ECOLE NORMALE SUPERIEURE

OcamlP3l

a functional parallel programming system

Marco DANELUTTO
Roberto DI COSMO
Xavier LEROY
Susanna PELAGATTI

LIENS - 98 - 1

Département de Mathématiques et Informatique

CNRS URA 1327

OcamlP3l
a functional parallel programming system

Marco DANELUTTO*
Roberto DI COSMO
Xavier LEROY**
Susanna PELAGATTI*

LIENS - 98 - 1

March 1998

Laboratoire d'Informatique de l'Ecole Normale Supérieure
45 rue d'Ulm 75230 PARIS Cedex 05

Tel : (33)(1) 44 32 30 00
Adresse électronique : Roberto.Di.Cosmo@ens.fr

*Dipartimento di Informatica
Corso Italia, 40 Pisa Italy

Adresse électronique : marcod@di.unipi.it , susanna@di.unipi.it

** INRIA Rocquencourt, Domaine de Voluceau
Rocquencourt 78153 Le Chesnay Cedex - France

Adresse électronique : Xavier.Leroy@inria.fr

OcamlP3l
a functional parallel programming system *

Marco Danelutto
Dipartimento di Informatica
Corso Italia, 40
Pisa - Italy
marcod@di.unipi.it

Roberto Di Cosmo
LIENS-DMI
Ecole Normale Supérieure
45, Rue d'Ulm
Paris - France
Roberto.Di.Cosmo@ens.fr

Xavier Leroy
INRIA Roquencourt
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex - France
Xavier.Leroy@inria.fr

Susanna Pelagatti
Dipartimento di Informatica
Corso Italia, 40
Pisa - Italy
susanna@di.unipi.it

March 3, 1998

KEYWORDS: Implementation, Parallel and distributed languages, Functional and logic languages, Multiparadigm languages, Skeleton/template model, Massive parallelism, Language design, Internet programming languages.

*The OcamlP3l system has been partially funded by the Galileo bilateral France-Italy project n.97029
.

Abstract

Writing parallel programs is not easy, and debugging them is usually a nightmare. To cope with these difficulties, a structured approach to parallel programs using skeletons and template based compiler techniques has been developed over the past years by several researchers, including the P3L group in Pisa.

This approach is based on the use of a set of primitives that are just functionals implemented via templates exploiting the underlying parallelism, so it is natural to ask whether marrying a real functional language like Ocaml with the P3L skeletons can be the basis of a powerful parallel programming environment.

We show that this is the case: our prototype, written entirely in Ocaml using a limited form of closure passing, allows a very simple and clean programming style, shows real speed-up over a network of workstations and, as an added fundamental bonus, allows logical debugging of a user parallel program in a sequential framework without changing the user code.

Résumé

L' écriture de programmes parallèles est difficile, et leur mise à point est souvent un cauchemar. Pour pallier à ces difficultés, dans les dernières années différents groupes de recherche, dont celui de Pisa, ont developpé un approche structuré à la programmation parallèle qui se base sur des "squelettes" et des "modèles" de compilation.

Au coeur de cette approche on trouve un ensemble de primitives (les "squelettes") qui sont des vraies fonctions de rang deux, implementèes par des modèles de compilation qui utilisent le parallèlisme de la machine sousjacente. Il est donc naturel de se demander si le fait de fusionner les squelettes P3L avec un vrai langage de programamtion fonctionnel comme Ocaml peut fournir un environnement de developpement parallel avantageux.

Nous montrons ici que c'est bien le cas: notre prototype, écrit entièrement en Ocaml en utilisant une forme simple de transmission de fermetures sur des canaux, permet un style de programmation homogène, simple et élegant, exhibe un vrai *speed-up* sur un réseau de stations de travail et, en plus, permet la mise à point du code dans un environnement sequentiel, sans modifications du code écrit par l'utilisateur.

1 Introduction and Overview

Skeleton based parallel programming models [5, 6, 7, 1] provide to the user/programmer a set of *skeletons*, i.e. of second order functionals modeling common parallelism exploitation patterns. The programmer must use the skeletons to give parallel structure to his/her application and he/she uses a plain sequential language to express the sequential portions of the parallel application as parameters to the skeletons. He/she has no other way to express parallel activities but skeletons: no explicit process creation, scheduling, termination, no communication primitives, no shared memory, no notion of being executing a program onto a parallel architecture at all. All these details relative to parallel execution are actually dealt with by the skeleton compiler and/or run time system.

Objective Caml [13] (abbreviated `ocaml` in the sequel) is a functional language from the ML family [15]. It supports functions as first-class values: not only code pointers, but full mathematical functions, which may contain free variables. It is not a purely functional language, in that it is also equipped with full imperative power, in particular arrays modifiable in-place. This combination of features makes it well adapted to skeleton-based programming: skeletons are naturally higher-order functions (functions taking user-provided functions as arguments), while the sequential parts of the program are naturally written in the imperative style. Other useful features of `ocaml` include a powerful module system, allowing several implementations of the skeletons to be substituted for one another without recompiling the user code, and a built-in marshaler, allowing transmission of arbitrary data structures over byte streams, based on the same structural information used by the `ocaml` garbage collector.

OcamlP3l is a programming environment that allows to write parallel programs in `ocaml` according to the skeleton model supported by the parallel language p3l[1], provides seamless integration of parallel programming and functional programming and advanced features like sequential logical debugging (i.e. functional debugging of a parallel program via execution of the parallel code onto a sequential machine) of parallel programs and strong typing, useful both in teaching parallel programming and in building full-scale applications[2].

In this paper, we will first clearly state the goals of our system design, then briefly

[1] See URL `http://www.di.unipi.it/~susanna/p3l.html`

[2] In `http://qui.di.unipi.it/ocamlp3l.html` you will find relevant information, up to date references, documentation, examples, distribution code and dynamic web pages showcasing the OcamlP3l features.

recall the basic notions of the skeleton model for structured parallel programming, providing an informal parallel semantics as well as a formal sequential semantics. It will be then time to describe how we achieved our goals using to our advantage the flexibility of the Ocaml system, and we will conclude with some examples showcasing the power and flexibility of our approach.

1.1 The system design goals

The main goal of the project was to test the possibility to integrate parallel programming in a functional language using the skeleton model: after all, as we will briefly see later, skeletons are just functions, so a functional language should provide the natural setting for them. We also wanted to preserve the elegance and flexibility of the functional model, and the strong type system that comes with Ocaml. These goals have been achieved.

However, during the implementation of the system, it turned out that we could get more than that: in our implementation, the sequential semantics that is traditionally used to describe the functional behavior of the skeletons could actually be used to provide an elementary library allowing to execute the user code in a sequential mode, exactly as it is. This is a major advantage of the approach: in our system, the user can easily debug the logic of his/her program running it with the sequential semantics on a sequential machine using all the traditional techniques (including tracing and step by step execution which are of no practical use on parallel systems), and when the program is logically correct he/she is guaranteed (assuming the runtime we provide is correct) to obtain a correct parallel execution. This is definitely not the case of programs written using a sequential language and directly calling communication libraries/primitives such as the Unix socket interface or the MPI or PVM libraries, as the logic of the program is inextricably intermingled with low level information on data exchange and process handling.

Following this same idea (no changes to the user code, only different semantics for the very same skeletons), we also provided a "graphical semantics" that produces as a result of the execution of the user program a picture of the process network used during the parallel execution of the user program.

Finally, we wanted a *simple* way to generate from the user source code the various executables to be run on the different nodes of a parallel machine: here the high level of abstraction provided by functional programming, coupled with the ability to send closures over a channel among copies of the same program provided the key to an

elementary and robust runtime system that consists of a very limited number of lines of code.

But let's first of all introduce the skeleton model.

1.2 The skeleton model and P3L

The skeleton parallel programming model supports structured parallel programming [5, 7, 6]. Using this model, the parallel structure/behavior of an application has to be expressed by using *skeletons* picked up from of a set of predefined ones, possibly in a nested way. Each skeleton models a typical pattern of parallel computation and it is parametric in the computation performed in parallel. Skeletons can be understood as second order functionals that model the parallel computation coming out from the application of a given parallelism exploitation pattern to the parameter functions. As an example, pipeline and farm have been often considered to be in the skeleton set. A *pipeline* models the execution of a number of computations (stages) in cascade over the input data items. Therefore, the pipeline skeleton models all those computations where the function $f(x)$ to be computed can be decomposed as a cascade of functions $f_n(f_{n-1}(\ldots(f_2(f_1(x)))\ldots))$ and the computation of f_i onto different data items (either belonging to the input data (in case $i = 1$), or being the result of a previously evaluated f_{i-1}) takes place in parallel. A *farm* models the execution of a given function in parallel over a number of data items provided in input. Therefore the farm models all those computations where the same function $f(x)$ has to be computed over n input data items in parallel.

Using the skeletons, the choice of the form of parallelism to be exploited is completely in charge of the programmer, while the efficient exploitation of the skeleton structure of any applications is partially or completely in charge of either the run time support or the compiler. This means, for instance, that a programmer has no responsibility in deriving code for creating parallel processes, mapping and scheduling processes on target hardware, establishing communication frameworks (channels, shared memory locations, etc.) or performing actual interprocess communications.

In some cases, the compiler/run time environment also computes some parameters such as the parallelism degree or the communication grain needed to optimize the execution of the skeleton program onto the target hardware [2].

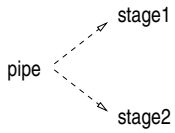
The skeletons we consider in this paper are basically of two kinds:

- *control parallel* skeletons, modeling parallelism exploited between processing activities relative to different input data. In this set we have: seq (*cf. 1.4.1*),

pipe (*cf. 1.4.2*), farm (*cf. 1.4.3*), loop (*cf. 1.4.4*). Such skeletons correspond to the usual control parallel skeletons appearing both in p3l and in other skeleton models [5, 7].

- *data parallel* skeletons, modeling parallelism exploited between processing activities relative to parts of the same input data. In this set, we provide map (cf. 1.4.6) and reduce (cf. 1.4.5). In the current release, these skeletons are not as powerful as the corresponding skeletons of p3l; instead, they closely correspond to the map and reduce functionals of the Bird-Meertens formalism discussed in [3].

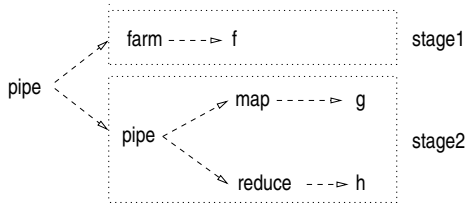
As an example, let us suppose a programmer wants to write an application whose behavior is clearly divided in two consecutive phases (stages). Therefore, he/she finds that the pipeline is the proper skeleton modeling this "two phase" behavior, and builds an OcamlP3l program whose parallel structure (in terms of skeletons) is the following:



Let us also suppose the programmer recognizes that the first stage is computationally intensive (and corresponds to the function f), but it cannot be further decomposed using the available skeletons, and that the second stage actually first applies an operation to all the elements of a (vector) data structure (the function g) and then "sums" up the results by using an associative and commutative operator h.

By looking at the available skeletons and at their semantics, the programmer understands that the farm skeleton is useful to enhance the parallelism in the computation of the "heavy" function f, as well as the map and the reduce skeletons exactly model the behavior of the second stage. The *map* skeleton models the parallel application of a generic function f to all the items of a vector data structure, whereas the *reduce* skeleton models the parallel computation folding a generic function f over the items of a vector data structure.

Therefore the programmer may refine the skeleton structure of his/her application by using the following skeletons:



The first stage has been "farmed out" in such a way that it can process input data exploiting control parallelism, whereas in the second stage further (data) parallelism has been introduced by introducing the pipeline of a map and a reduce skeletons.

These two "application outlines" (skeleton structures) correspond to the following (incomplete) OcamlP3l code:

1. ```
   let stage1 x = ... ;;
   let stage2 x = ... ;;
   let prog () = seq(stage1) ||| seq(stage2);;
   ```

2. ```
   let f x = ... ;; let g x = ... ;; let h x = ... ;;
   let prog () =
       seq(farm(seq f),3) ||| (map((seq g),5) ||| reduce(seq h));;
   ```

(In the code, the integer parameters to `farm` and `map` functions represent the parallelism degree the user requires for the implementation of those skeletons) Section 2.2 will explain how you can develop OcamlP3l programs.

Currently, the user is supposed to explicitly give the number of processors to be used in each farm and map skeleton. In other words, the choice of the parallelism degree of such skeletons is up to the programmer. It is foreseeable in a future release to ask the system to guess optimal values depending on available resources (following the approach of p3l [2]), but we avoided such complication for the time being.

Applications with a parallel structure given by skeletons (such as the one outlined above) can be implemented by using *implementation templates* [16]. An implementation template is a known, parametric way of exploiting the kind of parallelism modeled by a skeleton onto a particular target architecture. As an example, a template corresponding to the map skeleton will take some input vector data, it will split the data into chunks holding one or more data items of the vector, schedule them to a set of "worker" processes computing the map function f and, finally, collect the results and rebuild the output vector data structure. All of these operations will be performed by some processes, using either communications or shared memory for data communication. Such a template must, as its primary goal, implement in an efficient way the map skeleton and therefore:

- it must work with any kind of map function f, i.e. must be parametric with respect to the input and output data types (`'a` and `'b`, in case the function f has type `'a` $\rightarrow$ `'b`)

- it must support any reasonable parallelism degree, i.e. it must work (and provide effective parallelism exploitation) when executed on an arbitrary number of processing elements.

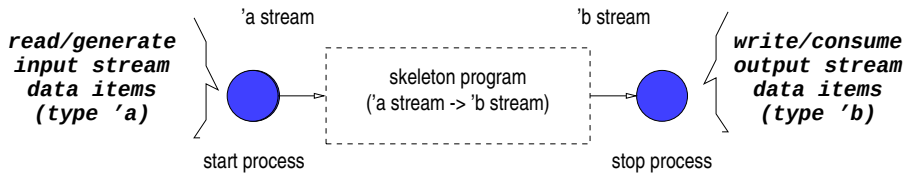
Although this feature is not currently exploited in the OcamlP3l prototype, implementation templates come in most cases with an associated *analytical performance model* able to compute the expected performance of a template once some parameters are known (both machine dependent ones, such as the communication latency and bandwidth, and user code dependent, such as the average time spent in computing user defined (sequential) functions) [18].

In the general case, such models take the amount of processing resources used as a parameter. Therefore, they can be used by the skeleton language compiler or run time support to devise the proper values (i.e. those maximizing the speedup) for parameters such as the parallelism degree of a farm [3].

1.3 The skeletons in OcamlP3l: sequential semantics

We define the sequential semantics of the OcamlP3l skeletons in terms of the interpreter computing the functional results of an OcamlP3l program using the skeletons. We assume that a program always begins with a special form `startstop` corresponding to the closure of the process network via a start and a stop node which respectively generate and absorb the data flow of information on the network (i.e. "generate" (possibly by reading a file) the data items that have to be processed and "display" (possibly by writing to a file) the data items output by the skeleton process network). The `start` and `stop` processes also take care of computing initialization functions (on the input data stream, `start`, and output data stream, `stop` side) as well as termination functions (at the output data stream side).

[3] Currently, the user is supposed to explicitly give the "proper" values for these parameters. We leave as future work the integration of suitable performance models for the implementation templates used within OcamlP3l, in such a way that these computations could be performed by the OcamlP3l runtime system.



This assumption allows us to give the other skeletons a very simple semantics which describes how they act on a single data item, and not on the stream of items flowing through it during the execution: all the stream behavior is really concentrated in the `startstop` combinator, which simulates the stream of data by repeatedly iterating the semantics of the network on the items generated by the `start` node and passing the results to the `stop` node.

Hence, even if we will often describe the skeletons as operators over `'a stream`, in our current implementation we will be allowed to assume `'a stream` to be just `'a`.

Here a formal sequential semantics for the skeletons follows, expressed in `ocaml` syntax.

```
let seq(f) = f;;
let farm (f,n) = f;;
let (|||) f g = fun x -> g (f x);;
let rec loop(c,f) = function x ->
    let r = f x in if c r then loop (c,f) r else r ;;
let mapvector (f,n) = (Array.map f);;
let reducevector (f,n) v =
  let rec reduce accu idx =
    if idx >= Array.length v
    then accu
    else reduce (f(accu,v.(idx))) (idx+1)
  in reduce v.(0) 1;;
let startstop (f1,init1) (f2,init2,finalize) expr =
  init1();init2();
  try
    while true do f2(expr (f1 ())) done
  with End_of_file -> finalize();;
```

1.4 The skeletons in OcamlP3l: informal parallel semantics

Here we briefly give an informal semantics of the skeletons when executed with the parallel library of OcamlP3l.

1.4.1 seq

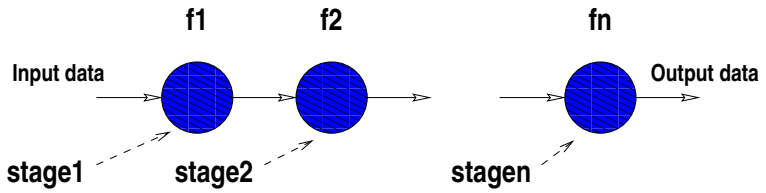
Functionally *seq f* has type $'a\ stream\ \rightarrow\ 'b\ stream$ provided that the function f has type $'a\ \rightarrow\ 'b$: it is this skeleton that converts a usual sequential function into a process of the parallel network described by the skeleton structure.

1.4.2 pipe

Functionally, *pipe* $f_1 \ldots f_n$ has type $'a_0\ stream\ \rightarrow\ 'a_n\ stream$ provided that the skeletons $f_1 \ldots f_n$ respectively have type $'a_0\ stream\ \rightarrow\ 'a_1\ stream, 'a_1\ stream \rightarrow\ 'a_2\ stream, \cdots,$ $'a_{n-1}\ stream\ \rightarrow\ 'a_n\ stream$ Given the input stream $x_m : ... : x_1 : x_0$, the n-stage pipe `f1 ||| f2 ||| ... ||| fn` computes the output stream

$$f_n(...(f_2(f_1\ x_m))...) : ... : f_n(...(f_2(f_1 x_1))...) : f_n(...(f_2(f_1\ x_0))...)$$

In terms of (parallel) processes, a sequence of data appearing onto the input stream of a pipe is submitted to the first pipeline stage. This stage computes the function f_1 onto every data item appearing onto the input stream. Each output data item computed by the first stage is submitted to the second stage, computing the function f_2 and so on until the output of the $n-1$ stage is submitted to the last stage. Eventually, the last stage delivers its own output onto the pipeline output channel. The resulting process network looks like the following:



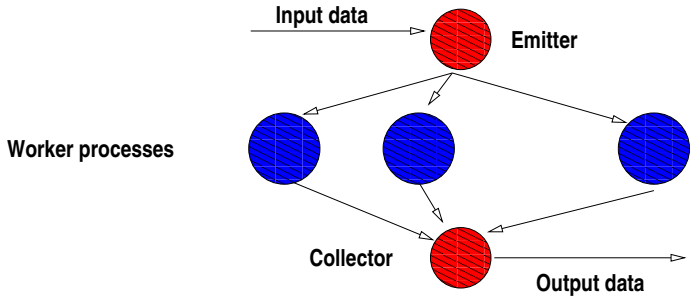
where the circles represent processes and the arrows represent communication channels.

1.4.3 farm

Functionally *farm f* has type $'a\ \ stream\ \rightarrow\ 'b\ \ stream$ provided that the skeleton f has type $'a\ stream\ \rightarrow\ 'b\ stream$.

Given the input stream $x_m : ... : x_1 : x_0$ the farm `farm(f,n)` computes the output stream $f(x_m) : ... : f(x_1) : f(x_0)$ by using n worker processes.

In terms of (parallel) processes, a sequence of data appearing onto the input stream of a farm is submitted to a set of worker processes. Each worker process applies the same function to the data items received and delivers the result onto the output stream. The resulting process network looks like the following:



The emitter process takes care of task-to-worker scheduling (possibly taking into account some load balancing strategies), while the collector process takes care of reordering the output data items with respect to the input ordering and of delivering them onto the output data stream.

1.4.4 loop

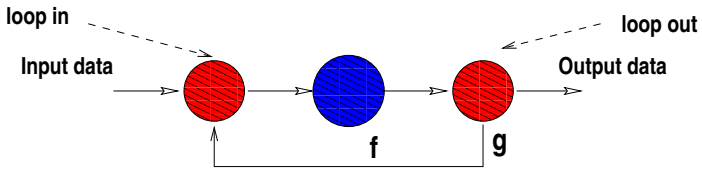
Functionally, a *loop* has type $'a\ stream \rightarrow 'a\ stream$ provided that the skeleton f has type $'a\ stream \rightarrow 'a\ stream$ and the function g has type $'a \rightarrow bool$.

Every data item *x* appearing onto the input stream is processed by a function *f*. The result is processed by using another (boolean function) *g*. If the result of *g(f x)* is true, then *(f x)* is submitted again to the loop input, otherwise, *(f x)* is delivered onto the output stream.

Given the input stream $x_m : ... : x_1 : x_0$ the loop *loop(g,f)* computes the output stream $f(f(f...(f\ x_m)...)) : ... : f(f...(f\ x_0)...)$, where the number of times f is computed onto each stream item depends on the number of times the function g computed onto the same data holds true.

In terms of (parallel) processes, a sequence of data appearing onto the input stream of a loop is submitted to a *loop-in* stage. This stage just merges data coming from the input channel and from the feedback channel and delivers them to the *loop-body* stage. The `loop-body` stage computes f and delivers results to the *loop-end* stage. This latter stage computes g and either delivers (f x) onto the output channel (in case $(g(fx))$ turns out to be `true`) or it delivers the value to the `loop-in` process along

the feedback channel ($(g(fx)) = \texttt{false}$). The resulting process network looks like the following:



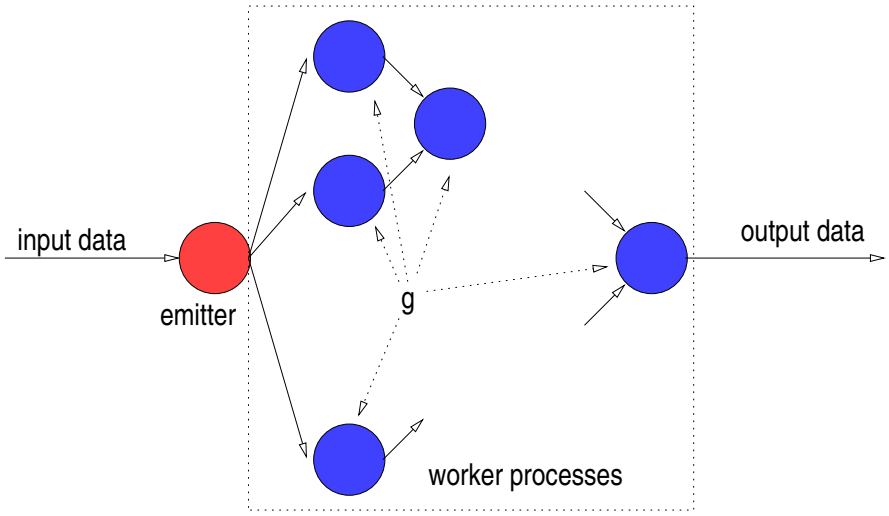
1.4.5 reduce

Functionally, *reduce* g has type $'a\ array\ stream\ \rightarrow\ 'a\ stream$ provided that the skeleton g has type $'a\ stream\ \rightarrow\ 'a\ stream\ \rightarrow\ 'a\ stream$.

Given the input stream $x_n : ... : x_1 : x_0$ the reduce *reduce(g)* computes the output stream $(fold\ g\ x_n) : ... : (fold\ g\ x_1) : (fold\ g\ x_0)$ where we have

```
let rec fold g = function
        [x] -> x
    | x::rx -> g(x (fold g rx));;
```

In terms of (parallel) processes, a vector data item appearing onto the input stream of a map is processed by a logical tree of processes. Each one of the processes is able to compute the function g. The resulting process network looks like the following tree:



In this case, the emitter process is the one delivering either couples of input vector data items or couples of sub-vectors of the input vector to the processes belonging to the tree base. In the former case, $log(n)$ levels of processes are needed in the tree, in the latter one, any number of process levels can be used, and the number of sub-vectors to be produced by the emitter can be devised consequently.

1.4.6 map

Functionally, *map f* has type $'a\ array\ stream\ \rightarrow\ 'b\ vector\ stream$ provided that the skeleton f has type $'a\ stream\ \rightarrow\ 'b\ stream$.

Given the input stream $x_m : ... : x_1 : x_0$ the map *map(f,n)* computes the output stream $(Array.map\ f\ x_m) : ... : (Array.map\ f\ x_1) : (Array.map\ f\ x_0)$ by using n worker processes.

In terms of (parallel) processes, a vector data item appearing onto the input stream of a map is split in components and each component is submitted to one worker computing f. Each worker process applies the same function to the data items received and delivers the result onto the output stream. The resulting process network looks like the one of the farm shown above. In this case, however, for every data item `[|x1;...;xm|]` appearing onto the emitter input data stream, the emitter may take different choices relative to data splitting/worker scheduling. As an example:

- it may round robin each `xi` to the workers (`{w1,...wn}`). The workers in this case simply compute the function $f :' a \rightarrow' b$ over all the elements appearing onto their input stream (channel).

- it may split the input data vector in exactly `n` sub-vectors to be delivered one to each one of the worker processes. The workers in this case compute an $(Array.mapf)$ over all the elements appearing onto their input stream (channel).

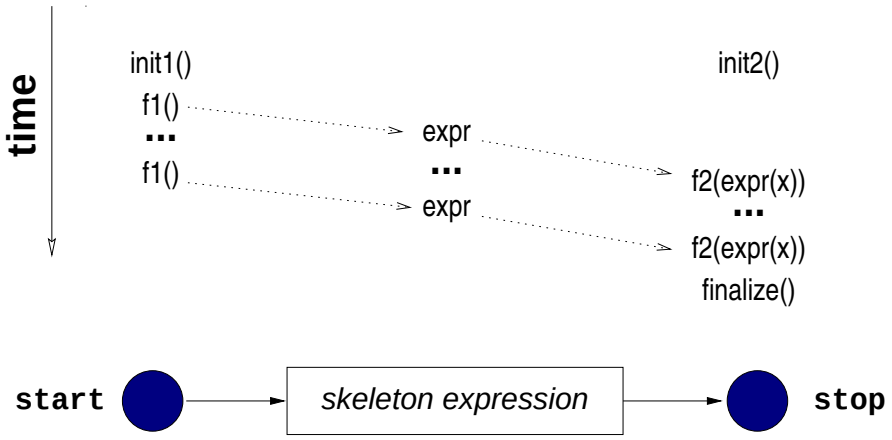
The emitter process takes care of (sub)task-to-worker scheduling (possibly implementing some kind of load balancing policy). The collector process takes care of rebuilding the vector with the output data items and of delivering the new vector onto the output data stream. The resulting process network looks like the one for `farm`, but here the emitter (and collector) which respectively splits incoming arrays into smaller packets, dispatches them to the workers and then recomposes data into outgoing arrays.

1.4.7 startstop

Although `startstop` is not a skeleton, here we discuss its informal parallel semantics, which is needed in order to understand how OcamlP3l are actually executed.

Functionally, *startstop* has type $(unit \rightarrow' a) * (unit \rightarrow' b) \rightarrow ('c \rightarrow' d) * (unit \rightarrow' e) * (unit \rightarrow unit) \rightarrow ('a \rightarrow' c) \rightarrow unit$ where $(unit \rightarrow' a)$ is the type of a function generating the data items of the input data stream of the whole skeleton program, $('c \rightarrow' d)$ is the type of the function "consuming" (e.g. displaying to screen, writing to

files) the data items of the output stream of the whole skeleton program, $(unit \rightarrow' b)$ and $(unit \rightarrow' e)$ are the types of the initialization functions at the `start` side and at the `stop` side, respectively, $(unit \rightarrow unit)$ is the type of the finalization (termination) function computed at the `stop` side, and finally $('a \rightarrow' c)$ is the type of the skeleton program (expression) we want to execute.



In terms of (parallel) processes, two processes are executed, beyond the ones belonging to the skeleton program: the first one, the `start` process, executes an initialization function, then starts executing a user supplied function that produces the data items of the input data stream. The second one, the `stop` process, executes an initialization function, then starts receiving and processing the output stream data items according to a user supplied function.

2 The skeleton syntax in OcamlP3l

We are now in a position to introduce the syntax we choose for writing OcamlP3l programs.

2.1 Skeleton syntax

Skeletons are denoted as functions:

- *farm* is denoted by a binary function `farm` whose first argument is the skeleton expression computed by the farm workers and whose second argument is the number of workers that have to be included in the farm.
- *pipe* is denoted by the binary infix operator `|||`.

- *map* is denoted by a binary function `mapvector` whose first argument is the skeleton expression computed by the map workers and whose second argument is the number of workers that have to be included in the map.

- *reduce* is denoted by the binary function `reducevector` whose first argument is the skeleton expression denoting the associative and commutative binary function that has to be computed by the reduce workers and whose second argument is the number of workers that have to be included in the reduce logical tree.

2.2 Structure of a program

An OcamlP3l program is syntactically built out of the following items:

- the `open` directives needed to compile and run the program. The OcamlP3l-specific directives are the following ones:

  - ```
    open Seqp3l;;
    ```
    in case you want sequential execution

  - ```
    open Parp3l;;
    open Nodecode;;
    open Template;;
    ```
    in case you want parallel execution

  - ```
    open Graphp3l;;
    ```
    in case you want the graph of the process network

  Note that these include directives are the only things you need to change in order to run the skeleton code sequentially or in parallel.

- the sequential `ocaml` code defining the functions/data used within the skeleton code.

- a function that, when applied to the unit element `()`, computes a legal skeleton expression, i.e. an expression built out of sequential code and skeleton combinators; the skeleton expression must be enclosed in a `startstop` combinator.

- a *pardo* expression applied to the function computing the legal skeleton expression. The goal of the pardo expression is to evaluate the skeleton expression, in parallel or sequentially or graphically, according to the modules opened. Since
    ```

in the parallel evaluation the pardo expression is executed by all the nodes participating in the computation, it is necessary to ensure that only the root node performs the actual parsing/mapping/optimization/allocation work hidden behind the skeleton expression. This can be achieved by evaluating the skeleton expression lazily, which can be easily done here by the usual "freeze/thaw" trick: pardo takes not the expression directly, but a function that encapsulates the expression, which is thus "frozen" and can be either "thawn" and executed in the root node, or thrown away without execution on the other nodes.

2.2.1 An example

Suppose we want to write a parallel OcamlP3l application plotting the Mandelbrot set. First of all, we will write code that produces the color of a pixel in the plane, provided the resolution, the plane coordinates and the position of the pixel are provided. The following functions computes the color of a pixel and the color of a whole row of pixels:

```
let color_pixel (x0,y0,x1,y1) n res i j =
  let dx = (x1-.x0)/.(float n) and dy = (y1-.y0)/.(float n) in
  let zr = ref (y0 +. (dy *. (float i)))
  and zi = ref (x0 +. (dx *. (float j))) in
  let cr = ref (!zr)
  and ci = ref (!zi) and col= ref 0
  and comp_col c res = Pervasives.truncate
      (((float c)/.(float res))*.(float Graphics.white)) in
  begin
    for k=0 to (res-1) do
      if(!zr *. !zr +. !zi *. !zi <= 4.0)
      then begin
        zi := 2.0 *. !zr *. !zi *. !ci;
        zr := !zr *. !zr -. !zi *. !zi +. !cr;
        col:= k
      end
    done; (comp_col !col)
  end;;
let plot_mandel_row (x0,y0,x1,y1) n res j =
  let line = Array.create n black in
  for k = 0 to n - 1 do
```

```
    line.(k) <- (color_pixel (x0,y0,x1,y1) n res k j) res
  done; (j, line);;
```

Now, in order to compute in parallel the rows that have to be plotted on the screen, we can set up a function generating a stream of rows to be plotted. The row data type will include plane coordinates as well as resolution and row index:

```
let gen_tasks (x0,y0,x1,y1) n res =
  let rec gen_tasks0 (x0,y0,x1,y1) n res k =
    if(k=(n-1)) then []
    else (k,(x0,y0,x1,y1),n,res)
          ::(gen_tasks0 (x0,y0,x1,y1) n res (k+1))
  in  gen_tasks0 (x0,y0,x1,y1) n res 0;;
let start =
  (* ok: these are just constants drawing a nice picture ... *)
  let tasks = ref (gen_tasks (0.704,0.704,0.709,0.709) 500 500)
  in function () ->
      match !tasks with
        (t0::rest) -> (tasks := rest; t0)
      | [] -> print_string "Hit return to finish";
              print_newline();
              let _ = read_line() in raise End_of_file;;
```

The `gen_tasks` function generates a list of tasks to be computed. Each task represent a row of pixels that have to be coloured. `start` return one of these rows at a time and it is actually used to generate the input stream. The idea is to use such a stream to feed a *farm* skeleton, where each worker actually computes the colors that have to be displayed onto a given row. The following code sets up the *farm* skeleton stuff, including the functions used to initialise the `start` and `stop` node (`start` and `stopfinit`) and the function called at the `stop` node when the computation terminates (`stopfinalize`):

```
let compute_a_line (row,interval,n,res) =
    (plot_mandel_row interval n res row);;
let display_a_line (j,col) =
  draw_image (make_image [| col |]) 0 j;;
let start =
  let tasks = ref (gen_tasks (0.704,0.704,0.709,0.709) 500 500)
```

```
  in function () ->
       match !tasks with
         (t0::rest) -> (tasks := rest; t0)
       | [] -> print_string "Hit return to finish";
               print_newline();
               let _ = read_line() in raise End_of_file;;
let stopinitf() = print_string "opening...";
  print_newline();open_graph " 500x500";;
let stopfinalize() = Unix.sleep 30;
  print_string "finishing... type ENTER to close down";
  print_newline(); let _ = read_line() in ();;
let mandelexpr () =
  startstop
    (start,fun()->())
    (display_a_line,stopinitf,stopfinalize)
    (farm(seq(compute_a_line),10))
in pardo mandelexpr;;
```

All these functions may be grouped into a file (`mandel3.ml`, say) starting with the usual includes:

```
open Graphics;;
open Seqp3l;;
```

and compiled with the command:

```
ocamlc  -custom graphics.cma seqp3l.ml -o mandel3.seq mandel3.ml
        -cclib -lgraphics -cclib -L/usr/X11R6/lib -cclib -lX11
```

in order to get the sequential executable. Note that most of the libraries have to be linked just because we use the `Graphics` `ocaml` library.

In case we want to generate the parallel executable, we change the include line

```
open Seqp3l;;
```

to

```
open Parp3l;;
open Nodecode;;
open Template;;
```

and we simply compile again with the command:

```
ocamlc -custom unix.cma graphics.cma p3lpar.cma -o mandel3.par
       mandel3.ml -cclib -lunix -cclib -lgraphics
       -cclib -L/usr/X11R6/lib -cclib -lX11
```

and we get the parallel code `mandel3.par`. Now, in order to run the sequential code we simply have to issue the command:

```
mandel3.seq
```

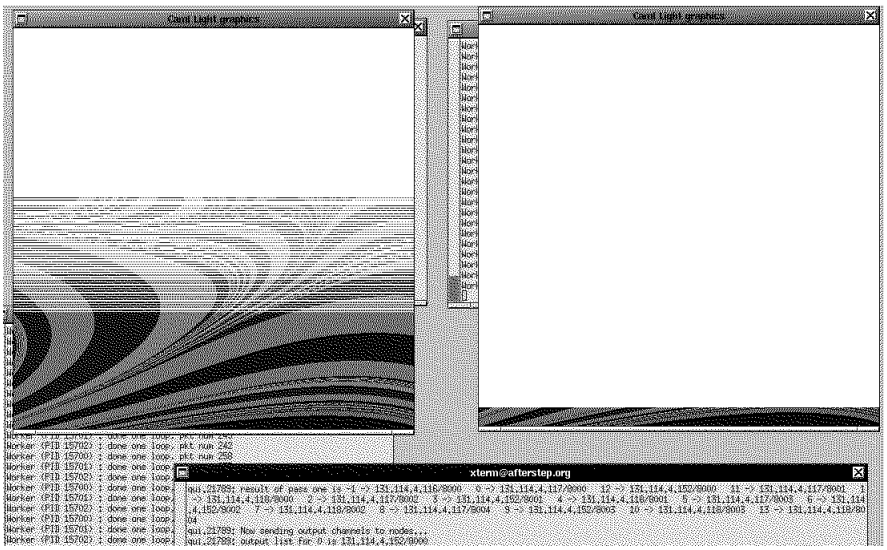
whereas in order to run the parallel code we must first issue a:

```
rsh wsX.my.domain.edu:mandel3.par
```

command for each one of the workstations (`wsX` within the domain `my.domain.edu`) we want to use (provided the `mandel3.par` code is accessible via `NFS`, overwise we must `rcp` it explicitly), and then issue the command:

```
mandel3.par p3lroot ws1.my.domain.edu ... wsN.my.domain.edu
```

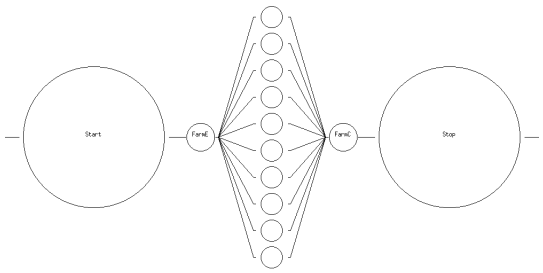
The overall application shows a linear speedup on a small number of workstations (i.e. on a small number of worker processes in the farm), due to the fact that each row of the picture takes a long time to be computed. By simultaneously launching the sequential Mandelbrot code (the one exploiting the sequential skeleton semantics) on a machine and the parallel one (the one exploiting the parallel skeleton semantics) on a cluster of workstation the result will look like the following:



The left graphic display is the one coming out from the parallel execution, while the right one is the one coming from the sequential code. Sequential and parallel code only differ for the semantic file include and have been started at the same time[4].

One can then look at the performance, i.e. consider the number of workstations used and the time spent in the computation, and consider if it is the case to vary either the number of workers appearing in the farm skeleton within the code or the number of workstations used for executing the parallel code.

In so doing, it is often useful to look at the application process network by linking the graphical skeleton semantics via a `open GrafP3l;;` statement. For example, for the above program the output will be the following (we specified 10 workers in the farm):



Now, one can ask the following question: *as we are actually plotting something which is a logical vector of lines, why can't we use a* `map` *skeleton instead of the* `farm` *one?*

Well in this case we must modify our program in such a way that:

- `start` delivers a stream of row vector, instead of delivering single row data structures,
- the `stop` node displays whole picture instead of displaying rows of a picture,
- skeleton code uses a `mapvector` call instead of a `farm` one.

This can be done by simply modifying our previous code as follows:

```
let gen_tasks interval n res =
  let tasks = Array.create n (0, interval, n, res) in
  for i = 0 to (n - 1) do
```

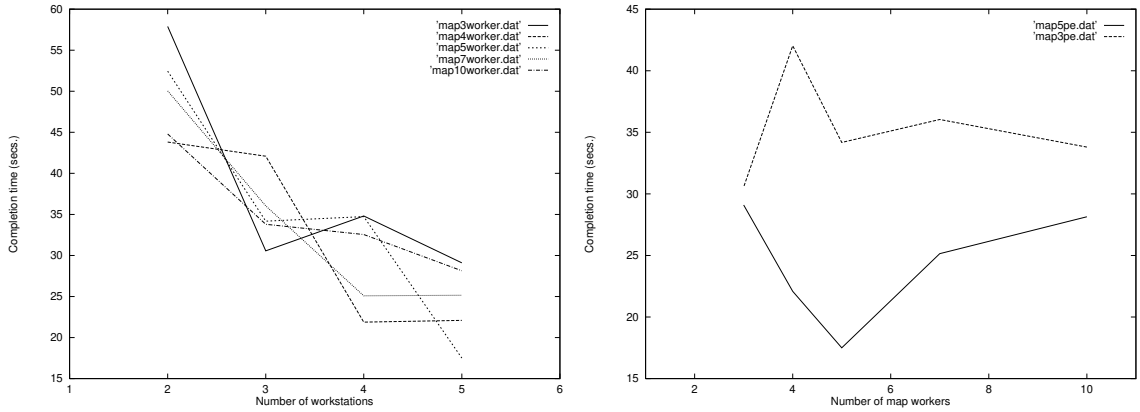
[4]We can notice that the left display gets filled much faster than the right one. By performing precise time profile activity, we found a linear speedup of the parallel code with respect to the sequential one when using up to 4 workstations.

```
    Array.set tasks i (i,interval,n,res)
  done; [tasks];;
let display_image img =
  for i = 0 to (Array.length img)-1 do
    let (j,col) = img.(i) in
    draw_image (make_image [| col |]) 0 j
  done;;
let mandelexpr () =
  startstop
    (start,fun()->())
    (display_image,stopinitf,stopfinalize)
    (mapvector(seq(compute_a_line),10)) in
    pardo mandelexpr;;
```

By moving from the `farm` code to the `mapvector` code, we actually changed the kind of parallelism exploited in the Mandelbrot picture computation: in the former case, we artificially built a stream of "rows" to be plotted and we exploited control parallelism between the computation of each row, whereas in second case we just built the vector of rows and exploited data parallelism in the computation of the whole picture[5].

An idea of the effect of varying the number of workers included in a skeleton with respect to varying the number of processing elements used (i.e. of the performance tuning activity a programmer must perform using the current version of OcamlP3l) is given in the following pictures. Here, we plotted the time spent in computing eight different views of the Mandelbrot set, by using the map version of the program just outlined above.

[5] At the risk of repeating ourselves, we would like to draw again here the reader's attention to the fact that such an apparently elementary modification would require a *huge* amount of work on a program written in, say, C plus MPI calls.



The left picture shows the completion time of the mandelbrot program with respect to the number of processing elements (workstations, actually) we used to run the program. Different plot lines refer to mandelbrot programs specifying different parallelism degree in the `map` skeleton. E.g. `map3worker.dat` plot refers to an instance of the Mandelbrot program where `mandelexpr` is defined as:

```
let mandelexpr () =
  startstop
    (start,fun()->())
    (display_image,stopinitf,stopfinalize)
    (mapvector(seq(compute_a_line),3)) in
    pardo mandelexpr;;
```

whereas `map10worker.dat` plot refers to an instance of the Mandelbrot program where `mandelexpr` is defined as:

```
let mandelexpr () =
  startstop
    (start,fun()->())
    (display_image,stopinitf,stopfinalize)
    (mapvector(seq(compute_a_line),10)) in
    pardo mandelexpr;;
```

The right picture shows the effect of varying the amount of workstations used to execute the Mandelbrot program. In particular, the `map5pe.dat` plot refers to usage of 5 workstations and the `map3pe.dat` plot refers to the usage of 3 workstations.

The program compiled by using the sequential semantics took something like 84.60 secs to complete on a 120 MHz Pentium PC LINUX machine, while it took just 37.41

secs on a 200 MHz Pentium Pro machine (in both cases we used bytecode compilation, not the one producing native code). The measures have been taken using PCs during normal user activity, i.e. when other programs and daemons where running over the PCs. By looking at the time plot, we see that in general, the best results are achieved when a number of workers in the map is chosen such that it is slightly higher than the number of processing elements used. This because different processes belonging to the implementation template process network can be mapped onto the same machine. The resulting "parallelism excess" helps hiding the communication latencies that are particularly high when using Unix sockets.

3 Implementing OcamlP3l

Now, let us point out the peculiar features relative to the implementation of OcamlP3l. First, we will discuss the mechanism used to implement different processes onto different nodes, by exploiting a particular form of "closure communication". Then, we will point out some details relative to the interprocess communication layer and we will motivate the choice of the Unix sockets as the OcamlP3l communication layer. Finally, we will discuss some details of the templates we used to implement the skeletons provided by OcamlP3l.

3.1 Closure passing as distributed higher order parameterization

A sequential implementation of an OcamlP3l program is quite easy to provide: just use a library `seqp3l.ml` containing *precisely* the definitions given in section 1.3[6]. The type safety is a direct consequence of the fact that we are not using here anything from outside the safe core of Ocaml.
Similarly, providing the graphical semantics poses no real challenge.

But what about the parallel semantics? What is the right way to implement such a thing? We must guarantee the type safety and ensure that the runtime is reasonably small as to allow the verification of its properties, which will become an important point in industrial applications. Both points posed problems which we overcame during the development of the system.

First of all, to ensure that the system is manageable and safe, we immediately discarded the approach based on parsing the source file to extract the code corresponding

[6]Indeed, that is the *actual* code in the system.

to each node of the network: this would impose to use external tools to perform an analysis of the user code which is difficult, error prone, and whose semantics would have a very unclear status.

Instead, we choose to use an SPMD approach: all the nodes of the network will run the *same* program (in a sense this is the "template process interpreter code", as we will see in while) , which will be the result of the compilation of the full user code, and a control node[7] will dispatch to the rest of the other nodes in the network the parameterization information needed to specialize it to the particular function it is really supposed to perform (`start` node, `stop` node, emitter, collector, sequential node running a given function f, etc.).

In order to achieve this behavior, the control node performs the following tasks:

- executes the skeleton expression, which has as a consequence to build a data structure describing the process network. From this data structure, we compute behind the scenes the configuration information for each node in the process network.
- executes the `pardo` expression: this has the following effect
  - maps virtual nodes to the processor pool given on the command line,
  - initializes a socket connection with all the participating nodes,
  - gets the port addresses from each of them,
  - 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),
  - 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).

[7]The control node runs the same program as the others, but it is invoked by the user with a special designating option `-p3lroot`.

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 contain a modified marshaling library, originary 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 parameterization that kept the runtime code to a minimum size (the source codes of the full system is less than twenty kilobytes).

3.2 Communication and process support

As far as the general mechanism of closure passing is concerned, no particular requirement/restrictions have been posed onto the physical communication implementation. Even considering the fact that we need to move data between the different processes making up the parallel implementation of an OcamlP3l program, we derived no particular constraint onto the communication layer.

Thus, at the very beginning of the OcamlP3l project, we faced the problem of choosing a suitable communication system. We had as a goal to come out with the maximum "portability" of OcamlP3l. Furthermore, we wanted to fully demonstrate the feasibility of integrating the parallel skeleton world within a functional framework. These two goals had priority over the classical "efficiency and performance" goal one usually has to achieve when dealing with parallelism.

The result is that we have adopted the plain Unix socket world as the communication layer. This has some (very) positive consequences on the overall OcamlP3l design:

- the socket communication support is available on any Unix systems, and it turns out to be available even in the Windows world, even if in this case reliability is often a problem,

- no particular customization of the support is needed to match the OcamlP3l features,
- the point-to-point, connection oriented, stream model provided by Unix sockets is perfect to model data streams flowing around between the processes belonging to the process network derived by OcamlP3l to implement the user skeleton code,
- last but not least, there was an existing and suitable `ocaml` interface to Unix system calls, including those relative to sockets.

On the down side, the adoption of Unix sockets presents an evident disadvantage which is the low performance achieved in communications (a raw synchronization (i.e. zero length data communication) takes several milliseconds to be performed, even in those cases when the data transmission media turns out to be free, i.e. no collisions are detected).

At the moment, we are considering to use in the next version of OcamlP3l a communication layer based on an optimized communication library such as MPI [14], as an efficient alternative to the socket communication layer, which will be nevertheless retained for its ease of deployment, that makes it attractive for programming courses.

This will require some modifications in the template code used within OcamlP3l, and will not necessarily completely solve the performance problems of the socket communication layer when run on a network of computers, where most MPI libraries are still implemented using sockets, but will allow to target real multiprocessor machines where MPI is efficiently implementes, without touching the code. Also, we will be able to delegate to the MPI system the administrative tasks involved in copying and launching the programs on the different machines.

As far as the process model is concerned, we felt happy with the Unix one. All we need is a mechanism allowing an instance of the template interpreter (the one specialized by using the closure passing mechanism) to be run onto different workstations belonging to a local area network. The Unix `rsh` mechanism matches this requirement. Note that, as processes are generated and run on different machines just at the beginning of the OcamlP3l program execution, any considerations about performance in `rsh`-ing processing is irrelevant.

3.3 Template implementation

OcamlP3l implements each skeleton appearing in the application code by generating

a proper instantiation of the corresponding implementation template. In OcamlP3l, a single implementation template is provided for each one of the skeletons supported. The implementation templates provided within the current prototype closely resemble the ones discussed in the informal parallel semantics section (Section 1.4). Actually, only the *reduce* template is slightly different, in that the tree discussed in Section 1.4 is actually implemented by a process network similar to the one discussed from the farm, where partially evaluated data is iteratively passed back from the collector to the emitter process.

Each template appearing in OcamlP3l:

- is parametric in the parallelism degree exploited As an example the farm template may accommodate any positive numbers of worker processes. Currently, the programmer must specify this parameter, which is actually taken from the second parameter of a $farm(F, n)$ skeleton call.

- is parametric in the function computed as the body of the skeleton For instance, the farm skeleton accepts as a parameter the function that has to be computed by the worker processes. This function is to be a skeleton itself. Therefore, either it is a `seq` skeleton call, modeling a sequential computation or it is another skeleton call modeling a parallel computation. In the former case, the skeleton is implemented by a process network whose workers just perform the sequential computation `f` denoted by some `seq(f)`. In the latter case, each worker process is itself a process network known by the emitter and collector processes implementing the farm just as channels where data has to be delivered/fetched.

- provides a set of process templates i.e. parametric process specifications that can be instantiated to get the real process codes building out the implementation template process network. As an example, consider again the farm template. The emitter process behavior can be fully specified by the data type of items that have to be processed, by the channel from which those data items have to be read and by the set of channels onto which the data items have to be scheduled (written) to the worker processes, possibly with some "clever" (e.g. achieving load balancing) scheduling strategy. Such a process can be completely specified by providing a function

```
farmetempl (OutChanSel f) ic ocl
```

whose first parameter provides the worker scheduling function, the second one provides the input channel where data has to be fetched and the third one provides the set of channels used to deliver tasks to be computed to the farm workers. The type of such a function turns out to be

```
val farmetempl : ('a, 'b) Parp3l.action -> in_channel ->
                 out_channel list -> unit
```

The process template definition in the OcamlP3l code looks like the following:

```
let farmetempl (OutChanSel f) ic ocl =
  while true do
    try
      let theoc = f ocl in
      match (Marshal.from_channel ic) with
        UserPacket(p,seqn,tl) ->
          Marshal.to_channel
            theoc
            (UserPacket (p,seqn,Farmtag::tl))
            [Marshal.Closures];
          flush theoc;
      | EndStream  ->
          List.iter
           (fun x -> Marshal.to_channel
                       x
                       EndStream
                       [Marshal.Closures];
                     flush x)
           ocl;
          List.iter close_out ocl; close_in ic; exit 0
    with End_of_file -> List.iter close_out ocl;
                        close_in ic
  done;;
```

Therefore the whole compilation process transforming an OcamlP3l skeleton program into the parallel process network implementing the program can be summarized in the following steps:

1. the skeleton code is parsed and transformed into a skeleton tree data structure, recording all the significant details of the skeleton nesting supplied by the user code,

2. the skeleton tree is traversed and processes are assigned to each skeleton according to the implementation templates. During this phase, processes are denoted by their input/output channels, identified via a unique number.

3. once the number and the kind of parallel processes building out the skeleton code implementation is known, code is generated that either delivers the proper closures, derived by using the process templates, to the "template interpreter" instances running on distinct workstations (this happens just on one node, the "root" one), or waits for a closure and repeatedly computes this closure on the proper input and output channels until an `EndOfFile` mark is received.

4 Multivariant semantics and logical debugging

By providing modules that implement the three OcamlP3l skeleton semantics (the sequential one, the parallel one and the graphical one), we allow the OcamlP3l user to perform the following parallel application development process:

- develop skeleton code modeling the application at hand. This just requires a full understanding of the skeleton sequential semantics and usually allows the user to reuse consistent portions of existing applications written in plain `ocaml`.

- test the functionality of the new application by supplying relevant input data items and looking at the results computed using the sequential skeleton semantics. In case of problems, the user may run the sequential debugging tools to overcome the problem.

- link the parallel skeleton semantics module and run the application onto the workstation network. Provided that the application was sequentially correct, no new errors will be found at this step (we assume that the run time is guaranteed correct!).

- look at the performance results of running the application on the number of processing nodes available and possibly adjust the significant performance parameters, such as the number of workers of the `farm`, `map` and `reduce`. This is

actually the real problem in the development of an *efficient* parallel application. The next version of OcamlP3l will include analytical performance models for the templates and these models will be used to automatically instantiate the performance parameters in the compiler. During this phase, the programmer may link the graphic semantic skeleton module and look at the results of the program execution, i.e. at the resulting process graph, in order to understand where bottlenecks are or which parts of the program must be further decomposed using skeletons in order to get better performant application code.

Let us spend now some words concerning logical, sequential debugging of OcamlP3l appplications.

A user developing an OcamlP3l application may link the sequential skeleton semantics module to his/her code and debug the application by using the plain sequential debugging tools of `ocaml`. This debugging activity can be performed on a single machine, provided the machine supplies `ocaml`.

Once the application has been debugged, i.e. the user perceives it computes the expected results, he/she can compile the application in such a way that the parallel code is obtained, by linking the parallel skeleton semantics. As we guarantee that the implementation templates for the different skeletons of OcamlP3l are correct (deadlock free, load-balanced, etc.) and as we guarantee that the process transforming the skeleton code in the process code is correct, the user does not need to perform any explicit activities in order to check that the results computed by the parallel code are correct.

In particular, the user does not need to check that all the processes have been correctly scheduled for execution, or that the communication channels have been set up properly between these processes, or that data of type `'a` has been never delivered on channels transmitting data of type `'b`. This is a very short list of bad things that may affect the correct behavior of an explicitly parallel program, indeed. The fact that the user is not required at all to take them into account is one of the biggest pro's of the functional skeleton approach.

5 Related work

Many researchers are currently working on skeletons and most of them are building some kind of parallel implementation, but our work, as far as we know, is unique in its combination of a fully functional strongly typed language with the skeleton approach.

In particular, Darlington's group at Imperial College in London is actively working on skeletons. They have explored the problems relative to implementing a skeleton programming system, but the approach taken uses an imperative language as the implementation language. Currently there is a "local" prototype implementation but no public domain implementation of their skeleton approach and they seem deeply involved in the study of the data-parallel and coordination aspects of skeletons. [9, 8, 1]

A different approach relative to skeleton parallel programming within a functional framework has been discussed by Bratvold [4]. Bratvold takes into account plain ML programs and looks for skeletons within them, compiling these skeletons by using process networks that look like implementation templates. However, both the final target language and the implementation language are imperative.

Finally, Serot [17], presents an embedding of skeletons within `ocaml` that seems to be close to our work, although independently developed. The message passing is performed by interfacing the MPI library with `ocaml`. The skeletons taken into account are different. She considers data-parallel `farm`, roughly corresponding to our `mapvector` skeleton, and two further skeletons, `scm` and `filt`. `filt` is a plain filter skeleton, canceling data items from a list, while `scm` (Split, Compute and Merge) looks like a map skeleton working on lists with explicit, user defined, decomposition/recomposition functions.

Serot's implementation of the skeletons within `ocaml` is quite different from ours and only allows one skeleton at a time to be realised on the processor networks, thus preventing skeleton composition (you cannot nest two `scm` skeletons for example), and only allowing for a limited form of staging of the parallel computation: you can perform an `scm`, then when this is finished, you can reorganize your network and perform another `scm`. This way, the mapping of virtual processors to real processors on the network is a trivial task, and is done inside each skeleton at run-time instead of beforehand in a specific pass like in OcamlP3l. Serot implements the skeletons included in the language by providing second order functions that directly call MPI and realize an SPMD execution model.

As for the relevant effort done in the field of languages for mobile agents, like for example [12, 11], it should be noted that they address quite a different kind of problems, but once stable, these languages could form the basis of a next generation fully fault-tolerant and dynamically load-balanced version of our system.

6 Conclusions and perspectives

Here we showed how a skeleton parallel programming model such as the one provided by p3l can be successfully married with the functional programming environments such as the one provided by `ocaml`.

In particular, we discussed how skeletons can be embedded within `ocaml` as second order functions and how modules implementing both the sequential and the parallel skeleton semantics can be supplied that allow users to write, functionally debug and run in parallel skeleton applications using `ocaml` to express sequential computations *and* data types. The whole process preserves the strong typing properties of `ocaml`.

At the moment, the prototype OcamlP3l implementation is being tested as described in this paper and is available from the OcamlP3l project home Web page `http://qui.di.unipi.it/ocamlp3l.html`, together with a user manual detailing the tools which are available with the distribution to compile, run and trace OcamlP3l code.

In the near future we want first of all to include some more skeletons (or variant of the existing ones) into the prototype. Then we want to have a more efficient communication layer, by using MPI [14] instead of the Unix socket library. At the same time, we investigate the feasibility of porting the system on the ubiquitous Windows boxes, for didactical purposes. Finally, we plan to write some significant parallel applications in order to fully test the prototype, and then release the prototype to the public domain.

References

[1] P. Au, J. Darlington, M. Ghanem, Y. Guo, H.W. To, and J. Yang. Co-ordinating heterogeneous parallel computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Europar '96*, pages 601–614. Springer-Verlag, 1996.

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

[3] 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.

[4] T. Bratvold. *Skeleton-Based Parallelisation of Functional Programs*. PhD thesis, Heriot-Watt University, 1994.

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

[6] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and support of massively parallel programs. *Future Generation Computer Systems*, 8(1–3):205–220, July 1992.

[7] 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.

[8] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, and M. Kohler. Fortran-S: A Uniform Functional Interface to Parallel Imperative Languages. In *Third Parallel Computing Workshop (PCW'94)*. Fujitsu Laboratories Ltd., November 1994.

[9] 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.

[10] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.

[11] Cedric Fournet and Luc Maranget. *The Join-Calculus language*. INRIA, June 1997. Software and documentation available electronically, (`http://pauillac.inria.fr/join`).

[12] F. C. Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1995. CMU-CS-95-223; also published as Technical Report ECRC-95-36.

[13] Xavier Leroy, Jérôme Vouillon, and Damien Doligez. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[14] M.P.I.Forum. Document for a standard message-passing interface. Technical Report CS-93-214, University of Tennessee, November 1993.

[15] Lawrence C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[16] S. Pelagatti. A methodology for the development and the support of massively parallel programs. Technical Report TD-11/93, Dept. of Computer Science – Pisa, 1993. PhD Thesis.

[17] Jocelyn Serot. Embodying parallel functional skeletons: an experimental implementation on top of MPI. In *Proceedings of the EuroPar 97*. Springer Verlag, LNCS No. 1300, 1997. Passau, Germany.

[18] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.