Int. Conf. on Computational Methods for Coupled Problems in Science and Engineering
COUPLED PROBLEMS 2005
M. Papadrakakis, E. Oñate and B. Schrefler (Eds)

DOMAIN DECOMPOSITION AND FUNCTIONAL PROGRAMMING WITH OCAMLP3L

F. Clément*, **V. Martin***, **A. Vodicka***, **R. Di Cosmo**† **and P. Weis**‡

*Projet Estime, INRIA-Rocquencourt
B.P. 105, F-78153 Le Chesnay Cedex, France.
E-mail: Francois.Clement@inria.fr, web page: http://www-rocq.inria.fr/~fclement/.

†Case 7014, Université Paris 7
2, place Jussieu, F-75251 Paris Cedex 05, France.
E-mail: roberto@dicosmo.org, web page: http://www.pps.jussieu.fr/~dicosmo/.

‡Projet Cristal, INRIA-Rocquencourt
B.P. 105, F-78153 Le Chesnay Cedex, France.
E-mail: Pierre.Weis@inria.fr, web page: http://pauillac.inria.fr/~weis/.

Key words: Domain decomposition, Mixed finite elements, Mortar finite element, Parallelism, Implementation, Functional programming

Abstract. *When considering coupled problems, besides the issue of the existence of a global solution, arises also the difficulty of the implementation. Objective Caml, a high-level functional programming language based on formal semantics, is particularly well suited for the implementation of complex algorithms. Moreover, the OCamlP3l library provides a structured approach to parallel programming using skeletons and templates based compilation techniques: designing and debugging is performed on a sequential version, and the parallel version is automatically deduced by a recompilation of the source program. As an alternative to MPI, our approach to code coupling is then to use external programs to solve each subproblem, and to couple them through basic communications driven by an OCaml(P3l) coupling program. In this work, we present the application of a non-overlapping domain decomposition method with non-matching grids based on Robin interface conditions to the problem of flow surrounding a deep underground nuclear waste disposal. Numerical results for a realistic 3D flow problem are also shown.*

1 INTRODUCTION

We are concerned with the simulation of flow and transport in porous media. These problems have become more and more popular in the recent past because underground environmental issues such as water supplies management, pollution... have shown to have important consequences. In particular, in many industrial countries, the feasibility of an underground nuclear waste disposal is being studied[1]. The reliable simulation of flow and transport in porous media is crucial to forecast the behavior of contaminants in the geological layers. Most of these simulations should be computed in 3 dimensions on very long time scales (transport simulations over 10 million years), with very different length scales (going from the meter to 20 or 30 kilometers), involving possibly different physics. They obviously require very efficient numerical methods and computational capabilities. A good way of treating such a large scale problem is to divide it into smaller subproblems, and then to couple them. We use a domain decomposition method to divide the problem in a proper manner and the code coupling is obtained through a specific external coupling program.

We will restrict ourselves in this paper to the efficient simulation of stationary flow. In the Section 2, we present in full details a non-overlapping domain decomposition method that allows the treatment of non-matching grids. We used a mixed finite element method to solve the subdomain problems. This coupling method is based on Robin interface conditions and was studied in [3] in the mixed finite element case and in [1] in the finite volume case. It is somewhat different from the mortar method first introduced in [4]. This kind of nonconforming domain decomposition method is particularly practical as it allows a separate meshing of the subdomains. For example, this is useful to the underground storage simulations as it allows a local refinement around the storage, and the rest of the domain can be meshed with a coarser mesh, following the geological layers.

Code coupling is a very demanding implementation task, especially when the codes to couple have been developed separately. The main difficulty is the fine tuning of the communications between the codes. Moreover, in the case of huge memory requirements and/or heavy cpu capabilities, the only issue is parallelism. And again, the key point is to properly take care of the communications. Most of these aspects actually correspond to a sequence of generic basic tasks that should be automatically set up by a program, or better via compilation. This is where OCamlP3l enters the picture. Objective Caml (or OCaml) is a fast modern high-level functional programming language based on sound theoretical foundation and formal semantics, see `http://www.ocaml.org/`. It is particularly well suited for the implementation of complex algorithms. As an example, the OCamlP3l system, see `http://www.ocamlp3l.org/` and [8], provides a structured approach to parallel programming using skeletons and template based compilation techniques, see [12]. It brings all the piping capabilities we need to implement the communications, and moreover, it offers the parallelization for free: designing and debugging is driven on a sequential version, and the parallel version is automatically deduced by a simple recompilation of the same source program. In Section 3, we present the OCamlP3l system and an example of coupling program for the parallelization of the stationary flow simulation code, i.e. the coupling of the code with itself. And in Section 4, we show numerical results for both extensibility tests and a realistic 3D flow computation on data provided by ANDRA.

[1]In France, ANDRA (the French National Agency for the Management of Nuclear Wastes) is in charge of supervising these studies. See `http://www.andra.fr/`.

2 THE MULTIBLOCK PROBLEM

Let Ω be a convex domain in $\mathbb{R}^d$, $d = 2$ or 3, and we denote by $\Gamma = \partial\Omega$ the boundary of Ω. We suppose that the flow in Ω is governed by a conservation equation together with Darcy's law relating the gradient of the pressure p to the Darcy velocity $\mathbf{u}$:

$$\begin{array}{rcll} \operatorname{div} \mathbf{u} & = & q & \text{in } \Omega \\ \mathbf{u} & = & -\mathbf{K} \boldsymbol{\nabla} p & \text{in } \Omega \\ p & = & \overline{p} & \text{on } \Gamma, \end{array} \tag{1}$$

where p is the pressure, $\mathbf{u}$ the Darcy velocity, $\mathbf{K}$ the hydraulic conductivity (or permeability) tensor, q a source term and $\overline{p}$ the given pressure on the boundary Γ. We suppose that $\mathbf{K}$ is a symmetric, positive definite and uniformly bounded tensor.

Let Ω be decomposed into n non-overlapping subdomains, also called blocks, $\Omega_i, i \in I = \{1, 2, \dots, n\}$, and we denote by Γ_i the part of the boundary of Ω_i, $i \in I$ in common with the boundary of Ω. Hence,

$$\bar{\Omega} = \bigcup_{i \in I} \bar{\Omega}_i, \qquad \text{and} \qquad \Gamma_i = \partial\Omega_i \cap \Gamma, \quad i \in I = \{1, 2, \dots, n\},$$

where $\bar{\Omega}_i$ is the closure of the open set Ω_i. Let Σ_i be the geometric *internal boundary* of Ω_i, i.e. the interior of the internal boundary of Ω_i, and Σ the union of all internal boundaries,

$$\Sigma_i = (\partial\Omega_i \backslash \Gamma), \quad i \in I, \qquad \Sigma = \bigcup_{i \in I} \Sigma_i.$$

Then we can define the geometric *interface between subdomains* Ω_i and Ω_j as

$$\Sigma_{ij} = \Sigma_{ji} = \Sigma_i \cap \Sigma_j, \quad i \neq j \in I, \qquad \Sigma_{ii} = \emptyset, \quad i \in I.$$

If $\Sigma_{ij} \neq \emptyset$, the subdomains Ω_i and Ω_j, $i \neq j \in I$, are called neighbors. Let ν_i denote the external unit normal to $\Omega_i, i \in I$. We denote the $L^2(\Omega_i)$ or $(L^2(\Omega_i))^d$ the inner product by $(\cdot, \cdot)_i$. In these notations, we omit the subscript i to refer to the entire domain Ω. Let the interface inner product on $L^2(\Sigma_i)$ be denoted by $\langle \cdot, \cdot \rangle_i$ and on $L^2(\Sigma_{ij})$ by $\langle \cdot, \cdot \rangle_{ij}$. Note that if $\Sigma_{ij} = \emptyset$, this inner product is zero.

We denote by $p_i, \mathbf{u}_i, \mathbf{K}_i$, and q_i the restrictions of $p, \mathbf{u}, \mathbf{K}$, and q respectively to Ω_i, $i \in I$, and by $\overline{p}_i$ the restriction of $\overline{p}$ to Γ_i, $i \in I$. We can rewrite the above problem (1) as a transmission, or multiblock, problem:

$$\begin{array}{rcll} \operatorname{div} \mathbf{u}_i & = & q_i & \text{in } \Omega_i, \quad i \in I, \\ \mathbf{u}_i & = & -\mathbf{K}_i \boldsymbol{\nabla} p_i & \text{in } \Omega_i, \quad i \in I, \\ p_i & = & \overline{p}_i & \text{on } \Gamma_i, \quad i \in I, \end{array} \tag{2}$$

together with transmission conditions

$$\begin{array}{rcll} p_i & = & p_j & \text{on } \Sigma_{ij}, \quad i, j \in I, \\ \mathbf{u_i} \cdot \nu_\mathbf{i} & = & -\mathbf{u_j} \cdot \nu_\mathbf{j} & \text{on } \Sigma_{ij}, \quad i, j \in I. \end{array} \tag{3}$$

The two equations in (3) yield the continuity of pressure and of normal velocity (as $\nu_i = -\nu_j$) across each interface $\Sigma_{ij}(\neq \emptyset)$ between two neighboring subdomains Ω_i and $\Omega_j,\ i,j \in I$. These interface equations can be rewritten as Robin conditions. Let $\alpha_{ij} > 0$ and $\alpha_{ji} > 0$ be the Robin coefficients on the interface $\Sigma_{ij} = \Sigma_{ji},\ i,j \in I$, the two equations in (3) are equivalent to the Robin interface conditions

$$\begin{aligned} -\mathbf{u_i} \cdot \nu_\mathbf{i} + \alpha_{ij} p_i &= \mathbf{u_j} \cdot \nu_\mathbf{j} + \alpha_{ij} p_j \qquad \text{on } \Sigma_{ij},\quad i,j \in I, \\ -\mathbf{u_j} \cdot \nu_\mathbf{j} + \alpha_{ji} p_j &= \mathbf{u_i} \cdot \nu_\mathbf{i} + \alpha_{ji} p_i \qquad \text{on } \Sigma_{ij},\quad i,j \in I. \end{aligned} \tag{4}$$

2.1 Weak multiblock problem

Throughout the sequel of this section, for simplicity, we will assume that homogeneous Dirichlet boundary conditions are imposed on Γ: $\overline{p} = 0$ on Γ.

Let us define the approximation spaces used in this paper:

$$\begin{aligned} &\mathbf{Z_i} = \mathcal{H}(\mathrm{div}, \Omega_i), && i \in I, && \mathbf{Z} = \bigoplus_{i\in I} \mathbf{Z_i} \not\subset \mathcal{H}(\mathrm{div}, \Omega), \\ &N_i = L^2(\Omega_i), && i \in I, && N = \bigoplus_{i\in I} N_i = L^2(\Omega), \\ &\Lambda_i = L^2(\Sigma_i), && i \in I, && \Lambda = \bigoplus_{i\in I} \Lambda_i. \end{aligned}$$

With these notations, we can write the mixed formulation. A weak solution of (2)-(4) is a couple $(\mathbf{u}, p) \in \mathbf{Z} \times N$ such that for each $i \in I$, $p_i|_{\Sigma_i} \in \Lambda_i$, and $\mathbf{u_i} \cdot \nu_\mathbf{i}|_{\Sigma_i} \in \Lambda_i$, and

$$\begin{aligned} (K^{-1}\mathbf{u},\, \mathbf{v})_i - (\mathrm{div}\, \mathbf{v},\, p)_i &= -\langle \mathbf{v} \cdot \nu_\mathbf{i},\, p_i \rangle_i\,, && \mathbf{v} \in \mathbf{Z_i}, \\ (\mathrm{div}\, \mathbf{u},\, r)_i &= (q_i,\, r)_i\,, && r \in N_i, \\ \sum_{j\in I} \langle -\mathbf{u_i} \cdot \nu_\mathbf{i} + \alpha_{ij} p_i,\, \mu_i \rangle_{ij} &= \sum_{j\in I} \langle \mathbf{u_j} \cdot \nu_\mathbf{j} + \alpha_{ij} p_j,\, \mu_i \rangle_{ij}\,, && \mu_i \in \Lambda_i. \end{aligned} \tag{5}$$

2.2 Meshes on the subdomains

Let $\mathcal{T}_i^h$ be a conforming finite element partition of the subdomain $\Omega_i,\ i \in I$, where the maximal element diameter is bounded by the real $h > 0$, and the interior of each element face (or edge in 2D) are entirely contained in Σ_i or in Γ_i. The meshes of two neighboring subdomains need not match at their interface Σ_{ij}.

Let us now define more precisely the interface meshes. We have seen that two subdomains are called *neighbors* if their interface $\Sigma_{ij} \neq \emptyset$. For $i \in I$, we denote the number of neighbors of a subdomain Ω_i by n_i, and we denote the set of indices of these neighbors by $\mathcal{N}_i = \{j_1, j_2, \ldots, j_{n_i}\}$. The sets I and $\mathcal{N}_i$ inherit the natural order on $\mathbb{N}$. Thus $k \in 1, 2, \ldots, n_i \mapsto j_k \in \mathcal{N}_i$ is increasing. For an $i \in I$ and a $j \in \mathcal{N}_i$, the trace on Σ_{ij} of the mesh $\mathcal{T}_i^h$ of Ω_i is denoted by $\mathcal{S}_{ij}^h$. In the same way, for a $j \in I$ and an $i \in \mathcal{N}_j$, the trace on $\Sigma_{ji} = \Sigma_{ij}$ of the mesh $\mathcal{T}_j^h$ is denoted by $\mathcal{S}_{ji}^h$. In general $\mathcal{S}_{ji}^h \neq \mathcal{S}_{ij}^h, i \neq j$, as the meshes $\mathcal{T}_i^h$ and $\mathcal{T}_j^h$ do not match, see Figure 1. We denote by $\mathcal{S}_i^h$ the trace on Σ_i of the mesh $\mathcal{T}_i^h$

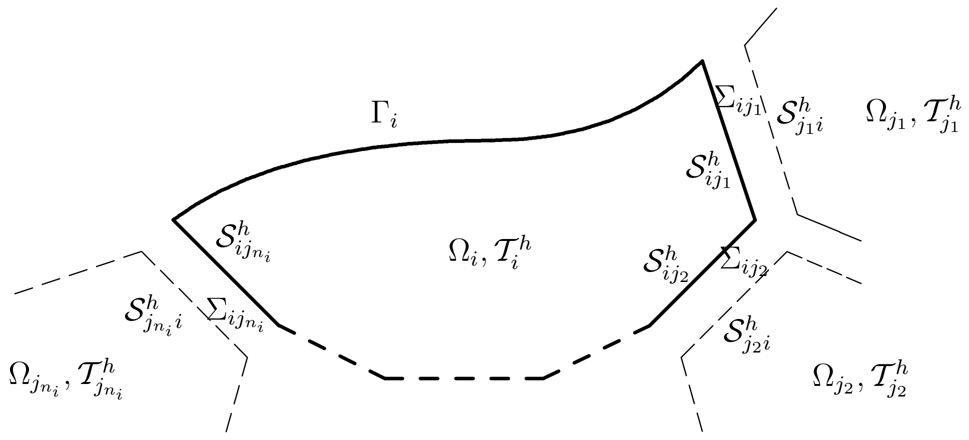

Figure 1: The subdomain Ω_i and its n_i neighbors Ω_{j_k}, $j_k \in \mathcal{N}_i$. The trace meshes $\mathcal{S}^h_{ij_k}$ and $\mathcal{S}^h_{j_k i}$ on each side of the interface Σ_{ij_k} are different. For the sake of simplicity, the picture has been drawn in two dimensions.

of Ω_i. It is also the union of all the interface meshes coming from the subdomain Ω_i. We call the mesh $\mathcal{S}^h_i$ the *inner interface mesh* of Ω_i,

$$\forall i \in I,\ \forall j \in \mathcal{N}_i \quad \mathcal{S}^h_{ij} = \mathcal{T}^h_i|_{\Sigma_{ij}}. \qquad \mathcal{S}^h_i = \biguplus_{j \in \mathcal{N}_i} \mathcal{S}^h_{ij}.$$

The set of the ordered couples of neighbors is denoted by $\mathcal{N} = \{(i,j), i \in I, j \in \mathcal{N}_i\}$. It is ordered according to lexicographic order on $\mathbb{N}^2$. We denote by s the permutation

$$\forall (i,j) \in \mathcal{N}, \quad s(i,j) = (j,i) \in \mathcal{N}.$$

We call *inner interface structure* $\mathcal{S}^h$ of the domain Ω the union of all inner interface meshes (each geometric interface is thus counted twice). It is ordered by $\mathcal{N}$,

$$\mathcal{S}^h = \biguplus_{i \in I} \mathcal{S}^h_i = \biguplus_{i \in I} \biguplus_{j \in \mathcal{N}_i} \mathcal{S}^h_{ij} = \biguplus_{(i,j) \in \mathcal{N}} \mathcal{S}^h_{ij}.$$

Up to now, we have defined the interface meshes that were seen from a subdomain Ω_i. Now, we define the interface meshes of the neighbors of Ω_i. For $(i,j) \in \mathcal{N}$, we denote $\tilde{\mathcal{S}}^h_{ij} = \mathcal{S}^h_{ji}$. We denote by $\tilde{\mathcal{S}}^h_i$ the *outer interface mesh* of the subdomain Ω_i that is the union of the interface meshes that are the traces of the neighbors of Ω_i. It is also ordered by $\mathcal{N}_i$,

$$\tilde{\mathcal{S}}^h_i = \biguplus_{j \in \mathcal{N}_i} \tilde{\mathcal{S}}^h_{ij} = \biguplus_{j \in \mathcal{N}_i} \mathcal{S}^h_{ji}.$$

In the same way, we denote by $\tilde{\mathcal{S}}^h$ the *outer interface structure* of the domain Ω that is the union of all outer interface meshes. It is also the union of all interface meshes, seen from a subdomain or its neighbor, but this time it is ordered by $s(\mathcal{N})$,

$$\tilde{\mathcal{S}}^h = \biguplus_{i\in I} \tilde{\mathcal{S}}_i^h = \biguplus_{i\in I} \biguplus_{j\in\mathcal{N}_i} \tilde{\mathcal{S}}_{ij}^h = \biguplus_{i\in I} \biguplus_{j\in\mathcal{N}_i} \mathcal{S}_{ji}^h = \biguplus_{(i,j)\in s(\mathcal{N})} \mathcal{S}_{ij}^h.$$

2.3 Discrete weak multiblock problem

Let

$$\mathbf{Z_{h,i}} \times N_{h,i} \times \Lambda_{h,i} \subset \mathbf{Z_i} \times N_i \times \Lambda_i, \quad i \in I,$$

be the usual mixed finite element approximation made of Raviart-Thomas (and Nedelec in 3D) spaces of lowest order ([13], [14], [10]), with the hybrid Lagrange multiplier spaces $\Lambda_{h,i}$ on Σ_i, [5]. We recall that $Z_{h,i}$ consists of some linear vector functions and that $N_{h,i}$ (respectively $\Lambda_{h,i}$) are made up of cellwise (resp. facewise in 3D or edgewise in 2D) constant scalar functions. Let

$$\mathbf{Z_h} = \bigoplus_{i\in I} \mathbf{Z_{h,i}}, \qquad N_h = \bigoplus_{i\in I} N_{h,i}, \qquad \Lambda_h = \bigoplus_{i\in I} \Lambda_{h,i}.$$

We also recall that

$$\mathrm{div}\mathbf{Z_{h,i}} = N_{h,i} \qquad \Lambda_{h,i} = \mathbf{Z_{h,i}} \cdot \nu_{\mathbf{i}}|_{\Sigma_i} = P_0(\mathcal{S}_i^h), \quad i \in I,$$

where $P_0(\mathcal{S}_i^h)$ denotes the space of facewise (or edgewise in 2D) constant functions on $\mathcal{S}_{ij}^h$.

One can notice that the space Λ_h contains two interface unknowns on each interface $\Sigma_{ij}, i \in I, j \in \mathcal{N}_i$: the one that lives in $P_0(\mathcal{S}_{ij}^h)$, and the other one that lives in $P_0(\mathcal{S}_{ji}^h) = P_0(\tilde{\mathcal{S}}_{ij}^h)$. As we did in the previous section for the interface meshes, we introduce on the geometric interface between two subdomains Σ_{ij} two different approximation spaces $\Lambda_{h,ij}$ and $\tilde{\Lambda}_{h,ij} = \Lambda_{h,ji}$,

$$\begin{aligned} \Lambda_{h,ij} &= \mathbf{Z_{h,i}} \cdot \nu_{\mathbf{i}}|_{\Sigma_{ij}} = P_0(\mathcal{S}_{ij}^h), \quad i \in I,\ j \in \mathcal{N}_i \quad \text{(i.e. } (i,j) \in \mathcal{N}), \\ \tilde{\Lambda}_{h,ij} = \Lambda_{h,ji} &= \mathbf{Z_{h,j}} \cdot \nu_{\mathbf{j}}|_{\Sigma_{ij}} = P_0(\mathcal{S}_{ji}^h), \quad j \in I,\ i \in \mathcal{N}_j. \end{aligned}$$

Next we define the space $\tilde{\Lambda}_{h,i}$ that contains the functions on the interface meshes $\tilde{\mathcal{S}}_i^h$ of the neighbors of $\Omega_i,\ i \in I$. This yields

$$\begin{aligned} \Lambda_{h,i} &= \bigoplus_{j\in\mathcal{N}_i} \Lambda_{h,ij} &&= \mathbf{Z_{h,i}} \cdot \nu_{\mathbf{i}}|_{\Sigma_i} = P_0(\mathcal{S}_i^h), \qquad i \in I, \\ \tilde{\Lambda}_{h,i} &= \bigoplus_{j\in\mathcal{N}_i} \tilde{\Lambda}_{h,ij} = \bigoplus_{j\in\mathcal{N}_i} \Lambda_{h,ji} &&= \mathbf{Z_{h,j}} \cdot \nu_{\mathbf{j}}|_{\Sigma_i} = P_0(\tilde{\mathcal{S}}_i^h), \qquad i \in I, \end{aligned}$$

and

$$\begin{aligned} \Lambda_h &= \bigoplus_{i\in I} \Lambda_{h,i} = \bigoplus_{i\in I} \bigoplus_{j\in\mathcal{N}_i} \Lambda_{h,ij} = \bigoplus_{(i,j)\in\mathcal{N}} \Lambda_{h,ij}, \\ \tilde{\Lambda}_h &= \bigoplus_{i\in I} \tilde{\Lambda}_{h,i} = \bigoplus_{i\in I} \bigoplus_{j\in\mathcal{N}_i} \tilde{\Lambda}_{h,ij} = \bigoplus_{i\in I} \bigoplus_{j\in\mathcal{N}_i} \Lambda_{h,ji} = \bigoplus_{(i,j)\in s(\mathcal{N})} \Lambda_{h,ij}. \end{aligned}$$

We write the discrete mixed finite element approximation for the original problem (1) or, equivalently, for the transmission problem (5) introducing inter-element multipliers $\lambda_{h,i}$ (see [7], [2] and references therein). We seek $(\mathbf{u_h}, p_h, \lambda_h) \in \mathbf{Z_h} \times N_h \times \Lambda_h$ such that

$$\begin{array}{rcll} (K^{-1}\mathbf{u_h},\, \mathbf{v})_i - (\operatorname{div} \mathbf{v},\, p_h)_i &=& -\left\langle \mathbf{v}\cdot\nu_\mathbf{i},\, \lambda_{h,i}\right\rangle_i, & \mathbf{v} \in \mathbf{Z_{h,i}}, \\ (\operatorname{div} \mathbf{u_h},\, r)_i &=& (q_i,\, r)_i, & r \in N_{h,i}, \\ \sum\limits_{j\in\mathcal{N}_i} \left\langle -\mathbf{u_{h,i}}\cdot\nu_\mathbf{i} + \alpha_{ij}\lambda_{h,i},\, \mu_i\right\rangle_{ij} &=& \sum\limits_{j\in\mathcal{N}_i} \left\langle \mathbf{u_{h,j}}\cdot\nu_\mathbf{j} + \alpha_{ij}\lambda_{h,j},\, \mu_i\right\rangle_{ij}, & \mu_i \in \Lambda_{h,i}. \end{array} \tag{6}$$

It is proved that this problem (6) is well-posed in [3] and in [1] for the finite volume method. It is pointed out in [3] that local mass conservation is insured everywhere except on the interfaces. There is continuity of the normal velocity $\mathbf{u}\cdot\nu$ across every element faces (or edges in 2D) that does *not lie on* Σ. This is the price to pay the non-matching meshes. The flux is not conserved across any portion of Σ, but since $\mu_i = 1 \in \Lambda_{h,i}$, we continue to conserve the mass globally.

A simple method to solve this problem (6) is to solve iteratively the fixed point problem (7). Given a $x_h^0 \in \Lambda_h$, where the superscript is used for the iteration $k \geq 0$ in the method, solve iteratively: compute $x_{h,i}^{k+1} = (\mathbf{u_{h,i}^{k+1}}\cdot\nu_\mathbf{i} + \alpha_{ij}\lambda_{h,i}^{k+1})|_{\Sigma_{ij}} \quad j \in \mathcal{N}_i$, where $(\mathbf{u_{h,i}^{k+1}}, p_{h,i}^{k+1}, \lambda_{h,i}^{k+1}) \in \mathbf{Z_{h,i}} \times N_{h,i} \times \Lambda_{h,i}$ such that

$$\begin{array}{rcll} \left(K^{-1}\mathbf{u_h^{k+1}},\, \mathbf{v}\right)_i - \left(\operatorname{div} \mathbf{v},\, p_h^{k+1}\right)_i &=& -\left\langle \mathbf{v}\cdot\nu_\mathbf{i},\, \lambda_{h,i}^{k+1}\right\rangle_i, & \mathbf{v} \in \mathbf{Z_{h,i}}, \\ \left(\operatorname{div} \mathbf{u_h^{k+1}},\, r\right)_i &=& (q_i,\, r)_i, & r \in N_{h,i}, \\ \sum\limits_{j\in\mathcal{N}_i} \left\langle -\mathbf{u_{h,i}^{k+1}}\cdot\nu_\mathbf{i} + \alpha_{ij}\lambda_{h,i}^{k+1},\, \mu_i\right\rangle_{ij} &=& \sum\limits_{j\in\mathcal{N}_i} \left\langle x_{h,i}^k,\, \mu_i\right\rangle_{ij}, & \mu_i \in \Lambda_{h,i}. \end{array} \tag{7}$$

When the method has converged, the Robin interface conditions (4) are imposed weakly, and thus the solution of (6) is reached. It is proved in [1] that this fixed point method converges. One can notice that it is non-symmetric even though the Laplace operator is symmetric. When using an iterative method, it is advisable to accelerate the convergence with a non-symmetric Krylov method, such as a BiCGStab or a GMRes method.

2.4 Interface operators

In order to clearly define the fixed point problem, we introduce discrete Robin-to-Robin operators S_{iq_i} in each subdomain Ω_i for $q_i \in N_{h,i}$ defined by:

$$\begin{array}{rcll} S_{iq_i} : \Lambda_{h,i} &\longrightarrow& \Lambda_{h,i}, & i \in I \\ S_{iq_i} x_{h,i} &=& (\mathbf{u_{h,i}}(x_{h,i})\cdot\nu_i + \alpha_{ij} p_{h,i}(x_{h,i}))|_{\Sigma_{ij}}, & i \in I, \quad \forall j \in \mathcal{N}_i, \end{array} \tag{8}$$

such that $\mathbf{u_{h,i}}(x_h), p_{h,i}(x_h)$ is the solution of the following problem set on Ω_i, $i \in I$: seek $(\mathbf{u_{h,i}}, p_{h,i}, \lambda_{h,i}) \in \mathbf{Z_{h,i}} \times N_{h,i} \times \Lambda_{h,i}$ such that

$$\begin{array}{rcll} (K^{-1}\mathbf{u_{h,i}},\, \mathbf{v})_i - (\operatorname{div} \mathbf{v},\, p_{h,i})_i &=& -\left\langle \mathbf{v}\cdot\nu_\mathbf{i},\, \lambda_{h,i}\right\rangle_i, & \mathbf{v} \in \mathbf{Z_{h,i}}, \\ (\operatorname{div} \mathbf{u_{h,i}},\, r)_i &=& (q_i,\, r)_i, & r \in N_{h,i}, \\ \sum\limits_{j\in\mathcal{N}_i} \left\langle -\mathbf{u_{h,i}}\cdot\nu_\mathbf{i} + \alpha_{ij}\lambda_{h,i},\, \mu_i\right\rangle_{ij} &=& \sum\limits_{j\in\mathcal{N}_i} \left\langle x_{h,ij},\, \mu_i\right\rangle_{ij}, & \mu_i \in \Lambda_{h,i}. \end{array} \tag{9}$$

For all neighbors Ω_i and Ω_j, $(i,j) \in \mathcal{N}$, we define the L^2-projection operator onto $\tilde{\Lambda}_{h,ij} = \Lambda_{h,ji}$ by

$$\begin{array}{rcll} P_{i\to j} : L^2(\Sigma_{ij}) & \longrightarrow & \tilde{\Lambda}_{h,ij}, & \\ x_{h,ij} & \longmapsto & P_{i\to j} x_{h,ij} & \text{such that} \\ \langle x_{h,ij} - P_{i\to j} x_{h,ij}, \tilde{\mu}_{ij} \rangle_{ij} & = & 0, & \quad \tilde{\mu}_{ij} \in \tilde{\Lambda}_{h,ij}. \end{array} \tag{10}$$

For all subdomain Ω_i, $i \in I$, we define a projection operator and two restriction operators. The L^2-projection operator onto $\tilde{\Lambda}_{h,i}$ maps the *inner* boundary interface onto the *outer* boundary interface,

$$\begin{array}{rcl} P_i : L^2(\Sigma_i) & \longrightarrow & \tilde{\Lambda}_{h,i}, \\ x_{h,i} = (x_{h,ij_1}, x_{h,ij_2}, \ldots, x_{h,ij_{n_i}}) & \longmapsto & (P_{i\to j_1} x_{h,ij_1}, P_{i\to j_2} x_{h,ij_2}, \ldots, P_{i\to j_n} x_{h,ij_n}). \end{array}$$

And the restriction operators are defined by

$$\begin{array}{rcl} R_i : x_h = (x_{h,1}, x_{h,2}, \ldots, x_{h,n}) \in \Lambda_h & \longmapsto & x_{h,i} \in \Lambda_{h,i}, \\ \tilde{R}_i : \tilde{x}_h = (\tilde{x}_{h,1}, \tilde{x}_{h,2}, \ldots, \tilde{x}_{h,n}) \in \tilde{\Lambda}_h & \longmapsto & \tilde{x}_{h,i} \in \tilde{\Lambda}_{h,i}. \end{array}$$

Finally, we define the global operators

$$\begin{array}{rcl} P = \sum_{i\in I} \tilde{R}_i^\top P_i R_i : \Lambda_h & \longrightarrow & \tilde{\Lambda}_h \\ S_q = \sum_{i\in I} \tilde{R}_i^\top S_{iq_i} R_i : \Lambda_h & \longrightarrow & \Lambda_h \end{array} \tag{11}$$

and we set

$$A_h = \mathrm{Id}_{\Lambda_h} - sPS_0 \quad \text{and} \quad b_h = sPS_q(0). \tag{12}$$

With all these notations, solving the problem (6) is equivalent to find $x_h \in \Lambda_h$ solution to the linear system

$$A_h x_h = b_h. \tag{13}$$

3 IMPLEMENTATION USING OCAMLP3L

3.1 The OCamlP3l parallel programming system

Caml is a strongly-typed functional programming language from the ML (Meta Language) family. OCaml (Objective Caml) is an open source implementation of Caml developed at INRIA, see `http://caml.inria.fr/` and `http://www.ocaml.org/`.

Caml was issued in the 80's from theoretical considerations on the relationship between category theory and lambda-calculus, hence its name which stands for "Categorical Abstract Machine Language". Caml is safe: the compiler ensures that data manipulated by programs may *never* be corrupted. It is strongly typed-checked, but type annotations are fully automatic and handled by the compiler. It is functional since there is no restrictions in the definition and usage of functions, that can be passed as arguments or returned as values. Caml features "polymorphic typing": some types may be left unspecified, standing for "any type"; that way, general purpose functions may be applied to any kind of data, regardless of their types.

OCaml features objects with a fully fledged object oriented programming style. Moreover, OCaml provides a module system that is powerful and safe: modules can include other (sub-)modules at any depth, and functions from modules to modules are supported (to define modules that are parameterized by other modules).

To give an example of the capabilities of OCaml, here is the (single line of) code to implement the composition of functions in its full generality:

```
#let compose f g = function x -> g (f x);;
```

Its type is `('a -> 'b) -> ('b -> 'c) -> 'a -> 'c`, meaning that this is a function taking two arguments: a function from a set `'a` to another set `'b` and another function from the same set `'b` to a third set `'c`; and returning a third function from `'a` to `'c`.

P3L (Pisa Parallel Programming Language) is a structured parallel programming language developed at the Department of Computer Science of the University of Pisa. It is based upon skeleton/templates and allows parallel programs to be developed composing a small set of primitive parallel forms, see `http://www.di.unipi.it/~susanna/p3l.html`.

OCamlP3l is a parallel programming system based on OCaml and P3l languages, providing seamless integration of parallel programming and functional programming and advanced features like sequential logical debugging of parallel programs and strong typing, useful both in teaching parallel programming and in the building of full-scale applications, see `http://www.ocamlp3l.org/`.

We give now a flavor of the capabilities of the OCamlP3l system, and the reader can refer to [6] for more details on the implementation.

3.1.1 Three semantics

A distinctive feature of the OCamlP3l system, among all skeleton-based systems, is that the semantics of the skeletons is not hard-wired: the system allows the user to compile his code without any source modification using three possible semantics, i.e. three implementations of the skeletons. The *sequential semantics* produces an executable that can run on a single machine, as a single process, and easily debugged using standard techniques and tools for sequential programs. The *parallel semantics* produces a generic SPMD program that can be deployed on a parallel machine, a cluster, or a network of workstations. The *graphical semantics* produces a program that displays a picture of the parallel computational network that is deployed when running the parallel version.

A key issue in the further development of OCamlP3l will be the proof of the adequation theorem that states the agreement between sequential and parallel executions: for any user program the two semantics should produce exactly the same result. Hence, the user has only to debug the sequential version!

3.1.2 Skeletons combinators

The OCamlP3l skeletons are compositional: the skeletons are combinators that form an algebra of functions and functionals called the *skeleton language*. More precisely, a skeleton is a *stream processor*, i.e. a function that transforms an input stream of incoming data into an output stream of outgoing data. Then, these functions can be composed arbitrarily, thus leading to trees of combinators that define the parallel behavior of programs.

In the current version of OCamlP3l (1.9), the eight combinators (or basic building blocks) of the skeleton language pertain to five kinds:

- the *task parallel* skeletons `farm` and *pipeline* model the parallelism of independent processing activities relative to different input data.
- the *data parallel* skeletons `mapvector` and `reducevector` model the parallel computation of different parts of the same input data.
- the *data interface* skeletons `seq` and `parfun` provide (dual) injection and projection between sequential and parallel worlds.
- the *parallel execution scope delimiter* skeleton `pardo` must encapsulate all the code that invokes `parfun`'s.
- the *control* skeleton `loop` provides the repetitive execution of a given skeleton expression.

The `farm` combinator computes in parallel a function over different data items of the input stream. Given a stream of data items $x_1, \ldots, x_n$, the expression `farm`(f,k) computes $f(x_1), \ldots, f(x_n)$ by having k independent processes that compute f.

The *pipeline* combinator is denoted by the infix operator `|||`, it computes in parallel different stages of a function composition over different data items of the input stream. The expression f_1 `|||` f_2 `|||` $\ldots$ `|||` f_k computes $f_k(\ldots f_2(f_1(x_i))\ldots)$ over all data items x_i by having k independent processes, each process computes a function f_i over the data items produced by the process computing f_{i-1} and delivers its results to the process computing f_{i+1}.

The `mapvector` combinator computes in parallel a function over all the components of *vector* data items of the input stream. The expression `mapvector`(f,k) computes $(f(x_i^1), \ldots, f(x_i^n))$ over all (vector) data items $x_i = (x_i^1, \ldots, x_i^n)$ by having k independent processes that compute f over different components of the vector.

The `reducevector` combinator folds in parallel a function over all the components of *vector* data items of the input stream. The expression `reducevector`(f,k) computes $x_i^1 f x_i^2 f \ldots f x_i^n$ over all (vector) data items $x_i = (x_i^1, \ldots, x_i^n)$ by having k independent processes that folds f over different components of the vector.

The `seq` combinator converts a sequential function into a node of the parallel computational network.

The `parfun` combinator converts a parallel computational network into a sequential stream processing function.

All skeleton combinators allow for global or local initialization, meaning that independent processes will share, or not, their initialization data.

3.2 Implementation of the code coupling

3.2.1 Codes to couple

To implement the coupling, we use three C++ codes.

The build_interface_meshes code inputs the name of the file describing the 3D mesh associated with a subdomain, e.g. Ω_i. It reads this file from the disk, computes all the 2D inner interface meshes $\mathcal{S}_{ij}^h$, $j \in \mathcal{N}_i$ and then writes the files describing these 2D meshes on the disk. It outputs nothing. It needs no I/O redirections.

The build_projection_matrix code inputs the couple of names of the two files describing the two 2D meshes on the same interface, e.g. Σ_{ij}. It reads these files from the disk, computes and finally outputs the corresponding projection matrix $P_{i\to j}$ defined in (10). It needs redirection of its standard output.

The solve_on_a_subdomain code inputs the name of the file describing the 3D mesh associated with a subdomain, e.g. Ω_i. It reads this file from the disk and enters an infinite loop waiting for a keyword.

When given the keyword `"init"`, it computes the inverse of matrix used to solve the inner linear systems (9), then computes and outputs $S_{iq_i}0_i$ needed for the computation of the right-hand side b_h given by (12).

When given the keyword `"loop"`, it inputs $x_{h,i}$, then computes and outputs $S_{i0_i}x_{h,i}$ needed for the computation of the matrix-vector product.

When given the keyword `"final"`, it inputs $x^\star_{h,i}$, then computes and writes on the disk the solution $\mathbf{u}^\star_{\mathbf{h},\mathbf{i}}, p^\star_{h,i}$ to the problem (9) associated with $S_{i0_i}x^\star_{h,i}$.

It needs redirection of both its standard input and standard output. Moreover, the `"init"` phase is very costly and must be performed once and for all. Therefore, this code has to be *locally initialized*, and needs also to have the ability to be recalled, by having its I/O channels stored.

Since the matrix A_h of the outer linear system (13) is non-symmetric, it is advisable to accelerate the convergence with a non-symmetric Krylov method as BiCGStab. The implementation of BiCGStab only need a matrix-free matrix-vector product routine `aah` to compute the action of matrix A_h on any vector x_h. The `parfun` skeleton allows us to make this *ordinary routine* be a parallel computation network.

3.2.2 The coupling algorithm

Then, the coupling algorithm is the following.

Initialization

- build in parallel the interface meshes for all subdomains.
- compute in parallel the projection matrices for all entries in the connectivity table.
- compute in parallel the inverse of the matrices of the inner systems for all subdomains.
- define `aah` that applies in parallel matrix A_h given by (12) to any vector $x_h \in \Lambda_h$.
- compute in parallel the right-hand side b_h given by (12).
- choose $x_h^0 = 0$.

Iteration

- run BiCGStab with the parallel matrix-vector product `aah`, the right-hand side b_h and the initial guess x_h^0.
- call the solution $x_h^\star$.

Finalization

- solve in parallel the inner systems associated with $S_q x_h^\star$ and store the $\mathbf{u}_{\mathbf{h},\mathbf{i}}^\star$'s and $p_{h,i}^\star$'s for all subdomains.

3.2.3 The OCamlP3l coupling code

The coordination code itself is so short that it can be presented in extenso.

```
(* we have underlined the OCamlP3l skeletons *)

let encapsulate_mapvector prog n =
  let body () = mapvector (seq prog, n) in
  parfun body;;

let build_all_interface_meshes =
```

```
  let prog () i =
    let file = Ddec.filename_of_3D_mesh_number i in
    let command = "build_interface_meshes "^file in
    Sys.command command in
  encapsulate_mapvector prog Ddec.number_of_processors;;

let build_all_projection_matrices =
  let prog () i j =
    let file_i = Ddec.filename_of_2D_mesh_number i j
    and file_j = Ddec.filename_of_2D_mesh_number j i in
    let command = "build_projection_matrix "^file_i^" "^file_j in
    let ic = Unix.open_process_in command in
    Ddec.read_projection_matrix ic in
  encapsulate_mapvector prog Ddec.number_of_processors;;

let solve_on_all_subdomains =
  let prog () =
    let cin = ref None and cout = ref None in
    (fun (i, tagged_xh_i) ->
       let file_i = Ddec.filename_of_3D_mesh_number i in
       let command = "solve_on_a_subdomain "^file_i in
       let ic, oc = My_unix.spawn command cin cout in
       Ddec.print_tagged_internal_boundary_values oc tagged_xh_i;
       flush oc;
       let yh_i = Ddec.read_internal_boundary_values ic in
       Ddec.iter_vector_of yh_i) in
  encapsulate_mapvector prog Ddec.number_of_processors;;

pardo (fun () ->
  (* Initialization *)
  let n = Ddec.number_of_subdomains in
  let v = Array.init n (fun i -> i) in
  build_all_interface_meshes v;
  let projection_of =
    let projection_matrices =
      let v = Array.of_list Ddec.connectivity_table in
      build_all_projection_matrices v in
    Ddec.projection_with projection_matrices in
  let f_of =
    (fun v ->
      let yh = solve_on_all_subdomains v in
```

```
          let yh_tilde = projection_of yh in
50        permutation_of yh_tilde) in
      let aah =
        (fun xh ->
           let yh_2 = f_of (Ddec.iter_vector_of xh) in
           (Ddec.axpy (-.1.) yh_2 xh)) in
55    let bh = f_of (Ddec.init_vector_of_size n) in
      let xh_0 = Ddec.zero_structure_values n in
      let algorithm = Bicgstab.algorithm Ddec.axpy Ddec.dot in
      (* Iteration *)
      let xh_star = algorithm aah b xh_0 in
60    (* Finalization *)
      let v = Ddec.final_vector_of xh_star in
      solve_on_all_subdomains v;
    );;
```

The Ddec module is dedicated to domain decomposition. In particular, it delivers types for the unknown vector x_h that ease the implementation of the restriction and reconstruction operators R_i and $\tilde{R}_i^T$. Similar types are designed for the projection matrix P and a function to apply this matrix is provided.

The function `encapsulate_mapvector` (lines 3–5) is used to encapsulate a `mapvector` skeleton inside a `parfun` skeleton. This allows to compute in parallel a function over all the components of a vector anywhere in the sequential code.

The functions `build_all_interface_meshes` (lines 7–12), `build_all_projection_matrices` (lines 14–21) and `solve_on_all_subdomains` (lines 23–34) correspond to the three codes to couple of section 3.2.1. They are all defined through the `encapsulate_mapvector` network factory, but using ad'hoc techniques to connect —or not— their standard I/O's. Only the third one uses local initialization to reserve memory slots to store the I/O channels: this corresponds to the code between the `()` argument and the other arguments of the function `prog`.

The domain decomposition algorithm is implemented inside the `pardo` scope delimiter (lines 36–63). It uses the three previously defined stream processing networks, the third one been repeatedly employed. The function `f_of` (lines 46–50) defines the function $f = sPS_q$ for which we are searching the fixed point. It is used in both the body of the matrix-vector product `aah` (lines 51–54) and the computation of the right-hand side `bh` (line 55).

4 NUMERICAL TESTS

Let us first report on the fact that we effectively did not have to debug our program in the parallel mode. Once the code ran correctly in the sequential mode on our development

Linux Intel workstation, we simply copied all executable files and data files on a Linux Intel cluster, and it worked. Without any parallel debugging needs, but also without installing anything specific on the cluster —neither OCamlP3l nor OCaml. This is quite a good point about development delays.

We present now numerical tests obtained with the OCamlP3l implementation of the nonconforming Robin-Robin algorithm detailed in the previous sections. The first series concern performance measurements on an academic 3D test case. And the second series shows a more realistic 3D flow computation. More details can be found in [9].

All simulations have been run on the Cristal Cluster deployed at INRIA-Rocquencourt. It is made of sixteen 2.8 GHz Intel Xeon bi-processors with 2 Gb of RAM each interconnected on a dedicated Gigabit network.

4.1 Extensibility results

The principle of this test is the following: we associated one processor to each subdomain, and while keeping a constant load per processor and we increased the number of processors, i.e. the number of subdomains; and the goal is to keep as much as possible a constant computation time. All subdomains were made of about 50000 cells, which was the limit imposed by the memory requirements of the factorization of the inner linear systems in order to avoid swapping problems during the initialization phase.

Obviously, there is always an overhead when going from one subdomain to two subdomains, as in the first case we do not need any coupling (BiCGStab iteration, projection, communication), but we noticed that all computation times remain in the 20–23 minutes range when the interface meshes matched, and in the 25–34 minutes range for the non-matching case. The variations were mainly due to the way the global domain is decomposed, and of course to the number of interfaces. This leads to a flat 30%-overhead in the conforming case (the projection operator is the identity), and no more than an 100%-overhead in the non conforming case, which is not bad for up to 16 subdomains, i.e. up to 800000 cells.

The main drawback of the current version of OCamlP3l is the existence of a bottleneck since all communications are centralized in the `mapvector` skeleton. Obviously, this point was not crucial in our tests, but this may become an issue for larger tests. So a specific skeleton accounting for direct communication driven by a connectivity table is under development.

4.2 A realistic 3D flow simulation

We consider now the problem of a 3D flow simulation in a realistic porous media designed by ANDRA to study the feasibility of an underground nuclear waste disposal, see [11]. We have simplified the model by making all the subdomains homogeneous, but the numerical difficulty remains the flat aspect of the domain, and the high heterogeneity between the subdomains as the permeability jumps by a factor of several orders of magnitude from a geological layer to the other.

The domain is defined by a global mesh provided by ANDRA containing about 400000 hexahedric cells. It is decomposed following the geology into 12 subdomains with matching meshes at the interfaces, see Figure 2 on the left. The pressure solution is displayed on the right of the same Figure.

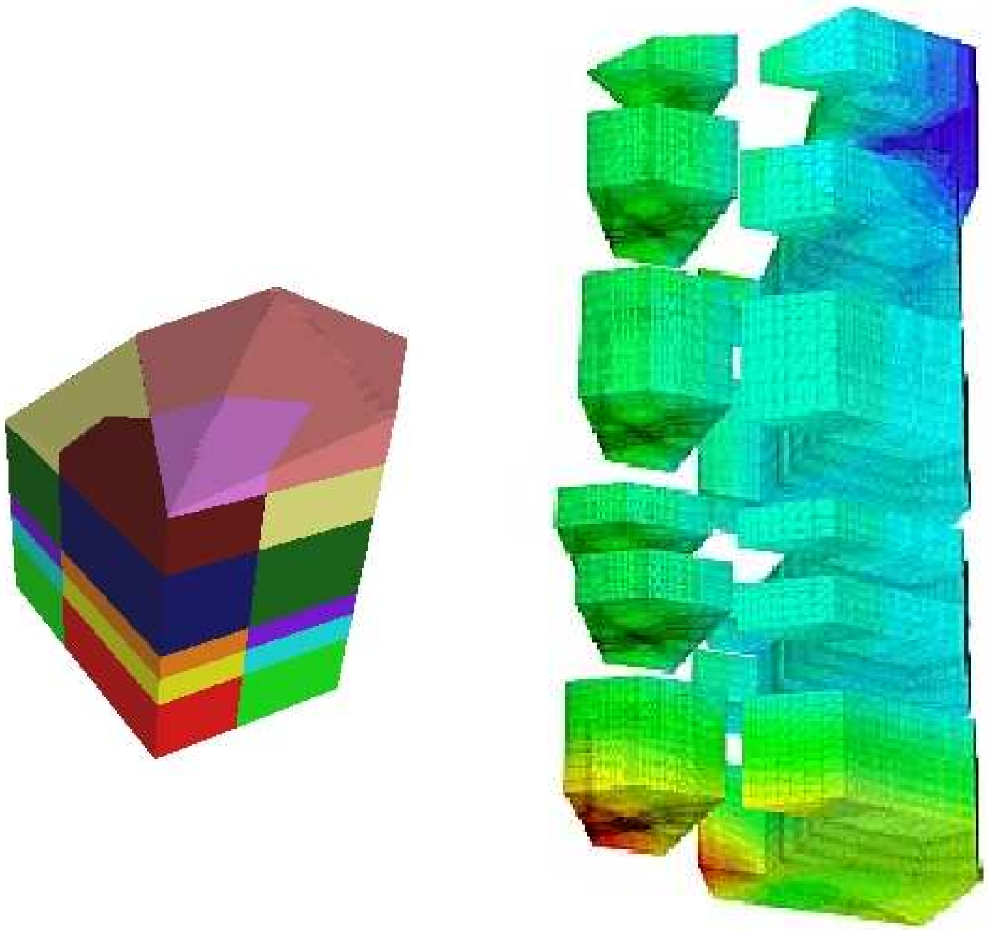

Figure 2: A realistic 3D flow simulation. On the left: the domain, seen from above. Unit on the vertical axis is multiplied by a factor of 100 (actually, the domain is flat). On the right: a split view of the pressure, seen from below. High loads are in red, and low values in blue.

The main difficulty here was to fine tune the Robin coefficients on each interface to insure convergence of the coupling algorithm. Our empirical tests based on the physics of the porous medium and the geometry of the domain led us to take into account the aspect ratio of the subdomains and the permeability jumps by choosing

$$\alpha_{ij} = \alpha_{ji} = \frac{2K_iK_j}{(K_i + K_j)L_{ij}^{\Omega}} \tag{14}$$

where L_{ij}^{Ω} is the characteristic length of the domain Ω along the normal direction to the interface Σ_{ij}.

5 CONCLUSIONS

We have presented a non-overlapping domain decomposition method for non-matching meshes based on Robin interface conditions for the computation of 3D flow in porous media. The algorithm has been fully detailed in the case of an arbitrary number of subdomains.

The main contribution of this paper is the way this algorithm is implemented. We have used the OCamlP3l parallel programming system that comes from the functional programming world. This system provides a structured approach to parallel programming obtained from skeletons and template based compilation techniques. The user describe its parallel algorithm by combining basic building blocks, then the (same) source code can be compiled either for sequential execution or for parallel execution. Because OCamlP3l is implemented in OCaml, a *safe* programming language, this opens the hope to prove an adequation theorem that will state that both executions will always produce the same result, for all possible user code. In short, this means that the user has never to take care of bugs specific to parallelism!

We presented extensibility results on an academic test case and numerical results on a realistic example of the parallel computation of 3D flow aiming at simulating the area surrounding a deep nuclear waste disposal.

We are now working on both numerical and programming aspects: on one side, we are implementing another domain decomposition method based on Neumann-Neumann preconditioning with balancing, and on the other side, we are developing a new `mapvector` skeleton with communication capabilities based on a connectivity table that will reduce the bottleneck problem.

6 ACKNOWLEDGMENTS

The authors thank Martial Mancip for providing an efficient implementation for a conservative L^2-projection routine.

This work has been partially supported by ANDRA, the French National Agency for radioactive waste management.

REFERENCES

[1] Y. Achdou, C. Japhet, Y. Maday, and F. Nataf. A new cement to glue non-conforming grids with Robin interface conditions: the finite volume case. *Numer.*

Math., 92(4):593–620, 2002.

[2] Todd Arbogast, Lawrence C. Cowsar, Mary F. Wheeler, and Ivan Yotov. Mixed finite element methods on nonmatching multiblock grids. *SIAM J. Numer. Anal.*, 37(4):1295–1315 (electronic), 2000.

[3] Todd Arbogast and Ivan Yotov. A non-mortar mixed finite element method for elliptic problems on non-matching multiblock grids. *Comput. Methods Appl. Mech. Engrg.*, 149(1-4):255–265, 1997. Symposium on Advances in Computational Mechanics, Vol. 1 (Austin, TX, 1997).

[4] C. Bernardi, Y. Maday, and A. T. Patera. A new nonconforming approach to domain decomposition: the mortar element method. In *Nonlinear partial differential equations and their applications. Collège de France Seminar, Vol. XI (Paris, 1989–1991)*, pages 13–51. Longman Sci. Tech., Harlow, 1994.

[5] F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods*. Springer-Verlag, Berlin, 1991.

[6] F. Clément, R. Di Cosmo, Z. Li, V. Martin, A. Vodicka, and P. Weis. Parallel programming with the OcamlP3l system. Applications to numerical code coupling. Rapport de Recherche 5131, Inria, Rocquencourt, France, 2004.

[7] Lawrence C. Cowsar, Jan Mandel, and Mary F. Wheeler. Balancing domain decomposition for mixed finite elements. *Math. Comp.*, 64(211):989–1015, 1995.

[8] Marco Danelutto, Roberto Di Cosmo, Xavier Leroy, and Susanna Pelagatti. Parallel functional programming with skeletons: the OCamlP3l experiment. *The ML Workshop*, 1998.

[9] V. Martin. *Simulations multidomaines des écoulements en milieu poreux*. PhD thesis, Université de Paris 9, France, 2004.

[10] J.-C. Nedelec. A new mixed finite elements in R^3. *Numer. Math.*, 50:57–81, 1986.

[11] G. Pépin, B. Vialay, and D. Perraud. Cahier des charges relatif à la réalisation de calculs de transport de radionucléides avec le code castem2000. Cahier des charges, Andra, Châtenay-Malabry, France, March 2001. In French.

[12] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, London, 2003.

[13] P.-A. Raviart and J.-M. Thomas. A mixed finite element method method for second order elliptic problems. In I. Galligani and E. Magenes, editors, *Mathematical Aspects of Finite Element Methods; Lecture Notes in Mathematics 606*, pages 292–315. Springer-Verlag, Berlin, 1977.

[14] J.E. Roberts and J.-M. Thomas. Mixed and hybrid methods. In P.G. Ciarlet and J.L. Lions, editors, *Handbook of Numerical Analysis Vol.II*, pages 523–639. North Holland, Amsterdam, 1991.