PANACEA

Predicting **a**nd mo**n**itoring the long-term beh**a**vior of **C**O_2 inj**e**cted in deep geologi**c**al form**a**tions

Contract #282290

Deliverable Number	**D7.2**
Title	**Parallel code comparison**
Work-Package	**WP7: Prediction and validation of the long term behavior of the stored CO_2**

Lead Participant EWRE
Contributors EWRE

Version: **1**
Revision level: **01**
Due Date: 12
Reviewed by: Jacob Bensabat
Status: Final Draft
Dissemination level PU

Executive summary

Large, regional, scale simulations of the fate of the stored CO_2 will require the construction of large computational models. Standard software for the simulation of such cases may require large computing times and may make their use impractical for these cases. Therefore, models capable of exploiting the available computing resources may be needed. These are simulators with parallel processing capabilities. In this deliverable, we review two parallel codes for the simulation of CO2 injection in a reservoir, which are available as public domain. We show that the codes scale almost perfectly, i.e., an almost linear speed-up is obtained as function of the number of available processors. We demonstrate the use of parallel codes on a small cluster of 24 processors and describe the hardware that is suggested for the use.

Keywords	

Table of Contents

1. Introduction

The purpose of this deliverable is to assess the performance of parallel codes for the simulation of the long fate of the stored CO_2. The assessment of reservoirs for CO_2 storage will require the construction of regional, large, scale models, which could comprise at least hundreds of thousands of cells and often millions of cells, with a relatively large number of variables if chemical processes are to be taken into account. On standard, sequential simulators, this would mean impractical computing times, even when disregarding chemical reactions. Therefore, codes with parallel processing capabilities, high performance computing or HPC, seem to be more suitable for the construction and simulation of large scale models. Together with the parallel codes, one needs also the availability of suitable computing resources, i.e. the availability of enough computational capacity at affordable cost.

We have reviewed two, public domain, simulation codes having intrinsic parallel computing capabilities: LANL (Los Alamos National Laboratories, USA) **PFLOTRAN** and the University of Stuttgart **DuMuX**. Both codes have modules capable of simulating the CO_2 reservoir.

We demonstrate the use of these codes in the frame of a parallel computing framework as well as we outline, the computing resources needed for the simulations.

2. Hardware for high performance computations (HPC)

In this application we have used in house parallel computation resources. These include:

- 2 Dell Precision R5550 Workstations (2 nodes).
- Each node contains 2 Intel Xeon CPU X5690 with 6 Cores;
- 24GB RAM per node;
- Each core runs at 3.46 GHz;
- The workstations are connected via Mellanox InfiniBand high-performance adapters, which deliver up to 40Gb/s transfer rates.

This structure is expandable to an unlimited number of nodes (workstations). Each node can host up to GP_GPU (General Purpose Graphical Processing Units). However, the incorporation of these and the full exploitation of their computing capabilities would require substantial software modifications, which is at best problematic as it would require massive modifications of the codes. Lately, Intel has announced the XEON PHI Coprocessor, a computing unit based on standard parallel processing with up to 60 processor per card. A workstation of the type we have can host up to four XEON Phi coprocessor cards (two XEON Phis coprocessors per XEON CPU). This new technology is currently under testing.

3. Demonstration of a parallel application for a regional model

As a first step we shall demonstrate the construction and performance of a parallel application for the case of a regional model, the Heletz reservoir. The steps carried out in the construction of the regional model include:

- Creating a 3D model of the Heletz injection site;
- Simulating a Multiphase CO_2 injection;

- Trying different quantities of injected gas and tracking the expansion of CO2's footprint over time.
- Calculating pressure, temperature and saturations in the reservoir.
- Comparing performance when using structured vs. unstructured grids.

3.1 Tools

3.1.1 Software

- **PFLOTRAN** :

  A software which simulates Multiscale-Multiphase-Multicomponent Subsurface Reactive Flows Using Advanced Computing.

  It was developed in Los Alamos National Laboratory (LANL), one is free to request access to the source, and it is currently under a LGPL license (Lesser GNU Public License).

  Its website is in: http://ees.lanl.gov/pflotran/ , you might need to contact the developers directly by email to obtain an access to the download site.

  PFLOTRAN is written in Fortran 90, and it is designed to run on massively parallel computing architectures, as well as single workstations/laptops.

  Besides the original code written for PFLOTRAN, it relies on certain libraries which are also free to download:

  - PETSc – Portable Extensible Toolkit for Scientific Computation, it's a suite of functions and data structures, for parallelized scientific applications, modeled by partial differential equations.
    It is the most important library PFLOTRAN relies on, and it is being used to achieve parallelization through "domain decomposition".

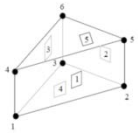
  - ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering.
    This library is an MPI-based (parallel) library that implements algorithms for partitioning *unstructured* grids. We wouldn't be able to simulate injection in the Heletz Unstructured model without it.

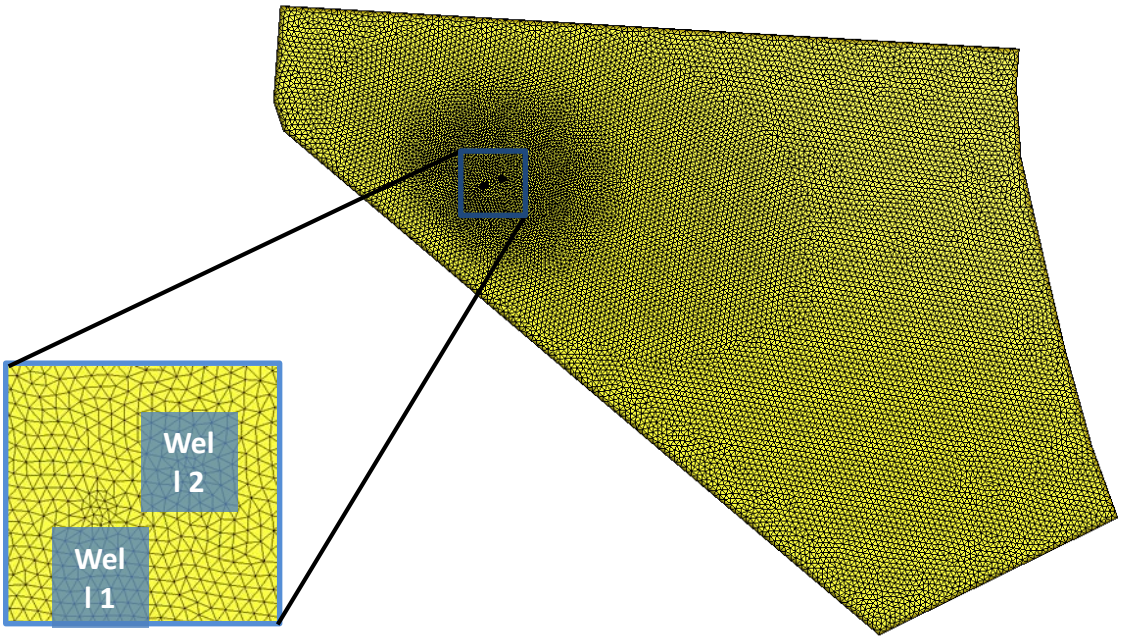
  - HDF5 – It is the name of a data-structure, and a file format, which is used for storing high volume data.
    It supports unlimited data types, and it is has a flexible and efficient I/O operation , and handling of complex and large chunks of data.
    This library basically enables us to create one file which contains the coordinates/elements of our unstructured grid, comprised of millions of cells, and save the pressure/temperature/concentrations/etc. for each cell.

- MPICH2 – A library which *implements* the MPI "standard".
  MPI (Message Passing Interface) is a system for passing information in parallel programs. MPICH2 enables PFLOTRAN to divide the calculations, I/O operations, and more, to all available CPUs .

- **VASP** – Visual Analysis Simulation Platform. We use VASP for Pre and Post Processing. With VASP we built Heletz's unstructured grid, and we will analyze the results (given in HDF format by PFLOTRAN) in VASP as well.

3.1.2 Grid

- Unstructured grid, based on Heletz topography.
- Built with VASP.
- Has 216,000 Cells.
- Cell size ranges between 0.2 to 2700 cubic meters.
- Cells have the shape of a Wedge : 
- Cells near the injection points ('wells'), were set to be smaller and more refined, to improve the quality of results :



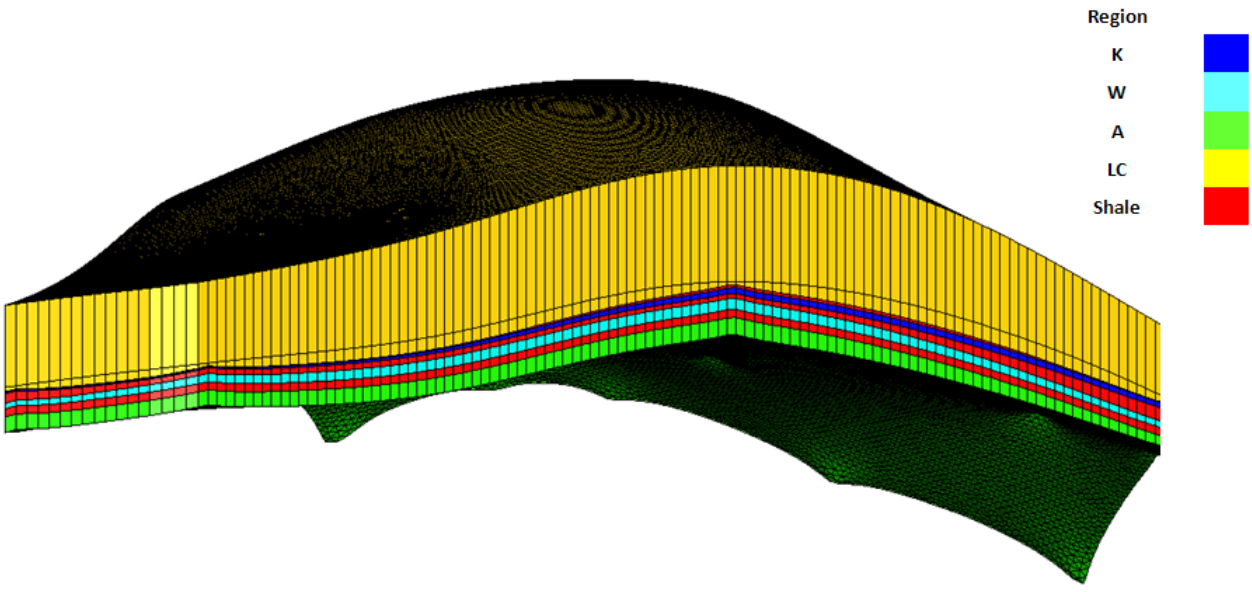
3.1.3 Materials and Regions in the Grid

The grid comprises 8 layers , in the following order (from top to bottom):

Caprock
LC-11
Shale
K
Shale
W
Shale
A

The Caprock and LC-11 are united as one region, they contain the same material in our model. The region "Shale" is divided to 3 sectors, as seen in the table above

The Topography of our model is illustrated below :



- The Materials can be characterized by the following table:

3.1.4 Simulations Results

Layer	LC-11	Shale	Sandstone K	Sandstone W	Sandstone A
Porosity	5%	5%	15%	15%	15%
Tortuosity	1	1	1	1	1
Rock Density ($K_g m^{-3}$)	2,840	2,840	2,840	2,840	2,840
Permeability (m^2)	$5\ 10^{-13}$	10^{-15}	$5\ 10^{-13}$	$5\ 10^{-13}$	$5\ 10^{-13}$
Specific Heat	1000	1000	1000	1000	1000

The table below summarizes the results of the various parallel simulations.

Permeability	Shale : 10^{-15} Sandstone : $5*10^{-13}$	Shale : 10^{-17} Sandstone : 10^{-13}	Shale : 10^{-17} Sandstone : 10^{-13}
Mode	MPHASE	FLASH2	FLASH2
Wells	1	2 * Injection Rate divided relative to the layer's height	2 * Injection Rate divided relative to the layer's height
Total Injection Rate [kg/sec]	1	1	1
Total Injection Quantity [Tons]	18,000	15,000	100,000
Simulation Time [Years]	10	10	15
Run Time	10.8 Hours (2 Intel Xeons)	9.5 Hours (2 Intel Xeons) 16 Hours (1 Intel Xeon)	28 Hours (2 Intel Xeons)

4. A General Overview of DuMuX as a tool to Simulate Multiphase CO_2 Injection

4.1 Overview

DuMuX is a generic framework for the simulation of multiphase fluid flow and transport processes in porous media using continuum mechanical approaches.

It is built as a set of C++ Templates, developed at the University of Stuttgart. The developers have implemented their own "property system" for customizing classes and structures, and creating a class hierarchy.

It heavily relies on C++ Template Programming and Polymorphism and from that point of view the code is written using state of the art computing languages.

DuMuX was developed by numerous students, who each contributed a part to DuMuX during their PhD studies, and then left it to others. So it can qualified as a research/academic software.

Currently, Dr. Bernd Flemisch (Stuttgart University) is responsible for DUMUX. The code is freely available.

You can obtain the latest DuMuX build (or the source code), through their website:
http://www.dumux.org/download.php

A guide for DuMuX is also available at:
http://www.dumux.org/documents/dumux-handbook-2.3.pdf

Currently, you can only build it (with all its features) in Linux, We've tested building it on "Ubuntu" operating system.

DuMuX is built on top of a few libraries:

- DUNE - the Distributed and Unified Numeric Environment.
  It's a 'Modular toolbox' for solving partial differential equations (PDEs) with grid-based methods.
  DuMuX is built on top of DUNE, and heavily relies on it.

- ALUGrid - The ALUGrid Library provides both Hexahedral and Tetrahedral grids which can be used for parallel computations.
  This library is under the framework of DUNE.

4.2 CO_2 Injection Problem

In DuMuX, there are a few available "examples" / "tests" that can potentially be transformed into a CO_2 injection problem. There are 2 examples which were specifically built to simulate CO_2 injection, but they are not necessarily the most suitable ones. In this document, we essentially describe how to use DuMuX in order to implement a CO_2 injection problem, and how to achieve a basic understanding of DuMuX. We tested the examples found at /dumux/test/boxmodels. The example called 'co2' was tested first. This example was written by default to use unstructured grids. Using the ALUGrid library, and transforming

this example into a structured grid (which uses a library called YASPgrid), was causing too many errors at the time. DuMuX doesn't have a built-in Mesh Generator, so basically, the user is expected to build its own customized mesh generator whenever it's needed. The code is being reviewed at this time and in the future it might be possible to easily use a structured grid in this example. The best choice for a starting-point, was taken from an example named '2p2cni', and transforming it into a multiphase CO_2 injection problem.

2p2cni is an example which simulates a Non-Isothermal, 2-Phase, 2-Components, injection of a gas (which by default is air), into a fully water-saturated medium. As any other simulation in DuMuX, 2p2cni can be viewed and edited *only by editing the code itself,* which can be found in the directory /dumux/test/boxmodels/2p2cni.

The most important alteration was of course injecting CO2 instead of air. Afterwards, we edited all the different simulation details and the model's grid. DuMux was compiled under the Linux operating system, while it failed under any version of Windows. On the other hand we were able to compile and run PFLOTRAN both under LINUX and Windows.

The actual editing of the 2p2cni problem, to suit our needs for the simulation, are described in Appendix 6.1.

4.3 Introducing the Tools for Development under Linux

Working with DuMuX requires a new set of tools, if you're used to working with windows. We introduce here 2 tools:

- Eclipse - A Multi-language Software Development Environment, allows to edit code in a comfortable workspace.

  We're editing DuMuX's code in Eclipse.

  You can download it at:
  http://www.eclipse.org/downloads/?osType=linux

- Paraview – An open-source, Multi-Platform application, designed to visualize data sets of size varying from small to very large.

  Dumux outputs the results into a file which can be visualized / plotted in Paraview.

  You can download it at:
  http://www.paraview.org/paraview/resources/software.php

4.4 Using Eclipse to edit a DuMuX problem more easily

To create a project in Eclipse, suitable for working with DuMuX, do the following steps:

- File -> New -> Makefile Project with Existing Code -> Choose;
- Languages – C , C++;
- Toolchain for Indexer Settings – Linux GCC;
- Click 'Finish'.
- Next, specify the name of the final executable to be created,

- Which in general should have the same name as the .cc file in the test folder we work with :
- In Eclipse, locate the 'Folder Tree' on the right side.
- Go to the folder of the test you wish to execute (for example, DUMUX/test/boxmodels/2p2cni).
- Right click on the test folder name, and press 'New'.
- Write in 'Target name': test_2p2cni.
- For the test '2p2cni'.
- Press 'Ok'.

4.5 Additional Properties and Information

Materials are specified in the "Spatial Parameters" file. The file name is "…spatialparams.hh". Our example, '2p2cni', uses a CO2 table to obtain certain information (like the pressure, if we already know the temperature). It is not complete, and so the certain values in the CO_2 super-critical phase could not be used. In DuMuX, as in PFLOTRAN, true 2D models do not exist, what we have instead, is a 3D model which has a height of 1-meter in the Z-Axis.

A 3D model which has a height of 20 meters was too slow for the simulation, even with MPI running on 8 Cores, so we had to use a 2D model. PFLOTRAN can run a simulation with the 3D model in a reasonable amount of time.

- In order to compare the footprints that DuMuX and PFLOTRAN produced, we had to divide the injection rate in the DuMuX simulation by 20 (since the layer's height in the Z-axis got 20 times smaller);
- Q (2D) = Q (3D)/20.
- Changing a simulation which currently runs on a 3D unstructured grid, to one that runs on a 2D structured grid, is a complicated process in DuMuX which requires many changes in more than one file.

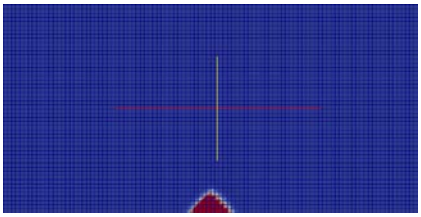
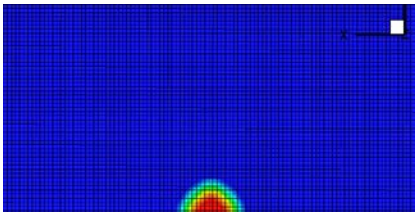
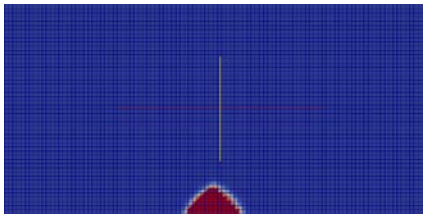
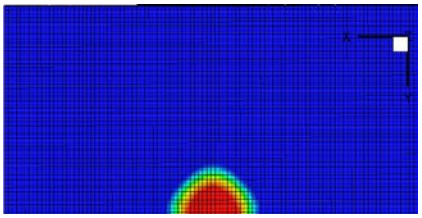
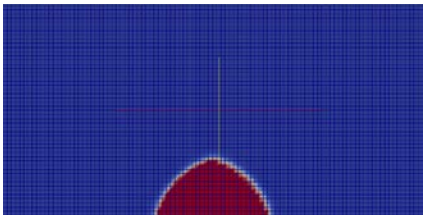
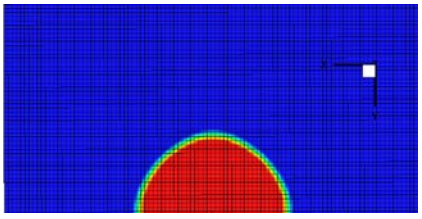
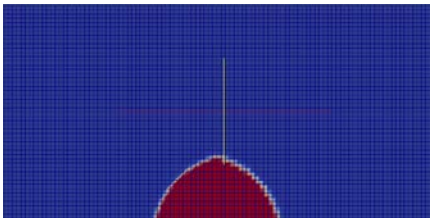
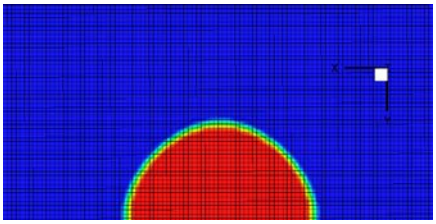
4.6 Comparing DuMuX and PFLOTRAN for the ability to simulate a simple CO2 injection test

In this section, I'll describe what is currently possible or impossible, using DuMuX and PFLOTRAN. Both programs are still under development but in different stages.

Criteria	DuMuX	PFLOTRAN
Direct switch from unstructured to structured grid and vice-versa.	No (Modifying a simulation which uses unstructured grids (ALUGrid) to use structured grids (YASPGrid) is a long process)	Yes
Injecting from any point in the grid, without heavy modification of the code	No	Yes
Changing initial and boundary conditions of the entire layer : porosity, permeability, temperature and pressure	Yes	Yes
Running a simulation with non-supercritical CO2	Yes	No
Running a simulation with supercritical CO2	No (Currently missing a supercritical CO2 table)	Yes
Directly modifying the simulation's mode (isothermal/non-isothermal and heat transfer, available phases : gas/liquid/supercritical , and more)	No (Requires major changes in the simulation's code)	Yes
Most of the properties in a simulation can be defined after compilation	No	Yes
Default support for defining regions and boundaries in the model	No	Yes
Running Parallel	Yes	Yes

4.7 Testing a Simple Simulation

- We've injected 2.5 Million tons of CO_2 over 5 years.
- We've allowed a relaxation period of 45 years.
- We used a structured grid, of 5km x 5km x 20m.
- The entire layer is made of homogenous sandstone.
- These are the results we got from DuMuX and PFLOTRAN:

Results of CO2 (non-liquid) Distribution at Different Times	DuMuX	PFLOTRAN
0.5 Years		
1 Year		
5 Years		
50 Years		

5. Conclusions

In this work we have carried a preliminary evaluation of two CO2 simulators with HPC capabilities. We have restricted the search to publicly available codes and therefore we have not tested the parallel version of the well-known TOUGH2 software, which requires substantial investment. The two codes which have been considered, DuMux (University of Stuttgart) and PFLOTRAN (LANL laboratories, USA) provided good parallel performance (almost linear speed-up with increasing number of cores).

PFLOTRAN is written in the FORTRAN 90 language and is well structured. On the other hand DuMux is written in c++, which is a more flexible language. We were able to compile PFLOTRAN both under the Linux and Windows operating systems, while we failed to compile DuMux under Windows and succeeded with Linux.

The setting of a simulation under PFLOTRAN is straightforward, as there is a clear separation between the code and the input data. On the other hand setting a simulation under DuMux requires intervention in (changing) the code and therefore a clear understanding of the code structure and the programming language. This can be seen as a major drawback, as the need to program the code for each case may result in many errors and is probably time consuming.

Overall, it seems that PFLOTRAN is a more mature and ready to use simulator with very good HPC characteristics.

The simulations we have carried out show substantial differences between the outputs of the two codes, even for this very simple case. We shall continue investigating these features together with the evaluation of the new co-processor XEON PHI, which we are now integrating towards the use for simulation.

6. Appendix A

6.1 Converting '2p2cni' into a CO_2 Injection Problem

The following sample code describes how to specify key-properties in the '2p2cni' example. Each case has its own problem and spatial parameters files, which describe the simulation properties and the grid/materials and uses one of the available Fluid Systems.

If one wishes to edit settings like the fluid diffusion coefficient, you need to change the fluid system file, but the problem is that if another example will use the same fluid system, it will also 'see' the changed file. The solution is to create a copy of the fluid system file, put it in the same folder as the old one, and rename it to a unique name that only your example will then use.

The Actual Changes that were made to the original simulation (waterairproblem.hh) are:

Fluid System – we chose a co2/brine based fluid system:

```
#include <dumux/material/fluidsystems/brineco2fluidsystem.hh>
```

Provides the program with the tabulated values of CO2:

```
#include "heterogeneousco2tables.hh"
```

Defining the problem class:

```
template <class TypeTag >
class WaterAirProblem : public PorousMediaBoxProblem<TypeTag>
{ ... }
```

Inside the class's definition, we find the class's constructor, where we will define a lot of important parameters:

```
maxDepth_ = 1650.0; // [m], The depth of our 2D mode
```

Defining the injection rate:

```
injectionRate_ = 10 * 5 * 1.0e3/3600 * 0.5 /20; // [Kg/s]
```

Defining various parameters of the simulation:

```
injectionPressure_ = 150*1e5;
injectionTemperature_ = 273.15+60;
injectionTime_ = 5 /*year*/ * 365 /*days*/ * 86400/*sec*/;
```

Setting the layer's temperature:

```
   Scalar temperature() const
   {
        return 273.15 + 60; // -> 10°C
   };
```

When DuMuX builds the grid, it sets the initial pressure/temperature/flux properties for each cell separately. In order to address the current cell which the DuMuX grid builder is working with a variable called "GlobalPos" was made. GlobalPos – specifies the current position in the grid.

Setting the injection area:

- In our simulation we want to inject from the lower boundary of the model. We want the middle point of the boundary to be the injection well. So first, we specify that the entire lower boundary has a certain flux moving through it, by setting the boundary as Neumann type.

```
void boundaryTypes(BoundaryTypes &values, const Vertex &vertex) const
{
   const GlobalPosition globalPos = vertex.geometry().center();
   values.setAllDirichlet();
   if (onLowerBoundary_(globalPos) )
   {
      values.setAllNeumann();
   }
   #if !ISOTHERMAL
       values.setDirichlet(temperatureIdx, energyEqIdx);
   #endif
}
```

The injection well is set in the following code (notice the implementation of the well is not an easy task:

```
   void neumann(PrimaryVariables &values,
             const Element &element,
             const FVElementGeometry &fvGeometry,
             const Intersection &is,
             const int scvIdx,
             const int boundaryFaceIdx) const
{
    const GlobalPosition &globalPos = element.geometry().corner(scvIdx);
    values = 0;
    Scalar simTime = this->timeManager().time();
    // negative values for injection
   if (onLowerBoundary_(globalPos) && (globalPos[0] > 2500 - 1) && (globalPos[0] < 2500 +
   1) && globalPos[1] < eps_)
    {
          if (simTime<injectionTime_)
          {
          values[contiNEqIdx] = -injectionRate_ / 100 ;
   values[energyEqIdx] = -injectionRate_*CO2::
          gasEnthalpy(injectionTemperature_, injectionPressure_); // W/(m^2)
          }
          else
                values = 0;
    }
}
```

Setting the initial conditions:

```
// internal method for the initial condition (reused for the
// dirichlet conditions!)
void initial_(PrimaryVariables &values, const GlobalPosition &globalPos) const
{
    Scalar densityW = 1000.0;
    values[pressureIdx] = 150*1e5;
    values[switchIdx] = 0.0;
```

```
        #if !ISOTHERMAL
                values[temperatureIdx] = 273.15 + 60;    // [K]
        #endif
    }
```

Custom functions, which define spatial regions, in this case, the 4 boundaries of the grid, and the position of the injection well:

```
bool onLeftBoundary_(const GlobalPosition &globalPos) const
{
  return globalPos[0] < this->bboxMin()[0] + eps_;
}

bool onRightBoundary_(const GlobalPosition &globalPos) const
{
  return globalPos[0] > this->bboxMax()[0] - eps_;
}

bool onLowerBoundary_(const GlobalPosition &globalPos) const
{
       return globalPos[1] < this->bboxMin()[1] + eps_;
}

bool onUpperBoundary_(const GlobalPosition &globalPos) const
{
    return globalPos[1] > this->bboxMax()[1] - eps_;
}

bool injectionWellPos_(const GlobalPosition &globalPos) const
{
    return (onLowerBoundary_(globalPos) && (globalPos[0] > 2500 - 1) && (globalPos[0] < 2500 +
1) && globalPos[1] < eps_);

}
```