Parallel programming with the OcamlP3l system, with applications to coupling numerical codes.*

François Clément, Arnaud Vodicka
Project Estime - INRIA Rocquencourt - France

Roberto Di Cosmo, Zheng Li
Project Cristal - INRIA Rocquencourt University of Paris 7 - France

Vincent Martin
Project Estime - INRIA Rocquencourt and ANDRA - France

Pierre Weis
Projet Cristal - INRIA Rocquencourt - France

November 20, 2003

Abstract

Writing parallel programs is not easy, and debugging them is usually a nightmare. To cope with these difficulties, a structured approach to parallel programming using skeletons and templates based compilation techniques has been developed over the past years by several researchers, including the P3L group in Pisa. The OcamlP3l system marries the Ocaml functional programming language with the P3L skeletons, yielding a powerful parallel programming system and methodology: OcamlP3l allows the programmer to write and debug a *sequential* version of his program (which, if not easy, could be considered as routine), and then the parallel version is *automatically deduced* by recompilation of the source program. The invaluable advantage of this approach is stagging: the programmer has just to concentrate on the easy part, the sequential programming, relieving on the OcamlP3l system to obtain the hard part, the parallel version. As an additional benefit, the semantics adequacy of the sequential and parallel versions of the program is no more the programmer's concern: it is now the entire responsability of the OcamlP3l compiler.

In this paper, we report on the successful application of OcamlP3l in the field of scientific computing, where the system has been used to solve a problem of coupling numerical codes, obtaining parallelization for free.

The interaction has been quite successful, as, in the process of solving the coupling problem, a wealth of new ideas have emerged on the design of the system, which are now incorporated in the current version of OcamlP3l: *coloring* of virtual and physical computing network nodes to specify their relative mapping, and the new notion of *parfuns* or parallel computing subnetworks reified as functions at the programmer's level. Those two notions both increase efficiency and ease the writing of programs, being a step to smoother integration of parallel computing into the functional programming paradigm.

*The OcamlP3l system has been partially funded by a Galileo bilateral France-Italy project.

1 Introduction and Overview

In the field of Scientific Computing, one observes an increase in the complexity and size of the problems to solve and it becomes of major importance to consider the coupling of appropriate physical models. For instance, when dealing with underground storage, the transport equation, that describes the movement of particles in porous media, needs to be coupled with equations that model chemical reactions that can occur. In most cases, there exists different codes that treat separately the transport and the chemistry. One major issue is then to couple these existing codes, in order to avoid rewriting them from scratch.

In this paper, as a first step, we limit ourselves to a specific coupling: we couple a code with itself in a domain decomposition fashion. For that purpose, several key points have to be addressed: dealing with interface conditions of different nature, choice of a domain decomposition strategy, local refinements of the discretization in time and space, linking non-matching grids, ... Furthermore one has to take into account the design and implementation of the software interface. This latter problem could become cumbersome, especially when coupling codes written in various languages and with different kind of data structures. In addition, planning a parallel version of the application is another tedious issue.

Focusing on the software aspect, our long term goal is the automation of the implementation of the coupling interface between the different codes, or at least an help to write this interface, e.g., the design of a coupling library dedicated to Scientific Computing applications.

We have chosen the Ocaml language for its state-of-the-art high-level capabilities and the recently developed OcamlP3l system for *automagically* providing all the piping necessary to the communications between the different programs to couple. This means that we were already interested by the sequential execution of the application, and the parallel ability came for free!

We have first experienced the OcamlP3l environment on a very simple—easy-to-code—example implementing Schwarz algorithm (overlapping domain decomposition method with matching and structured grids based on simple Dirichlet conditions) for a 2D Poisson solver, details can be found in [CVDW03]. In the present paper, we consider a non-overlapping domain decomposition method with non-matching and unstructured grids based on Robin interface conditions applied to a problem of 3D flow simulation in porous media.

The paper is organized as follows. In the next section, we describe in details the current state of the OcamlP3l skeleton-based system. In particular, this includes lately developed features as the possibility to locally initialize the data parallel skeletons, the `parfun` construction that encapsulates any stream processor as an ordinary function, and the color option that allows to balance the load. We give a brief manual in section 3. The fourth section is dedicated to the presentation of a scientific computing application, the simulation of flow in 3D porous media, and its implementation within the framework of the OcamlP3l environment. We give some results that illustrate the new capabilities of the system in section 5. And then, we conclude and propose further developments. More details about the technique of domain decomposition are given in the appendix, as well as the interface of an Ocaml module dedicated to it.

2 The OcamlP3l system

2.1 Overview of the system

In OcamlP3l, as in all skeleton-based systems, the user describes the parallel structure of the computation by means of a set of skeletons. One distinctive feature of OcamlP3l, though, is that the semantics of these skeletons is not hard-wired: the system allows the user to compile his code, without any source modification whatsoever, using a choice of various possible semantics.

2.2 Three strongly related semantics

In OcamlP3l, as described in the seminal paper [DDCLP98], we provides three semantics, for any user program.

sequential semantics the user's program is compiled and linked against a sequential implementation of the skeletons, so the resulting executable can be run on a single machine, as a single process, and easily debugged using standard debugging techniques and tools for sequential programs,

parallel semantics the user's program is compiled and linked against a parallel implementation of the skeletons, and the resulting executable is a generic SPMD program that can be deployed on a parallel machine, a cluster, or a network of workstations,

graphical semantics the user's program is compiled and linked against a graphical implementation of the skeletons, so that the resulting program, when executed, displays a picture of the parallel computational network that is deployed when running the parallel version.

From the user's point of view, those three different semantics are simply obtained by compiling the program with three different options of the compiler.

Of course, our goal is to guarantee that the sequential and the parallel execution agree: for any user program the two semantics should produce exactly the same results.

2.3 Skeletons as stream processors

The OcamlP3l skeletons are *compositional*: the skeletons are combinators that form an algebra of functions and functionals that we call the *skeleton language*.

To be precise, 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. Those functions can then be composed arbitrarily, thus leading to trees of combinators that define the parallel behaviour of programs.

This mapping of skeletons to stream processors is evident at the type level, since the skeletons are all assigned types that reflect their stream processing functionality. Of course, the compositional nature of skeletons is also clear in their implementation:

For the parallel semantics implementation, a skeleton is realized as a stream processor parameterized by some other functions and/or other stream processors.

For the sequential semantics implementation, we provide an abstract data type of streams (the polymorphic `'a stream` data type constructor), and the sequential implementation of the skeletons is defined as a set of functions over those streams.

2.4 The skeleton combinators in OcamlP3l

In the current release of OcamlP3l, the combinators (or basic building blocks) of the skeleton language pertain to five kinds:

- the *task parallel* skeletons that model the parallelism of *independent* processing activities relative to different input data. In this set, we have *pipe* and *farm*, that correspond to the usual task parallel skeletons appearing both in p3l and in other skeleton models [Col89, DFH$^+$93, DGTY95].

- the *data parallel* skeletons that model the parallel computation of different parts of the same input data. In this set, we provide `mapvector` and `reducevector`. The `mapvector` skeleton models the parallel application of a generic function f to all the items of a vector data structure, whereas the `reducevector` skeleton models a parallel computation that folds the elements of a vector with a commutative and associative binary operator ($\oplus$).

  Those two skeletons are simplified versions of their respective `map` and `reduce` analogues in p3l. They provide a functionality quite similar to the map($*$) and reduce ($/$) functionals of the Bird-Meertens formalism discussed in [Bir87] and the map and fold skeletons of SCL [DGTY95].

- the *data interface* skeletons that provide injection and projection between the sequential and parallel worlds: `seq` converts a sequential function into a node of the parallel computational network, `parfun` converts a parallel computational network into a stream processing function.

- the *parallel execution scope delimiter* skeleton, the `pardo` combinator, that must encapsulate all the code that invokes a `parfun`.

- the *control* skeleton, the `loop` combinator, that provides the necessary repetitive execution of a given skeleton expression (`loop` is not a parallel construct *per se*).

2.5 Skeleton syntax, semantics, and types

We briefly describe here the syntax, the informal semantics, and the types assigned to each of the combinators of the skeleton language.

2.5.1 Combinators as skeleton generators

First of all, let's explain why the actual Ocaml types of our skeletons are a bit more complex than a naive view would have guessed. In effect, those types seem somewhat polluted by spurious additional `unit` types, compared to the types one would consider as natural. Of course, this additional complexity is not purely incidental: it has been forced by strong practical considerations to implement new functionalities that where mandatory to effectively run the numerical applications described below.

We now explain the rationality of these decisions for the simplest skeleton, the `seq` combinator. As explained above, `seq` encapsulates any Ocaml function f into a sequential process which applies f to all the inputs received in the input stream. Writing `seq f`, any Ocaml function with type `f : 'a -> 'b` is wrapped into a sequential process (this is reminiscent to the `lift` combinator used in many stream processing libraries of functional programming languages). Hence, we would expect `seq` to have the type

```
('a -> 'b) -> 'a stream -> 'b stream.
```

However, in OcamlP3l, `seq` is declared as

```
seq : (unit -> 'a -> 'b) -> unit -> 'a stream -> 'b stream
```

meaning that the lifted function argument `f` gets an extra `unit` argument. In effect, in real-world application, the user functions may need to hold a sizeable amount of local data, like those huge matrices that have to be initialised in the numerical application described further on, and we must allow the user to finely describe where and when those data have to be initialized and/or copied.

Reminiscent to partial evaluation and λ-lifting, we reuse the classical techniques of functional programming to initialize or allocate data globally and/or locally to a function closure. This is just a bit complicated here, due to the higher-order nature of the skeleton algebra, that in turn reflects the inherent complexity of parallel computing:

- *global initialization*: the data is initialised once and for all, and is then replicated in every copy of the stream processor that a `farm` or a `mapvector` skeleton may launch; this was already available in the previous versions of OcamlP3l, since we could write

  ```
  let f =
    let localdata = do_huge_initialisation_step () in
    fun x -> compute (localdata, x);;
  ...
  farm (seq f, 10)
  ```

- *local initialization*: the data is initialised by each stream processor, *after* the copy has been performed by a `farm` or a `mapvector` skeleton; this was just impossible in the previous versions of OcamlP3l; with the new scheme it is now easy:

  ```
  let f () =
    let localdata = do_huge_initialisation_step () in
    fun x -> compute (localdata, x);;
  ...
  farm (seq f, 10)
  ```

  when the `farm` skeleton creates 10 copies of `seq f`, each copy is created by passing () to the `seq` combinator, which in turn passes () to f, producing the allocation of a different copy of *localdata* for each instance[1]. Note also that the old behaviour, namely, a unique initialization shared by all copies, is still easy (and can be freely combined to further local initializations if needed):

  ```
  let f =
    let localdata = do_huge_initialisation_step () in
    fun () -> fun x -> compute (localdata, x);;
  ...
  farm (seq f, 10)
  ```

To sum up, the extra `unit` parameters give the programmer the hability to decide whether local initialisation data in his functions are shared among all copies or not. In other words, we can regard the skeleton combinators in the current version of OcamlP3l as "delayed skeletons", or "skeleton factories", that produce *an instance* of a skeleton every time they are passed an () argument.

[1] In practice, the initialization step may do weird, non referentially transparent things, like opening file descriptors or negociating a network connection to other services: it is then crucial to allow the different instances of the user's function to have their own local descriptors or local connections to simply avoid the chaos.

2.5.2 Typing and semantics of skeletons

We can now detail the other skeletons:

The farm skeleton computes in parallel a function f over different data items appearing in its input stream.
From a functional viewpoint, given a stream of data items $x_1, \ldots, x_n$, and a function f, the expression $\texttt{farm}(f, k)$ computes $f(x_1), \ldots, f(x_n)$. Parallelism is gained by having k independent processes that compute f on different items of the input stream.
If f has type `(unit -> 'b stream -> 'c stream)`, and k has type `int`, then $farm(f, k)$ has type `unit -> 'b stream -> 'c stream`.

The pipeline skeleton is denoted by the infix operator `|||`; it performs in parallel the computations relative to different stages of a function composition over different data items of the input stream.
Functionally, $f_1 \,|||\, f_2 \ldots |||\, f_n$ computes $f_n(\ldots f_2(f_1(x_i))\ldots)$ over all the data items x_i in the input stream. Parallelism is now gained by having n independent parallel 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}.
If f_1 has type `(unit -> 'a stream -> 'b stream)`,
and f_2 has type `(unit -> 'b stream -> 'c stream)`,
then $f_1|||f_2$ has type `unit -> 'a stream -> 'c stream`.

The map skeleton is named `mapvector`; it computes in parallel a function over all the data items of a vector, generating the (new) vector of the results.
Therefore, for each vector X in the input data stream, $\texttt{mapvector}(f, n)$ computes the function f over all the items of X, using n distinct parallel processes that compute f over distinct vector items.
If f has type `(unit -> 'a stream -> 'b stream)`, and n has type `int`, then $mapvector(f, n)$ has type `unit -> 'a array stream -> 'b array stream`.

The reduce skeleton is denoted `reducevector`; it folds a function over all the data items of a vector.
Therefore, $\texttt{reducevector}(f, n)$ computes $x_1 f x_2 f \ldots f x_n$ out of the vector $x_1, \ldots, x_n$, for each vector in the input data stream. The computation is performed using n different parallel processes that compute f.
If f has type `(unit -> 'a * 'a stream -> 'a stream)`, and n has type `int`, then
$reducevector(f, n)$ has type `unit -> 'a array stream -> 'a stream`.

2.6 The `parfun` construction

In the original p3l system, a program is clearly stratified into two levels: there is a skeleton *cap*, that can be composed of an arbitrary number of skeleton combinators, but as soon as one goes outside this cap, passing into the sequential code through the `seq` combinator, there is no way for the sequential code to call a skeleton. To say it briefly, the entry point of a p3l program *must* be a skeleton expression, and no skeleton expression is allowed anywhere else in the code.
This stratification is quite reasonable in the p3l system, as the goal is to build *a single* stream processing network described by the skeleton cap. However, it has several drawbacks:

- it breaks uniformity, since even if the skeletons *look like* ordinary functionals, they *cannot* be used as ordinary functions, in particular inside sequential code,

- as examplified in the application of section 4, many numerical algorithms boil down to simple nested loops, some of which can be easily parallelised, and some cannot; forcing the programmer to push all the parallelism in the skeleton cap could lead to rewriting the algorithm in a very unnatural way,

- as in our numerical application, a parallelizable operation can be used at several stages in the algorithm: the p3l skeleton cap does not allow the user to specify that parts of the stream processing network can be shared among different phases of the computation, which is an essential requirement to avoid wasting computational resources.

To overcome all these difficulties and limitations, the `1.9` version of OcamlP3l introduces the new `parfun` skeleton, the very *dual* of the `seq` skeleton. In simple words, one can wrap a full skeleton expression inside a `parfun`, and obtain a regular stream processing function, usable with no limitations in any sequential piece of code: a `parfun` encapsulated skeleton behaves exactly as a normal function that receives a stream as input, and returns a stream as output. However, in the parallel semantics, the `parfun` combinator gets a parallel interpretation, so that the encapsulated function is actually implemented as a parallel network (the network to which the `parfun` combinator provides an interface).
Since many parfun expressions may occur in a OcamlP3l program, there may be several disjoint parallel processing networks at runtime. This implies that, to constrast with p3l, the OcamlP3l model of computation requiers a *main* sequential program: this main program is responsible for information interchange with the various `parfun` encapsulated skeletons.

One would expect `parfun` to have type `(unit -> 'a stream -> 'b stream) -> 'a stream -> 'b stream`: given a skeleton expression with type `(unit -> 'a stream -> 'b stream)`, `parfun` returns a stream processing function of type `'a stream -> 'b stream`.

`parfun`'s actual type introduces an extra level of functionality: the argument is no more a skeleton expression but a functional that returns a skeleton:

```
val parfun :
  (unit -> unit -> 'a stream -> 'b stream) -> 'a stream -> 'b stream
```

This is necessary to guarantee that the skeleton wrapped in a `parfun` expression will only be launched and instanciated by the main program, not by any of the multiple running copies of the SPMD binary, even though thoses copies may evaluate the `parfun` skeletons; the main program will actually create the needed skeletons by applying its functional argument, while the generic copies will just throw the functional away, carefully avoiding to instanciate the skeletons.

2.7 The `pardo` parallel scope delimiter

`pardo` typing

Finally, the `pardo` combinator defines the scope of the expressions that may use the `parfun` encapsulated expressions.

```
val pardo : (unit -> 'a) -> 'a
```

`pardo` takes a thunk as argument, and gives back the result of its evaluation. As for the `parfun` combinator, this extra delay is necessary to ensure that the initialization of the code will take place exclusively in the main program and not in the generic SPMD copies that participate to the parallel computation.

Parallel scoping rule

The scoping rule has three requisits:

- functions defined via the `parfun` combinator must be *defined before* the occurrence of the `pardo` combinator,
- those `parfun` defined functions can only be *executed within* the body of the functional parameter of the `pardo` combinator,
- no `parfun` can occur as a sub tree of a `pardo` combinator.

2.7.1 Structure of an OcamlP3l program

Due to this scoping rule, the general structure of an OcamlP3l program looks like the following:

```
(* (1) Functions defined using parfun *)
let f = parfun(skeleton expression)
let g = parfun(skeleton expression)

(* (2) code referencing these functions under abstractions *)

let h x = ... (f ...) ... (g ...) ...

...
(* NO evaluation of code containing a parfun
   is allowed outside pardo *)

(* (3) The pardo occurrence where parfun encapsulated
       functions can be called. *)
pardo
 (fun () ->
    (* NO parfun combinators allowed here *)

    (* code evaluating parfun defined functions *)
    ...
    let a = f ...
    let b = h ...
    ...
 )
(* finalization of sequential code here *)
```

At run time, in the sequential model, each generic copy just waits for instructions from the main node; the main node first evaluates the arguments of the `parfun` combinators to build a representation of the needed skeletons; then, upon encountering the `pardo` combinator, the main node initializes all the parallel computation networks, specialising the generic copies (as described in details in [DDCLP98]), connects these networks to the sequential interfaces defined in the `parfun`'s, and then runs the sequential code in its scope by applying its function parameter to `():unit`. The whole picture is illustrated in Figure 1. The skeleton networks are initiated only once but could be invoked many times during the execution of `pardo`.

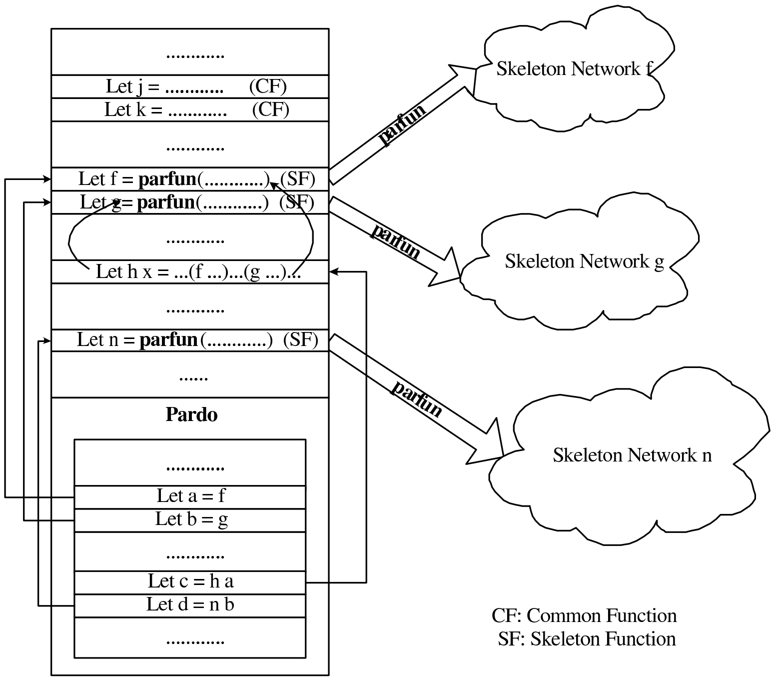

Figure 1: parfun and pardo: the overall structure

2.8 Load balancing: the colors

In the OcamlP3l system, the combinators expressions govern the shape of the virtual processor network that is entirely designed during compilation. However, the mapping of these virtual nodes to physical processors is thus delegated to the OcamlP3l system. This automatic mapping is not realistic and we describe here a new feature, the *colors*, that allow the programmer to annotate trees of combinators to specify virtual-to-physical mapping requests that drive the system automatic strategy. Pushing the difficult part of the mapping specification to the programmer's knowledge and ability, this simple and practical idea gives surprisingly good results in practice.

Let's consider as an example, the expression
`farm` (`seq` $(fun\ x \rightarrow x\ *\ x), 16)$
that corresponds to a network of one emitter node, one collector node, and 16 worker nodes which compute the square function. Such a computational structure is clearly shown as the result of the execution of the graphical semantics of the program. There are numerous ways of mapping a set of virtual nodes to a set of physical nodes. Which is the best mapping and how to obtain it from the compiler is still unclear at this stage: for the time being, the process is completely automatic and the programmer has no control over the strategy used by the system to found a good mapping.

The simplest automatic solution is a round robin algorithm which maps a line of virtual nodes and a line of physical nodes, one by one, until all virtual nodes are allocated. If the line of physical nodes is exhausted first, allocation starts again from the beginning of the physical line as if the line were circular. Unfortunately, such a solution doesn't take into account the load balancing constraints: all the physical (resp. virtual) nodes are considered equivalent in computing power and are used evenly. But this is not realistic, since the programmer does know that some virtual nodes will have a huge amount of computation to perform, while some others will not; the programmer also knows that those physical nodes (probably being different machines) also vary a lot in their capability. So we need a programatic way to pair the virtual nodes with heavy computational work to powerful physical nodes. We introduce the notion of *color* as a general method to specify the relative capability ranks of both virtual and physical processors.

A *c*olor is an optional integer parameter that is added to OcamlP3l expressions in the source program and to the execution command line of the compiled program. We use the regular Ocaml's optional parameters syntax, with keyword *col*, to specify the colors of a network of virtual nodes. For example, writing `farm ~col:k (f, n)` means that all virtual nodes inside this farm structure should be mapped to some physical nodes with a capability ranking k. The scope of a color specification covers all the inner nodes of the structure it qualifies: unless explicitely specified, the color of an inside expression is simply inherited from the outer layer (the outermost layer has a default color value of 0 which means no special request).

For combinators `farm`, `mapvector` and `reducevector`, in addition to the color of the combinator itself, their is an additional optional color parameter *colv*. A *colv*[ ] specification is a `color list` (i.e. an `int list`) that specifies the colors of the parallel worker structures that are arguments of the combinator. As an example, the OcamlP3l expression

```
map ~col:2 ~colv:[ 3; 4; 5; 6 ] (seq f, 4)
```

is a `mapvector` skeleton expression, with emitter and collector nodes of rank 2, and four worker nodes (four copies of `seq f`) whith respective ranks 3, 4, 5, and 6.

To carefully map virtual nodes to physical nodes, we also need a way to define the colors of physical nodes. This information is specified on the command line when launching the program. One can write:

```
prog.par -p3lroot ip1:port1#color1 ip2:port2#color2 ... \
                  ip_i:port_i#color_i ...
```

where `ip_i:port_i#color_i` indicates the ip address (or name), the port, and the color of the physical node `i` participating to the computation. The port and color here are both optional. With no specified port, a default p3lport is used; with no color specification, the default color 0 is assumed.

If the colors of all the virtual processors and all the physical processors have a one-to-one correspondance, the mapping is easy. But such a perfect mapping does not exist in general: first of all, there is not always equality between the amount of physical processors we have and the amount of virtual processors we need; second, in some very complex OcamlP3l expressions, it is complex and boring for the programmer to calculate manually how many virtual nodes are needed for each color class.

So, we decided to use a simple but flexible mapping algorithm, based on the idea that *what a color means is not the **exact** capability required but the **lowest** capability acceptable.* For example, a virtual node with color value 5 means a physical node of color 5 is needed, but if there is no physical node with value 5, and there exists a physical node of color 6 free and available, why don't we take it instead? In practice, we sort the virtual nodes in decreasing order of their color values, to reflect their priority in choosing a physical node: virtual nodes with bigger colors should have more privilege and choose their physical node before the nodes with smaller colors. Then, for each virtual node, we lists all the physical nodes with a color greater than or equal to the virtual node color. Among all those qualified ones, the algorithm finally associates the virtual node with the qualified node which has the smallest work load (the one that has the least number of virtual nodes that have been assigned to it).

This algorithm provides a mapping process with some degree of automatization and some degree of manual tuning, but one has to keep in mind that the *color* designs a computational class, qualitatively, and is not an exact quantitative estimation of the computational power of the machine, as the current version of OcamlP3l does not provide yet the necessary infrastructure to perform an optimal mapping based on precise quantitative estimations of the cost of each sequential function and the capabilities of the physical nodes, so that we cannot guarantee our color-based mapping algorithm to be highly accurate or highly effective.

Still, the "color" approach is accurate and simple enough to be quite significant to the programmer: according to the experiments we have conducted, it indeed achieved some satisfactory results in our test bed case (see section 5).

Figure 2 summarizes the types of the combinators, exactly as they are currently available to the programmer in the `1.9` version of OcamlP3l, including the optional color parameters.

```
type color = int

val seq :
  ?col:color ->
  (unit -> 'a -> 'b) -> unit -> 'a stream -> 'b stream
val ( ||| ) :
  (unit -> 'a stream -> 'b stream) ->
  (unit -> 'b stream -> 'c stream) -> unit -> 'a stream -> 'c stream
val loop :
  ?col:color ->
  ('a -> bool) * (unit -> 'a stream -> 'a stream) ->
  unit -> 'a stream -> 'a stream
val farm :
  ?col:color ->
  ?colv:color list ->
  (unit -> 'b stream -> 'c stream) * int ->
  unit -> 'b stream -> 'c stream
val mapvector :
  ?col: color ->
  ?colv:color list ->
  (unit -> 'b stream -> 'c stream) * int ->
  unit -> 'b array stream -> 'c array stream
val reducevector :
  ?col:color ->
  ?colv:color list ->
  (unit -> ('b * 'b) stream -> 'b stream) * int ->
  unit -> 'b array stream -> 'b stream
val parfun :
  (unit -> unit -> 'a stream -> 'b stream) -> 'a stream -> 'b stream
val pardo : (unit -> 'a) -> 'a
```

Figure 2: The types of the OcamlP3l skeleton combinators

3 Using the system

Compared with most parallel programming system, the coding, debugging and execution of OcamlP3l programs are simpler and more elegant.

As mentioned in previous sections, OcamlP3l provides three strongly related semantics: sequential, parallel and graphical semantics. In practice, the programmer can compile any OcamlP3l program into three different but semantically related executable codes by specifying different compiling options to the `ocamlp3lcc` compiler driver, without modifying her source code at all.

- With the option `-seq`, the source files is compiled into sequential code running on a single machine. By invoking the command

  ```
  ocamlp3lcc -seq sourcename.ml
  ```

  we get an executable file called `sourcename.seq` that can be directly run on a single machine, tested and debugged using the regular OCaml source level debugger as any other sequential program.

- With the `-gra` option, we get an executable file which draws the skeleton network specified by the program expressions. Typing the command

  ```
  ocamlp3lcc -gra sourcename.ml
  ```

  we obtain `sourcename.gra` whose execution displays in a graphical window the parallel networks used by the program.

- Finally, the parallel code version is obtained with the option `-par`. The compilation command is

  ```
  ocamlp3lcc -par sourcename.ml
  ```

  that produces `sourcename.par`, the SPMD generic executable that will be deployed on all the nodes participating in the computation. One copy of this executable is launched on all the working processors, and waits for configuration information which is sent by a designated root node, which will also run the `pardo` encapsulated sequential code. The designated root node is just another copy of the same executable that is simply run with the specific command line option `-p3lroot`. In addition the root node receives a list of arguments that specifies all needed information about the nodes involved in the computational network (their ip address or name, their port and color), together with some other optional running parameters. In short, the command should spell like

  ```
  sourcename.par -p3lroot nodes_info_list
  ```

  An example has already been given in section 2.8 where we described how colors are assigned to the physical nodes.

The implementation in these three semantics greatly facilitates the development of parallel applications. The parallel implementation is the final goal of the development, but the programming, and especially the debugging of the parallel application, is awfully complicated when compared to the same tasks in the sequential world. This is why in the OcamlP3l phylosophy, development and debugging starts on a sequential prototype, where the programmer can focus on the algorithmic aspects of the *function* to compute, and then moves on to the parallel implementation, relying on the equivalence between sequential and parallel semantics. Indeed, an OcamlP3l program compiled and executed successfully in sequential mode should always get success in the parallel mode.

Sequential development and parallel execution is one of the most distinguished and powerful features of OcamlP3l, and this is why we plan in the future to prove *formally* the equivalence between these two semantics. This task is far from easy, but the outcome is very well worth the effort, as we will then get for free the equivalence between parallel and sequential semantics for *all* user programs.

Finally, the graphic semantics is also helpful, as it allows to get an intuitive view of the whole parallel structure of the parallel program directly from the sources, and this information is precious to tune the mapping from virtual to physical nodes, and to help in the design and analysis of the parallel application.

Several tools are also provided to help programing in OcamlP3l system. Such as the `ocamlp3lrun` which is developed for the automatic distribution of the parallel code copies to a set of machines, the `ocamlp3lps` to print PIDs of all processes running OcamlP3l programs on the net, the `ocamlp3lkiller` to terminate all the processes executing an OcamlP3l program, and many others that we do not detail here.

But the execution model is so simple, that the user can very often write a small script devoted to run the program at hand on the specific hardware configuration available: as an example, we present in Figure 3 the short shell script that we use to launch the computation detailed in the next section in parallel on 8 nodes of a cluster, the first one being slower than the others.

```
#!/bin/sh

NODES="clus-101 clus-102 clus-103 clus-104 \
       clus-105 clus-106 clus-107 clus-108"
PAR="./coupling_subdomains.par"

echo -n "Launching P3L amorphous nodes on the cluster:"
for NODE in $NODES; do
    echo -n " $NODE"
    ssh $NODE $PAR 1> log-$NODE 2> err-$NODE &
    case $NODE in
        clus-101) COLORED_NODES="$COLORED_NODES $NODE#1";;
        *) COLORED_NODES="$COLORED_NODES $NODE#2";;
    esac
done

echo "Starting computation with $COLORED_NODES..."
$PAR -p3lroot $COLORED_NODES 1> log-root 2> err-root

echo "Finished."
```

Figure 3: A simple script to handle program launch and termination.

4 A Scientific Computing application

We report now our experiment devoted to the implementation using the OcamlP3l environment of a non trivial scientific computing application: the parallel simulation of flow in 3D porous media.

4.1 The coupling problem

Let us first present the main organization of the application to implement. The reader will find in appendix A some hints about the domain decomposition method we used: the non-overlapping non-conforming domain decomposition method based on Robin interface conditions. The interested reader may also refer to [AY97] and [AJMN02], where the method was first introduced and analyzed, or to [CMV$^+$03] for further details about the implementation aspects.

In the next section 4.1.1, we briefly explain to a reader unfamiliar with finite element discretization, how one can obtain a linear system such as (1) whose solution is an approximate representation of the fields we are looking for. In section 4.1.2, we also briefly describe the coupling between subdomains method. A zoology of notations and operators is introduced that a hurried reader might ignore, jumping directly to the equation (10) at the end of section 4.1.2. This equation (10) is the final linear system that we want to solve, where A is a block sparse matrix and λ is a block vector of the form $\lambda = (\lambda_0, \dots, \lambda_{n-1})$ with $\lambda_i = (\lambda_{ij_1}, \dots, \lambda_{ij_{n_i}})$.

4.1.1 The problem of 3D flow simulation in porous media

The 3D flow simulation problem, amongst almost every problems from the physics, can be set as a Partial Derivative Equation (PDE) problem in a domain $\Omega \subset \mathbb{R}^3$ with boundary conditions on $\Gamma = \partial\Omega$ for which we have to search for functions depending on the coordinate variables (space and/or time), here a pressure field and a velocity field depending on the 3D space coordinates, see appendix A.

To numerically solve such a PDE problem on a computer, we have to choose approximation spaces in which we search for discrete approximations of the unknown functions. These finite dimension spaces are built from a 3D mesh associated with the domain Ω and are the so-called Finite Element spaces. After discretization, we obtain a linear system of the form

$$Lv = g, \tag{1}$$

where L is a—generally huge—sparse matrix, the unknown v is a vector collecting the degrees of freedom for the approximation of the pressure and the velocity and the right-hand side g is another vector collecting the known data.

Common algorithms to solve the linear system (1) have a complexity of $O(N^2)$ and memory requirements of $O(N^{\frac{4}{3}})$ where N is the dimension of the unknown v, see [Dem97]. Thus, it is interesting to cut the original domain into pieces, and solve—coupled—similar subproblems of smaller size. Therefore, the domain decomposition method consists in working with an auxiliary unknown set only at the interfaces between the subdomains. Of course, solving the subproblems in parallel brings another range of gain.

4.1.2 The coupling technique

We suppose now that the domain Ω is decomposed into n non-overlapping subdomains Ω_i, with $i \in I_n = \{0, 1, \dots, n-1\}$, and we denote by $\Gamma_i = \partial\Omega_i \cap \Gamma$ the—external—part of the boundary of Ω_i in common with the boundary of Ω. Let $\Sigma_i = \partial\Omega_i \backslash \Gamma$ be the *internal boundary* of Ω_i and let $\Sigma = \bigcup_{i \in I_n} \Sigma_i$ be the *structure* of the decomposition. Then, we can define the *interface* between subdomains Ω_i and Ω_j as $\Sigma_{ij} = \Sigma_{ji} = \Sigma_i \cap \Sigma_j$. When Σ_{ij} is neither empty nor reduced to a point or a line, the two subdomains are called *neighbors*. We denote the number of neighbors of subdomain Ω_i by n_i and the set of their indices by $\mathcal{N}_i = \{j_1, j_2, \dots, j_{n_i}\}$. See Figure 4 for an illustration. The set of all couples of neighbors, called *connectivity table*, is denoted by $\mathcal{N} = \{(i,j)/i \in I_n,\ j \in \mathcal{N}_i\}$; it is naturally ordered by the

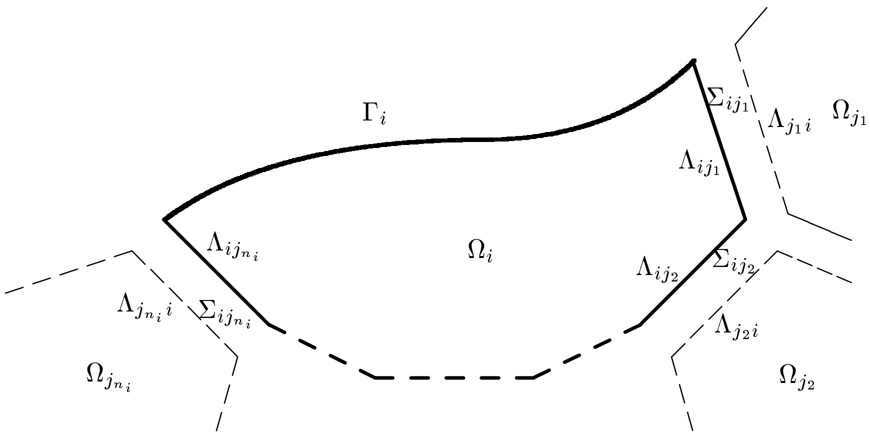

Figure 4: The subdomain Ω_i and its n_i neighbors Ω_{j_k}, $j_k \in \mathcal{N}_i$. The approximation spaces Λ_{ij_k} and $\Lambda_{j_k i}$ on each side of the interface Σ_{ij_k} are different. For the sake of simplicity, the picture has been drawn in two dimensions.

lexicographic order on $\mathbb{N}^2$. Let s be the permutation involution of $\mathcal{N}$ defined by

$$s : (i,j) \in \mathcal{N} \longmapsto (j,i) \in \mathcal{N}. \tag{2}$$

Given a 3D mesh associated with Ω_i, we can build 2D meshes associated with all the interfaces of the internal boundary Σ_i simply by collecting the trace of the 3D mesh on the Σ_{ij}'s. And, for each interface $\Sigma_{ij} = \Sigma_{ji}$, we can use the two 2D meshes on each side to build two 2D approximation spaces denoted by Λ_{ij} and Λ_{ji}. But, the two 3D meshes associated with two neighboring subdomains may have been built separately, thus they can be non matching at the interface and the two 2D meshes on each side may be different. Hence, we generally have $\Lambda_{ij} \neq \Lambda_{ji}$ (see Figure 4) and then, it is useful to define the projection $P_{i\to j}$ from Λ_{ij} onto $\Lambda_{ji} = \tilde{\Lambda}_{ij}$ such that

$$\forall \lambda_{ij} \in \Lambda_{ij} \quad \forall \tilde{\mu}_{ij} \in \tilde{\Lambda}_{ij}, \quad \langle \lambda_{ij} - P_{i\to j}\lambda_{ij}, \tilde{\mu}_{ij} \rangle_{ij} = 0, \tag{3}$$

where $\langle \lambda, \mu \rangle_{ij} = \int_{\Sigma_{ij}} \lambda(x)\mu(x)\,dx$ is the L^2 dot product between functions defined on Σ_{ij}. We set[2]:

$$\begin{aligned} \Lambda_i &= \bigoplus_{j\in\mathcal{N}i} \Lambda_{ij}, & \Lambda &= \bigoplus_{i\in I_n} \Lambda_i = \bigoplus_{(i,j)\in\mathcal{N}} \Lambda_{ij}, \\ \tilde{\Lambda}_i &= \bigoplus_{j\in\mathcal{N}i} \tilde{\Lambda}_{ij} = \bigoplus_{j\in\mathcal{N}i} \Lambda_{ji}, & \tilde{\Lambda} &= \bigoplus_{i\in I_n} \tilde{\Lambda}_i = \bigoplus_{(i,j)\in s(\mathcal{N})} \Lambda_{ij}. \end{aligned} \tag{4}$$

The vectors λ_{ij} of Λ_{ij} are called *interface values*. Similarly, the λ_i's (resp. $\tilde{\lambda}_i$'s) are called *inner* (resp. *outer*) *internal boundary values* and the λ's (resp. $\tilde{\lambda}$'s) *inner* (resp. *outer*) *structure values*. By extension, we still denote by s the permutation involution operator mapping Λ onto $\tilde{\Lambda}$.

[2]Take care to the different ordering of the—same—components in Λ and $\tilde{\Lambda}$. For example, with three subdomains neighboring each other, we simply have

$$\lambda = ((\lambda_{01}, \lambda_{02}), (\lambda_{10}, \lambda_{12}), (\lambda_{20}, \lambda_{21})) \quad \text{and} \quad s(\lambda) = \tilde{\lambda} = ((\lambda_{10}, \lambda_{20}), (\lambda_{01}, \lambda_{21}), (\lambda_{02}, \lambda_{12})).$$

Let R_i (resp. $\tilde{R}_i$) be the restriction operator from Λ onto Λ_i (resp. from $\tilde{\Lambda}$ onto $\tilde{\Lambda}_i$). Their transposed are the reconstruction operators, thus

$$(5)\qquad \begin{array}{lrcl} R_i: & (\lambda_0,\ldots,\lambda_{n-1})\in\Lambda & \longmapsto & \lambda_i\in\Lambda_i \\ \tilde{R}_i^T: & \tilde{\lambda}_i\in\tilde{\Lambda}_i & \longmapsto & (0,\ldots,\tilde{\lambda}_i,\ldots,0)\in\tilde{\Lambda}, \end{array}$$

let P_i be the projection operator from Λ_i onto $\tilde{\Lambda}_i$ define from the $P_{i\to j}$'s above by

$$(6)\qquad P_i : (\lambda_{ij_1},\ldots,\lambda_{ij_{n_i}})\in\Lambda_i \longmapsto (P_{i\to j_1}\lambda_{ij_1},\ldots,P_{i\to j_{n_i}}\lambda_{ij_{n_i}})\in\tilde{\Lambda}_i.$$

Let S_{g_i} be the Robin-to-Robin interface operator defined by, see appendix A,

$$(7)\qquad S_{g_i} : \lambda_i\in\Lambda_i \longmapsto \mu_i = \mu_i(v_i)\in\Lambda_i,$$

where μ_i depends explicitly on the solution v_i of the *inner* linear system,

$$(8)\qquad L_i v_i = (g_i, \lambda_i).$$

Matrix L_i and right-hand side g_i are the similar counterparts of L and g appearing in (1) but set on the subdomain Ω_i.

We also define the global operators

$$(9)\qquad P = \sum_{i\in I_n} \tilde{R}_i^T P_i R_i : \Lambda \longrightarrow \tilde{\Lambda} \quad \text{and} \quad S_g = \sum_{i\in I_n} R_i^T S_{g_i} R_i : \Lambda \longrightarrow \Lambda.$$

Then, the domain decomposition method amounts to replace the linear system (1), set in the 3D domain Ω, by an—equivalent—linear system, set on all the 2D interfaces of the decomposition of Ω into subdomains, i.e. on the structure Σ, that we call the *outer* linear system, see appendix A. Thus, we search now for a vector $\lambda\in\Lambda$ solving

$$(10)\qquad A\lambda = b$$

where the nonsymmetric matrix A and the right-hand side b are given by

$$(11)\qquad A = \mathrm{Id} - sPS_0 \qquad \text{and} \qquad b = sPS_g 0.$$

4.1.3 Codes to couple

To implement the coupling, we have access to three C++ codes.

The build_interface_meshes **code** inputs the name of the file describing the 3D mesh associated with a subdomain, say of number i. It reads this file from the disk, computes all the 2D meshes for the approximation spaces Λ_{ij}, $j\in\mathcal{N}_i$ and then writes the files describing these 2D meshes on the disk. It outputs nothing.
It needs no I/O redirections.
Parallel execution for all subdomains may be balanced according to the size of the 3D meshes, e.g. with the color feature of section 2.8.

The build_projection_matrix **code** inputs a couple of names of the two files describing the two 2D meshes on the same interface, say of numbers i and j. It reads these files from the disk, computes and finally outputs the corresponding projection matrix $P_{i\to j}$.
It needs redirection of its standard output.
Parallel execution for all couples of neighboring subdomains may be balanced according to the product of the size of the 2D meshes on both sides of all interfaces, with the color feature of section 2.8.

The solve_on_a_subdomain code inputs the name of the file describing the 3D mesh associated with a subdomain, say of number 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 L_i[3], then computes and outputs $S_{g_i}0_i = \mu_i(L_i^{-1}(g_i, 0_i))$ needed for the computation of the right-hand side b.

When given the keyword "loop", it inputs λ_i, then computes and outputs $S_{0_i}\lambda_i = \mu_i(L_i^{-1}(0_i, \lambda_i))$ needed for the computation of the matrix-vector product.

When given the keyword "final", it inputs $\lambda_i^\star$, then computes and writes on the disk $v_i^\star = L_i^{-1}(g_i, \lambda_i^\star)$.

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*, with the skeleton generator feature of section 2.5.1, and has also to have the ability to be recalled, by having its I/O channels stored.
Parallel execution for all subdomains *has to* be balanced according to the size of the 3D meshes, again with the color feature of section 2.8, as it is important to allow a fine tuning of the memory swapping difficulties, see section 5.

Since the matrix A of the outer linear system (10) is nonsymmetric, it is advisable to accelerate the convergence with a nonsymmetric Krylov method, e.g. Bi-CGStab [van92] or GMRes [SS86]. Such an iterative algorithm, here Bi-CGStab, only needs the axpy and dot routines[4] from the BLAS—Basic Linear Algebra Subprograms—library and a—preferably matrix-free—matrix-vector product routine, denoted here aax, that computes the action of matrix A on any vector λ (and, of course, the right-hand side b and an initial guess λ^0).

Rather than coding the whole Bi-CGStab algorithm with OcamlP3l skeletons, it is very interesting to define the sole aax routine as a stream processing network that can be used as an *ordinary function*, i.e. with the `parfun` skeleton, see section 2.6. Furthermore, we will keep the ability to change the algorithm, e.g. for GMRes, without recoding the aax routine.

4.1.4 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 L_i^{-1} of the matrices of the inner systems for all subdomains.
- define aax that applies in parallel matrix $A = \mathrm{Id} - sPS_0$ to any vector λ.
- compute in parallel the right-hand side $b = sPS_g0$.
- choose $\lambda^0 = 0$.

[3] Actually, only a sparse LU factorization is performed.

[4] axpy computes the vector $a.x + y$ from the scalar a and the vectors x and y and dot computes the—scalar—dot product $\langle x, y\rangle$ from the vectors x and y.

- choose an algorithm to solve the outer system, e.g. Bi-CGStab.

Iteration

- run the algorithm with the parallel matrix-vector product aax, the right-hand side b and the initial guess λ^0.
- call the solution $\lambda^\star$.

Finalization

- solve in parallel the inner systems associated with $S_g \lambda^\star$ and store the $v_i^\star$'s for all subdomains.

4.2 The OcamlP3l implementation

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

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

```
(* we have emphasized the OcamlP3l skeletons *)
let spawn_it command cin cout =
  match !cin, !cout with
  | Some ic, Some oc -> ic, oc
  | _ -> Printf.printf "Calling %s\n" command;
    let ic, oc = Unix.open_process command in
    cin := Some ic; cout := Some oc;
    ic, oc;;

let encapsulate_mapvector colv prog n =
  let body _ = mapvector ~colv:colv (seq prog, n) in
  let network = parfun body in
  (fun v ->
     let s = network (P3lstream.of_list [v]) in
     List.hd (P3lstream.to_list s));;

let build_all_interface_meshes =
  let colv = Ddec.colv_sorting_subdomains in
  let prog =
    (fun _ -> fun i ->
       let file = Ddec.filename_of_3D_mesh_number i in
       let command = "build_interface_meshes "^file in
       Sys.command command) in
  encapsulate_mapvector colv prog Ddec.number_of_processors;;

let build_all_projection_matrices =
  let colv = Ddec.colv_sorting_connectivity_table in
  let prog =
    (fun _ -> fun i -> fun 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
        read_projection_matrix ic) in
    encapsulate_mapvector colv prog Ddec.number_of_processors;;

  let solve_on_all_subdomains =
    let colv = Ddec.colv_sorting_subdomains in
    let prog _ =
      let cin = ref None and cout = ref None in
      (fun (i, tagged_lambda_i) ->
         let file_i = Ddec.filename_of_3D_mesh_number i in
         let command = "solve_on_a_subdomain "^file_i in
         let ic, oc = spawn_it command cin cout in
         Ddec.print_tagged_internal_boundary_values
           oc tagged_lambda_i;
         flush oc;
         let mu_i = Ddec.read_internal_boundary_values ic in
         Ddec.iter_vector_of mu_i) in
    encapsulate_mapvector colv prog Ddec.number_of_processors;;

  pardo (fun () ->
    Printf.printf "Beginning of 3D coupling\n";
    (* 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 mu = solve_on_all_subdomains v in
        let mu_tilde = projection_of mu in
        permutation_of mu_tilde) in
    let aax =
      (fun lambda ->
         let v = Ddec.iter_vector_of lambda in
         let mu_double_tilde = f_of v in
         (Ddec.axpy (-.1.) mu_double_tilde lambda)) in
    let b =
      let v = Ddec.init_vector_of_size n in
      f_of v in
    let lambda0 = Ddec.zero_structure_values n in
    let algorithm = Bicgstab.algorithm Ddec.axpy Ddec.dot in
    (* Iteration *)
    let lambda_star = algorithm aax b lambda0 in
    (* Finalization *)
    let v = Ddec.final_vector_of lambda_star in
    solve_on_all_subdomains v;
    Printf.printf "Ending of 3D coupling\n"
  );;
```

We can get automatically the graphics describing the parallel structure from the graphics semantics (Figure 5), and immediately see the three independent parallel computation networks that the system is using. One of them is heavily used in the code of the Bi-CGStab solver, which is standard sequential code, not shown here, calling the parallel network as a function along its iterations.

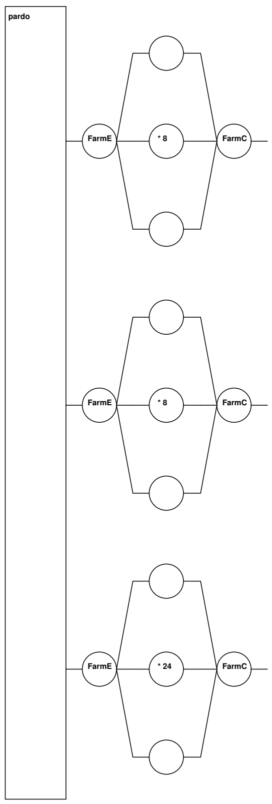

Figure 5: The parallel structure of `coupling.ml`. Configuration with 8 subdomains and 24 interfaces. See the decomposition in Figure 6

The function `spawn_it` (lines 2–8) has type `string -> in_channel option ref -> out_channel option ref -> in_channel * out_channel`. It allows to spawn processes and to collect once the channels connected to both their standard input and standard output, and still keeping the ability to feed and listen to them again later.

The function `encapsulate_mapvector` (lines 10–15) has type
`int array -> (unit -> 'a -> 'b) -> int -> 'a array -> 'b array`. It returns a stream processing network that encapsulates 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, i.e. inside any Ocaml function.

The functions `build_all_interface_meshes` (lines 17–24),
`build_all_projection_matrices` (lines 26–36) and `solve_on_all_subdomains` (lines 38–51) correspond to the three codes to couple of section 4.1.3. They are all defined through the `encapsulate_mapvector` network factory, with the color feature to balance their loads, but using ad'hoc techniques to connect —or not— their standard I/O's. Only the third one uses the scheme of section 2.5.1 for local initialization[5].

[5] Here an _ generalizes the `unit:()` for further developments of OcamlP3l.

The algorithm is implemented inside the `pardo` scope delimiter (lines 53–85). It uses the three previously defined stream processing networks, the third one been repeatedly employed. The function `f_of` (lines 63–67) defines the function $f = sPS_g$ from equation (20) for which we are searching the fixed point. It is used in both the bodies of the matrix-vector product `aax` (lines 68–72) and of the right-hand side `b` (lines 73–75).

5 Evaluation of the results

The domain decomposition coupling using OcamlP3l was tested on academic problems. The goal was to show the efficiency both in terms of development delays and in terms of CPU time needed to solve a model problem.

About the first issue, what certainly remains the climax of this collaboration was the first parallel run on our cluster. The very first time the code ran correctly in the sequential mode on our development Intel/PC station under Linux, we decided to test immediately on the newly arrived cluster of Intel/PC nodes also under Linux: we recompiled in the parallel mode with the `-par` option on the station, we `scp`'ed on the cluster all executable files (the OcamlP3l code and the three codes to couple) and data files and ran the code. And it worked! Without any parallel debugging needs, and this maybe anecdotal, but also without the bother to install anything specific on the cluster, neither OcamlP3l nor Ocaml.

We solve the problem (12) in the domain $\Omega = [0,1]\times[0,2]\times[0,5]$, with a constant scalar permeability ($\mathbf{K} \equiv 1$). We choose a right-hand side $(q, \overline{p}) = (q, 0)$, such that the solution pressure is the regular function $p^\star = x(1-x)y(2-y)z(5-z)$. An example of a numerical solution can be seen in Figure 6.

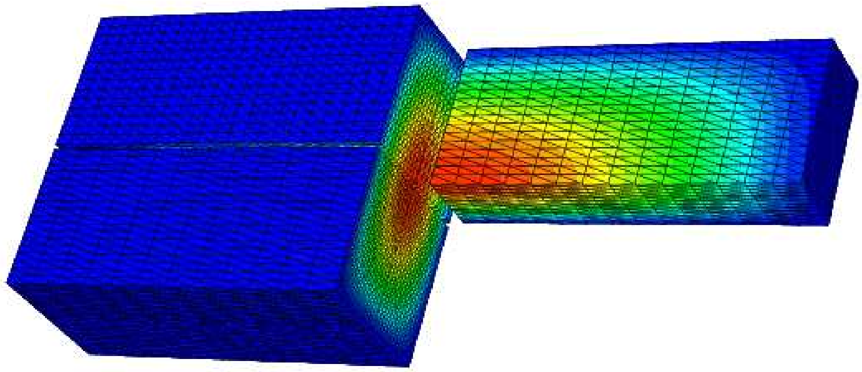

Figure 6: Pressure field for a computation over 8 non matching subdomains. For picture purposes, we present a split view of 5 subdomains. Note that the meshes do not match at the interfaces. Large (resp. small) values of the pressure are represented in red (resp. blue).

The problem (12) was discretized with first order Raviart and Thomas Mixed Hybrid Finite Elements, see [BF91] or [CR91]. The subproblems (18) in each subdomain are solved with the sparse direct LU solver provided in the UMFPack V4.1

library, see [Dav03]. The meshes used in the tests consisted of regular hexahedra. In the domain decomposition method, the Robin coefficient α_{ij}'s are all set to 5 for all the interfaces. The convergence of the Bi-CGStab algorithm is reached when the norm of the error is reduced by a factor 10^{10}.

All the tests were run on the INRIA cluster composed of 16 Xeon bi-processors with 2 GBytes of memory and more than 2 GHz of frequency. The cluster is not fully homogeneous: 13 nodes have a frequency of 2.8 GHz and 3 nodes have a slower frequency of 2.0 GHz or 2.2 GHz. The communication between the nodes is made via a Gigabit dedicated network.

We present the extensibility results in two cases: when the interface meshes match, see Table 1, and when the interface meshes do not match, see Table 2. The principle of this extensibility test is as follows: one keeps a constant load per processor and then increases the number of processors. The goal is to keep as much as possible a constant computation time, although the overall task has increased and communications are required to couple the global problem. In the context of domain decomposition, one processor is associated with one subdomain and the global domain is divided into 1, then into 2, 4, ... subdomains.

Various manners of decomposing the global domain Ω in a structured way are explored, the number of subdomains along the Ox (resp. Oy, Oz) axis is denoted by N_x (resp. N_y, N_z). Each subdomain possesses approximately 50 000 cells. The total load for, say, 16 subdomains is approximately 16 times greater than for 1 subdomain, but as it is run over 16 processors, one could expect that the CPU time remains constant, or does not increase too much.

Several remarks can be made.

- An overhead always exists when going from one subdomain to two subdomains. This is normal as no Bi-CGStab iteration, nor projection, nor communication is necessary when there is only one subdomain.

- When the number of subdomains increases, the CPU time is approximately constant and the number of Bi-CGStab iterations increases slowly. It depends on the test case: when the interface meshes match, the projection phase is much faster and cheaper in terms of memory requirements than when the interface meshes do not match; this explains why the number of Bi-CGStab iterations and the CPU is greater in the latter case. This also explains why the way the domain is divided into subdomains plays a more important role in the non-matching case: the more interfaces, the more required projections, and the more expensive each iteration.

- The `"init"` phase, during which the factorization of the operators L_i is performed, see section 4.1.3, is the costliest part of the global computation in terms of CPU and of memory: for a 51 200 cells subdomain, it requires approximately 1.4 GBytes of memory, i.e. 69% of the total amount of the memory available on a node. In the matching test case, no swap was necessary, and the slowest processor was delaying all the others.

  But in the non-matching test case, the subdomains were not identical: some had about 40 000 cells and some others 60 000 cells. The largest subdomains needed to swap during the `"init"` phase. In this case, it became crucial to balance the load and the computer resources. The colors described in section 2.8 were useful to obtain such a balance: slow processors had to treat the smallest subdomains, while the other processors took care of the rest. A comparison between tests using the colors and a series of tests without colors, and therefore an arbitrary—a priori not the worst—load distribution can be seen in Table 2. One can gain up to 25% or even 40% of the total elapsed time.

- There exists a bottleneck in communication that is due to both the way the algorithm is implemented and the way the communications are treated by OcamlP3l. All the communications between the subdomains are centralized, and therefore, when the number of subdomains increases, the large amount of communication treated by one process may cause an important overhead at each iteration. In these experiments, this drawback did not seem to be the crucial issue. But in larger tests, one may need to override this bottleneck problem. This is why a structure to allow communication from process to process is one of the developments planned in OcamlP3l.

(N_x, N_y, N_z)	# SD	CPU	# Iter	total # cells	T_n/T_1
$1 \times 1 \times 1$	1	17′20″	0	51 200	1
$1 \times 1 \times 2$	2	21′21″	6	102 400	1.23
$1 \times 1 \times 4$	4	21′51″	18	204 800	1.26
$1 \times 1 \times 8$	8	20′20″	18	409 600	1.17
$1 \times 1 \times 16$	16	21′47″	22	819 200	1.26
$1 \times 2 \times 2$	4	17′59″	8	204 800	1.04
$1 \times 2 \times 4$	8	20′20″	8	409 600	1.17
$1 \times 2 \times 8$	16	21′59″	21	819 200	1.27
$1 \times 4 \times 4$	16	22′41″	23	819 200	1.31
$2 \times 2 \times 2$	8	21′08″	8	409 600	1.22
$2 \times 2 \times 4$	16	23′03″	20	819 200	1.33

Table 1: Extensibility test for the conforming case. All the subdomains are identical. CPU time as a function of the number of subdomains (=number of processors used). CPU times are given in minutes and seconds. The configuration of the decomposition into subdomains, the number of Bi-CGStab iterations, the total number of cells in the global domain and the ratio between the CPU time for n subdomains and 1 subdomain are also given.

6 Conclusions and Future work

This approach was profitable and fruitful for both solving a numerical computation problem (namely a code coupling problem) and to benefit from the INRIA cluster powerful computation capabilities. The ability to write the numerical code in a purely sequential framework, then to quickly test and debug it while still running on small amount of data was just mandatory to get the program correct in the first place. The additional benefit of running the graphical semantics was a plus to understand and explain how the computation was proceding to get the final result.

Finally the huge boost obtained by parallelizing the program via a mere recompilation was just magic: the parallel executable ran just correctly right out of the box the first time we had access to the cluster!

On the research and social point of view, the collaboration of our two teams was just examplary: we needed not less than 6 months to start grasping one each other's ideas and problems, but since then the collaboration was very fruitful, producing new results on each sides. Numerical researchers succeeded at resolving a complex and not yet well understood problem using a functional programming approach that

(N_x, N_y, N_z)	# SD	CPU No Col	CPU Col	# Iter	total # cells
$1 \times 1 \times 1$	1	$17'20''$	$17'20''$	0	51 200
$1 \times 1 \times 2$	2	$44'29''$	$26'25''$	13	100 604
$1 \times 1 \times 4$	4	$30'50''$	$25'12''$	16	208 046
$1 \times 1 \times 8$	8	$32'36''$	$25'03''$	18	386 092
$1 \times 1 \times 16$	16	$41'08''$	$29'13''$	23	804 654
$1 \times 2 \times 2$	4	$39'33''$	$29'14''$	19	201 878
$1 \times 2 \times 4$	8	$50'53''$	$27'57''$	28	398 916
$1 \times 2 \times 8$	16	$44'28$	$34'13''$	32	816 904
$1 \times 4 \times 4$	16	$37'05''$	$31'28''$	26	785 496
$2 \times 2 \times 2$	8	$37'37''$	$33'25''$	30	421 691
$2 \times 2 \times 4$	16	$45'05''$	$31'00''$	31	812 194

Table 2: Extensibility test for the non conforming case. The subdomains are all different. CPU time as a function of the number of subdomains (=number of processors used). CPU times are given in minutes and seconds in two configurations: in the "No Col" column, the color option was not used, whereas in the "Col" column, a coloration of tasks and physical nodes was used to balance the load.

they started to like very much, while the language specialists added new interesting features to OcamlP3l to answer in a clean way to effective practical computational problems (such as the load-balancing specification using colors, the precise study of partial evaluation for initialisation of nodes, and the new `parfun` skeleton to handle lifting of sequential code).

We all appreciated very much the nice and clean theoretical fundation of OcamlP3l and found it invaluable to be able to give a precise semantics to the parallel programs: **parallel and sequential versions do have the same semantics**.

Our plan for future work is to proceed on the practical and theoretical levels as follows:

- test the coupling between different codes treating different physics,
- enrich the offer of OcamlP3l by providing `mapvector` and `reducevector` skeletons adapted to specific local communications to avoid bottleneck problems,
- enrich also the semantics of `mapvector` to recover the full power of the original `map` combinator, particularly relevant to the huge matrices computation that are typical of the numerical problems we have to solve,
- set up a general library or programming plateform to solve numerical code-coupling,
- prove the complete adequation between the sequential and parallel semantics of OcamlP3l programs.

References

[AJMN02] 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.

[AY97] 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).

[BF91] F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods.* Springer-Verlag, Berlin, 1991.

[Bir87] R. S. Bird. An introduction to the Theory of Lists. In Manfred Broy, editor, *Logic of programming and calculi of discrete design.* NATO ASI Series, 1987. International Summer School directed by F. L. Bauer, M. Broy, E. W. Dijkstra and C. A. R. Hoare.

[CMV+03] F. Clément, V. Martin, A. Vodicka, R. Di Cosmo, and P. Weis. Domain decomposition with local refinement for flow simulation around a nuclear waste disposal: direct computation and simulation using code coupling with ocamlp3l. In *Proc. of Internat. Conf. on Supercomputing in Nuclear Applications*, 2003.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[CR91] G. Chavent and J. Roberts. A unified physical presentation of mixed, mixed-hybrid finite elements and standard finite difference approximations for the determination of velocities in waterflow problems. *Advances in Water Ressources*, 14(6):329–348, 1991.

[CVDW03] F. Clément, A. Vodicka, R. Di Cosmo, and P. Weis. Couplage de codes numériques, parallélisme et langages de haut niveau. Rapport de Recherche 4825, Inria, Rocquencourt, France, 2003.

[Dav03] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. Technical Report TR-03-006, U. of Florida, 2003. Submitted to ACM Trans. Math.

[DDCLP98] Marco Danelutto, Roberto Di Cosmo, Xavier Leroy, and Susanna Pelagatti. Parallel functional programming with skeletons: the ocamlp3l experiment. *The ML Workshop*, 1998.

[Dem97] James W. Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997.

[DFH+93] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In *PARLE'93*, pages 146–160. Springer, 1993. LNCS No. 694.

[DGTY95] J. Darlington, Y. Guo, H. W. To, and J. Yang. Parallel Skeletons for Structured Composition. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM Press, July 1995.

[SS86] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Scient. Statist. Comput.*, 7(3):856–869, 1986.

[van92] H. A. van der Vorst. BI-CGSTAB: a fast and smoothly converging variant of BI-CG for the resolution of nonsymmetric linear systems. *SIAM J. Scient. Statist. Comput.*, 13(2):631–644, 1992.

A The domain decomposition method

We set first the 3D flow simulation problem in the domain Ω, then we consider the non-overlapping domain decomposition method based on Robin interface conditions described throughout section 4.1. We recall that one should refer to [AY97], [AJMN02] to have a full overview of the method. We only give here the main steps, more details and further references can also be found in [CMV$^+$03].

In particular, for the sake of simplicity, everything is set in the continuous case; so, here, the Λ_{ij}'s are infinite dimensional spaces and the $P_{i\to j}$'s are equal to the identity operator. Then, performing discretization roughly amounts to replace functions by vectors, operators by matrices and problems as (12) by linear systems.

A.1 The continuous 3D flow simulation problem

Let Ω be a convex domain in $\mathbb{R}^3$, and let $\Gamma = \partial\Omega$ be its boundary. 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 $\vec{\mathbf{u}}$,

$$(12) \qquad \left\{ \begin{array}{rcll} \operatorname{div} \vec{\mathbf{u}} & = & q & \text{in } \Omega \\ \vec{\mathbf{u}} & = & -\mathbf{K}\,(\vec{\operatorname{grad}}\, p) & \text{in } \Omega \\ p & = & \overline{p} & \text{on } \Gamma, \end{array} \right.$$

where $\mathbf{K}$ is the permeability tensor in the domain, q is a source term and $\overline{p}$ the given pressure on the boundary Γ[6]. Given $\mathbf{K}$, q and $\overline{p}$, we solve (12) for a scalar function $p(x,y,z)$ and a vector function $\vec{\mathbf{u}} = (u_x(x,y,z), u_y(x,y,z), u_z(x,y,z))^T$.

In section 4.1, we summarize in (1) the unknowns by $v = (p, \vec{\mathbf{u}})$, the—known—right-hand sides by $g = (q, \overline{p})$ and the partial derivative operator by the matrix L.

A.2 The multiblock problems

We denote by p_i, $\vec{\mathbf{u}}_i$, $\mathbf{K}_i$ and q_i the restrictions of p, $\vec{\mathbf{u}}$, $\mathbf{K}$ and q to Ω_i, and by $\overline{p}_i$ the restriction of $\overline{p}$ to Γ_i. We can show that problem (12) in Ω is equivalent to the following problems in the subdomains Ω_i,

$$(13) \qquad \forall i \in I_n, \quad \left\{ \begin{array}{rcll} \operatorname{div} \vec{\mathbf{u}}_i & = & q_i & \text{in } \Omega_i \\ \vec{\mathbf{u}}_i & = & -\mathbf{K}_i\,(\vec{\operatorname{grad}}\, p_i) & \text{in } \Omega_i \\ p_i & = & \overline{p}_i & \text{on } \Gamma_i, \end{array} \right.$$

together with the transmission conditions,

$$(14) \qquad \forall j \in \mathcal{N}_i, \quad \left\{ \begin{array}{rcll} p_i & = & p_j & \text{on } \Sigma_{ij} \\ \vec{\mathbf{u}}_i \cdot \vec{\nu}_i & = & -\vec{\mathbf{u}}_j \cdot \vec{\nu}_j & \text{on } \Sigma_{ij}, \end{array} \right.$$

where $\vec{\nu}_i$ denotes the external unit normal to Ω_i. The two equations in (14) express the continuity of the pressure and of the normal Darcy velocity across each interface Σ_{ij} between two neighboring subdomains Ω_i and Ω_j. We can also show that, for any $\alpha_{ij}, \alpha_{ji} > 0$, the system (14) is equivalent to the Robin transmission conditions,

$$(15) \qquad \forall j \in \mathcal{N}_i, \quad \left\{ \begin{array}{rcll} -\vec{\mathbf{u}}_i \cdot \vec{\nu}_i + \alpha_{ij} p_i & = & \vec{\mathbf{u}}_j \cdot \vec{\nu}_j + \alpha_{ij} p_j & \text{on } \Sigma_{ij} \\ -\vec{\mathbf{u}}_j \cdot \vec{\nu}_j + \alpha_{ji} p_j & = & \vec{\mathbf{u}}_i \cdot \vec{\nu}_i + \alpha_{ji} p_i & \text{on } \Sigma_{ij}. \end{array} \right.$$

[6] $\operatorname{div} \vec{\mathbf{u}} = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}$ is the divergence of the—column—vector field $\vec{\mathbf{u}} = (u_x, u_y, u_z)^T$ and $\vec{\operatorname{grad}}\, p = (\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z})^T$ is the gradient of the scalar field p.

Finally, the Dirichlet problem (12) is equivalent to the following Robin problems,

$$(16) \forall i \in I_n, \quad \left\{ \begin{array}{rcll} \operatorname{div} \vec{\mathbf{u}}_i & = & q_i & \text{in } \Omega_i \\ \vec{\mathbf{u}}_i & = & -\mathbf{K}_i \, (\vec{\operatorname{grad}}\, p_i) & \text{in } \Omega_i \\ p_i & = & \overline{p}_i & \text{on } \Gamma_i \\ -\vec{\mathbf{u}}_i \cdot \vec{\nu}_i + \alpha_{ij} p_i & = & \vec{\mathbf{u}}_j \cdot \vec{\nu}_j + \alpha_{ij} p_j & \text{on } \Sigma_{ij} \ (\forall j \in \mathcal{N}_i). \end{array} \right.$$

Of course, all these subproblems are coupled through the Robin conditions, and using a direct method that builds the inverse of the complete matrix would be as costly as solving the initial problem (12). Hence, we need to consider an iterative method.

A.3 The fixed point formulation

The latter problem (16) can be formulated as a fixed point problem, i.e. for all $i \in I_n$, choose initial guesses p_i^0 and $\vec{\mathbf{u}}_i^0$, then iterate for $n \geq 0$,

$$(17) \quad \left\{ \begin{array}{rcll} \operatorname{div} \vec{\mathbf{u}}_i^{n+1} & = & q_i & \text{in } \Omega_i \\ \vec{\mathbf{u}}_i^{n+1} & = & -\mathbf{K}_i \, (\vec{\operatorname{grad}}\, p_i^{n+1}) & \text{in } \Omega_i \\ p_i^{n+1} & = & \overline{p}_i & \text{on } \Gamma_i \\ -\vec{\mathbf{u}}_i^{n+1} \cdot \vec{\nu}_i + \alpha_{ij} p_i^{n+1} & = & \vec{\mathbf{u}}_j^{n} \cdot \vec{\nu}_j + \alpha_{ij} p_j^{n} & \text{on } \Sigma_{ij} \ (\forall j \in \mathcal{N}_i). \end{array} \right.$$

A.4 The Robin-to-Robin interface operator

At this stage, it is natural to introduce the interface operators S_{g_i}, for $i \in I_n$, that input their own Robin conditions on all their interfaces and then output that for their neighbors[7]. They are defined in each subdomain Ω_i by

$$(18) \quad \begin{array}{l} S_{g_i} : \ (\lambda_{ij_1}, \ldots, \lambda_{ij_{n_i}}) \in \Lambda_i \longmapsto (\mu_{ij_1}, \ldots, \mu_{ij_{n_i}}) \in \Lambda_i \\ \text{such that } \forall j \in \mathcal{N}_i, \quad \mu_{ij} = \vec{\mathbf{u}}_i^\star \cdot \vec{\nu}_i + \alpha_{ji} p_i^\star \end{array}$$

where $p_i^\star$ and $\vec{\mathbf{u}}_i^\star$ are solutions to

$$(19) \quad \left\{ \begin{array}{rcll} \operatorname{div} \vec{\mathbf{u}}_i & = & q_i & \text{in } \Omega_i \\ \vec{\mathbf{u}}_i & = & -\mathbf{K}_i \, (\vec{\operatorname{grad}}\, p_i) & \text{in } \Omega_i \\ p_i & = & \overline{p}_i & \text{on } \Gamma_i \\ -\vec{\mathbf{u}}_i \cdot \vec{\nu}_i + \alpha_{ij} p_i & = & \lambda_{ij} & \text{on } \Sigma_{ij} \ (\forall j \in \mathcal{N}_i). \end{array} \right.$$

Again, in section 4.1, we summarize in (8) the unknowns by $v_i = (p_i, \vec{\mathbf{u}}_i)$, the—known—right-hand sides by $g_i = (q_i, \overline{p}_i)$ and the partial derivative operator by the matrix L_i.

Then, using the restriction/reconstruction and projection operators defined in section 4.1 by (5), (6) and (9), the fixed point problem (17) consists in solving

$$(20) \qquad \lambda = sPS_g\lambda = f(\lambda).$$

A.5 The nonsymmetric linear system formulation

Since the "operator S" is bilinear, namely $S_g\lambda = S_0\lambda + S_g 0$, the fixed point problem (20) is clearly equivalent to solve the linear system (10) defined by (11) in section 4.1. The S_{g_i}'s matrices are symmetric, but because of the projections, the matrix A is nonsymmetric.

Of course, at the end, when the solution $\lambda^\star$ of the system is reached, we have to solve once more the inner subproblems (19) associated with $S_g\lambda^\star$ to obtain the sought pressures p_i and Darcy velocities $\vec{\mathbf{u}}_i$ in each subdomain.

[7]Notice that on the two sides of an interface, the sign of the external normal changes.

B The interface of the Ddec module

```
 (* Module Ddec for 3D domain decomposition *)

val filename_of_3D_mesh_number : int -> string
val filename_of_2D_mesh_number : int -> int -> string

val number_of_subdomains : int
val number_of_processors : int

type interface
type connectivity_table = interface list
val connectivity_table : connectivity_table

val colv_sorting_subdomains : int array
val colv_sorting_connectivity_table : int array

type interface_values
and internal_boundary_values = interface_values array
and tagged_internal_boundary_values =
  | Init
  | Loop of internal_boundary_values
  | Final of internal_boundary_values
and structure_values = internal_boundary_values array

val zero_structure_values : int -> structure_values

val read_internal_boundary_values :
    in_channel -> internal_boundary_values
val print_internal_boundary_values :
    out_channel -> internal_boundary_values -> unit
val print_tagged_internal_boundary_values :
    out_channel -> tagged_internal_boundary_values -> unit

val init_vector_of_size :
    int -> int * tagged_internal_boundary_values array
val loop_vector_of : structure_values
  -> int * tagged_internal_boundary_values array
val final_vector_of : structure_values
  -> int * tagged_internal_boundary_values array

val axpy : float -> structure_values -> structure_values
  -> structure_values
val dot : structure_values -> structure_values -> float

val permutation_of : structure_values -> structure_values

type projection_matrix
type projection_matrices = projection_matrix array array

val projection_with :
    projection_matrices -> structure_values -> structure_values
```