ISRN LUTMDN/TMHP-15/5333-SE

ISSN 0282-1990

Development and implementation of algorithms for the future large-scale computing in CFD

Robin Qvarfordt

Thesis for the degree of Master of Science in Engineering
Division of Mechanics
Department of Energy Sciences
Faculty of Engineering | Lund University

Development and implementation of algorithms for the future large-scale computing in CFD

Robin Qvarfordt

05 2015, Lund

This degree project for the degree of Master of Science in Engineering has been conducted at the Division of Mechanics, Department of Energy Sciences, Faculty of Engineering, Lund University.

Supervisor at the Division of Mechanics was Robert-Zoltán Szász.
Examiner at Lund University was Professor Johan Revstedt.

Thesis for the Degree of Master of Science in Engineering

ISRN LUTMDN/TMHP-15/5333-SE

ISSN 0282-1990

Efficient Energy Systems
Department of Energy Sciences
Faculty of Engineering, Lund University
Box 118, 221 00 Lund
Sweden

www.energy.lth.se

LTH

Abstract

Faculty of engineering
M-house, LTH

Master's degree

Development and implementation of algorithms for the future large-scale computing in CFD

by Robin Qvarfordt

The aim of this thesis is to study how the BCM (Building Cube Method) can improve performance of simulations in CFD with the steady increasing performance of modern parallel computers. A parallel program is developed and tested on different grid configurations and problems, among them the Navier-Stokes equations.

The first part of the thesis includes theory of numerical methods and software to be used when writing the program. Here is discussed how staggered grid is used to avoid the checkerboard effect, the basics of finite difference method, the PISO algorithm and software like openMPI (open Message Passing Interface) which is used to parallelize. This is followed by a description of the implementation and testing.

Results are found continuously in chapter 4 and chapter 5. Chapter 4 is about stability and convergence and covers some explicit and implicit implementation of the wave equation. The results in chapter 5 are focused on accuracy and efficiency. Here, a discretization of Poisson's equation is done and solved on different grid spacings in order to verify the second order implementation. Further, a comparison between Multigrid and Gauss-Seidel is done while solving Stokes flow. Results on speed-up tests are found in the last section if this chapter, as well as a somewhat more superficial comparison with the OpenFOAM's solver.

The results show the implementation works as intended except for the PISO algorithm due to some inconsistency. Otherwise the scaling of the speed-up test turned out good.

Acknowledgements

I would like to thank my supervisor Robert-Zoltán Szász for taking the time to guide me through this project and supplying me with data.

I would also like to thank for the resources provided by the Center of Scientific and Technical Computing at Lund University (LUNARC) where part of the computations have been carried out.

Contents

List of Figures

Abbreviations

CFD	**C**omputional **F**luid **D**ynamics
MPI	**M**essage **P**assing **I**nterface
BCM	**B**uilding **C**ube **M**ethod
PISO	**P**ressure **I**mplicit **S**plitting **O**perator

Symbols

Symbol	Name	Unit
μ	dynamic viscosity	Pa$\cdot s$
ρ	density	kg/m^3
τ	stress tensor	Pa
p	pressure	Pa
u	velocity	m/s
N	number of cells	-

Chapter 1

Introduction

1.1 Background

CFD (Computational Fluid Dynamics) is increasingly spread in both academia and industry. Previously, simulations within CFD have been seen as a bit more exotic and non concretely way to solve problems as compared to other scientific fields. Over the years, however, computer performance has improved and simulations are seen more and more as a tool to complement and in some cases completely take over the analysis of reality-based problems.

Both [1] and [2] describe how today's computers are more and more towards parallel processors, as a result of the limitations of computations on an individual CPU. As a consequence, algorithms must be adapted in order to fully exploit the capabilities of the computer. Dividing the work of n processors can at best speed up the overall process n times, (if not super-linear). Parallelization can only be used by those parts of the solution that is not serially constructed, making the total speed up less than n times faster. Further, accelerators such as GPU's may be interesting to further speed up the calculations. These are designed to carry heavier calculations than CPU's.

Two important factors for simulations are accuracy and efficiency. Accuracy is affected mainly by the solution method and choice of discretization algorithms. These in turn affect properties such as stability and convergence. The grid that forms the calculation domain also affects the accuracy. In [2] it is described in a comparison between the Cartesian grid and unstructured grid, how the former can be generated up to three

times as fast as the unstructured grid. In conjunction with today's faster computers and transition to parallel processors, the Cartesian grid has many advantages. Compared with the unstructured grid, the Cartesian grid has benefits in terms of easier and faster discretization algorithms, with higher accuracy and can quickly solve higher order of discretizations. However, its simplicity affects the ability to locally refine the grid. Complex geometries with curved stripes and rapidly changing gradients are therefore still not easy to solve using only the Cartesian grid.

Further, in [2] it is explained how BCM (Building Cube Method) combines the advantages of structured and unstructured grid by generating Cartesian cubic sub domains, arranged in a tree (called octree- structure). All such cubes may have eight equal subcubes and hence all the cubes are treated equally. This favors parallel computing and also local refinement as irregular cells are avoided. The simplicity of the Cartesian grid is still there, but also the ability to deal with more complex geometries.

1.2 Goal

The goal of this work is to develop algorithms that allow efficient large-scale CFD calculations. A milestone can be to obtain a working implementation of a simplified canonical flow situation. This can be followed by improvements in terms of optimizations, which can then form the basis for solving more complex problems. Several solution algorithms will be developed and then evaluated in terms of accuracy and efficiency. Comparisons with open source software can be done to verify the implementation in various ways.

Chapter 2

Theory

2.1 Navier-Stokes Equations

The Navier-Stokes equations are widely used to describe viscous fluid flows. Together with the continuity equation they can be written in tensor form as

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0 \tag{2.1}$$

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} = \frac{\partial \tau_{ij}}{\partial x_j} - \frac{\partial p}{\partial x_i} \tag{2.2}$$

The stress tensor for linearly viscous gas or Newtonian fluid is $\tau_{ij} = \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$. There is also an energy equation used for compressible and more advance incompressible models. But in this thesis, only simple incompressible cases are covered.

Many simplifications can be done in order to solve for some special cases. Maybe the simplest form of equation 2.2 arises for a Newtonian incompressible flow where the viscous forces are dominant, i.e the Reynolds number is very low. In this case, the viscous term reduces to $\frac{\partial \tau_{ij}}{\partial x_j} = \mu \frac{\partial^2 u_i}{\partial x_j \partial x_j}$ and the non-linear term can be regarded as negligible small due to the low velocities. Thus, the equations for Stokes flow are

$$\frac{\partial \rho u_i}{\partial x_i} = 0 \tag{2.3}$$

$$\mu \frac{\partial^2 u_i}{\partial x_j \partial x_j} = \frac{\partial p}{\partial x_i} \tag{2.4}$$

2.2 Building Cube Method

There are mainly two ways to obtain a grid mesh from a given domain, the Cartesian way and the irregular or unstructured way. A three dimensional domain with a Cartesian mesh consists of equal sized cubes or some stretched variants of these. This type of mesh were used in the early days of computing, but when complex geometries became more important, so did the mesh describing them. Unstructured mesh became a faster way of generating the mesh. Nowadays, when computer performance has increased considerable since the early days and parallel processing starts to play a more significant role, better adapted algorithms need to be developed. [2]

As described in [2], a new way of treating the mesh has been developed. This consists of dividing the domain into so called, building cube blocks. Each of these blocks are self similar, thus containing 8 smaller blocks. The whole structure becomes an octree with one single or many root blocks. In this way, regions can be refined locally to a predefined level and each block can be solved for using a Cartesian mesh, see figure 2.1.

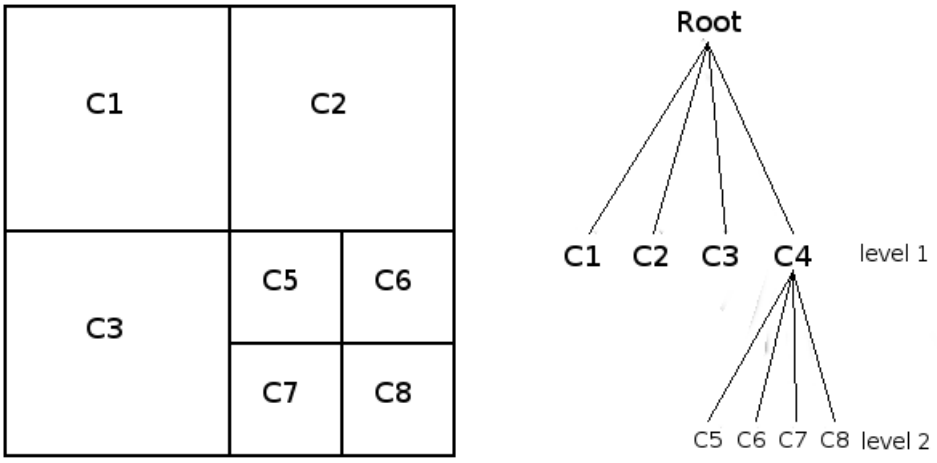

FIGURE 2.1: An example of how the refinement levels are constructed in BCM for a simple 2 dimensional case.

The BCM has advantages inherited from both the unstructured mesh and the simple Cartesian mesh. Since all cubes are self similar, it's easy to code in parallel. Also, higher order boundary connections between blocks are simplified compared to the unstructured

mesh. There is no need for calculating angles, so the mesh generation becomes much faster as well. The Cartesian mesh is still there, but the flexibility is more like the unstructured mesh.

2.3 Discretisation

2.3.1 Finite Difference Method

The simplest form of discretisation can be obtained by extracting derivatives from the Taylor series expansion [3].

$$f(x_0 + h) = f(x_0) + \frac{f'(x_0)}{1!}h + \frac{f^{(2)}(x_0)}{2!}h^2 + ... + \frac{f^{(n)}(x_0)}{n!}h^n + R_n(x_0) \tag{2.5}$$

In the above expansion around x_0 it is easy to find a forward finite difference for the first derivative by rearranging the terms and dividing by the step size h.

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - \frac{R_2(x_0)}{h} \tag{2.6}$$

The last term is the truncation error which is the error when compared to the real analytic derivative function. It is of first order since the lowest exponent of h is 1.

It is possible to combine equations like 2.5 with different signs on h in order to produce central and backward finite differences as well. The order of the truncation error is then an important measurement of how fast the error of a solution algorithm should decrease as the step size decreases.

2.3.2 Staggered Grid

When the grid is meshed up, it is time to decide where to assign the different variables. Maybe the most intuitive is to use the cell center or corner, but this may not always be the best way. When dealing with the Navier-Stokes equations both velocities and pressure are to be calculated. There is a strong coupling between the pressure and the velocities, why a highly non-uniform pressure field can act like a uniform field in the

momentum equations if all variables share the same points in space [4]. This is called the checker-board effect and this is illustrated in figure 2.2.

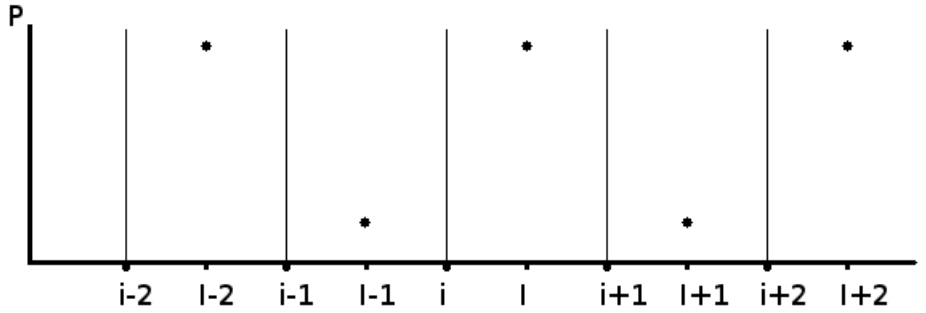

FIGURE 2.2: Example of a pressure field as a function of the spatial indices. Capital letter I indicates cell center, while lower case i indicates cell face. The vertical lines are border between cells (cell faces). The dots are example of how pressure values can behave.

Assume that the density is one and the convective and viscous terms in equation 2.2 are zero. Then, using a central difference scheme in space, the discretised momentum equation in one dimension reads

$$\frac{u_I^{n+1} - u_I^n}{\Delta t} = \frac{p_{I+1}^n - p_{I-1}^n}{2\Delta x} \tag{2.7}$$

If both velocity and pressure were to be saved at the same points, the pressure field in figure 2.2 would generate a zero velocity field. One way to deal with this situation is to use a staggered grid, i.e save the non scalar variables at the cell faces, between the scalar variables. This way, the velocities are calculated as

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{p_{I+1}^n - p_I^n}{\Delta x} \tag{2.8}$$

Now, the oscillations of the pressure are transferred to the velocity in a more physical meaningful way than before. If the analytic oscillations have an even higher frequency than this, the cells have to be smaller in order to capture it, but at least it is captured whenever the frequency is $1/2$ per cell length, so the checker-board effect is avoided.

2.4 Parallel Processing

To speed up the implementation, cubes can be distributed between multiple processors and dedicated libraries like OpenMPI [5] can be used to run the parallel parts. This section describes how an efficient distribution can be achieved.

2.4.1 Hilbert Curve

Figure 2.3 shows a two dimensional Hilbert curve. If two points in space are close to each other, then there is a great chance that they are also close each other on the Hilbert curve. This locality property is useful when coding in parallel. Later, when the cells are to be distributed between many processors, locality of cells that belong to the same processor is advantageous and requires less communication between processors.

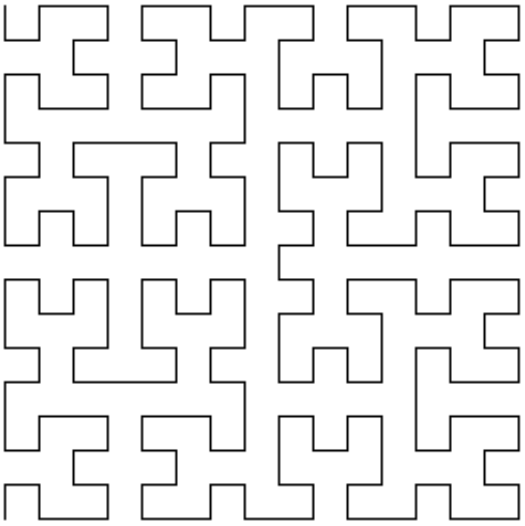

FIGURE 2.3: A two dimensional Hilbert curve

In [6] a way to implement this into an BCM-based structure is presented. It is described how cubes can be distributed among the processors by going in depth first and follow a schematic rule system to extract the cells from the octree to the processors.

2.4.2 Distribution Of Work

There exists a condition on certain partial differential equations (e.g. the hyperbolic wave equation) that reads $\Delta t \left(\frac{u_x}{\Delta x} + \frac{u_y}{\Delta y} + \frac{u_z}{\Delta z} \right) \leq C_{max}$, where C_{max} is the maximum

Courant number depending on the decretisation scheme [7]. This means that smaller cells need smaller steps in time in order to satisfy the CFL condition when explicit steps in time are used. If the size of a cell is half of another cell, the time step of the smaller one needs to be half of the bigger one in order to still satisfy the CFL condition. This implies that twice as many time steps needs to be run for the smaller cell compared to the bigger cell in order to traverse the same amount in time. This means that smaller cells require more work from the processor handling it.

If all processors should have the same amount of work, a weight can be assigned to each cube containing equal sized cells, corresponding to the size or level of the cube. For example, if the work of level 1 cubes is 1, then the work of level 2 cubes is $2^{2-1} = 2$ and the work of level k cubes is 2^{k-1}. In this way, it is easy to keep track of the total work for each processor.

2.5 Multigrid

Another great feature of the Cartesian grid is that it is easy to apply the powerful Multigrid technique. As described in [8] Multigrid is based on calculating the residual on a given grid and then restrict it to a coarser grid where the corresponding error is computed and returned to the origin grid in a recursive fashion. The simplest way is to use V-cycles where one or more recursive calls are used to coarser grids. Then, when the coarsest grid is reached, the error is solved and refined to fit the finer grid.

Compared to other simpler iterative methods such as Jacobi and Gauss-Seidel, the Multigrid strategy can perform 10 to 100 times faster [8]. But since Multigrid is more of a strategy than a well defined method, it can not replace the other methods. It is however, a great accelerator i cases were convergence is slow.

2.6 PISO Algorithm

As stated in [9], the implicit discretisation of the Navier-Stokes momentum equations 2.2 may be written as

$$A_P^{u_i} u_{i,P}^{n+1} + \sum_l A_l^{u_i} u_{i,l}^{n+1} = Q_{u_i}^{n+1} - \left(\frac{\delta p^{n+1}}{\delta x_i}\right)_P \tag{2.9}$$

The index i represent the current spatial velocity component, while index P denotes the current cell. Index l denotes the neighboring cells that are dependent. $A_P^{u_i}$ and $A_l^{u_i}$ are the coefficients depending on size of time step and size of the cells and type of discretisation scheme. Further, $Q_{u_i}^{n+1}$ contains the non-linear convective term and other terms that cannot be explicitly expressed as a linear combination of $u_{i,P}^{n+1}$ or $u_{i,l}^{n+1}$. The last term is the pressure gradient, for which discretisation method is not specified.

When solving the implicit equation 2.9 for $u_{i,P}^{n+1}$, the idea is to fix the right hand side at the current time step, so that only the velocity being solved for is updated. Using the same notation as done in [9], the most current updated values are given index $m*$ while the fix values are given the index $m-1$ to indicate values just before the momentum equations are being solved. In this way equation 2.9 becomes

$$A_P^{u_i} u_{i,P}^{m*} + \sum_l A_l^{u_i} u_{i,l}^{m*} = Q_{u_i}^{m-1} - \left(\frac{\delta p^{m-1}}{\delta x_i}\right)_P \tag{2.10}$$

To be able to solve these equations, another equation for the pressure is required. This should come from the continuity equation since it is also involved. Since it is not sure that u_i^{m*} satisfy continuity, it needs to be corrected to a new value u_i^m which satisfies continuity. Again, according to [9], this is done by inserting the expression for u_i^{m*} with the pressure gradient updated to $\frac{\delta p^m}{\delta x_i}$ into the continuity equation 2.1. Then the Poisson equation for the pressure appears as

$$\frac{\delta}{\delta x_i}\left[\frac{\rho}{A_P^{u_i}}\left(\frac{\delta p^m}{\delta x_i}\right)\right]_P = \left[\frac{\rho \tilde{u}_i^{m*}}{\delta x_i}\right]_P \tag{2.11}$$

Here, $\tilde{u}_{i,P}^{m*} = \frac{Q_{u_i}^{m-1} - \sum_l A_l^{u_i} u_{i,l}^{m*}}{A_P^{u_i}}$.

After solving 2.11 for new pressure, the velocities can be corrected so that continuity is fulfilled. This is done by

$$u_{i,P}^m = \tilde{u}_{i,P}^{m*} - \frac{1}{A_P^{u_i}}\left(\frac{\delta p^m}{\delta x_i}\right) \tag{2.12}$$

Now, in the PISO algorithm, equation 2.11 is solved one more time with the new velocities so that the velocities can be corrected one more time. After that, the momentum equation is not satisfied and a new round of solving the momentum equation followed by correction steps are necessary, until the changes are considered small. In each such round or inner loop, the terms $A_P^{u_i}$, $A_l^{u_i}$ and $Q_{u_i}^{n+1}$ are updated with the most current values corresponding to time step n+1. Then a new time step can be computed in a new outer loop.

A schematic flow chart of the PISO algorithm for the incompressible Navier-Stokes equations is given in figure 2.4.

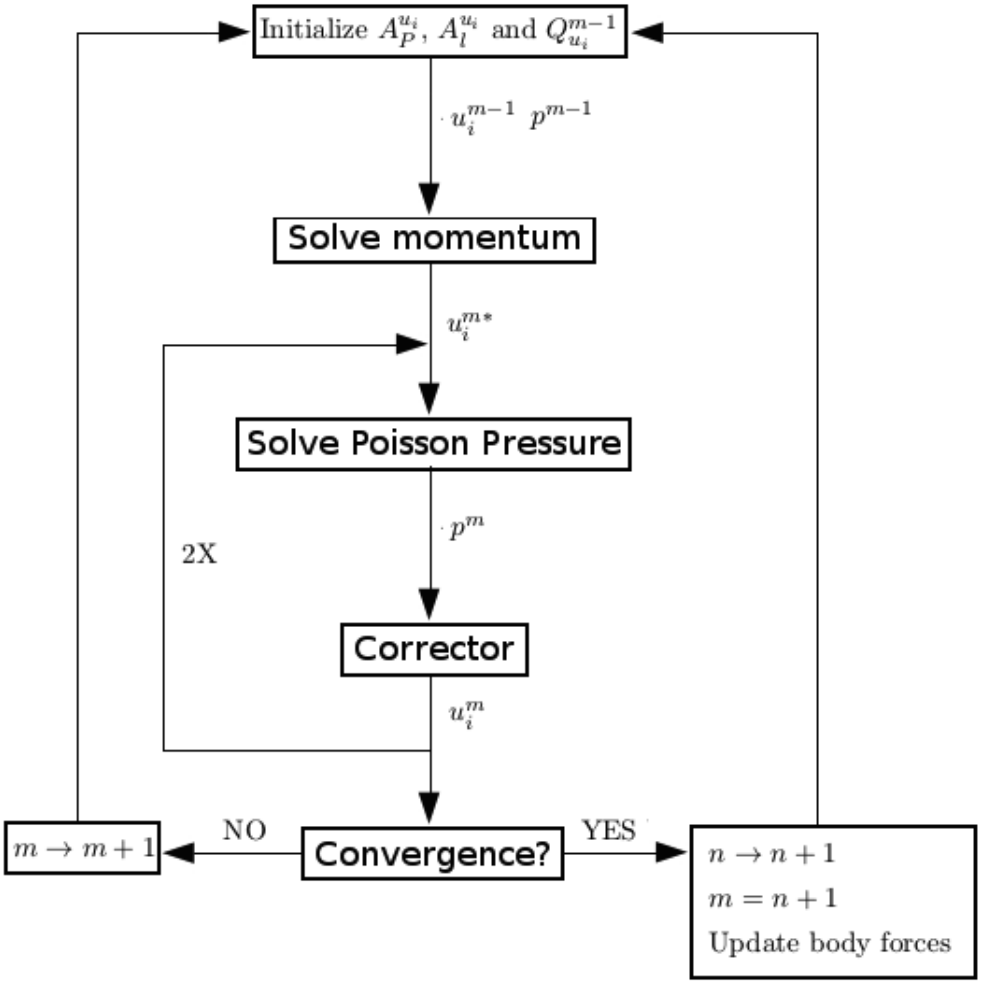

FIGURE 2.4: Schematic flow chart of the PISO algorithm for the incompressible Navier-Stokes equations. After each box the variables to the right of the arrow indicates that they have been updated and are used as input parameters to the operation in the next box.

Chapter 3

Implementation

3.1 Overview

This chapter explains the implemented program as a whole and tries to go into detail where its necessary. No code is given, but rather figures are used as illustrations of how a specific part of the program works.

As compared to other CFD-programs both a pre-processor and a part of a post-processor are implemented and deals with the reading and writing as well as the mesh generation. Further, there is a solver and a part that reads and writes the structure for multiple processors. A schematic overview is given in figure 3.1.

3.2 The Cubes

3.2.1 Refinement levels

The building cubes used in this implementation have a Cartesian grid which is used to solve a problem with some given stencil and boundary conditions.

These cubes are implemented in an octree structure as described in section 2.2. A cube can either have no children or 8 children, but it is only the cubes with no children that are used when solving equations.

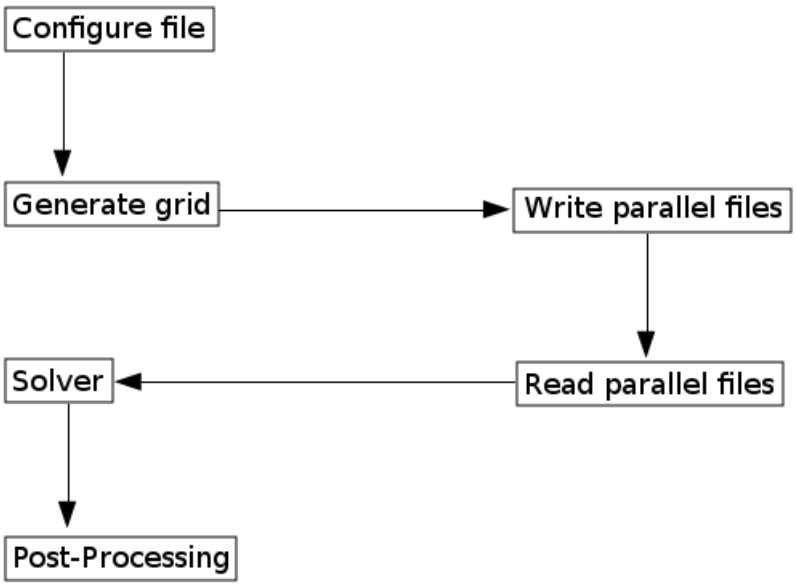

FIGURE 3.1: Schematic overview of the program.

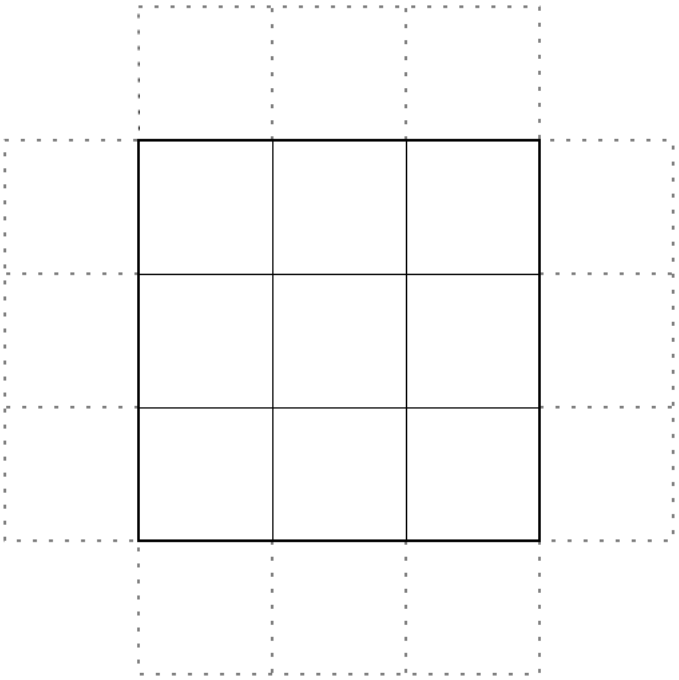

FIGURE 3.2: Example of a two dimensional cube with N=3 cells along each direction. Boundary cells are being marked with dashed lines.

In the configure file, all information about the octree structure should be specified. This includes the number of cubes on the root grid, which are the cubes that are on level 1. Depending on which regions that are interesting, some of these cubes may be further refined. The rule is that neighboring cubes cannot differ more than one level.

The regions where calculations are done are called fluid regions and are defined by objects. The objects can be a geometrical shape, like a sphere or a cylinder. The dimensions of those objects are read from the configuring file, whereby the regions covering the objects are refined to a predefined level. All the cubes inside an object are inactive, while the ones outside belong to the fluid region. In order to get good resolution at the border of a object, the cells inside the cubes are also marked as active/inactive. Figure 3.3 illustrates this.

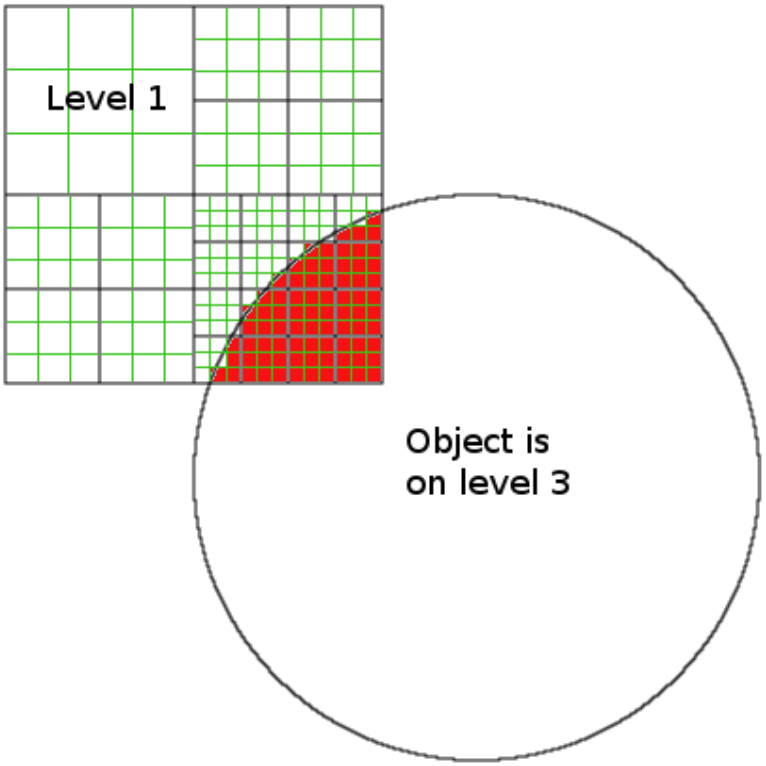

FIGURE 3.3: Example of how the cells of two dimensional cubes are marked inactive (red cells) when their cells centers are inside the object. In this case, N=3 for all cubes where the green lines mark the cells, while the black lines mark the cubes.

3.2.2 Communication Between Cubes

Figure 3.2 shows a two dimensional cube with boundary cells of the first order. If there were more layers of boundary cells, these would have been of second and higher orders. The order of the boundary is important for accuracy since then it's possible to use higher order discretisation schemes. In this thesis though, only first order boundary cells are implemented.

Communication between cubes are done after each of them has been solved. Then they receive their boundary from the cube located in the corresponding direction. If the communicating cubes are on different levels of refinement and/or they have a different number of cells, then some interpolation is necessary in order to get as good estimation of the boundary values as possible. The above part of figure 5.3 in chapter 5 gives an example of communication between two cubes on different levels. Here, the smaller cube tries to receive its boundary by picking the closest boundary point in the larger cube. Figure 3.4 shows the opposite case where the larger cube needs to interpolate.

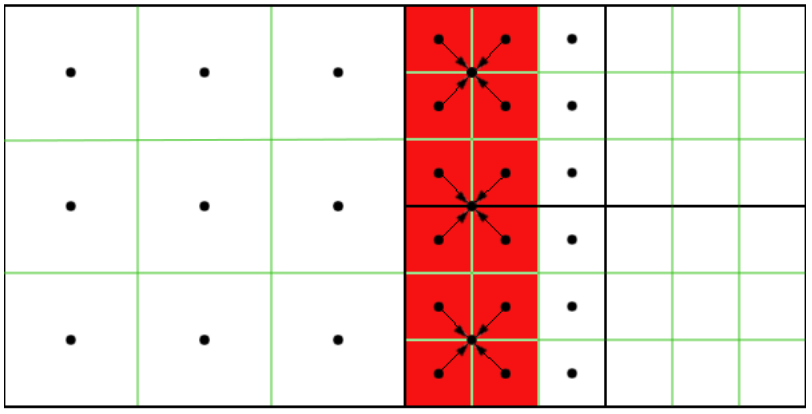

FIGURE 3.4: A two dimensional example of communication between two cubes on different levels of refinement. The boundary cells of the larger cube are marked with red. Interpolation of every boundary cell in the larger cube is done with an average value of 4 cells in the smaller cube, see the arrows. The corresponding case in three dimensions would have been an interpolation of 8 cells.

Apart from communication between cubes internally for a single processor, cubes have to be able to send and receive boundaries to and from neighboring processor cubes. How this is done, is discussed more in the next section.

3.3 Read and Write

3.3.1 Writing

The first place where reading occurs is in the pre-processor, where the configurations are read from a simple txt-file. Here, all grid specific data are read, such as dimensions, objects and parameters necessary to generate the cubes. The objects as described earlier, define the maximum level of refinement.

After the cubes have been generated, they are distributed among different processors according to their Hilbert index and then saved as an octree structure by writing to parallel files, one for each processor. As described in chapter 2.4.2, the cubes can have different weights according to which problem are to be solved. The way of writing them is not straight forward since not only do the cubes communicate within the same processor, they do also communicate with other processor cubes. In this implementation, this is solved by also including the cubes that belong to another processors which communicates with one or many cubes from this processor, (called bordercubes). In this way it is easy to update the boundary of the bordercubes and send it to the other processor. One cube may be written to multiple processor files, but only belongs to one of them. This is similar to the communication between the cube's cells as describes in section 3.2.2, only now it is a communication between the processor's cubes instead of the cube's cells. The hard part is to find all such bordercubes and write them only once to the right processor file.

As stated in section 2.4.1, the Hilbert index is unique for a given level of refinement. This index can be used as a hash value in a map with cube pointers, one map for every level of refinement. This map has been implemented and can be used to uniquely find a cube with a given index and level of refinement, as well as collecting a set of unique cubes that should be written to a processor file. Figure 3.5 illustrates this.

The bordercubes also need to be marked with the directions in which they should send their boundary values. This is done by setting logical bits, where the bit on place k indicate whether the boundary in direction k should be sent from this bordercube. So every time a bordercube is visited when trying to find all bordercube for a certain processor, the logical bits representing the directions of that bordercube is updated. Figure 3.6 shows a case for two dimensions. In the three dimensional implementation,

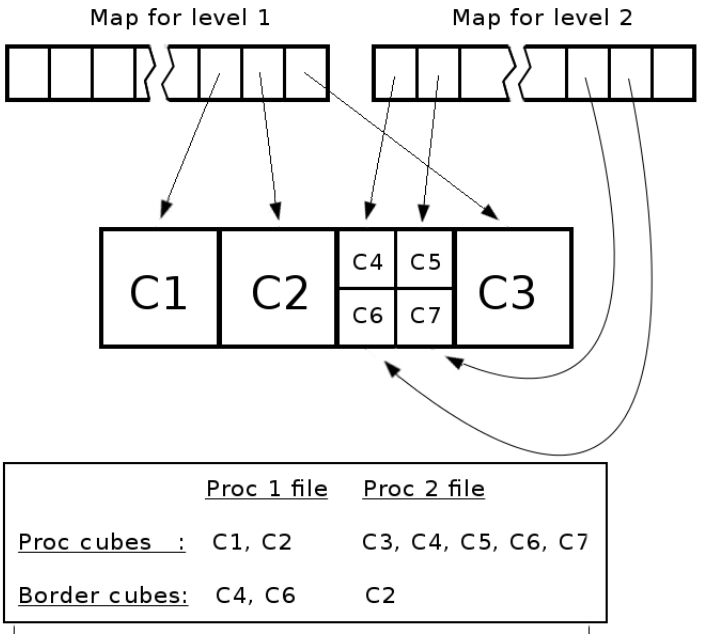

FIGURE 3.5: The cubes C1-C7 are inserted in a map corresponding to their level of refinement. The table shows how the bordercubes are written to the parallel files in the case of 2 processors of this simple two dimensional case.

there are 18 different directions, 6 sides and 12 edges. The corner are not used in any calculation so these are skipped.

3.3.2 Reading

The written processor files contain all necessary information about the cubes to recreate them. The most important information for the reconstruction is about the children and parents of a cube. As described in the previous section, maps together with a index were used to uniquely write cubes to the different processor files. It actually becomes handy when reconstructing the cubes again. For example, if both a cube and its child has been read and inserted in a map according to their indices and levels, that cube can find its child by recalculating its child's index and look in the map with the level one more than the cube's level. The same idea applies for the parent. The index doesn't need to be the hilbertindex, its enough that it is unique for a given level of refinement. This can be a very easy calculation as a function of the the discrete coordinates of the cube being searched for. This is illustrated in figure 3.7.

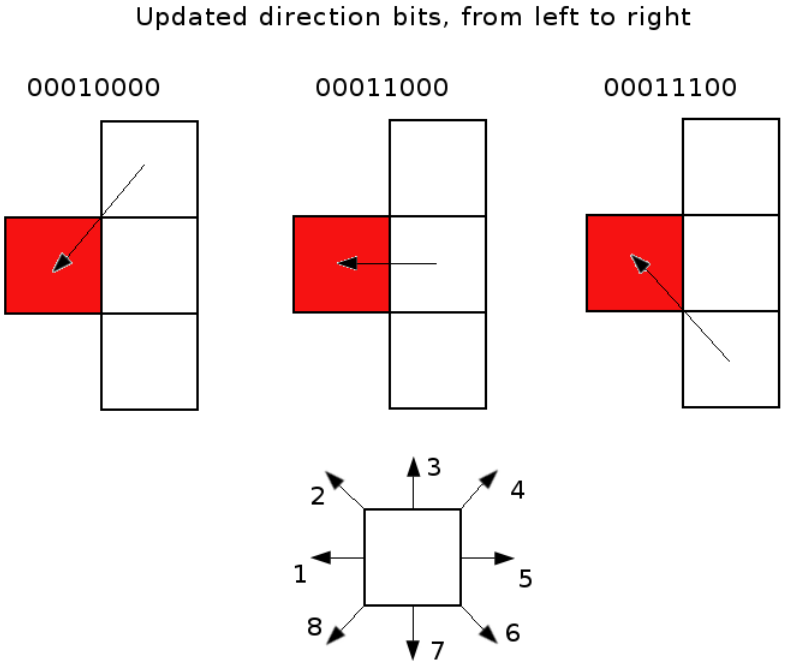

FIGURE 3.6: The red cube is a bordercube and the white cubes belong to the same processor. The figure illustrates how the direction bits are updated in the bordercube each time it is visited from one of the white cubes. These bits are written together with the bordercube to the corresponding processor file and can later be used to send boundary values to whatever processor red corresponds to.

3.4 Overview Of Main Solver

In this thesis, the solver is built on iterative methods since it should deal with very large and complex problems, like the Navier-Stokes equations. Basic solvers such as the Jacobi and Gauss-Seidel are implemented as well as Multigrid. Both explicit and implicit equations should be solved and these are treated in much the same way, but are still separated due to the unnecessary residual calculation in the explicit case. The stencils are more extensive in a way that they describe the complete set of equations that are being solved and also depends on parameters such as the grid spacing and time step which may vary from one level to another. Multigrid also relies on this flexibility of the stencil, thus a separate section for this is given below. An overview sketch of the most important methods in the main solver is given in figure 3.8.

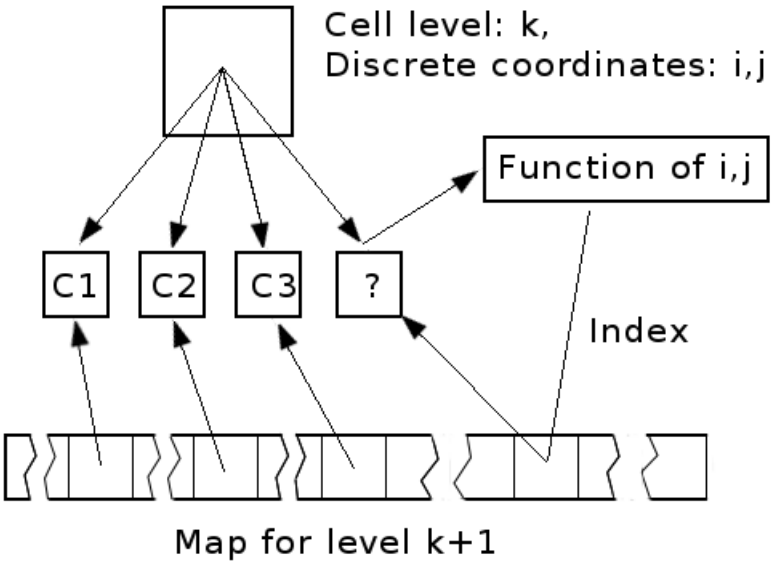

FIGURE 3.7: The mapping technique used to connect cubes with their children and parents that are read and inserted in a map. The function of the discrete coordinates i and j can be as simple as index=i*jmax+i where jmax is size of the domain in the j-direction. The same principle applies for three dimensions.

3.4.1 The stencils

The basic idea of the stencils is to update every cell inside a cube using either Jacobi, Gauss-Siedel or some other iterative method. If the underlying equation is explicit, there is no need to use these since then it's just a matter of overwriting every cell with values that are already known from previous time steps and/or variables. On the other hand, it might be a good idea to have a scheme for the different variables and time steps that are to be combined to a new value for a given cell, so that this value can be obtained for every time that equation is solved. For an implicit equation, the terms that don't belong to the new time-step could be collected by a similar scheme and saved in the source term. This is demonstrated by introducing a general equation in a single variable, called u.

$$u_N^{n+1} = \sum_{0<k<n} \sum_{l_k} \alpha_{l_k} u_{l_k}^{n+1-k} \tag{3.1}$$

The index N denotes the cell that is being solved for and l_k denoted the current neighbouring index to N at time-step $n+1-k$. The α_{l_k} is the weighting factor that should

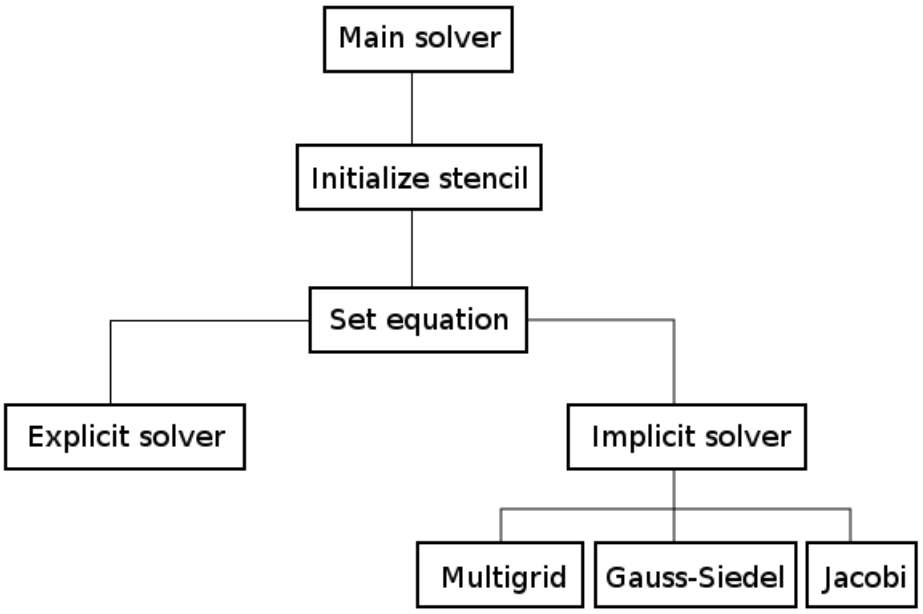

FIGURE 3.8: An overview sketch of the main solver.

be saved in the stencil. But α_{l_k} may depend both on the time-step and the grid-spacing, thus $\alpha_{l_k} = \alpha_{l_k}(\Delta t, \Delta x)$ where Δx is same in every direction of a cell. Now, Δt is not any problem if it is the same for all cubes, but as mentioned in section 2.4.2, the time-step may depend on the size of the cube. Also, the grid-spacing is different for different levels of refinement. To solve this problem in this thesis, a global stencil structure is implemented that keeps track of every weight for all possible combinations of parameters. The idea sketch of this structure can be seen in figure 3.9.

Both level and grid-spacing needs to be sent to the structure in order to get an unique stencil. This is because Multigrid uses stencils based on the same time-step Δt as defined for the coarsest grid. If solving a cube at a specified level with Multigrid, only the grid-spacing parameter should be changed when refining and coarsening, not the level which has Δt constant for all grid-spacings.

Looking at equation 2.2, the incompressible density ρ appears in the non-linear term. Since it may depend on the space coordinates, so will the weights w1-w6 in figure 3.9.

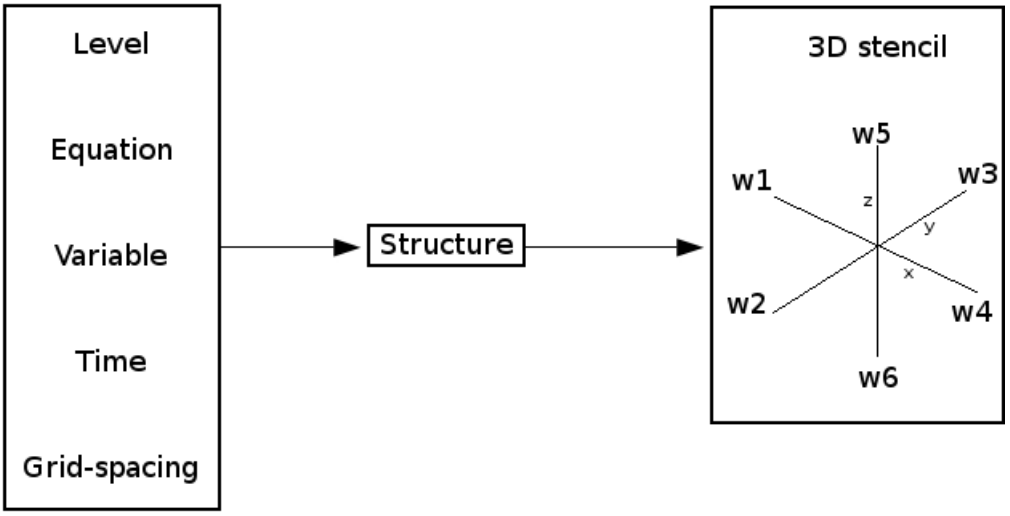

FIGURE 3.9: The input parameters to the left and the implemented structure in the middle yields a three dimensional stencil that can be used to solve the given equation.

Temporarily, in this thesis, a function for every weight-node in the stencil was implemented. Every node in the stencil structure gets a pointer to a function which returns the right weight for every space coordinate.

Chapter 4

Stability and Convergence

4.1 Solving the Wave Equation

There are many ways to test the implementation described in chapter 3. Here is given a description of how an explicit and an implicit form of the wave equation are implemented and tested by the solver. The wave equation has the following form

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 u}{\partial z^2} \right) + f \tag{4.1}$$

Here, c is the velocity of the wave, u the solution amplitude and f an external force. The equation is hyperbolic and has a condition for stability, namely the CFL-condition, see section 2.4.2. In the next section, a simple explicit scheme is introduced to solve equation 4.1 with a CFL-condition. After that, an implicit scheme is given that tries to avoid the CFL-condtion by separating variables and solving for an elliptic equation instead.

4.1.1 Explicit solution

A second order central difference scheme applied to equation 4.1 in both space and time, yields

$$u_{i,j,k}^{n+1} = 2u_{i,j,k}^{n} - u_{i,j,k}^{n-1} + \frac{c^2 \Delta t^2}{\Delta x^2} \left[D_{xx} u^n + D_{yy} u^n + D_{zz} u^n \right] + \Delta t^2 f_{i,j,k}^{n} \tag{4.2}$$

Here, $D_{\kappa\kappa}u^n = u^n_{\kappa+1} - 2u^n_\kappa + u^n_{\kappa-1}$, where κ is either the x,y or z index that is being differentiated. Figure 4.1 shows the result of this with a wave propagating from an initial pulse.

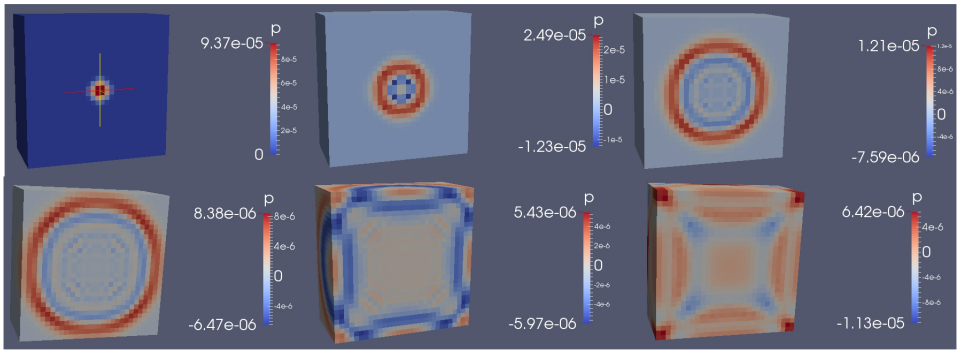

FIGURE 4.1: Explicit wave-equation solved on one cube block with size 1m, N=30 and Δt=1e-5. Boundary condition is zero and initial velocity and amplitude are both zero. A pulse is generated at time-step 1 at $(x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 < 0.005$ with amplitude 1e5. Upper row (left to right): time-step 10, 30 and 50. Lower row (left to right): time-step 140, 200 and 250.

4.1.2 Implicit solution

The implementation of an implicit solution is already done in [10] for the two dimensional case. A simple generalization to three dimensions yields the following equations.

$$v^{n+1} - \frac{\Delta t^2}{4\Delta x^2} D_{xx,yy,zz} v^{n+1} = \Delta t D_{xx,yy,zz} u^n + v^n + \frac{\Delta t^2}{4\Delta x^2} D_{xx,yy,zz} v^n \tag{4.3}$$

$$u^{n+1} = u^n + \Delta t \frac{v^{n+1} + v^n}{2} \tag{4.4}$$

Here, $D_{xx,yy,zz} = D_{xx} + D_{yy} + D_{zz}$, where $D_{\kappa\kappa}$ is defined as in the previous section. The variable v is coupled with the solution amplitude u as $D_t u = v$ which implies the second equation $D_t v = D_{xx,yy,zz} u$ from which equations 4.3 and 4.4 are found. This is as in the explicit case a second order central difference scheme in both time and space. The first one is actually Helmholtz equation which is always elliptic and does always converge. Results can be seen in figure 4.2.

A comparison between figure 4.1 and 4.2 gives a confidence that both the explicit and the implicit solver works.

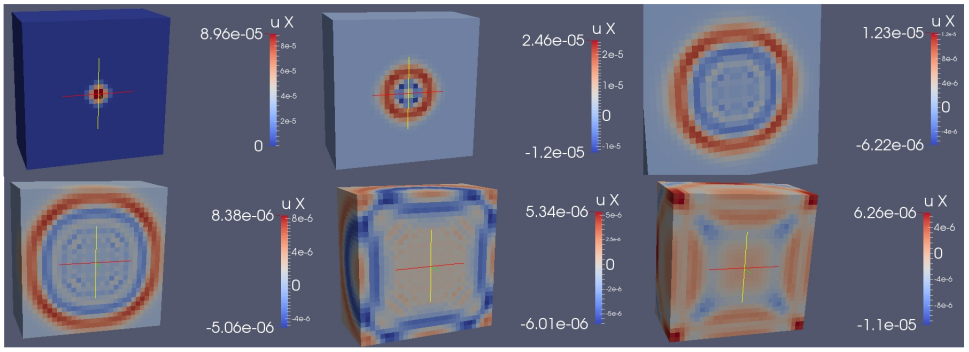

FIGURE 4.2: Implicit wave-equation solved on one cube block with size 1m, N=30 and Δt=1e-5. Boundary condition is zero and initial velocity and amplitude are both zero. A pulse is generated at time-step 1 at $(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 < 0.005$ with amplitude 1e5. Upper row (left to right): time-step 10, 30 and 50. Lower row (left to right): time-step 140, 200 and 250.

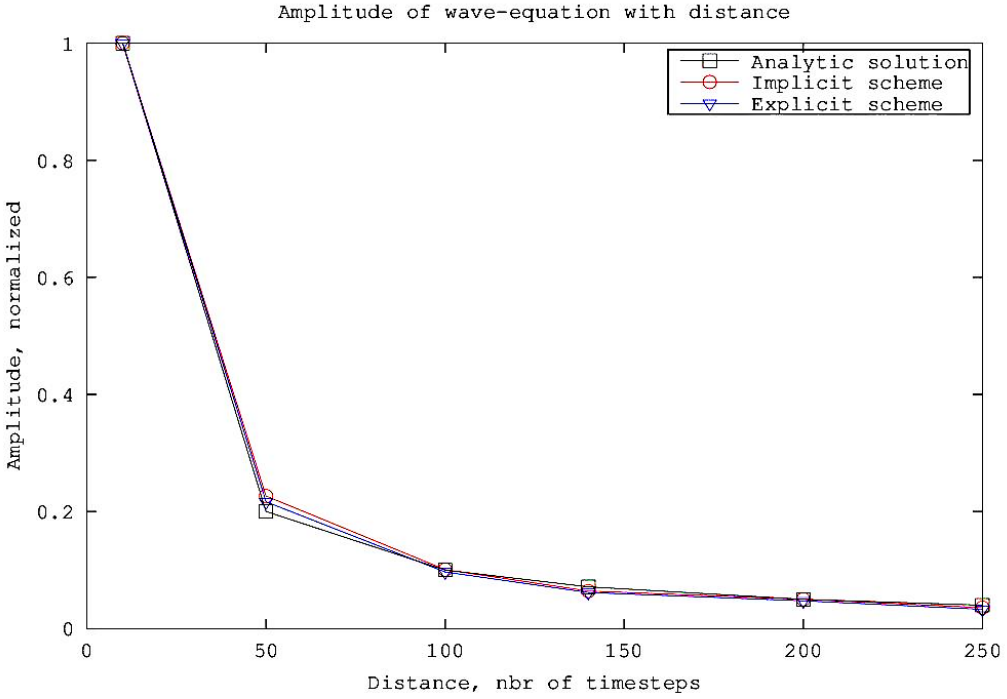

FIGURE 4.3: Comparison of solutions from explicit solver and implicit solver according to figure 4.1 and 4.2.

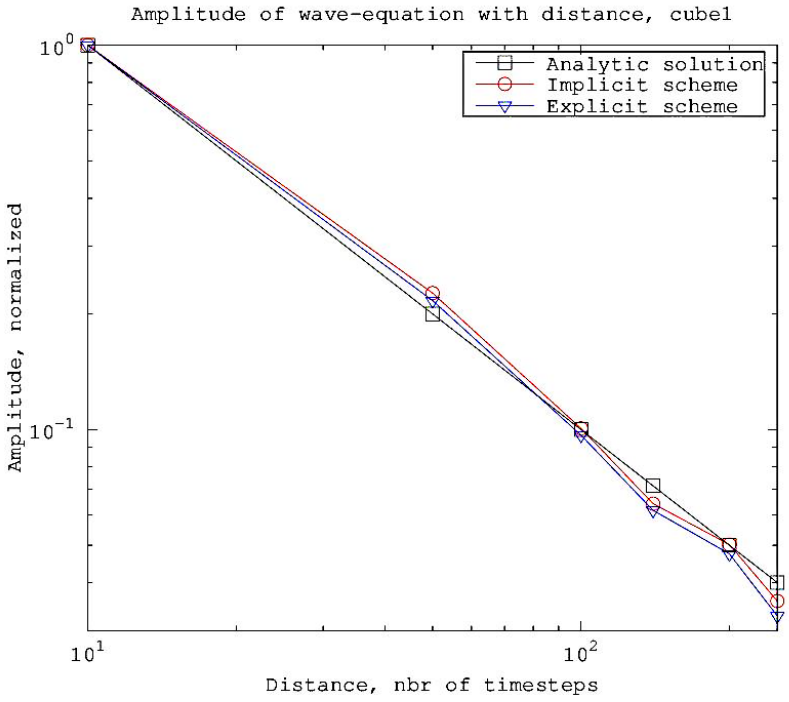

FIGURE 4.4: Comparison of solutions from explicit solver and implicit solver according to figure 4.1 and 4.2 in loglog-scale.

Chapter 5

Accuracy And Efficiency

5.1 Solving Poisson's Equation

One of the most common differential equation is the Poisson's equation. Together with some oscillating force, the equation is

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = -p^2 sin(px) \tag{5.1}$$

The equation above is in only one dimension because of the right hand side force, but it will be solved on a unit cube in three dimensions. The solution to equation 5.1 is $f(x, y, z) = sin(px)$ and this is used to set the boundary conditions.

The implementation of equation 5.1 is now tested on two different cube configurations. The first configuration is a single cube with N=30 cells along each direction. The second configuration consists of 3x3x3 cubes at the root level, while the middle cube has 8 children on level 2, N=10 for all of these cubes. The solution on these configurations are interesting because its possible to see if and how well communication between cubes work. Since the discretization of equation 5.1 is of second order, the error is supposed to behave as such when the size of the cells narrows down. Figure 5.1 and 5.2 shows the solutions with the corresponding errors for the given configurations.

As can be seen, both configurations seems to produce results that more or less equals the analytic solution. The single cube performs best and has a smooth solution with

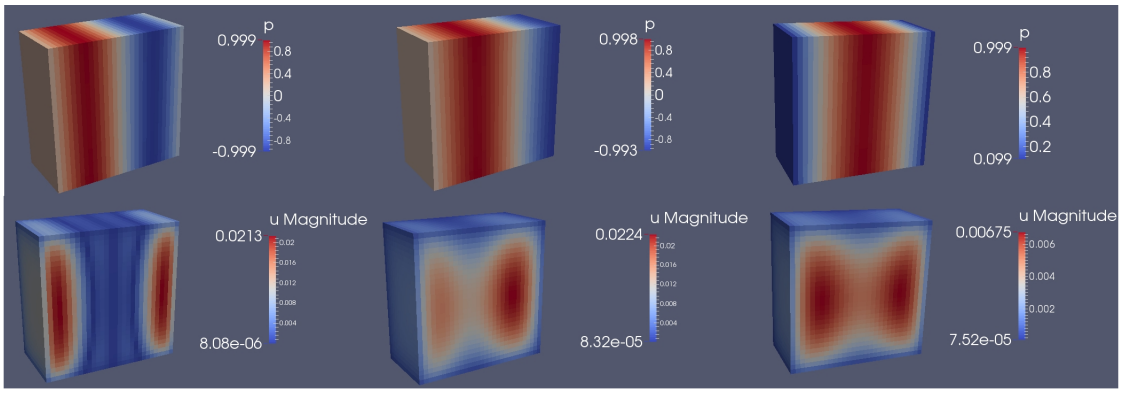

FIGURE 5.1: Solutions (upper row) and errors (lower row) of the Poisson's equation 5.1 on one single cube at the root level with N=30 cells along each direction. Iterations were done until residual < 1e-7. Results are shown for different values of p. Left to right: p = 2π, $\frac{3}{2}\pi$ and π. The cubes are cut at z=0.5 and thus shows the interface of a x-y plane.

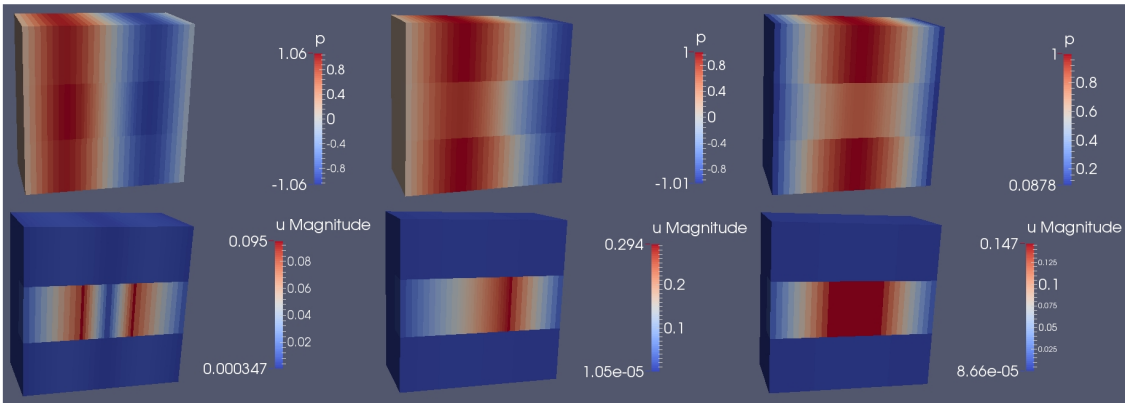

FIGURE 5.2: Solutions (upper row) and errors (lower row) of the Poisson's equation 5.1 with 3x3x3 cubes at root level and the middle cube refined to level 2. All cubes have N=10 cells along each direction. Iterations were done until residual < 1e-7. Results are shown for different values of p. Left to right: p = 2π, $\frac{3}{2}\pi$ and π. The cubes are cut at z=0.5 and thus shows the interface of a x-y plane.

no visible distortion of any kind. On the other hand, the configuration in figure 5.2 seems to get distorted around the communicating area between large and small cubes. Looking at the magnitude of the error, it is about 10 times greater than the error of the single cube. The problem must come from the interpolation done when updating the boundary of the smaller cubes, from large to small cells. The error seems to be smallest for p=2π and greatest for p=$\frac{3}{2}\pi$. Also, the error is moved to the right side of the inner cube for p=$\frac{3}{2}\pi$. This is explained in figure 5.3.

From the plot of the single cube in figure 5.4 it's clear that the distretisation is of second order. The other plots shows how the interpolation (of first order) between large and

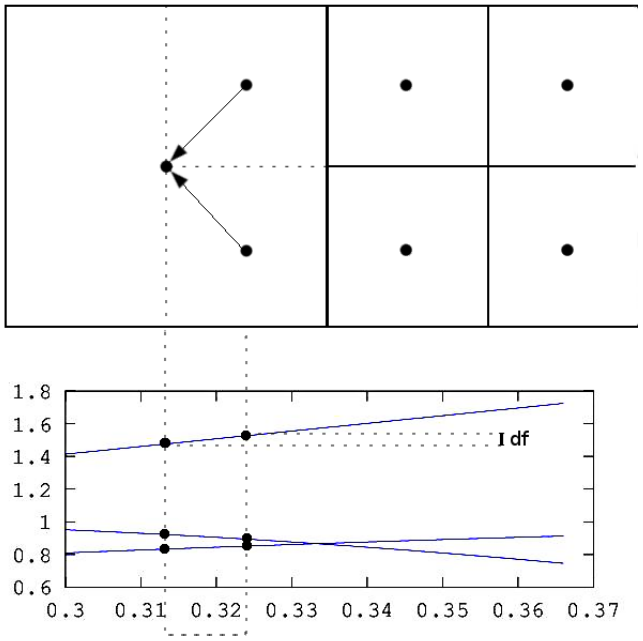

FIGURE 5.3: The problem of communication from large to smaller cells. Interpolation of the first order like in this figure, can only use values that lie further from the point in interest in one direction. This means that the functional value changes an amount df depending on the gradient of the source term function. In this case three functions are plotted to mimic the situation in figure 5.2.

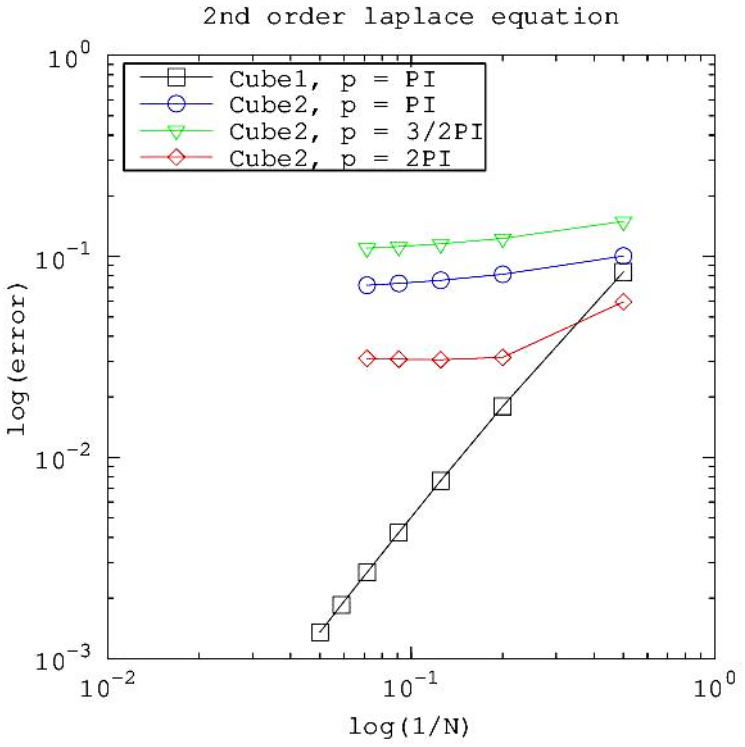

FIGURE 5.4: Shows how the error reduces as the size of the cells narrows down. Cube 1 represents the single cube configuration, while Cube 2 represents the 3x3x3 cubes with the middle cube refined to level 2.

small cells introduces an even greater error than this second order discretisation error. These errors will decrease more or less in an linear rate, depending on the gradients of the source term.

5.2 Solving Navier-Stokes Equations

5.2.1 Stokes flow

Here the solution of Stokes flow around a sphere is studied and compared with the analytic solution.

The equations 2.3 and 2.4 are straight forward to discretize. The corresponding equation for the pressure is just the Laplace equation, since the non-linear term is zero. According to [11] the velocities and pressure around a sphere in polar coordinates are

$$u_r = W cos(\theta)\left(1 + \frac{a^3}{2r^3} - \frac{3a}{2r}\right) \tag{5.2}$$

$$u_\theta = -W sin(\theta)\left(1 - \frac{a^3}{4r^3} - \frac{3a}{4r}\right) \tag{5.3}$$

$$p = p_0 - \frac{3}{2}\frac{\mu W a}{r^3} cos(\theta) \tag{5.4}$$

Here, a is the radius of the sphere, W the velocity in z-direction, p_0 the pressure far away from the sphere and μ the viscosity. Now, introduce the angle ϕ in the x-y plane and let (x_c, y_c, z_c) be the origin of the sphere. Then the corresponding velocity equations in Cartesian coordinates becomes

$$u_x = cos(\phi)\left(u_r sin(\theta) + u_\theta cos(\theta)\right) \tag{5.5}$$

$$u_y = sin(\phi)\left(u_r sin(\theta) + u_\theta cos(\theta)\right) \tag{5.6}$$

$$u_z = u_r cos(\theta) - u_\theta sin(\theta) \tag{5.7}$$

Here, $cos(\theta) = \frac{z-z_c}{r}$, $sin(\theta) = \frac{y-y_c}{r}$, $cos(\phi) = \frac{x-x_c}{r_{xy}}$ and $sin(\phi) = \frac{y-y_c}{r_{xy}}$, where $r_{xy} = \sqrt{(x-x_c)^2 + (y-y_c)^2}$.

The above has been implemented on a staggered grid and solved for a fix spacing Δx on a domain consisting of a sphere object of radius 2 m. The performance was thereafter compared between Multigrid and Gauss-Siedel for different number of cubes covering the domain.

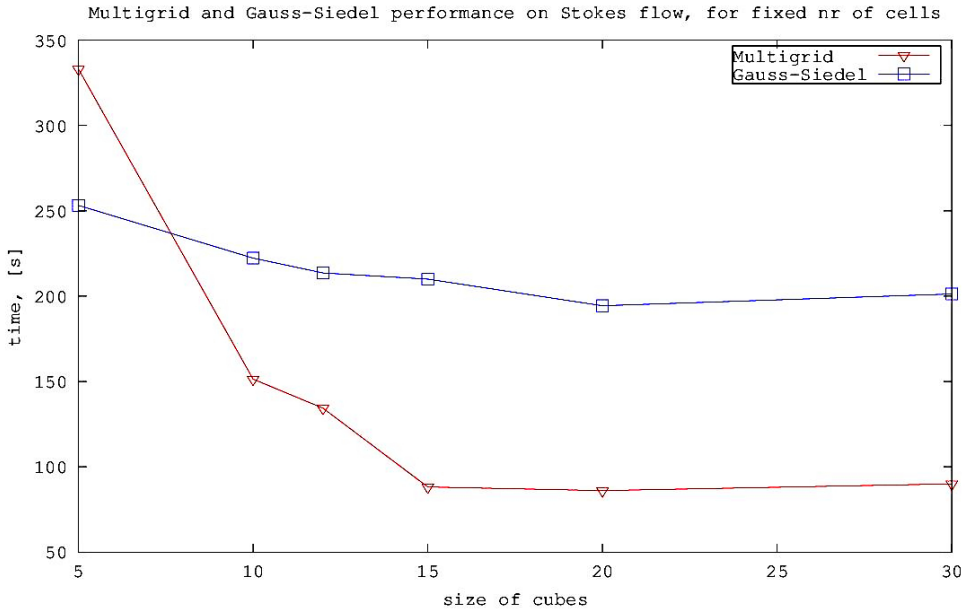

FIGURE 5.5: The comparison between Multigrid and Gauss-Siedel on a domain consisting of a varying number of cubes, but fixed number of cells over the whole domain, hence a fixed Δx. The whole domain is 60x60x60 cells. The x-axis is just the N-value (number of cells along each cube), while the y-axis is the time it takes to converge to an error of 1e-3. Multigrid is set to do 5 sweeps at base level N=5, why it takes longer for that cube size.

As can be seen in figure 5.5, the Multigrid method performs better, at least for sufficiently large values of N. Both Gauss-Siedel and Multigrid performs less when N decreases, due to the number of communications between cubes then increases. But Multigrid has a much higher gradient on that curve since it also depends directly on N. Having many cubes along the domain doesn't benefit Multigrid, because it simply cannot transfer information faster than a cube length per iteration. So, to maximize the performance, Multigrid should be used when N is sufficiently large and on a cube that is large compared to the size of the domain.

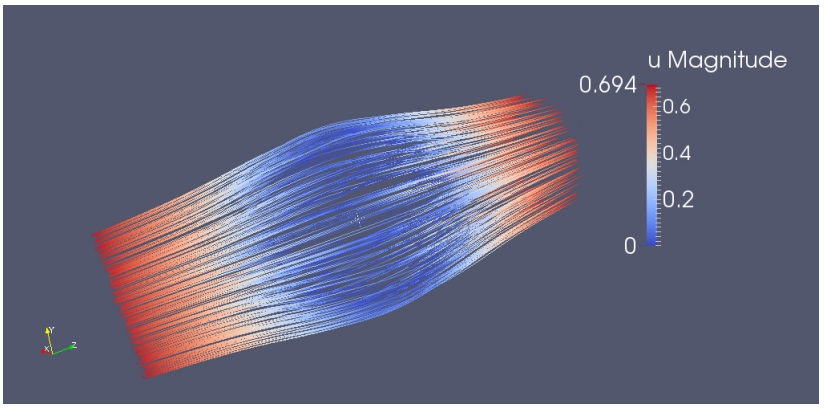

FIGURE 5.6: Result of Stokes flow on 3x3x3 cube domain, N=20. Domain size: 10m x 10m x 10m. Radius of sphere: 2m. Inlet velocity is 1 m/s and with viscosity set to 1. The pressure far away is set to 0.

5.2.2 Discretize Navier-Stokes Equations

Looking back at the discretized momentum equation 2.9 and its corresponding continuous equation 2.2, it is noticeable that there is a problem with the non-linear term $\frac{\partial \rho u_i u_j}{\partial x_j}$. The momentum equation cannot directly be written in the linear form of equation 2.9 without doing something about this non-linear term. One possibility is to use some non-linear solver like the Newton-Raphson method. But in this thesis, this is solved by linearizing the non-linear term. This is done in the same way as described in [9]. The linearized momentum equation becomes

$$\rho \Delta u_i = \Delta t \left(-\frac{\delta \left(\rho u_i u_j\right)^n}{\delta x_j} - \frac{\delta \left(\rho u_i^n \Delta u_j\right)}{\delta x_j} - \frac{\delta \left(\rho \Delta u_i u_j^n\right)}{\delta x_j} - \frac{\delta p^n}{\delta x_i} + \frac{\delta \Delta p}{\delta x_i} + \frac{\tau_{ij}^n}{\delta x_j} + \frac{\Delta \tau_{ij}}{\delta x_j} \right) \tag{5.8}$$

Here, $\Delta u_i = u_i^{n+1} - u_i^n$. From this, the Poisson's equation can be found as

$$\frac{\delta}{\delta x_i}\left(\frac{\delta \Delta p}{\delta x_i}\right) = \frac{1}{\Delta t}\frac{\delta (\rho u_i^{m*})}{\delta x_i} \tag{5.9}$$

The asterisk sign over u_i^m indicates that this is the current velocity field computed by equation 5.8. The corrector equation, corresponding to equation 2.12 now becomes

$$u_i^{m+1} = u_i^{m*} - \frac{\Delta t}{\rho}\frac{\delta \Delta p}{\delta x_i} \tag{5.10}$$

The implementation of this is tested in the next section.

5.3 Comparison With OpenFOAM

A very basic problem in CFD is the cavity problem, where fluid is flowing over a cubic cavity such that a swirl occurs inside the cavity. This was implemented according to the PISO algorithm, see section 2.6. But due to some inconsistency in the solution algorithm, no numeric comparison with openFOAM could be done. Still, the current solution is compared in figure 5.7, but just for qualitative comparison.

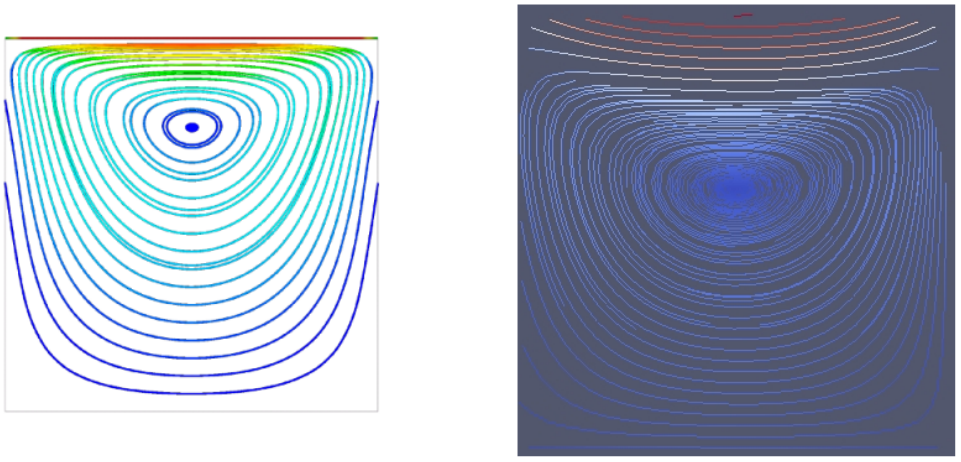

FIGURE 5.7: Left: openFOAM solved streamlines for the cavity case. Right: Implemented PISO algorithm solved cavity case with similar settings as in openFOAM.

5.4 Speed-up

Instead of further comparisons, a speed-up test was performed on the stokes flow. The results can be seen in figure 5.8

FIGURE 5.8: Speed-up test. P1-P64 indicates the number of processors and the first letter A,B or C correspond to different configurations. The time is normalized to the total number of cells. There are 3 configurations in the graph. The first configuration (A) is just a single cube. The second configuration (B) is a cube refined to level 2, thus 2x2x2 cubes. The last configuration (C) is a cube refined to level 3, thus 4x4x4 cubes. Nv is the number of cells along the total size of the domain in one direction. Gauss-Seidel was used in the calculations.

Chapter 6

Conclusions and Future Work

One of the main goals of this thesis was to get a working implementation of a parallel CFD program using the BCM method. This was done and is described in chapter 3. There are always alternative ways to implement different parts of a program, but this was simplified by following the already implemented structure and the description of the BCM method in [2], [1] and [6].

The decomposition of cubes as done in [1] is a little bit more complex then the way it was done in this thesis. In this thesis, cubes on the border to other processors were assigned a direction index which tells in which direction it should update its boundary in the other processor, which it belongs to. Further, maps where used to keep track of cube pointers with different Hilbert- or flat indices. There is a somewhat similar thinking in [1], which mentions window cells and halo cells as communicating cells over different processor domains. Still, an important thing when efficiency is required, is that the boundaries can be sent from one processor to another as fast as possible. Here, the implemented maps became very useful since they provided a way to directly connect to the cube which should be updated, by using the cube's index and level as an id.

The implementation of the parallel parts using OpenMPI seems to give good scaling as can be seen in the speed-up test, figure 5.8. When N=40 on 64 processors, the scaling gets worse, but this is just 1000 cells per processor. The larger number of cells per processor, the more time is spent on solving and less on communication. This should yield a curve that is linear for an even more processors, which also seems to be the case.

Even though some parts of the implementation works as intended, many things has to be further developed. For example, a drawback from just having one layer of boundary cells is that the accuracy gets worse than the accuracy expected from the discretization, when large and small cubes communicate. This was illustrated in figure 5.3. Cubes that are refined close to objects often have some interesting flows with high gradients. Here it is especially important to have a good transition from large to small cubes. To get satisfying results in a further developed program, higher order boundary cells need to be implemented.

The implemented stencils did contribute with another weak point to the implementation and are far away from ideal. The most time-consuming part when solving with a stencil with space dependent weights, is the repeated calls to weight functions. A run under gprof shows that one such function took about 5 percent of the total runtime, while another only took about 0.1 percent. The preferable way would be to calculate every weight before entering the solver.

A last objective of this thesis was to compare the implementation with some open source software, like OpenFOAM. The implemented PISO algorithm was able to solve the cavity case with some high tolerance on the residual. But due to some inconsistency causing the convergence problem, it did not work good enough for any further comparison. This is considered as something to be further developed and tested in the future.

Bibliography

[1] J.H. Grimmen C. Günther M. Meinke W. Schröder A. Lintermann, S. Schlimpert. Massively parallel grid generation on hpc systems. *Computer Methods in Applied Mechanics and Engineering*, pages 1–32, April 2014. URL http://dx.doi.org/10.1016/j.cma.2014.04.009.

[2] Kazuhiro Nakahashi. Aeronautical cfd in the age of petaflops-scale computing: From unstructured to cartesian meshes. *European Journal of Mechanics B/Fluids*, pages 75–86, February 2013. URL http://www.elsevier.com/locate/ejmflu.

[3] Finite difference method. http://en.wikipedia.org/wiki/Finite_difference_method, . Accessed: 2015-03-03.

[4] H. K. Versteeg and W. Malalasekera. *An introduction to computional fluid dynamics The finite volume method*, pages 136–138. Longman Scientific and Technical, 1995.

[5] Open MPI open source high performance computing. http://www.open-mpi.org/. Accessed: 2015-03-03.

[6] Daisuke Sasaki Kazuhiro Nakahashi and Noriyoshi Ishikawa. Large-scale distributed computation using building-cube method. Technical Report AIAA 2011-754, American Institute of Aeronautics and Astronautics, 2011.

[7] Courant-friedrichs-lewy condition. http://en.wikipedia.org/wiki/Courant-Friedrichs-Lewy_condition, . Accessed: 2015-03-03.

[8] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*, pages 291–308. Cambridge University Press, second edition, 2009.

[9] Joel H. Ferziger and Milovan Perić. *Computational Methods for Fluid Dynamics*, pages 170–178. Springer, third edition, 2002.

[10] Indroduction to multigrid methods - final project. http://www.maths.lth.se/na/courses/FMNN15/media/material/computerXPMG3.11.pdf. Accessed: 2015-03-03.

[11] Chiang C.Mei. Stokes flow past a sphere. http://www.web.mit.edu/2.21/www/Lec-notes/chap2_slow/2-5Stokes.pdf, 2007. Accessed: 2015-03-03.