, the structure of the Poisson solver which was not touched in the current thesis due to shortage of time could be modified. As a third future research direction, we propose that the pressure correction time integration method, used in the current study, could be replaced by solving for the velocity and pressure at the new time step in a coupled manner to have a more accurate kinetic energy conserving solution. A study in this direction is presented in Appendix A where we discuss the preconditioners to improve the convergence of the saddle point problem which would result if we indeed solved a coupled system. As a short conclusion to that study, we suggest the usage of the MSIMPLER preconditioner, as it is suitable for the unsteady problems and does not require formation of the Schur complement at every time step.

Appendix A

Future Work - Saddle Point Preconditioners

As explained in Chapter 2, the current procedure of time integration does not preserve kinetic energy of the fluid. To preserve the kinetic energy one has to solve the non-linear system ((2.2)) in a coupled manner, i.e., simultaneously solve for both the velocity and pressure. As explained, since solving such a non-linear system is very expensive, in future endeavors we can solve the linearized system ((2.4)/(2.3)) in a coupled manner which results in a saddle point problem. The saddle point problem is characterized by very slow convergence and hence an appropriate preconditioner is required to improve the convergence. This Chapter describes the preconditioners used to solve the saddle point problems, and serves as the building step on which the future works in this direction could be based.

For the saddle point problems broadly two types of preconditioners are available, i.e., the block preconditioners and the ILU type preconditioners. The ILU type preconditioners are better for finite element solvers where both matrix builder and solver must be adapted, since the splitting of velocity and pressure unknowns is required [20]. In this study, however, we only focus on the block type preconditioners.

Block preconditioners are based on the block LDU decomposition of the coefficient matrix A in equation (2.4). We write

$$A = LDU = \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} I & 0 \\ BF^{-1} & I \end{bmatrix} \begin{bmatrix} F & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & F^{-1}B^T \\ 0 & I \end{bmatrix}$$

where $S = -BF^{-1}B^T$ is the Schur complement matrix. Almost all the preconditioners are some form of combination of these blocks with an appropriate Schur complement matrix approximation.

If we choose a preconditioner (P) based on the product of only the diagonal matrix D and the upper diagonal matrix U, i.e.,

$$P = \begin{bmatrix} F & B^T \\ 0 & S \end{bmatrix},$$

then it is easy to show that the eigenvalues of the preconditioned system are all 1, hence GMRES converges in 2 iterations in exact arithmetic [21]. Computing the inverse of S and F is naturally not practical, hence a cheap approximation of S is used and the system $F\boldsymbol{u} = \boldsymbol{f}$ is solved approximately using iterative methods. Application of such a preconditioner requires solving $P\boldsymbol{z} = \boldsymbol{r}$ where $\boldsymbol{z} = [\boldsymbol{z}_1; \boldsymbol{z}_2]$ and $\boldsymbol{r} = [\boldsymbol{r}_1; \boldsymbol{r}_2]$. Now all that remains is to form an approximation of the Schur complement. As it turns out, there is a plethora of ways we can approximate the Schur complement. The way we do it leads to various block preconditioners. In the following subsections we discuss a few of these preconditioners.

A.1 PRESSURE CONVECTION-DIFFUSION PRECONDITIONER

Based on [22], let the Convection Diffusion (C-D) operator L be defined on the velocity space. Also let $\boldsymbol{w}_h$ be the approximate discrete velocity computed in the most recent Piccard iteration. Then L is given by

$$L = -\Delta \cdot (\nu\Delta) + \boldsymbol{w}_h) \cdot \Delta \ .$$

Let the commutator of L be $\epsilon = L\Delta - \Delta L_p$, where L_p is analogous to L but does not carry any physical meaning. If $\boldsymbol{w}_h$ is constant, the commutator is zero in the interior of domain and is small for smooth $\boldsymbol{w}$, hence the discrete commutator (in terms of matrices) defined as,

$$\epsilon_h = (Q_v^{-1}F)(Q_v^{-1}B^T) - (Q_v^{-1}B^T)(Q_p^{-1}F_p)$$

will also be small. Here, Q_v is the velocity mass matrix, Q_p is the pressure mass matrix and F_p is the discrete C-D operator on the pressure space. Assuming the commutator is indeed small then left multiplication of the discrete commutator by $BF^{-1}Q_v$ and right multiplication by $F_p^{-1}Q_p$ gives an approximation of the Schur complement

$$-BF^{-1}B^T \approx -BQ_v^{-1}B^TF_p^{-1}Q_p \tag{A.1}$$

Since $BQ_v^{-1}B^T$ is expensive to compute, we replace it with its spectral equivalent matrix A_p known as pressure Laplacian matrix, thus giving

$$S = -BF^{-1}B^T = A_pF_p^{-1}Q_p.$$

This preconditioner gives a very good convergence if used with the Krylov subspace methods for enclosed flows (if the convective terms are linearized by the Piccard lin-

earization). But for other problems this preconditioner may not be ideal as A_p is defined only for the enclosed flow problems [20].

A.2 LEAST SQUARES COMMUTATOR PRECONDITIONER

The Least Square Commutator (LSC) preconditioner was given by Elman et al. [23], and is based on the same principle as the PCD preconditioner discussed above. We approximate F_p in a way which gives us a small discrete commutator. Therefore we solve the following least squares problem

$$min\|[Q_v^{-1}FQ_v^{-1}B^T]_j - Q_v^{-1}B^TQ_p^{-1}[F_p]_j\|_{Q_v},$$

where the j-th column of matrix F_p is represented by $[F_p]_j$, the j-th column of matrix $Q_v^{-1}FQ_v^{-1}$ is $[Q_v^{-1}FQ_v^{-1}]_j$, and $\|.\|_{Q_v}$ is the energy norm with respect to Q_v. The associated normal equations are

$$Q_p^{-1}BQ_v^{-1}B^TQ_p^{-1}[F_p]_j = [Q_p^{-1}BQ_v^{-1}FQ_v^{-1}B^T]_j,$$

which gives the following equation for F_p:

$$F_p = Q_p(BQ_v^{-1}B^T)^{-1}(BQ_v^{-1}FQ_v^{-1}B^T). \tag{A.2}$$

Equation (A.1) and (A.2) gives the Schur complement approximation

$$BF^{-1}B^T \approx (BQ_v^{-1}B^T)(BQ_v^{-1}FQ_v^{-1}B^T)^{-1}(BQ_v^{-1}B^T), \tag{A.3}$$

where Q_v is approximated by its diagonal elements to reduce the complexity of inverting it. This gives rise to the following algorithm:

//First compute $\boldsymbol{r}_u = \boldsymbol{f}$ and $\boldsymbol{r}_p = -BK^{-1}\boldsymbol{f} + \boldsymbol{g}$ then,
1. Solve $S_p\boldsymbol{z}_p = \boldsymbol{r}_p$, where $S_p = BD_v^{-1}B^T$, D_v =diag(Q_v)
2. Update $\boldsymbol{r}_p = BD_v^{-1}FD_v^{-1}B^T\boldsymbol{z}_p$
3. Solve $S_p\boldsymbol{z}_p = -\boldsymbol{r}_p$
4. Update $\boldsymbol{r}_u = \boldsymbol{r}_u - B^T\boldsymbol{z}_p$
5. Solve $F\boldsymbol{z}_u = \boldsymbol{r}_u$

Algorithm 3: LSC preconditioner

A.3 SIMPLE PRECONDITIONER

SIMPLE method was first introduced by Patankar & Spalding as a method to solve the coupled system iteratively. The following steps outline the method proposed by Patankar and Spalding

- Initialize pressure and velocity with the pressure velocity from previous step.
- Then we solve for velocity using the momentum equation, and pressure from the Poisson equation obtained while imposing the solenoidicity of velocity.
- We continue this procedure until desired convergence is reached.

The SIMPLE method, though very easy to implement, shows poor convergence properties for most of the problems. Although if used as a preconditioner [20], the spectral properties of Krylov subspace methods are much improved. It can be proven that some of the eigenvalues are clustered near 1, while the others are dependent on the approximation of the Schur complement. Further, we discuss the formulation of such a preconditioner.
Let the system we have to solve be $A\boldsymbol{x} = \boldsymbol{b}$ where,

$$A = \begin{bmatrix} F_r & 0 & 0 & G_r \\ 0 & F_\theta & 0 & G_\theta \\ 0 & 0 & F_z & G_z \\ G_r^T & G_\theta^T & G_z^T & 0 \end{bmatrix} = \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix}, \boldsymbol{x} = \begin{bmatrix} \boldsymbol{u}_r \\ \boldsymbol{u}_\theta \\ \boldsymbol{u}_z \\ \boldsymbol{p} \end{bmatrix}, \text{ and } \boldsymbol{b} = \begin{bmatrix} b_r \\ b_\theta \\ b_z \\ b_p \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix}$$

Then we base the SIMPLE preconditioner on the approximation of LU where L & U are lower and upper diagonal of matrix A. Thus

$$\begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} \approx \begin{bmatrix} F & 0 \\ B & S_a \end{bmatrix} \begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix},$$

where S_a is an approximate Schur complement constructed by approximating F by its diagonals in the definition of S, i.e., $S = -BF^{-1}B^T \approx -BD^{-1}B^T = S_a$. Thus one iteration of SIMPLE is to solve

$$\begin{bmatrix} F & 0 \\ B & S_a \end{bmatrix} \begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix} \begin{bmatrix} \delta\boldsymbol{u} \\ \delta\boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix} - \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{u}^{(k)} \\ \boldsymbol{p}^{(k)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_u \\ \boldsymbol{r}_p \end{bmatrix},$$

where $\delta\phi = \phi^{k+1} - \phi^k$. The above equation can be solved in two steps:

$$\begin{bmatrix} F & 0 \\ B & S_a \end{bmatrix} \begin{bmatrix} \delta\hat{\boldsymbol{u}} \\ \delta\hat{\boldsymbol{p}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_u \\ \boldsymbol{r}_p \end{bmatrix}$$

and

$$\begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix} \begin{bmatrix} \delta\boldsymbol{u} \\ \delta\boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \delta\hat{\boldsymbol{u}} \\ \delta\hat{\boldsymbol{p}} \end{bmatrix}.$$

Then we update the velocity and pressure based on

$$\begin{bmatrix} \boldsymbol{u}^{(k+1)} \\ \boldsymbol{p}^{(k+1)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}^{k} \\ \boldsymbol{p}^{k} \end{bmatrix} + \begin{bmatrix} \delta\boldsymbol{u} \\ \delta\boldsymbol{p} \end{bmatrix}.$$

Here, one of the above iterations is used as the SIMPLE preconditioner.
The SIMPLE algorithm has a problem, i.e., the method suffers a lot if the Reynolds number increases or the mesh size decreases. Another point to note is that many of the Krylov subspace methods, for instance CG and GMRES, require a constant preconditioner, or more specifically, a constant inverse of the preconditioner. If a preconditioner is changing due to the requirement of the problem to be solved or if we invert the preconditioner iteratively (i.e., we solve $P\boldsymbol{x} = \boldsymbol{y}$ iteratively where P is the preconditioner), this requirement of the Krylov subspace methods can not be fulfilled. The SIMPLE preconditioner is one of the above iterations, which itself is solved iteratively, hence the preconditioner changes in each iteration [20]. Therefore we use the GCR solver which, though a bit expensive, can handle variable preconditioners.

1. $\boldsymbol{u}^k$ and $\boldsymbol{p}^k$ are known from the previous iteration
2. Set $\boldsymbol{r}_u = \boldsymbol{f} - F\boldsymbol{u}^k - B^T\boldsymbol{p}^k, \boldsymbol{r}_p = \boldsymbol{g} - B\boldsymbol{u}^k$
3. Solve $F\delta\hat{\boldsymbol{u}} = \boldsymbol{r}_u$
4. $S_a\delta\hat{\boldsymbol{p}} = \boldsymbol{r}_p - B\delta\hat{\boldsymbol{u}}$
5. $\delta\boldsymbol{p} = \delta\hat{\boldsymbol{p}}$
6. $\delta\boldsymbol{u} = \delta\hat{\boldsymbol{u}} - D^{-1}B^T\delta\boldsymbol{p}$
7. $\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \delta\boldsymbol{u}$, $\boldsymbol{p}^{k+1} = \boldsymbol{p}^k + \delta\boldsymbol{p}$

Algorithm 4: SIMPLE preconditioner

A.4 SIMPLER PRECONDITIONER

There are many variants of the SIMPLE preconditioner available, one such preconditioner is SIMPLER [20]. In SIMPLER we first solve for the pressure $\hat{\boldsymbol{p}}$ instead of assuming it to be the same as the previous iteration's pressure $\boldsymbol{p}^k$, and then we apply the SIMPLE algorithm with $\hat{\boldsymbol{p}}$ instead of $\boldsymbol{p}^k$. The algorithm is given below

$$\text{Solve } S_a\hat{\boldsymbol{p}} = \boldsymbol{g} - B\boldsymbol{u}^k - BD^{-1}(\boldsymbol{f} - F\boldsymbol{u}^k), \text{ then}$$

$$\begin{bmatrix} F & 0 \\ B & S_a \end{bmatrix} \begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \delta\boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix} - \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} 0 \\ \hat{\boldsymbol{p}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_u \\ \boldsymbol{r}_p \end{bmatrix},$$

which can be solved in two steps

$$\begin{bmatrix} F & 0 \\ B & S_a \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{u}} \\ \delta\hat{\boldsymbol{p}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_u \\ \boldsymbol{r}_p \end{bmatrix}$$

and

$$\begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \delta \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \hat{\boldsymbol{u}} \\ \delta \hat{\boldsymbol{p}} \end{bmatrix}.$$

Then we update the velocity and pressure.

1. Solve $S_a \hat{\boldsymbol{p}} = \boldsymbol{g} - B\boldsymbol{u}^k - BD^{-1}(f - F\boldsymbol{u}^k)$
2. Set $\boldsymbol{r}_u = \boldsymbol{f} - B^T \hat{\boldsymbol{p}}, \boldsymbol{r}_p = \boldsymbol{g}$
3. Solve $F\hat{\boldsymbol{u}} = \boldsymbol{r}_u$
4. Solve $S_a \delta\hat{\boldsymbol{p}} = \boldsymbol{r}_p - B\hat{\boldsymbol{u}}$
5. Update $\delta \boldsymbol{p} = \delta \hat{\boldsymbol{p}}$
6. Update $\boldsymbol{u} = \hat{\boldsymbol{r}} - D^{-1}B^T \delta \boldsymbol{p}$
7. Update $\boldsymbol{p} = \hat{\boldsymbol{p}} + \delta \boldsymbol{p}$

Algorithm 5: SIMPLER preconditioner without pressure update damping

The SIMPLER preconditioner is supposed to have a convergence which is independent of the Reynolds number. Unfortunately, in practice, not much improvement is seen when graduating from SIMPLE to SIMPLER preconditioners. In fact for many of the test cases, SIMPLER performs worse than SIMPLE [20]. Hence, this preconditioner is not further discussed, however we discuss an another variant of SIMPLER, the so called MSIMPLER which is supposed to give better results than SIMPLER.

A.5 MSIMPLER PRECONDITIONER

The MSIMPLER preconditioner is an improved SIMPLER preconditioner presented by Segal et al. in [20]. It is inspired by the similarities between SIMPLE and the commutator preconditioners presented by Elman et al. [23]. For the commutator preconditioners, a more general form of Schur decomposition is given by

$$BF^{-1}B^T \approx (BM_1^{-1}B^T)F_p^{-1} \text{ with,}$$
$$F_p = (BM_2^{-1}B^T)^{-1}(BM_2^{-1}FM_1^{-1}B^T).$$

In the equations (A.2) and (A.3) we took $M_1 = M_2 = diag(Q_v)$. Now, if we were to create a new block factorization preconditioner in which Schur complement is built on SIMPLE's approximate block factorization while being based on a commutator approximation, we get

$$P = LU \begin{bmatrix} I & 0 \\ 0 & F_p^{-1} \end{bmatrix} \tag{A.4}$$

If $S = -BF^{-1}B^T \approx -BD^{-1}B^T = S_a$, $M_1 = D$ and F_p is identity then equation (A.4) corresponds to the SIMPLE preconditioner. Similarly, if the pressure update step (step 4) in the SIMPLE algorithm is solved with $S_a = -(BM_1^{-1}B^T)F_p^{-1}$, then the SIMPLE preconditioner would be equivalent to equation (A.4).
In our case we deal with the time dependent multiphase flows. Hence, while discretizing the time dependent Navier-Stokes equations, we get a $\frac{1}{timestep}$ term on the diagonal of matrix (which is obtained from the implicit time discretization of the momentum equation). The presence of this $\frac{1}{timestep}$ term on the diagonal makes the diagonal entries larger than the off diagonal entries, hence FD^{-1} is close to identity. This also implies that F_p is also close to identity. For the time dependent problems, D is also close to the diagonal of the velocity mass matrix. Hence, if we choose again $M_1 = M_2 = diag(Q_v) = Q_{vd}$ we get

$$F_p = (BQ_{vd}^{-1}B^T)^{-1}(BQ_{vd}^{-1}FQ_{vd}^{-1}B^T).$$

From the above definition it is easy to see that if FQ_{vd}^{-1} is close to unity then F_p is also close to identity, hence the Schur complement becomes

$$BF^{-1}B^T \approx BQ_{vd}^{-1}B^T.$$

Therefore, if we replace D with the diagonal of the mass matrix of velocity Q_{vd} in the SIMPLE/ SIMPLER method we get the MSIMPLER method. The algorithm is presented later.
For time dependent Navier-Stokes problems, the MSIMPLER preconditioner is better than SIMPLER because in SIMPLER we must form the Schur complement after every time step (as it is based on diagonal elements of F which needs to be updated after each time step), while the MSIMPLER preconditioner has the velocity mass matrix in its definition of the Schur complement which does not change per time step, hence we save time in building the Schur complement.

1. Solve $S_a\hat{\boldsymbol{p}} = \boldsymbol{g} - B\boldsymbol{u}^k - BQ_{vd}^{-1}(\boldsymbol{f} - F\boldsymbol{u}^k)$
2. Solve $\boldsymbol{r}_u = \boldsymbol{f} - B^T\hat{\boldsymbol{p}}, \boldsymbol{r}_p = \boldsymbol{g}$
3. Solve $F\hat{\boldsymbol{u}} = \boldsymbol{r}_u$
4. Solve $S_a\delta\boldsymbol{p} = \boldsymbol{r}_p - B\hat{\boldsymbol{u}}$
5. Update $\delta\boldsymbol{p} = \delta\hat{\boldsymbol{p}}$
6. Update $\boldsymbol{u} = \hat{\boldsymbol{u}} - Q_{vd}^{-1}B^T\delta\boldsymbol{p}$
7. Update $\boldsymbol{p} = \hat{\boldsymbol{p}} + \delta\boldsymbol{p}$

Algorithm 6: MSIMPLER preconditioner without pressure update damping

Below we present the GCR algorithm with SIMPLE/SIMPLER type preconditioners, for more details about the algorithm one is referred to [24]. We choose GCR because

it is stable, minimizes the residual norm, and allows for variable preconditioning which is typical of SIMPLE type preconditioners as discussed above.

$\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$
for $k = 0, 1, ..., ngcr$ **do**
  $\boldsymbol{s}_{k+1} = P^{-1}\boldsymbol{r}_k$
  $\boldsymbol{v}_{k+1} = A\boldsymbol{s}_{k+1}$
  for $i = 0, 1, ..., k$ **do**
    $\boldsymbol{v}_{k+1} = \boldsymbol{v}_{k+1} - (\boldsymbol{v}_{k+1}, \boldsymbol{v}_i)\boldsymbol{v}_i$
    $\boldsymbol{s}_{k+1} = \boldsymbol{s}_{k+1} - (\boldsymbol{v}_{k+1}, \boldsymbol{v}_i)\boldsymbol{s}_i$
  end
  $\boldsymbol{v}_{k+1} = \boldsymbol{v}_{k+1}/\|\boldsymbol{v}_{k+1}\|_2$
  $\boldsymbol{s}_{k+1} = \boldsymbol{s}_{k+1}/\|\boldsymbol{s}_{k+1}\|_2$
  $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + (\boldsymbol{r}_k, \boldsymbol{v}_{k+1})\boldsymbol{s}_{k+1}$
  $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k + (\boldsymbol{r}_k, \boldsymbol{v}_{k+1})\boldsymbol{v}_{k+1}$
end

Algorithm 7: GCR - MSIMPLER preconditioner to solve $A\boldsymbol{x} = \boldsymbol{b}$

Where matrix P depends on the type of SIMPLE/SIMPLER preconditioner used, for MSIMPLER it is

$$P = H_r M_r^{-1} - H_r M_r^{-1} A M_l^{-1} H_l + M_l^{-1} H_l \text{ where}$$

$$H_r = \begin{bmatrix} I & 0 & 0 & -Q^{-1}_{vr\ d} G_r \\ 0 & I & 0 & -Q^{-1}_{v\theta\ d} G_\theta \\ 0 & 0 & I & -Q^{-1}_{vz\ d} G_z \\ 0 & 0 & 0 & I \end{bmatrix},\ M_r = \begin{bmatrix} F_r & 0 & 0 & 0 \\ 0 & F_\theta & 0 & 0 \\ 0 & 0 & F_z & 0 \\ G_r^T & G_\theta^T & G_z^T & S_a \end{bmatrix}$$

$$H_l = \begin{bmatrix} I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \\ -G_r Q^{-1}_{vr\ d} & -G_\theta Q^{-1}_{v\theta\ d} & -G_z Q^{-1}_{vz\ d} & I \end{bmatrix},\ M_r = \begin{bmatrix} F_r & 0 & 0 & G_r \\ 0 & F_\theta & 0 & G_\theta \\ 0 & 0 & F_z & G_z \\ 0 & 0 & 0 & S_a \end{bmatrix}$$

and W is the block diagonal of $M_l + M_r - A$.

Bibliography

[1] S. Osher, J.A. Sethian. *Fronts propagating with curvature-dependent speed: algorithms based on HamiltonJacobi formulations.* J. Comput. Phys. 79, (1988) pp. 1249.

[2] D. Gueyffier, J. Li, A. Nadim, S. Scardovelli, S. Zaleski. *Volume of Fluid interface tracking with smoothed surface stress methods for three-dimensional flows.* J. Comput. Phys. 152, (1999) pp. 423456

[3] S. van der Pijl. *Computation of bubbly flows with a Mass-Conserving Level-Set Method.* PhD Thesis, TU-Delft (2005).

[4] Y. Morinishi, O.V. Vasilyev, Takeshi Ogi. *Fully Conservative finite difference scheme in cylindrical coordinates for incompressible flow simulations.* Journal of Computational Physics 197, (2004) pp. 686-710.

[5] A. Arakawa, V.R. Lamb. *Computational design of the basic dynamical processes of the UCLA general circulation model.* Methods of Computational Physics 17, (1977) pp.173265.

[6] T.W.H Sheu, R.K. Lin. *Newton linearization of the incompressible NavierStokes equations.* Int. J. Numer. Meth. Fluids 44, (2004) pp. 297-312.

[7] J. van Kan. *A Second-Order Accurate Pressure-Correction Scheme For Viscous Incompressible Flow.* SIAM J. Sci. Stat. Comput. 7(3), (1986) pp. 870-891.

[8] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Second Edition, SIAM. ISBN 978-0-89871-534-7.

[9] A. J. Wathen. *Preconditioning.* Acta Numerica, 24, (2015) pp. 329-376.

[10] Y. Saad, M.H. Schultz. *GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems.* SIAM J. Sci. Stat. Comput., 7, (1986) pp. 856-869.

[11] Magnus R. Hestenes, Eduard Stiefel. *Methods of Conjugate Gradients for Solving Linear Systems.* Journal of Research of the National Bureau of Standards. 49, (1952) pp. 409436.

[12] P. Sonneveld, M.B. van Gijzen. *IDR(s): A Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Stsrems of Linear Equations.* SIAM J. Sci. Comput., 31 (2), (2008) pp. 1035-1062.

[13] J.A. Meijerink, H.A. van der Vorst. *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix.* Math. Comp., 31, (1977) pp. 148162.

[14] R. Nabben, C. Vuik. *Domain Decomposition Methods and Deflated Krylov Subspace Iterations.* ECCOMAS CFD (2006).

[15] J. Verkaik, C. Vuik, B.D. Paarhuis, A. Twerda. *The Deflation Accelerated Schwarz Method for CFD.* ICCS, (2005) pp 868-875.

[16] J. Frank, C. Vuik, A. Segal. *On The Construction of Deflation-Based Preconditioners.* SIAM J. Sci. Comput., 23 (2), (2001) pp. 442-462.

[17] C. Vuik, J. Frank. *Coarse Grid Acceleration of a Parallel Block Preconditioner.* Future Generation Computer Systems. 17 (2001), pp. 933-940.

[18] T.B. Jonsthovel, M.B. van Gijzen, C. Vuik, A. Scarpas. *On The Use Of Rigid Body Modes In The Deflated Preconditioned Conjugate Gradient Method.* SIAM J. Sci. Comput., 35 (1), (2012) pp. B207-B225.

[19] B. de Jong. *Contact Line Dynamics in Oil Water Simulations.* Internship Report, TNO (2015).

[20] A. Segal, M. ur Rehman, C. Vuik. *Preconditioners for Incompressible Navier-Stokes Solvers.* Numer. Math. Theor. Meth. Appl. 3(3), (2010) pp 245-275.

[21] M. F. Murphy, G. H. Golub, A. J. Wathen. *"A Note on Preconditioning for Indefinite Linear Systems.* SIAM J. Sci. Comput., 21(6), (2000) pp. 19691972.

[22] D. Kay, D. Loghin, A. J. Wathen. *A preconditioner for the steady-state Navier-Stokes equations.* SIAM J. Sci. Comput., 24, (2002) pp. 237256.

[23] H.C. Elman, V.E. Howle, J. Shadid, R. Shutterworth, R. Tumirano. *Block Preconditioner Based on Approximate Commutators.* SIAM J. Sci. Comput., 27 (5) , (2006) pp. 1651-1668.

[24] C. Vuik, A. Saghir, G.P. Boerstoel. *The Krylov Accelerated SIMPLE(R) Method for Flow Problems in Industrial Furnaces.* Int. J. Numer. Meth. Fluids 33, (2000) pp. 1027-1040.