A high performance semi lagrangian fluid solver

by

Carlos Alejandro Roig Pina

Submitted to the Department of Computer Architecture AC
in partial fulfillment of the requirements for the degree of

Master in research and innovation in informatics MIRI

at the

UNIVERSITAT POLITECNICA DE CATALUNYA

June 2015

Author .....
Department of Computer Architecture AC
Octover 13, 2015

Certified by.....
Pooyan Dadvand
Associate Professor
Thesis Supervisor

Certified by.....
Marisa Gil
Associate Professor
Thesis Supervisor

A high performance semi lagrangian fluid solver

by

Carlos Alejandro Roig Pina

Submitted to the Department of Computer Architecture AC
on Octover 13, 2015, in partial fulfillment of the
requirements for the degree of
Master in research and innovation in informatics MIRI

Abstract

Incompressible fluid simulations are widely used by the industry in a great deal of applications such as wind tunnel tests or flow studies. The main objective of these simulations is to achieve the greatest possible precision in the smaller possible time. This arise a problem since more accurate results means that the granularity of the problem has to decrease and as a consequence the simulation time is increased. In this dissertation we introduce a new implementation of a well-known already existing algorithm, the Back and forth error compensation and correction (BFECC), that in some scenarios is able to increase the accuracy of the results while not sacrificing performance in the process.

Thesis Supervisor: Pooyan Dadvand
Title: Associate Professor

Thesis Supervisor: Marisa Gil
Title: Associate Professor

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n 611636 (NUMEXAS).

I would like to extend my sincerest thanks and appreciation to all the people who has helped me during the development of this project, both in CIMNE and out of it, and specially to Marisa Gil, Riccardo Rossi, Pablo Becker and of course Pooyan Dadvand who has always had time to give me all the help I could have need and more.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

Incompressible fluid simulations are widely used by the industry and have proven to deliver successful results in a great variety of problems such as wind tunnel simulations. Among the methods to perform such simulations we can find techniques like the Finite volume method (FVM), Finite difference method (FDM), Finite element method (FEM), and Lattice Boltzmann (LB). In this dissertation we focus our studies in FEM[1] and FDM[2] methods which makes use of the Navier-Stokes equations[3].

We define a simulation as the execution of a series of loops over a matrix that contains the data of the studied case. A series of operations are executed at each iteration, in our case this is consists two operations: the advection and the diffusion. The bigger the matrix grows, the longer it takes to finish each of the iterations. Additionally, the size of the matrix also has a direct impact in the number of the iterations and the precision we achieve. As more accuracy is expected in the results, bigger matrices and more iterations are needed. This growth in size increments the execution time by roughly an order of magnitude (2^3 to 2^4) each time the size of the matrix is doubled.

There are several procedures that allow us to reduce the growth in size caused by the increment of iterations by changing some of the algorithms used to solve the operations specially in the advection operation but these are typically slower and hence do not represent a real solution to the problem. The BFECC[4] is an

example of a method which allows us to estimate part of the error and remove from the result.

1.1 The objective

Our work is part of a contribution to NUMEXAS project. Our part consist in creating a solver based on FDM that increases the accuracy of our simulation without increasing its execution time. To accomplish this we explore a new BFECC implementation which under some specific scenarios will allow us to introduce further optimizations the code to decrease the penalty hit in performance.

Specifically, the focus of the optimizations will be put into solve the zones of the simulation where no contour conditions are present. Here two characteristics of the BFECC algorithm are going to be key: First, the fact that the greatest performance penalty is caused by the search of neighbor elements while interpolating the data, and secondly, the fact that the absence of contour conditions allow us to use a representation of the matrix which would be otherwise prohibitive in terms of size and which enables the use different parallelization scheme some of which will prove to be easily maintained and expanded in a future.

Chapter 2

Technology overview

As this dissertation combines parts from both numerical simulation and HPC fields many new concepts will be introduced in the document. In this section we provide a glimpse of the most relevant topics that will be of interest during the development of the document. Especially we focus in giving an introduction of the different memory models that exist in the present, the different parallelization techniques in SMP (Symmetric Multiprocessing) environments and a final overview of the different ways to represent our mesh and the formulation that will be used in the solver.

2.1 Memory models

We call memory model the way in which the memory layout of a system is designed. This design has direct implications in the performance of the applications that have to run in the machine. This layout is normally represented by hierarchical levels. The memory that is close to a processor unit is typically faster and smaller and as we introduce lager memory this decreases in performance. When there are more than one processor in our system, such as a modern CPU with multiples cores or a GPU, higher levels of memory are typically shared between the processors at multiple levels creating different memory schemes. Among the most important ones we can difference SMP, MPP, NUMA, and Clustering.

SMP stands for Symmetric Multiprocessing. In this memory scheme all processors have its own low level cache memory but share the same large memory. This means that each time that any the processors needs a data it needs to go to the main memory and bring it to its own low level memory. These copies also need to guarantee the consistency of the memory at some points and typically require additional protocols to control how the data is accessed that need to be implemented by the system. These additional protocols normally have a great performance penalty when the system grows in size.

In architectures based on MPP (massive multiprocessing) on the other side each processor has its own memory resources. Hence there is no control over by the side of the system over what happens in the memory, but is responsibility of the user to keep the coherence of the data. As there is no protocols in-between, there is no penalty for adding more processors to the system. This systems are, however, very expensive.

In the middle we found the NUMA (non-uniform memory access). If in SMP all processors access to a single main memory, in NUMA this main memory is distributed among the processor but the combined memory of all CPU is view as a single memory space. This type of memory also makes use of consistency protocols and presents an asymmetry in the access data speed, for example, a given CPU1 would access faster a data in allocated in its physical memory than data located in the physical memory of CPU2.

Clusters are a way to get the most interesting characteristics of all systems. These are formed of nodes, which can be considered as a stand-alone computers, and a set of these nodes act as a MPP model being completely independent of each other. Inside a node, we often find different NUMA contexts which are the same time can be implemented using SMP. In figure 2-1 it can be seen some examples of the layouts commented.

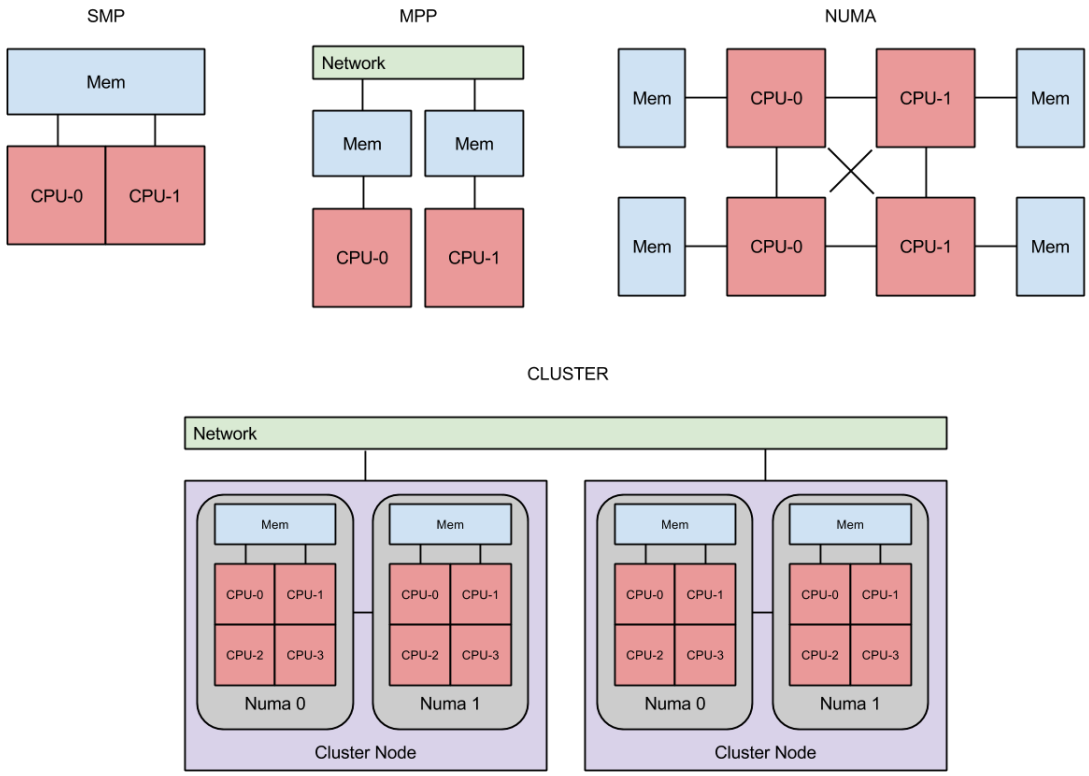

Figure 2-1: Different types of memory models, from top to down and left to right: SMP, MPP, NUMA and Clusters

2.2 OpenMP

Machines that use a shared memory model allow the user to ignore the interconnection details of parallel machines and exploit the use of parallelism with a minimal effort. In order to exploit such parallelism different programing models can be used. These models are often extensions of existing programming languages that allow the parallelization by adding some directives or annotations in the code. OpenMP [5] is one of such set of directives that allow the automatic generation of parallel code which extends the runtime of C, C++ and FORTRAN.

The execution model of OpenMP relies on creating and merging threads which is also known as fork-join model. A normal OpenMP program starts its execution with a single thread which often is known as the master thread. Once a parallel directive is found, the execution thread is divided in as much as threads as indicated until it reaches the end of the parallel section. OpenMP is easy to use in loops which can be almost automatically parallelized with a single directive with no further actions required

OpenMP also allow the parallelization of the code based on tasks. In this case instead of parallelizing sections of the code, one indicates that some portions of the code, typically functions, can be executed in parallel and the data dependences among this functions which allow the runtime to create a dependency tree and parallelize the code using a different approach. In figure as can be 2-2 we can see two different approaches to perform an operation over an array using loops (left) and tasks (right)

Either using loop or task parallelization, OpenMP offers a series of directives to control the behavior of the code executed. There are also some other directives that allow the user to override this behavior and add, for example, synchronization points. OpenMP also have a set of in-built functions that can query and set parameters like the number of processors running or the current thread-ID among others. A set of directives to control the execution flow and implement

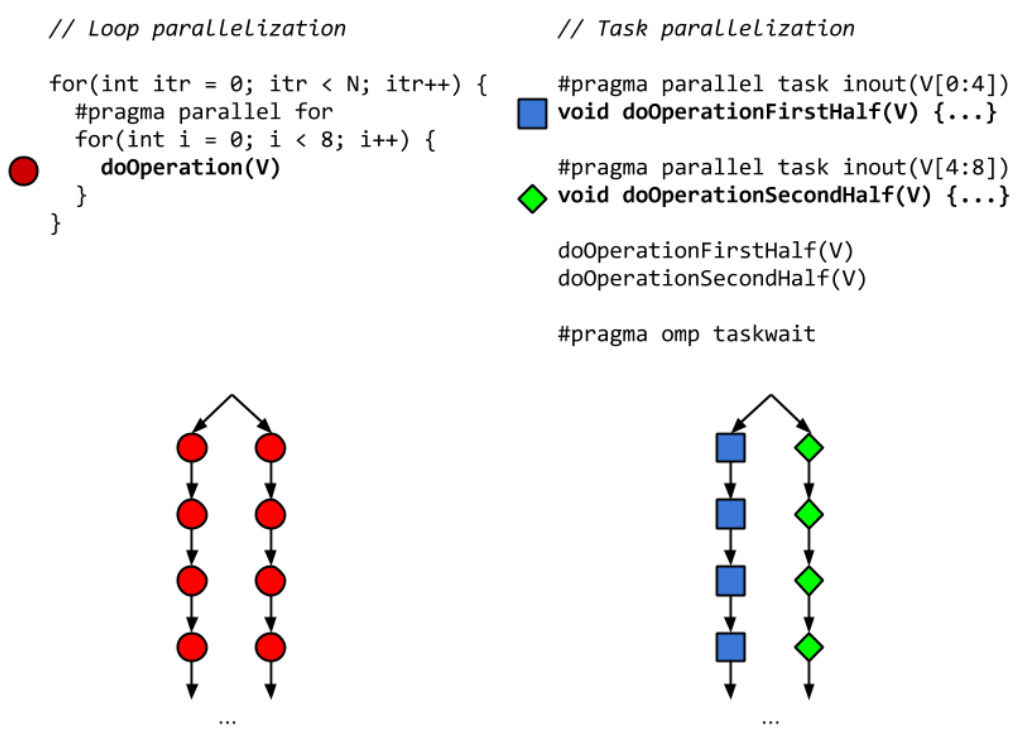

Figure 2-2: Left: Omp loop parallelization, Right: Omp Task parallelization

atomic operations are also present and finally OpenMP also offers different ways to control the load balance of the program.

2.3 OmpSs

OmpSs is a programing model developed by the BSC which integrated the features of the old StarSs programing model also developed by the BSC. OmpSs works as an extension of OpenMP and expands it with new task directives that enable support for asynchronous parallelism and automatic heterogeneity support for languages like CUDA, OpenCL and other API's, even MPI. OmpSs environment is built on top of the Mercurium compiler and Nanos++ runtime system.

These features, are achieved by the use of data-dependencies between the different tasks that are defined in the program and the use of constructs, an implementations of the same tasks able to run in other devices. In OmpSs both task a constructs can be annotated in its declaration and each call to such functions become a task creation point. There is a restriction that does not allow these functions to have any return value which means that every data transfers needs to happen in the arguments of the function.

There are three main ways to express the new data dependencies that appear in OmpSs: the in, out and inout clauses (which are equivalent to the depend(in) depend(out) and depend(inout) directives in OpenMP). as they name suggest each one of this clauses allow the user to specify which data is needed before executing the task and which data is going to be obtained when it has finished, but it does not check if the data is really used or generated in the function which is in the last instance responsibility of the programmer. Each time the runtime of the program reaches a task it is created and its dependencies analyzed and added to the scheduler which will execute the task immediately if possible.

All dependency clauses allow extended values from those of C/C++. Two different extensions are allowed: array sections allow referring to multiple elements of an array (or pointe data) in single expression and shaping expressions allow to recast pointers into recover the size of dimensions that could have been lost across calls.

Finally OmpSs also defines the ”taskwait” construct and the ”taskwait on” which are the equivalents to the barrier clauses in regular OpenMP and taskwait clauses in OpenMP 4.0 and allow us to wait all tasks or just the tasks that meet the data conditions, leaving the others being executed without any restriction.

2.4 Mesh and representation

In fluid simulations we call mesh to the basic data structure used to represent our real model in the computer in a way that can be manipulated by the simulation algorithm. A mesh is formed by elements, which are the same time are formed by nodes. In general, the more detailed the model is needed, the more elements and nodes the mesh will need to have.

There are different methods that allow us to classify this meshes among different groups based on its properties. In our context it is interesting to separate the meshes by the organization and connectivity of the nodes. In this context we distinguish three different types of meshes:

1. **Structured mesh**: This first group, as depicted in 2-3-A is characterized for having all its elements equal and being these distributed evenly in the mesh. This is proven to be a very interesting characteristic in order to efficiently store the mesh in memory and also cause that some important operations like the refinement or the value interpolation become trivial. The backside of this meshes is that the amount of elements needed to model a contour can be prohibitively high in terms of number of elements needed. We distinguish a specific case of structured mesh, the Euclidean grid, in which the model has been decomposed using regular square elements.

2. **Unstructured grid**: The second set of meshes is characterized for not having a fixed connectivity between nodes. Computationally speaking, this makes these meshes hard to represent in memory as a variable number of elements can share a variable number of nodes. This is the less restrictive type of grid for simulation and is widely used since it adapts very well to some very popular solving methods like FEM. As a backside, this type of meshes have more complex refinement algorithms but are optimal in terms of number of elements, as new ones are created only where are needed.

3. **Hybrid grids**: This last family of meshes take parts of both structured and unstructured meshes. The rule here is to use an unstructured approach

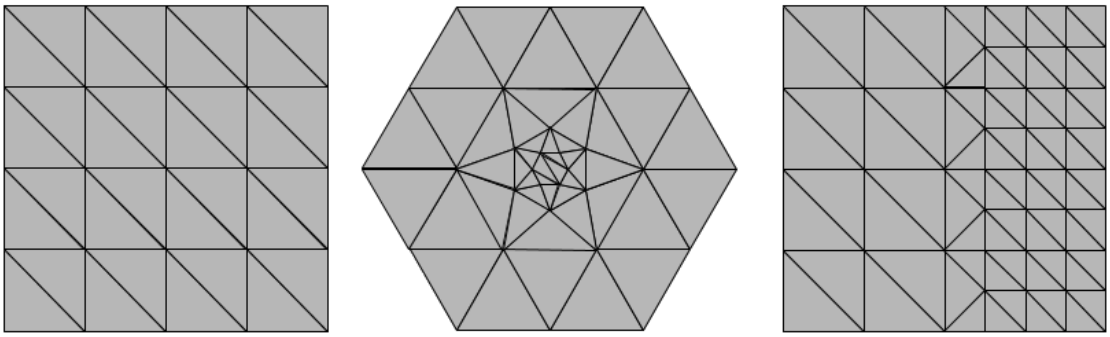

Figure 2-3: From left to right: (A) Structured mesh, (B) Unstructured mesh and (C) Hybrid mesh

for the regions of the model with more details, that will need to be refined, and a structured approach for the rest of the model, that need less detail but can handle a bigger error in the result.

2.5 Numerical methods

There are many different approaches in order to drive incompressible fluid studies, we call them numerical methods. Generally speaking these methods are, at the same based on the Navier-Strokes equations or the Lattice Boltzamn equations. The most common of this methods are the ones known as the Finite volume Method (FVM), the finite element method (FEM) and the finite difference method (FDM). Other techniques exist like the Lattice Boltzammn (LB) method, but they are not based on the Navier-stokes formulation. This formulation, which was developed by Claude-Louis Navier and George Gabriel Stokes define a series of derivative equations to describe the movement of the fluid and have the property that, expect in some concrete situations, does not have an analytical solution.

The FVM is probably the most popular choice for fluid problems, but is less used in structural problems. Since the final goal of the project in which this dissertation is bases is the fluid and structure interaction, FEM becomes the most interesting choice. All methods above mentioned present the characteristic of

being compatible with structured grids, some of them, especially the LB does not support the use of unstructured grids, or like behave with a decreased performance, like the FVM

Interpolation is defined as a method of constructing new data in the range of a discrete set of known data. This will means basically that we are going to create new data from the data that we already know. As this data is not real this is going to introduce an error. For example figure X shows how a possible interpolation of the points of a exponential function where a simple $1/2 * n - 1 + 1/2 * n + 1$ criteria has been applied to calculate the value of the point between the known values, and the error that it's being introduced.

There are several techniques that can be applied in order to reduce this error. One common technic to improve the precision of the interpolation is increasing the order, or how many values are we considering choosing while obtaining our results.

2.6 Fractional Solvers

The Navier-Stokes equation for incompressible fluid can be written in following form:

$$\rho\frac{\delta u}{\delta t} + \rho(u.\nabla)u + \nabla p - \mu\nabla.(\frac{\nabla u + \nabla u^T}{2}) = \rho b \ \nabla.u = 0$$

Applying discretization and Galerkin method [] the above equation can be deduced to following matrix form:

$$\mathbf{M}\frac{\delta u}{\delta t} + \mathbf{C}(u)u + \mathbf{G}(p) - \mathbf{L}(u) = f \ \mathbf{D}(u) = 0$$

Which leads to following system of linear equations []:

$$\begin{bmatrix} \mathbf{K} & \mathbf{G} \\ \mathbf{D} & \mathbf{S} \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} r_u \\ r_p \end{bmatrix}$$

This monolithic system of equations can be used to solve at once the pressure and velocities in the model. It can be easily verified that this leads to a system of 4n equations where n is the number of grid Points. Considering for simplicity a Backward Euler scheme one can write the matrix form equation as follow:

$$\mathbf{M}\frac{u_{n+1}-u_n}{\delta t} + \mathbf{C}(u_{n+1})u_{n+1} + \mathbf{G}p_{n+1-\mathbf{L}(u)} = f$$

$$\mathbf{D}(u) = 0$$

The first equation can be developed as:

$$\mathbf{M}\frac{u_{n+1}-\hat{u}}{\delta t} + \mathbf{M}\frac{\hat{u}-u_n}{\delta t} + \mathbf{C}(u_{n+1})u_{n+1} + \mathbf{G}p_n + \mathbf{G}(p_{n+1}-p_n) - \mathbf{L}(u) = f$$

$$\mathbf{D}(u) = 0$$

Which, even being more restrictive than the previous equation, can be split to:

$$\mathbf{M}\frac{\hat{u}-u_n}{\delta t} + \mathbf{C}(\hat{u})\hat{u} + \mathbf{G}p_n - \mathbf{L}(u) = f$$

$$\mathbf{M}\frac{u_{n+1}-\hat{u}}{\delta t} + \mathbf{G}(p_{n+1}-p_n) = 0$$

The incompressibility equation can be used to de couple the velocity and pressure and arrive to following equations:

$$\mathbf{M}\frac{\hat{u}-u_n}{\delta t} + \mathbf{C}(\hat{u})\hat{u} + \mathbf{G}p_n - \mathbf{L}(u) = f$$

$$\Delta t L(p_{n+1}-p_n) = Du$$

$$u_{n+1} = \hat{u} - \Delta t \mathbf{M}^{-1}\mathbf{G}(p_{n+1}-p_n)$$

This leads to the Fractional Steps Method which consists in solution of the fluid in separate steps. The first advantage is the fact that now a system of n equation should be solved each time. Another advantage is the fact that the nature of resulted system of equation in each step is uniform and can be preconditioned better. Although in our work the most important characteristic of splitting the conservation of momentum and conservation of mass equations is the fact that this allows us to calculate the advection component independently of the pressure.

Chapter 3

General aspects of our fluid solver

In this section we explain which is the specific problem we intend to solve and which are the algorithms used to do it. The first thing that needs to be clearly defined is the domain in which our solver is going to act. Simulations themselves can be separated into several parts or subdomains. As stated in the introduction we are interested in the subdomains of the solver that hold certain characteristics. In fact, we are interested only in one characteristic, whether our subdomain has or not contour conditions. We can see an example of subdomains with an without conditions in Fig 3-1

Our simulation can be reduced to the application of several basically operators over a matrix: The laplacian operator, the gradient, the divergence and the advection, which will be approximated using the BFECC algorithm. As it will be explained latter advection will also need of an interpolation operator. An explanation of the different operations is detailed below:

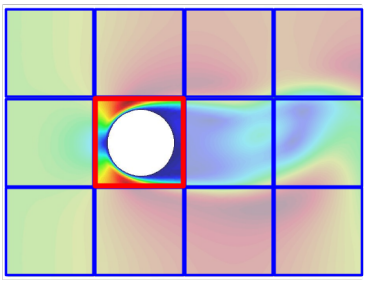

Figure 3-1: Decomposition of a problems with blocks. Red blocks have complex contour conditions, blue blocks have no complex contour conditions

3.1 Laplace Operator

Stencils are operations that update arrays elements according to some fixed patterns. They are widely used in image and signal processing where they are known as filters. Each point of this patterns can have different weight and depending in that weight, size and distribution is going to be used for different purposes (smoothing functions, laplacian operator, etc...).
Formally a given smooth stencil (fig. 3-2) in the plane could be defined for example as:

$$e_{i,j} = e_{i+1,j} + e_{i,j+1} + e_{i-1,j} + e_{i,j-1} - 4 * e_{i,j}$$

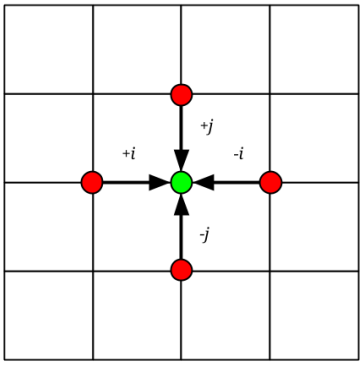

Figure 3-2: Example of a 5 points laplace operator

3.2 Interpolation

We call interpolation to the operation of constructing new data from the information that is currently available. In our specific problem this operation will consist on calculating the value of a given fictional point based on its coordinates. There are several algorithms to calculate point interpolations and our choice for this project has been the trilinear interpolation [6]. This algorithm is especially useful for our case as it allows us to calculate the value of a point inside a cube from the values of the vertices of such cube as is depicted in fig. 3-3.

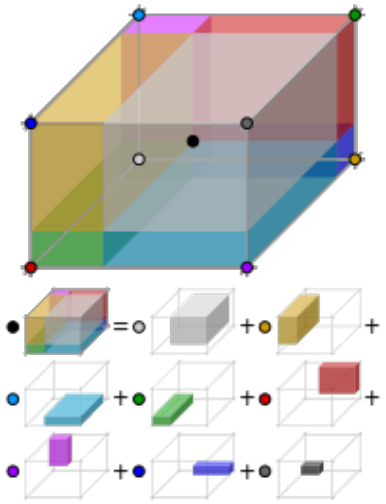

Figure 3-3: Trilinear interpolation [7]

3.3 Gradient

The gradient can be defined as the derivative of a function in one dimension to a function of several dimensions. Being P a scalar field in the plane, we can define ΔP as vector field with coordinates X and Y, where each component represent the partial derivative in that component as shown in the figure 3-4

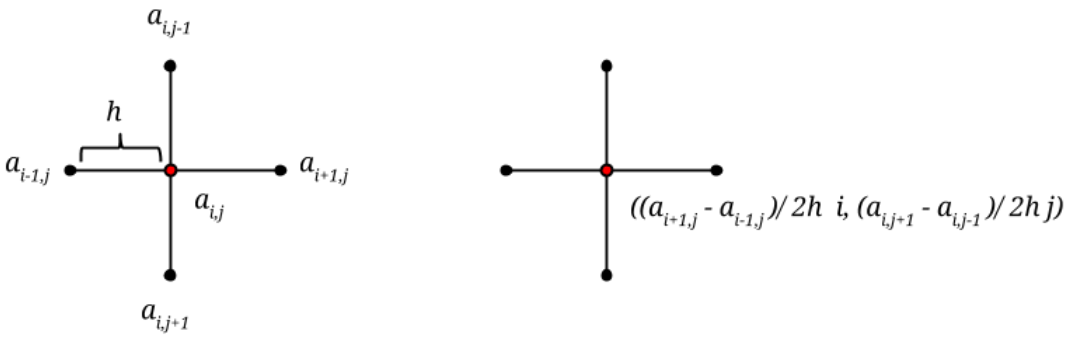

Figure 3-4: Gradient operator

3.4 Divergence

The divergence (∇) can be defined as a vector operator that measures the magnitude of a vector field's source or sink at a given point. As opposite as the gradient operator, the divergence convers a vector field in a scalar field (fig 3-5).

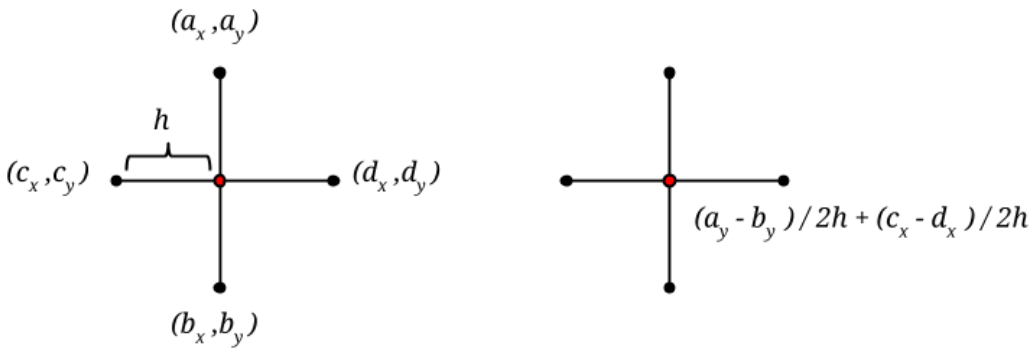

Figure 3-5: Divergence operator

3.5 The Back and Forth error compensation and correction algorithm

We have stated that in order to improve the precision of the code, a different approach has to be considered while solving our models. In this section we are going to introduce the back and forth error compensation and correction (BFECC) algorithm.

The BFECC algorithm [4][8] was originally developed as a method to reduce the error introduced while calculating different equations with level set methods. Level set methods in general are used to represent the evolution of a surface over the time being the last discretized. In fluid simulation this is also known as the Eulerian approach and inside this method the BFECC is used in the advection operation. Actually, the BFECC is not an algorithm but a strategy that can be

implemented in several manners which by utilizing different operations tries to identify and reduce the error caused by the spatial discretization of the model.

The BFECC approach used to implement the advection operation of the simulation will consist in three different steps: the Backward (Back), the Forward (Forth), and the Error Correction and Compensation (Ecc) that will minimize the error of the motion equation. For the clarity of the description, we are going to consider ϕ as a scalar independent variable, although it can be easily extended to N-dimension variables, and v as our velocity field.
The backward step is the first step of the process. Our objective will be to approximate the positon of every point of our model in the previous step based on the actual velocity. By calculating it this way, we are going to introduce an error represented by e which is characteristic of this method.

Formally, lets consider ϕ_{n+1} as our initial independent variable, L, as our level set operator and v_{n+1} as our velocity field for a given node n. We hence, would like to obtain:

$$\phi_{n+1} = L(-v_{n+1}, \phi_n)$$

In the forward step, we are going to calculate the value of the previously moved points once we move them forward. Ideally, this operation would return the particles back to its original position, but due to the presence of the systematic error, we are going to introduce an error e again. Formally:

$$\phi^{''} = L(v_{n+1}, \phi^{'})$$

In the final step we are going to proceed to remove the error. As we stated in the two previous steps, we have introduced e both in the forward movement and in the backward movement. Also, by moving our particle backward and forward, ideally would have ended in the same place as it was before. As a consequence, we can closely approximate this error e as the difference between or expected and real positions:

$$\phi_n = \phi^{''} - 2e$$

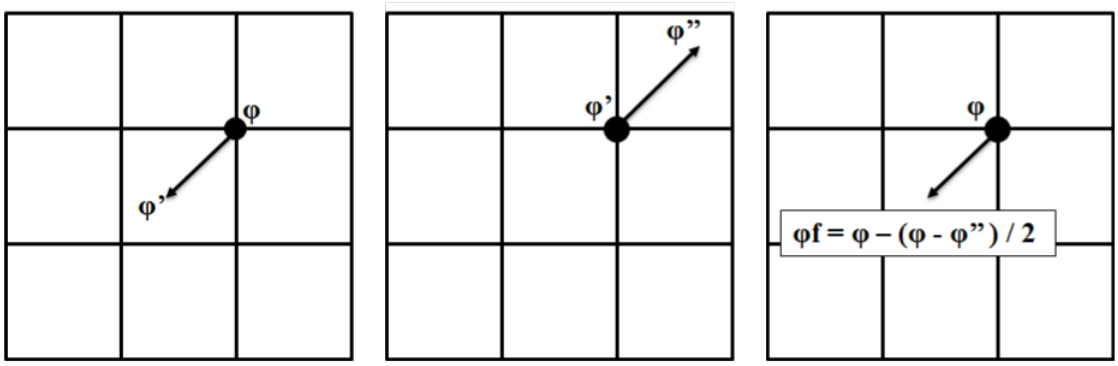

Figure 3-6: From left to right, (A) Back, (B) Forth, (C) Error correction and compensation

Hence, our error can be expressed as:

$$e = \tfrac{1}{2}(\phi'' - \phi_n)$$

Finally, we want to compensate such error and obtain the correct value of ϕ. To do that so, it will be enough to repeat the backward step, but this time we interpolate the new value over the field of corrected values provided by e

$$\phi_{n+1} = L(-v_{n+1}, \phi_n + e)$$

Replacing the error we obtain the expression:

$$\phi_{n+1} = L(v_{n+1}, \phi_n + \tfrac{1}{2}(\phi'' - \phi_n))$$

And finally:

$$\phi_{n+1} = L(v_{n+1}, \tfrac{3}{2}\phi_n - \tfrac{1}{2}\phi'')$$

Chapter 4

Design of the solver

In this section we discuss the main criteria that has been used in order to design our code. The problem presents some other characteristics that are relevant form the design point of view: First, the size of the problem is small (10K to 1M Elements) and secondly, we assume that in this part of the problem there will be no contour conditions. Additionally, as our code is part of a larger solution, some of the presented here does not find inside the objective of the dissertation, but is essential to understand why some decisions have been made.

A bottom-up methodology has been followed based on the different parallelism levels that we want to introduce in the solver: The main idea of the solver is divide our mesh in blocks. The way in which this decomposition is performed is only relevant in the terms of that it provides blocks with and without contour conditions. Each block hence, will represent a section of our model.
Every single one of these blocks then, is planned to be executed in one node of an HPC environment. This will represent the high level parallelism, which will be implemented using MPI in a future.

The blocks assigned to specific nodes then will be solved with the appropriate strategy: The first strategy, which is discussed in the document solves the blocks of the code with no contour conditions, the second is a classical FEM solver that will handle the rest of the model. This distinction over blocks with or without contour conditions will allow us to use structured meshes which would be im-

possible otherwise as the presence of contour would mean that we need to use unstructured mesh refinement in that block.

The necessity of communicate this two different solvers will result in the future addition of the coupler classes for BFECC-BFECC, BFECC-FEM. and FEM-FEM blocks. The code hence, is designed to support multiple levels of parallelism in a hierarchical form. In 4-1 the parts implemented in this step of the development appear shadowed.

Hence, our solver is composed by the following classes:

BlockSolver: Is the class that holds the implementation of the solver. At the same time, there are different types of solvers for the different operations of the process and the different implementation methods, typically advection (Advection-BFECC Solver) and diffusion (Diffusion Solver). The design presented in 4-1 allow a convenient way to enable or disable specific implementations or operations of the solver without any further problem.

Block: Is the basic computational element. The block class in our code represents the portion of the problem that we are solving. It also holds information about the boundary conditions, if any, and the width of the overlapping section, a set of extra elements needed for communication, with the neighbor blocks along with a list of them. Blocks hence are intended to be stand-alone elements in which we can apply the desired operations.

BlockCoupler is used to manage the communication between the different blocks of the problem if there is more than one, as in the current step of the development we are interested in the performance of a single block this class has not been implemented. In a future this class will also control the coupling between blocks in different nodes through MPI.

Solver: Stores the physical characteristics of the problems and is in charge of control the workflow of the application by applying the required BlockSolvers to the blocks composing the problem.

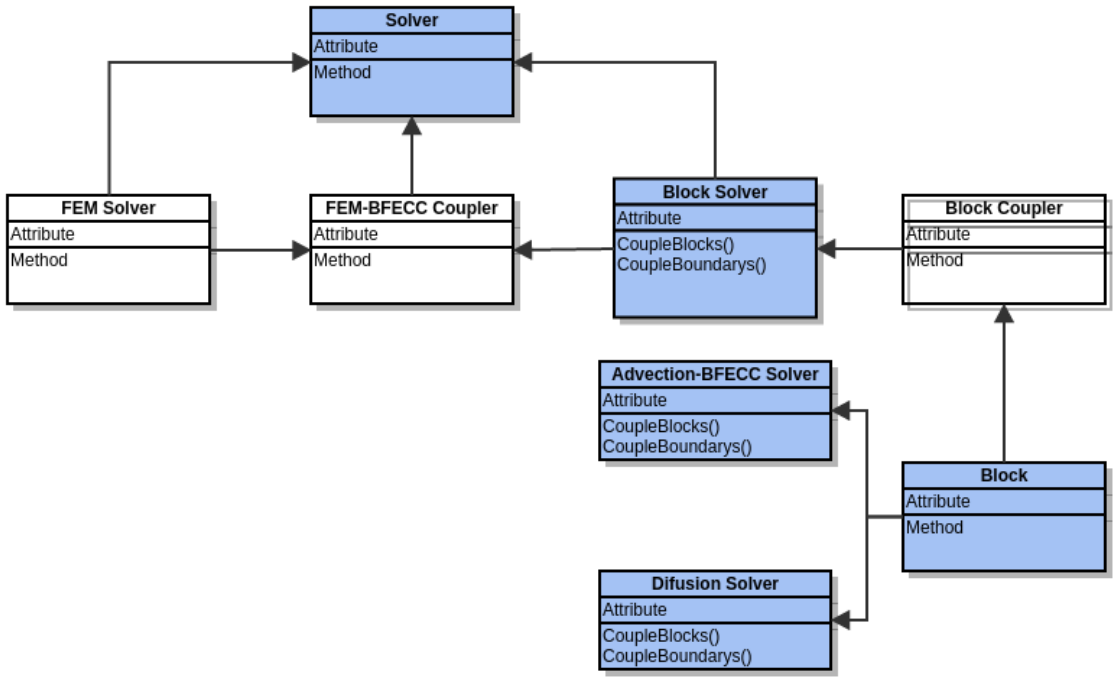

Figure 4-1: Uml design of the solver with implemented parts in blue

Chapter 5

Implementation

In this section a first implementation of the full solver is presented. No major optimizations have been implemented at this stage.

5.1 Data structures

Once the design of our solver is defined we need to decide how to implement the data structures of the problem. As explained in the design the main element of the solver will be the block. For each of the blocks used in the solver, which in this step of the implementation will be one, we will need to represent de different numerical values for the pressure, velocity, and our ϕ variable.

Since the coupling code that is being developed only accept cubic domains at the moment only cubes are allowed, so every problem will be of size N^3. In addition to the size of the problem we will need an overlapping region. This region will add approximately one or two extra cells of the matrix per side. Under this restrictions we will have to define the following data structures:
Being:

$$ProblemSize = (N + Overlap * 2)^3$$

1. **The velocity field**: A 3D matrix of $ProblemSize$ which will store the X, Y and Z components of the velocity
2. **Pressure field**: A 3D matrix of $ProblemSize$ which will store the scalar value of the pressure
3. **Variable fields**: A set of 3D matrices of $ProblemSize$ which will store the intermediate values of our variable
4. **Flag matrix**: A 3D matrix of $ProblemSize$ which will store information about fixed parameters, boundary conditions and other restrictions for each particle of the model.

5.2 Description of the solver workflow

The implementation of the code seeks to solve the Navier-Stokes equation. As described in section 2 we can split this equation in two separated parts, the conservation of mass, and the conservation of momentum. The conservation of momentum is the part we are interested on in this dissertation and represents the movements of the particles that configure the fluid. The conservation of mass is used to calculate the pressure and compressibility of the fluid but it will be subject of major changes in the future, so although it is implemented so the solver is functional no special attention is going to be put there at this stage.
Once the equation is converted to numerical operations, we end with a list of tasks than need to be executed for each iteration of the solver. For the conservation of momentum, these steps are:

1. Advection calculation
2. Acceleration component calculation
3. Pressure gradient component calculation
4. Diffusion component calculation
5. Integration of all components

And for the conservation of mass, the steps are:

1. Calculate the pressure difference, by applying a divergence operator over the new velocity field
2. Apply a smoothing operator over the pressure difference
3. Update the pressure

5.2.1 Advection calculation

In this step we take the original variable and we apply the advection algorithm. In our case we take the velocity itself as we are interested in the movement of the fluid. We implement this operation using the BFECC algorithm in order to achieve the desired increment in precision and hence the operation consists on executing the Back, Forth and Error correction operations as shown in figure 5-1 and 5-2. This will be the main subject of our optimizations as it represents more than 50% of all execution time. Notice also that in order to avoid race conditions in a future optimization, different buffers (pPhiB, pPhiC) are used to store the results of every step.

```
1  for(k = 0; k < sizeZ; k++)
2    for(j = 0; j < sizeY; j++)
3      for(i = 0; i < sizeX; i++)
4        Apply(pPhiB,pPhiA,pPhiA,-1.0f,0.0f,1.0f,i,j,k) // Back
5
6  for(k = 0; k < sizeZ; k++)
7    for(j = 0; j < sizeY; j++)
8      for(i = 0; i < sizeX; i++)
9        Apply(pPhiC,pPhiA,pPhiB,1.0f,1.5f,-0.5f,i,j,k) // Forth
10
11 for(k = 0; k < sizeZ; k++)
12   for(j = 0; j < sizeY; j++)
13     for(i = 0; i < sizeX; i++)
14       Apply(pPhiA,pPhiA,pPhiC,-1.0f,0.0f,1.0f,i,j,k) // Error Correction
```

Figure 5-1: Different steps of the advection operation

```
void Apply(
  PrecisionType * Phi,
  PrecisionType * PhiAuxA,
  PrecisionType * PhiAuxB,
  const PrecisionType &Sign,
  const PrecisionType &WeightA,
  const PrecisionType &WeightB,
  const size_t &i,
  const size_t &j,
  const size_t &k) {

size_t cell = IndexType::GetIndex(i,j,k,pBlock->mPaddY,pBlock->mPaddZ);

PrecisionType iPhi[MAX_DIM];
PrecisionType origin[MAX_DIM];
PrecisionType displacement[MAX_DIM];

origin[0] = (PrecisionType)i * rDx;
origin[1] = (PrecisionType)j * rDx;
origin[2] = (PrecisionType)k * rDx;

for(size_t d = 0; d < 3; d++) {
  displacement[d] = origin[d] + Sign * pVelocity[cell*rDim+d] * rDt;
}

InterpolateType::Interpolate(
  pBlock,
  PhiAuxB,
  (PrecisionType*)iPhi,
  displacement,
  rDim
);

if(!(pFlags[cell] & FIXED_VELOCITY_X))
  Phi[cell*rDim+0] = WeightA * PhiAuxA[cell*rDim+0] + WeightB * iPhi[0];
if(!(pFlags[cell] & FIXED_VELOCITY_Y))
  Phi[cell*rDim+1] = WeightA * PhiAuxA[cell*rDim+1] + WeightB * iPhi[1];
if(!(pFlags[cell] & FIXED_VELOCITY_Z))
  Phi[cell*rDim+2] = WeightA * PhiAuxA[cell*rDim+2] + WeightB * iPhi[2];
}
```

Figure 5-2: Apply function for every of the BFECC steps

5.2.2 Acceleration component calculation

With the advection calculated it is possible to calculate the acceleration of our fluid by just subtracting the initial velocity to the one calculated by the BFECC algorithm, which will be used to approximate the final position of the fluid.

```
inline void calculateAcceleration(
  PrecisionType * gridA,
  PrecisionType * gridB,
  PrecisionType * gridC,
  const size_t &cell,
  const size_t &X,
  const size_t &Y,
  const size_t &Z,
  const size_t &Dim) {

for (size_t d = 0; d < Dim; d++) {
  gridC[cell*Dim+d] = (gridB[cell*Dim+d] - gridA[cell*Dim+d])/rDt;
}
}
```

Figure 5-3: Acceleration calculation

5.2.3 Pressure gradient component calculation

Here we calculate the component of the velocity that is introduced by the pressure difference in our fluid, which will move from high pressure areas to low pressure ones.

```
inline void gradientPressure(
  PrecisionType * press,
  PrecisionType * gridB,
  const size_t &cell,
  const size_t &X,
  const size_t &Y,
  const size_t &Z) {

PrecisionType pressGrad[3];

pressGrad[0] = (
  press[(cell + 1)] -
  press[(cell - 1)]);

pressGrad[1] = (
  press[(cell + (X+BW))] -
  press[(cell - (X+BW))]);

pressGrad[2] = (
  press[(cell + (Y+BW)*(X+BW))] -
  press[(cell - (Y+BW)*(X+BW))]);

for (size_t d = 0; d < rDim; d++) {
  gridB[cell*rDim+d] = pressGrad[d] * 0.5f * rIdx;
}
}
```

Figure 5-4: Pressure gradient calculation

5.2.4 Diffusion component calculation

In this step we approximate the laplacian velocity divergence by calculation the diffusion in the velocity field. This will appear as a velocity in an opposite direction to the fluid. Notice also that that this is the operator that we are going to use in order to smooth the pressure difference while updating the pressure in the conservation of mass.

```
1  inline void lapplacian(
2    PrecisionType * gridA,
3    PrecisionType * gridB,
4    const size_t &cell,
5    const size_t &X,
6    const size_t &Y,
7    const size_t &Z,
8    const size_t &Dim) {
9
10 for (size_t d = 0; d < Dim; d++) {
11   gridB[cell*Dim+d] = (
12     gridA[(cell - 1)*Dim+d]    +                  // Left
13     gridA[(cell + 1)*Dim+d]    +                  // Right
14     gridA[(cell - (X+BW))*Dim+d]    +             // Up
15     gridA[(cell + (X+BW))*Dim+d]    +             // Down
16     gridA[(cell - (Y+BW)*(X+BW))*Dim+d] +         // Front
17     gridA[(cell + (Y+BW)*(X+BW))*Dim+d]) /
18     1.0f/6.0f;
19 }
20 }
```

Figure 5-5: Laplacian Calculation

5.2.5 Integrate all the components and calculation the new velocity

With all the elements of the equation calculated we can finally define the velocity of the fluid in the new time step.

```
for(size_t k = rBWP; k < rZ + rBWP; k++) {
  for(size_t j = rBWP; j < rY + rBWP; j++) {
    size_t cell = k*(rZ+rBW)*(rY+rBW)+j*(rY+BW)+rBWP;
    for(size_t i = rBWP; i < rX + rBWP; i++) {

      if(!(pFlags[cell] & FIXED_VELOCITY_X))
        initVel[cell*rDim+0] += (rMu * velLapp[cell*rDim+0] -
        pressGrad[cell*rDim+0] + force[0] / rRo + acc[cell*rDim+0]) * rDt;

      if(!(pFlags[cell] & FIXED_VELOCITY_Y))
        initVel[cell*rDim+1] += (rMu * velLapp[cell*rDim+1] -
        pressGrad[cell*rDim+1] + force[1] / rRo + acc[cell*rDim+1]) * rDt;

      if(!(pFlags[cell] & FIXED_VELOCITY_Z))
        initVel[cell*rDim+2] += (rMu * velLapp[cell*rDim+2] -
        pressGrad[cell*rDim+2] + force[2] / rRo + acc[cell*rDim+2]) * rDt;

      cell++;
    }
  }
}
```

Figure 5-6: Integration of the motion equation components

5.2.6 Calculate the new pressure

Finally we calculate the new pressure by calculation the divergence of the velocity and applying a diffusion over the difference of pressures to smooth the results. This part of the code will not be subject of optimization as is only a placeholder until the implicit formulation for the pressure is implemented.

```
inline void calculateAcceleration(
      PrecisionType * gridA,
      PrecisionType * gridB,
      PrecisionType * gridC,
      const size_t &cell,
      const size_t &X,
      const size_t &Y,
      const size_t &Z,
      const size_t &Dim) {

    for (size_t d = 0; d < Dim; d++) {
      gridC[cell*Dim+d] = (gridB[cell*Dim+d] - gridA[cell*Dim+d])/rDt;
    }
  }
```

Figure 5-7: Apply function for every of the BFECC steps

Chapter 6

Solver results and validation

Before starting to optimize the code so it fulfils the objectives for which it was developed. A set of control cases need to be defined to validate the correctness of the solution. The purpose of this cases will be to observe the expected behavior of the code with the desired precision and the stability of the solution. A couple of tests are going to be performed, the still water tank and the Cavity Test.
We are mainly interested in the effects that BFECC algorithm could have had and hence the cases that are present seek to check the correctness of the algorithm. We also going to show the stand alone results of the BFECC algorithm versus its implementation using FDM and versus its current implementation in KratosMultiphisics software using tetrahedron elements.
At this stage we are interested in check that the physical result of the simulation is correct. A comparison of the precision and the execution speed will be driven later.

6.1 BFECC results

We have conducted a couple of simple experiments in order to measure the precision increment introduced by the BFECC algorithm. This experiments consist on a cubic mesh of arbitrary size in which we have added a heat focus. We apply a rotational velocity field and we convect the heat focus over the mesh using that velocity.

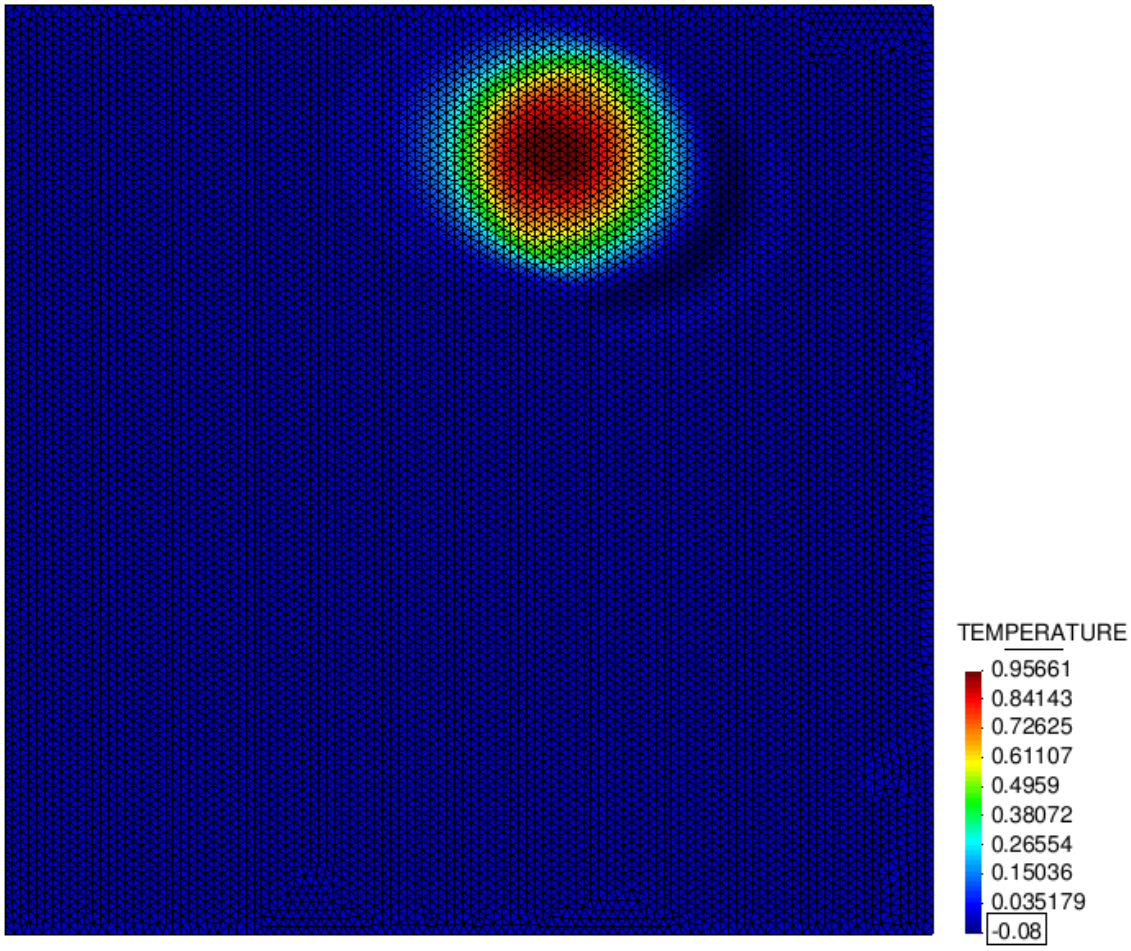

Figure 6-1: Convection of a heat focus using a pure convection algorithm (FEM). Approx. 15.000 elements

In a perfect situation, the heat is conserved over all the simulation. In the practice, the error introduced while convecting the temperature will cause a small diffusion in the process which will reduce the peak of heat focus.

In figure 6-1 we appreciate some overheads and a maximum peak of temperature of 0.9566. We can also observe that the shape of the heat focus is starting to deform (blue blob in the left) and a trailing incorrect value also appears in the opposite direction of the velocity (blue blob in the right). In the BFECC implementation, figure 6-1, it can be seen that the overshoots and shape deformities are gone, except for a small deformation at the left bottom (in dark blue) and

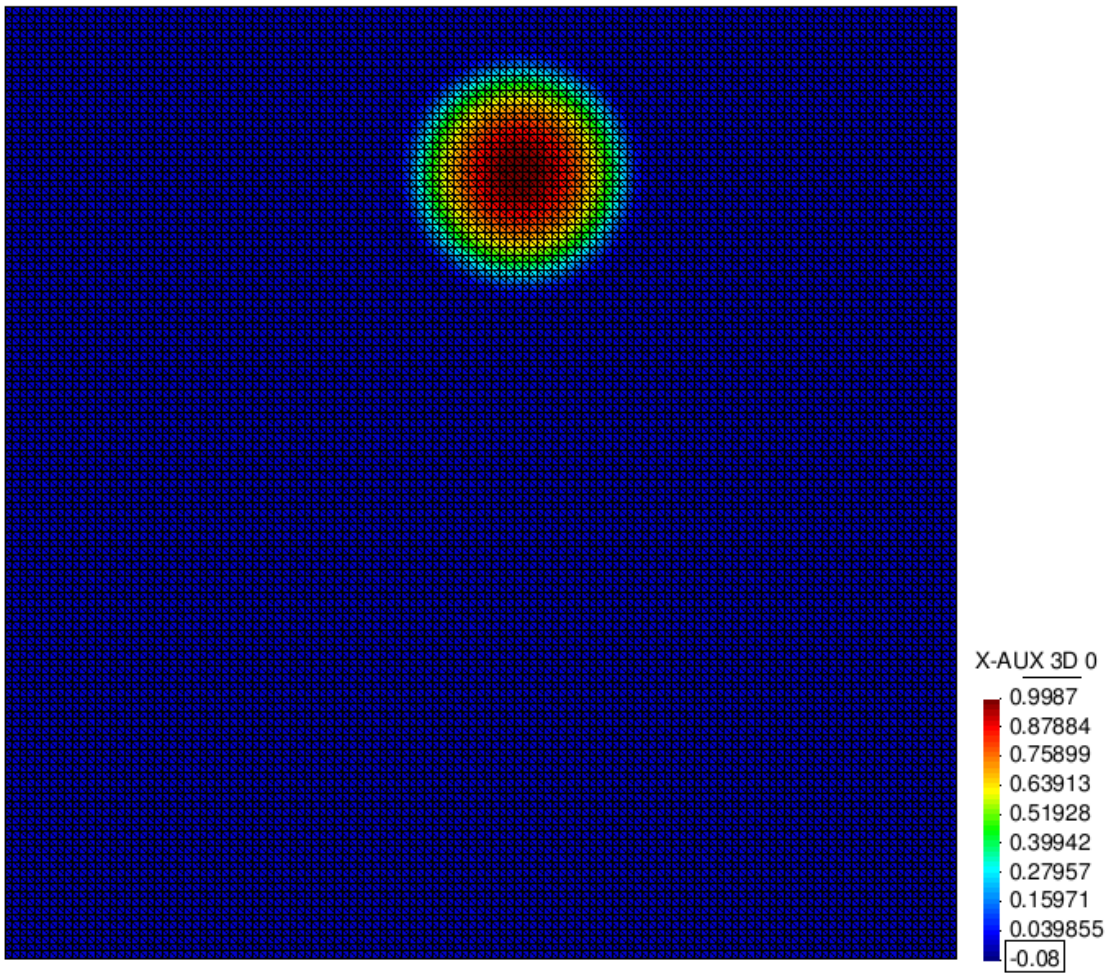

Figure 6-2: Convection of a heat focus using BFECC. Approx. 15.000 elements

the peak of the heat holds a value of 0.9987, which is one order of magnitude better.

At the sight of the results we are confident that the precision obtained by the BFECC method is better than the achieved by its predecessors. Both in shape and quantitative value.

6.2 Full solver results

In this section we show the results for the full solver. We present three different control cases: A sill water tank and a Cavity test case.

6.2.1 Still water tank example

In the still water tank we fix the velocity of all our walls and expect that all the forces of the problem enter in a state of equilibrium. As our problem takes into account compressibility it may happen that this equilibrium state is in fact an oscillation of the forces over time.

We start from to initial conditions. First a soft initial condition in which a pressure gradient already exist. In this example, no force would have to appear in the model, as the initial condition itself is already an equilibrium state, which is precisely the results that we obtain, as we can see in the figure 6-3.
In the second initial condition no pressure field is imposed but is then generated after some oscillations of the fluid inside the tank, achieving again an equilibrium state. We can see the evolution of the velocity of the model in the figure 6-4.

6.2.2 Cavity 3D example

The cavity 3d example consists on imposing a fixed velocity in the top of a cavity, usually a cube, and fixing the velocity of all the other sides to 0. Depending on the Reynold number, the relation between the viscosity and the fluid velocity, different vorticities are generated in the fluid inside the cube. The cavity example

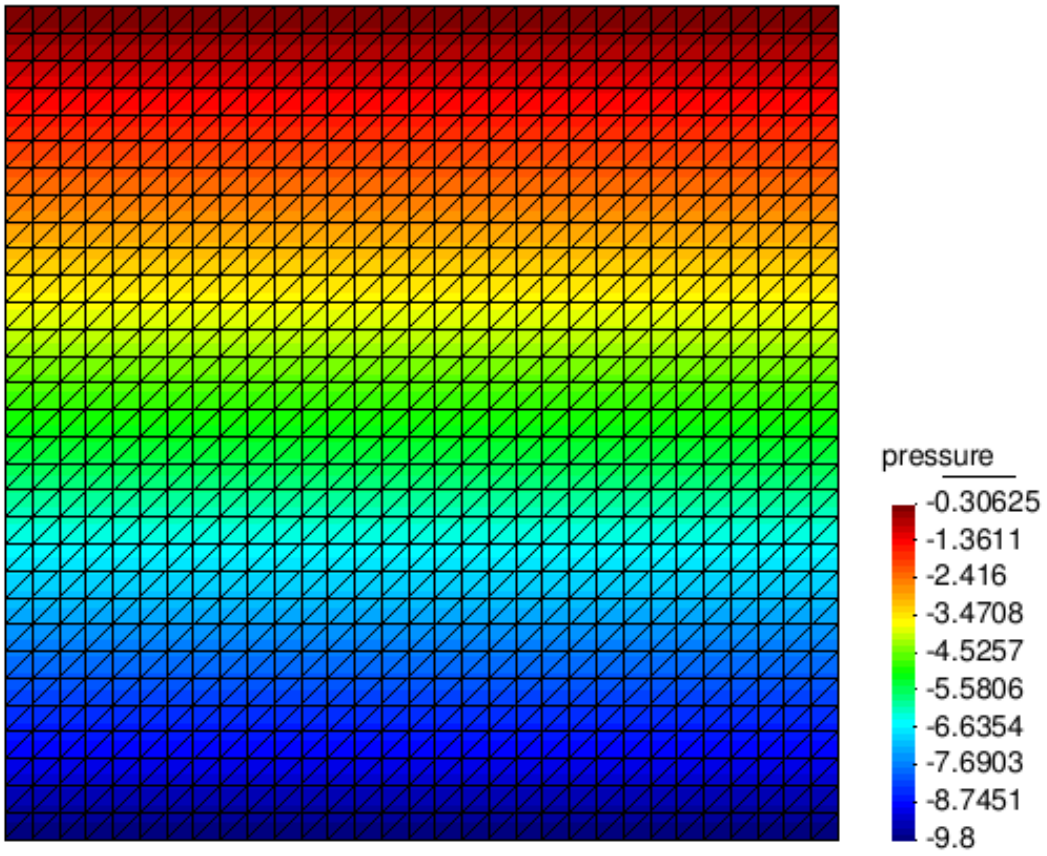

Figure 6-3: Still water tank with initial pressure

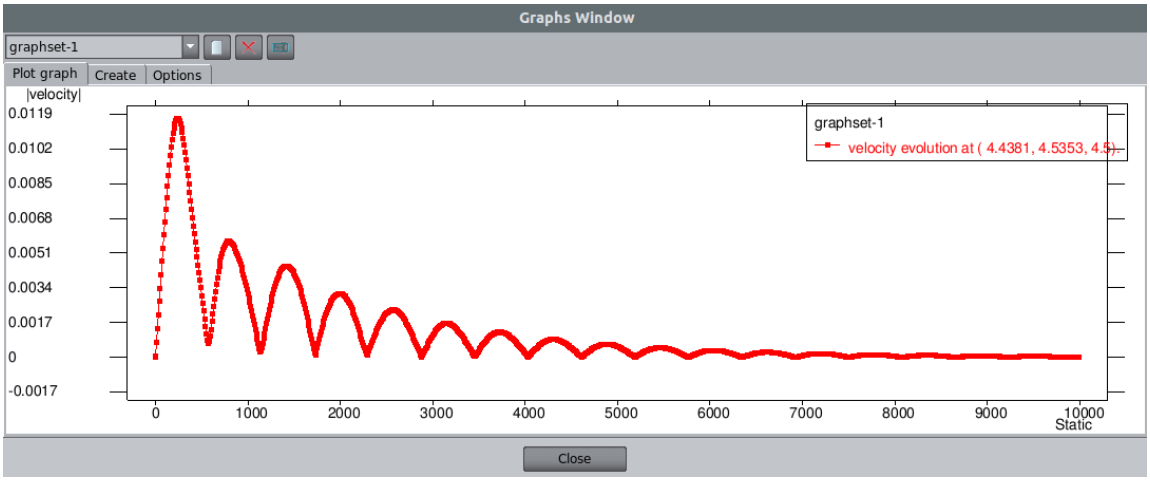

Figure 6-4: Evolution of the velocity without an initial pressure gradient

is a very well-known example and its results have been widely analyzed both numerically and analytically.

In our example we select a Reynold number of 1000 (values of viscosity and velocity may change between executions). The resolution of the grid is 32x32 so a total of 1024 elements have been used in the simulation of FEM and 32768 in the BFECC method, as it cannot run for 2D models.

Comparing the results of the FEM-Based solution in figure 6-5 with the ones obtained in our simulation in 6-6 we can check that, even with a very small resolution grid, the results are close to the ones expected analytically. The use of the low resolution grid although makes the two vortices at the bottom of the cavity almost inappreciable.

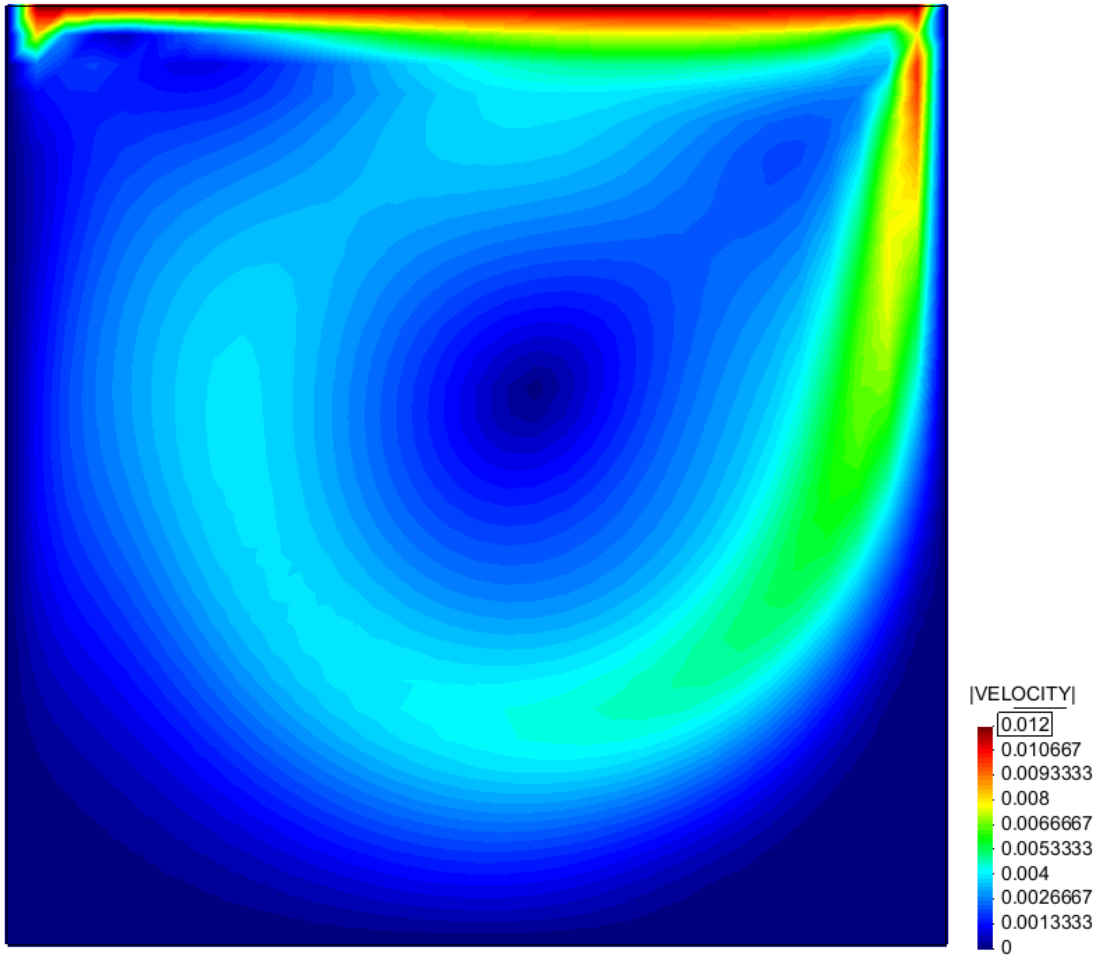

Figure 6-5: Experimental results of Cavity example for Rn=1000 using the FEM-Based implementation

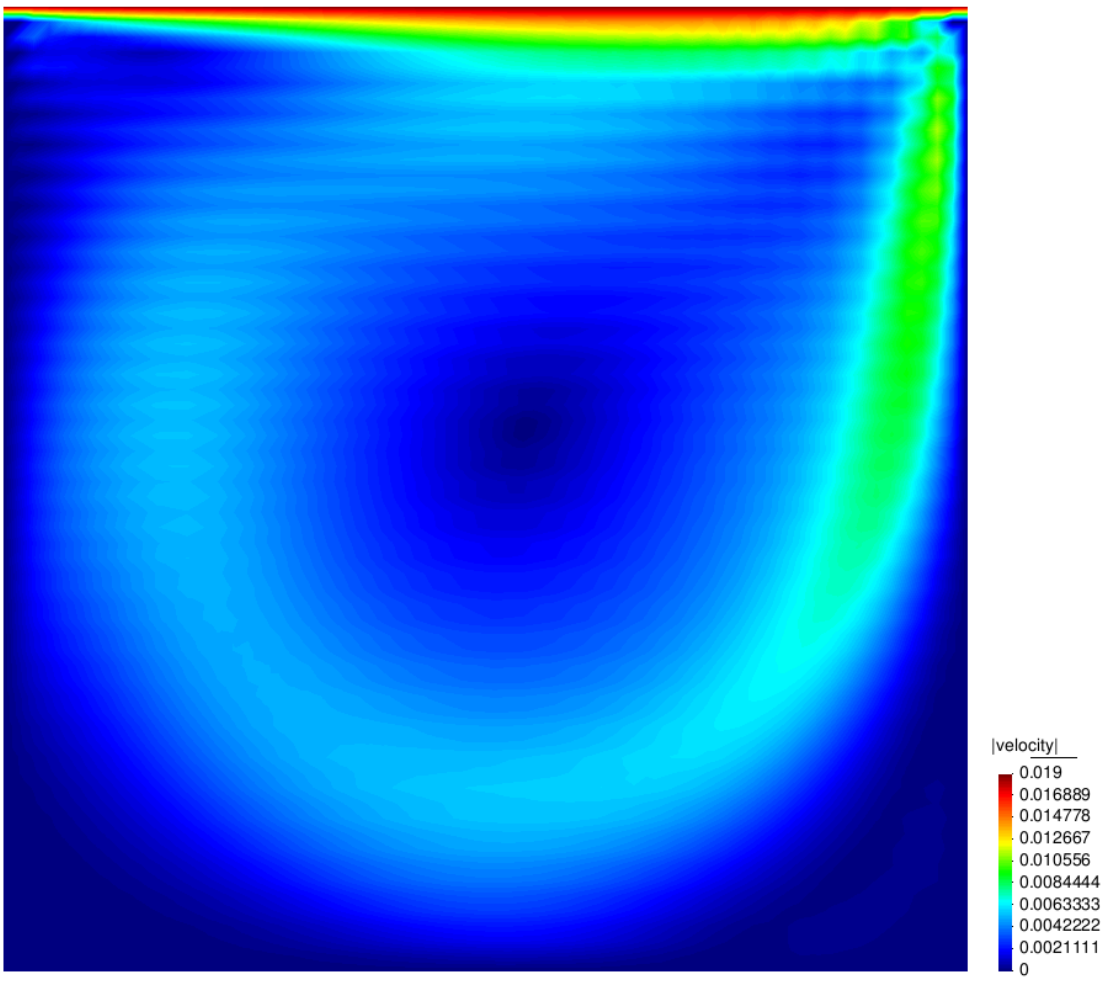

Figure 6-6: Experimental results of Cavity example for Rn=1000 using the BFECC implementation

Chapter 7

Optimizations of the BFECC algorithm

Once the requirements of precession have been achieved by applying the BFECC algorithm, the structure of data obtained allows us to incorporate further technics to improve the speed of the code as well. We are going to focus our optimizations in two separate blocks, optimizations related the advection and optimizations relating the diffusion

7.1 Neighbor search replacement

The first and probably the most important of these optimizations is the opportunity to completely remove the search of neighbors from the advection step. Typically a neighbor search has an optimal cost of $O(NLogN)$. For our problem this operation need to be executed for every single particle in the problem.

There are several methods in the literature that are used to perform this operation such as the Octree, R-Trees, or even brute force if the number of elements is relatively small. Typically our interests would be in hash based method that groups the particles in small containers which would cause our search space to be smaller. The calculation of that container would be the so called hash function h.

The optimization that we present is basically an extreme case of a geometrical hash reducing the search space to 1 element due to the fact that the code is already organized as a matrix. Basically we have seen that our nodes are going to be the vertices of the cells that compose our matrix. Let's consider n_i as the right node of a bar element e_{i-1} in the position p_i, and lets consider $v_i = (-0.5)$ the velocity of the field in the position of n_i. Since our mesh is regular we can also consider Δx as the distance between n_i and n_{i+1}. Finally we are going to consider our step is given by Δt.

The process of moving a node, hence, is going to provide us with $p_{i+1} = p_i + v_i \Delta t$. Since the dimension of the elements is know we can use this very same operation as our hash function an additionally we also know that $p_{i+1} = p_i + k\Delta x + r$, being r a residue.

Hence, is quite obvious to see that the landing element e_j of a moved node n_i with a speed of v_i is going to be $j = i - \Delta x * k$. This landing element is going to be unique provided that no overlaps occur in our mesh which means that our node is going to have the elements e_j and e_{j+1} as neighbors. Moreover, r is the weight that we are going to need latter in the interpolation of the neighbor values.

In conclusion, we do not need to perform a search for a given element neighbor, as it can be easily calculated as an offset. The main drawback of this optimization is the fact that it requires for a float modular operation in order to calculate the correct cell. This however could avoided by using a Δx of 1, which removes the necessity of a modulus operator. This dx can be trivially obtained by rescaling the mesh at the beginning and the end of the calculous without having to pay any performance penalty.

7.2 Loop reordering

The main goal with this optimization is to store our matrix in a way that minimizes the cache misses. This is a very well-known technic that basically consist

on changing the order of the iterations in order to exploit the locality of the cache. The technic is language specific, for example C++ matrices are sorted by columns internally while in FORTRAN they are iterated by rows. As our language is C++ the matrix will be stored by planes, then columns and finally rows, hence the most beneficial way to traverse our 3D matrix would be from Z to X, or typically k,j,i.

This traverse order causes that for any given point in our mesh; at most 9 rows are loaded simultaneously. The CFL condition also has impact in these tests as it determines how distant are the rows that we need to check from the row of the original point. Typically smaller CFL values would result in a better cache locality, as closer lines would need to be retrieved in order to calculate the new positions. In the practice the explicit implementation of the pressure limits the CFL to values lower than 1, so at the end it does not interfere with the results.

The different times obtained for the permutations in the loop order with $CFL = 1$ are presented in the figure 7-1.

Optimization	Time
k-major	12.8771
j-major	13.1920
i-major	19.18556

Table 7.1: Impact of different traverse order

7.3 Manual Vectorization

Compilers are capable of automatically vectorize codes so they can make use of SIMD instructions like SSE, MME and AVX among others, but there is still work to do to fully automate the vectorization of complex code [9], we believe that some parts of our code can be vectorized manually without affection the performance of the parallelism.

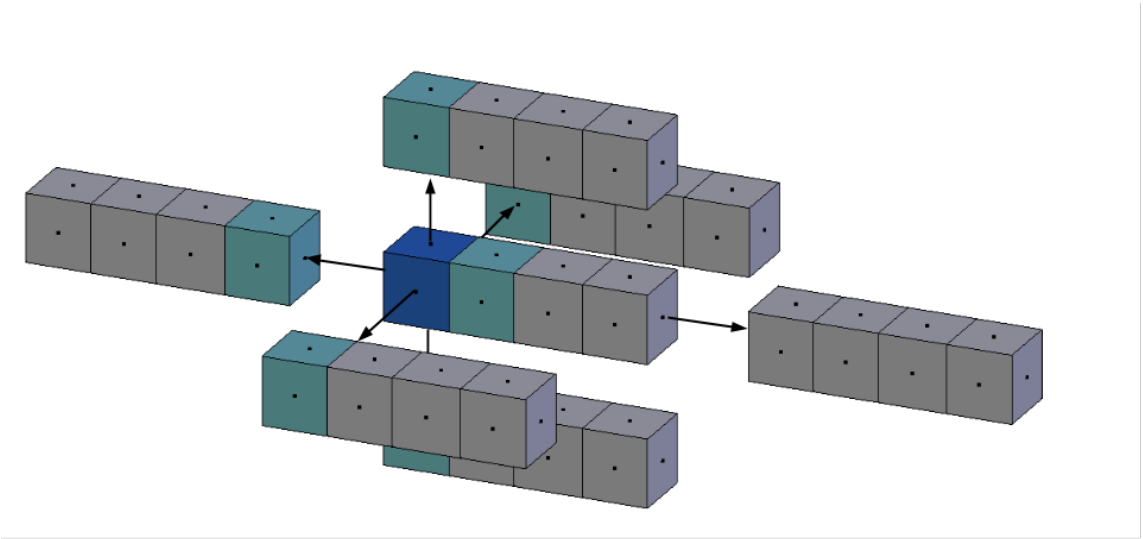

Figure 7-1: Naive manual vectorization of the diffusion stencil. Each block represents a vectorial register of 4 elements

We identify mainly one interesting manual vectorization opportunity in the diffusion step. As a reminder the operation that spends most of the time in this step is the stencil. In our concrete case we use a cross stencil. A naive transformation of this code in order to be able to run with vector instruction sets as shown in 7-1 implies to pack several consecutive elements of one of the matrix rows and access the upper bottom front and back elements as well in blocks. This approach although, has a major problem, and it is the fact that the left and right data, as contained in the same row, cannot be vectorized, and it implies doing a normal loop over the X axis in order to compute the left and right values.

This symmetry problem can be avoided if we apply two modifications to our algorithm. First, we will need to extend the width of the boundary of our matrix from one, to the size of our vector register, typically 2, 4 or 8. Second we are going to change the way the data is stored in the matrix by interleaving the X row using a stride of length of the vectorial register, again 2 4 or 8.

By doing this two changes we can iterate over all the elements of the row by vectors as shown in 7-2. The only exceptions to this would be when we have to apply the stencil to the elements in extremes of our matrix. For the vector

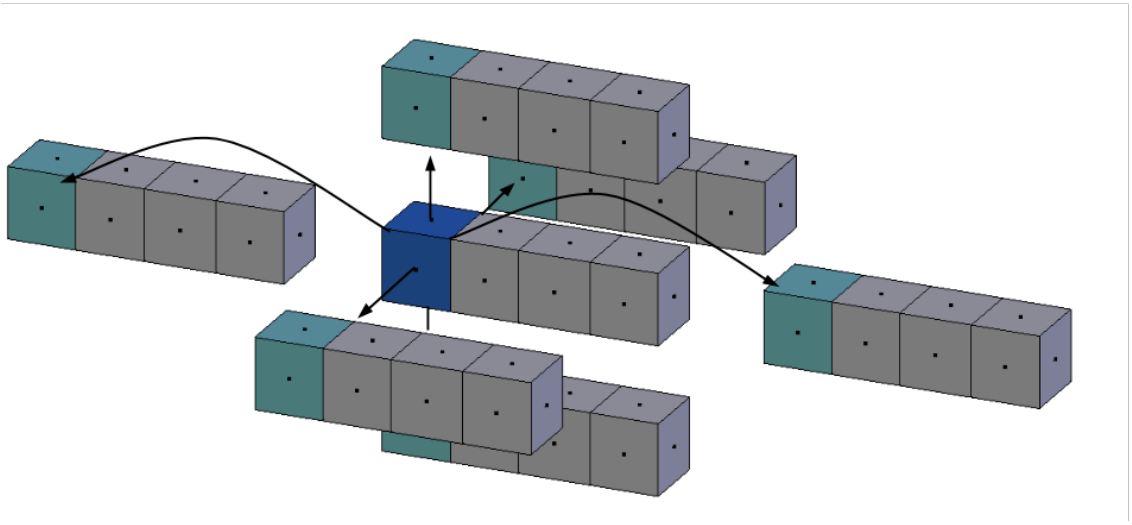

Figure 7-2: Correct manual vectorization of the diffusion stencil

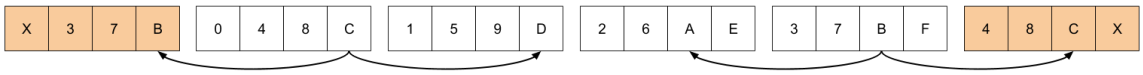

Figure 7-3: Stencil in the sides of the matrix (Extra blocks shadowed)

register at the beginning of one matrix of size N this can be solve by replacing the $i-1$ register vector with last vector register of the row with its values shifted one position to the right. Analogously the same happens with the last vector register. In this case we need to replace the $i+1$ register with the first register of the row shifted one position to the left as is depicted in 7-3.

7.4 Parallelization of the BFECC

In general the optimizations seen until this point focus on accelerate specific parts of the code exploiting concepts like cache locality, reducing operations, or branch suppression. This improvements are important as they reduce the total computational time but in general are not scalable. Nowadays the most challenging part of a code optimization is the introduction of parallelism.

The first choice here is to decide which technology is going to be used to parallelize that code. We can distinguish several interesting options here such as OpenMP, MPI or even CUDA or OpenCL.

It is important to state here again that the focus of the project is to develop a code that performs optimally in a single compute node. This restriction make MPI would not be the best choice as is more indicated for inter node parallelism rather than intra node parallelism, moreover, we want to use MPI to implement the communication of several of this blocks in a near future so its use is discarded.

CUDA and OpenCL would be a very promising choice here as at a first sight a many core architecture would heroically be able to process all elements of our matrices individually. The problem arise as the data needed to execute the interpolate operations is not always in the same locations, moreover, two threads could need to access the same data, and this breaks the coalescence in memory. Additionally we would need to develop an extra coupling module for GPU based blocks and a balancing module to deal with the time difference between CPU and GPU executed blocks. So although it is a very promising choice we rather prefer to explore this option in a future.

This gives as only one choice which is OpenMP. OpenMp allow us to efficient and easily parallelize codes in shared memory context, which is in fact the target of the algorithm. Even with the technology decided there several implementation of OpenMP that can be explored in order to parallelize our code that would be loop-based parallelism and task-based parallelism.

Historically the official OpenMP begun as a loop-based parallelism technology but it added the task support in the version 3.0. Although there are other OpenMP implementations that support task-based parallelism for our work we have selected the library called OmpSs by the BSC, which in addition to task-based parallelism offers extensions of the language that makes the process of taskification much more easy and intuitive.

7.4.1 Lope based parallelization

In this section we cover the loop based parallelization of both the diffusion and the advection operations.

Parallelization of the diffusion

The diffusion operator is responsible of simulate the diffusive effect of the fluid while. This operator has multiple possible implementations and among these the code uses a standard seven-point stencil. This stencil is applied over a three dimensional input matrix and the results are stored in an output matrix, being these matrices allocated in different memory spaces.

As is not the primary focus of this document to explain the parallelization of this problem, which has already been studied in detail, only a few guidelines to justify the method used to implement the parallelization are given. The seven-point stencil code itself is pretty straightforward but can introduce decency problems if the input and output matrices used in the operator are the same. In this situation the order in which the elements are updated affects the final results. Some methods in the literature such as the Red-Black or Gauss-Seidel have already been developed in order to address that problem proposing execution orders that are both parallelizable and do not affect the final result of the computation. As we already have two matrices due to other requirements of the code and since the amount of data that is intend to process is relatively small (order of millions of elements per matrix) a simple Jacobi method, which requires to store the results in a separate memory space but is fully parallelizable, has been implemented for the parallelism of this step.

Parallelization of the advection

The advection operation is used to move our fluid and it is executed once each iteration of the simulation. This operation is implemented using the BFECC algorithm as it has been shown in previous sections. Reduced to its basic elements this algorithm consists on executing three different steps over a series of matrices. These steps are the Back, Forth and Error (Fig 7-4). The instructions inside each one of these steps are essentially the same and consist on calculating, for each element of the matrix, a displacement followed by an interpolation of the values of a neighbor cell. Both the displacement and the interpolation are inherently serial operations, and thus are not going to be considered to be parallelized in the first place.

```
1  for(k = 0; k < sizeZ; k++)
2      for(j = 0; j < sizeY; j++)
3          for(i = 0; i < sizeX; i++)
4              ApplyBack(pPhiB,pPhiA,pPhiA,i,j,k)
5
6  for(k = 0; k < sizeZ; k++)
7      for(j = 0; j < sizeY; j++)
8          for(i = 0; i < sizeX; i++)
9              ApplyForth(pPhiC,pPhiA,pPhiB,i,j,k)
10
11 for(k = 0; k < sizeZ; k++)
12     for(j = 0; j < sizeY; j++)
13         for(i = 0; i < sizeX; i++)
14             ApplyError(pPhiA,pPhiA,pPhiC,i,j,k)
```

Figure 7-4: Advection step

Each one of these steps uses one input matrix (two in the Forth step) and its own output matrix to store the results. Having different memory spaces for the inputs and outputs means that there are no restriction in the calculation order of each element inside each step, but there is a dependency between the output matrix of one step and the input of the one that follows (Fig 7-5). This dependences yield to a natural parallelization strategy where each one of these steps is going to be parallelized independently without any restriction but where the order of the steps has to be maintained.

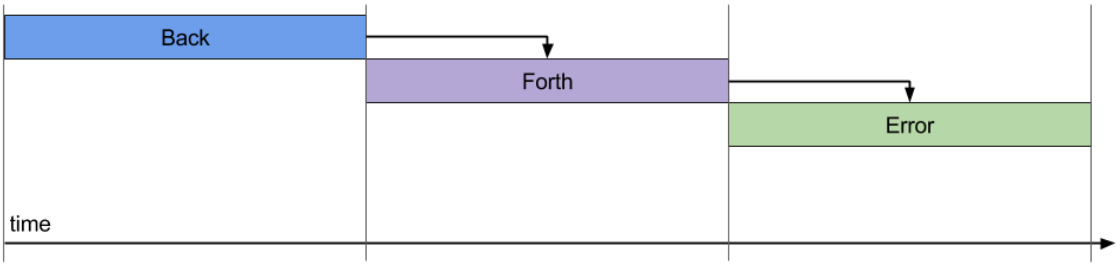

Figure 7-5: Temporal dependencies between the advection phases

Since the code itself is, due to its implementation, already divided into three sections each of them corresponding to one of the steps, the required dependencies that need to be represented can be expressed either executing all three steps in a single parallel region with barriers in between(Fig 7-6), or by executing the loops corresponding to each one of the steps in its own parallel region (Fig 7-7) where the implicit OpenMP barrier mechanism will guarantee that at the end of each loop all the threads are synchronized. Either the former or the later represent a good solution but as the implicit barrier mechanism is slightly faster than using explicit barriers the second option has been chosen to implement the first approach of the parallel code.

```
1  #pragma omp parallel
2  {
3      for(k = 0; k < sizeZ; k++)
4          for(j = 0; j < sizeY; j++)
5              for(i = 0; i < sizeX; i++)
6                  ApplyBack(pPhiB,pPhiA,pPhiA,i,j,k);
7
8      #pragma omp barrier
9
10     for(k = 0; k < sizeZ; k++)
11         for(j = 0; j < sizeY; j++)
12             for(i = 0; i < sizeX; i++)
13                 ApplyForth(pPhiC,pPhiA,pPhiB,i,j,k);
14
15      #pragma omp barrier
16
17     for(k = 0; k < sizeZ; k++)
18         for(j = 0; j < sizeY; j++)
19             for(i = 0; i < sizeX; i++)
20                 ApplyError(pPhiA,pPhiA,pPhiC,i,j,k);
21 }
```

Figure 7-6: Barriers implementing dependencies restrictions

```
    #pragma omp parallel for
    for(k = 0; k < sizeZ; k++)
        for(j = 0; j < sizeY; j++)
            for(i = 0; i < sizeX; i++)
                ApplyBack(pPhiB,pPhiA,pPhiA,i,j,k)

    #pragma omp parallel for
    for(k = 0; k < sizeZ; k++)
        for(j = 0; j < sizeY; j++)
            for(i = 0; i < sizeX; i++)
                ApplyForth(pPhiC,pPhiA,pPhiB,i,j,k)

    #pragma omp parallel for
    for(k = 0; k < sizeZ; k++)
        for(j = 0; j < sizeY; j++)
            for(i = 0; i < sizeX; i++)
                ApplyError(pPhiA,pPhiA,pPhiC,i,j,k)
```

Figure 7-7: Possible choice for parallel region of the advection operation using three different regions

It is also worth to be mentioned that it is possible to parallelize the entire advection call in the main loop (Fig 7-8) but this solution has a main drawback: As the parallel region covers all the step loop context, all parallel code will be scheduled using the same strategy, for example the static scheduler which is the default one. Moreover, if the advection call is parallelized, the IO operations that are needed in order to print the results have to be serialized. This serial operations need to be executed only by one process, so a special region needs to be created as showing in Fig 7-9.

```
#pragma omp parallel
{
    for(int i = 0; i < steeps; i++)
        ...
        Advection();
        ...
}

void Advection()
{
    for(k = 0; k < sizeZ; k++)
        for(j = 0; j < sizeY; j++)
            for(i = 0; i < sizeX; i++)
                ApplyBack(pPhiB,pPhiA,pPhiA,i,j,k)

    ...
}
```

Figure 7-8: Possible choice for parallelizing the advection call

```
#pragma omp parallel
{
    for(int i = 0; i < steeps; i++)
        ...
        Advection();
        ...
        #pragma omp serial
        {
            if(print_steep)
                printResults()
        }
}
```

Figure 7-9: Possible serialization situation

To sum up, the characteristic of not having dependencies between elements in the same step allows us to parallelize the matrix traverse loops, but dependencies exists between the three steps, so the execution order has to be maintained. Among the different ways to parallelize them, we have chosen to parallelize each loop separately to exploit the implicit barrier system at the same time that we keep the original execution order.

Once the general skeleton of the parallelism has been stablished the specifics needs to be addressed. All three steps are contained in nested loops that iterate over the three dimensions of the matrix and since there is no real difference in the computational complexity of the three steps ($O(N^3)$ being N the side of the matrix) the same approach can be used for every section.

As shown in Fig 7-4 each step consists of a loop over the three dimensions of the matrix, and parallelism can be applied to each one of this loops. Furthermore, as there is no restriction in how much OpenMP clauses can be combined together, it is possible to combine parallelism in two or more loops at the same time.

In the first alternative only one loop is parallelized (Fig 7-10). The choice of the parallelized loop has a direct impact in the performance of the code as the granularity level of the problem changes. As shown in Fig 7-11 each time a pragma is reached by the runtime of the application N number of threads are created.

This creation of threads has an associated a cost which is considered as an overhead. Hence, as we increase the number of times that the code has to create new threads, or in other words parallelizing the inner-most loops, which result in greater overheads in the application.

```
1  #pragma omp for
2  for(k = 0; k < sizeZ; k++)
3      for(j = 0; j < sizeY; j++)
4          for(i = 0; i < sizeX; i++)
5              ApplyStep(pPhiB,pPhiA,pPhiA,i,j,k);
6
7  for(k = 0; k < sizeZ; k++)
8      #pragma omp for
9      for(j = 0; j < sizeY; j++)
10         for(i = 0; i < sizeX; i++)
11             ApplyStep(pPhiB,pPhiA,pPhiA,i,j,k);
12
13 for(k = 0; k < sizeZ; k++)
14     for(j = 0; j < sizeY; j++)
15         #pragma omp for
16         for(i = 0; i < sizeX; i++)
17             ApplyStep(pPhiB,pPhiA,pPhiA,i,j,k);
```

Figure 7-10: Single loop parallelization possibilities

As an example, if we consider a case with $N = 64$ parallelizing the outermost loop will result in the creation 2 threads for the entire execution of the code, parallelizing the middle loop will mean creating 128 threads, parallelizing the innermost loop will result into 8192 threads being created. Furthermore, parallelizing the inner loops of the code will cause that the fraction of time that the code is being executed in parallel is slightly smaller than when parallelizing the most external loops, which will result in a further reduction of the performance. Table 7.2 show this difference in performance using 2 threads over the different loop levels.

Additionally Fig 7-12 and Fig 7-13 show alternative possible implementations of the parallelism but in this case applied to two or all the three loops at the same time. In this situation not only the loops parallelized are critical but the number of total threads also plays a special role. Fig 7-14 shows which is the behavior for one loop(A) two loops(B) and three loops(C) parallelized at the same time running with $num\_threads = 2$. The ideal case 7-14-A maintains 2 threads during the

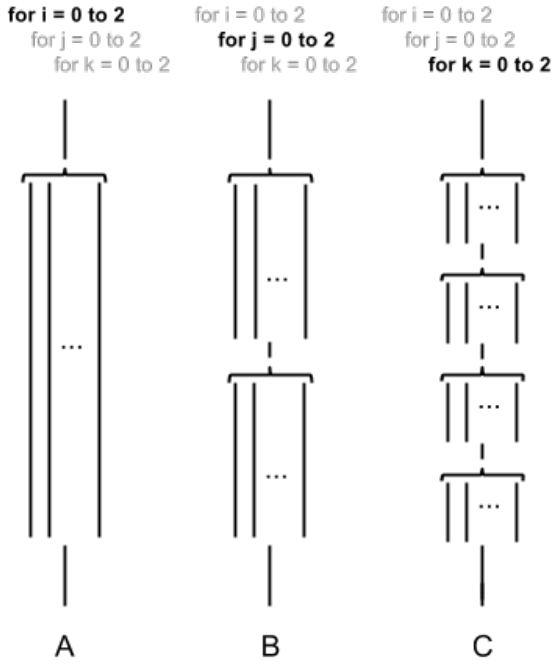

Figure 7-11: Thread creation and merge for outermost (A) middle (B) and innermost (C) loops of size 2 (only first iteration show)

execution. Case 7-14-B duplicates the number of simultaneous threads running (each thread from the previous loop is again divided in N) finally in 7-14-C it can be seen that the number of simultaneous threads running has increased to 8. In conclusion.

```
1 #pragma omp for
2 for(k = 0; k < sizeZ; k++)
3     #pragma omp for
4     for(j = 0; j < sizeY; j++)
5         #pragma omp for
6         for(i = 0; i < sizeX; i++)
7             ApplyStep(pPhiB,pPhiA,pPhiA,i,j,k);
```

Figure 7-12: 3-loop parallelization possibilities

This solution only would be good if we know the characteristics of the machine where we run and it happens that the number of threads that can be launch simultaneously is a power of 2 with minimum of $NThreads^{NLoops}$. Table 7.3 shows the results for the different combinations using 4 threads (as is the minimum

```
#pragma omp for
for(k = 0; k < sizeZ; k++)
    #pragma omp for
    for(j = 0; j < sizeY; j++)
        for(i = 0; i < sizeX; i++)
            ApplyStep(pPhiB,pPhiA,pPhiA,i,j,k);

for(k = 0; k < sizeZ; k++)
    #pragma omp for
    for(j = 0; j < sizeY; j++)
        #pragma omp for
        for(i = 0; i < sizeX; i++)
            ApplyStep(pPhiB,pPhiA,pPhiA,i,j,k);

#pragma omp for
for(k = 0; k < sizeZ; k++)
    for(j = 0; j < sizeY; j++)
        #pragma omp for
        for(i = 0; i < sizeX; i++)
            ApplyStep(pPhiB,pPhiA,pPhiA,i,j,k);
```

Figure 7-13: 2-loop parallelization possibilities

number of threads if we want to parallelize at least two loops simultaneously)

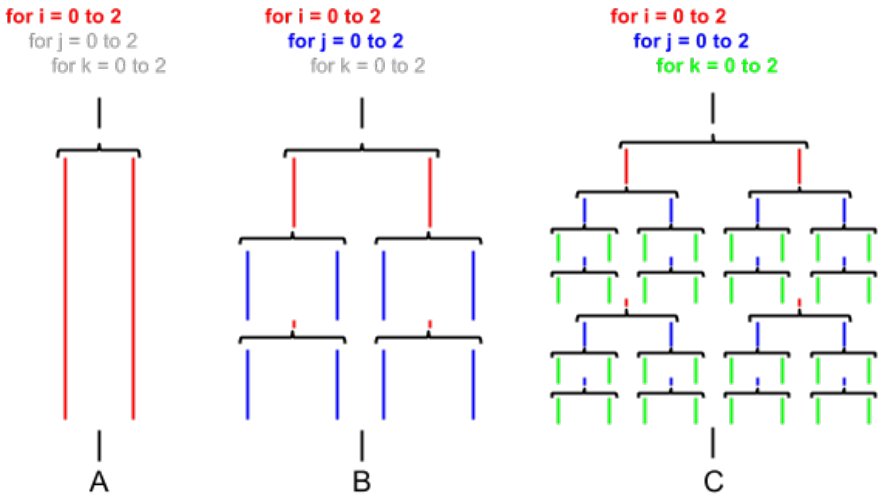

Figure 7-14: Threads with no nesting level (A) one nesting level (B) and two nesting levels(C) for 2 processors (only first iteration show)

More tests have also been conducted in order to observer the difference of time using some of the additional balance schemes that OpenMP offers. Dynamic and Guided schemes have been tested using chunk values of 5 and 50.

The results in the table 7.4 show that no real improvement is appreciable in any of the cases tested, moreover, a drop of performance is observed in general while using chunks of size 50 and again while parallelizing the innermost loop, both with 2 and 4 threads.

Optimization	Time	Speedup	Efficiency
No parallelization	26.1819	1.00	100.00%
k-loop	12.8771	2.03	100.00%
j-loop	13.1920	1.98	99.23%
i-loop	19.18556	1.36	68.23%

Table 7.2: Impact of parallelization granularity for one loop

Optimization	Time	Speedup	Efficiency
No parallelization	26.18197	1.00	100.00%
k-loop	7.23150	3.62	90.51%
j-loop	7.22840	3.62	90.51%
i-loop	16.07290	1.62	40.72%
kj-nested-loop	12.93513	2.02	50.60%
ji-nested-loop	15.77619	1.65	41.48%
ki-nested-loop	14.86704	1.76	44.02%

Table 7.3: Impact of parallelization granularity for two loops

Optimization	Time 2 cores	Time 4 cores	Speedup-2	Speedup-4
No parallelization	26.18197	26.18197	1.00	1.00
k-loop, dynamic 5	13.29184	7.36262	1.97	3.56
j-loop, dynamic 5	13.76279	8.05539	1.90	3.25
i-loop, dynamic 5	23.26886	21.50931	1.13	1.22
k-loop, dynamic 50	20.61854	22.90412	1.27	1.14
j-loop, dynamic 50	22.45688	21.87780	1.17	1.20
i-loop, dynamic 50	28.71933	32.41115	0.91	0.81
k-loop, guided 5	13.00510	7.95700	2.00	3.29
j-loop, guided 5	13.17151	7.94915	1.99	3.29
i-loop, guided 5	23.55007	21.61508	1.11	1.21
k-loop, guided 50	20.62844	24.31840	1.27	1.08
j-loop, guided 50	21.43830	22.23170	1.22	1.18
i-loop, guided 50	28.30279	33.07729	0.93	0.79

Table 7.4: Impact of parallelization schedulers

7.4.2 Parallelization of the advection based on tasks

It has been shown that one of the problems parallelizing this code is caused by the dependencies between the $back$, $forth$ and $error$ steps. It cannot be avoided easily using loop based parallelism and hence an approach based on tasks implemented with OmpSs [10] [11] is presented.

Both OpenMP tasks and OmpSs allow us to parallelize the code using tasks instead of loops. The principal difference between tasks and loops relies in the fact that when the code is executed a parallel loop is executed at the moment, while task are only spawned and executed whenever is possible according to some rules and the number of threads available.

```
1
2  #pragma parallel for in(pPhiA) out(pPhiB)
3  for(k = 0; k < sizeZ; k++)
4      for(j = 0; j < sizeY; j++)
5          for(i = 0; i < sizeX; i++)
6              ApplyBack(pPhiB,pPhiA,pPhiA,i,j,k)
7
8  #pragma omp taskwait
9
10 #pragma parallel for in(pPhiA,pPhiB) out(pPhiC)
11 for(k = 0; k < sizeZ; k++)
12     for(j = 0; j < sizeY; j++)
13         for(i = 0; i < sizeX; i++)
14             ApplyForth(pPhiC,pPhiA,pPhiB,i,j,k)
15
16 #pragma omp taskwait
17
18 #pragma parallel for in(pPhiA,pPhiC) out(pPhiA)
19 for(k = 0; k < sizeZ; k++)
20     for(j = 0; j < sizeY; j++)
21         for(i = 0; i < sizeX; i++)
22             ApplyError(pPhiA,pPhiA,pPhiC,i,j,k)
23
24 #pragma omp taskwait
```

Figure 7-15: Advection step using OmpSs, Naive version

The most important characteristic of this parallelization strategy which is exploited in this implementation is the ability to declare dependencies between tasks. These dependencies are indicated using three pragmas which can vary depending on the language, as a guideline we can express them with: in to indicate which data is required to execute the block, out to indicate which data is

available at the end and $inout$ if there is data that is at the same time a requirement and a result.

In our code, a first naive implementation can be obtained by using the clauses in and out over the different matrices that store the partial results of the three steps, as shown in Fig 7-15. Also notice that since we are specifying the whole matrix as our output result, we need to wait for all threads outputting that matrix in order to be true. We use the $taskwait$ directive as an equivalent to the barrier.

This optimization alone does not run the code in parallel at all. Firstly, because we are telling the runtime to create three tasks one for each step, and secondly because all these tasks have a taskwait directive just below. Blocks here, and specifically as commented in Sec. 7.4.1 are the key to avoid the use of barriers and hence the extra performance desired.

The idea behind this new implementation is that not every data in the matrix is actually needed to execute the next step. Two new characteristics of the algorithm are used to make this statement. Firstly, the CFL parameter, and second, the possibility of using a block-based version of the algorithm without incurring in a performance penalty.

As stated before CFL used to increase the accuracy of the result, and basically forces the maximum distance that our particles will be able to move in a single time step. A CFL = N typically means that at max, our particles will move N elements length from our current position as shown in 7-16. If we move this fact to the block world for a problem of size N, and specifically we consider that our blocks are of size NxNxK (in other words, slices of the cube with K width) it is clear to see that to calculate each slice, we will need the data of the N slices of width 1 above and below the current one. In fig 7-17 we can see these dependencies, from now on called Z-dependency in terms of slices between the different steps of the advection.

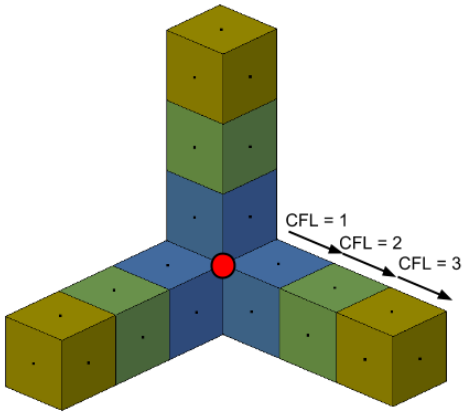

Figure 7-16: maximum particle displacement with a given CFL

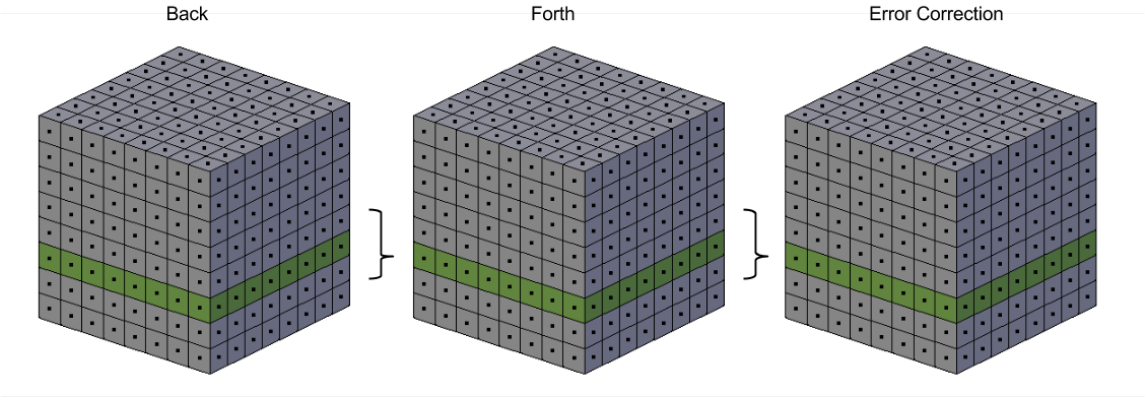

Figure 7-17: Z-dependency for a block (in green) of size N=1 nad CFL=1 for each step

Hence, as far as we guaranty that a slice of the $forth$ step is going to modify data that will be no longer need by any of the slices in the $back$ step, it is safe to modify that concurrently. In general this becomes specially easy to indicate if we guaranty that our slices at least have width = CFL, which means that every block will be safe two execute if is at least two slots before the previous step, as shown in Fig 7-18

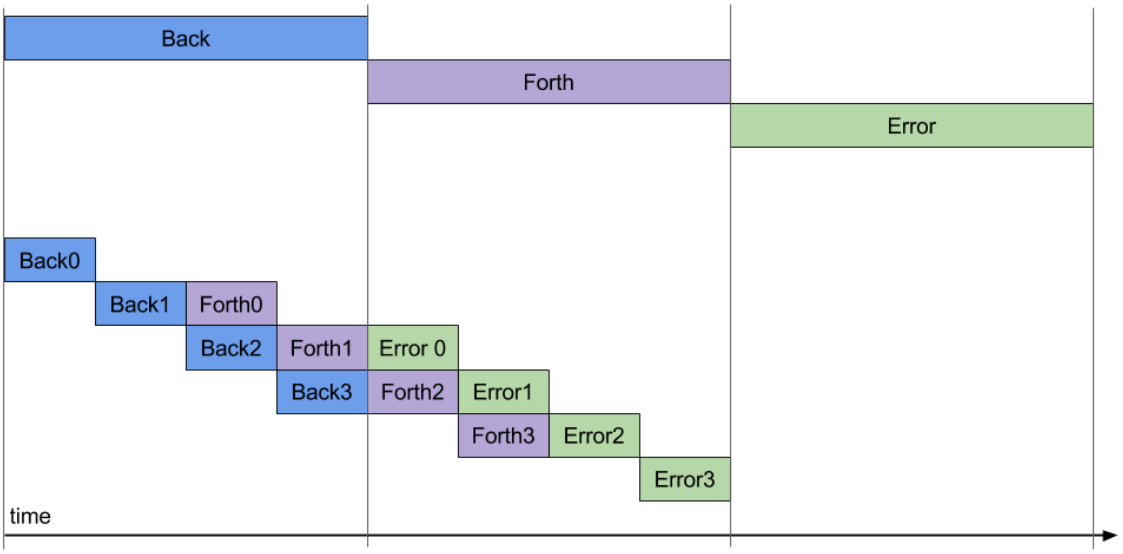

Figure 7-18: Execution timeline decomposed in slices

Code 7-19 shows how the final implementation of the algorithm is made. Notice that all taskwaits except the last one, which is needed to correctly finish the iteration, have been removed since all the dependencies now are mixed between the three steps

```
1
2  int slice_size;
3
4  for(slice = 0; k < num_slices; slice++)
5      #pragma parallel for
6          in(pPhiA[slice-1:3*slice_size])
7          out(pPhiB[slice:slice_size])
8      for(k = sliceBegib; k < sliceEnd; k++)
9          for(j = 0; j < sizeY; j++)
10             for(i = 0; i < sizeX; i++)
11                 ApplyBack(pPhiB,pPhiA,pPhiA,i,j,k)
12
13 for(slice = 0; k < num_slices; slice++)
14     #pragma parallel for
15         in(pPhiA[slice-1:3*slice_size],
16            pPhiB[slice-1:3*slice_size])
17         out(pPhiC[slice:slice_size])
18     for(k = sliceBegib; k < sliceEnd; k++)
19         for(j = 0; j < sizeY; j++)
20             for(i = 0; i < sizeX; i++)
21                 ApplyForth(pPhiC,pPhiA,pPhiB,i,j,k)
22
23 for(slice = 0; k < num_slices; slice++)
24     #pragma parallel for
25         in(pPhiA[slice-1:3*slice_size]
26            pPhiC[slice-1:3*slice_size])
27         out(pPhiA[slice:slice_size])
28     for(k = sliceBegib; k < sliceEnd; k++)
29         for(j = 0; j < sizeY; j++)
30             for(i = 0; i < sizeX; i++)
31                 ApplyError(pPhiA,pPhiA,pPhiC,i,j,k)
32
33 #pragma omp taskwait
```

Figure 7-19: Advection step using OmpSs

Chapter 8

Results of the advection optimization

In this section we are going to present the results of the optimizations. All the results have been run under three machine configurations being the first one a standard desktop computer and the other two computing clusters with different node characteristics:

Profile 1: MPP2 Node in HLRN:

1. 2x Intel Xeon E5-2680v3 12-Core at 2,50 GHz
2. 64 GB of RAM memory, 30MB of cache memory

Profile 2: Minotauro Node in BSC:

1. 2x Intel Xeon E5649 6-Core at 2,53 GHz
2. 24 GB of RAM memory, 12MB of cache memory

The objective of improving the precision in the advection step, as seen already in the validation section, has been successfully achieved for the advection step. The second objective was to implement the code in such a way that would be scalable for a relatively small number of cores in order to run in computer nodes.

Here we present a series of scalability tests under different conditions to prove that this has been achieved as well.

The first set of results has been executed in the machine with the profile 2, the HLRN under a normal OpenMP parallelization with all the optimizations that decreased the time enabled:

Cores	Time	Speedup	Efficiency
1	25.09724	1.00	100.00%
3	8.99470	2.79	93.01%
6	4.82322	5.20	86.72%
12	2.92848	8.57	71.42%
24	2.19847	11.42	47.57%

Table 8.1: BFECC scalability for size 32x32x32, 10000 iterations, OpenMP based and computer profile 1

Cores	Time	Speedup	Efficiency
1	22.18188	1.00	100.00%
3	8.09783	2.74	91.31%
6	4.50735	4.92	82.02%
12	2.46644	8.99	74.95%
24	1.24995	17.75	73.94%

Table 8.2: BFECC scalability for size 64x64x64, 1000 iterations, OpenMP based and computer profile 1

Cores	Time	Speedup	Efficiency
1	19.26885	1.00	100.00%
3	6.88481	2.80	93.29%
6	3.67868	5.24	87.30%
12	1.95007	9.88	82.34%
24	1.09095	17.66	73.59%

Table 8.3: BFECC scalability for size 128x128x128, 100 iterations, OpenMP based and computer profile 1

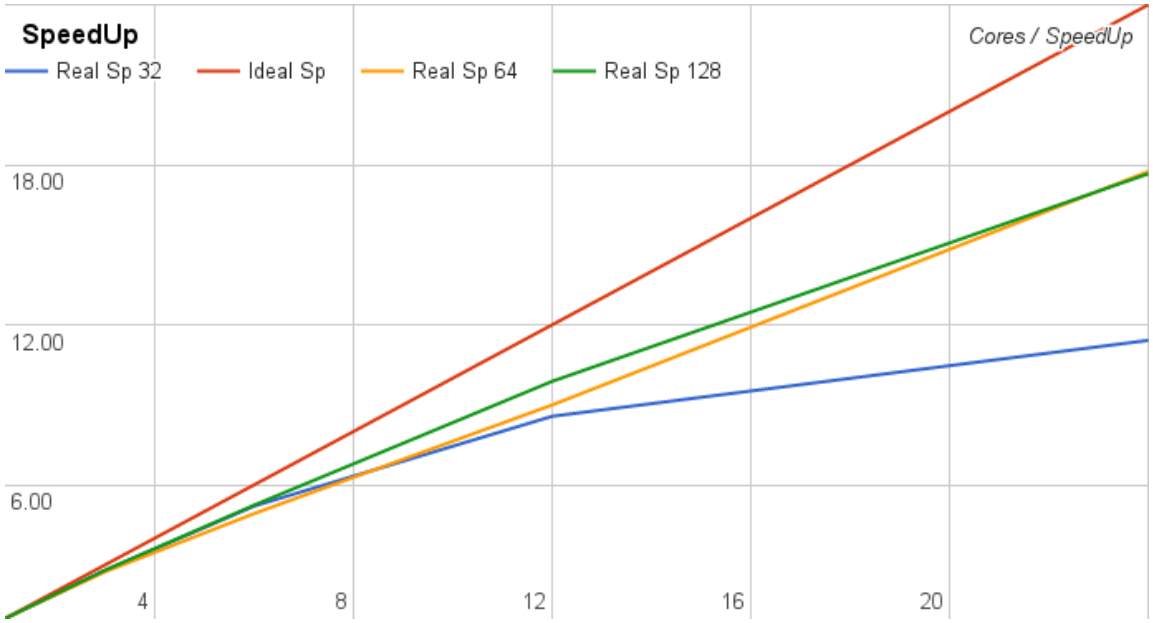

Figure 8-1: Scalability for HLRN machine using OpenMp

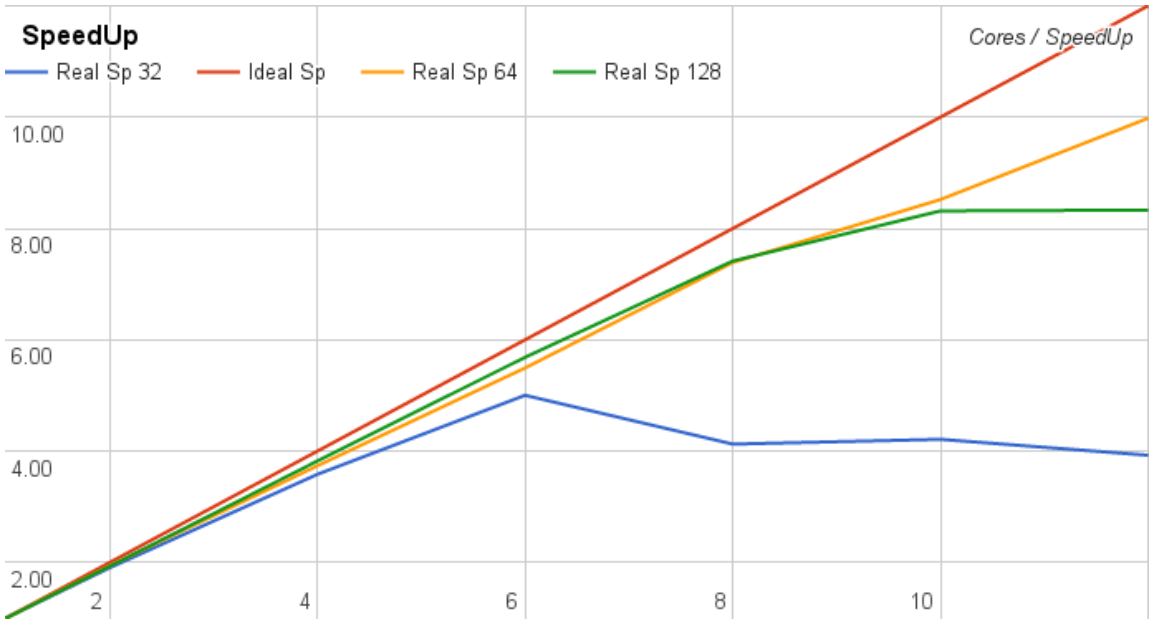

Figure 8-2: Scalability for minotauro machine using OpenMp

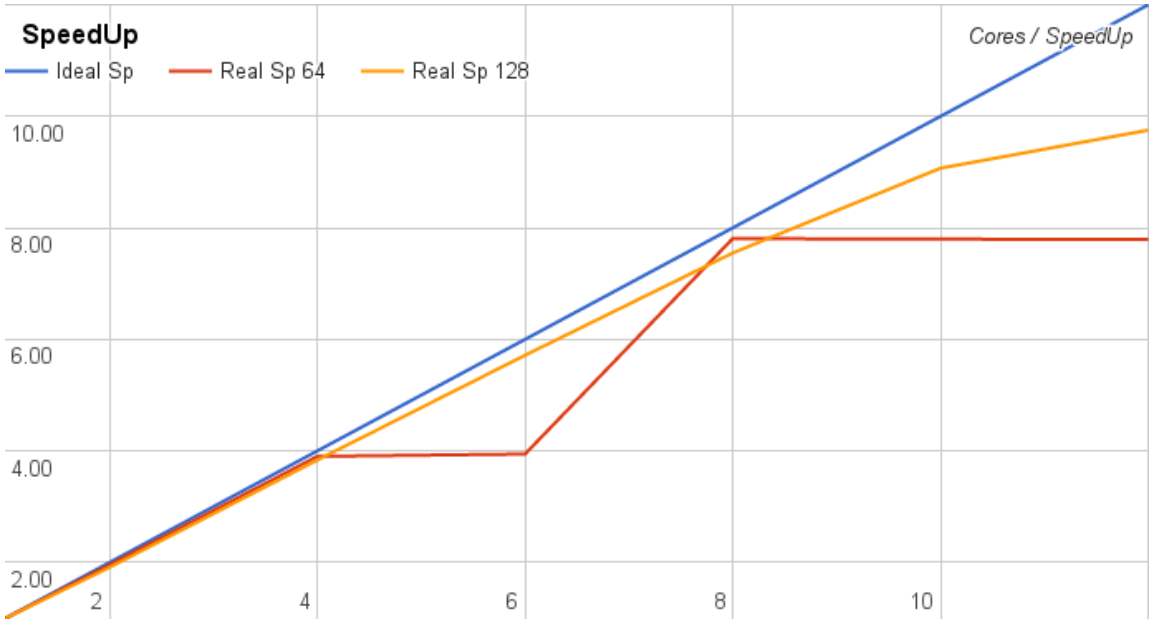

Figure 8-3: Scalability for minotauro machine using OpenMp and blocks

Figure 8-4: Scalability for minotauro machine using OmpSs

By the amount of code hat is executed in parallel we would expect to obtain values that are above 80% of efficiency. At the sight of these results we can say that the code achieves a degree of parallelism but it starts to have a suboptimal behavior when it reaches approximately the barrier of 6-12 processors.

We also observe an interesting characteristic, as the 32 side case behave notably worse than the 64 and 128 side tests. As we are parallelizing the outmost loop of the computation is straightforward to see that, once the first 24 only 6 of them (as we never iterate over the boundary) will be executed in the second half of the block. We can check that this is although the expected value for this situation by simply calculating the ideal speedup of this situation which would be:

$$0.74T/24 + 0.25T/6 = 0.073T$$

And check that our result is proximately 75% of the as optimal as the ideal, which keeps in the mean. Moreover we can check that this value is obtained as well for 64 and 128 cases.

The 64 and 128 behave identically as there is always enough data to process in parallel but they have, in a minor extend the same problem that was detected in the 32 size example as the number of processors does not divide exactly the size of our problem.

Cores	Time	Speedup	Efficiency
1	30.95927	1.00	100.00%
2	16.25343	1.90	95.24%
4	8.63942	3.58	89.59%
6	6.18741	5.00	83.39%
8	7.50232	4.13	51.58%
10	7.34968	4.21	42.12%
12	7.88506	3.93	32.72%

Table 8.4: BFECC scalability for size 32x32x32, 10000 iterations, OpenMP based and computer profile 2

Cores	Time	Speedup	Efficiency
1	22.87972	1.00	100.00%
2	11.79220	1.94	97.01%
4	6.12282	3.74	93.42%
6	4.16426	5.49	91.57%
8	3.09669	7.39	92.36%
10	2.68537	8.52	85.20%
12	2.29229	9.98	83.18%

Table 8.5: BFECC scalability for size 64x64x64, 1000 iterations, OpenMP based and computer profile 2

Cores	Time	Speedup	Efficiency
1	19.96572	1.00	100.00%
2	10.33464	1.93	96.60%
4	5.22316	3.82	95.56%
6	3.51142	5.69	94.77%
8	2.69236	7.42	92.70%
10	2.40111	8.32	83.15%
12	2.39656	8.33	69.42%

Table 8.6: BFECC scalability for size 128x128x128, 100 iterations, OpenMP based and computer profile 2

Cores	Time	Speedup	Efficiency
1	24.14245	1.00	100.00%
2	12.23714	1.97	98.64%
4	6.19502	3.90	97.43%
6	4.20349	5.74	95.72%
8	3.32786	7.25	90.68%
10	2.73771	8.82	88.18%
12	2.38867	10.11	84.23%

Table 8.7: BFECC scalability for size 64x64x64, 1000 iterations, OmpSs based and computer profile 2

Cores	Time	Speedup	Efficiency
1	20.16828	1.00	100.00%
2	10.14235	1.99	96.60%
4	5.04678	4.00	99.91%
6	3.42740	5.88	98.07%
8	2.63335	7.66	95.73%
10	2.14128	9.42	94.19%
12	1.82054	11.08	92.32%

Table 8.8: BFECC scalability for size 128x128x128, 100 iterations, OmpSs based and computer profile 2

Chapter 9

GPU parallelization of the advection

Once a regular CPU implementation of the code has been presented and proven to work under different parallel schemes and taking into account that the structures of this problem present some desirable characteristics for GPU computing an implementation using CUDA is presented.
We have seen that the code implemented basically consists on operating over matrices which ideally are powers of two. Along with this, the operations we have to perform present high spatial locality (advection) or are directly stencils (diffusion). This characteristics make this code especially interesting to be implemented with GPU
Specifically, we are going to focus again in the advection step. The implementation in CUDA is very similar to the ones used in the OpenMP section.

9.1 GPGPU

GPU were initially designed as a side processing units for video-game graphics. This units are specialized in operations involving visual elements such as texture interpolations and matrix operations. The architecture of a GPU allow this calculations to be performed much faster than in a regular processor by the use of their elevate number of concurrent threads, but paying the price of having very

simple computing units. In 2007 NVIDIA released CUDA, a GPGPU programming language that allow the explicit programming of these devices opening a new opportunity in the HPC world.

9.1.1 Introduction CUDA

Before begin with the implementation is worth to say that not every code in the market is a good choice to be ported to GPU. In order to understand why some codes are suitable of being implemented in that devices and other cannot (or can but with a poor performance) we are going to explain some of the most important characteristics of the GPU's and how they affect the normal life-cycle of a program.

There are basically two main changes that need to be addressed. Firstly the execution model and secondly the memory model.

9.2 Execution Model

The execution model is different of which we are used in a CPU. These feature a low number of processors that can be used to execute our code, typically a normal desktop computer would have up to 4 cores while HPC nodes can have up to 24. GPU's have around of 128 cores per SM, and it is usual to see from two of these chips (laptops GPU) to eight (Desktop GPU) or even twelve (Professional GPU) of them, meaning values in the order of hundreds to thousands of cores.

All executions of a program hence, have to be inherently parallel and there is a very specific way to split the problem in order to exploit such massive parallelism.

Instead of run our programs directly on a CPU CUDA will see all our work as a grid. This grid of work is divided into blocks which have a given dimension (X, Y and Z coordinates). The combination of different blocks will process all the

elements of a grid. Each block is assigned to a SM (Stream Multiprocessor) at the beginning of the execution and it remains assigned there until the end. Each SM can run one block at the same time, so in any given point we can have as much blocks as SM running concurrently.

Each block is then divided at the same time into threads, which also will be referenced using x y and z coordinates. The maximum number of threads that can be executed at once in a block is called $warp$ and typically its size is 32. The number of threads inside a warp may be less than 32 for several causes like not having a number of threads multiple of 32 inside the block or if each thread needs more resources than the ones that the SM can provide, in any case we say that we are in a a suboptimal occupancy of the GPU. In an ideal situation we want to execute as much warps as possible concurrently, but one block can have multiple warps on it.

We can see this computational model depicted in figure 9-1

9.3 Memory Model

The second and perhaps most important change is the memory layout present in a GPU. In a GPU the memory is organized as well in a hierarchical way, but with important differences and restrictions that need to be addressed.

Depending on the technology that we use for the GPGPU the name of this memories can change. In order to be as standard as possible we are going to enumerate the ones proposed by CUDA, as is the most extended GPGPU language:

Global memory: This memory is used to hold large amount of data. The size of the global memory is in the range of the GB and is slow. It is also the space of memory used to make transferences between GPU and CPU memory spaces.

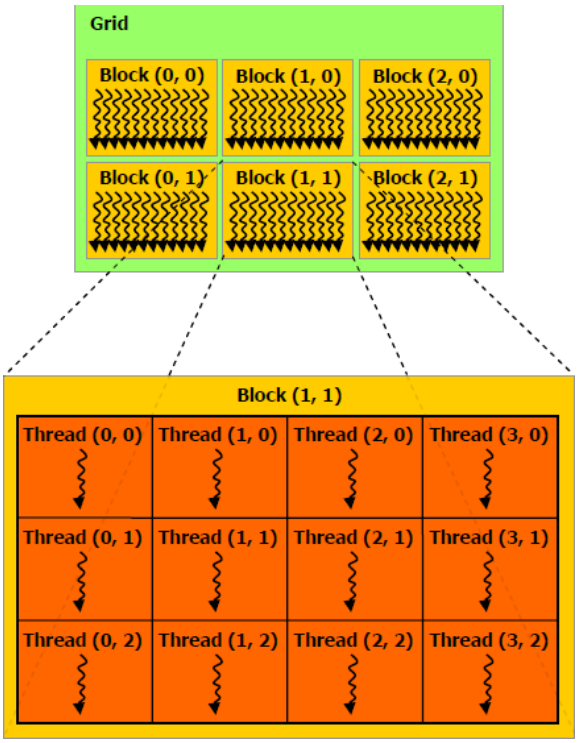

Figure 9-1: Cuda execution model [12]

The trend for this memory is to ”merge” with the CPU memory creating a unified memory space between the device and the host.

Constant memory: This memory is a read-only memory, several times faster than the global memory but much smaller. It is intended to be a place to put data that is small and accessed very frequently.

Texture memory: This memory can be seen as a larger constant memory but optimized to make some geometrical operations like interpolation. This reason for this is that this memory was initially developed to speed the texture calculations, but it can be useful in some scenarios. Most of the textures memories are organized as 2DTexture memories but they can handle 3DTextures as well

Local memory: This is a private memory that every thread has. The characteristics of this memory is that is very fast. But also very small, approximately 64Kb to share among all threads executed at the same time.

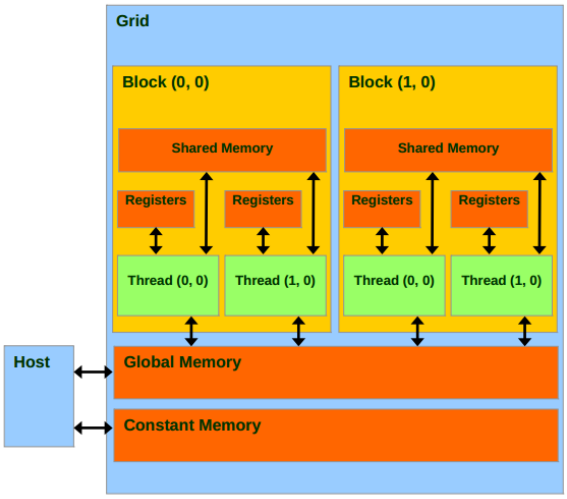

Figure 9-2: CUDA memory model [12]

Shared memory: As its names indicates is a memory that can be accessed by all the threads in a block. It is normally used to make internal communication inside a block. In any case can be used to perform communication between blocks

Registers: Finally, registers are the same as the CPU registers. Its number is limited as well and can limit the number of threads

9.4 Implementation using CUDA

Several parts of our code, as its based mainly on operations over matrices can be ported to GPU where, theoretically, they should benefit from the fact of having many cores to execute, as we have seen that the operations executed are rather simple.

Initially we are going to base our implementation in a naive decomposition of the problem in blocks and see how the optimal solution converges again in a slice-decomposition of the problem.

The first change that needs to be done in the code is to perfectly define which are going to be the size problem, GPU blocks, number of threads, and warp size. In order to choose this parameters correctly we have to fulfil the equation that states that:

$$N = B * T$$

Where N is the size of our problem, B is the number of blocks and T is the number of threads per block. Also notice that, since we have no restriction regarding the size of our blocks, we are going to choose always size multiples of $B * T$.

Since the equation above has multiple solutions in order to maximize the performance we should try to increase the occupancy of the GPU, i.e the number of active threads in the GPU at any given time. Each block will have a maximum number of active warps at given time, let call this MAX_{warp}. If the number of active warps inside a block is lower that the number of maximum active warps, we will have a problem in performance.

For example, if we take:

$$Warpsize = 32$$
$$Problemsize = 2048$$
$$Numberofblocks = 32$$
$$MaxWarpPerBlock = 4$$

We will end having

$$Blocksize = 2048/32 = 64$$
$$WarpsPerblock = 64/32 = 2$$

which will represent an occupancy of 0.5

If we instead take

$$Numberofblocks = 8$$
$$Blocksize = 2048/8 = 256$$
$$WarpsPerblock = 256/32 = 8$$

Which will means that our occupancy is not restricted by the size of the block. As stated in the introduction there are other factors than may reduce our occupancy, but the current kernels do not have any restriction in terms of register or shared memory usage, so there are not problems at this point. In general it will be a good practice to avoid the use of small block sizes as long as possible. As our code is based on a 3D matrix, we have to repeat this calculations for the x, y and z axis of our blocks, as shown in 9-3

```
1
2 #define TILE_X 16
3 #define TILE_Y 16
4 #define TILE_Z  4
5
6 dim3 threads(TILE_X, TILE_Y, TILE_Z);
7 dim3 blocks(SizeX / TILE_X, SizeY / TILE_Y, SizeZ / TILE_Z);
```

Figure 9-3: CUDA Setup size of blocks and threads

9.5 Advection Implementation

As we stated along the document advection operation consists in three different sub-steps, back forth and error correction. In the first naive implementation in GPU we simply execute this operations a kernel advection as shown in 9-4 where h_ and d_ prefixes in the name of the variables indicate if they are either host or device memory. Variable rDx represents the size of the cells, and rDt is the time-step selected

The code first copy the necessary data to the GPU, spawns three kernels corresponding with the operations of the advection and then copy back the solution

to de CPU. Also notice that the code of the kernels is relatively similar as the code un the CPU, as can be seen in code 9-5

```

// Copy input data to GPU
cudaMemcpy(
  d_velocity, h_velocity,
  num_bytes * sizeof(double) * 3,
  cudaMemcpyHostToDevice
);

cudaMemcpy(
  d_phi_a, h_phi,
  num_bytes * sizeof(double),
  cudaMemcpyHostToDevice
);

// Execute the kernels
BackCUDA<<< threads, blocks >>>(
  d_phi_b, d_phi_a, NULL, d_vel,
  rDx, rDt, size
);

ForthCUDA<<< threads, blocks >>>(
  d_phi_c, d_phi_a, d_phi_b, d_vel,
  rDx, rDt, size
);

EccCUDA<<< threads, blocks >>>(
  d_phi, d_phi, NULL, d_vel,
  rDx, rDt, size
);

// Copy back the results to GPU
cudaMemcpy(
  h_phi_a, d_phi_a,
  num_bytes * sizeof(double),
  cudaMemcpyDeviceToHost
);
```

Figure 9-4: Naive CUDA execution

The result of the code in 9-4 can be represented over a time-line as shown in 9-6. We can appreciate several problems. Without entering in the discussion about if the total time of the computation is bigger or lower than the CPU we can see that more than 50% of the time spend by an iteration consist in copying back and forth memory between the device and the GPU.

In order to solve this problem we can use Pinned memory. Pinned memory is a special way of allocating full pages in the host that are transferred up to 4 times faster (we can use all the bandwidth available on the GPU) to the device (fig 9-7).

```
1
2  __global__ void BackCUDA(
3      double * out,
4      double * PhiAux, double * vel,
5      const double dx, const double idx, const double dt, const int N,
6      const int ii, const int jj, const int kk) {
7
8    int i = blockIdx.x * blockDim.x + threadIdx.x + ii;
9    int j = blockIdx.y * blockDim.y + threadIdx.y + jj;
10   int k = blockIdx.z * blockDim.z + threadIdx.z + kk;
11
12   double dsp[3];
13
14   if (i > 0 && j > 0 && k > 0 && i < N - 1 && j < N - 1 && k < N - 1) {
15
16     dsp[0] = fma((double)i, (double)dx, (double)(-vel[__GETINDEX(i,j,k)*3+0] * dt));
17     dsp[1] = fma((double)j, (double)dx, (double)(-vel[__GETINDEX(i,j,k)*3+1] * dt));
18     dsp[2] = fma((double)k, (double)dx, (double)(-vel[__GETINDEX(i,j,k)*3+2] * dt));
19
20     InterpolateCUDA(dsp, PhiAux, &out[__GETINDEX(i, j, k)], idx, N);
21   }
22 }
```

Figure 9-5: Example of CUDA kernel for the Back step of the advection

Once done this, we can see how the transfers times have decreased visibly (last copy back not shown) in 9-8.

Although the time of the transfers can be decreased using pinned memory there is still a gap between computation and transfer time. There are several well-known techniques to overcome this problem. We are going to focus in the double buffering technique for our problem.

The idea behind this model is to transfer a small portion of the work that one must do and, while being calculated start to transfer another portion of work. Our case can be easily dive into slides which are executed one after another in the exact same way we scheduled for the OpenMP Task-Based parallelism. The results obtained in 9-10 and the code can be seen at figure 9-9

This implementation still have some problems. As we can clearly see in the time-line we have divided the problem in 4 slices, but we lose the control in which the kernels are executed. For example in figure 9-10 that the first ”Forth” kernel is executed just after the first ”back” kernel has ended, which would lead

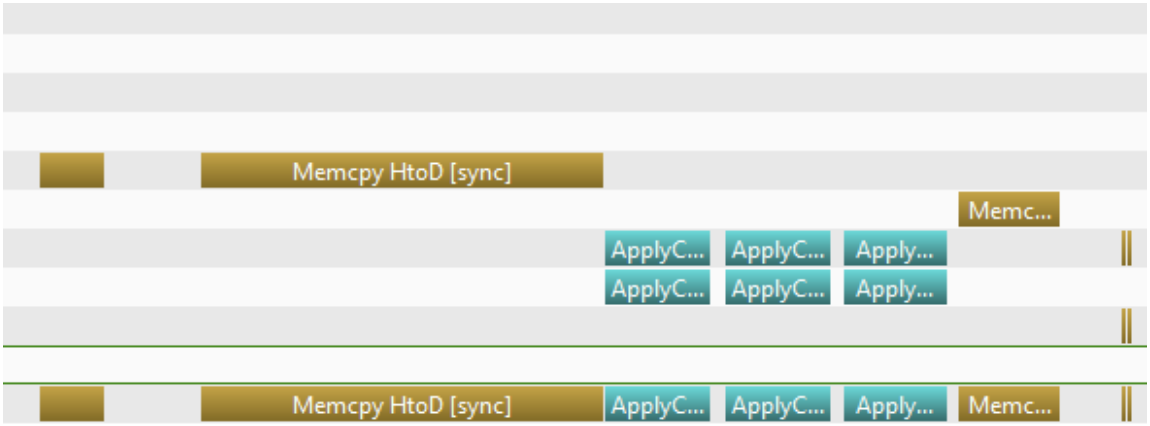

Figure 9-6: Initial GPU timeline

```
1 cudaMallocHost((void**)mem, size)
```

Figure 9-7: Declaration of pinned memory

to a numerical error, as the information of the 2nd "back" kernel is also needed while calculating the sides of the slice.

In order to overcome this problem the GPU streamlines can be used. The streamlines let us specify dependencies between different kernels of the code. In our case, back kernels will always be dependent of their portion of memory. Forth i will be dependent of Back kernel $i-1$, and finally Error correction kernel j will be dependent of Forth kernel $j-1$. in figure 9-11 we can see how the time-line changes after applying the correct dependences of the Back kernel, and in the figure 9-12 we can see the final time-lien after all dependencies are set correctly this time with 8 slices.

There are still a couple of problems that need to be address to hide the computational part. First, there we can overlap the communications back to the GPU, resulting in 9-13. Also, as we can see in 9-12 and 9-13 the combined time of the Back, Forth and ErrorCorrection kernels is still bigger than the time it takes to the GPU to transfer the next chunk of memory. In order to solve this problem, we can dynamically calculate the size of the slices of the problem in order to use more or less according to our necessities. Using 16 slices instead of 8 in our

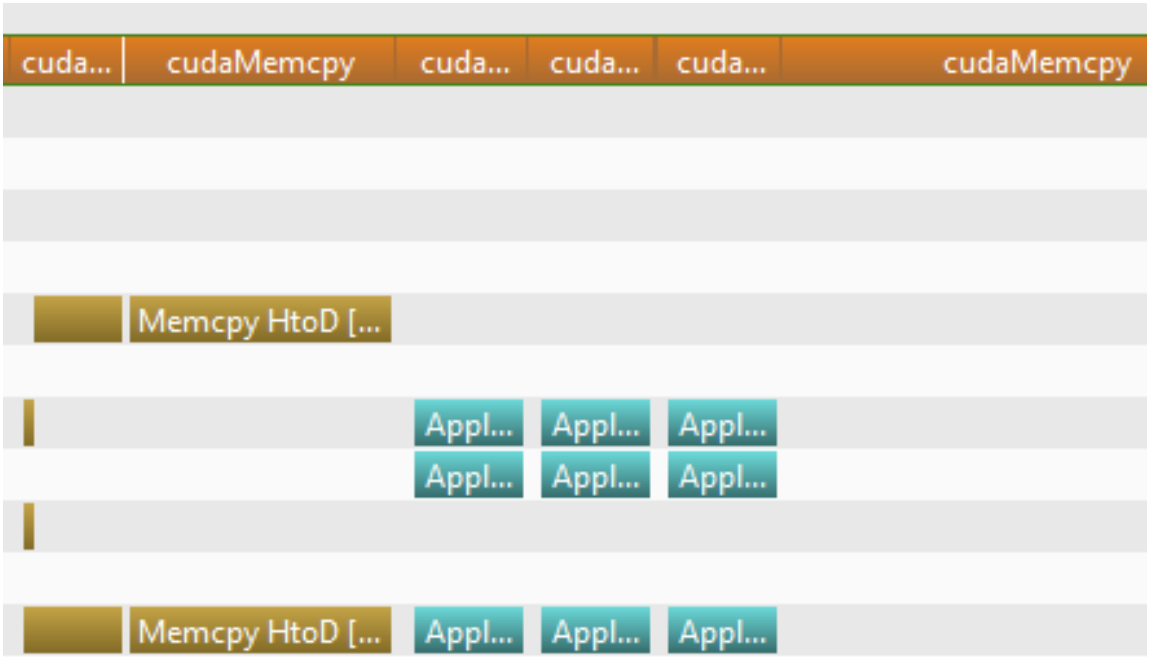

Figure 9-8: Pinned memory

problems results in the time-line shown in figure 9-14 where there is almost no time wasted in transfers. The final code with the fully parallelized transfer can be shown in 9-15 (Initialization), 9-16 (Body) and 9-17 (Trailer).

```
1
2  // Define the number of slices ( blocks in Z )
3  BlocksInX = 1;
4  BlocksInY = 1;
5  BlocksInZ = 4;
6
7  ELX = (X + OverlapX) / BlocksInX;
8  ELY = (Y + OverlapY) / BlocksInY;
9  ELZ = (Z + OverlapZ) / BlocksInZ;
10
11 // Define the dimensions of our problem
12 dim3 threads(TILE_X, TILE_Y, TILE_Z);
13 dim3 blocks(rX / TILE_X, rY / TILE_Y, rZ / TILE_Z);
14
15 // Copy input data to GPU
16 cudaMemcpy(
17   d_velocity, h_velocity,
18   num_bytes * sizeof(double) * 3,
19   cudaMemcpyHostToDevice
20 );
21
22 cudaMemcpy(
23   d_phi_a, h_phi,
24   num_bytes * sizeof(double),
25   cudaMemcpyHostToDevice
26 );
27
28 // Launch the kernels for each slice
29 for (int i = 0; i < BBX; i++)
30   for (int j = 0; j < BBY; j++)
31     for (int k = 0; k < BBZ; k++)
32       BackCUDA <<< threads, blocks >>>(
33         d_PhiB, d_PhiA,
34         d_vel, rDx,
35         rDt, rX + rBW,
36         i * ELX, j * ELY, k * ELZ
37       );
38
39 for (int i = 0; i < BBX; i++)
40   for (int j = 0; j < BBY; j++)
41     for (int k = 0; k < BBZ; k++)
42       ForthCUDA<<< threads, blocks >>>(
43         d_PhiC, d_PhiA,
44         d_PhiB, d_vel, rDx,
45         rDt, rX + rBW,
46         i * ELX, j * ELY, k * ELZ
47       );
48
49 for (int i = 0; i < BBX; i++)
50   for (int j = 0; j < BBY; j++)
51     for (int k = 0; k < BBZ; k++)
52       EccCUDA  <<< threads, blocks >>>(
53         d_PhiA, d_PhiC,
54         d_vel, rDx,
55         rDt, rX + rBW,
56         i * ELX, j * ELY, k * ELZ
57       );
58
59 // Copy back the results to GPU
60 cudaMemcpy(
61   h_phi_a, d_phi_a,
62   num_bytes * sizeof(double),
63   cudaMemcpyDeviceToHost
64 );
```

Figure 9-9: CUDA execution by slices

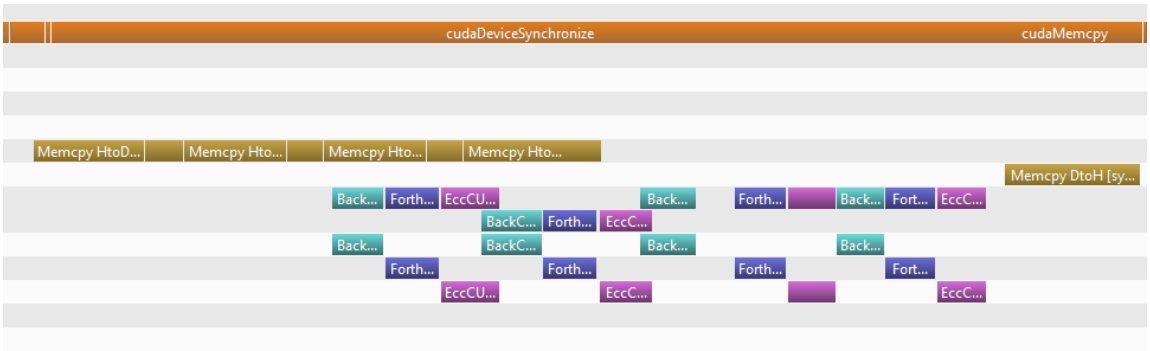

Figure 9-10: Overlapping

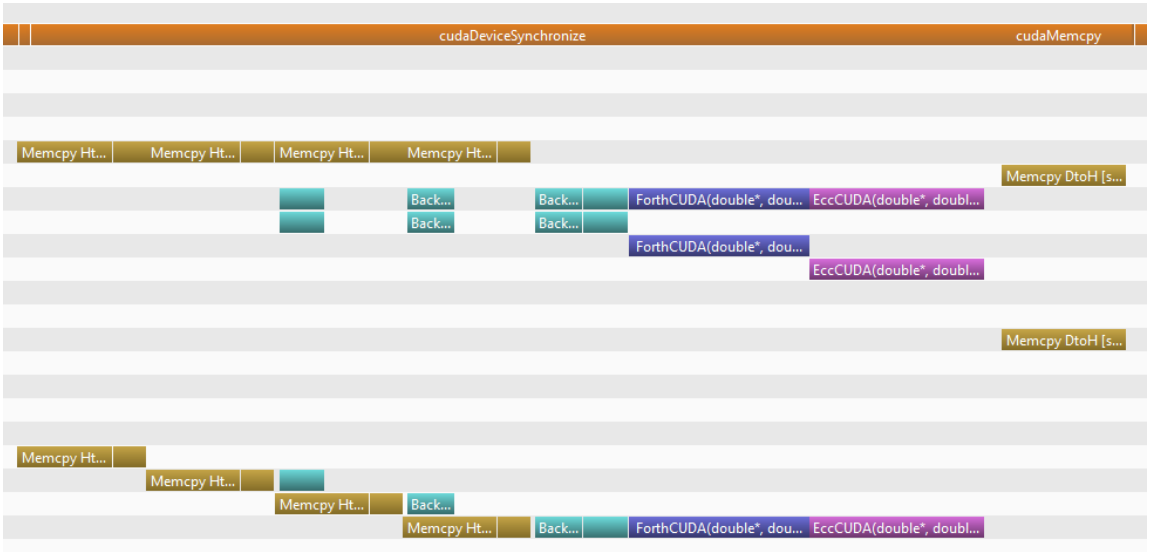

Figure 9-11: Correct overlapping GPU

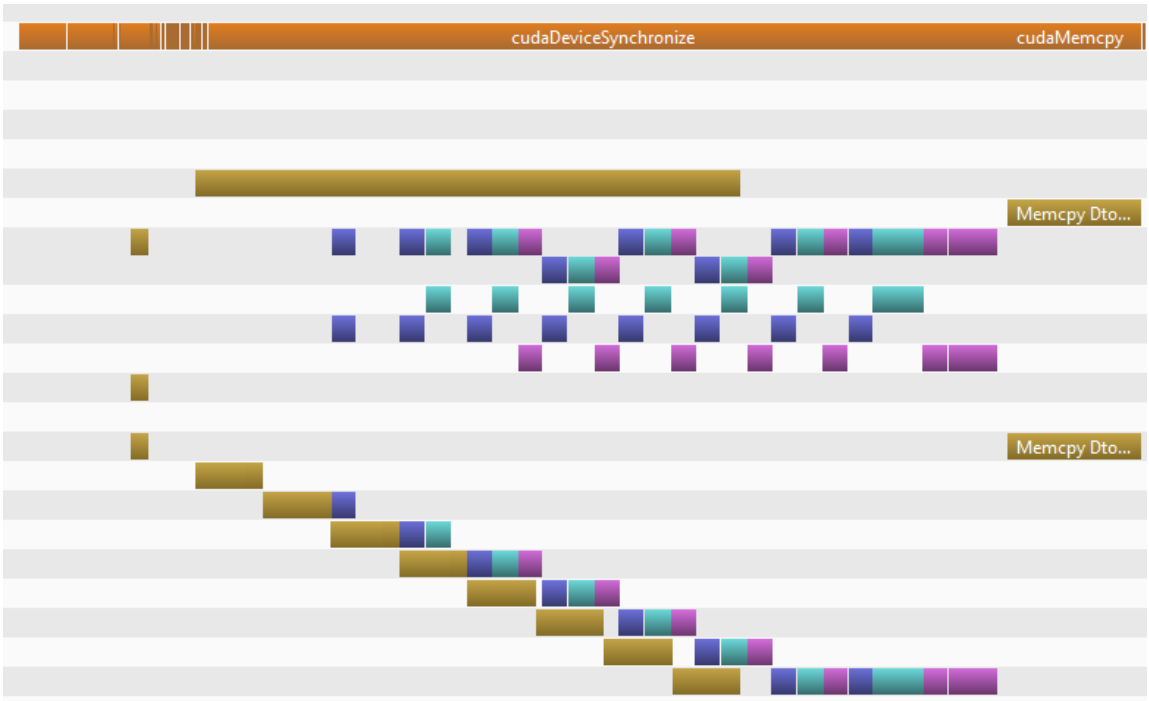

Figure 9-12: Multiple Kernel Overlapping with 8 slices

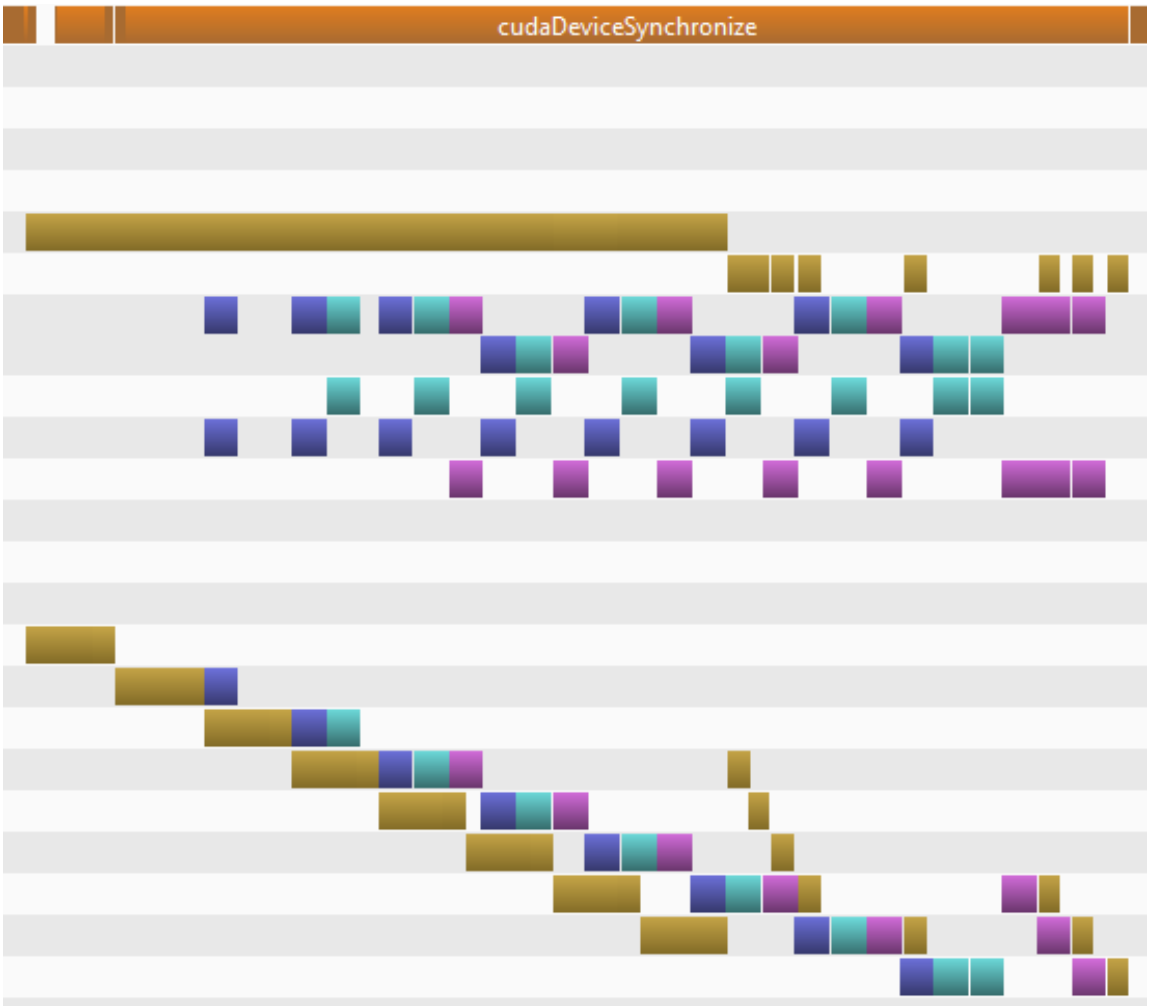

Figure 9-13: Full memory and kernel overlapping with 8 slices

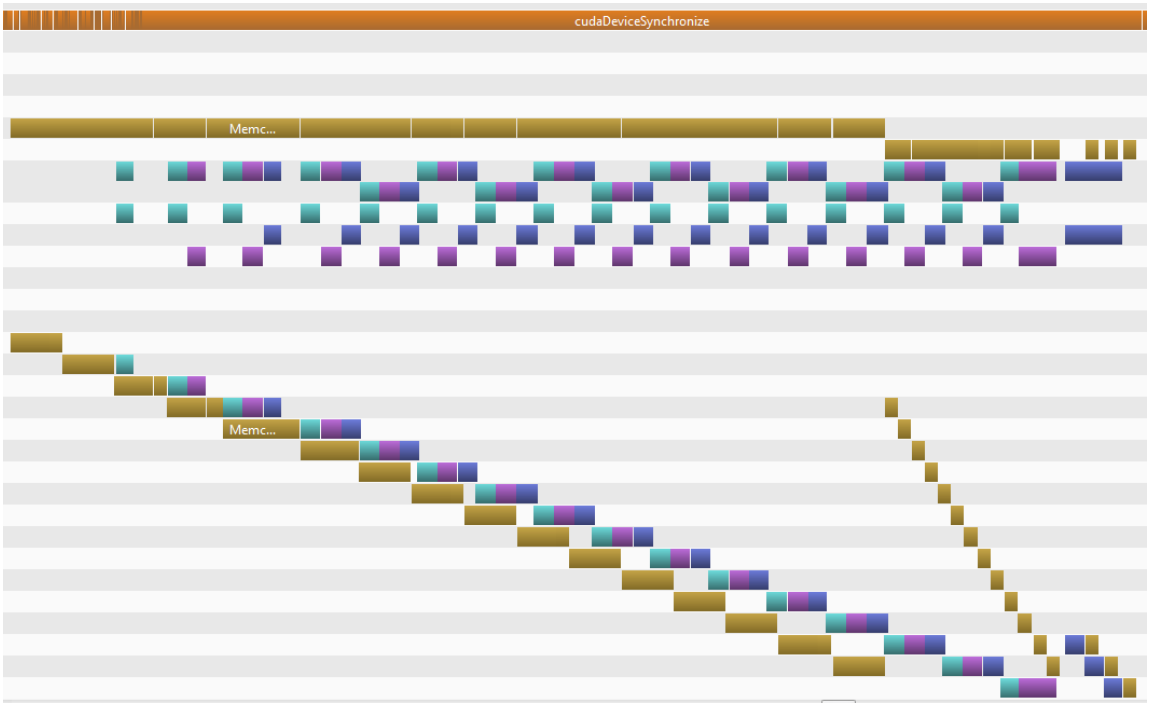

Figure 9-14: Full memory and kernel overlapping with 16 slices

```
1
2   void ExecuteCUDA() {
3
4     dim3 threads(TILE_X, TILE_Y, TILE_Z);
5     dim3 blocks(ELX / TILE_X, ELY / TILE_Y, ELZ / TILE_Z);
6
7     int chunk_size = num_bytes / BBZ;
8
9     ResultType     * itr_phi = pPhiA;
10    double         * d_itr_phi = d_PhiA;
11
12    Variable3DType * itr_vel = pVelocity;
13    double         * d_itr_vel = d_vel;
14
15    ResultType     * itr_phi_res = pPhiA;
16    double         * d_itr_phi_res = d_PhiA;
17
18    cudaMemcpyAsync(
19      d_itr_vel,
20      itr_vel,
21      chunk_size * sizeof(double) * 3,
22      cudaMemcpyHostToDevice,
23      dstream1[0]
24    );
25
26    cudaMemcpyAsync(
27      d_itr_phi,
28      itr_phi,
29      chunk_size * sizeof(double),
30      cudaMemcpyHostToDevice,
31      dstream1[0]
32    );
```

Figure 9-15: Final cuda code - Part 1 (initialization)

```
33
34      for (int c = 1; c < BBZ; c++) {
35
36        itr_phi   += chunk_size;
37        d_itr_phi += chunk_size;
38        itr_vel   += chunk_size;
39        d_itr_vel += (chunk_size * 3);
40
41        cudaMemcpyAsync(
42          d_itr_vel,
43          itr_vel,
44          chunk_size * sizeof(double) * 3,
45          cudaMemcpyHostToDevice,
46          dstream1[c]
47        );
48
49        cudaMemcpyAsync(
50          d_itr_phi,
51          itr_phi,
52          chunk_size * sizeof(double),
53          cudaMemcpyHostToDevice,
54          dstream1[c]
55        );
56
57        BackCUDA <<< threads, blocks, 0, dstream1[c] >>>(
58          d_PhiB, d_PhiA, d_vel,
59          rDx, rIdx, rDt,
60          rX + rBW,
61          0 * ELX, 0 * ELY, (c-1) * ELZ
62        );
63
64        if (c > 1) {
65          ForthCUDA<<< threads, blocks, 0, dstream1[c] >>>(
66            d_PhiC, d_PhiA, d_PhiB, d_vel,
67            rDx, rIdx, rDt,
68            rX + rBW,
69            0 * ELX, 0 * ELY, (c-2) * ELZ
70          );
71
72          if (c > 2) {
73            EccCUDA <<< threads, blocks, 0, dstream1[c] >>>(
74              d_PhiA, d_PhiC, d_vel,
75              rDx, rIdx, rDt,
76              rX + rBW,
77              0 * ELX, 0 * ELY, (c-3) * ELZ
78            );
79          }
80        }
81      }
```

Figure 9-16: Final cuda code - Part 2 (Main loop)

```
82      for (int c = 0; c < BBZ - 3; c++) {
83        cudaMemcpyAsync(
84          itr_phi_res,
85          d_itr_phi_res,
86          chunk_size * sizeof(double),
87          cudaMemcpyDeviceToHost,
88          dstream1[3 + c]
89        );
90
91        itr_phi_res += chunk_size;
92        d_itr_phi_res += chunk_size;
93      }
94
95      for (int c = BBZ - 1; c < BBZ; c++) {
96        BackCUDA <<< threads, blocks, 0, dstream1[(BBZ)] >>>(
97          d_PhiB, d_PhiA, d_vel,
98          rDx, rIdx, rDt,
99          rX + rBW,
100         0 * ELX, 0 * ELY, c * ELZ
101       );
102     }
103
104     for (int c = BBZ - 2; c < BBZ; c++) {
105       ForthCUDA<<< threads, blocks, 0, dstream1[(BBZ)] >>>(
106         d_PhiC, d_PhiA, d_PhiB, d_vel,
107         rDx, rIdx, rDt,
108         rX + rBW,
109         0 * ELX, 0 * ELY, c * ELZ
110       );
111     }
112
113     for (int c = 0; c < 3; c++)
114       cudaEventCreate(&trail_eec[c]);
115
116     for (int c = 1; c <= 3; c++) {
117       if (c > 1)
118         cudaStreamWaitEvent(dstream1[(BBZ - 3 + c - 2)], trail_eec[c - 2], 0);
119
120       EccCUDA<<< threads, blocks, 0, dstream1[(BBZ - 3 + c)] >>>(
121         d_PhiA, d_PhiC, d_vel,
122         rDx, rIdx, rDt,
123         rX + rBW,
124         0 * ELX, 0 * ELY, c * ELZ
125       );
126
127       cudaEventRecord(trail_eec[c - 1], dstream1[(BBZ - 3 + c)]);
128
129       cudaMemcpyAsync(
130         itr_phi_res,
131         d_itr_phi_res,
132         chunk_size * sizeof(double),
133         cudaMemcpyDeviceToHost, dstream1[(BBZ - 3 + c)]
134       );
135
136       itr_phi_res += chunk_size;
137       d_itr_phi_res += chunk_size;
138     }
139
140     cudaDeviceSynchronize();
141   }
```

Figure 9-17: Final cuda code - Part 3 (Trailing)

Chapter 10

Conclusions

The main goal of this dissertation was to provide an implementation of a semi-lagrangian fluid solver which increased precision without sacrificing performance. We also seek to provide an implementation of that solver such it can be efficiently parallelizable on HPC environments and specifically in isolated NUMA nodes
We proposed the use of a structured mesh for our problem in order to apply the BFECC algorithm for the advection step and analyzed the results. This has allowed us to improve the precision of the solution and at the same time increase the overall performance of the system.

We have tested different parallelism techniques and we have shown that, while there are not important changes in performance, different divisions of the problem adapt better to some parallelization schemes than others, concretely we have explored approaches using Omp and OmpSs and analyzed the scalability of the resulting code. Both approaches result in almost perfect scalability curves while the implementation with OmpSs has shown to be more efficient as the number of threads grows.

We also have introduced some algorithmic optimizations to the code, such the boundary elements, which help both with the efficiency of the code and with the future integration with the other parts of the solver.

Once finished, the problem look suitable to be ported into GPU and we tested some implementations of the solver using CUDA, which have shown us that, in case of existence, GPGPU is factor to take into account and can speed up the code or provide additional computational resources. Further investigating into GPU implementations we have seen that some of them could benefit from intensive use of local memory, but we have proven that with the actual transfer times between Host and Device, all the computation time in the GPU can be hidden in the memory transfer using buffering technics.

In the other side, the code still present some challenges like the interpolation scheme. We have seen that, with the actual ratio of instructions required to calculate a memory access versus the ones that are required to calculate the interpolation itself, is not worth to store the results of the later, but probably there is an intermediate solution that can exploit the benefits of both approaches.

Chapter 11

Future work

We have demonstrated that regions of our fluid simulation can effectively be computed using block and how this introduces benefits both in precision and speed in some regions of the simulation. Nevertheless, the final goal of the NUMEXAS project is to a have a full fluid simulation.
As the final goal of the NUMEXAS project is to have a full functional fluid simulator that can run on large systems, there is still some planned work in the way

Firstly a scheme to couple the different blocks of the simulation needs to be developed and tested. The design of such scheme of integration is partially developed, and we are in process of implementing it. This will allow, through the use of MPI, to communicate different cubic regions that will be host in different physical nodes.

Such scheme is the perfect candidate to overlap communications and computing, in a similar way as the GPU solution work, as only the regions in the boundary of the cube are restricted by serialization. Moreover, parameters like the interpolation of the position based on the velocity could be calculated before the results of the other cubes are integrated but experiments need to be run in order to check if it is worth to pre-calculate this results. Another factor that will be subject to change will be the implementation of the mass conservation equation from a explicit scheme to an implicit one.

Once, or before, this ideas are implemented, the stand alone test code is projected to be ported to KratosMultihpyscis framework. Along with the integration a real multiphysics framework the code will need to be adapted to work with standard mesh formats used in the industry like for instance HDF5, as it now uses its own simple input format

Another step of the project will be coupling the cubic regions with the non-cubic regions. This is not trivial anymore as different meshes, structured and unstructured need to be coupled, since the necessity to integrate the different parts of the codes into Kratos. Additionally, cubic region sizes were designed to be multiple of each other, but it does not happen the same with classic regions. So further interpolation will need to be applied and again precision of the results checked.

Finally, we have started the implementation of a small part of the code in GPU. This accelerators have proven to be convenient for the nature of the algorithms we are dealing with. We have focused our efforts into hiding the transfer time of the data, since at this moment the execution is bound to the transfer time of the data and we do not have GPU with higher bandwidths no further improvements have been done on that area, but additional optimizations can be done in the kernels executed in the device. Furthermore, GPU offers a range of characteristics yet to be exploited. Among them it is especially interesting to exploit the use of the texture memory to accelerate operations like the trilinear interpolation or gradients, which are usually build-in functions in the texture memory and hence perform much faster than in a typical CPU. This will be especially interesting as new memory models on the GPU with higher bandwidths are begning to appear, as a reduction in the computation time will be required to match the new transfer speeds.

Bibliography

[1] O. C. Zienkiewicz, R. L. Taylor, O. C. Zienkiewicz, and R. L. Taylor, *The finite element method*, vol. 3. McGraw-hill London, 1977.

[2] A. R. Mitchell and D. F. Griffiths, *The finite difference method in partial differential equations*. John Wiley, 1980.

[3] V. Girault and P.-A. Raviart, "Finite element approximation of the navier-stokes equations," *Lecture Notes in Mathematics, Berlin Springer Verlag*, vol. 749, 1979.

[4] B. M. Kim, Y. Liu, I. Llamas, and J. Rossignac, "Advections with significantly reduced dissipation and diffusion," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, no. 1, pp. 135–144, 2007.

[5] L. Dagum and R. Enon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[6] D. Yoder, "Trilinear Interpolation," 2003. [Online; accessed 12-March-2015].

[7] W. C. (Cmglee), "Trilinear interpolation visualisation," 2014. File: Trilinear_interpolation_visualisation.svg.

[8] T. F. Dupont and Y. Liu, "Back and forth error compensation and correction methods for removing errors induced by uneven gradients of the level set function," *Journal of Computational Physics*, vol. 190, no. 1, pp. 311–324, 2003.

[9] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel, "Automatic simd vectorization of fast fourier transforms for the larrabee and avx instruction sets," in *Proceedings of the international conference on Supercomputing*, pp. 265–274, ACM, 2011.

[10] V. K. Elangovan, R. M. Badia, and E. A. Parra, "Ompss-opencl programming model for heterogeneous systems," in *Languages and compilers for parallel computing*, pp. 96–111, Springer, 2013.

[11] R. Gayatri, *Increasing Parallelism through Speculation in a Task-based Programming Model*. PhD thesis, Universitat Politécnica de Catalunya, 2015.

[12] W. H. Wen-Mei, *GPU Computing Gems Emerald Edition*. Elsevier, 2011.