ETH Zürich
and
Delft University of Technology

A Navier Stokes Solver for the GPU

by

Pratik Nayak

supervised by

Prof. Peter Arbenz, Computer Science Department,
ETH Zürich

and

Dr. Rene Pecnik, Process and Energy Lab,
TU Delft

A project report submitted in partial fulfillment for the
Master of Science programme
in
Solid and Fluid Mechanics
at the
Mechanical, Maritime and Materials Engineering Department, TU Delft

December 5th, 2016

“God put me on this earth to accomplish a certain number of things. Right now, I am so far behind, I will never die.”

Calvin from Calvin and Hobbes by Bill Watterson.

“The human brain is finite. The powers of its imagination, infinite.”

Acknowledgements

I would like to thank Prof. Peter Arbenz for the opportunity to work on a project with him and guiding me throughout the project. I would also like to thank Dr. Rene Pecnik for his support. I am indebted to the IDEA League for supporting me for this project and helping me cope with the living costs in Zürich. I would also like to heartily thank Prof. Arbenz for supporting me to attend the CSCS Summer School for High Performance computing, which helped me prepare for the project.

I would like to also thank the Swiss National Supercomputing Centre (CSCS) at Lugano, Switzerland and the Max Planck Insitute at Garching, Germany for providing computing resources. This project would not have been possible without them.

Last but not the least, I would like to thank the Delft Wind Energy Institute (DUWIND) for giving me an opportunity to study at TU Delft by supporting me through their scholarship. I am extremely grateful to them.

Contents

Acknowledgements **ii**

Symbols **v**

1 Introduction **1**
1.1 Motivation 1
1.2 Background 2
1.3 Goals and Objectives 3

2 Methodology and Ideas **4**
2.1 Methodology 4
2.2 Ideas 6
2.2.1 Directional splitting 6
2.2.2 Solution strategies 8
2.2.2.1 Problems with Algorithm 1 10
2.3 Meshing and post-processing 11
2.3.1 Meshing 11
2.3.2 Post-processing 12

3 MPI code and optimizations **13**
3.1 Message Passing Interface and usage 13
3.2 Modules and subroutines 14
3.3 Optimizations 20
3.3.1 Memory Optimizations 20

4 GPU code and optimizations **22**
4.1 Graphical Processing Units and programming for general purpose 22
4.1.1 Programming model 23
4.1.2 Threads and Blocks 23
4.1.3 Memory handling 24
4.1.3.1 Global Memory 24
4.1.3.2 Shared Memory 25
4.1.3.3 Pinned Memory 25
4.1.3.4 Unified Memory 26
4.2 Harnessing the parallelism 26

5 Results and Conclusions **32**

5.1 MPI Results and Observations 32
5.1.1 Observations and explanations 33
5.2 GPU results and observations 39
5.2.1 Observations and explanations 39
5.3 Summary and conclusions 41
5.4 Future work 41

Bibliography **42**

Symbols

Symbol	***Name and Usage***
∇	Gradient operator
$\nabla\cdot$	Divergence operator
Ω	Boundary of the grid
ν	Viscosity of fluid
Re	Reynolds Number
p	Pressure
$\mathbf{u} = (u, v, w)$	Velocity vector
ρ	Density
$\mathbf{u}_\epsilon$	Perturbation of $\mathbf{u}$ around ϵ
ϵ	Perturbation parameter

Chapter 1

Introduction

Graphical Processing Units are massively parallel processors that have the capability to solve problems that have a large amount of parallelism in them, much faster than on a CPU. GPU's have become increasingly popular due to their capability to decrease the computation time compared to the CPU. This project concerns the computation of the solution of the Navier–Stokes equations by using the direction splitting method where the velocity and pressure updates take the form of one dimensional linear systems[1] which can be solved using tri-diagonal solution methods which are highly parallel and hence suitable for GPU's.

1.1 Motivation

Moore's so-called law[1] predicts the approximate doubling of transistors in a chip every 24 months. But as the size of the transistors hits a plateau, this law will no longer hold. The increase in power consumption of the computing core is another reason that the clock speed has hit a plateau. Therefore, computing cores that could work in parallel were invented. The power consumed by each core would be the same, but the total amount of time taken to do a task would be considerably shorter than when done by a single core, depending on the task parallelizability.

GPU's are gaining traction especially in the field of scientific computing where the problems being solved range from bio-informatics to the astrophysics. Though the problem type may differ the idea behind is similar: Parallelizable code run on massively parallel computing cores. The efficiency of the GPU depends on the amount of work given to each of the computing cores. The main aspect that needs to be taken into account is that the algorithm has enough amount of work that can be computed on each of the parallel computing cores.

[1] Moore's law, http://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html

The Navier–Stokes equations are the governing equations for most of the fluid flows. They govern the flow of fluid in a pipe to the prediction of weather patterns. The fact that the existence and uniqueness of the solutions to these equations is undetermined[2] does not provide us with an exact solution to work with. The non-linearities in the equation make the computation of the solution difficult even when the problem is well defined, due to the chaotic nature and bifurcations. As fluid flow governs most of the engineering applications and nature as well, it becomes very important that we construct methods that are able to approximate the exact solution with least amount of computational power and time as possible. As the Navier–Stokes equations are non-linear, partial differential equations that have an inherent pressure-velocity coupling, only certain algorithms that are specifically designed to have enough parallelism can be run with high efficiency on GPU's.

1.2 Background

Projection methods have been very popular since their introduction by Chorin[2]. The basic idea was to solve for pressure by solving the Poisson equation with appropriate boundary conditions. This pressure Poisson equation, derived from the decomposition of the velocity field is the main idea that is behind most of the pressure/velocity decoupling methods. The algorithm that is used in this project comes from the method proposed by Guermond and Minev[3]. They propose a fractional time stepping technique that applies the direction splitting technique to both the momentum equation and the pressure correction equation. The pressure correction equation is derived from the single perturbation of the mass conservation equation by penalizing the incompressibility constraint in a negative norm induced by the splitting[3]. This splitting produces one dimensional linear systems where the system matrices are tridiagonal. These linear systems can be very efficiently solved on parallel processors.

To get a higher order variant of the direction splitting method, an incremental scheme, the rotational version as explained in [4] is used.

[2] Millenium Problem, http://www.claymath.org/millennium-problems/navier-stokes-equation

1.3 Goals and Objectives

The objectives of the project were the following:

1. Identify the bottlenecks and problems, if any, in the existing MPI code.
2. Optimize the solver module of the existing MPI code.
3. Identify strategies to port the solver module and the time loop to multiple GPU's.
4. Optimize the GPU version for memory and locality of data.
5. Compare the speedup obtained with GPU's to the plain MPI version and the serial version.

Chapter 2

Methodology and Ideas

The methodology of this project was constructed based on the objectives and the tools used were also based on them. As the main objective of the project was to port the MPI code to run on multiple GPU's, analyzing, profiling and optimizing of the MPI code was an essential part of the project. Profiling was necessary to identify the bottlenecks in the code, so that if possible, the bottlenecks could be removed in the GPU version.

The ideas that are involved in the code are also explained in the following section.

2.1 Methodology

The flowchart explaining the steps involved during the process are shown in Fig 2.1.

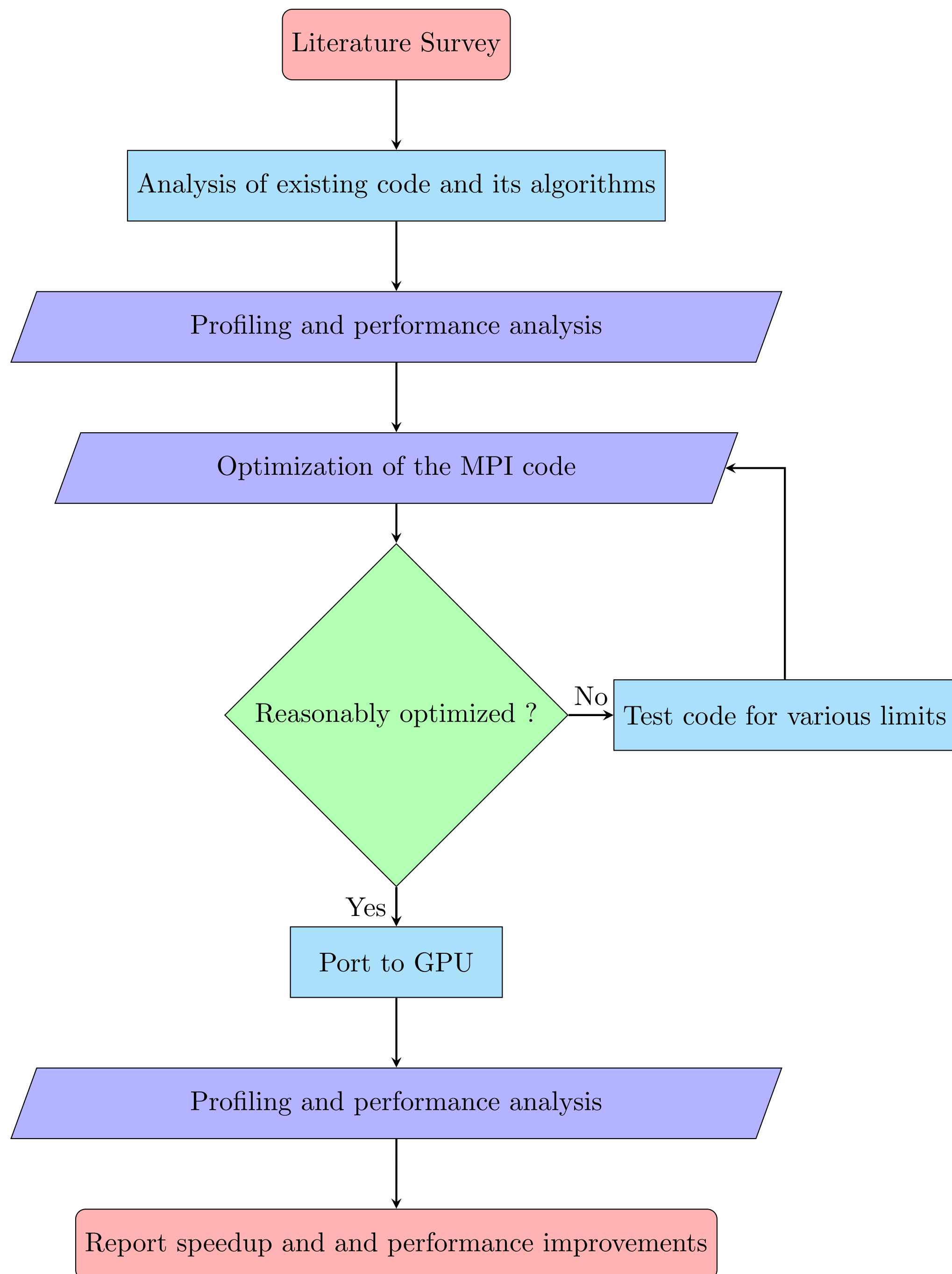


FIGURE 2.1: Methodology Flow chart

A summary of the tools used for the different steps for the project is given below:

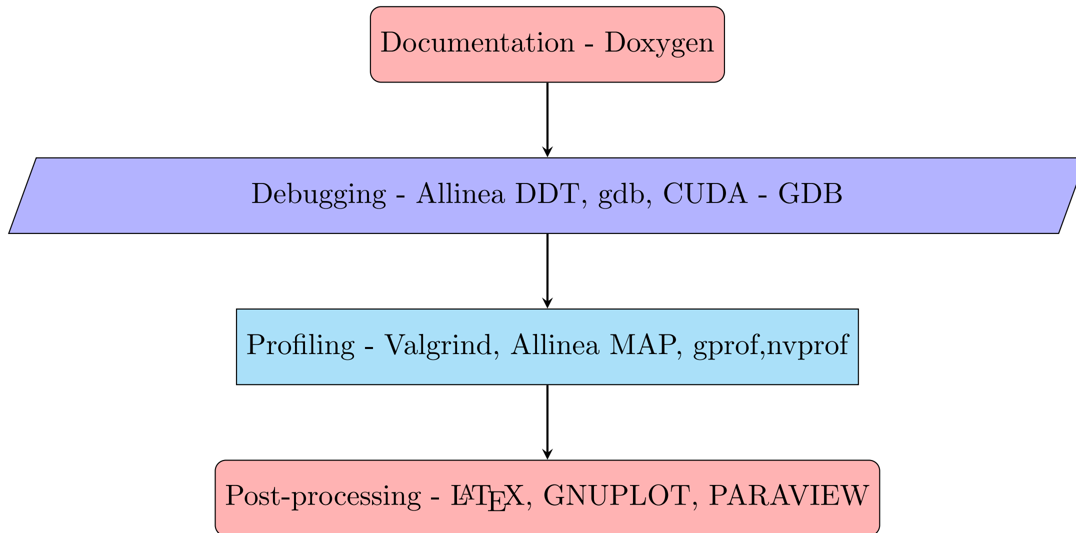


FIGURE 2.2: Summary of tools used

2.2 Ideas

The code by Minev [3] has been written to run on multiple processes thereby reducing the time required to compute the solution. The main idea involves solving the pressure correction equation by splitting it into one-dimensional linear systems that involve tridiagonal matrices. This fractional time stepping is not only applied to the pressure correction equation but also to the momentum equation, thereby reducing the computational complexity of the method.

2.2.1 Directional splitting

As the main idea is to uncouple the velocity and the pressure, the first step is to solve an alternative form of the Navier–Stokes equation that is a $\mathcal{O}(\Delta t^2)$ perturbation of the linearized version of the Navier–Stokes equations.

The Navier–Stokes equations are given by

$$\left.\begin{aligned} &\partial_t \mathbf{u} + (\mathbf{u}\cdot\nabla\mathbf{u}) - \frac{1}{Re}\Delta\mathbf{u} + \nabla p = \mathbf{f}, \quad \text{in} \quad \Omega\times(0,T] \\ &\nabla\cdot\mathbf{u}, \quad \text{in} \quad \Omega\times[0,T] \\ &\mathbf{u}|_{\partial\Omega} = \mathbf{a}, \quad \text{in} \quad \partial\Omega\times(0,T] \\ &\mathbf{u}|_{t=0} = \mathbf{u_0}, \quad \text{in} \quad \Omega \end{aligned}\right\} \tag{2.1}$$

where $\mathbf{f}$ is the source term, $\mathbf{a}$ is the boundary term and $\mathbf{u_0}$ is the initial condition, Δt is the time step and $\Omega = [0, L_x] \times [0, L_y] \times [0, L_z]$ which is a subset of $\mathbb{R}^3$.

Most of the incremental pressure correction schemes are semi-discrete versions of the single perturbation of the linearized version of the above Navier–Stokes equation and can be written as:

$$\left.\begin{aligned}
&\partial_t \mathbf{u}_\epsilon - \frac{1}{Re}\Delta \mathbf{u}_\epsilon + \nabla p_\epsilon = \mathbf{f}, \quad \text{in} \quad \Omega \times (0,T], \;\; \mathbf{u}_\epsilon|_{\partial\Omega\times(0,T]} = \mathbf{a}, \;\; \mathbf{u}_\epsilon|_{t=0} = 0 \\
&-\Delta t \Delta \phi_\epsilon + \nabla \cdot \mathbf{u}_\epsilon, \quad \text{in} \quad \Omega \times [0,T], \quad \partial_n \phi_\epsilon|_{\partial\Omega\times(0,T]} = 0 \\
&\Delta t \partial_t p_\epsilon = \phi_\epsilon - \frac{\chi}{Re}\nabla \cdot \mathbf{u}_\epsilon, \quad p_\epsilon|_{t=0} = p_0
\end{aligned}\right\} \tag{2.2}$$

where $p_0 = p|_{t=0}$, the value of the pressure at the initial condition, Δt is the perturbation parameter, that is $\epsilon = \Delta t$ and $\chi \in (0,1]$ is an adjustable parameter. This perturbation has shown to be stable in [5] and in [6]. Consequently, u_ϵ has shown to be a $\mathcal{O}(\Delta t^2)$ perturbation in the L^2 - norm and a $\mathcal{O}(\Delta t^{3/2})$ perturbation in the Hilbert, H^1- norm for all $0 < \chi \leq 1$.

The main idea of Guermond and Minev[3, 7] was to replace the Poisson operator in the pressure correction equation by an operator A such that the operator and the domain of the operator $D(A)$ satisfy the bilinear form, $a(p,q) := \int_\Omega qApdx$ properties of symmetricity and the L^2 norm of the element of the domain, $||\nabla q||^2_{L^2} \leq a(q,q)$. These properties of the operator A is necessary to ensure the regularity of pressure and to make sure that the solution is smooth enough in the limit.

The next issue is the choice for the operator, A. Many operators exist that satisfy the above properties. For example, the usual pressure correction equation can be recovered by taking the operator $A := -\Delta_N$, which is the Laplace operator with the homogeneous Neumann boundary condition applied. The operator A can also be taken to be $I - \Delta_N$, where I is the identity operator. Guermond and Minev suggest to use the operator $A := (1-\partial_{xx})(1-\partial_{yy})(1-\partial_{zz})$ with appropriate boundary conditions.

Now, the pressure correction equation is computed with the above operator and the momentum equation is solved by using the direction splitting method of Douglas [1], which is a second order accurate method.

The complete algorithm consists of 4 steps. First the *pressure predictor*, where the corrected pressure is obtained using the perturbation pressure, $\phi \quad \forall n \geq 0$ as :

$$p^{*,n+\frac{1}{2}} = p^{n-\frac{1}{2}} + \phi^{n-\frac{1}{2}} \tag{2.3}$$

and setting $p^{-\frac{1}{2}} = p_0$ and $\phi^{-\frac{1}{2}} = 0$. The second step involves the *velocity update* which involves the direction splitting for the momentum equations and thereby reducing them to one-dimensional equations.

$$\begin{aligned}
&\frac{\xi^{n+1}-\mathbf{u}^n}{\Delta t} - \frac{1}{Re}\Delta\mathbf{u}^n + \nabla p^{*,n+\frac{1}{2}} + \frac{3}{2}(\mathbf{u}^n\cdot\nabla)\mathbf{u}^n - \frac{1}{2}(\mathbf{u}^{n-1}\cdot\nabla)\mathbf{u}^{n-1} = 0 \ ,\xi^{n+1}|_{\partial\Omega} = \mathbf{a} \\
&\frac{\eta^{n+1}-\xi^{n+1}}{\Delta t} - \frac{1}{2Re}\partial_{xx}(\eta^{n+1}-\mathbf{u}^n) = 0, \quad \eta^{n+1}|_{x=0,L_x} = \mathbf{a} \\
&\frac{\zeta^{n+1}-\eta^{n+1}}{\Delta t} - \frac{1}{2Re}\partial_{yy}(\zeta^{n+1}-\mathbf{u}^n) = 0, \quad \zeta^{n+1}|_{y=0,L_y} = \mathbf{a} \\
&\frac{\mathbf{u}^{n+1}-\zeta^{n+1}}{\Delta t} - \frac{1}{2Re}\partial_{zz}(\mathbf{u}^{n+1}-\mathbf{u}^n) = 0, \quad \mathbf{u}^{n+1}|_{z=0,L_z} = \mathbf{a}
\end{aligned} \tag{2.4}$$

The third step involves the application of the operator A to the pressure correction equation and thereby obtaining one-dimensional equations. This is called the *penalty step.*

$$\begin{aligned}
&\psi - \partial_{xx}\psi = -\frac{1}{\Delta t}\nabla\cdot\mathbf{u}^{n+1}, \ \ \partial\psi|_{x=0,L_x} = 0 \\
&\varphi - \partial_{yy}\varphi = \psi, \quad \partial_y\varphi|_{y=0,L_y} = 0 \\
&\phi^{n+\frac{1}{2}} - \partial_{zz}\phi^{n+\frac{1}{2}} = \varphi, \quad \partial_z\phi^{n+\frac{1}{2}}|_{z=0,L_z} = 0
\end{aligned} \tag{2.5}$$

The last step consists of updating the pressure with the pressure increment, $\phi^{n+\frac{1}{2}}$.

$$p^{n+\frac{1}{2}} = p^{n-\frac{1}{2}} + \phi^{n+\frac{1}{2}} - \frac{\chi}{Re}\nabla\cdot\left(\frac{1}{2}(\mathbf{u}^{n+1}+\mathbf{u}^n)\right) \tag{2.6}$$

2.2.2 Solution strategies

The discretization of the equations (2.3) - (2.6) has been done using the Marker and Cell stencils using central differences for the first and second order derivatives. The parallelization has been done in MPI with the natural cartesian domain decomposition. The domain decomposition induces unknowns additional to the internal unknowns. Therefore, the Schur complement method is used to solve for all the unknowns, which take the form of tri-diagonal linear systems after the one-dimensional linear problems in equation (2.4) and equation (2.5) are approximated in space. It has to be noted here that the linear systems are identical thereby allowing for a simpler solution strategy.

The algorithm allows for a lot of parallelism because each one dimensional system can be solved independently. This allows us to create a cartesian decomposition of the whole mesh. MPI allows for a logical cartesian decomposition of the processes, thereby allowing for each logical process to solve for one cartesian block of unknowns, which contains its own internal unknowns, u_i and the unknowns at its interface, that are a result of the cartesian domain decomposition, u_e. Therefore a large number of systems of the form

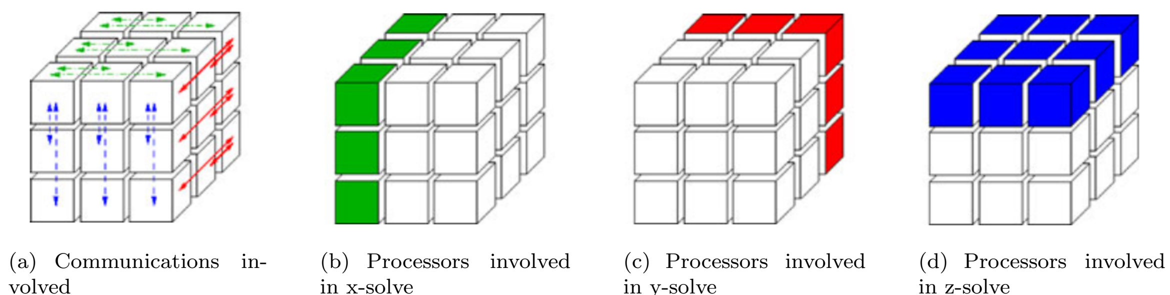
$$\begin{pmatrix} A_{ii} & A_{ie} \\ A_{ei} & A_{ee} \end{pmatrix} \begin{pmatrix} u_i \\ u_e \end{pmatrix} = \begin{pmatrix} f_i \\ f_e \end{pmatrix} \tag{2.7}$$

have to be solved in each time step in each of the space directions. If we consider the number of processes in the direction d as $proc_D$, then the number of interfaces would be equal to $n_e = proc_D + 1$. Note that the interval end points are added to the interface points. Let the number of grid points in direction d be equal to n. Then the matrix A_{ii} are of size $(n - n_e) \times (n - n_e)$ and the matrix A_{ee} are of size $n_e \times n_e$.

The two algorithms Algorithm 1 and Algorithm 2 are the two variants that are used to solve for the unknowns. Algorithm 1 pre-computes the Schur complement on the *master process* for the particular direction and sub-communicator (created through the `MPI_Cart_create`). The local systems are solved, communicated to the *master process* and the Schur complement is solved for the interface unknowns, u_e. The interface unknowns are communicated back to all the respective processes and the internal unknowns, u_i are solved by all processes in parallel.

Algorithm 1 `Schur_solve`: Type 1

1: **if** (*process* == *master_proc*) **then**
2: Pre-compute Schur complement, $S = A_{ee} - A_{ei}A_{ii}^{-1}A_{ie}$
3: **End If**
4: **procedure** SCHUR SOLVE(FOR D = 1,2,3) $\begin{pmatrix} A_{ii} & A_{ie} \\ A_{ei} & A_{ee} \end{pmatrix} \begin{pmatrix} u_i \\ u_e \end{pmatrix} = \begin{pmatrix} f_i \\ f_e \end{pmatrix}$
5: Solve local tri-diagonal systems, $A_{ii}x_i = f_i$ on all processes
6: **if** (*process* == *master_proc*) **then**
7: Gather $A_{ei}x_i$ from all processes in the direction d and current sub-communicator
8: Solve the tri-diagonal linear system $Su_e = f_e - A_{ei}A_{ii}^{-1}f_i$
9: Scatter $A_{ie}u_e$ to all processes in the direction d and current sub-communicator
10: **End If**
11: Solve the tri-diagonal linear system $A_{ii}u_i = f_i - A_{ie}u_e$ on all processes



(a) Communications involved (b) Processors involved in x-solve (c) Processors involved in y-solve (d) Processors involved in z-solve

FIGURE 2.3: Communications required and the process topology involved [7] for Algo 1.

2.2.2.1 Problems with Algorithm 1

1. The schur complement is solved only on the master processes of the sub-communicators. This creates a load imbalance as during this step, all the other processes are idle.

2. The messages to and from the master processes during this step are lengthy and therefore inefficient. Communication takes up a lot of time during this step.

Algorithm 2 resolves the above issues by solving systems in slices or blocks. Here each slice has been made equal to the number of processes to facilitate the Gather operation to be inside the same loop.

Algorithm 2 `Schur_solve`: Type 2

```
1:  Pre-compute Schur complement, S = A_ee − A_ei A_ii^{-1} A_ie on all processes
2:
3:  procedure SCHUR SOLVE(FOR D = 1,2,3)(A_ii A_ie; A_ei A_ee)(u_i; u_e) = (f_i; f_e)
4:      for iter = 1 to proc_D do
5:          for k = 1 to n/proc_D, in steps of proc_D do
6:              Solve proc_D local tri-diagonal systems, A_ii x_i = f_i,
7:              Gather A_ei x_i from all processes on process iter,
8:          End For
9:      End For
10:     Solve the tri-diagonal linear system S u_e = f_e − A_ei A_ii^{-1} f_i on all processes in parallel
11:     for iter = 1 to proc_D do
12:         for k = 1 to n/proc_D, in steps of proc_D do
13:             Scatter A_ie u_e to process iter,
14:         End For
15:     End For
16:     Solve the tri-diagonal linear systems A_ii u_i = f_i − A_ie u_e on all processes
```

The algorithm proceeds by pre-computing the Schur complements on all processes. All the processes then solve for the local unknowns in slices and the respective slices are gathered in the required processes in a loop. This solving of the unknowns in slices reduces the message lengths that are sent because the gather is called in the loop as well. The interface unknowns are then solved for in parallel by each of the processes. The interface unknowns,u_e, are then scattered to the respective processes in slices. The internal unknowns, u_i, are solved for in each of the processes in parallel.

The algorithm has the advantage of solving for the interface unknowns in parallel whereas Algorithm 1 solves for the interface unknowns only in the *master process*. In Algorithm 2, the communication message volume remains the same but the length of the messages reduce. This can be very helpful if the available bandwidth is lower. Also, the load is more balanced in Algorithm 2 because all the processes solve for the interface unknowns rather than just the master processes. Therefore for large cases, Algorithm 2 proves to be faster than its predecessor. The performance comparisons are done in Chapter 5.

The communications required for Algorithm 1 are shown in Fig 2.3(a). Figs 2.3(b),2.3(c),2.3(d) show the processes that are involved in the Schur complement solves in Algorithm 1. For Algorithm 2, all the processes are involved in the Schur complement solves. Another aspect that should be observed is the logical process topology involved for a 3D problem and the fact that the logical process decomposition mimics the physical problem decomposition.

If we denote N by the total number of grid points and the total number of processes to be P and that the number of processes in each direction are equal, then in each row of processes there are $n^2 = (N/P)^{2/3}$ linear systems of size $N^{1/3}$ to be solved, which are solved using the tridiagonal solver similar to Thomas algorithm which is shown in Algorithm 3.

Algorithm 3 `tridiag_solver(A,x,d)`: Thomas algorithm

1: **procedure** SOLVE $Ax = d,\ A = \text{TRIDIAG}(a_i, b_i, c_i)\ ,x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix},\ d = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix}$

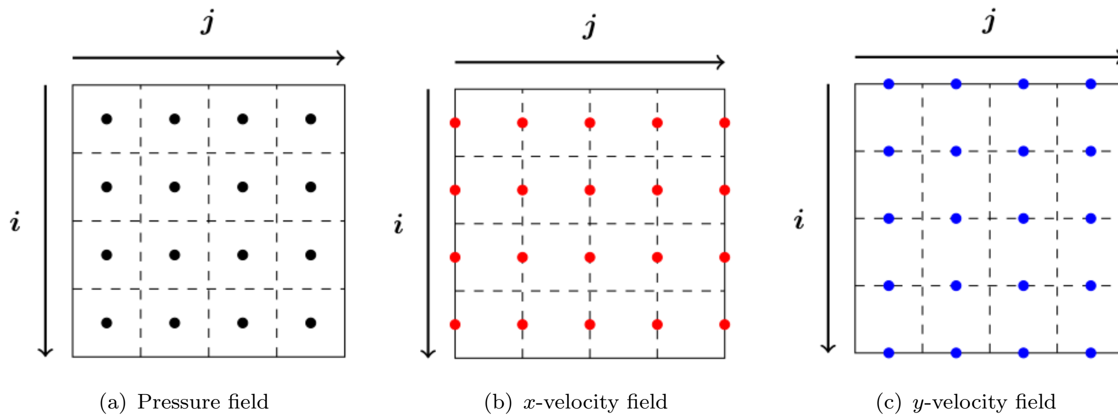
```
2:      cp(1) = c(1)/b(1)
3:      dp(1) = d(1)/b(1)
4:  Forward sweep:
5:      for i = 2 to sizeof(diag(A)) − 1 do
6:          id = (b(i) - cp(i-1) * a(i))
7:          cp(i) = c(i)/ id
8:          dp(i) = (d(i) - dp(i-1) * a(i))/id
9:      dp(n) = (d(n) - dp(n-1) * a(n))/ (b(n) - cp(n-1) * a(n))
10:     x(n) = dp(n)
11: Back Substitution:
12:     for i = sizeof(diag(A)) − 1 to 1 do
13:         x(i) = dp(i) - cp(i) * x(i + 1)
```

2.3 Meshing and post-processing

2.3.1 Meshing

The code by Guermond and Minev allows for two types of meshing: A uniform mesh, where all the grid spacings are equal or a non-uniform mesh, where the grid is refined near the boundaries, the region where the changes in the field gradients are higher. As the test case in consideration is the Lid-driven cavity, the meshing strategy of the non-uniform grid was to refine the grid near the $x-z$ and $y-z$ walls with the z wall being the moving boundary(lid).

The Marker and Cell stencil has been used. Therefore, the velocity and pressure field are present as in Figure 2.4. The velocity fields are present on the cell faces, whereas the pressure field is on the cell center. This prevents checker-boarding and hence is a very effective method



(a) Pressure field (b) x-velocity field (c) y-velocity field

FIGURE 2.4: Marker and Cell Stencil.

so that the central difference method can be used along with a staggered grid to satisfy the div-stability condition. But a disadvantage of the method includes that two separate grid coordinates need to be calculated or stored along with the mesh widths for the case of the non-uniform mesh.

2.3.2 Post-processing

The output of the fields(u, v, w, p, ρ) are written in `*.vtk` format. As the code is a parallel code, each process writes its own fields to separate files. These files have to be stitched together to obtain the final overall field as required. As writing the file to the `*.vtk` format is arbitrary, the files can be written in any format as required depending on the post-processing software.

Depending on the size of the problem, the size of the files can get very large. Therefore, one must be very careful while visualizing or saving the fields required, as the file writing can considerably increase the run-time of the code. Parallel visualization software such as VisIt might prove to be useful for very large cases when multiple time steps and all the fields are required.

Chapter 3

MPI code and optimizations

The MPI code by Guermond and Minev was written to run on a large number of processes and tested for upto 1024 processes for the case of the Lid-driven cavity. This chapter explains the MPI code and the optimizations that are possible to make the code faster. Memory and cache will be some of the important considerations. The results of the optimizations and the performance improvements observed, if any, will be discussed in Chapter 5.

3.1 Message Passing Interface and usage

This section explains some of the basics of the Message Passing Interface and how it has been made use of in the code so that a general idea is gained.

The Message Passing Interface is a standard that has been defined by the MPI Forum and is used to define the syntax and semantics of the MPI library routines that is used to give the user explicit control over the communication between different processes sharing a job. There are many implementations of the MPI standard such as OpenMPI, MPICH, MVAPICH which are open source and there are others which are proprietary such as Intel MPI or IBM PE. Each of the implementations has its own advantages and disadvantages. For purposes of this project, for the basic MPI code atleast, all MPI implementations prove to be same because the MPI routines used are most basic. Only when we get into the GPU, we realise which MPI implementations are more favourable as will be explained in Chapter 4.

The main usage of MPI in the code has been the creation of a logical process grid and passing data between these processes when required. Therefore, the main MPI routines[1] that have been used are shown in Table 3.1.

[1] OpenMPI Documentation, https://www.open-mpi.org/doc/v1.8/

MPI routines used		
Name	*Brief explanation*	*Output*
`MPI_Init(err)`	Initializes the MPI environment	`err`
`MPI_Comm_Size(MPI_Comm,size,err)`	Returns the size of the communicator, `MPI_Comm`	`size` , `err`
`MPI_Comm_Rank(MPI_Comm,rank,err)`	Returns the rank of the calling process from the `MPI_Comm` group	`rank`, `err`
`MPI_Cart_Create(old_comm,ndims,dims, period,reorder,new_comm,err)`	A new communicator with the cartesian topology is created	`new_comm`, `err`
`MPI_Cart_Coords(old_Comm,rank, maxdims,coords,err)`	Stores the coordinates of the process given the rank in the communicator	`coords`, `err`
`MPI_Cart_Sub(old_comm,remain_dims, new_comm,err)`	Creates new groups from `old_comm` which form lower dimensional groups	`new_comm`, `err`
`MPI_Bcast(buff,count,type,root, MPI_Comm,req,err)`	Broadcasts `buff` from `root` to all processes in `MPI_Comm`	`err`
`MPI_Gather(sbuff,scount,stype,rbuff, rcount,rtype,root,MPI_Comm,err)`	Gathers `sbuff` from all processes in `MPI_Comm` to `rbuff` on `root`	`rbuff`,`err`
`MPI_Scatter(sbuff,scount,stype,rbuff ,rcount,rtype,root,MPI_Comm,err)`	Scatters `sbuff` from `root` to `rbuff` on all processes in `MPI_Comm`	`rbuff` ,`err`
`MPI_Finalize(err)`	Ends the MPI environment	`err`

TABLE 3.1: MPI routines used in the code.

As explained above, `MPI_Cart_Create` creates a cartesian topology of processes as shown in Figure 2.3(a). The `MPI_Cart_Sub` then creates a communicator in each physical direction which are basically lines of processes in the direction perpendicular to the direction that is not being considered.

3.2 Modules and subroutines

The list of the modules with the important subroutines are listed below. The call graph shown in Figure 3.1 shows the calling of the different subroutines and their dependencies.

1. `PROGRAM nst`: This is the main program which calls all the other modules. Input is given by means of a data file which is read by the `read_my_data` subroutine from the `data` module.

2. `MODULE data`: This module contains the subroutine `read_my_data` which reads the input settings for the solver from the text data file, parses it and stores it in the respective `structs`. All the reading is done by the master process i.e the process with rank 0 from the communicator, `MPI_Comm_World`. Therefore, after reading and storing the variables in the master process, it uses MPI to broadcast the variables to the different processes involved as well. Therefore after control is returned from this subroutine, all processes have the required data and variables from the input text file.

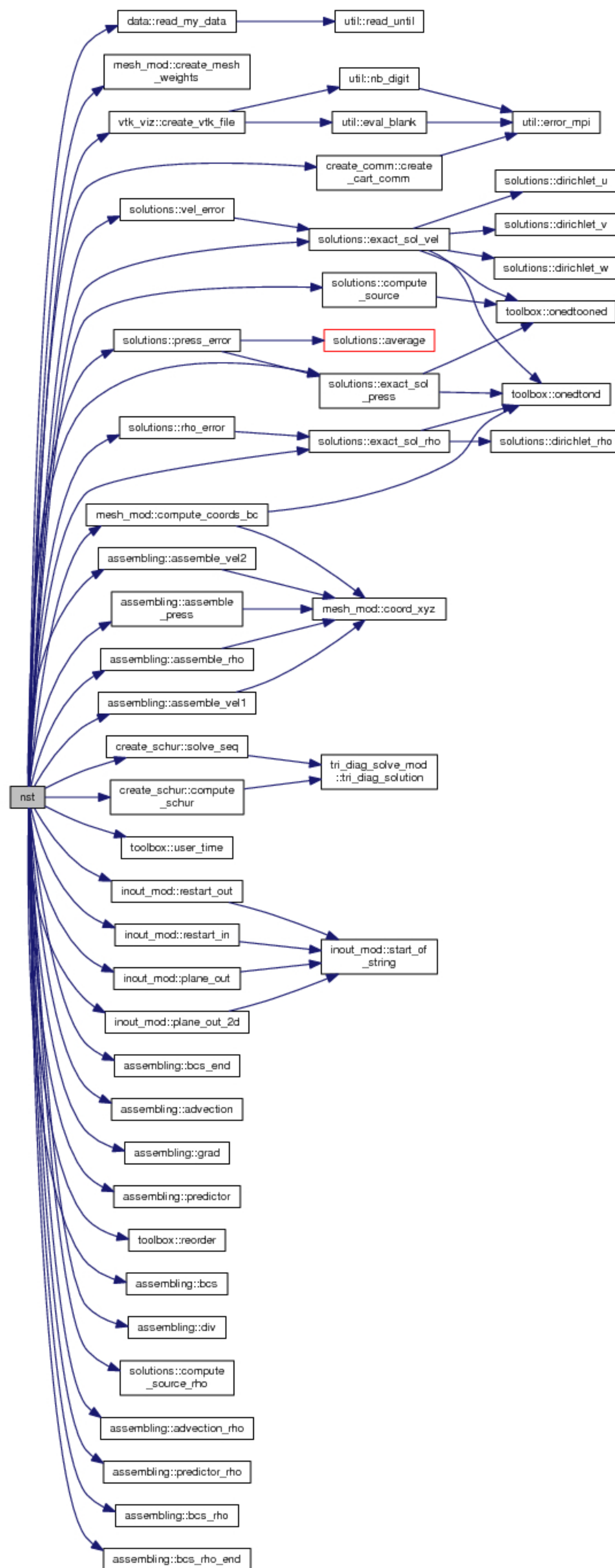


FIGURE 3.1: Call graph for the program with the all the called subroutines and modules.

3. `MODULE mesh_mod`: This module creates the mesh and the coordinates in the following subroutines:
   (a) `create_mesh_weights`: This subroutine creates the mesh weights which is the offset for calculaton of grid coordinates and stored in the `mesh` struct, specifically in `mesh%weights` which is an array of size `dim` × `dim`.
   (b) `compute_coords_bc`: This subroutine computes the coordinates of the grid points and also the spatial step sizes of the regular and the staggered grids and stored in `mesh%h` and `mesh%s`, respectively which are user-defined types containing the spatial step sizes for each `dim`.
4. `MODULE create_comm`: This module contains the subroutine `create_cart_comm`, which creates the logical process grid, the lower dimensional cartesian sub-groups through which the processes can communicate. The variable `comm_one_d` contains the cartesian sub-groups, and the variable `comm_cart` contains the cartesian communicators. The subroutine also calculates the coordinates of the process (using `MPI_Cart_Coords(...)`) in a given group based on its rank in the group and that is stored in the variable `coord_cart`.
5. `MODULE assembling`: This module contains subroutines that assembles the tridiagonal matrices and some subroutines that perform operations such as the divergence and the gradient on the required fields.
   (a) `assemble_vel1`: This subroutine assembles the velocity matrix for the velocity field when the derivative is in the direction of the field itself. Hence the subroutine uses the regular grid spatial step, `mesh%h`. The Reynolds number Re and the viscosity ν are also taken into consideration. The velocity matrix, `mat_vel` is an array of size `dim` × `dim`, where the `assemble_vel1` updates the `mat_vel(i,i)` entries for `i = 1 to dim`. Each of the entries of the `mat_vel` is a block matrix of the form of the matrix in Equation (2.7). The A_{ii} matrix contains the matrix entries referring to the internal grid points and is a tri-diagonal matrix, the matrix A_{ee} contains the entries corresponding to the unknowns on the interfaces of the processes which are present due to the domain decomposition.
   (b) `assemble_vel2`: This subroutine updates the matrices of `mat_vel(i,j)` when `i` ≠ `j` and works on the staggered grid as required by the MAC stencil and other aspects are similar to that of the previous subroutine. The Equation (2.4) shows the linear systems of the matrices that are being assembled in the subroutines, `assemble_vel1` and `assemble_vel2`.
   (c) `assemble_press`: This subroutine assembles the pressure matrix and makes use of both the staggered and the regular spatial step size. The Equation (2.5) shows the linear systems of the matrices that are being assembled in the subroutines, `assemble_press`.

(d) `assemble_rho`: This subroutine assembles the density matrix, which is also a block matrix as before.

(e) `bcs`: This subroutine is used inside the time stepping loop to update the boundary conditions for the linear systems as shown in Equation (2.4) and Equation (2.5) after solving one step and reordering the fields as required.

(f) `grad`: This subroutine effectively computes the gradient of the pressure field in the `mesh%current_dir` direction along with some correction to the boundary terms. It effectively outputs the `rhs` as required by the first equation in Equation (2.4) along with the subroutine `div`.

(g) `div`: This subroutine computes the divergence of the velocity field as shown in Equation (2.4).

(h) `advection`: This subroutine is called only if the non-linear part of the NS equations has been set to be computed in the input data settings. It computes the non-linear advection part of Equation (2.4). Some part of the required boundary condition is added in this subroutine and the other part is added in the Schur complement computation subroutine.

(i) `predictor`: This subroutine effectively computes the Laplacian of the velocity field by first computing the diffusion operator and then computing the Laplacian as a series of matrix vector products.

6. `MODULE solutions`: This module contains some subroutines that compute the exact solution as a function of final time to compute the errors in the solution. Similar to the `exact_sol_vel` subroutine as explained there are also subroutines that compute the pressure exact solution (`exact_sol_press`) and for the density(`exact_sol_rho`).

   (a) `compute_source`: This subroutine computes the source term in the Equation (2.4). It adds the source term part to the `rhs` only if the source term has been set in the input setting in the data file.

   (b) `exact_sol_vel`: This subroutine calculates the coordinates as required by using the subroutines in the `toolbox` module and updates the output vector with the exact solution at the final time. This subroutine is also used to compute the initial solution if the restart has not been set or is not present.

7. `MODULE create_Schur`: This module contains subroutines that compute for the Schur complement and also solve for it.

   (a) `compute_Schur`: This subroutine computes the Schur complement matrix as shown in Algorithm 1 given the main matrix. This Schur complement matrix is computed by all processes. As an inverse is required, this is computed using the `tri_diag_solution` subroutine of the `tri_diag_solve_mod` module. The Schur complement matrix is a tri-diagonal matrix.

(b) `solve_schur1`: This subroutine is the Schur complement solver that solves for the unknowns using Algorithm 1. The Schur complement part of the solution is done only on the master process of the particular subgroup of the process grid. The steps involved are explained in Algorithm 1.

(c) `solve_schur2`: This subroutine is the Schur complement solver that solves for the unknowns using Algorithm 2. The Schur complement part of the solution is done on all processes of the process grid. The steps involved are explained in Algorithm 2.

8. `MODULE toolbox`: This module contains some useful subroutines such as reordering, transposing and other similar routines that are used in most of the other modules.

   (a) `OneDtoOneD`: This subroutine reorders the array as required by the next step and the direction that is input to it and outputs the offset that the array has to be operated on.

   (b) `OneDtoND`: This subroutine outputs the offsets in all three dimensions based on the current and the next directions.

   (c) `reorder`: This subroutine is basically a transposition subroutine. In 2D, it effectively transposes the 1D array given to it that has been unrolled. In 3D, it does the transpose for all the planes perpendicular to the solution direction.

9. `MODULE inout_mod`: This module contains the subroutines that are used to read back (`restart_in`) the restart files and also write (`restart_out`) the restart files. It also contains the (`plane_out`) subroutine that writes the solution of the required points to check the solution for accuracy and stability.

10. `vtk_viz`: This subroutine contains the `create_vtk_file` subroutine that writes the `*.vtk` format file of the scalar and the vector fields (u, v, w, p, ρ) that can be read by Paraview for visualization.

The complete algorithm the code executes is

Algorithm 4 `PROGRAM nst`, The Navier–Stokes solver using MPI

```
 1: procedure CREATE MESH AND DOMAIN DECOMPOSITION
 2:     Read the mesh details from input file: CALL read_my_data(mesh)
 3:     Create Cartesian decomposition of the process grid: CALL create_comm(...)
 4: procedure ASSEMBLE MATRICES
 5:     for i = 1 to dim do
 6:         Assemble pressure matrix, CALL assemble_press(...)
 7:         Assemble density matrix, CALL assemble_rho(...)
 8:         Compute Schur complement of pressure and density matrices, CALL
    compute_Schur(...)
 9:         Assemble velocity matrices, CALL assemble_vel1(...) and CALL
    assemble_vel2(...)
10:         Compute Schur complement of velocity matrices, CALL compute_Schur(...)
11: procedure ALLOCATE MEMORY AND SET INITIAL SOLUTION
12:     Allocate memory for pressure, velocity and density fields
13:     if restart_in==yes then
14:         Load the fields from restart file, CALL restart_in(...)
15:     Compute the exact solution at time, t = 0 and update the fields and rhs.
16: procedure TIME STEPPING
17:     while t<tmax do
18:         procedure SOLVE FOR VELOCITY FIELDS
19:             for j = 1 to dim do
20:                 if source==yes then
21:                     Update the rhs with the source data, CALL compute_source(...)
22:                 Compute gradient, CALL grad(...)
23:                 if advection==yes then
24:                     Update the rhs with the advection data, CALL advection(...)
25:                 Update the rhs with CALL predictor(...)
26:                 for i = 1 to dim do
27:                     Transpose the rhs, to solve the linear system, CALL reorder(...)
28:                     Add the boundary conditions, CALL bcs(...)
29:                     Solve the linear system for (dir,dir) = (i,j), solve_schur2(...)
30:             Recover the solution and update the fields for next time step computations
    and also apply boundary conditions, CALL bcs_end(...)
31:         procedure SOLVE FOR PRESSURE FIELD
32:             Compute divergence on the velocity fields, CALL div(...)
33:             for i = 1 to dim do
34:                 Solve the linear system, solve_schur2(...)
35:                 Transpose the rhs, for the next direction, CALL reorder(...)
36:             Update the pressure fields for next time step and the corrected pressure as
    well.
```

```
37:         if rho==yes then
38:             procedure Solve for density field
39:                 if source_rho==yes then
40:                     Update the rhs with the source data, CALL compute_source_rho(...)
41:                 if advection_rho==yes then
42:                     Update the rhs with the advection data, CALL advection_rho(...)
43:                 Update the rhs with CALL predictor_rho(...)
44:                 for i = 1 to dim do
45:                     Add the boundary conditions, CALL bcs_rho(...)
46:                     Solve the linear system, solve_schur2(...)
47:                     Transpose the rhs, to solve the linear system, CALL reorder(...)
48:                 Recover the solution and update the fields for next time step computations
    and also apply boundary conditions, CALL bcs_rho_end(...)
49:         Store results (field data at a certain time) if needed for post-processing, CALL
    plane_out(...)
50: End of time loop
51: procedure Store results to vtk file and terminate program
52:     Write the required *.vtk files, CALL create_vtk_file(...)
53:     if restart_out==yes then
54:         Write the fields to a restart file, CALL restart_out(...)
55:     Terminate Program
```

3.3 Optimizations

This section explains the possible optimizations. The results are presented in Chapter 5.

3.3.1 Memory Optimizations

Memory optimizations include optimal use of the cache, reducing the communication and last but not least taking care of the memory access while operating on the arrays. Memory optimizations become important when the problem size becomes large. The length of the arrays may go up to a few billion in each of the processes. Sending these arrays, transposing them and performing operations on them can very easily be sub-optimal if done without care.

Some optimizations include:

1. Allocate memory when required and deallocate after the use in the subroutine. This prevents the overflow of memory.

2. If an array has operations inside a loop, allocate the array before the start of the loop. Preallocating the array means that the array does not have to be allocated and deallocated every time within the loop.

3. Utilizing the cache effectively is one of the most important aspects of optimizations. Most processes have a L1 cache which is a small, high speed memory. A larger L2 cache which is slightly slower and sometimes L3 and L4 caches depending on the processes and the main DRAM memory. Utilizing the caches effectively can be the difference between a fast code and a slow one. To effectively use the cache, one has to recognize the arrays and variables that are used most frequently in the code and make sure that they are if possible, pinned to the caches. This makes sure that they are in the fast memory and accessible whenever required with as less of a delay as possible. Most of the above is usually done by the compiler fully when set to the highest optimization possible, `-O3`. Some other user possible optimizations include:

    (a) Loop Interchanges: Interchanging the loops changes the memory access patterns and therefore allows a larger contiguous chunk of an array to be loaded to the cache, the faster memory. This improves register usage but this is possible only if the execution order of the nested loops are not important.

    (b) Loop fusion: Fusing loops that operate on same sized array into one loop, allows for multiple arrays to be operated on simultaneously thereby reducing the need to update the arrays separately. This is an example of increased instruction level parallelism.

    (c) Loop tiling: This increases the depth of the loops by adding additional loops to the inner loops. This improves data locality and the data reuse in the cache, by loading a chunk of the locally iterated array and thereby improving the access and allowing for reuse of data.

    (d) Pre-fetching of data: Pre-fetching of data before the computation can in a sense overlap the access and computation part of the subroutine thereby increasing the efficiency of the code. Pre-fetching is a pretty complex process and is usually handled by the compiler.

4. Another important aspect that needs to be considered is the data layout. These optimizations are called data layout optimizations. They are intended fo avoid cache conflict misses and they improve the spatial locality of the code. Array padding and merging of arrays are some of the commonly used techniques in these optimizations.

Most of the above optimizations are performed very effectively by good compilers. The optimizations that were applied explicitly include the allocation and deallocation of arrays and vectorization. Vectorization improves the data access patterns and allows for efficient use of caches. The results of the performance for different vectorization lengths and problem sizes are explained in Chapter 5.

Chapter 4

GPU code and optimizations

This chapter explains the different aspects that need to be considered before porting the code to multiple GPU's. Finally, the GPU algorithm and its intricacies are explained. This chapter is also meant to be a documentation for the GPU code along with the thought process that went into the decision making that ensued.

4.1 Graphical Processing Units and programming for general purpose

Graphical Processing Units (GPU's) have been around since the 1990's, popularized by NVIDIA[1]. Increasing power consumption with the high clock speeds is an issue due to overheating of the chips. Moore's law will sustain only for so long before we will have to adapt to other technologies to make our codes faster and more efficient.

Another aspect is the size of the silicon die is also reaching the limit and hence there is a definite stall expected in Moore's law in the future. NVIDIA and AMD are the two main companies that manufacture GPU's. NVIDIA's CUDA(Compute Unified Device Architecture) framework is presently at the forefront with the other open source programming frameworks catching up such as OpenCL and OpenACC. As most of the accelerators in most of the supercomputing clusters are NVIDIA GPU's, this project will focus on programming with the CUDA framework. As the code by Minev was a FORTRAN code, the GPU version has been written with CUDA Fortran which is a proprietary extension to CUDA developed by PGI.

[1] GPU, http://www.nvidia.com/object/IO_20020111_5424.html

4.1.1 Programming model

The CUDA programming model consists of execution of functions called kernels that are executed by all the threads being called. The main idea of GPU computing is the execution of multiple threads which makes use of the parallelism in the code. Therefore it is important to make sure that the kernel being executed has enough parallelism.

A kernel can be called from the host or the device (called dynamic parallelism) and the number of threads is specified using the chevron syntax, `<<<...>>>` . The arguments will be explained later. In CUDA Fortran, the kernel is made to be executed from the host by giving it the attribute `global`. The other `attributes` that can be given are `host` and `device`. Each thread that is executing the kernel has its own ID in built-in `threadIdx` variable. As will be explained later, the `threadIdx` is a 3 component vector corresponding to the three dimensions that the threads can be created in, logically.

4.1.2 Threads and Blocks

Threads and blocks are an important aspect of general purpose GPU programming, not only because their execution is the most important part of the kernel but because care has to be taken to make sure that they are optimally configured. Threads are the most basic level of execution of instructions. Threads are organized into blocks and blocks are organized into grids. Logically speaking, the threads and blocks can be 3-dimensional. The system that this project is based on, use the NVIDIA Tesla P100 cards, at the CSCS cluster, after their recent upgrade. It has a compute capability of 6.0[2]. The compute capability restricts some features based on the hardware. For example, from compute capability 3.0 upwards the maximum number of threads in a block was increased to 1024 from 512.

A derived data type in the standard is `dim3`. The arguments the kernel is launched with is specified by using these `dim3` type variables that hold the block and thread dimensions in each direction. Figure 4.1 shows the arrangement of thread and blocks in 2D[3].

A block can have a maximum of 1024 threads and it is recommended to have the number of threads in a block to be a multiple of 32 as 32 threads make up a warp, which a collection of threads. The maximum number of blocks in one direction is 65536. Therefore the maximum number of blocks possible in a device of compute capability 3.5 is $65536 \times 65536 \times 65536$.

[2] Product Specs, https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[3] Threads and blocks, https://docs.nvidia.com/cuda/cuda-c-programming-guide/

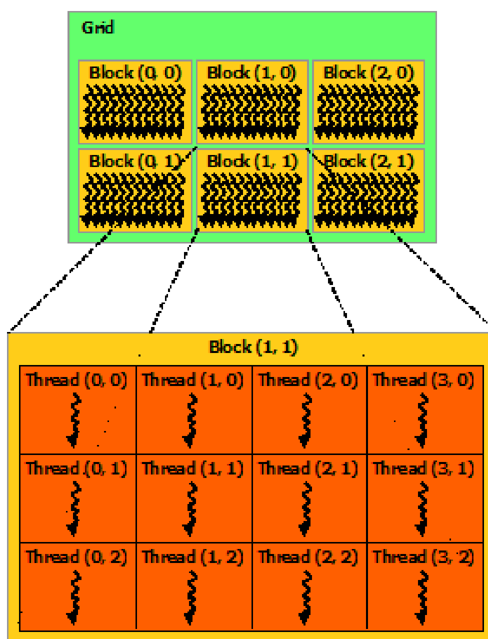


FIGURE 4.1: Thread, block and grid arrangement

4.1.3 Memory handling

GPU memory is one of the most important aspects of GPU programming because in most cases that have enough parallelism to be executed on the GPU, the memory is the bottleneck. Especially when there are multiple GPU's involved, it becomes very important to handle the memory very carefully and keep the memory as local as possible. The different memory aspects in the GPU are explained below. The memory hierarchy is as as shown in Figure 4.2[4].

4.1.3.1 Global Memory

This is the largest memory available on a GPU. It is available globally to all the threads and depending on the hardware and the GPU can go upto a few GB's. The P100 has a global memory of 16GB but not all of it may be available. Error correction usually occupies some memory. As the memory is large the transfer speed is lower. When an attribute of `device` is given to a variable, it is stored in the global memory. Copies to and from the host (CPU) are generally done using the `cudamemcpy()` function, either synchronously or asynchronously.

[4] Threads and blocks, https://docs.nvidia.com/cuda/cuda-c-programming-guide/

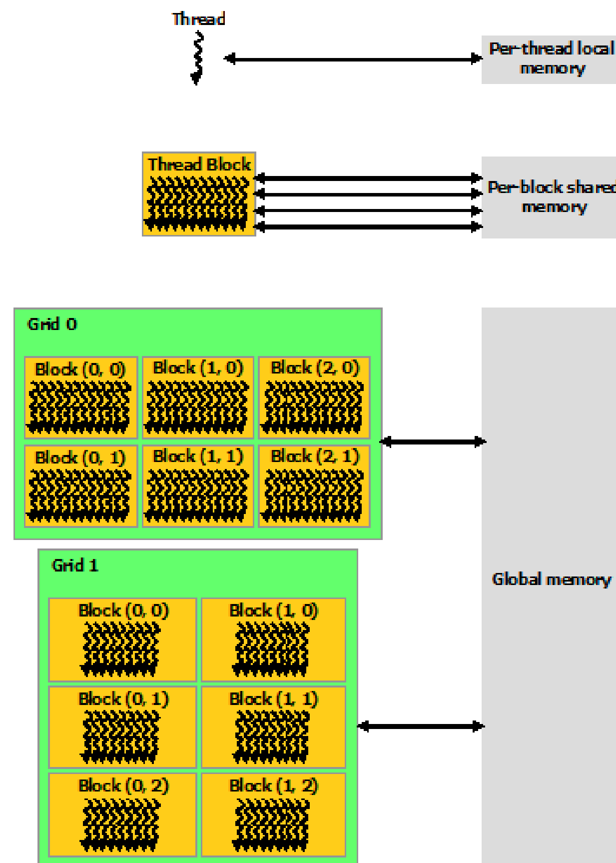


Figure 4.2: Memory with the threads and blocks

4.1.3.2 Shared Memory

All the threads in a thread block share a fast memory called the shared memory which is small and short lived but much faster. In Tesla P100 the shared memory can be upto 64KB per SM. If possible, the shared memory must be taken advantage of.

4.1.3.3 Pinned Memory

On the host the memory allocated usually resides in the pageable memory. But this memory may be overwritten during the various memory fetching phases. If one knows that a variable is required for most of the instructions and fetching phases, one can make the variable reside in a page-locked memory which is a memory that resides on the host and is beneficial for frequent transfers to and from the host.

4.1.3.4 Unified Memory

Unified memory or as NVIDIA calls it Unified Virtual Addressing(UVA)[5] allows for a single address space for both the memory on the host and the device to reside on. This feature is available in devices that are compute capability 2.0 or higher. This feature simplifies the memory handling and movement of data from and to the host from the device.

Specifically for the case when there are multiple devices and CPU's in working. it becomes much simpler the create a context between one rank(task) of a CPU and a GPU on the node and when the transfers are required from one GPU to the other, one transfers data from one MPI rank to another using the MPI communication standards. This simplifies the communication and allows for easier programming.

New features and technologies such as GPU Direct and some CUDA-aware MPI's such and OpenMPI, MPICH2 and MVAPICH2 MPI standards simplify the programmers efforts and faster communications by allowing transfers from one GPU to another directly. This is extremely important when all the kernels exist and run on the GPU and the CPU is just a context establisher. Unfortunately, the cluster at Max- Planck, Garching does not yet have MPI standard that are capable of doing this.

4.2 Harnessing the parallelism

The programming models in a MPI code and a GPU are different. Hence care must be taken to ensure that the efficiency of the code remains high when porting the code from MPI. As the GPU model consists of millions of threads working at once, an algorithm will be efficient only if it has enough amount of parallelism. Once the algorithm has been verified for scalability to the multiple threads, it becomes important that one is able to transform and make use of the available hardware and software on the GPU's to harness the parallelism in the algorithm. One of the techniques that is necessary to harness the parallelism is called tiling. It is the idea of computing the available sections in tiles whose sizes can vary from 8×8 to much larger sizes as well.

The main algorithm under consideration here is the Thomas algorithm for solving the linear systems. It allows for a maximum of n linear systems to be solved at once in a 2D problem of size $n \times n$, which is not enough to harness the parallelism of the GPU. Hence solving 2D problems will not be very efficient on the GPU's. But for the case of 3D, where the problems are of size $n \times n \times n$, we get to solve n^2 systems, hence for practically sized 3D problems GPU's can be effective. For the GPU, the algorithm changes slightly from Algorithm 4 and is given in Algorithm 5.

[5] Unified Memory, https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Algorithm 5 `PROGRAM nst`, The Navier–Stokes solver using MPI and GPU

```
1: procedure INITIALIZATIONS ON CPU'S
2:     procedure CREATE MESH AND DOMAIN DECOMPOSITION
3:         Read the mesh details from input file: CALL read_my_data(mesh)
4:         Create Cartesian decomposition of the processor grid: CALL create_comm(...)
5:     procedure ASSEMBLE MATRICES
6:         for i = 1 to dim do
7:             Assemble pressure matrix, CALL assemble_press(...)
8:             Assemble density matrix, CALL assemble_rho(...)
9:             Compute Schur complement and pressure and density matrices, CALL compute_schur(...)
10:            Assemble velocity matrices, CALL assemble_vel1(...) and CALL assemble_vel2(...)
11:            Compute Schur complement of velocity matrices, CALL compute_schur(...)
12:    procedure ALLOCATE MEMORY AND SET INITIAL SOLUTION
13:        Allocate memory for pressure, velocity and density fields
14:        if restart_in==yes then
15:            Load the fields from restart file, CALL restart_in(...)
16:        Compute the exact solution at time, t = 0 and update the fields and rhs.
17: procedure ENABLE PEER TO PEER TRANSFERS AND ESTABLISH CONTEXTS
18:    for i = 1 to max_rank do
19:        Get node number and name, get number of GPU's on each node
20:        if same_node==yes then
21:            Check compute capability of device, cudaGetDeviceProperties(...)
22:            if compute_capability .geq. 2.0 then
23:                Enable Peer to Peer access, cudaDeviceEnablePeeraccess(device(j), device(k))
24:        if first_time==yes then
25:            Get rank of calling CPU process, MPI_Comm_rank(comm,rank,ierr)
26:            Get node number and name, MPI_Get_Processor_name(comm,ierr)
27:            Store names in the array host_names.
28:            Broadcast the node name of the process to all processes , MPI_Bcast(host_names(rank, ...))
29:            Sort the names according to the node names
30:            for n = 1 to max_procs do
31:                if host_names(n) == host_names(n-1) then ! implies on the same node.
32:                    color = color + 1
33:                if host_name .neq. host_names(n) then
34:                    break
35:                Split the communicator based on the color assigned , MPI_Comm_split(comm, color, ..., newC omm,...)
36:                Get the new ranks in the new communicator, MPI_Comm_rank(newComm, myrank,ierr)
37:                Set the device based on the rank and establish the context, cudaSetDevice(myrank)
```

```
38: procedure Time stepping( Solved on the GPU)
39:     while t<tmax do
40:         procedure Solve for velocity fields
41:             for j = 1 to dim do
42:                 if source==yes then
43:                     Update the rhs with the source data, CALL cu_compute_source(...)
44:                 Compute gradient, CALL cu_grad(...)
45:                 if advection==yes then
46:                     Update the rhs with the advection data, CALL cu_advection(...)
47:                 Update the rhs with CALL cu_predictor(...)
48:                 for i = 1 to dim do
49:                     Transpose the rhs, to solve the linear system, CALL cu_reorder(...)
50:                     Add the boundary conditions, CALL cu_bcs(...)
51:                     Solve the linear system for (dir,dir) = (i,j), CALL
    cu_solve_schur2(...)
52:             Recover the solution and update the fields for next time step computations
    and also apply boundary conditions, CALL cu_bcs_end(...)
53:         procedure Solve for pressure field
54:             Compute divergence on the velocity fields, CALL cu_div(...)
55:             for i = 1 to dim do
56:                 Solve the linear system, cu_solve_schur2(...)
57:                 Transpose the rhs, for the next direction, CALL cu_reorder(...)
58:             Update the pressure fields for next time step and the corrected pressure as
    well.
59:         if rho==yes then
60:             procedure Solve for density field
61:                 if source_rho==yes then
62:                     Update the rhs with the source data, CALL
    cu_compute_source_rho(...)
63:                 if advection_rho==yes then
64:                     Update the rhs with the advection data, CALL cu_advection_rho(...)
65:                 Update the rhs with CALL cu_predictor_rho(...)
66:                 for i = 1 to dim do
67:                     Add the boundary conditions, CALL cu_bcs_rho(...)
68:                     Solve the linear system, cu_solve_schur2(...)
69:                     Transpose the rhs, to solve the linear system, CALL cu_reorder(...)
70:                 Recover the solution and update the fields for next time step computations
    and also apply boundary conditions, CALL cu_bcs_rho_end(...)
71: End of time loop
72: procedure Store results to vtk file and terminate program
73:     Write the required *.vtk files, CALL create_vtk_file(...)
74:     if restart_out==yes then
75:         Write the fields to a restart file, CALL restart_out(...)
76:     Terminate Program
```

NVIDIA recommends that the MPI ranks to GPU ratio be equal to 1:1 and hence this ratio has been maintained in this code. Algorithm 5 is the general algorithm for clusters with more than one GPU per node and with GPU's on multiple nodes being used. The cluster

at CSCS had one NVIDIA Tesla P100 per node and hence the association of a GPU to a MPI rank is not necessary. To simplify matters further the only the tridiagonal systems were solved on the GPU and the arrays were transferred back and forth from the GPU to the host. Therefore the simplified version of the algorithm above was used which is as given in Algorithm 6.

Algorithm 6 `PROGRAM nst`, The Navier–Stokes solver using MPI and GPU(no association)

```
1: procedure INITIALIZATIONS ON CPU'S
2:     procedure CREATE MESH AND DOMAIN DECOMPOSITION
3:         Read the mesh details from input file: CALL read_my_data(mesh)
4:         Create Cartesian decomposition of the processor grid: CALL create_comm(...)
5:     procedure ASSEMBLE MATRICES
6:         for i = 1 to dim do
7:             Assemble pressure matrix, CALL assemble_press(...)
8:             Assemble density matrix, CALL assemble_rho(...)
9:             Compute Schur complement and pressure and density matrices, CALL
   compute_schur(...)
10:            Assemble velocity matrices, CALL assemble_vel1(...) and CALL
   assemble_vel2(...)
11:            Compute Schur complement of velocity matrices, CALL compute_schur(...)
12:    procedure ALLOCATE MEMORY AND SET INITIAL SOLUTION
13:        Allocate memory for pressure, velocity and density fields
14:        if restart_in==yes then
15:            Load the fields from restart file, CALL restart_in(...)
16:        Compute the exact solution at time, t = 0 and update the fields and rhs.
17: procedure TIME STEPPING
18:    while t<tmax do
19:        procedure SOLVE FOR VELOCITY FIELDS
20:            for j = 1 to dim do
21:                if source==yes then
22:                    Update the rhs with the source data, CALL compute_source(...)
23:                Compute gradient, CALL grad(...)
24:                if advection==yes then
25:                    Update the rhs with the advection data, CALL advection(...)
26:                Update the rhs with CALL predictor(...)
27:                for i = 1 to dim do
28:                    Transpose the rhs, to solve the linear system, CALL reorder(...)
29:                    Add the boundary conditions, CALL bcs(...)
30:                    Transfer the arrays to the GPU
31:                    Do the first solve on the GPU to compute the local schur rhs values
32:                    Copy the local schur rhs values to the host
33:                    Gather the local schur values on the master process
34:                    Solve for the interface unknowns on the master process.
35:                    Scatter the scaled interface unknowns to all the host processes and
   transfer them to the GPU's
36:                    Do the final solve on the GPU's
37:                    Copy back the internal unknowns to the host.
```

```
38:             Recover the solution and update the fields for next time step computations
    and also apply boundary conditions, CALL bcs_end(...)
39:         procedure SOLVE FOR PRESSURE FIELD
40:             Compute divergence on the velocity fields, CALL div(...)
41:             for i = 1 to dim do
42:                 Transfer the arrays to the GPU
43:                 Do the first solve on the GPU to compute the local schur rhs values
44:                 Copy the local schur rhs values to the host
45:                 Gather the local schur values on the master process
46:                 Solve for the interface unknowns on the master process.
47:                 Scatter the scaled interface unknowns to all the host processes and transfer
    them to the GPU's
48:                 Do the final solve on the GPU's
49:                 Copy back the internal unknowns to the host.
50:                 Transpose the rhs, for the next direction, CALL reorder(...)
51:             Update the pressure fields for next time step and the corrected pressure as
    well.
52:         if rho==yes then
53:             procedure SOLVE FOR DENSITY FIELD
54:                 if source_rho==yes then
55:                     Update the rhs with the source data, CALL compute_source_rho(...)
56:                 if advection_rho==yes then
57:                     Update the rhs with the advection data, CALL advection_rho(...)
58:                 Update the rhs with CALL predictor_rho(...)
59:                 for i = 1 to dim do
60:                     Add the boundary conditions, CALL bcs_rho(...)
61:                     Transfer the arrays to the GPU
62:                     Do the first solve on the GPU to compute the local schur rhs values
63:                     Copy the local schur rhs values to the host
64:                     Gather the local schur values on the master process
65:                     Solve for the interface unknowns on the master process.
66:                     Scatter the scaled interface unknowns to all the host processes and
    transfer them to the GPU's
67:                     Do the final solve on the GPU's
68:                     Copy back the internal unknowns to the host.
69:                     Transpose the rhs, to solve the linear system, CALL reorder(...)
70:                 Recover the solution and update the fields for next time step computations
    and also apply boundary conditions, CALL bcs_rho_end(...)
71: End of time loop
72: procedure STORE RESULTS TO VTK FILE AND TERMINATE PROGRAM
73:     Write the required *.vtk files, CALL create_vtk_file(...)
74:     if restart_out==yes then
75:         Write the fields to a restart file, CALL restart_out(...)
76:     Terminate Program
```

The detailed solution algorithm for the GPU is as given in Algorithm 7

Algorithm 7 `GPU_solve`

1: **procedure** GPU TRI-DIAGONAL SOLVER
2: Allocate required device memory from host
3: **procedure** TRANSFER DATA TO THE GPU
4: Copy the tri-diagonal matrices to the device memory
5: Copy the `rhs` vector to the device with the required data layout
6: Launch the GPU solver with number of threads equal to `nb_arrows`
7: **procedure** SOLVE
8: Calculate the thread index
9: Solve the `nb_arrows` tridiagonal linear systems on `nb_arrows` threads using Algorithm 3
10: Transfer the arrays back to the host
11: Gather the local schur rhs values on the respective master processes.
12: Solve for the local interface unknowns on the host master processes.
13: Scatter the calculated scaled interface unknowns back to the respective processes
14: Transfer the scaled values to the GPU's.
15: Launch the GPU solver with number of threads equal to `nb_arrows`
16: **procedure** SOLVE
17: Calculate the thread index
18: Solve the `nb_arrows` tridiagonal linear systems on `nb_arrows` threads using Algorithm 3
19: Transfer the arrays back to the host.

Chapter 5

Results and Conclusions

5.1 MPI Results and Observations

The results presented here show the timings for the various versions of the MPI code. The code has been run for different process topologies, For details of correctness of the code and the values of the solution fields, refer to [7]. The different versions of the code include:

1. MPI code with solution algorithm set to Algorithm 1.
2. MPI code with solution algorithm set to Algorithm 2.
3. MPI code with vectorization (multiple tri-diagonal solutions using multiple threads) and Algorithm 2.

Referring to Table 5.1, grid size refers to the total number of grid points in the problem considering all the 3 directions. All simulations performed are for 3D cases. The process topology explains the total number of nodes and the number of tasks on each node. The vect length is the length of the vectorization, which is nothing but the length of the unrolled inner loop.

The First solve time refers to the first tri-diagonal solution as in step 5 for Algorithm 1 or steps 3-8 for Algorithm 2. The gather refers to the `MPI_Gather`. The schur solve refers to the tri-diagonal solution of the schur complement. The scatter refers to the `MPI_Scatter` and the Final solve refers to the final tri-diagonal solution. The Overall time is the complete CPU time which includes the run-time of the whole code including the writing of the visualization files. The number of grid points in each direction can be calculated by taking the cube root of the grid size. For simplicity, the number of grid points and the number of processes in each direction are equal. But, this is not necessary. The total number of points in each logical process can be calculated by dividing the grid size by the total number of tasks. For

example, if the grid size is 0.512×10^6, then the number of grid points in each direction is $\sqrt[3]{0.512 \times 10^6} = 80$ and the total number of grid points in each process is $0.512 \times 10^6/8 = 64000$. For higher Reynolds numbers the time step needs to smaller. The number of time steps taken here was equal to 1000.

The hardware used for the results that follow are from the CSCS Supercomputing cluster at Lugano, Switzerland. They use the XC50 Intel Haswell 12-code nodes with each node having a NVIDIA Tesla P100 GPU.

5.1.1 Observations and explanations

The following observations can be made from the timings shown in Table 5.1.

1. The solution with Algorithm 1 is less efficient than the solution with Algorithm 2. For very large cases, this is evident. Using vectorization improves the efficiency further by allowing multiple threads to work on multiple data with the same instructions. Though the compiler performs some vectorization, in some cases where there is a secondary function call within the loop, the compiler does not unroll the loop. Hence explicitly instructing the vectorization of these loops can increase the efficiency quite a lot.
2. The main improvements between Algorithm 1 and 2 is the difference in communication times. The time taken for Algorithm 2 for the `MPI_Gather` is considerably smaller and the time taken for `MPI_Scatter` is almost negligible compared to that for Algorithm 1.
3. It is observed that the tri-diagonal solution times remain almost the same for both the Algorithms. When vectorization is applied, the tri-diagonal solution time reduces.
4. For the cases when the total number of grid points in a process are the same but the total number of processes vary, for example the cases when grid sizes are 262.144×10^6 and 2097.152×10^6, the tri-diagonal solution times remain almost equal, but the communication times for the gather and scatter increase due to the usage of more processes.
5. Due to the reduction in message length, there is a improvement in the communication timings as can be confirmed from Figures 5.2(a) and 5.2(b).
6. The Schur solve step takes much less time than the first and the final solves because the size of the matrix and the unknowns are much smaller than for the other solves.
7. Some comparisons are done in the table. The improvements, shown in the brackets are in form of a ratio with respect to the original Minev algorithm, Algorithm 1.

8. For cases where each rank has to solve a large number of unknowns (greater than 4 million), without vectorization the Algorithm 2 is slower than Algorithm 1. This may be attributed to the serialization of the message sending frequencies along with the stalling of the message length sizes. Therefore, for larger cases vectorization is extremely important for efficiency.

TABLE 5.1: Comparison of MPI version timings

Parameters				Time (s)					
Grid size ($\times 10^6$)	Process Topology (nodes) $\times$ (tasks/node)	Vect length	Algorithm	First solve	Gather	Schur solve	Scatter	Final solve	Overall
$80^3 = 0.512$	$1 \times 8 = 8$	0	1	13	4	2.5	2.5	14	109
$80^3 = 0.512$	$1 \times 8 = 8$	0	2	17	0.8(5x)	2.4	0.1(25x)	17.9	68(1.6x)
$80^3 = 0.512$	$1 \times 8 = 8$	8	2	5(2.6x)	0.9(4.4x)	0.5(5x)	0.1(25x)	6.3(2.2x)	43(2.5x)
$160^3 = 4.096$	$1 \times 8 = 8$	0	1	98	36	10	10	107	975
$160^3 = 4.096$	$1 \times 8 = 8$	0	2	116	5.8(6.2x)	9.7	0.4(25x)	127	661(1.5x)
$160^3 = 4.096$	$1 \times 8 = 8$	8	2	47(2x)	6.4(5.6x)	2.2(4.5x)	0.5(25x)	57.1(1.8x)	521(1.9x)
$320^3 = 32.768$	$4 \times 16 = 64$	0	1	102	129	15	119	110	992
$320^3 = 32.768$	$4 \times 16 = 64$	0	2	113	14(9.2x)	5.6(2.6x)	1.8(66x)	121	548(1.8x)
$320^3 = 32.768$	$4 \times 16 = 64$	8	2	46(2.2x)	13(10x)	1.8(10.7x)	1.8(66x)	60(1.8x)	410(2.4x)
$640^3 = 262.144$	$4 \times 16 = 64$	0	1	767	1032	63	858	847	9138
$640^3 = 262.144$	$4 \times 16 = 64$	0	2	786	978	16.1(4x)	9.1(94x)	875	9336(0.97x)
$640^3 = 262.144$	$4 \times 16 = 64$	8	2	394(1.95x)	283(3.6x)	5.8(10.8x)	6.3(136x)	463(1.8x)	4338(2x)
$1280^3 = 2097.152$	$32 \times 16 = 512$	0	1	790	1051	103	1072	875	9576
$1280^3 = 2097.152$	$32 \times 16 = 512$	0	2	786	1085	11.1(9.3x)	19.0(56x)	872	9482
$1280^3 = 2097.152$	$32 \times 16 = 512$	8	2	439(1.8x)	338(3.1x)	4.5(23x)	15.6(69x)	489(1.8x)	3708(2.6x)

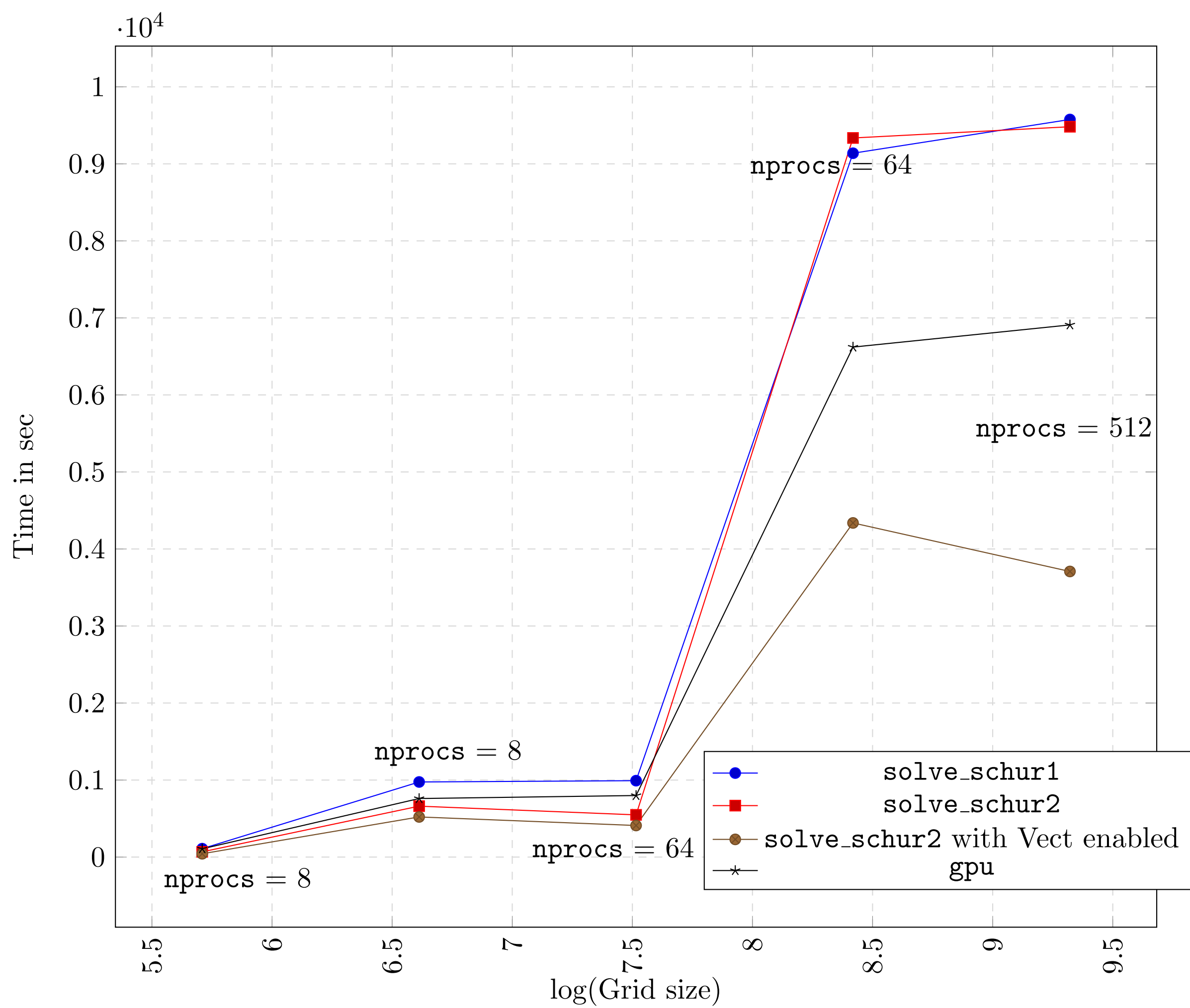
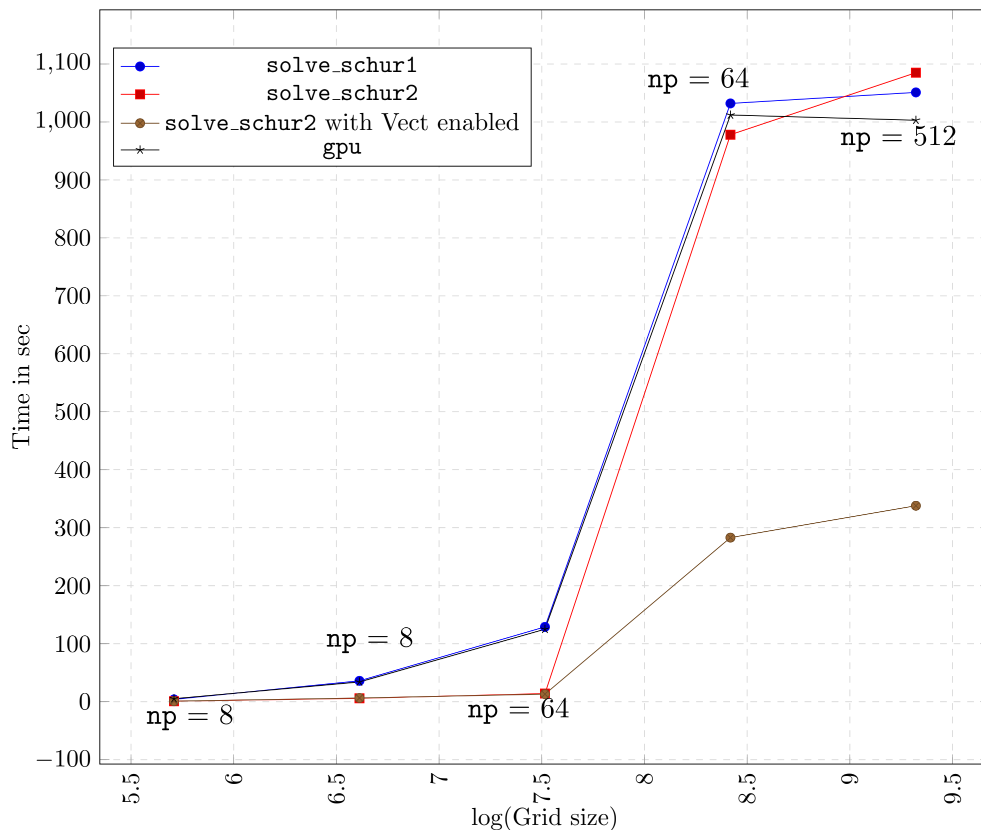
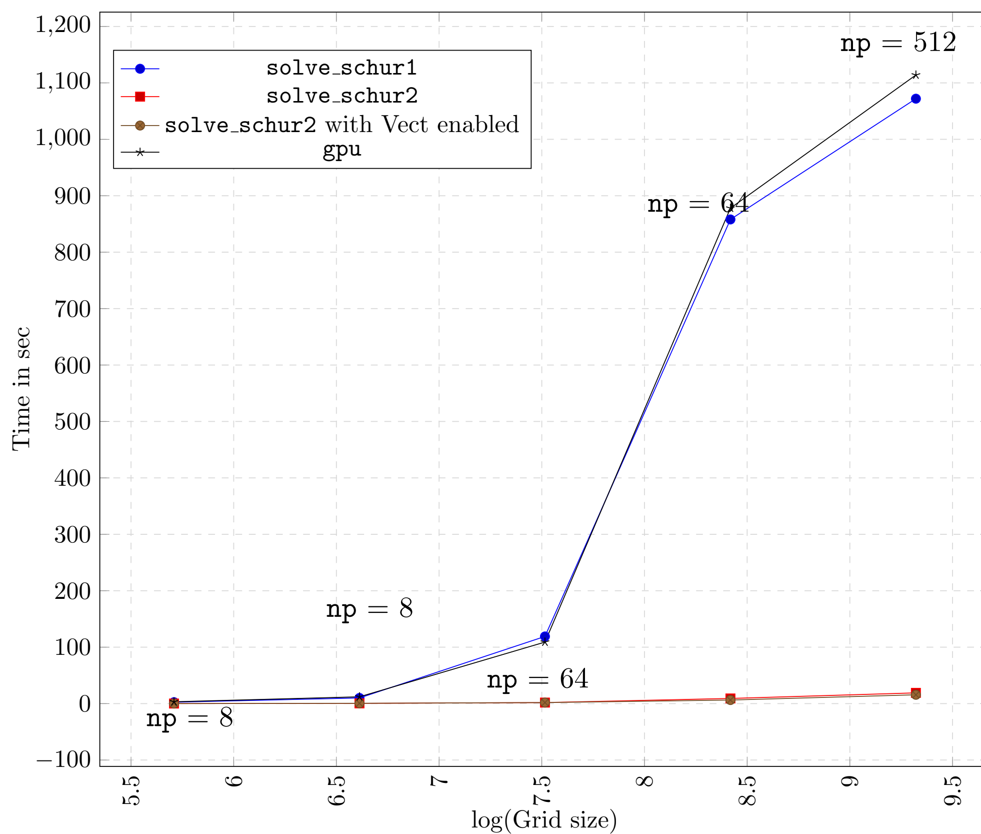


FIGURE 5.1: Overall timings for different grid sizes

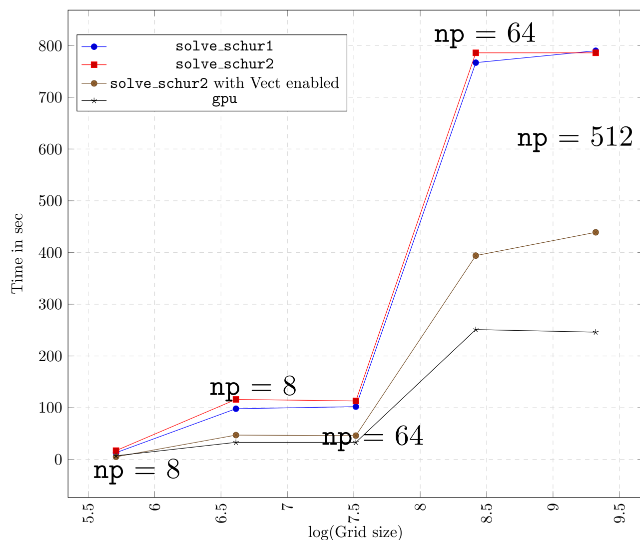


(a) Gather timings

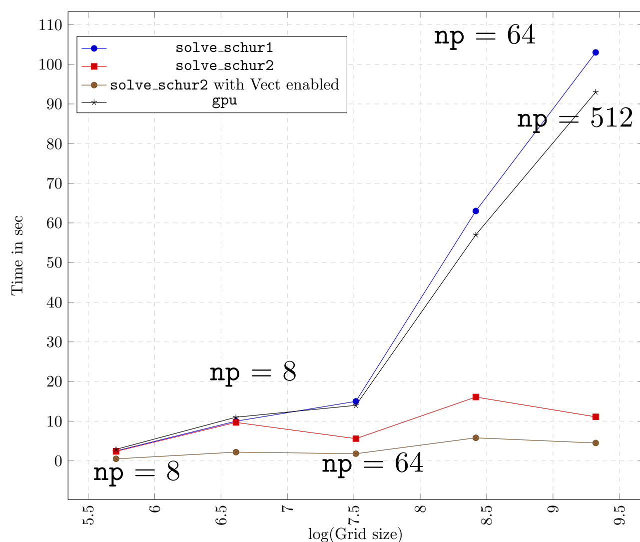


(b) Scatter timings

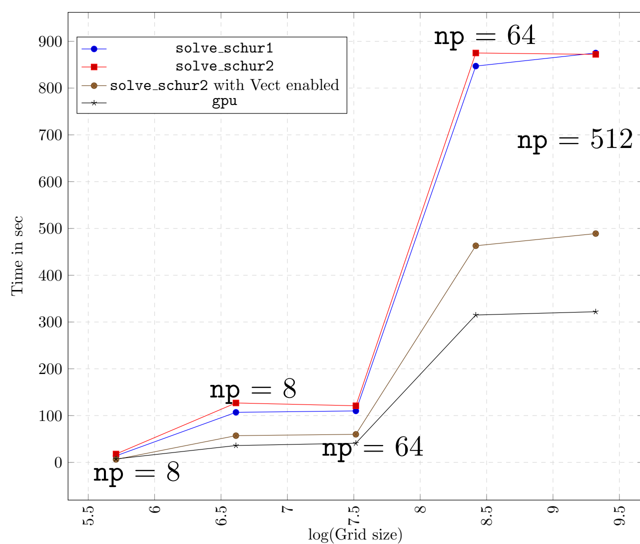
FIGURE 5.2: Communication timings for different grid sizes



(a) First solve timings



(b) Schur solve timings



(c) Final solve timings

FIGURE 5.3: Tri-diagonal solution timings for different grid sizes

5.2 GPU results and observations

The results for the multiple GPU's has been shown below. Some observations and comparisons with respect to the optimized MPI code and the original MPI code has been also shown. The timings for various parts of the solution are also tabulated to understand the improvements of the solution obtained through the GPU.

5.2.1 Observations and explanations

1. It can be observed that the tridiagonal solution timings are considerably reduced when the code is run on GPU's. A speedup of more than 3 times is observed for large cases.

2. As the current GPU code uses Algorithm 1, the communication times are the same as that of the MPI Alg 1 version.

3. The memory transfer timings are included in the solution timings, First solve and Final solve itself. Therefore the timings are indicative of memory transfer to and from the GPU and the computation of the solution on the GPU.

4. The Vectorization vector length here has been set to 16 as only the generic transpose uses the vectorization and therefore this reduces the overall time and not any of the other timings.

5. The GPU code is not as fast as the MPI vectorized version even though the first and final solves are faster because the communication timings between the different ranks are still inefficient compared to the MPI vectorized version.

6. The Schur solve step for the GPU code is also similar to that of the MPI Alg 1 because the schur solve is done on the CPU and not on the GPU in this case.

Table 5.2: Comparison of timings with GPU version

Parameters				Time (s)					
Grid size ($\times 10^6$)	Process Topology (nodes) $\times$ (tasks/node)	Vect length	Algorithm	First solve	Gather	Schur solve	Scatter	Final solve	Overall
$80^3 = 0.512$	$1 \times 8 = 8$	0	1	13	4	2.5	2.5	14	109
$80^3 = 0.512$	$1 \times 8 = 8$	8	2	5(2.6x)	0.9(4.4x)	0.5(5x)	0.1(25x)	6.3(2.2x)	43(2.5x)
$80^3 = 0.512$	$1 \times 8 = 8$	8	GPU, 1	7(1.8x)	5.1	2.9	3.2	7.4(1.9x)	107(1x)
$160^3 = 4.096$	$1 \times 8 = 8$	0	1	98	36	10	10	107	975
$160^3 = 4.096$	$1 \times 8 = 8$	8	2	47(2x)	6.4(5.6x)	2.2(4.5x)	0.5(25x)	57.1(1.8x)	521(1.9x)
$160^3 = 4.096$	$1 \times 8 = 8$	16	GPU, 1	33(3x)	34	11	12	38(2.8x)	760(1.3x)
$320^3 = 32.768$	$4 \times 16 = 64$	0	1	102	129	15	119	110	992
$320^3 = 32.768$	$4 \times 16 = 64$	8	2	46(2.2x)	13(10x)	1.8(10.7x)	1.8(66x)	60(1.8x)	410(2.4x)
$320^3 = 32.768$	$4 \times 16 = 64$	16	GPU, 1	33(3x)	125	14	109	39(2.9x)	799(1.25x)
$640^3 = 262.144$	$4 \times 16 = 64$	0	1	767	1032	63	858	847	9138
$640^3 = 262.144$	$4 \times 16 = 64$	8	2	394	283(3.6x)	5.8(10.8x)	6.3(136x)	463(1.8x)	4338(2x)
$640^3 = 262.144$	$4 \times 16 = 64$	16	GPU, 1	251(3.1x)	1012	57	878	318(2.7x)	6621(1.4x)
$1280^3 = 2097.152$	$32 \times 16 = 512$	0	1	790	1051	103	1072	875	9576
$1280^3 = 2097.152$	$32 \times 16 = 512$	8	2	439(1.8x)	338(3.1x)	4.5(23x)	15.6(69x)	489(1.8x)	3708(2.6x)
$1280^3 = 2097.152$	$32 \times 16 = 512$	16	GPU, 1	246(3.2x)	1003	93	1114	319(2.8x)	6909(1.4x)

5.3 Summary and conclusions

1. Memory and data access optimizations can bring about upto 2.5x improvement in the performance for the MPI version of the code.
2. A higher number of nodes is more beneficial as the memory available is higher. For very large cases, a large number of nodes is necessary so that the code does not run out of memory.
3. Reducing the communication message length can decrease the run-time of the code considerably.
4. A change in the process topology can also change the timings considerably. For the run time for two cases with the same number of processes, the one with higher number of nodes is more efficient than the one with lesser number of nodes. This may be attributed to the memory usage and availability.
5. The code is quite sensitive to the hardware architecture, compiler optimizations. The results and comparisons done above are for the same hardware and compiler optimizations.
6. For the GPU version, it is very important there is enough amount of parallelism in the algorithm. When multiple GPU threads are used to solve the tr-diagonal systems, the solution time reduces considerably even though the memory is being transferred to and from the GPU to the CPU.

5.4 Future work

Future work may include:

1. Extension to higher order time accurate methods.
2. Inclusion of the new GPU features such as GPU Direct Async to further improve efficiency.
3. Using the Algorithm 2 to solve the system on the GPU to reduce the communication timings.
4. Application of method to the higher order space accurate methods and to different splitting methods.

Bibliography

[1] Jim Douglas. Alternating direction methods for three space variables. *Numerische Mathematik*, 4(1):41–63, 1962. ISSN 0029599X. doi: 10.1007/BF01386295. URL http://dx.doi.org/10.1007/BF01386295.

[2] Alexandre Joel Chorin. Numerical solution of the Navier-Stokes equations. *Mathematics of Computation*, 22:745–762, 1968. ISSN 00255718. doi: 10.2307/2004575.

[3] J. L. Guermond and P. D. Minev. A new class of massively parallel direction splitting for the incompressible Navier-Stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 200(23-24):2083–2093, 2011. ISSN 00457825. doi: 10.1016/j.cma.2011.02.007. URL http://dx.doi.org/10.1016/j.cma.2011.02.007.

[4] J. L. Guermond, P. Minev, and Jie Shen. An overview of projection methods for incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 195(44-47):6011–6045, 2006. ISSN 00457825. doi: 10.1016/j.cma.2005.10.010.

[5] J L Guermond and Jie Shen. Velocity-Correction Projection Methods for Incompressible Flows. *SIAM Journal on Numerical Analysis*, 41(1):112–134, 2003. ISSN 0036-1429. doi: 10.1137/S0036142901395400.

[6] J. L. Guermond and Jie Shen. On the error estimates for the rotational pressure-correction projection methods. *Mathematics of Computation*, 73(248):1719–1738, 2003. ISSN 0025-5718. doi: 10.1090/S0025-5718-03-01621-1. URL http://www.ams.org/journal-getitem?pii=S0025-5718-03-01621-1.

[7] J. L. Guermond and P. D. Minev. Start-up flow in a three-dimensional lid-driven cavity by means of a massively parallel direction splitting algorithm. *International Journal for Numerical Methods in Fluids*, 68(7):856–871, 2012. ISSN 02712091. doi: 10.1002/fld.2583.