Parallel Processing Letters

SKELETAL PARALLEL PROGRAMMING WITH OcamlP3l 2.0*

ROBERTO DI COSMO, ZHENG LI†
Laboratory "Preuves, Programmes et Systémes", University of Paris VII
175, rue du Chevaleret, F-75013 Paris France
e-mail:{Roberto.DiCosmo,Zheng.Li}@pps.jussieu.fr

SUSANNA PELAGATTI
Dipartimento di Informatica, University of Pisa
Largo B. Pontecorvo, 3 I-56127 Pisa Italy e-mail:susanna@di.unipi.it

PIERRE WEIS
Project "Crystal" – INRIA Rocquencourt France e-mail:Pierre.Weis@inria.fr

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

Parallel programming has proven to be an effective technique to improve the performance of computationally intensive applications. However, writing parallel programs is not easy, and activities such as debugging are usually hard and time consuming. To cope with these difficulties, skeletal parallel programming has been widely explored in recent years with very promising results. However, prototypal skeletal systems developed so far tend to be rather inflexible and difficult to adapt to many practical parallelization scenarios. For instance, many systems restrict *all* the sub structures inside a parallel application to be encapsulated together in term of possibly nested skeletons, which may be cumbersome when parallelizing some large and complex applications. Moreover, it is usually difficult to share resources among different skeleton instances and to reuse the same instance of a skeleton in different parts of the code. This paper reports on the current status of the OcamlP3l (2.0) systems, which sensibly changes the skeletal model of the previous versions, to make it more usable and flexible. In particular, we describe the new skeletons, the new skeletal execution model as well as related issues on design and implementation, and we conclude with some small examples and preliminary results.

1. Introduction and Overview

Many parallel applications achieve high speedups at the price of software quality and maintainability. Programmers are asked to handle explicitly a number of activities related to communication/synchronization of the parallel activities and resource management. The choices made are usually hard wired in the program using low level libraries such as MPI. This makes software development a complex task, and

*OcamlP3l has been partially funded by a Galileo bilateral France-Italy project and by CARAML, a project in the French ACI-GRID Programme.
†Di Cosmo and Li participate also to Project Crystal – INRIA Rocquencourt

the resulting programs are difficult to maintain, port and prove correctness against specifications.

In a skeleton-based parallel programming model[6,11,2], the programmer is not forced to program every single process interaction or to devise resource allocation and data distribution strategies. Such models provide typical organization patterns of parallel programs as high order constructors or library functions (i.e. *skeletons*). A programmer uses skeletons and their combinations to give parallel structure to an application, and uses a plain sequential language to present the activities inside each processing unit, as parameters to the skeletons.

In recent years, many researchers have explored the potential of skeletons in both parallel and grid environments and have reported some important benefits of this model [15,10,1,16,14,8,19]. When the application structure naturally fits into some combination of available skeletons, programming is amazingly simple and concise. It is also very easy to port skeletal programs from one parallel platform to another. Sophisticated implementation techniques can be developed for each single skeleton, and, thanks to the structural nature of skeletal programming, global optimization can be simply automated with cost models [18,15].

Our contribution is this field is the OcamlP3l system [8], a programming environment that provides a skeletal model and at the same time provides seamless integration of parallel programming and functional programming with advanced features like sequential logical debugging (i.e. functional debugging of a parallel program via execution of all parallel code onto a sequential machine) and strong typing, useful both as a testbed for innovative parallel programming style and a practical tool in building full-scale applications for scientific computation.

Despite these encouraging features, skeletal systems have not become mainstream in the real parallel world so far: they have been around for a decade now, but most parallel application developers still use low level libraries and models. From our experience with OcamlP3l (1.0) when working in multidisciplinary teams to apply skeletal programming to large real world applications [5], we had the impression that part of the problem could be caused by the over-restrictive nature that exists in several aspects in many of the skeletal systems.

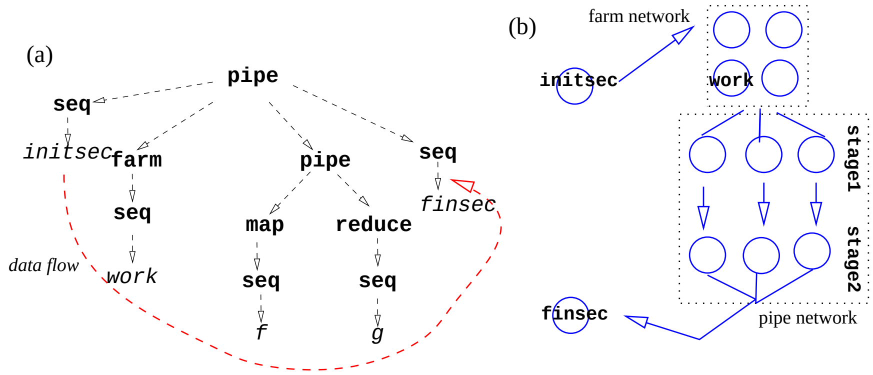
To face the problem, We have taken into account the desiderata that emerged from users in the field of applied mathematics, without sacrificing the elegance of the skeleton model, by extending our skeleton set and defining a new execution model, which is the heart of OcamlP3l (`2.0`) .

This paper describes the current stable status in the evolution of the OcamlP3l skeleton-based functional parallel programming system, and is structured as follows. Section 2 and 3 review our motivations and introduce the new skeletal mode. Section 4 gives two introductory small examples. Detailed syntax, semantics and typing issues are mainly addressed in Section 5. Section 6 and 7 give a brief account on some implementation stuff and present a few results. Section 8 concludes.

2. Motivations

2.1. Mixing sequential and parallel code

The first version of OcamlP3l (1.0) [8] imported the skeletal model proposed by P3L [2,18] with some minor changes due to the functional nature of the system. According to this model, a programmer can structure parallelism by nesting task parallel skeletons (`farm` and `pipe`) and data parallel skeletons (`map`, `reduce` and `scan`). Moreover, any skeletal structure can be iterated using the `loop` control skeleton and sequential code can be encapsulated using the `seq` skeleton. For instance, a typical structure for a P3L program is depicted below.



Here, on the left hand side (a) we can see a typical skeleton nesting and on the right hand side (b) there is the corresponding process network. Our application is a pipeline of four stages in which the first and the last stage are sequential stream processes and the other two are further parallelized using a farm and a pipeline respectively. The process network (b) works on a stream of input data produced by the initial stage `initsec` and produces a steam of output data from `finsec`.

From the picture, we can see that a P3L program is clearly stratified into two levels: there is a skeleton *cap* (in bold in the picture) which describes all the parallel structure in the application. The cap 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 when the goal is to build *a single* stream processing network described by the skeleton cap. However, it has several drawbacks in the general case:

breaks uniformity : Though the skeletons *look like* ordinary functions, they are actually in different classes and can never been uniformly mixed together; hence, the programmers have to program in a style that strictly conforms with the two-level style, especially, the skeletons *cannot* be invoked as ordinary functions from sequential code, even if they could have appropriate types.

may produce contrived programs : many applications boil down to simple nested loops, some of which can be easily parallelized, and some cannot; in some cases, like the numerical algorithms described in [5], what the user was really asking for, is the possibility of just parallelizing a particular very heavy matrix computation deep inside the sequential code, while our old model enforced the user to rewrite all the program logic in a very unnatural way with the control parallel skeletons like *loop*;

prevents sharing : in various numerical algorithms, some operation, like multiplying some vector v by the very same large matrix A, may be performed at different places of the sequential algorithm, and the user naturally wants to a way to assure that this computation be performed by the same processing resources (sharing the large matrix A). The P3L skeleton cap does not allow the user to specify this sharing.

To overcome all these difficulties and limitations, the `2.0` version of OcamlP3l introduces the new `parfun` skeleton, the very *dual* of the `seq` skeleton. In simple words, one used to warp a regular function to be a skeleton unit with `seq`, now one can also wrap a full skeleton expression inside a `parfun` to obtain a regular stream processing function, usable with no limitations in any sequential piece of code.[‡] A `parfun` encapsulated skeleton function behaves exactly as a normal function that receives a stream as input value, and returns a stream as output value. Under the parallel semantics, an actual implementation for the structure inside a `parfun` combinator then turns out to be a parallel network, to which the `parfun` 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, in contrast with P3L, the OcamlP3l model of computation requires a *main* sequential program (modeled by the `pardo` skeleton). This main program is responsible for information interchange among the various `parfun` encapsulated skeletons. For instance, in our new model we can express the structure in Fig. 1.

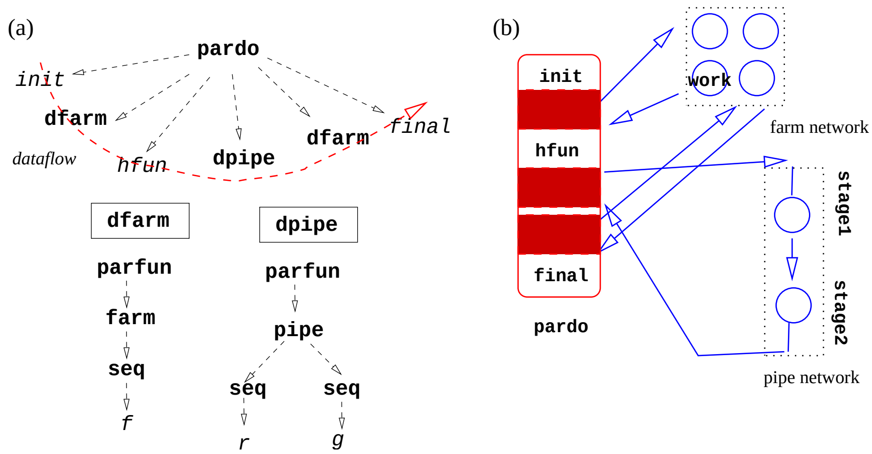


Figure 1: A parallel structure mixing sequential and parallel execution

[‡]Actually, in any sequential code that is not used in building a skeleton.

Here, we have two skeleton instances `dfarm` and `dpipe` which have been defined once and for all using a parfun skeleton. The main `pardo` skeleton is executed by a root node which first deploys all the parallel networks (farm network and pipe network in our case) and then orchestrates the computation alternating the execution of sequential functions (`init`, `hfun` and `final`) with the communication of data to the parallel networks. In this case, the function `init` may generate a stream of data, which is then passed to the farm network for processing. Then, the root node receives the stream of results produced and applies to them the sequential function `hfun` before feeding the results into the pipe network. The stream of results coming out of the pipe is then sent again to the same farm network for further processing. When the root node gets the output from the `dfarm` skeleton for another time, the `final` function will be invoked for the last sequential elaboration which usually involves some finalization and output tasks.

2.2. Skeleton specialization

Another problem which we encountered repeatedly in our real-world application experiences was the need of using a set of skeletons which are parallel in structure but slightly different in function. Typically these skeletons are specialized versions of a same skeleton which instantiate the working function with different data bases. For instance, each worker skeletons nesting inside the same farm skeleton may need to hold some sizable local data (e.g. some huge matrices as the basic for the computation over each items inside the incoming streams) which could be different from one another on each worker, though the parallel structures' behavior itself remains identical.

The original P3L model did not allow this kind of specialization, all the parallel sub-structures have to be absolutely the same. The working functions holding different data bases will have to be considered as different ones. So the implementation of such a farm skeleton would involve a large amount of data replication (i.e., to employ just one same worker function, each worker will have to get all the huge matrices as a whole to cover everyone's needs, but may only touch one of them).

Instead, we wanted to offer the user the freedom to finely describe where and when the data are used and to limit unnecessary replication as much as possible. This turned out to be possible at the price of a slightly more complicated but still comprehensible refinement of skeleton types that we detail in Section 5.

3. The OcamlP3l 2.0 computing model

3.1. Skeletons

OcamlP3l provides three kinds of skeletons: task parallel skeletons, data parallel skeletons and control skeletons. Each skeleton is a *stream processor*, i.e. a function which transforms an input stream of incoming data into an output stream of outgoing data. Skeletons can be composed to define the parallel behavior of programs.

Task parallel skeletons model the parallelism of *independent* processing activities related to different input data. In this set, we have pipe and farm. Both skeletons transform a stream of independent input data into a stream of results. The `farm` replicates a skeleton into a pool of identical copies (the *farm workers*) each one computing independent data items in the input stream. The `pipe` exploits parallelism in the execution of a sequence of skeletons defining independent stages of a computation. Both correspond to the usual task parallel skeletons appearing in P3L and in other skeleton models [12]. *Data parallel* skeletons exploit parallelism in the computation of different parts of the same input data. In this set, we provide `mapvector` and `reducevector`. Both work on dense one-dimensional arrays. 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. Those two skeletons are simplified versions of their respective `map` and `reduce` analogies in P3L. They provide a functionality quite similar to the map($*$) and reduce ($/$) functions of the Bird-Meertens formalism discussed in [4]. *Control skeletons* are combinators which do not express parallelism *per se*, but orchestrate the interaction and control flow among the sequential and parallel parts of an application. We have three kinds of control skeletons:

- the *iteration* skeleton (`loop`), that iterates the execution of inside skeleton over any items in the incoming stream, until they finally meet some given condition.

- the *data interface* skeletons (`seq` and `parfun`) that provide a way to pass from the sequential to the parallel world and *vice versa*. `seq` converts a sequential function into a stream processing unit which is necessary for instantiating any of the skeletons inside a parallel network, and `parfun` turns a parallel network defined via a skeleton expression into a standard Ocaml function.

- the *parallel execution scope delimiter* skeleton (`pardo`) which encapsulate all the parallel code, i.e. the code that invokes the `parfun` networks.

3.2. Parallel execution model

A parallel computation in OcamlP3l is defined by three components: (1) a set of plain sequential Ocaml functions (CF, common functions in Figure 2); (2) a set of parallel clusters, each one defined by a suitable composition of skeleton combinators enclosed in a `parfun` (SF, skeleton functions in Figure 2) and (3) a `pardo` application. Each definition of a `parfun(h)` function will create a corresponding network of processes according to the skeleton composition in `h`. Each network transforms a stream of independent input data $\ldots x_1, x_0$ in a stream of output data $\ldots h(x_1), h(x_0)$ according to `h`.

When a `pardo` is evaluated, applications of common functions boil down to normal sequential evaluation, while applications of skeletal functions feed arguments data to the corresponding skeletal process network and are evaluated in parallel. In practice, a `pardo` defines a network built out of all the processes in skeletal networks (`parfun` defined functions) plus a *root* process orchestrating all the computation. Both the root node and the common nodes run in SPMD model. Initially, the root

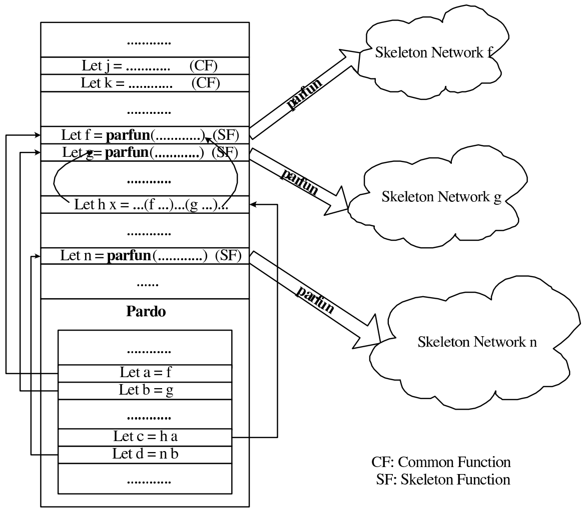


Figure 2: Parallel execution model: the role of `parfun` and `pardo` skeleton

specializes all the common nodes by sending information on the actual process to be executed (e.g., a farm dispatcher, a farm worker, a mapvector worker etc). Then, the root process starts executing the `pardo`. The usual sequential code is executed locally on the root node. Instead, when a call to functions that have been defined via a `parfun` function is encountered, the root node just passes the argument stream of the function to the corresponding network and returns as a result the stream of data produced by the network. The same network can be activated many times, each time a call of the corresponding `parfun` function is encountered.

Notice that the execution model so far implicitly assumes an unlimited number of homogeneous processors. In practical situations, processors will be less than processes and have heterogeneous performance. It is then necessary to reasonably arrange the mapping relation from the virtual resources implied by the program into the actual resources at hand. The support, possibly with some help from the programmer (using colors [17], see Sec. 5.5), is in charge of performing the mapping task appropriately, with the possibilities of choosing from either an automatic or an explicit manner on their respective advantages.

4. A simple example: farming out square computation

It is now time to discuss a simple but complete OcamlP3l program. The following program uses a farm to compute a very simple function over a stream of floats

```
(* computes x square *)
let farm_worker _ = fun x -> x *. x;;
(* prints a result *)
let print_result x =
  print_float x; print_newline();;
let compute =
```

```
5.   parfun (fun () ->  (farm (seq(farm_worker),4)));;
6. pardo(fun () ->
7.   let is = P3lstream.of_list [1.0;2.0;3.0;4.0;5.0;6.0;7.0;8.0] in
8.     let s' = compute is in P3lstream.iter print_result s';
9. );;
```

We have two standard Ocaml functions (1–3): `farm_worker` which simply computes the square of a float argument and `print_result` which dumps results on the standard output. Notice that the `farm_worker` has two parameters instead of one as it would seem reasonable. The extra parameter (`_`) is required by the `seq` skeleton type and is used in general to provide local initialization data (for instance, an initialization matrix, some initial seed or the like). This optional initialization is provided for all OcamlP3l skeletons (see Section 5). In this simple case, initialization data is not needed and the parameter is just ignored by `farm_worker`. Function `compute` (4–5) uses a `parfun` skeleton to define a parallel network built by a single farm, in particular:

```
seq(farm_worker)
```

turns the sequential `farm_worker` function into a 'stream processor' applying it to a stream of input values. Then, an instance of the farm skeleton is defined with

```
farm (seq(farm_worker),4)
```

which spawns four workers. Finally,

```
parfun (fun () ->  (farm (seq(farm_worker),4)));;
```

encapsulates the skeleton network into a standard Ocaml function.

The last `pardo` (6–9) defines how sequential functions and parallel modules are interconnected. In this case, we have a single parallel module (`compute`) and two sequential parts. The first sequential part builds up the data stream, using the standard OcamlP3l function

```
P3lstream.of_list [1.0;2.0;3.0;4.0;5.0;6.0;7.0;8.0]
```

which turns a list into a stream. The second sequential part applies the sequential function (`print_result`) to all the elements in the stream (using an iterator `P3lstream.iter` provided by the standard stream module). The global structure of the network is shown in Figure 3. Here, arrows show the data flow among processes.

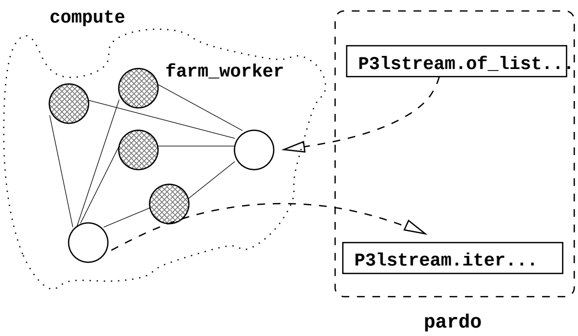


Figure 3: Overall process network of the simple farm example.

4.1. A PDE solver on multiple domains

Our second example is a parallel PDE solver which works on a set of subdomains. On each subdomain it applies a fast Poisson solver written in C. A fragment of the code is

```
1.  let solver =
2.     parfun (fun () ->
3.       (loop ((fun (v,continue) -> continue ),
4.             seq(fun _ -> fun (v,_) -> v)
5.           ||| mapvector(seq(fun _ -> calcul_sous_domaine),3)
6.           ||| seq(fun _ -> projection)
7.           ||| seq(fun _ -> bicgstab)
8.           ||| seq(fun _ -> plot)
9.           ) )  ) ;;

10.  pardo( fun () ->
11.    List.iter print_result
12.      (P3lstream.to_list (solver (P3lstream.of_fun generate_input_stream)))
);;
```

Figure 4: Code fragment from a Poisson solver.

shown in Fig. 4. We only show and discuss the parallel structure. Here, we have a five stage pipeline (4–8) `seq|||mapvector||| ... ||| seq` which computes a single iteration of the method and a loop skeleton which iterates the computation until `continue` is true. Being the first skeleton in a loop, the first seq (4) receives a pair of data to be computed and a termination flag and selects the first one for further computation discarding the flag (which is used only by the loop controller. The data `v` is a vector defining the number and structure of subdomains to be computed in parallel. `mapvector` simply applies `calcul_sous_domaine` to each subdomain, using 3 parallel workers. Each worker actually spawns a Unix process implementing the fast C solver and then simply feeds it with the subdomains to be computed until termination. When the computation on all subdomains for a given data set is finished, mapvector glues the results together and propagates them along the pipe. The subsequent pipeline stages (6–8) simply trim overlapping borders and plot the result in a formatted way.

5. Skeleton syntax, semantics, and types

Here we describe the syntax, the informal semantics, and the types assigned to each of the combinators of the skeleton language. Each skeleton is a stream processor, transforming an input stream into an output stream and is equipped with three semantics: a *sequential semantics*, a *parallel semantics* and a *graphical semantics*. The sequential semantics is a suitable sequential Ocaml function `f` transforming all the elements `x` of the input stream in the corresponding output `f x`. The parallel semantics is a process network implementing the skeleton in parallel (the same skeleton can be implemented by several different process networks, corresponding to different *implementation templates*). The graphical semantics is a pictorial representation of the process network implementing the skeleton.

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. According to the parallel semantics, 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.

5.1. On the type of skeleton combinators

First of all, let us explain why the actual Ocaml types of our skeleton combinators are a bit more complex than those used by other skeleton systems (e.g.., [12]). In effect, our types seem somewhat polluted by spurious additional `unit` types, compared to the types one would expect.

For instance, consider the `seq` combinator. As informally discussed above, `seq` encapsulates any Ocaml function f into a sequential process which applies f to all the inputs received in the input stream. This means that, writing `seq f`, the Ocaml function `f` with type `'a -> 'b` is wrapped into a parallel process that applies `f` pointwise to the stream of input data of type `'a` to produce a stream of outputs of type `'b` (this is reminiscent of the `lift` combinator used in many stream processing libraries of functional programming languages). Hence, a straightforward type for `seq` would be

```
('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 sizable amount of local data (e.g. some huge matrices that have to be initialized in a numerical application), and we decided to have a type general enough to allow the user to finely describe where and when those data have to be initialized and/or copied.

Similarly to what is done in 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. We discuss both local and global initialization below.

***global initialization*:** we want to initialize the data once and for all, and then replicate it in every copy of the stream processor that a `farm` or a `mapvector` skeleton may create; to achieve this result we can write

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

This was also trivially possible in the previous versions of OcamlP3l, where we could write (without the extra unit parameter)

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

***local initialization*:** we want to initialize the data locally in each stream processor, *after* the duplication has been performed by a `farm` or a `mapvector` skeleton; this was just impossible in the previous versions of OcamlP3l; with the extra unit type parameters we can now write:

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

To understand why this works, we need to explain a bit the mechanics of the creation of the process network: when the `farm` skeleton is created, it *first* creates 10 copies of `seq f`, that contains the function `f` not yet applied to any parameter; only *after* the creation of the processing node containing `f` the runtime passes () as argument to f to obtain the function that will be applied at each stream element; the evaluation of `f ()` then produces the allocation of a different copy of *localdata* for each instance of the `seq f` skeleton.[§]

To sum up, the extra `unit` parameters give the programmer the ability to decide whether local initialization 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, and is reminiscent of the phase separation found in languages like Klaim [3].

5.2. Task and data parallel skeletons

We can now detail the types and semantics of the remaining 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, $f_1 \texttt{|||} f_2 \ldots \texttt{|||} f_n$ 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 `mapvector` skeleton applies a function to all elements 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 $[f(x_1), \ldots, f(x_n)]$. 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 `reducevector` skeleton folds a binary operator $\oplus$ over all the data items of a vector. Therefore, $\texttt{reducevector}(\oplus, n)$ computes $x_1 \oplus x_2 \oplus \ldots \oplus x_n$ out of each the vector $x_1, \ldots, x_n$ in the input data stream. If $\oplus$ has type `(unit -> 'a * 'a stream -> 'a stream)`, and n has type `int`, then $reducevector(\oplus, n)$ has type `unit -> 'a array stream -> 'a stream`.

5.3. The `parfun` *skeleton*

[§] In practice, the initialization step may do weird, non referentially transparent things, like opening file descriptors or negotiating 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.

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`.

However, `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 instantiated by the main program (`pardo`), not by all of the multiple running copies of the SPMD binary, even though those 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 instantiate the skeletons.

5.4. The `pardo` *skeleton: a parallel scope delimiter*

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

```
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.

In order to have the `parfun` and `pardo` work correctly together the following *parallel scoping rule* has to be followed:

- 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 be used directly inside a `pardo` combinator.

Due to this scoping rule, the general structure of an OcamlP3l program is the one shown in Figure 5.

5.5. Load balancing: the colors

The execution model of OcamlP3l assumes an unlimited number of *virtual* processors, which are then allocated on the available physical machines. In the previous implementation, this mapping mechanics is simply a round robin, which roughly eliminates the capability differences between physical servers. Though one can make some tricky adjustment, e.g. by providing a single physical machine as several independent instances in the configuration, it's still rather difficult to make the load balance particularly even in many cases. Hence *color*, an integer parameter representing the relative weights of each processor, is now introduced for both virtual processors and physical ones.

Let us consider as an example, the skeleton expression we discussed in the example of section 4: `parfun (fun () -> (farm (seq(farm_worker), 4)))` that corresponds to a skeleton network of one emitter node, one collector node, and 4 worker nodes computing the square function. Now we are allowed to add the *optional* color parameter to *any* skeleton combinator in order to specify the relative weights of the exact part. The keyword `col` is used, with a `~` precedent, following the Ocaml convention of optional argument. The updated version of our example now becomes

```
(* (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 defined 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 *)
```

Figure 5: Generic structure of an OcamlP3l program

```
parfun (fun () -> (farm ~col:2 (seq(farm_worker), 4)))
```

which tells that all virtual nodes inside this farm structure (both the four worker nodes and the emitter/collector nodes) have the ranks 2, and should accordingly be mapped to some physical nodes with a capability ranking at least 2. The scope of a color specification covers all the inner nodes of the structure it qualifies: unless explicitly 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). As in the following expression,

```
parfun (fun () -> (farm ~col:2 (seq ~col:7 (farm_work), 4)))
```

the emitter and collector nods are still of rank 2, while the four worker nodes are now ranked to 7, which generally implies more computation power needed.

For `farm`, `mapvector` and `reducevector`, in addition to the color of the combinator itself, there is an additional optional color parameter *colv*. A *colv* specification is a `color list` (i.e. an `int list`) used to assign *different* weights for the inside parallel working structures. For an example, the OcamlP3l expression

```
parfun (fun () -> farm ~col:2 ~colv:[3;4;5;6] (seq(farm_worker), 4))
```

assigns the emitter and collector rank 2, and the four worker nodes (four copies of `seq`) with respective ranks 3, 4, 5, and 6.

On the other hand, in order to map the processors from the virtual side with the machines of real world, additional parameters for the physical machines are correspondingly demanded. The configuration of physical nodes is still acquired through the launching command as in previous implementation, with several additional parameters. The specification of each physical machine is comprised of four elements in the following format: `ip_or_name:port#color%volume`, of which *port*, *color* and *volume* are all optional parameters with default values. The *color* parameter is defined with the same convention as the virtual side. The *volume* parameter is an integer that stands for the maximum numbers of virtual processors being allowed to map on current machine simultaneously. An typical command launching a OcamlP3l parallel application looks like the following example,

```
prog.par -p3lroot 192.168.0.1:4080#7%4 192.168.0.2#2%2 192.168.0.4 ...
```

where `prog.par` is the name of executable compiled with parallel semantics, and the `-p3lroot` option states that current node is the root node of this SPMD computation. The list follows that is the customized configuration of each physical machine for participating in the current round of computation.

We are left to explain the mechanics we used to pair virtual nodes with physical nodes. Two different modes have been developed to satisfy different requirements: a *strict* mode and a *non-strict* mode.

The strict mode is enabled by launch the computation with an extra `-strict` option. In this case, the number of virtual nodes defined in the program must exactly match the number of physical nodes provided on the command line, as well as their distribution of colors. In brief, it's a one-to-one relation based on color equivalence, so the mapping mechanics is simply straightforward, and the programmer has full control over the mapping; the obvious disadvantage is that one has to manually adjust the color distribution with patience until they finally fit, which can be quite boring for structurally complicated applications.

On the other hand, the non-strict mode, which is enabled by default, performs mapping in a mostly automatic way (except in some extreme cases where the condition provided by the programmer is not satisfiable). We adopt the following algorithm for mapping: we first sort the virtual nodes in decreasing order of their color values, to reflect the priority in choosing a physical node — the nodes with higher ranks should have more privilege on choosing their targets than the nodes with lower ones; then we begin to pair the virtual nodes in order — for each virtual node x with color c_x, we pick, among all physical nodes with a color $c \geq c_x$, the one that has been assigned to the minimum number of virtual processors but still under its *volume*;[¶] the pairing will continue until all virtual nodes in the queue have been consumed in which case the mapping is successful, or there is no physical nodes left satisfying the requirements of current virtual node in which case the mapping fails.

The practical experience showed that the non-strict mode allows very flexible configurations without losing the convenience, but in a few cases the fine tuning provided by the was necessary in order to achieve optimal performances. A general purpose suggestion under the non-strict mode, is to configure those heavy working nodes (from both sides) with highly distinct color values to ensure a fixed mapping relation as wanted, and leave all other light-task nodes and coordination nodes automatically mapped.

All color related parameters are designed as optional: this saves unnecessary coding effort at many positions where default values work just fine, and it guarantees full compatibility with the source code of former projects developed on OcamlP3l .

Notice that a *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 optimized mapping based on precise quantitative estimations of the cost of each sequential function and the capabilities of the physical machines, so that we cannot guarantee our color-based mapping algorithm to be highly accurate or highly effective. Still, the *color* approach has

[¶]In non-strict mode, a physical node without explicit volume argument are considered as "no limitation on volume", which is, by default, compatible with the concept in the previous round-robin strategy. In strict mode, the bijection must be assured. A physical node without volume argument is then considered as "no additional instances allowed" and therefore has a default volume 1.

proved to be quite effective and practical language feature in the real-world cases.

6. Implementation

OcamlP3l is completely implemented in Ocaml and depending on the option used, it executes source sequentially, produces a graphical output or carries on actual parallel execution on different machines. We here focus on parallel execution, as the other two are straightforward (see [7] for details).

In parallel execution, all the nodes in the network run the very same program, which is the result of the compilation of the user code. One node, the root, will organize the process network and provide the others with specialization parameters. At run time, each generic copy just waits for instructions from the root node; the root node first evaluates the arguments to the `parfun` combinators to build a representation of the needed skeletons; then, upon encountering the `pardo` combinator, the root node initializes all the parallel computation networks, specializing the generic copies by closure passing (as described in details in [8] and [7]), 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 2. The skeleton networks are initiated only once but could be invoked many times during the execution of `pardo`.

In particular, during the execution of a `pardo` expression, the root node must accomplish the following activities: (1) maps virtual nodes to the processor pool given on the command line, (2)initializes a socket connection with all the participating nodes, (3) gets the port addresses from each of them (a fixed port number —`p3lport`— or some dynamically generated number if more than one copy run on the same machine), (4) sends out to each node the addresses of its connected neighbors (this step together with the previous two provides an implementation of a centralized deadlock free algorithm to interconnect the other nodes into the process network specified by the skeleton expression), (5) sends out to each node the specialization information that consists of the *function* it must perform.

This very last task requires a sophisticated operation: sending a *function* (or a closure) over a communication channel. This is usually not possible in traditional functional programming languages, since sending an arbitrary function supposes that we are able to find on the receiving side the code corresponding to the function name received *or* that we can transfer executable code (a feature known as *mobility* today). Now, mobility is necessary to send closures between arbitrary programs (since two different programs have no reason to know each other's function code), but *not* between two copies of the *same* program: in the latter case, it suffices to send what essentially amounts to a code pointer. Starting from version 1.06, Ocaml contains a modified marshaling library, originally designed for the OcamlP3l system, that performs closure sending between copies of the same program (this is checked by means of an MD5 signature of the program code). The `ocaml` run time system takes care of dealing with differences in endianness and word size between communicating machines, as well as flattening tree-shaped data structures. On the other side, all the other nodes simply wait for a connection to come in from the root node, then send out the address of the port they allocate to do further communication, wait for the list of neighbors and for the specialization function, then simply perform it until termination.

To summarize, in the implementation the possibility of sending closures allowed us to obtain a kind of higher order distributed parametrization that kept the runtime code to a minimum size.

6.1. Using the system

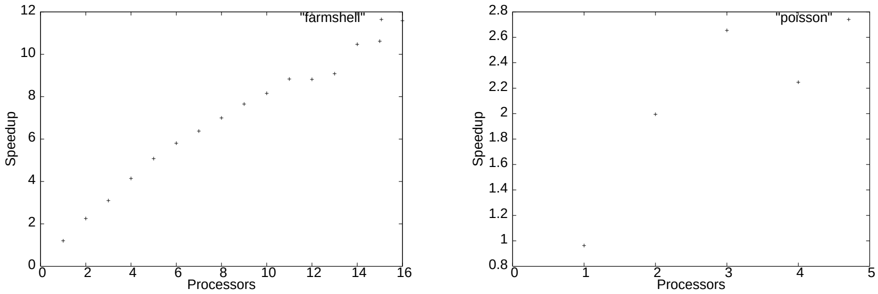


Figure 6: Some results on a shared cluster with heterogeneous load

Once written, an OcamlP3l source file can be compiled using sequential, parallel and graphical semantics (options `-seq/par/gra`). With sequential compilation, the executable is a sequential one, which can be directly run on a single machine, tested and debugged using the regular Ocaml source level debugger as any other sequential program. With graphical compilation, we get an executable file whose launch displays the skeleton network specified by the program. Finally, parallel compilation produces a 'generic' SPMD parallel executable. One copy of this executable is launched on all the working machines, 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. An example has already been given in Section 5.5 where we described how colors are assigned to the physical nodes. More info on the system can be found in [7] or on the web site.[‖]

7. Some results

Figure 6 shows some results obtained with the two example programs: *farmshell* (left) computes an embarrassingly parallel computation discovering prime numbers using a farm and *poisson* (right) is the PDE solver described in Section 4. Programs were run during busy working days in the student laboratory in Pisa, using a simple script to select a pool of n machines available for *ssh* regardless of their load. All the machines have the same configuration (AMD Athlon 2.3Ghz with 256 KB cache and 512MB RAM). The network is a fast Ethernet shared by all the student labs (around 300 machines with typical Web, mail, NFS etc. traffic). Results show the average measurements over multiple runs. Despite of the unfriendly scenario, both programs show quite good speedups. *farmshell* had 16 workers with a stream of 100 tasks: when the number of tasks falls under 8/10 load unbalance starts to show its effects where one single overloaded worker can slow down all the rest. Regarding *poisson*, the heaviest task is performed by a `mapvector` with only 3 workers (one for each subdomain), which explains the saturation point at 3 machines. Experiments with larger number of subdomains are on the way and we hope to include them in the final version of the paper.

8. Conclusions and Future work

[‖]`http://www.dicosmo.org/ocamlp3l/`

We summarized the main features of OcamlP3l 2.0. This new release radically changes the execution model, adds skeletons, and makes skeleton types more flexible, encouraging code conciseness and reuse. A few examples were discussed and some results given. The new features were profitably and fruitfully exploited also for programming a real size numerical application [5] which was then run successfully on the INRIA cluster at Roquencourt. In this case, 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 preceding 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!

Our plan for future work is to proceed on the practical and theoretical levels as follows. We intend to use OcamlP3l to program more complex numerical code. This will fertilize our work suggesting new ideas and tuning points for the skeletal system. Then we intend to enrich the the offer of OcamlP3l by providing a more general data parallel skeleton, generalizing the original skeleton in P3L[9]. Preliminary work on this subject has shown the potential of such an extension[17] particularly relevant to the huge matrices computation that are typical of the numerical problems we have to solve. Moreover, we would like to add to OcamlP3l more specific features for specific fields, for instance a general library or programming platform to solve numerical code-coupling problems. Finally, we would like to expand our theoretical work [13] to prove the prove the complete equivalence between the sequential and parallel semantics of OcamlP3l programs.

References

[1] M. Alt, H. Bischof, and S. Gorlatch. Algorithm design and performance prediction in a Java-based grid system with skeletons. In *EuroPar 2002*, pp. 899–906. 2002.

[2] B. Bacci, *et al.* . P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[3] L. Bettini, V. Bono, R. Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice, 2003.

[4] 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.

[5] F. Clèment *et al.* Parallel programming with the system applications to numerical code coupling. TR RR-5131, INRIA Rocquencourt, 2004

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

[7] R. Di Cosmo, Z. Li, M. Danelutto, S. Pelagatti, X. Leroy, and P. Weis. *OcamlP3L 2.0: User Manual*, 2005. `http://www.dicosmo.org/ocamlp3l/`.

[8] M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Parallel functional programming with skeletons: the ocamlp3l experiment. *The ML Workshop*, 1998.

[9] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in P3L. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of EURO-PAR '97, Passau, Germany*, volume 1300 of *LNCS*, pages 619–628, Berlin, August 1997. Springer.

[10] M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In *Proc. EuroPar 2000, Munchen*, volume 1900 of *LNCS*, pages 1175–1184, August 2000.

[11] 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. 1993.

[12] J. Darlington, Y. Guo, H. W. To, and J. Yang. Parallel Skeletons for Structured

Composition. In *Fifth ACM SIGPLAN PPoPP*. ACM Press, July 1995.

[13] R. Di Cosmo, Z. Li, and S. Pelagatti. A calculus for parallel computations over multi-dimensional dense arrays. *Computer Languages, Systems and Struct.*, 2005. To appear.

[14] A. J. Dorta, J. A. Gonzales, C. Rodriguez, and F. De Sande. llc: a parallel skeletal language. In *Proc. HLPP 2003, Paris*, 2003.

[15] K. Hammond and A.J. Rebon Portillo. HaskSkel: Algorithmic skeletons for Haskell. In *Proc. of IFL'99 Lochem, The Nederlands*, volume 1868 of *LNCS*, september 1999.

[16] H. Kuchen. A skeleton library. In B. Monien and R. Feldmann, editors, *Proc. of EuroPar 2002*, volume 2400 of *LNCS*, pages 620–629, August 2002.

[17] Z. Li. Efficient implementation of map skeleton for the ocamlp3l system. Master's thesis, Mèmoire de DEA, DEA Programmation, Université Paris VII, September 2003.

[18] S. Pelagatti. *Structured development of parallel programs*. Taylor&Francis, 1998.

[19] J. Sérot and D. Ginhac. Skeletons for parallel image processing: an overview of the SKiPPER project. *Parallel Computing*, 28(12):1785–1808, December 2002.