Universitat Politècnica de Catalunya

Escola Superior d'Enginyeries Industrial, Aeroespacial
i Audiovisual de Terrassa

Numerical simulation of Smith Hutton problem

Computational Engineering

220027

Professor

Carlos David Pérez Segarra

Francesc Xavier Trias Miquel

Author

Sergio Gutiérrez Sánchez

Saturday 22nd January, 2022

Abstract

In this report it is presented the transient numerical resolution of the Convection-Diffusion equation, more precisely, the Smith Hutton problem. The main studies carried out concern the mesh influence on the problem solution, the different suitable convective schemes for this case and the advantage of the implementation of parallel computing on the computational code. In addition to that, it is presented a brief description and explanation of the equations and numerical techniques used for the resolution of the proposed case.

1 Introduction

The Smith Hutton problem was firstly presented by R.M. Smith and A.G. Hutton in 1982. It was an attempt to solve a problem involving streamline curvature characteristic of recirculating flows and steep variation in a transported scalar physical variable [4].

This study case consists on a circular flow going through a cavity transporting an arbitrary physical property ϕ (figure 1).

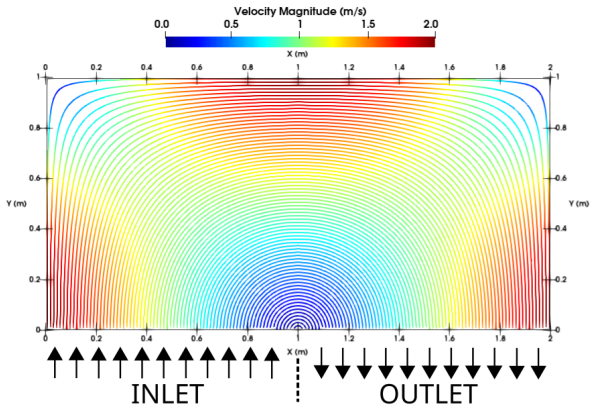

Figure 1: Smith Hutton problem velocity field.

For the reference system presented in previous figure, the velocity fields equations are the following.

$$u(x,y) = 2y(1-(x-1)^2) \tag{1}$$

$$v(x,y) = -2x(1-y^2) \tag{2}$$

As there can be seen in figure 1, the previously presented equations generate a velocity field in which on the left part of the domain, the flow enters to the cavity (and so the physical variable) and on the other part, it goes out.

This can be clearly seen in the boundary conditions of the problem.

On the two vertical walls and the upper one, a Dirichlet condition is applied. The value of the physical property transported is known by the following equation.

$$\phi = 1 - tanh(10) \approx 0.0 \tag{3}$$

And at the inlet.

$$\phi = 1 + tanh[10 \cdot (2 \cdot (x-1)+1)] \tag{4}$$

Finally, at the outlet, a Neumann condition is applied (known gradient).

$$\frac{\partial \phi}{\partial y} = 0 \tag{5}$$

Looking at the boundary conditions presented before, it can be clearly stated that Smith Hutton problem can be considered as an enclosed cavity with a physical property entering to the domain and going out through the outlet with no other contribution to it.

2 Description of numerical methods

In this section it is presented a brief explanation of the equations to solve and its numerical resolution.

In order to obtain a solution of the Smith Hutton problem, it is necessary to solve the Convection-Diffusion equation (eq. (6)).

$$\frac{\partial \rho\phi}{\partial t} + \nabla \cdot (\rho \mathbf{u}\phi) = \nabla \cdot (\Gamma_\phi \nabla\phi) + S_\phi \qquad (6)$$

This equation represents how the physical property ϕ is been transported through the domain. If it is due to the flow (Convective transport), or because of gradients (differences in the value between two different areas)(Diffusive transport).

In this problem, there are several assumptions that ease the numerical resolution of this equation.

First, in this case, there is no source term (S_ϕ), the inlet and outlet are already defined and are the only sources of the physical property. Secondly, the density of the flow (ρ) is considered to be constant, which allows to simplify some terms in the equation and reduces the computational cost. And finally, in the Convection-Diffusion equation, the variable Γ_ϕ is 1. Therefore the diffusive term $\nabla \cdot (\Gamma_\phi \nabla\phi)$ can be simplified to $\nabla^2\phi$.

2.1 Spatial Discretization

In order to solve the Smith Hutton problem it has been selected to use FVM (Finite Volume Method). Therefore, the discretization of the domain has been done using this same approach.

In this case there isn't large gradients near the walls, basically because there are no non-slip condition walls. The biggest gradients take place near the centre of the cavity and not only in the horizontal or vertical direction. Because of this, there has been chosen to develop an structured orthogonal regular mesh like the one presented in figure 2.

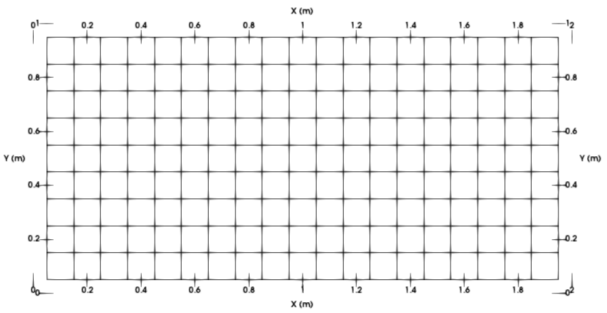

Figure 2: Regular mesh scheme

Since in this case there is no pressure involving the flow and the velocity field is initially fixed, there has been chosen to use a collocated mesh (there is no risk of checkerboard problems). Additionally to this, in order to take into account the boundary conditions, apart from the original mesh, there has been decided to add nodes on each of faces of the cavity.

2.2 Equation Discretization

With all the assumptions mentioned at the beginning of the section, the Convection-Diffusion equation has the following form.

$$\rho\frac{\partial \phi}{\partial t} + \rho\nabla \cdot (\mathbf{u}\phi) = \nabla^2\phi \qquad (7)$$

As there can be seen, this equation can be splited into 3 different terms.

Transient term

Since the approach of the study case has been decided to be transient, it is necessary to develop a time integration method. In order to do that,there has been decided to apply an explicit scheme in which CFL (Courant-Friedrich-Lewy) condition is used to determine the time step Δt.

Discretizing the transient term of equation (7) leads to the following result.

$$\rho\frac{\partial \phi}{\partial t} \longrightarrow \rho \cdot \frac{\phi^{n+1} - \phi^n}{\Delta t} \qquad (8)$$

For the CFL condition, the following expression have been used.

$$\Delta t_c = min\left(0.10\frac{\Delta x}{|u|}\ ,\ 0.35\frac{\Delta y}{|v|}\right) \quad (9)$$

$$\Delta t_d = min\left(0.10\frac{\rho\Delta x^2}{1}\ ,\ 0.10\frac{\rho\Delta y^2}{1}\right) \quad (10)$$

Convective term

In the discretization of the convective term of the equation there must be remarked the use of convective schemes.

Applying the FVM basis to this term of the equation leads to the following form.

$$C_P = \frac{\rho}{V_P}\cdot(u_e\phi_e - u_w\phi_w + u_s\phi_s - u_n\phi_n) \quad (11)$$

As there can be seen in previous equation, in order to compute the convective term, it is necessary to calculate the velocities and property values right at the wall of the control volumes.

In case of the velocities, it is pretty easy. It is just needed to substitute the coordinate values of the centre of these walls into the velocity field equations presented in section 1 (eq. (3) and (4)).

However, in case of the property values on the walls there are two different possibilities.

First, and the easiest one is to compute these as the mean value of the two nodes the wall is splitting them into. For instance, in case of the west wall,taking nodes ϕ_W and ϕ_P and calculating the mean. This is clearly the easiest approach, but the one with the highest limitations. First, this is only mathematically correct for regular meshes, and secondly, it doesn't take into account the surrounding conditions of the physical variable, such as the direction of the flow.

The second one, which has been chosen to develop in this case, is the application of convective schemes.

Convective schemes are a mathematical way to compute the value of any property or variable at a certain point but taking into account the surrounding flow conditions.

For instance, when computing the value ϕ_w, it is obviously necessary to know ϕ_W and ϕ_P. But it is also very important to know if the flow at that point of the wall is heading towards node W or P. Moreover, in order to have even better information about how this value is evolving through space, nodes WW and E values can also be provided and taken into account.

There are many different convective schemes depending of their characteristics and order. In this case, only first and second order schemes are used.

This order of the schemes is related to the error of the solution. When using a first order convective scheme, doubling the number of nodes reduces the total error by a factor of 2. However, in case of second order schemes, this factor increases up to 4. This means that, having a mesh with triple number of nodes reduces the error by a factor of 9.

This must be taken into account when applying one or other schemes. First order schemes are easier to program and their computational cost is smaller. However, more total nodes are needed in order to get the same error. This should be taken into account when selecting a higher or lower order schemes.

For this study, CDS (Central Difference Scheme), UDS (Upwind Difference Scheme), SUDS (Second Order Upwind Difference Scheme), QUICK (Quadratic Upwind Interpolation for Convective Kinematics Scheme) and SMART (Sharp and Monotonic Algorithm for Realistic Transport [1]) have been implemented [2].

Diffusive term

Last term of the equation is the Diffusive one. As in the two other cases, FVM basis is applied to its numerical discretization, which leads to

the following result.

$$D_w = \frac{1}{V_P} \cdot \frac{S_w}{d_{PW}} \cdot [\phi_P - \phi_W] \quad (12)$$

$$D_e = \frac{1}{V_P} \cdot \frac{S_e}{d_{PE}} \cdot [\phi_E - \phi_P] \quad (13)$$

$$D_s = \frac{1}{V_P} \cdot \frac{S_s}{d_{PS}} \cdot [\phi_P - \phi_S] \quad (14)$$

$$D_n = \frac{1}{V_P} \cdot \frac{S_n}{d_{PN}} \cdot [\phi_N - \phi_P] \quad (15)$$

Combining all the terms discretized previously leads to the following global equation.

$$\rho \frac{\phi_P^{n+1} - \phi_P^n}{\Delta t} + C_P = D_w + D_e + D_s + D_n \quad (16)$$

However, in order to solve the equation, it is necessary to rearrange this expression into the following one.

$$a_P \phi_P^{n+1} = a_W \phi_W^n + a_E \phi_E^n + a_S \phi_S^n + a_N \phi_N^n + b_P^n \quad (17)$$

Where:

- $a_W = \frac{1}{V_P} \cdot \frac{S_w}{d_{PW}}$
- $a_E = \frac{1}{V_P} \cdot \frac{S_e}{d_{PE}}$
- $a_S = \frac{1}{V_P} \cdot \frac{S_s}{d_{PS}}$
- $a_N = \frac{1}{V_P} \cdot \frac{S_n}{d_{PN}}$
- $a_P = a_W + a_E + a_S + a_N + \frac{\rho}{\Delta t}$
- $b_P^n = \frac{\rho \phi_P^n}{\Delta t} + C_P$

2.3 Resolution algorithm

In order to solve the physical property ϕ field in the domain, it is necessary to solve each step equation (17).

One of the main features of the programming code developed for this problem case is the parallel computing (which will be explained later in a more detailed way). This implementation of paralelisation doesn't allow to use algebraic solver such as TDMA (Tri Diagonal Matrix Algorithm) or LU (Lower Upper Factorization). Therefore, a simple but effective Gauss-Seidel solver has been programmed.

$$\phi_P^{n+1} = \frac{a_W \phi_W^n + a_E \phi_E^n + a_S \phi_S^n + a_N \phi_N^n + b_P^n}{a_P} \quad (18)$$

This is an iterative algorithm which allows to solve the discretized equation presented before (18). To do that,it is necessary to do multiple sweeps trough all nodes of the domain until a solution convergence is reached.

$$|\phi_{Computed} - \phi_{Supposed}| < \delta_{GS}? \quad (19)$$

If condition 19 is met, Gauss - Seidel algorithm stops and then begins a new time step (n+1). For this study, the convergence value for the Gauss - Seidel iterative process has been set to $1 \cdot 10^{-7}$ in order to get a great accuracy on the results.

3 Code structure

In this section are presented the main features of the computational codes developed for the resolution of the Smith Hutton problem.

The programming language selected is C++. It is one of the most commonly used languages for Computational Fluid Dynamics. Another important feature of the code is that is has been programmed following an object-oriented approach. Each important process or feature of the CFD simulation has its own C++ programming Class (see all the codes in annex A). For instance, all the processes related to the creation of the mesh and its geometrical entities have its own class (Mesher), and so for the solver, the dynamic memory allocation or the data input.

However, the most significant feature of the code is the parallelisation. CFD codes have to carry out millions or even billions of computational operations during a simulation. And, although nowadays the power of computers per core are huge in comparison with years ago, it isn't still enough to make parallelisation worthless in terms of time spent.

Therefore, MPI (Message Passing Interface) parallelisation processes have been implemented to the code (see both Header and Cpp codes in annex A). The theoretical basis of this parallelisation is relatively easy. The mesh has a certain number of nodes, in which many different calculations must be done. Therefore, why not splitting this workload between several cores of the computer.

When solving Gauss-Seidel, each core of the CPU have only to solve its part of the total mesh. The iterative sweep is only done on these division (figure 3).

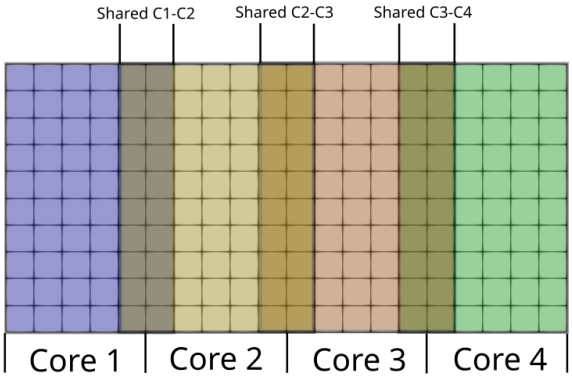

Figure 3: Mesh division with parallelisation

However, as there can be seen in equation (18), when approaching to the limit of core's domain, it is necessary to know the value of the next node, which is already outside of the initial division. Because of this, it is necessary to define what's known as *Halo*. This Halo is a number of rows or columns (columns in case of figure 3, whose values must be sent to the core next to it on every sweep of Gauss-Seidel process and on every step of the simulation. This is called communication between processes.

In figure 3 there can be seen shared areas of the domain between two adjacent cores or processes.

Although it requires a certain knowledge in parallel programming and it increases the complexity of the code, later in this report will be seen that it is completely worth to pay this price in favour of reduction in computational time.

Once all main features of the code have been mentioned, it is presented next up a summary of the simulation algorithm.

1. Read input data from text files.
2. Declaration of all parallel programming and communication functions.
3. Mesh creation.
4. Initialization of the variables.
5. Transient simulation until steady state.
    - Computation of discretization coefficients.
    - Resolution of step (n+1) with Gauss-Seidel.
    - Repeat until desired convergence ($1 \cdot 10^{-8}$).
6. Final calculations.
7. Visualization of the results.

4 Code Verification

In order to verify the developed computational code there has been selected to compare the obtained results for the highest (ρ/Γ) simulation carried out (10^6).

In an ideal Convection-Diffusion case, with an infinite (ρ/Γ) ratio, the exact solution of the outlet is not other than a symmetric inlet. Therefore, a great option to compare the obtained results is to see how the solution for the highest ratio simulated approaches to this.

This is presented in figure 4.

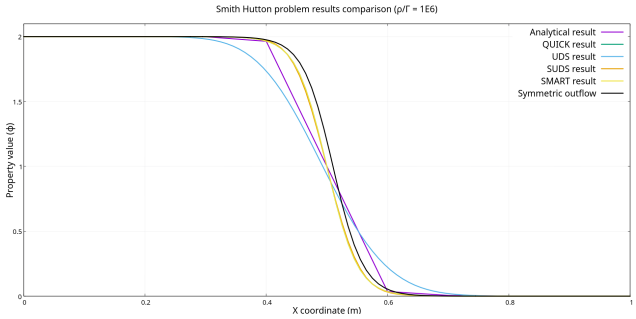

Figure 4: Outlet results for $(\rho/\Gamma) = 10^6$.

All the simulations present in previous figure have been carried out with a 200×100 mesh.

As there can be seen in the comparison, the obtained results match in a great way with a symmetric outflow (black plot) and with the ones extracted from [3] (purple plot). In case of the firsts, there must be mentioned that, although 10^6 is a high ratio, a symmetric outflow can't be only perfectly achieved since it requires an infinite ratio. However, it can be clearly observed the similarity and the same pattern between both, even with first order schemes.

5 Results and discussion

In this section are presented the results and studies carried out for this case, such as a mesh influence analysis, convective schemes comparison, Gauss-Seidel relaxation factor or even a paralelisation time comparison. Apart from the presentation of the results of the Smith Hutton case.

5.1 Mesh influence study

A high enough dense mesh is vital for the resolution of any CFD problem case. Setting a total number of nodes or another can decide if the simulation results can be considered as acceptable or not. However, computational power isn't unlimited and free. Selecting an absurdly high number of nodes may give a correct result, but it also may take more time or resources than available. Therefore, it is important to determine a balance between results accuracy and computational costs.

In this case, in order to do that, there has been selected to do the following.

The obtained results at the outlet of the cavity are compared with the ones extracted from [**pdf˙ejercicios**]. This comparison is made by the calculation of the relative error between both. Then, for these 11 errors, a mean value of them is computed for each mesh density selected.

These results are presented in figure 5.

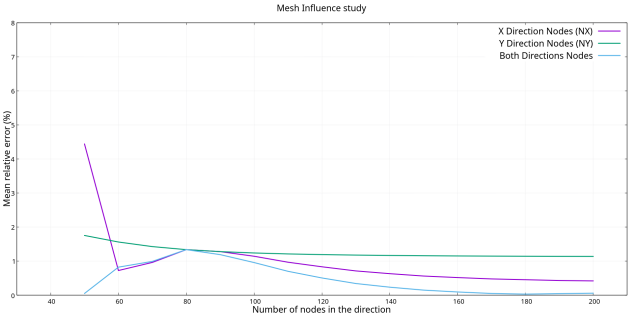

Figure 5: Mesh influence study results

Since the total number of nodes depend on the nodal density on direction X and Y, 3 different weeps have been made. First of them (*X direction Nodes (NX)* in figure 5) consists on setting the number of nodes of Y direction fixed to 80 and doing a sweep on the number of nodes in X direction to see the influence these have on the mean relative error.

The same basis is applied to the Y direction (*Y Direction Nodes (NY)*), setting X direction nodes to 80 again.

And lastly, *Both Direction Nodes*, consists on increasing the number of nodes in each direction going from 50×50, 60×60, etc mesh.

As it was expected, the plots clearly shows that between direction X and Y, the one that has bigger influence on the relative error is X. However, looking at *Both Direction Nodes* plot,

there can be observed that nodal density in Y axis has influence too.

Taking into account all of this, there has been selected as standard mesh for the simulations a 200×100. It clearly gives a more than acceptable results like the ones presented for the verification of the code (section 4, figure 4) without representing a useless waste in computational time.

5.2 Convective schemes comparison.

As it has been exposed in section 2.2, convective schemes have a great impact on the results obtained in the simulations. It is necessary to find a balance between computational cost because of the convective scheme and the reduction on the mesh density it has.

As mentioned before, the convective schemes implemented in the codes are CDS, UDS, SUDS, QUICK and SMART.

The results obtained for each of them can be seen in figure 4, in verification section.

As it was expected, second order schemes (SUDS, QUICK and SMART) have a much better performance when setting the same amount of nodes in the mesh. The plots approach much more to the symmetric outflow results and to the numerical results extracted from [3].

As it can be also observed, there is no difference in the obtained outflow between any of the second order schemes implemented. All of them give the same results. Therefore, the parameter to decide which is used in the simulations can't be the accuracy, but the computational time.

In C++ programming language, it is possible to measure the time spent between two points of the code thanks to *Chrono* library. For these convective schemes, the obtained computational times are the ones presented in table 1.

Table 1: Computational times for convective schemes

Convective Scheme	**Computational Time [s]**
SMART	75.17
SUDS	59.41
QUICK	96.76

As there can be seen, SUDS performance is much faster than the other two. It is the simplest one among the second order schemes presented and the one that requires less computational power.

However, as there has been observed, it has the same accuracy for this problem case and therefore it is the one selected for the simulations.

5.3 Relaxation factor

Another important parameter of the simulation if the relaxation factor FR. This is related to Gauss-Seidel iterative process in the following way.

$$\phi_{Calc.} = \phi_{Sup.} + FR \cdot (\phi_{Calc.} - \phi_{Sup.}) \quad (20)$$

Setting the right value to FR (between 0 and 2) results in a decrease of the total number of iterations Gauss-Seidel process has to make before reaching convergence, and so it cuts down the computational time.

In order to find this exact value, it is necessary to do a sweep. In this case, it has been selected to do this sweep between 0.40 and 1.75. Initially, this values were much closer to 0.0 and 2.0 respectively, however, the computational time in these cases is much larger, and doesn't provide any advantage.

For the selected range, the results obtained are presented in figure 6.

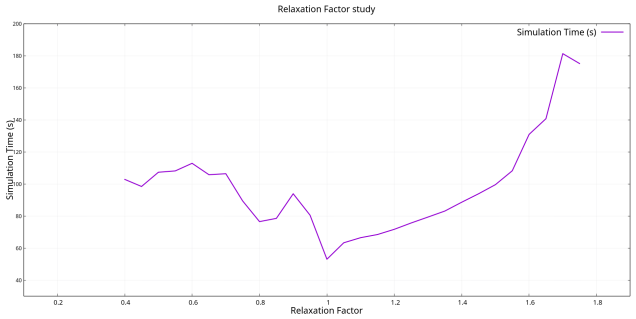

Figure 6: Relaxation factor results for sweep 1

In the previous figure, there can be clearly observed that there is a great difference in computational time depending on the relaxation factor value. In this case, the sweep has been done using a resolution of 0.05. However, in order to get a better resolution of the optimal value, a second sweep has been performed from 0.95 to 1.04 with increments of 0.01.

The results are presented in the following figure (figure 7).

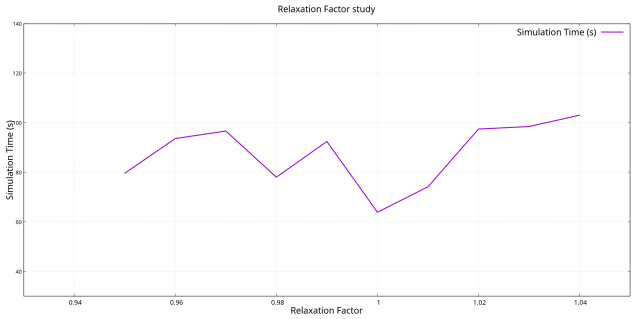

Figure 7: Relaxation factor results for sweep 1

As there can be seen in the previous figure, the optimal value for the relaxation factor is 1. This is basically not having any relaxation factor, since for this value, the Gauss-Seidel iterative equation (equation (20)) remains as the original one presented in section 2.2.

5.4 Paralelisation performance

As it has been mentioned before in section 3, one of the main features of the codes is the parallelisation implemented. It also has been mentioned before that, although it increases considerably the programming difficulty, it is worth in many cases.

In order to quantify the usefulness of this choice, there has been performed the same simulation, with the same parameters, but with a different number of cores. The results of the computational time obtained are presented in table 2.

Table 2: Computational times per number of cores

Number of cores	**Computational Time [s]**
2	95.34
3	72.05
4	57.83

In this case, it hasn't been possible to perform a simulation with just 1 core (sequential). The codes are designed for parallel programming and they would require mayor modifications. In addition to that, there has been only possible to test the computational time with up to 4 cores.

In what comes to the results obtained, as it was obviously expected, the higher the number of cores, the lower the computational time. However, as it would look like, this relationship isn't linear. The higher the number, the worse performance obtained per core. The communication between them takes time which must be added to the calculation one and so on. As there can be seen in table 2, doubling the number of cores doesn't imply having half of the simulation time. However, it must be admitted that this reduction is considerable.

The parallelisation of the codes, although more difficult than sequential programming, has allowed to perform long simulation studies, such as mesh influence or relaxation factor ones in a very short time. And the same with the other simulations carried out for this report.

5.5 Smith Hutton results

Once all the simulation variables and parameters have been completely defined, such as mesh, convective schemes or relaxation factor number, it is possible to carry out the final Smith Hutton simulations and present the results.

But first, it is necessary to mention the main parameter that defines the behaviour of the simulated flow. This is the ratio between density ρ and Γ_ϕ (equation 6) (ρ/Γ_ϕ).

This parameter determines how convective or how diffusive the flow is. In other words, if the ratio is relatively low, the flow is diffusive. This means that physical properties are more likely to spread through the domain thanks to gradients rather than the flow motion. However, if this ratio is high, the flow is said to be convective. These physical properties would mainly move because of the fluid motion, not because gradients on their values.

In table 3 are presented the parameters of the simulations carried out.

Table 3: Simulation parameters of the Smith Hutton problem

(ρ/Γ_ϕ) **ratio**	**Total number of CV**	**Convective Scheme**	**Convergence criteria**	**Mesh type**
10	200 ×100	SUDS	10^{-8}	Regular
10^3	200 ×100	SUDS	10^{-8}	Regular
10^6	200 ×100	SUDS	10^{-8}	Regular

As there can be seen, all the simulations carried out have been done using a more than enough 200×100 mesh. And, in addition to this, the value of the Courant number in the CFL for the step time (section 2.2) has been set to 0.10 or even lower, to 0.05.

The convective scheme used in every simulation is SUDS. As it has been shown, it provides the same accuracy as the other second order schemes implemented but in a much lower computational time. Another reason of this accuracy is the selected convergence, 10^{-8}. This can be considered as excessive and unnecessary. However, for the amount of total nodes selected and the number of cores used in the simulations, this convergence doesn't seem as something unreachable in terms of computational time, and it also ensures a great accuracy in the results.

Finally, as it has been said, not other type of mesh rather than regular has been implemented. In this problem there are no non-slip walls that may generate huge gradients. Having enough nodes in a regular mesh is enough to get a good accuracy on the results.

Next up, in figure 8 are presented the results obtained on the outflow for each simulation.

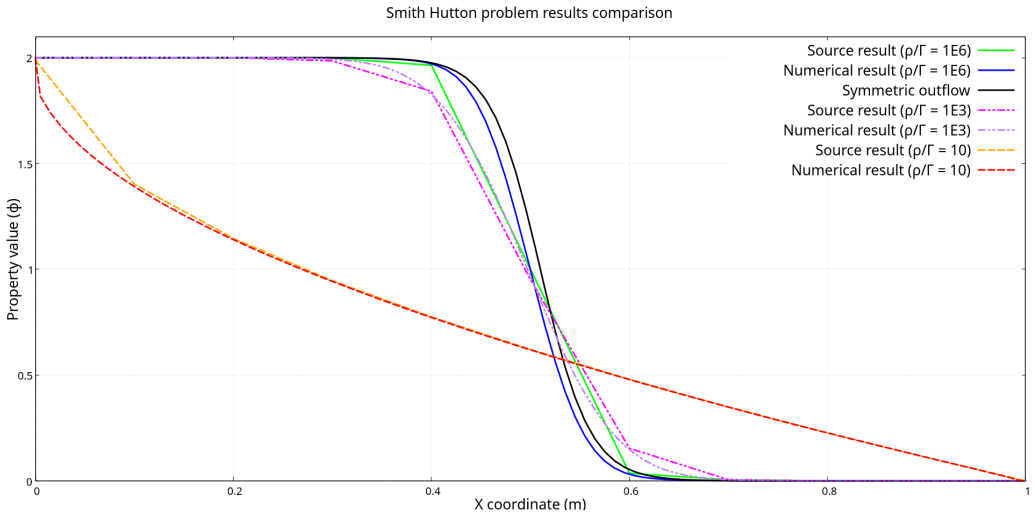

Figure 8: Smith Hutton outflow results comparison

There are several important aspects to mention about previous figure.

First of them concerns to the accuracy of the results. The source [3] only gives a few points for comparing (Source result plots). However, it is clear that the obtained numerical results match perfectly these.

Secondly, it is important to explain the results obtained in what comes to the convection and diffusion phenomena. In order to do that, there will also be presented the physical property ϕ fields of the steady converged simulations (figures 9 and 10).

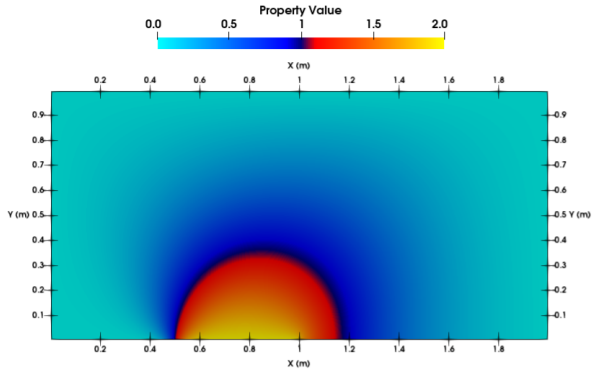

Figure 9: Smith Hutton ϕ field result for $(\rho/\Gamma_\phi) = 10$

In this first case of $(\rho/\Gamma_\phi) = 10$ (figure 9) there can be seen a clear dominance of the diffusive phenomena.

At the inlet, the value of ϕ is set to the boundary conditions established (section 1) near to 2.0. However, this only spreads out through the domain thanks to the gradient created by the inlet value, not because of the flow motion. Therefore, it spreads out in the cavity not following exactly the velocity field, but in every direction.

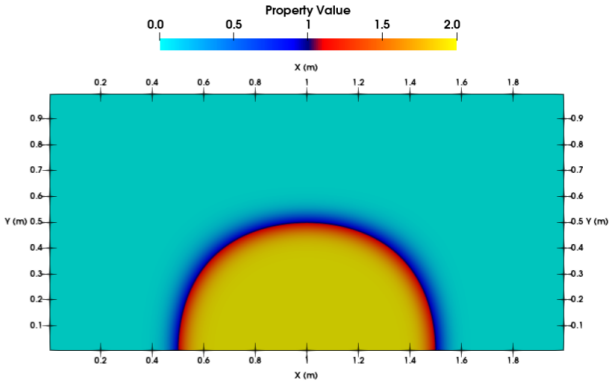

Figure 10: Smith Hutton ϕ field result for $(\rho/\Gamma_\phi) = 10^6$

On the other hand, in case of $(\rho/\Gamma_\phi) = 10^6$, there can be seen a great dominance of the convective term. As mentioned before, the inlet sets the value of ϕ to 2.0 approximately.

However, in this case, this value is both spread out through the domain due to gradients and transported by the flow motion. Nevertheless, there can't be denied the much bigger influence of latter. As it has been shown previously, for $(\rho/\Gamma_\phi) = 10^6$, the outlet ϕ value is close to be symmetric with the inlet (figure 8). This shows that this value reaches that point mainly because of the symmetric flow field imposed by the problem, and so thanks to the great dominance of the convective phenomena.

6 Conclusions

The resolution of the Smith Hutton problem, and therefore, the Convection - Diffusion equation is a great step towards more advanced CFD problems, which may include the resolution of Navier - Stokes equations.

Depending on the geometry case, it would be possible to recycle many parts of the already developed code for the present report. This could be the case of the mesher, the implemented functions of the convective schemes, the iterative solver, etc...

However, probably the most important aspect of the resolution of the Convective - Diffusion equation may be to understand the physical phenomena. It is important to acquire the knowledge of how to simulate or how a CFD simulation is done. However, learning and understanding the physical explanation behind the convection and diffusion phenomena is also crucial to carry out any further and more advanced study on the field.

References

[1] *Approximation Schemes for convective term - structured grids - Common – CFD-Wiki, the free CFD reference*. URL: https://www.cfd-online.com/Wiki/Approximation_Schemes_for_convective_term_-_structured_grids_-_Common.

[2] CTTC - Universitat Politècnica de Catalunya. "Numerical resolution of the generic convection-diffusion equation". In: (), pp. 1–28.

[3] CTTC - Universitat Politècnica de Catalunya. "Verification strategies for the convection-diffusion equation". In: (2014), pp. 1–5.

[4] B. P. Leonard and Simin Mokhtari. "Ultra-sharp solution of the smith-hutton problem". In: *International Journal of Numerical Methods for Heat & Fluid Flow* 2.5 (1992), pp. 407–427. ISSN: 09615539. DOI: 10.1108/eb017502.

A Programming Codes used

Main code

```
#include <fstream>
#include <cstdlib>
#include <iostream>
#include <cmath>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <chrono>
#include <mpi.h>

#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/
Codes/HeaderCodes/Memory.h"
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/
Codes/HeaderCodes/ReadData.h"
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/
Codes/HeaderCodes/ParPro.h"
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/
Codes/HeaderCodes/Mesher.h"
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/
Codes/HeaderCodes/Solver.h"

using namespace std;

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/
ConvectionDiffusion/"

int main(int argc, char* argv[]){

MPI_Init(&argc, &argv);
cout<<endl;

Memory M1;

ReadData R1(M1);
R1.ReadInputs();

ParPro MPI1(R1);
MPI1.Execute();

Mesher MESH(M1, R1, MPI1);
MESH.ExecuteMesher(M1, MPI1);

printf("Proceso %d, Initial X: %d, Final X: %d \n", MESH.Rank, MESH.Ix, MESH.Fx);

Solver S1(M1, R1, MPI1, MESH);
S1.ExecuteSolver(M1, R1, MPI1, MESH);


MPI_Finalize();
return 0;

}
```

Makefile code

```
#-------------------------------------------------------------------------------
# compile the UMFPACK demos (for GNU make and original make)
#-------------------------------------------------------------------------------

# UMFPACK Version 4.4, Copyright (c) 2005 by Timothy A. Davis.
# All Rights Reserved.  See ../Doc/License for License.

CODE = Main
DIRECTORIO = /home/sergiogus/Desktop/ComputationalEngineering/
ConvectionDiffusion/Codes/
default: $(CODE)


C = mpic++

#-------------------------------------------------------------------------------
# Create the demo programs, run them, and compare the output
#-------------------------------------------------------------------------------

$(CODE): $(CODE).cpp $(INC)
        $(C) -o $(CODE) $(CODE).cpp $(DIRECTORIO)/CppCodes/Memory.cpp
        $(DIRECTORIO)/CppCodes/ReadData.cpp $(DIRECTORIO)/CppCodes/ParPro.cpp
        $(DIRECTORIO)/CppCodes/Mesher.cpp $(DIRECTORIO)/CppCodes/Solver.cpp
#-------------------------------------------------------------------------------
# Remove all but the files in the original distribution
#-------------------------------------------------------------------------------

clean:
        - $(RM) $(CLEAN) $(CODE)
```

Memory Class Header Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <mpi.h>

using namespace std;

#define PI 3.141592653589793

class Memory{

        public:
                //Constructor de la clase
                Memory();

                //Metodos de la clase

                //M todos de alojamiento de memoria
                int *AllocateInt(int, int, int);
                double *AllocateDouble(int, int, int);
```

```
};
```

Memory Class Cpp Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <mpi.h>

#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/
Codes/HeaderCodes/Memory.h"
using namespace std;

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/
ConvectionDiffusion/"

Memory::Memory(){
        
}

//Memoria din mica matriz (double)
double *Memory::AllocateDouble(int NX, int NY, int Dim){
double *M1;

        M1 = new double [NX*NY*Dim];                                    
        return M1;
}

//Memoria din mica matriz (int)
int *Memory::AllocateInt(int NX, int NY, int Dim){
int *M1;

        M1 = new int [NX*NY*Dim];                                       
        return M1;
}
```

Read Data Class Header Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <mpi.h>

using namespace std;

class ReadData{
        private:


        public:

                string Esquema; //Problema Tobera/Tuberia
```

```
                double *GeometryData; //Datos de la geometr a del problema
                int *NumericalData; //Datos num ricos del problema
                double *ProblemData; //Datos del problema
                double *ProblemPhysicalData; //Datos fisicos sobre las
                condiciones del problema

                int *MeshStudyNX;
                int *MeshStudyNY;
                double *FactorRelax;

                //Constructor de la clase
                ReadData(Memory, int);

                void ReadInputs(); //Lector datos en ficheros
                void ReadArrays(string, int , double*);
                void ReadMeshStudy(string, int);

};
```

Read Data Class Cpp Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <fstream>
#include <sstream>
#include <bits/stdc++.h>
#include <string>
#include <mpi.h>

#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/
Codes/HeaderCodes/Memory.h"
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/
Codes/HeaderCodes/ReadData.h"

using namespace std;

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/
ConvectionDiffusion/InputData/"

//Constructor del lector de datos
ReadData::ReadData(Memory M1){

        GeometryData = M1.AllocateDouble(3, 1, 1); //Datos de la geometr a del problema
        NumericalData = M1.AllocateInt(5, 1, 1); //Datos num ricos del problema
        ProblemData = M1.AllocateDouble(2, 1, 1); //Datos del problema
        ProblemPhysicalData = M1.AllocateDouble(7, 1, 1); //Datos f sicos sobre las condi

}

void ReadData::ReadArrays(string FileName, int TotalData, double *Array){
int i = 0;
stringstream InitialName;
```

```
string FinalName;

        InitialName<<DIRECTORIO<<FileName;
        FinalName=InitialName.str();

        ifstream Data(FinalName.c_str());

                if (Data){
                        string line;
                        while (getline(Data, line)){
                                istringstream iss(line);
                                        if(i < TotalData){
                                                if (iss >> Array[i]){ i++; }    
                                        }
                        }
                }
                
        Data.close();

}

void ReadData::ReadInputs(){

        //Lectura datos en Arrays
        ReadArrays("GeometryData.txt", 3, GeometryData); //Input Datos Geometr a del probl
        ReadArrays("PhysicalData.txt", 7, ProblemPhysicalData);   //Input Datos f sicos d
        
int i = 0;
string FileName;
stringstream InitialName1;
string FinalName1;

        FileName = "ProblemData.txt";

        InitialName1<<DIRECTORIO<<FileName;
        FinalName1=InitialName1.str();
        
        ifstream DatosProblema(FinalName1.c_str());

                if (DatosProblema){
                        string line;
                        while (getline(DatosProblema, line)){
                                istringstream iss(line);
                                        if(i < 4){
                                                if (iss >> NumericalData[i]){ i++; }    
                                        }
                                        else if(i >= 4 && i < 6){
                                                if (iss >> ProblemData[i-4]){ i++; }    
                                        }
                                        else if(i == 6){
                                                if (iss >> NumericalData[4]){ i++; }
                                        }
                                        else{
                                                if (iss >> Esquema){ i++; }     
                                        }
                                        
```

```
                    }
                }
                DatosProblema.close();

}

void ReadData::ReadMeshStudy(string FileName, int TotalData){
int a, b;

int i = 0;
stringstream InitialName;
string FinalName;

        InitialName<<DIRECTORIO<<FileName;
        FinalName=InitialName.str();

        ifstream Data(FinalName.c_str());

        if (Data){
                string line;
                while (getline(Data, line)){
                        istringstream iss(line);
                        if ( iss >> MeshStudyNX[i] >> MeshStudyNY[i
                }
        }
        Data.close();

}
```

Parallel Porgramming Class Header Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <mpi.h>

using namespace std;

class ParPro{
        private:


        public:

                //Datos generales del problema
                int NX;
                int NY;

                //Datos y variables de computaci n paralela
                int Rank;
                int Procesos;
```

```
                int Ix;
                int Fx;
                int Halo;
                int pix, pfx;

                //Constructor de la clase
                ParPro(ReadData, int);

                //Metodos de la clase
                void Rango();
                void Processes();
                void Initial_WorkSplit(int, int&, int&);
                void Get_Worksplit(int, int, int, int&, int&);

                void SendData(double*, int, int);
                void ReceiveData(double*, int, int);
                void SendMatrixToZero(double*, double*, int, int, int, int, int);
                void SendDataToZero(double, double*);
                void SendDataToAll(double, double&);

                void Execute();

                int Get_Rank(){ return Rank; }
                int Get_Processes(){ return Procesos; }
                int Get_Halo(){ return Halo; }
                int Get_Ix(){ return Ix; }
                int Get_Fx(){ return Fx; }

};
```

Parallel Programming Class Cpp Code

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <math.h>
#include <stdlib.h>
#include <cmath>
#include <stdio.h>
#include <cassert>
#include <time.h>
#include <mpi.h>

#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade

using namespace std;

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/"

#define G(i,j,dim) (((j) + (i)*NY) + NX*NY*(dim)) //Global Index
#define GU(i,j,dim) (((j) + (i)*NY) + (NX+1)*NY*(dim)) //Global Index Matriz U
#define GR(i,j,dim) (((j) + (i)*(NY+1)) + NX*(NY+1)*(dim)) //Global Index Matriz R

#define MU(i,j,dim) ((j) + NY*(dim))
```

```
#define MR(i,j,dim) ((i) + (Fx - Ix)*(dim) + (Fx - Ix)*(4)*(j))

#define VR(i,j,dim) ((i) + (Fx - Ix)*(j))

#define LNH(i,j,dim) (((j) + ((i) - Ix + Halo)*NY)) //Local Index No Halo
#define LSH(i,j,dim) (((j) + ((i) - Ix)*NY) + NY*(Fx-Ix + 2*Halo)*dim) //Local Index Si Ha


ParPro::ParPro(ReadData R1){

                NX = R1.NumericalData[0];;
                NY = R1.NumericalData[1];;
                Halo = 2;

}

void ParPro::Rango(){
        int a;
        a = MPI_Comm_rank(MPI_COMM_WORLD, &Rank);
}

void ParPro::Processes(){
        int a;
        a = MPI_Comm_size(MPI_COMM_WORLD, &Procesos);
}

void ParPro::Initial_WorkSplit(int NX, int &Ix, int &Fx){
int Intervalo, Residuo;
int p;
        Intervalo = NX/Procesos;
        Residuo = NX%Procesos;

        if(Rank != Procesos-1){
                Ix = Rank*Intervalo;
                Fx = (Rank+1)*Intervalo;
        }
        else{
                Ix = Rank*Intervalo;
                Fx = (Rank+1)*Intervalo + Residuo;
        }
}

void ParPro::Get_Worksplit(int NX, int Procesos, int p, int &pix, int &pfx){
int Intervalo, Residuo;

        Intervalo = NX/Procesos;
        Residuo = NX%Procesos;

        if(p != Procesos-1){
                pix = p*Intervalo;
                pfx = (p+1)*Intervalo;
        }
        else{
                pix = p*Intervalo;
                pfx = (p+1)*Intervalo + Residuo;
        }
```

```
}

void ParPro::SendData(double *LocalSend, int Ix, int Fx){

        if(Rank != 0 && Rank != Procesos-1){
                MPI_Send(&LocalSend[LNH(Ix,0,0)], Halo*NY, MPI_DOUBLE, Rank-1, 0, MPI_COMM
                MPI_Send(&LocalSend[LNH(Fx - Halo,0,0)], Halo*NY, MPI_DOUBLE, Rank+1, 0, M
        }
        else if(Rank == 0){
                MPI_Send(&LocalSend[LNH(Fx - Halo,0,0)], Halo*NY, MPI_DOUBLE, 1, 0, MPI_COM
        }
        else{
                MPI_Send(&LocalSend[LNH(Ix,0,0)], Halo*NY, MPI_DOUBLE, Procesos-2, 0, MPI_C
        }

}

void ParPro::ReceiveData(double *LocalReceive, int Ix, int Fx){
int p;
MPI_Status ST;

        if(Rank != 0 && Rank != Procesos-1){
                MPI_Recv(&LocalReceive[LNH(Ix - Halo,0,0)], Halo*NY, MPI_DOUBLE, Rank-1, 0
                MPI_Recv(&LocalReceive[LNH(Fx,0,0)], Halo*NY, MPI_DOUBLE, Rank+1, 0, MPI_CO
        }
        else if(Rank == 0){
                MPI_Recv(&LocalReceive[LNH(Fx,0,0)], Halo*NY, MPI_DOUBLE, 1, 0, MPI_COMM_WO
        }
        else if(Rank == Procesos-1){
                MPI_Recv(&LocalReceive[LNH(Ix - Halo,0,0)], Halo*NY, MPI_DOUBLE, Procesos-
        }


}

void ParPro::SendMatrixToZero(double *LocalMatrix, double *GlobalMatrix, int NX, int NY, i
int i, j, p;
int pix, pfx;
MPI_Status ST;

        if(Rank != 0){
                MPI_Send(&LocalMatrix[LNH(Ix,0,0)], NY*(Fx-Ix), MPI_DOUBLE, 0, 0, MPI_COMM
        }

        if(Rank == 0){
                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                GlobalMatrix[G(i,j,0)] = LocalMatrix[LNH(i,j,0)];
                        }
                }

                for(p = 1; p < Procesos; p++){
                        Get_Worksplit(NX, Procesos, p, pix, pfx);
                        MPI_Recv(&GlobalMatrix[G(pix,0,0)], NY*(pfx - pix), MPI_DOUBLE, p,
                }
```

```
        }

}

void ParPro::SendDataToZero(double DataSent, double *DataReceived){
int i, p;
MPI_Status ST;

        if(Rank != 0){
                MPI_Send(&DataSent, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }

        if(Rank == 0){
                DataReceived[Rank] = DataSent;
                for(p = 1; p < Procesos; p++){
                        MPI_Recv(&DataReceived[p], 1, MPI_DOUBLE, p, 0, MPI_COMM_WORLD, &ST
                }
        }
}

void ParPro::SendDataToAll(double DataSent, double &DataReceived){
int p;
MPI_Status ST;

        if(Rank == 0){
                for(p = 1; p < Procesos; p++){
                        MPI_Send(&DataSent, 1, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
                }
                DataReceived = DataSent;
        }

        if(Rank != 0){
                MPI_Recv(&DataReceived, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &ST);
        }

}

void ParPro::Execute(){

        Rango();
        Processes();

}
```

Mesher Class Header Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <cmath>
#include <stdio.h>
#include <time.h>
#include <mpi.h>

using namespace std;
```

```
class Mesher{
        private:
                //Datos sobre el problema
                        string Problema; //Problema (Canal/Canal con Cilindro/Perfil)

                //Datos sobre la geometr a del problema
                        double ChannelLength; //Longitud del canal
                        double ChannelHeight; //Altura del canal
                        double ChannelDepth; //Profundidad del canal

                //Datos del mallado
                        int OpcionR; //Tipo de discretizaci n en la direcci n radial
                        int OpcionA; //Tipo de discretizaci n en la direcci n axial
                        double StretFactorX; //Factor de estrechamiento de la discretizaci
                        double StretFactorY; //Factor de estrechamiento de la discretizaci

        public:
                //Constructor de la clase
                Mesher(Memory, ReadData, ParPro, int);

                //Datos para la computaci n en paralelo
                int Procesos;
                int Rank;
                int Ix;
                int Fx;

                //Densidad del mallado
                int NX; //Direcci n axial total
                int NY; //Direcci n radial

                //Matrices de coordenadas de discretizaci n
                double *AxialCoord; //Matriz coordenada axial de la distribuci n

                //Caso mallado tipo Staggered
                double *MP; //Coordenadas matriz de presi n/temperatura
                double *MU; //Coordenadas matriz velocidades axiales
                double *MR; //Coordenadas matriz velocidades radiales

                //Matrices de distancias de vol menes de control
                double *DeltasMP; //Deltas X y R de la matriz de Presi n/Temperatura
                double *DeltasMU; //Deltas X y R de la matriz de velocidades axiales (U)
                double *DeltasMR; //Deltas X y R de la matriz de velocidades radiales (V)

                //Matrices de superficies de los vol menes de control
                double *SupMP;

                //Matriz de vol menes de control
                double *VolMP;

                //M todos de la clase del mallador
                void AllocateMemory(Memory); //Alojamiento de memoria para cada matriz

                void Get_AxialCoordinates(); //C lculo del la posici n axial de los nodo
                void Get_Mesh(); //Creaci n de todas las mallas
                void Get_Deltas(); //C lculo de las distancias entre nodos en cada una de
                void Get_Surfaces(); //C lculo de las superficies de los vol menes de co
```

```
                void Get_Volumes(); //C lculo de los vol menes de control

                void PrintTxt();
                void MallaVTK2D(string, string, string, double*, int, int);

                void ExecuteMesher(Memory, ParPro); //Ejecutar todos los procesos del mall
};
```

Mesher Class Cpp Code

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <math.h>
#include <stdlib.h>
#include <cmath>
#include <stdio.h>
#include <cassert>
#include <time.h>
#include <bits/stdc++.h>
#include <string>
#include <time.h>
#include <mpi.h>

using namespace std;

#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/"

#define PI 3.141592653589793

#define sind(x) sin(x * (PI/180.0)) //C lculo seno en grados
#define cosd(x) cos(x * (PI/180.0)) //C lculo coseno en grados
#define tand(x) tan(x * (PI/180.0)) //C lculo tangente en grados

#define Hyp(x1, x2, y1, y2) sqrt(pow(x2-x1,2.0) + pow(y2-y1,2.0)) //C lculo hipotenusa

#define G(i,j,dim) (((j) + (i)*NY) + NX*NY*(dim)) //Global Index
#define GU(i,j,dim) (((j) + (i)*NY) + (NX+1)*NY*(dim)) //Global Index Matriz U
#define GR(i,j,dim) (((j) + (i)*(NY+1)) + NX*(NY+1)*(dim)) //Global Index Matriz R

#define MU(i,j,dim) ((j) + NY*(dim))
#define MR(i,j,dim) ((i) + (Fx - Ix)*(dim) + (Fx - Ix)*(4)*(j))

#define VR(i,j,dim) ((i) + (Fx - Ix)*(j))

#define LNH(i,j,dim) (((j) + ((i) - Ix + Halo)*NY)) //Local Index No Halo
#define LSH(i,j,dim) (((j) + ((i) - Ix)*NY) + NY*(Fx-Ix + 2*Halo)*dim) //Local Index Si Ha

//Constructor del mallador
Mesher::Mesher(Memory M1, ReadData R1, ParPro MPI1){
```

```
        //Datos sobre el problema
                Problema = R1.NumericalData[4]; //Problema

        //Datos para la computaci n en paralelo
                Procesos = MPI1.Procesos;
                Rank = MPI1.Rank;

        //Datos sobre la geometr a del problema

                //Geometr a del caNAl
                ChannelLength = R1.GeometryData[0]; //Longitud del canal
                ChannelHeight = R1.GeometryData[1]; //Altura del canal
                ChannelDepth = R1.GeometryData[2]; //Profundidad del canal

                //Densidad del mallado
                NX = R1.NumericalData[0]; //N mero de nodos direcci n X
                NY = R1.NumericalData[1]; //N mero de nodos direcci n Y

                //Opciones del mallado
                OpcionR = R1.NumericalData[2]; //Tipo de discretizaci n en la direcci n
                OpcionA = R1.NumericalData[3]; //Tipo de discretizaci n en la direcci n
                StretFactorX = R1.ProblemData[0]; //Factor de estrechamiento de la discret
                StretFactorY = R1.ProblemData[1]; //Factor de estrechamiento de la discret

}

//C lculo del la posici n axial de los nodos de velocidad axial
void Mesher::Get_AxialCoordinates(){
int i;
double I;

        for(i = 0; i < NX + 1; i++){
                I = i;
                AxialCoord[i] = I*(ChannelLength/NX);
        }

}

//Alojamiento de memoria para cada matriz
void Mesher::AllocateMemory(Memory M1){

        AxialCoord = M1.AllocateDouble(NX + 1, 1, 1); //Matriz coordenada axial de la dist

        //Matrices de coordeNXdas de discretizaci n
        MP = M1.AllocateDouble(NX, NY, 2); //CoordeNXdas matriz de presi n/temperatura
        MU = M1.AllocateDouble(NX + 1, NY, 2); //CoordeNXdas matriz velocidades axiales
        MR = M1.AllocateDouble(NX, NY + 1, 2); //CoordeNXdas matriz velocidades radiales

        //Matrices de distancias de vol menes de control
        DeltasMP = M1.AllocateDouble(NX, NY, 2); //Deltas X y R de la matriz de Presi n/T
        DeltasMU = M1.AllocateDouble(NX + 1, NY, 2); //Deltas X y R de la matriz de velocid
        DeltasMR = M1.AllocateDouble(NX, NY + 1, 2); //Deltas X y R de la matriz de velocid

        //Matrices de superficies de los vol menes de control
        SupMP = M1.AllocateDouble(NX, NY, 4);
```

```
        //Matriz de vol menes de control
        VolMP = M1.AllocateDouble(NX, NY, 1);

}

//Creaci n de todas las mallas de tipo Staggered
void Mesher::Get_Mesh(){
int i, j;
double ny = NY;
double J;

        //CoordeNXdas Axiales

        //Coordenadas axiales matriz de velocidades axiales (U)
        for(i = 0; i < NX + 1; i++){
                for(j = 0; j < NY; j++){
                        MU[GU(i,j,0)] = AxialCoord[i];
                }
        }

        //CoordeNXdas axiales matriz de velocidades radial (V)
        for(i = 0; i < NX; i++){
                for(j = 0; j < NY + 1; j++){
                        MR[GR(i,j,0)] = 0.5*(AxialCoord[i] + AxialCoord[i+1]);
                }
        }

        //CoordeNXdas axiales matriz de Presi n/Temperatura
        for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                        MP[G(i,j,0)] = 0.5*(AxialCoord[i] + AxialCoord[i+1]);
                }
        }

        //CoordeNXdas Radiales

        //CoordeNXdas radiales de la matriz de velocidades radial (V)
        if(OpcionR == 1){ //Regular
                for(i = 0; i < NX; i++){
                        for(j = 0; j < NY + 1; j++){
                                J = j;
                                MR[GR(i,j,1)] = J*(ChannelHeight/NY);
                        }
                }
        }
        else if(OpcionR == 2){ //Tangencial hiperb lica
                for(i = 0; i < NX; i++){
                        for(j = 0; j < NY + 1; j++){
                                J = j;
                                MR[GR(i,j,1)] = (0.50*ChannelHeight)*(1.0 + (tanh(StretFact
                        }
                }
        }

        //CoordeNXdas radiales de la matriz de Presi n/Temperatura
        for(i = 0; i < NX; i++){
```

```
                for(j = 0; j < NY; j++){
                        MP[G(i,j,1)] = 0.5*(MR[GR(i,j,1)] + MR[GR(i,j + 1,1)]);
                }
        }

        //Coordenadas radiales de la matriz de velocidades axial (U)
        for(j = 0; j < NY; j++){
                //Parte Izquierda
                MU[GU(0,j,1)] = MP[G(0,j,1)];

                //Parte Derecha
                MU[GU(NX,j,1)] = MP[G(NX - 1,j,1)];

                for(i = 1; i < NX; i++){
                        MU[GU(i,j,1)] = 0.5*(MP[G(i-1,j,1)] + MP[G(i,j,1)]);
                }
        }

}

//Pasar los resultados de las mallas a un txt
void Mesher::PrintTxt(){
int i, j;
string Carpeta = "GnuPlotResults/Meshes/";
ofstream file;
string FileName;
stringstream InitialNameMP;
string FinalNameMP;

        FileName = "MallaMP.txt";

        InitialNameMP<<DIRECTORIO<<Carpeta<<FileName;

        FinalNameMP = InitialNameMP.str();
        file.open(FinalNameMP.c_str());

                for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                        file<<MP[G(i,j,0)]<<"\t"<<MP[G(i,j,1)]<<"\t"<<endl;
                        }
                file<<endl;
        }

        file.close();

stringstream InitialNameMU;
string FinalNameMU;

        FileName = "MallaMU.txt";

        InitialNameMU<<DIRECTORIO<<Carpeta<<FileName;

        FinalNameMU = InitialNameMU.str();
        file.open(FinalNameMU.c_str());

                for(i = 0; i < NX + 1; i++){
```

```
                for(j = 0; j < NY; j++){
                        file<<MU[GU(i,j,0)]<<"\t"<<MU[GU(i,j,1)]<<"\t"<<endl;
                        }
                file<<endl;
        }

        file.close();

stringstream InitialNameMR;
string FinalNameMR;

        FileName = "MallaMR.txt";

        InitialNameMR<<DIRECTORIO<<Carpeta<<FileName;

        FinalNameMR = InitialNameMR.str();
        file.open(FinalNameMR.c_str());

                for(i = 0; i < NX; i++){
                for(j = 0; j < NY + 1; j++){
                        file<<MR[GR(i,j,0)]<<"\t"<<MR[GR(i,j,1)]<<"\t"<<endl;
                        }
                file<<endl;
        }

        file.close();

}

//Clculo de las distancias entre nodos en cada uNX de las matrices
void Mesher::Get_Deltas(){
int i, j;

        //Deltas X e Y de la matriz de Presin/Temperatura
        for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                        DeltasMP[G(i,j,0)] = MU[GU(i + 1,j,0)] - MU[GU(i,j,0)]; //Deltas
                        DeltasMP[G(i,j,1)] = MR[GR(i,j + 1,1)] - MR[GR(i,j,1)]; //Deltas
                }
        }

        //Deltas X e Y de la matriz de velocidades axiales (U)
        for(j = 0; j < NY; j++){
                //Parte Izquierda
                DeltasMU[GU(0,j,0)] = MP[G(0,j,0)] - MU[GU(0,j,0)]; //Deltas X
                DeltasMU[GU(0,j,1)] = MR[GR(0,j + 1,1)] - MR[GR(0,j,1)]; //Deltas Y

                //Parte Derecha
                DeltasMU[GU(NX,j,0)] = MU[GU(NX,j,0)] - MP[G(NX - 1,j,0)]; //Deltas X
                DeltasMU[GU(NX,j,1)] = MR[GR(NX - 1,j + 1,1)] - MR[GR(NX - 1,j,1)]; //Delt

                for(i = 1; i < NX; i++){
                        DeltasMU[GU(i,j,0)] = MP[G(i,j,0)] - MP[G(i - 1,j,0)]; //Deltas X
                        DeltasMU[GU(i,j,1)] = MR[GR(i,j + 1,1)] - MR[GR(i,j,1)]; //Deltas
                }
        }
```

```
        //Deltas X e Y de la matriz de velocidades Verticales (V)
        for(i = 0; i < NX; i++){
                //Parte abajo
                DeltasMR[GR(i,0,0)] = MU[GU(i + 1,0,0)] - MU[GU(i,0,0)]; //Deltas X
                DeltasMR[GR(i,0,1)] = MP[G(i,0,1)] - MR[GR(i,0,1)]; //Deltas Y

                //Parte arriba
                DeltasMR[GR(i,NY,0)] = MU[GU(i + 1,NY - 1,0)] - MU[GU(i,NY - 1,0)]; //Delt
                DeltasMR[GR(i,NY,1)] = MR[GR(i,NY,1)] - MP[G(i,NY - 1,1)]; //Deltas Y

                for(j = 1; j < NY; j++){
                        DeltasMR[GR(i,j,0)] = MU[GU(i + 1,j,0)] - MU[GU(i,j,0)]; //Deltas
                        DeltasMR[GR(i,j,1)] = MP[G(i,j,1)] - MP[G(i,j - 1,1)]; //Deltas Y
                }
        }

}

//C lculo de las superficies de los vol menes de control
void Mesher::Get_Surfaces(){
int i, j;

        for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                        SupMP[G(i,j,0)] = DeltasMU[GU(i,j,1)]*ChannelDepth;
                        SupMP[G(i,j,1)] = DeltasMU[GU(i + 1,j,1)]*ChannelDepth;
                        SupMP[G(i,j,2)] = DeltasMR[GR(i,j,0)]*ChannelDepth;
                        SupMP[G(i,j,3)] = DeltasMR[GR(i,j + 1,0)]*ChannelDepth;
                }
        }

}

//C lculo de los vol menes de control
void Mesher::Get_Volumes(){
int i, j;

        for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                        VolMP[G(i,j,0)] = DeltasMP[G(i,j,0)]*DeltasMP[G(i,j,1)]*ChannelDep
                }
        }

}

//Pasar los resultados a un archivo VTK en 2D
void Mesher::MallaVTK2D(string Carpeta, string Variable, string NombreFile, double *MC, in
int i, j;

        ofstream file;
    stringstream InitialName;
    string FinalName;

        InitialName<<DIRECTORIO<<Carpeta<<NombreFile<<".vtk";
```

```
                FinalName = InitialName.str();
        file.open(FinalName.c_str());

        file<<"# vtk DataFile Version 2.0"<<endl;
        file<<Variable<<endl;
        file<<"ASCII"<<endl;
        file<<endl;
        file<<"DATASET STRUCTURED_GRID"<<endl;
        file<<"DIMENSIONS"<<"   "<<NX<<"   "<<NY<<"   "<<1<<endl;
        file<<endl;
        file<<"POINTS"<<"   "<<NX*NY<<"   "<<"double"<<endl;

                for(j = 0; j < NY; j++){
                        for(i = 0; i < NX; i++){
                                file<<MC[G(i,j,0)]<<"   "<<MC[G(i,j,1)]<<"   "<<0.0<<endl;
                        }
                }

        file<<endl;
                file<<"POINT_DATA"<<"   "<<NX*NY<<endl;
        file<<"SCALARS "<<Variable<<" double"<<endl;
        file<<"LOOKUP_TABLE"<<"   "<<Variable<<endl;
        file<<endl;
        for(j = 0; j < NY; j++){
                        for(i = 0; i < NX; i++){
                                file<<0.0<<" ";
                        }
                }

        file.close();
}

//Ejecutar todos los procesos del mallador
void Mesher::ExecuteMesher(Memory M1, ParPro MPI1){
int i,j;

        MPI1.Initial_WorkSplit(NX, Ix, Fx);
        AllocateMemory(M1); //Alojamiento de memoria para cada matriz
        Get_AxialCoordinates(); //C lculo del la posici n axial de los nodos de velocidad
        Get_Mesh(); //Creaci n de todas las mallas
        Get_Deltas(); //C lculo de las distancias entre nodos en cada uNX de las matrices
        Get_Surfaces(); //C lculo de las superficies de los vol menes de control
        Get_Volumes(); //C lculo de los vol menes de control

        if(Rank == 0){
                PrintTxt();
                MallaVTK2D("ParaviewResults/MeshResults/", "MallaBase", "MallaMP", MP, NX,
        }

        cout<<"Mesh created."<<endl;

}
```

Solver Class Header Code

```
#include <iostream>
```

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <mpi.h>

using namespace std;

class Solver{
        private:

                //Problema a simular
                int Problema;

                //Densidad del mallado
                int NX; //Direcci n horizontal
                int NY; //Direcci n vertical

                //Datos para la computaci n en paralelo
                int Ix;
                int Fx;
                int Procesos;
                int Halo;
                int Rank;

                //Datos de la geometr a
                double ChannelLength; //Longitud del canal
                double ChannelHeight; //Altura del canal
                double ChannelDepth; //Profundidad del canal

                //Datos F sicos del problema
                double Peclet;
                double Uref;
                double Vref;
                double RhoRef;
                double Alpha;
                double AlphaRad;

                double PhiIz;
                double PhiDer;
                double PhiTop;
                double PhiBotton;

                string EsquemaLargo;
                string EsquemaCorto;

                double FR;

                double DeltaT;

                //Datos sobre el solver iterativo
                double ConvergenciaGlobal;
                double ConvergenciaGS;

                double MaxDiffGS;
                double MaxDiffGlobal;
```

```
        double *PDiff;

    public:
        Solver(Memory, ReadData, ParPro, Mesher, int);

        //Matrices de los mapas de propiedades del problema

        //Matrices globales de propiedades

        //Step Presente
        double *PhiGlobalPres;
        double *UglobalPres;
        double *VglobalPres;

        //Step Futuro
        double *PhiGlobalFut;
        double *UglobalFut;
        double *VglobalFut;

        double *PDT;

        //Matrices locales de propiedades

        //Streamline
        double *PhiLocalPres;
        double *PhiLocalFut;

        double *PhiLocalSup;

        //Velocidad Horizontal
        double *UlocalPres;
        double *UlocalFut;

        //Velocidad Vertical
        double *VlocalPres;
        double *VlocalFut;

        //Matrices de valores de las propiedades en las paredes de los vol menes
        double *UwallsMU;
        double *VwallsMR;

        double *PhiWallsMU;
        double *PhiWallsMR;

        //Matrices de c lculo de contribuciones a las propiedades

        double *Dw;
        double *De;
        double *Ds;
        double *Dn;

        double *Cw;
        double *Ce;
        double *Cs;
        double *Cn;
```

```
        double *aw;
        double *ae;
        double *as;
        double *an;
        double *ap;

        double *bp;

        //Matrices y arrays de condiciones de contorno

        //Condiciones de contorno del mapa de valor de streamlines
        double *PhiLeft; //Presi n pared izquierda
        double *PhiRight; //Presi n pared derecha

        double *PhiUp; //Presi n pared superior
        double *PhiDown; //Presi n pared inferior

        //Condiciones de contorno del mapa de velocidades axiales (U)
        double *Uleft; //Velocidad axial pared izquierda
        double *Uright; //Velocidad axial pared derecha

        double *Uup; //Velocidad axial pared superior
        double *Udown; //Velocidad axial pared inferior

        //Condiciones de contorno del mapa de velocidades radiales (V)
        double *Vleft; //Velocidad radial pared izquierda
        double *Vright; //Velocidad radial pared derecha

        double *Vup; //Velocidad radial pared superior
        double *Vdown; //Velocidad radial pared inferior

        //Matrices para verificar el caso anal tico
        double *CoordenadasAnalitico;
        double *PhiAnalitico;

        double *PhiReal;

        double *MinAbajo;
        double *MinArriba;

        double *PhiAbajo;
        double *PhiArriba;

        //M todos de la clase Solver
        void AllocateMatrix(Memory);
        void Get_AnalyticalResults();
        double ConvectiveScheme(double, double, double, double, double, double, do
        void Get_StepTime(Mesher, ParPro);

        void InitializeFields(Mesher); //Pasar todos los coeficientes termoq mico
        void UpdateBoundaryConditions(Mesher);

        void Get_PhiWalls(Mesher, ParPro);
        void Get_PhiWalls2(Mesher, ParPro);
        void Get_Diffusive(Mesher);
```

```
                void Get_Convective(Mesher);

                void Get_Coeficients(Mesher); //C lculo de los coeficientes de discretiza
                void Get_BetaCoefficient(Mesher);

                void Get_StreamlinesFR(ParPro);
                void Get_Streamlines(ParPro);
                void Get_MaxDifGS(ParPro);
                void Get_Velocities(Mesher); //C lculo del mapa de velocidades futuro

                void Get_Stop(ParPro);

                void UpdatePropertiesFields();

                void Get_NumericalResults(Mesher, string, string);
                void RelativeError(Mesher, string, string);
                void EscalarVTK2D(Mesher, string, string, string, double*, double*, int, in
                void VectorialVTK2D(Mesher, string, string, string, double*, double*, doub

                void ExecuteSolver(Memory, ReadData, ParPro, Mesher, int);

};
```

Solver Class Cpp Code

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <chrono>
#include <mpi.h>

#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade
#include "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/Codes/Heade

using namespace std;

#define PI 3.141592653589793

#define G(i,j,dim) (((j) + (i)*NY) + NX*NY*(dim)) //Global Index
#define GU(i,j,dim) (((j) + (i)*NY) + (NX+1)*NY*(dim)) //Global Index Matriz U
#define GR(i,j,dim) (((j) + (i)*(NY+1)) + NX*(NY+1)*(dim)) //Global Index Matriz R

#define MU(i,j,dim) ((j) + NY*(dim))
#define MR(i,j,dim) ((i) + (Fx - Ix)*(dim) + (Fx - Ix)*(4)*(j))

#define VR(i,j,dim) ((i) + (Fx - Ix)*(j))

#define LU(i, j, dim) (((j) + ((i)-Ix) * NY)) //Local Index Axial Velocity Nodes
#define LR(i, j, dim) (((j) + ((i)-Ix) * (NY + 1))) //Local Index Radial Velocity Nodes
```

```
#define LNH(i,j,dim) (((j) + ((i) - Ix + Halo)*NY)) //Local Index No Halo
#define LSH(i,j,dim) (((j) + ((i) - Ix)*NY) + NY*(Fx-Ix + 2*Halo)*dim) //Local Index Si Ha

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDiffusion/"

//Constructor del mallador
Solver::Solver(Memory M1, ReadData R1, ParPro MPI1, Mesher MESH){

        //Datos del problema
        NX = MESH.NX;
        NY = MESH.NY;

        //Datos para la computacion en paralelo
        Procesos = MPI1.Procesos;
        Rank = MPI1.Rank;
        Ix = MESH.Ix;
        Fx = MESH.Fx;
        Halo = MPI1.Get_Halo();

        ConvergenciaGlobal = 1e-8;
        ConvergenciaGS = 1e-7;

        //Geometra del canal
        ChannelLength = R1.GeometryData[0]; //Longitud del canal
        ChannelHeight = R1.GeometryData[1]; //Altura del canal
        ChannelDepth = R1.GeometryData[2]; //Profundidad del canal

        Problema = R1.NumericalData[4];

        //Datos Fsicos del problema
        Peclet = R1.ProblemPhysicalData[4];
        RhoRef = R1.ProblemPhysicalData[2];

        Uref = Peclet/(RhoRef*ChannelLength);
        Vref = Peclet/(RhoRef*ChannelLength);

        Alpha =  R1.ProblemPhysicalData[3];
        AlphaRad = Alpha*PI/180;

        PhiIz = R1.ProblemPhysicalData[5];
        PhiDer = R1.ProblemPhysicalData[6];

        EsquemaLargo = R1.Esquema;

}

//Alojamiento de memoria para las matrices necesarias
void Solver::AllocateMatrix(Memory M1){

        //Matrices de los mapas de propiedades del problema

                //Matrices globales de propiedades

                if(Rank == 0){
```

```
                //Step Presente
                PhiGlobalPres = M1.AllocateDouble(NX, NY, 1);
                UglobalPres = M1.AllocateDouble(NX, NY, 1);
                VglobalPres = M1.AllocateDouble(NX, NY, 1);

                //Step Futuro
                PhiGlobalFut = M1.AllocateDouble(NX, NY, 1);
                UglobalFut = M1.AllocateDouble(NX, NY, 1);
                VglobalFut = M1.AllocateDouble(NX, NY, 1);

                PDiff = M1.AllocateDouble(Procesos, 1, 1);
                PDT = M1.AllocateDouble(Procesos, 1, 1);

                CoordenadasAnalitico = M1.AllocateDouble(11, 1, 1);
                PhiAnalitico = M1.AllocateDouble(11, 1, 1);

                PhiReal = M1.AllocateDouble(11, 1, 1);

                MinAbajo = M1.AllocateDouble(11, 1, 1);
                MinArriba = M1.AllocateDouble(11, 1, 1);

                PhiAbajo = M1.AllocateDouble(11, 1, 1);
                PhiArriba = M1.AllocateDouble(11, 1, 1);

        }


        //Matrices locales de propiedades

        //Streamline
        PhiLocalPres = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        PhiLocalFut = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

        PhiLocalSup = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

        //Velocidad Horizontal
        UlocalPres = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        UlocalFut = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

        //Velocidad Vertical
        VlocalPres = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        VlocalFut = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

        //Matrices de valores de las propiedades en las paredes de los vol menes
        UwallsMU = M1.AllocateDouble(Fx - Ix + 1, NY, 1);
        VwallsMR = M1.AllocateDouble(Fx - Ix, NY + 1, 1);

        PhiWallsMU = M1.AllocateDouble(Fx - Ix + 1, NY, 1);
        PhiWallsMR = M1.AllocateDouble(Fx - Ix, NY + 1, 1);

        //Matrices de c lculo de contribuciones a las propiedades

        Dw = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        De = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        Ds = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        Dn = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
```

```
                Cw = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
                Ce = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
                Cs = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
                Cn = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

                aw = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
                ae = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
                as = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
                an = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
                ap = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

                bp = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

                //Matrices y arrays de condiciones de contorno

                if(Rank == 0){

                        PhiLeft = M1.AllocateDouble(1, NY, 1); //Presi n pared izquierda
                        Uleft = M1.AllocateDouble(1, NY, 1); //Velocidad axial pared izqui
                        Vleft = M1.AllocateDouble(1, NY, 1); //Velocidad radial pared izqu

                }
                else if(Rank == Procesos - 1){

                        PhiRight = M1.AllocateDouble(1, NY, 1); //Presi n pared derecha
                        Uright = M1.AllocateDouble(1, NY, 1); //Velocidad axial pared dere
                        Vright = M1.AllocateDouble(1, NY, 1); //Velocidad radial pared der

                }

                PhiUp = M1.AllocateDouble(Fx - Ix, 1, 1);
                PhiDown = M1.AllocateDouble(Fx - Ix, 1, 1);

                Uup = M1.AllocateDouble(Fx - Ix, 1, 1);
                Udown = M1.AllocateDouble(Fx - Ix, 1, 1);

                Vup = M1.AllocateDouble(Fx - Ix, 1, 1);
                Vdown = M1.AllocateDouble(Fx - Ix, 1, 1);

}

//Inicializaci n de los campos de temperaturas
void Solver::InitializeFields(Mesher MESH){
int i, j;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){

                        PhiLocalPres[LNH(i,j,0)] = 0.0;
                        PhiLocalFut[LNH(i,j,0)] =  0.0;

                        PhiLocalSup[LNH(i,j,0)] =  0.0;

                }
        }
```

```
}


//Asignaci n de temperaturas a las condiciones de contorno
void Solver::UpdateBoundaryConditions(Mesher MESH){
int i, j;

        if(Rank == 0){
                if(Problema == 1){
                        for(j = 0; j < NY; j++){
                                PhiLeft[j] = PhiIz;
                        }
                }
                else if(Problema == 2){
                        for(j = 0; j < NY; j++){
                                PhiLeft[j] = PhiIz;
                        }
                }
                else if(Problema == 3){
                        for(j = 0; j < NY; j++){
                                PhiLeft[j] = PhiIz;
                        }
                }
                else if(Problema == 4){
                        for(j = 0; j < NY; j++){
                                PhiLeft[j] = 1.0 - tanh(10.0);
                        }
                }
        }
        else if(Rank == Procesos - 1){
                if(Problema == 1){
                        for(j = 0; j < NY; j++){
                                PhiRight[j] = PhiDer;
                        }
                }
                else if(Problema == 2){
                        for(j = 0; j < NY; j++){
                                PhiRight[j] = PhiDer;
                        }
                }
                else if(Problema == 3){
                        for(j = 0; j < NY; j++){
                                PhiRight[j] = PhiDer;
                        }
                }
                else if(Problema == 4){
                        for(j = 0; j < NY; j++){
                                PhiRight[j] = 1.0 - tanh(10.0);
                        }
                }
        }

        if(Problema == 1){
                for(i = Ix; i < Fx; i++){
                        PhiUp[i - Ix] = PhiLocalFut[LNH(i,NY-1,0)];
```

```
                        PhiDown[i - Ix] = PhiLocalFut[LNH(i,0,0)];
                }
        }
        else if(Problema == 2){
                for(i = Ix; i < Fx; i++){
                        PhiUp[i - Ix] = PhiLocalFut[LNH(i,NY-1,0)];
                        PhiDown[i - Ix] = PhiLocalFut[LNH(i,0,0)];
                }
        }
        else if(Problema == 3){
                for(i = Ix; i < Fx; i++){
                        PhiUp[i - Ix] = PhiIz;
                        PhiDown[i - Ix] = PhiDer;
                }
        }
        else if(Problema == 4){
                for(i = Ix; i < Fx; i++){
                        PhiUp[i - Ix] = 1.0 - tanh(10.0);
                        if((MESH.MR[GR(i,0,0)] - 0.50*ChannelLength) <= 0.0){
                                PhiDown[i - Ix] = 1.0 + tanh((2.0*(MESH.MR[GR(i,0,0)] - 0.5
                        }
                        else if((MESH.MR[GR(i,0,0)] - 0.50*ChannelLength) > 0.0){
                                PhiDown[i - Ix] = PhiLocalFut[LNH(i,0,0)];
                        }

                }
        }

}

//C lculo del DeltaT para el siguiente Step
void Solver::Get_StepTime(Mesher MESH, ParPro MPI1){
int i, j;
double DeltasT = 1000.0;
double Tpar = 0.05;
MPI_Status ST;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){

                        //CFL Velocidades (U + V)
                        if(abs((Tpar*MESH.DeltasMP[G(i,j,0)])/(abs(UlocalFut[LNH(i,j,0)]))
                                DeltasT = (Tpar * MESH.DeltasMP[G(i, j, 0)]) / (abs(UlocalF
                        }
                        if(abs((Tpar*MESH.DeltasMP[G(i,j,1)])/(abs(VlocalFut[LNH(i,j,0)]))
                                DeltasT = (Tpar * MESH.DeltasMP[G(i, j, 1)]) / (abs(VlocalF
                        }

                        //CFL Difusivo
                        if((Tpar*RhoRef*pow(MESH.DeltasMP[G(i,j,0)],2.0))/(1.0 + 1e-10) <
                                DeltasT = (Tpar * RhoRef * pow(MESH.DeltasMP[G(i, j, 0)],
                        }
                        if((Tpar*RhoRef*pow(MESH.DeltasMP[G(i,j,1)],2.0))/(1.0 + 1e-10) <
                                DeltasT = (Tpar * RhoRef * pow(MESH.DeltasMP[G(i, j, 1)],
                        }
                }
```

```
        }

        MPI1.SendDataToZero(DeltasT, PDT);

        double DT;
        if(Rank == 0){
                DT = PDT[0];
                for(i = 1; i < Procesos; i++){
                        if(PDT[i] <= DT){
                                DT = PDT[i];
                        }
                }
        }

        MPI1.SendDataToAll(DT, DeltaT);

}

//Funci n con los diferentes esquemas convectivos utilizados
double Solver::ConvectiveScheme(double CoordObjetivo, double Velocity, double Coord1, doub

double PhiObjetivo;

double CoordD;
double PhiD;

double CoordC;
double PhiC;

double CoordU;
double PhiU;

        if (Velocity < 0.0 || (Phi1 == 0.0 && Coord1 == 0.0)){

                CoordD = Coord2;
                PhiD = Phi2;
                CoordC = Coord3;
                PhiC = Phi3;
                CoordU = Coord4;
                PhiU = Phi4;

        }
        else if(Velocity >= 0.0 || (Phi4 == 0.0 && Coord4 == 0.0)){

                CoordD = Coord3;
                PhiD = Phi3;
                CoordC = Coord2;
                PhiC = Phi2;
                CoordU = Coord1;
                PhiU = Phi1;

        }

        //Adimensionalizacion
        double PhiAdimC;
        double AdimCoordC;
```

```
        double AdimCoordE;
        //double Xade;

        PhiAdimC = (PhiC - PhiU)/(PhiD - PhiU);

        AdimCoordC = (CoordC - CoordU)/(CoordD - CoordU);

        AdimCoordE = (CoordObjetivo - CoordU)/(CoordD - CoordU);

        //Evaluacion
        double PhiF;

        if (PhiD == PhiU){
                PhiObjetivo = PhiD;
        }
        else{
                if(Esquema == "CDS"){
                        PhiF = ((AdimCoordE - AdimCoordC)/(1.0 - AdimCoordC)) + ((AdimCoord
                }
                else if(Esquema == "UDS"){
                        PhiF = PhiAdimC;
                }
                else if(Esquema == "SUDS"){
                        PhiF = (AdimCoordE/AdimCoordC)*PhiAdimC;
                }
                else if(Esquema == "QUICK"){
                        PhiF = AdimCoordE + (((AdimCoordE*(AdimCoordE - 1.0))/(AdimCoordC*
                }
                else if(Esquema == "SMART"){
                        if(PhiAdimC > 0 && PhiAdimC < AdimCoordC/3.0){
                                PhiF = -((AdimCoordE*(1.0 - 3.0*AdimCoordC + 2.0*AdimCoordE
                        }
                        else if(PhiAdimC > AdimCoordC/6.0 && PhiAdimC < (AdimCoordC/AdimCo
                                PhiF = ((AdimCoordE*(AdimCoordE - AdimCoordC))/(1.0 - Adim
                        }
                        else if(PhiAdimC > (AdimCoordC/AdimCoordE)*(1.0 + AdimCoordE - Adi
                                PhiF = 1.0;
                        }
                        else{
                                PhiF = PhiAdimC;
                        }
                }

                //Dimensionalizacion
                PhiObjetivo = PhiU + (PhiD - PhiU)*PhiF;
        }

        return PhiObjetivo;

}



//C lculo de las densidades en las paredes de lo vol menes de control
void Solver::Get_PhiWalls(Mesher MESH, ParPro MPI1){
int i, j;
```

```
//Comunicaci n de densidades entre los procesos
MPI1.SendData(PhiLocalFut, Ix, Fx);
MPI1.ReceiveData(PhiLocalFut, Ix, Fx);

//Nodos R
for(i = Ix; i < Fx; i++){
        //Parte abajo
        PhiWallsMR[LR(i,0,0)] = PhiDown[i - Ix];
        PhiWallsMR[LR(i,1,0)] = ConvectiveScheme(MESH.MR[GR(i,1,1)], VwallsMR[LR(i

        //Parte arriba
        PhiWallsMR[LR(i,NY,0)] = PhiUp[i - Ix];
        PhiWallsMR[LR(i,NY - 1,0)] = ConvectiveScheme(MESH.MR[GR(i,NY - 1,1)], Vwa

        for(j = 2; j < NY - 1; j++){
                PhiWallsMR[LR(i,j,0)] = ConvectiveScheme(MESH.MR[GR(i,j,1)], Vwalls
        }
}

//Nodos U
if(Rank != 0 && Rank != Procesos - 1){

                for(i = Ix; i < Fx + 1; i++){
                        for(j = 0; j < NY; j++){
                                PhiWallsMU[LU(i,j,0)] = ConvectiveScheme(MESH.MU[G
                        }
                }

}
else if(Rank == 0){

                for(j = 0; j < NY; j++){
                        //Parte izquierda
                        PhiWallsMU[LU(0,j,0)] = PhiLeft[j];
                        PhiWallsMU[LU(1,j,0)] = ConvectiveScheme(MESH.MU[GU(1,j,0)

                        for(i = Ix + 2; i < Fx + 1; i++){
                                PhiWallsMU[LU(i,j,0)] = ConvectiveScheme(MESH.MU[G
                        }
                }

}
else if(Rank == Procesos - 1){

                for(j = 0; j < NY; j++){
                        //Parte derecha
                        PhiWallsMU[LU(NX,j,0)] = PhiRight[j];
                        PhiWallsMU[LU(NX - 1,j,0)] = ConvectiveScheme(MESH.MU[GU(N

                        for(i = Ix; i < Fx - 1; i++){
                                PhiWallsMU[LU(i,j,0)] = ConvectiveScheme(MESH.MU[G
                        }
                }
}
```

```
}

//C lculo de las densidades en las paredes de lo vol menes de control
void Solver::Get_PhiWalls2(Mesher MESH, ParPro MPI1){
int i, j;

        //Comunicaci n de densidades entre los procesos
        MPI1.SendData(PhiLocalFut, Ix, Fx);
        MPI1.ReceiveData(PhiLocalFut, Ix, Fx);

        //Nodos R
        for(i = Ix; i < Fx; i++){
                //Parte abajo
                PhiWallsMR[LR(i,0,0)] = PhiDown[i - Ix];

                //Parte arriba
                PhiWallsMR[LR(i,NY,0)] = PhiUp[i - Ix];

                for(j = 1; j < NY; j++){
                        PhiWallsMR[LR(i,j,0)] = 0.50*(PhiLocalFut[LNH(i,j,0)] + PhiLocalFu
                                }
        }

        //Nodos U
        if(Rank != 0 && Rank != Procesos - 1){

                        for(i = Ix; i < Fx + 1; i++){
                                for(j = 0; j < NY; j++){
                                        PhiWallsMU[LU(i,j,0)] = 0.50*(PhiLocalFut[LNH(i,j,
1,j,0)]);
                                        }
                        }

        }
        else if(Rank == 0){

                        for(j = 0; j < NY; j++){
                                //Parte izquierda
                                PhiWallsMU[LU(0,j,0)] = PhiLeft[j];

                                for(i = Ix + 1; i < Fx + 1; i++){
                                        PhiWallsMU[LU(i,j,0)] = 0.50*(PhiLocalFut[LNH(i,j,
                                }
                        }

        }
        else if(Rank == Procesos - 1){

                        for(j = 0; j < NY; j++){
                                //Parte derecha
                                PhiWallsMU[LU(NX,j,0)] = PhiRight[j];

                                for(i = Ix; i < Fx; i++){
                                        PhiWallsMU[LU(i,j,0)] = 0.50*(PhiLocalFut[LNH(i,j,
                                }
                        }
```

```
        }

}

//C lculo del t rmino difusivo de la ecuaci n de Convecci n-Difusi n
void Solver::Get_Diffusive(Mesher MESH){
int i, j;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        Dw[LNH(i,j,0)] = MESH.SupMP[G(i,j,0)]/MESH.DeltasMU[GU(i,j,0)];
                        De[LNH(i,j,0)] = MESH.SupMP[G(i,j,1)]/MESH.DeltasMU[GU(i + 1,j,0)]
                        Ds[LNH(i,j,0)] = MESH.SupMP[G(i,j,2)]/MESH.DeltasMR[GR(i,j,1)];
                        Dn[LNH(i,j,0)] = MESH.SupMP[G(i,j,3)]/MESH.DeltasMR[GR(i,j+1,1)];
                }
        }
}

//C lculo del t rmino convectivo de la ecuaci n de Convecci n-Difusi n
void Solver::Get_Convective(Mesher MESH){
int i, j;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        Cw[LNH(i,j,0)] = RhoRef*UwallsMU[LU(i,j,0)]*MESH.SupMP[G(i,j,0)];
                        Ce[LNH(i,j,0)] = - RhoRef*UwallsMU[LU(i+1,j,0)]*MESH.SupMP[G(i,j,1
                        Cs[LNH(i,j,0)] = RhoRef*VwallsMR[LR(i,j,0)]*MESH.SupMP[G(i,j,2)];
                        Cn[LNH(i,j,0)] = - RhoRef*VwallsMR[LR(i,j+1,0)]*MESH.SupMP[G(i,j,3
                }
        }

}

//C lculo de los coeficientes de discretizaci n A
void Solver::Get_Coeficients(Mesher MESH){
int i, j;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        aw[LNH(i,j,0)] = Dw[LNH(i,j,0)];
                        ae[LNH(i,j,0)] = De[LNH(i,j,0)];
                        as[LNH(i,j,0)] = Ds[LNH(i,j,0)];
                        an[LNH(i,j,0)] = Dn[LNH(i,j,0)];

                        ap[LNH(i,j,0)] = aw[LNH(i,j,0)] + ae[LNH(i,j,0)] + as[LNH(i,j,0)] -

                }
        }

}

//C lculo de los coeficientes de discretizaci n Betas
void Solver::Get_BetaCoefficient(Mesher MESH){
int i, j;

        for(i = Ix; i < Fx; i++){
```

```
                for(j = 0; j < NY; j++){
                        bp[LNH(i,j,0)] = (RhoRef*MESH.VolMP[G(i,j,0)]*PhiLocalFut[LNH(i,j,
                }
        }

}

void Solver::Get_MaxDifGS(ParPro MPI1){
int i, j;
MaxDiffGS = 0.0;
MPI_Status ST;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        if(abs(PhiLocalFut[LNH(i,j,0)] - PhiLocalSup[LNH(i,j,0)]) >= MaxDi
                                MaxDiffGS = abs(PhiLocalFut[LNH(i,j,0)] - PhiLocalSup[LNH(
                        //      PhiLocalSup[LNH(i,j,0)] = PhiLocalFut[LNH(i,j,0)];
                        }
                }
        }

        MPI1.SendDataToZero(MaxDiffGS, PDiff);

        double Diff;
        if(Rank == 0){
                Diff = PDiff[0];
                for(i = 1; i < Procesos; i++){
                        if(PDiff[i] >= Diff){
                                Diff = PDiff[i];
                        }
                }
        }

        MPI1.SendDataToAll(Diff, MaxDiffGS);

}

//Resoluci n de las ecuaciones con Gauss-Seidel
void Solver::Get_StreamlinesFR(ParPro MPI1){
int i, j;
MaxDiffGS = 2.0*ConvergenciaGS;

        while(MaxDiffGS >= ConvergenciaGS){

                if(Rank != 0 && Rank != Procesos - 1){

                        for(i = Ix; i < Fx; i++){
                                //Parte abajo
                                PhiLocalFut[LNH(i,0,0)] = (aw[LNH(i,0,0)]*PhiLocalFut[LNH(

                                //Parte arriba
                                PhiLocalFut[LNH(i,NY-1,0)] = (aw[LNH(i,NY-1,0)]*PhiLocalFu

                                for(j = 1; j < NY - 1; j++){
                                        PhiLocalFut[LNH(i,j,0)] = (aw[LNH(i,j,0)]*PhiLocall
                                }
```

```
                }

        }
        else if(Rank == 0){

                for(i = Ix + 1; i < Fx; i++){
                        //Parte abajo
                        PhiLocalFut[LNH(i,0,0)] = (aw[LNH(i,0,0)]*PhiLocall

                        //Parte arriba
                        PhiLocalFut[LNH(i,NY-1,0)] = (aw[LNH(i,NY-1,0)]*Ph

                        for(j = 1; j < NY - 1; j++){
                                PhiLocalFut[LNH(i,j,0)] = (aw[LNH(i,j,0)]*
                        }
                }

                //Parte izquierda
                for(j = 1; j < NY - 1; j++){
                        PhiLocalFut[LNH(0,j,0)] = (aw[LNH(0,j,0)]*PhiLeft[
                }

                //Esquina abajo izquierda
                PhiLocalFut[LNH(0,0,0)] = (aw[LNH(0,0,0)]*PhiLeft[0] + ae[

                //Esquina arriba izquierda
                PhiLocalFut[LNH(0,NY-1,0)] = (aw[LNH(0,NY-1,0)]*PhiLeft[NY

        }
        else if(Rank == Procesos - 1){

                for(i = Ix; i < Fx - 1; i++){
                        //Parte abajo
                        PhiLocalFut[LNH(i,0,0)] = (aw[LNH(i,0,0)]*PhiLocalFut[LNH(

                        //Parte arriba
                        PhiLocalFut[LNH(i,NY-1,0)] = (aw[LNH(i,NY-1,0)]*PhiLocalFu

                        for(j = 1; j < NY - 1; j++){
                                PhiLocalFut[LNH(i,j,0)] = (aw[LNH(i,j,0)]*PhiLocall
                        }
                }

                //Parte derecha
                for(j = 1; j < NY - 1; j++){
                        PhiLocalFut[LNH(NX-1,j,0)] = (aw[LNH(NX-1,j,0)]*PhiLocalFu
                }

                //Esquina abajo derecha
                PhiLocalFut[LNH(NX-1,0,0)] = (aw[LNH(NX-1,0,0)]*PhiLocalFut[LNH(NX

                //Esquina arriba derecha
                PhiLocalFut[LNH(NX-1,NY-1,0)] = (aw[LNH(NX-1,NY-1,0)]*PhiLocalFut[

        }
```

```
                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                PhiLocalFut[LNH(i,j,0)] = PhiLocalSup[LNH(i,j,0)] + FR*(Ph
                        }
                }
                Get_MaxDifGS(MPI1);

                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                PhiLocalSup[LNH(i,j,0)] = PhiLocalFut[LNH(i,j,0)];
                        }
                }

                MPI1.SendData(PhiLocalFut, Ix, Fx);
                MPI1.ReceiveData(PhiLocalFut, Ix, Fx);

        }

}

//Resoluci n de las ecuaciones con Gauss-Seidel
void Solver::Get_Streamlines(ParPro MPI1){
int i, j;
MaxDiffGS = 2.0*ConvergenciaGS;
FR = 1.0;
        while(MaxDiffGS >= ConvergenciaGS){

                if(Rank != 0 && Rank != Procesos - 1){

                        for(i = Ix; i < Fx; i++){
                                //Parte abajo
                                PhiLocalFut[LNH(i,0,0)] = (aw[LNH(i,0,0)]*PhiLocalFut[LNH(

                                //Parte arriba
                                PhiLocalFut[LNH(i,NY-1,0)] = (aw[LNH(i,NY-1,0)]*PhiLocalFu

                                for(j = 1; j < NY - 1; j++){
                                        PhiLocalFut[LNH(i,j,0)] = (aw[LNH(i,j,0)]*PhiLocall
                                }
                        }

                }
                else if(Rank == 0){

                                for(i = Ix + 1; i < Fx; i++){
                                        //Parte abajo
                                        PhiLocalFut[LNH(i,0,0)] = (aw[LNH(i,0,0)]*PhiLocall

                                        //Parte arriba
                                        PhiLocalFut[LNH(i,NY-1,0)] = (aw[LNH(i,NY-1,0)]*Ph

                                        for(j = 1; j < NY - 1; j++){
                                                PhiLocalFut[LNH(i,j,0)] = (aw[LNH(i,j,0)]*
                                        }
                                }
```

```
                                //Parte izquierda
                                for(j = 1; j < NY - 1; j++){
                                        PhiLocalFut[LNH(0,j,0)] = (aw[LNH(0,j,0)]*PhiLeft[
                                }

                                //Esquina abajo izquierda
                                PhiLocalFut[LNH(0,0,0)] = (aw[LNH(0,0,0)]*PhiLeft[0] + ae[

                                //Esquina arriba izquierda
                                PhiLocalFut[LNH(0,NY-1,0)] = (aw[LNH(0,NY-1,0)]*PhiLeft[NY

                        }
                        else if(Rank == Procesos - 1){

                                for(i = Ix; i < Fx - 1; i++){
                                        //Parte abajo
                                        PhiLocalFut[LNH(i,0,0)] = (aw[LNH(i,0,0)]*PhiLocalFut[LNH(

                                        //Parte arriba
                                        PhiLocalFut[LNH(i,NY-1,0)] = (aw[LNH(i,NY-1,0)]*PhiLocalFu

                                        for(j = 1; j < NY - 1; j++){
                                                PhiLocalFut[LNH(i,j,0)] = (aw[LNH(i,j,0)]*PhiLocall
                                        }
                                }

                                //Parte derecha
                                for(j = 1; j < NY - 1; j++){
                                        PhiLocalFut[LNH(NX-1,j,0)] = (aw[LNH(NX-1,j,0)]*PhiLocalFu
                                }

                                //Esquina abajo derecha
                                PhiLocalFut[LNH(NX-1,0,0)] = (aw[LNH(NX-1,0,0)]*PhiLocalFut[LNH(NX

                                //Esquina arriba derecha
                                PhiLocalFut[LNH(NX-1,NY-1,0)] = (aw[LNH(NX-1,NY-1,0)]*PhiLocalFut[

                        }

                        Get_MaxDifGS(MPI1);

                        for(i = Ix; i < Fx; i++){
                                for(j = 0; j < NY; j++){
                                        PhiLocalSup[LNH(i,j,0)] = PhiLocalFut[LNH(i,j,0)];
                                }
                        }

                        MPI1.SendData(PhiLocalFut, Ix, Fx);
                        MPI1.ReceiveData(PhiLocalFut, Ix, Fx);

                }

}

//Calculo de los campos de velocidades
void Solver::Get_Velocities(Mesher MESH){
```

```
int i, j;

        if(Problema == 1){

                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                UlocalFut[LNH(i,j,0)] = Uref;
                                VlocalFut[LNH(i,j,0)] = 0.0;
                        }
                }

        }
        else if(Problema == 2){

                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                UlocalFut[LNH(i,j,0)] = 0.0;
                                VlocalFut[LNH(i,j,0)] = -Vref;
                        }
                }

        }
        else if(Problema == 3){

                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                UlocalFut[LNH(i,j,0)] = Uref*cos(AlphaRad);
                                VlocalFut[LNH(i,j,0)] = Vref*sin(AlphaRad);
                        }
                }

        }
        else if(Problema == 4){
                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                UlocalFut[LNH(i,j,0)] = 2.0*MESH.MP[G(i,j,1)]*(1.0
- pow(MESH.MP[G(i,j,0)] - 0.50*ChannelLength,2.0));
                                VlocalFut[LNH(i,j,0)] = -2.0*(MESH.MP[G(i,j,0)] - 0.50*Cha
- pow(MESH.MP[G(i,j,1)],2.0));
                        }
                }
        }

        //Nodos R
        if(Problema == 1){

                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY + 1; j++){
                                VwallsMR[LR(i,j,0)] = 0.0;
                        }
                }

        }
        else if(Problema == 2){

                for(i = Ix; i < Fx; i++){
```

```
                    for(j = 0; j < NY + 1; j++){
                        VwallsMR[LR(i,j,0)] = -Vref;
                    }
                }

        }
        else if(Problema == 3){

                for(i = Ix; i < Fx; i++){
                    for(j = 0; j < NY + 1; j++){
                        VwallsMR[LR(i,j,0)] = Vref*sin(AlphaRad);
                    }
                }

        }
        else if(Problema == 4){
                for(i = Ix; i < Fx; i++){
                    for(j = 0; j < NY + 1; j++){
                        VwallsMR[LR(i,j,0)] = -2.0*(MESH.MR[GR(i,j,0)] - 0.50*Chan
                    }
                }
        }

        //Nodos U
        if(Problema == 1){

                for(i = Ix; i < Fx + 1; i++){
                    for(j = 0; j < NY; j++){
                        UwallsMU[LU(i,j,0)] = Uref;
                    }
                }

        }
        else if(Problema == 2){

                for(i = Ix; i < Fx + 1; i++){
                    for(j = 0; j < NY; j++){
                        UwallsMU[LU(i,j,0)] = 0.0;
                    }
                }

        }
        else if(Problema == 3){

                for(i = Ix; i < Fx + 1; i++){
                    for(j = 0; j < NY; j++){
                        UwallsMU[LU(i,j,0)] = Uref*cos(AlphaRad);
                    }
                }

        }
        else if(Problema == 4){
                for(i = Ix; i < Fx + 1; i++){
                    for(j = 0; j < NY; j++){
                        UwallsMU[LU(i,j,0)] = 2.0*MESH.MU[GU(i,j,1)]*(1.0 - pow(ME
                    }
```

```
                }
        }

}

//C lculo de la diferencia de resultados entre Steps
void Solver::Get_Stop(ParPro MPI1){
int i, j;
MaxDiffGlobal = 0.0;

        if(Rank == 0){
                for(i = 0; i < NX; i++){
                        for(j = 0; j < NY; j++){
                                if(abs((PhiGlobalFut[G(i,j,0)] - PhiGlobalPres[G(i,j,0)])/(
                                        MaxDiffGlobal = abs((PhiGlobalFut[G(i,j,0)] - PhiG
                                }
                        }
                }
        }

        MPI1.SendDataToAll(MaxDiffGlobal, MaxDiffGlobal);

}

void Solver::UpdatePropertiesFields(){
int i, j;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        PhiLocalPres[LNH(i,j,0)] = PhiLocalFut[LNH(i,j,0)];
                }
        }
}

void Solver::Get_NumericalResults(Mesher MESH, string Carpeta, string Problema){
int i,j;

ofstream file;
string FileName;
stringstream InitialNameMP;
string FinalNameMP;
int RHO = RhoRef;
string txt = ".txt";

        InitialNameMP<<DIRECTORIO<<Carpeta<<Problema<<"_"<<EsquemaLargo<<"_"<<RHO<<txt;

        FinalNameMP = InitialNameMP.str();
        file.open(FinalNameMP.c_str());

                file<<0.0<<"\t"<<2.000<<"\t"<<endl;

                for(i = 0; i < NX; i++){
                        if(MESH.MR[GR(i,0,0)] - 0.50*ChannelLength >= 0.0){
                                file<<(MESH.MR[GR(i,0,0)] - 0.50*ChannelLength)<<"\t"<<Phi
                        }
        }
```

```
                file <<1.0<<"\t"<<0.0<<"\t"<<endl;

        file.close();

}

void Solver::Get_AnalyticalResults(){
int i;

        for(i = 0; i < 11; i++){
                CoordenadasAnalitico[i] = 0.1*i;
        }

        if(RhoRef == 10){
                PhiAnalitico[0] = 1.989;
                PhiAnalitico[1] = 1.402;
                PhiAnalitico[2] = 1.146;
                PhiAnalitico[3] = 0.946;
                PhiAnalitico[4] = 0.775;
                PhiAnalitico[5] = 0.621;
                PhiAnalitico[6] = 0.480;
                PhiAnalitico[7] = 0.349;
                PhiAnalitico[8] = 0.227;
                PhiAnalitico[9] = 0.111;
                PhiAnalitico[10] = 0.00;
        }
        else if(RhoRef == 1000){
                PhiAnalitico[0] = 2.0000;
                PhiAnalitico[1] = 1.9997;
                PhiAnalitico[2] = 1.9990;
                PhiAnalitico[3] = 1.9850;
                PhiAnalitico[4] = 1.8410;
                PhiAnalitico[5] = 0.9510;
                PhiAnalitico[6] = 0.1540;
                PhiAnalitico[7] = 0.0000;
                PhiAnalitico[8] = 0.0000;
                PhiAnalitico[10] = 0.0000;
        }
        else if(RhoRef == 1000000){
                PhiAnalitico[0] = 2.000;
                PhiAnalitico[1] = 2.000;
                PhiAnalitico[2] = 2.000;
                PhiAnalitico[3] = 1.999;
                PhiAnalitico[4] = 1.964;
                PhiAnalitico[5] = 1.000;
                PhiAnalitico[6] = 0.036;
                PhiAnalitico[7] = 0.000;
                PhiAnalitico[8] = 0.000;
                PhiAnalitico[10] = 0.000;
        }

}

void Solver::RelativeError(Mesher MESH, string Carpeta, string Problema){
int i,j;
double ErrorRelativoTotal = 0.0;
```

```
double MediaErrorRelativo;


        //B squeda de los puntos m s cercanos a los anal ticos
        for(j = 0; j < 9; j++){
                MinAbajo[j] = ChannelLength;
                MinArriba[j] = ChannelLength;
                for(i = 0.40*NX; i < NX; i++){
                        if((MESH.MP[G(i,0,0)] - 0.50*ChannelLength) - CoordenadasAnalitico
                                if(abs((MESH.MP[G(i,0,0)] - 0.50*ChannelLength) - Coordena
                                        MinAbajo[j] = MESH.MP[G(i,0,0)];
                                        PhiAbajo[j] = PhiGlobalFut[G(i,0,0)];
                                }
                        }
                        if((MESH.MP[G(i,0,0)] - 0.50*ChannelLength) - CoordenadasAnalitico
                                MinArriba[j] = MESH.MP[G(i,0,0)];
                                PhiArriba[j] = PhiGlobalFut[G(i,0,0)];
                                break;

                        }

                }

        }


        //C lculo de la propiedad en esos puntos
        for(j = 1; j < 9; j++){
                PhiReal[j] = PhiAbajo[j] + ((PhiArriba[j] - PhiAbajo[j])/(MinArriba[j] - M
        }

        cout<<"MinAbajo \t MinArriba"<<endl;
        for(i = 1; i < 9; i++){
                cout<<MinAbajo[i]<<"\t"<<MinArriba[i]<<endl;
        }
        cout<<"PhiAbajo \t PhiArriba"<<endl;
        for(i = 1; i < 9; i++){
                cout<<PhiAbajo[i]<<"\t"<<PhiArriba[i]<<endl;
        }
        cout<<"PhiAnalitico \t PhiReal"<<endl;
        for(i = 1; i < 9; i++){
                cout<<PhiAnalitico[i]<<"\t"<<PhiReal[i]<<endl;
        }
        for(i = 1; i < 9; i++){
                if(PhiAnalitico[i] == 0.0){
                        ErrorRelativoTotal += abs(PhiAnalitico[i] - PhiReal[i])/(PhiAnalit
                }
                else{
                        ErrorRelativoTotal += abs(PhiAnalitico[i] - PhiReal[i])/(PhiAnalit
                }

        }

        MediaErrorRelativo = ErrorRelativoTotal/8.0;
```

```
int RHO = RhoRef;

        char Directorio[200];
        sprintf(Directorio,"/home/sergiogus/Desktop/ComputationalEngineering/ConvectionDif

        FILE *fp1;
                fp1 = fopen(Directorio,"a");
                        fprintf(fp1,"%d \t %d \t %d \t %f \n", NX, NY, NX*NY, MediaErrorRe

        fclose(fp1);

}

//Pasar los resultados a un archivo VTK en 2D
void Solver::EscalarVTK2D(Mesher MESH, string Carpeta, string Variable, string NombreFile,
int i, j;

        ofstream file;
    stringstream InitialName;
    string FinalName;

        InitialName<<DIRECTORIO<<Carpeta<<NombreFile<<".vtk";

        FinalName = InitialName.str();
    file.open(FinalName.c_str());

        file<<"# vtk DataFile Version 2.0"<<endl;
        file<<Variable<<endl;
        file<<"ASCII"<<endl;
        file<<endl;
        file<<"DATASET STRUCTURED_GRID"<<endl;
        file<<"DIMENSIONS"<<" "<<Na<<" "<<Nr<<" "<<1<<endl;
        file<<endl;
        file<<"POINTS"<<" "<<Na*Nr<<" "<<"double"<<endl;

                for(j = 0; j < Nr; j++){
                        for(i = 0; i < Na; i++){
                                file<<MC[G(i,j,0)]<<" "<<MC[G(i,j,1)]<<" "<<0.0<<endl;
                        }
                }

        file<<endl;
                file<<"POINT_DATA"<<" "<<Na*Nr<<endl;
        file<<"SCALARS "<<Variable<<" double"<<endl;
        file<<"LOOKUP_TABLE"<<" "<<Variable<<endl;
        file<<endl;

       for(j = 0; j < Nr; j++){
                        for(i = 0; i < Na; i++){
                                file<<GlobalField[G(i,j,0)]<<" ";
                    }
            }

    file.close();

}
```

```
//Pasar a un .vtk los resultados de campos vectoriales
void Solver::VectorialVTK2D(Mesher MESH, string Carpeta, string Variable, string NombreFile
int i, j;

    ofstream file;
    stringstream InitialName;
    string FinalName;

        InitialName<<DIRECTORIO<<Carpeta<<NombreFile<<".vtk";

        FinalName = InitialName.str();
    file.open(FinalName.c_str());

        file<<"# vtk DataFile Version 2.0"<<endl;
        file<<Variable<<endl;
        file<<"ASCII"<<endl;
        file<<endl;
        file<<"DATASET STRUCTURED_GRID"<<endl;
        file<<"DIMENSIONS"<<" "<<Na<<" "<<Nr<<" "<<1<<endl;
        file<<endl;
        file<<"POINTS"<<" "<<Na*Nr<<" "<<"double"<<endl;

                for(j = 0; j < Nr; j++){
                        for(i = 0; i < Na; i++){
                                file<<MC[G(i,j,0)]<<" "<<MC[G(i,j,1)]<<" "<<0.0<<endl;
                        }
                }

        file<<endl;
        file<<"POINT_DATA"<<" "<<Na*Nr<<endl;
        file<<"VECTORS "<<Variable<<" double"<<endl;
        file<<endl;

        for(j = 0; j < Nr; j++){
                        for(i = 0; i < Na; i++){
                        file<<GlobalField1[G(i,j,0)]<<" "<<GlobalField2[G(i,j,0)]<<" "<
            }
        }

    file.close();

}

//Ejecuci n de todos los procesos del solver
void Solver::ExecuteSolver(Memory M1, ReadData R1, ParPro MPI1, Mesher MESH, int i){
int Step = 0;
char FileName[300];
double Time = 0.0;

MaxDiffGlobal = 2.0*ConvergenciaGlobal;

 auto start = std::chrono::high_resolution_clock::now();

// Portion of code to be timed
```

```
        AllocateMatrix(M1);
        InitializeFields(MESH);
        Get_Velocities(MESH);
        if(Rank == 0){ Get_AnalyticalResults(); }

        //Pasar todas las matrices al ZERO
/*      //Matrices
        MPI1.SendMatrixToZero(UlocalFut, UglobalFut, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(PhiLocalFut, PhiGlobalFut, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(VlocalFut, VglobalFut, NX, NY, Procesos, Ix, Fx);

        if(Rank == 0){
                sprintf(FileName, "MapaStreamFunction_Step_%d", Step);
                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "StreamFunctions"

                sprintf(FileName, "MapaVelocidades_Step_%d", Step);
                VectorialVTK2D(MESH, "ParaviewResults/PropertiesResults/", "Velocidades",
        }
*/


        while(MaxDiffGlobal >= ConvergenciaGlobal){

                Step++;

                UpdateBoundaryConditions(MESH);

                Get_StepTime(MESH, MPI1);
                Time += DeltaT;

                Get_PhiWalls(MESH, MPI1);

        /*      if(Step == 2 && Rank == 0){
                        for(j = NY-1; j>= 0; j--){
                                for(i = Ix; i < Fx; i++){
                                        cout<<1<<", ";
                                }
                                cout<<endl;
                        }
                }*/

                Get_Diffusive(MESH);
                Get_Convective(MESH);

                Get_Coeficients(MESH); //C lculo de los coeficientes de discretizaci n
                Get_BetaCoefficient(MESH);

                Get_Streamlines(MPI1);
                //Get_StreamlinesFR(MPI1);


                if(Step%1000 == 0){

                        //Pasar todas las matrices al ZERO
```

```
                        //Matrices Step Presente
                        MPI1.SendMatrixToZero(PhiLocalPres, PhiGlobalPres, NX, NY, Procesos
                        MPI1.SendMatrixToZero(UlocalPres, UglobalPres, NX, NY, Procesos, Ix
                        MPI1.SendMatrixToZero(VlocalPres, VglobalPres, NX, NY, Procesos, Ix

                        //Matrices Step Futuro
                        MPI1.SendMatrixToZero(UlocalFut, UglobalFut, NX, NY, Procesos, Ix,
                        MPI1.SendMatrixToZero(PhiLocalFut, PhiGlobalFut, NX, NY, Procesos,
                        MPI1.SendMatrixToZero(VlocalFut, VglobalFut, NX, NY, Procesos, Ix,

                        Get_Stop(MPI1);

                        if(Rank == 0){
                                cout<<"Simulation: "<<i<<", Step: "<<Step<<", Total time: "

                                if(Problema == 4){
                                        Get_NumericalResults(MESH, "GnuPlotResults/Numeric
                                }

                                sprintf(FileName, "MapaStreamFunction_Step_%d", Step);
                                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "

                                sprintf(FileName, "MapaVelocidades_Step_%d", Step);
                                VectorialVTK2D(MESH, "ParaviewResults/PropertiesResults/",
                        }
                }

                UpdatePropertiesFields();

        }

        if(Rank == 0){
                RelativeError(MESH, "NumericalResults/", "SmithHutton");
        }
        // Record end time
        auto finish = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> elapsed = finish - start;
        std::cout << "Elapsed time: " << elapsed.count() << " s\n";
        if(Rank == 0){
                int RHO = RhoRef;
                char Directorio[200];
                sprintf(Directorio,"/home/sergiogus/Desktop/ComputationalEngineering/Conve

                FILE *fp1;
                        fp1 = fopen(Directorio,"a");
                                fprintf(fp1,"%f \t %f \n", FR, elapsed.count());

                fclose(fp1);
        }


}
```