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 *parfun*s 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 `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, `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, `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`.