Programming in JoCaml — Extended Version

Louis Mandel — Luc Maranget

N° 6261
Août 2007

Thème COM

Rapport de recherche
Programming in JoCaml — Extended Version

Louis Mandel, Luc Maranget

Thème COM — Systèmes communicants
Projets Moscova

Rapport de recherche n° 6261 — Août 2007 — 24 pages

Abstract: JoCaml is a language for concurrent and distributed programming. We here present a non-trivial distributed application written in JoCaml: a ray tracer. Thereby, we show how to program the coordination of multiple cooperating agents in a concise manner, in the spirit of functional programming. We also address the issue of agent failure.

Key-words: concurrency, distributed programming, programming language, functional programming, join-calculus, ML
Programmeur en JoCaml

Résumé : JoCaml est un langage pour la programmation concurrente et distribuée. Ici, nous décrivons une application consécutive écrite en JoCaml : un lanceur de rayons. À l'occasion de cet exemple, nous montrons comment programmer de façon concise la coordination de multiples agents coopérants, dans l'esprit de la programmation fonctionnelle. L'échec possible d'un agent est pris en compte.

Mots-clés : concurrence, programmation distribuée, language de programmation, programmation fonctionnelle, join-calcul, ML
1 Introduction

Concurrency has always been a fact of life in the areas of system and network programming. Networks now being standard, the interest for concurrency has long shifted from specialists to ordinary programmers. Moreover, the ever growing availability of multi-core machines augments the population of those that wish to have several computing “agents” to cooperate. However, as everyone who has tried knows, writing a distributed application, one that runs on several machines, is not an easy task.

The join-calculus [?] is a process calculus in the tradition of the \( \pi \)-calculus [?]. The main purpose of such calculi is to describe concurrent and distributed systems; programming such systems is a different, although related, issue. The issues are related because a good model offers suitable abstractions that help programmers. To shorten the distance between model and program, the join-calculus has been designed with implementation in mind. We will not discuss this point on theoretical grounds, but rather on practical grounds. Indeed, we are developing a programming language based upon the join-calculus.

Our language, JoCaml, is an extension of Objective Caml (OCaml) [?], a popular dialect of ML. By choosing to extend an existing language, and not to design one of our own, we first intend to minimize our work. We also intend to benefit of functional programming, of pre-existing code base, and of a population of programmers open to innovation.

In this work, we more illustrate the JoCaml language than the JoCaml system (however, see Section 2), by showing how a middle-sized OCaml program can be made into a JoCaml program (Sections 3–7) suited for execution on a network of computers. We then conclude after a few performance figures.

The complete source for the program is available at http://jocaml.inria.fr/pub/joex/JoCaml3RUs.tar.gz.

2 The new JoCaml

The new JoCaml system [?] is a re-implementation from scratch of the previous prototype [?]. The old JoCaml focussed on language expressiveness, by providing all the features of the underlying process calculus. In particular, the old system provides extensive support for the hierarchical location tree of the join-calculus, which acts as a model for the migration of processes from one computer to another. Implementing this feature has two consequences, which both hinder compatibility with OCaml:

1. Code migration makes native code execution problematic.

2. Code migration implies extending the byte-code machine and the linking mechanism of OCaml, leading to serious incompatibilities at the binary level.

Besides, the extensive alterations performed over the version 1.07 of OCaml that was taken as a starting point for developing JoCaml, made it very difficult to follow the evolution of OCaml, of which current version is 3.10.

\[ ^1 \] The OCaml system does not provide just-in-time compilation.
By contrast with the old system, the new JoCaml focuses on compatibility with OCaml. Briefly, we proceed by altering the OCaml compiler from parsing phase to first intermediate code generation, and by enriching the thread library of OCaml with specific support. Compiler alteration is justified by specific typing and pattern matching compilation [?, ?], which both need to be perform inside the compiler. Compiler alteration is limited in the sense that we change or add a few thousand lines in the compiler original source files, add a few source files, and retain the OCaml formats for binary files. Moreover, the JoCaml compiler produces compiled signatures files that are the same as the ones of OCaml, which is of crucial importance to the simple versioning policy of OCaml.

Our focus over compatibility and limited alteration of OCaml, made us abandon the mobility features of the join-calculus. Nevertheless, there are useful distributed programs that can be written without code mobility. Full compatibility with OCaml not only means that we can write such programs starting from existing, sequential, source code in OCaml, but it also means that the existing source code may call external libraries that need not be re-compiled.

3 An introduction to join-definitions

We first provide a “survival kit” introduction to the basic concept of the join-calculus: the join-definition. The JoCaml tutorial [?] presents join-definitions from a programming perspective in greater detail.

An OCaml program is a set of definitions. Definitions introduce types, values or functions. JoCaml extends OCaml with join-definitions. A join-definition is a list of reactions rules that are processes guarded by join-patterns. A join-pattern is a list of channel names with formal arguments. A guarded process is fired when there are messages present on all the channels of its join-pattern. A typical process is the parallel composition (operator “&”) of elementary processes.

In JoCaml there are two kind of channels: asynchronous and synchronous. Asynchronous channels are the classical channels of the join-calculus. Message sending on an asynchronous channel is an example of an elementary process. Synchronous channels can engage in any kind of join-pattern but the guarded process must send back a result to the emitter. In that aspect, sending a message on a synchronous channel can be seen as a function call.

Let us now consider an example of a join-definition: a concurrent buffer based on the two-lists implementation of functional FIFO queues.

```ocaml
def state(xs,ys) & put(x) = state(x::xs,ys) & reply () to put
  or state(xs,y::ys) & get() = state(xs,ys) & reply y to get
  or state(_,_ as xs,[]) & get() =
    state([],List.rev xs) & reply get() to get
  val put : '_a -> unit
  val state : ('_a list * '_a list) Join.chan
  val get : unit -> '_a
Join-definitions are introduced by def, reaction rules are separated by or, and the channel names in join-patterns are separated by “&”. The join-definition above defines one asynchronous channel (state) and two synchronous channels (put and get). Asynchronous channels have type \( \tau \) Join.chan where \( \tau \) is the type of the message.
```

INRIA
The idea of this buffer is to store the FIFO queue (implemented by a pair of lists) as a message on the channel state. By the organization of join-patterns, which all include state, exclusive access to the internal state of the buffer is granted to the callers of synchronous put and get.

The first join-pattern state(xs,ys) & put(x) is satisfied whenever there are messages on both state and put. The behavior of the guarded process is to perform two actions in parallel: (1) send a new message on state where the value x is added to the list xs and (2) return the value () to the caller of put.

The second join-pattern state(xs,y::ys) & get() is satisfied when there are messages on both state and put and that the message on state matches the pattern (xs,y::ys). That is, the message is a pair whose second component is a non-empty list. The process guarded by this join-pattern removes one value from the buffer and returns it to the caller of get. The last join-pattern state(_::_ as xs,[]) & get() is satisfied when there is a message on get and a message on state that matches a pair whose first component is a non-empty list and second component is an empty list. The corresponding guarded process transfers elements from one end of the queue to the other and performs get again. Notice that there is no join-pattern that satisfies state([],[]) & get(). As a consequence, a call to get is blocked when the buffer is empty.

To initialize the buffer, a message is sent on state. An expression that performs such a sending is spawn state([],[]). The spawn construct executes a process asynchronously. Syntactically, spawn lifts a process into an expression. It is to be noticed that processes and expressions are distinct syntactical categories.

To be able to create several buffers, the previous definition is encapsulated into a function create_buffer.

type 'a buffer = { put : 'a -> unit; get: unit -> 'a }

let create_buffer () =
  ... (* same definition of put/get/state as before *)
  spawn state([],[]) ;
  {put=put; get=get;}
val create_buffer : unit -> 'a buffer

A buffer is in fact a record, whose fields are the put and get synchronous channels. That way, we make buffers first-class values and additionally hide the internal channel state.

4 Organizing concurrent computations

We first present iterators (type enum) over the elements (type elt) of a collection (type t) [].

val start : t -> enum
val step : enum -> (elt * enum) option

The function start builds an enumerator from a structure of type t, while step returns the first value built from an enumerator and the enumerator that builds the next values. We illustrate the idea of iterators by an enumerator over an integer interval.
type elt = int and t = int * int and enum = int * int
let start c = c
let step (n,m) = if n > m then None else Some (n,(n+1,m))

4.1 Concurrent iteration

We then define a channel `par_iter` that applies a function `worker` to all the values produced by an enumerator, function calls being performed concurrently.

```ocaml
def par_iter (worker, enum) = match step enum with
  | None -> 0
  | Some (x,next) -> par_iter(worker, next) & begin
      worker x ; 0
  end
```

If `step enum` does not return a value, then there is nothing to do (0 is the empty process). Otherwise, a recursive sending to `par_iter` and a call to `worker` are performed concurrently. The construct `worker x ; 0` lifts the expression `worker x` (of type `unit`) into a process.

An example of `par_iter` usage is to print the integers from 1 to 3 in unspecified order:

```ocaml
let print x = print_char '(' ; print_int x ; print_char ')
let _ = spawn par_iter(print, start (1,3))
```

A possible output is: `(2)(1)(3)`.

To share parallel computations between several “agents”, we introduce the notion of a pool of functions.

```ocaml
type 'a pool = {
  register: ('a -> unit) Join.chan;
  compute: 'a -> unit;
}
```

Let `create_pool () =`:

```ocaml
def compute(x) & agent(worker) =
  worker x ;
  agent(worker) & reply () to compute in
  { register=agent; compute=compute; }
```

A pool exports two channels: `register` and `compute`. Viewed from caller side, `register` is for offering computational power, whereas `compute` is for exploiting computational power. The pool implementation matches computing agents and exploiting agents with a straightforward join-pattern.

Internally, available computing agents are messages pending on channel `agent`. Notice that several instances of a given `worker` agent cannot execute concurrently. Namely, a computation can start only when an agent is available, and an agent engaged in a computation returns to available status only when the computation (`worker x`) is over.

To illustrate the use of the pool, we introduce a second function to print integers:

```ocaml
let print_bis x = print_char '<' ; print_int x ; print_char '>
```

Then, we create a pool and register the `print` and `print_bis` functions.

---

2Due to abundant concurrency, another possible output is `((2)1)(3)`.
let pool = create_pool ()

let () = spawn (pool.register(print) & pool.register(print_bis))

Finally, by combining par_iter and the compute component of the pool, we have the integer interval printed by two agents:

let () = spawn par_iter(pool.compute, start (1,3))

A possible output is (2)<1><3>. Due to concurrency of distinct agents, another possible output is (<1>2)<3>.

4.2 Collecting the results of concurrent iteration

In the previous example, par_iter is an asynchronous channel. This clearly expresses that it cannot be known when par_iter has finished its work. We now wish not to perform side effects, as print does, but instead to collect returned values.

More precisely, we aim at building a new kind of pool, with a “fold” function that behaves as par_iter as regards concurrency, but additionally returns a combination of results.

type ('a,'b) pool =
  { register: (elt -> 'a) Join.chan;
    fold: t -> ('a -> 'b -> 'b) -> 'b -> 'b; };

Interface for exploiting the pool is the fold function that, with respect to the previous par_iter, takes the combination function and an initial value for the result as extra arguments, and returns a combined result of type 'b. Interface for offering computational power is register as before, but now registered functions return a result of type 'a.

The new pool is controlled by the following monitor, of which primary job is to collect results.

type ('a,'b) monitor =
  { enter: unit -> unit; leave: 'a Join.chan;
    wait: unit -> 'b; finished: unit Join.chan; };

let create_monitor combine init =
  def state(n,r) & enter() = state(n+1,r) & reply () to enter
  or state(n,r) & leave(v) = state(n-1,combine v r)
  or state(0,r) & wait() & finished() = reply r to wait
  in spawn state(0,init) ;

val create_monitor : ('a -> 'b -> 'b) -> 'b -> ('a, 'b) monitor

A monitor provides four channels: enter, leave, finished and wait. Channel enter (resp. leave) is used by the monitored pool (see below) to signal that a new task starts (resp. ends). The monitored pool will send a message on finished when iteration has come to an end. Finally, wait is a function that returns the combination of all the results of the monitored tasks.

A monitor has an internal state of which first component n counts the number of tasks being computed. This counter is updated with the channels enter and leave. Additionally, the message on leave is a task result to be combined with

RR n° 6261
the second component \( r \). The last join-pattern \((\text{state}(0,r) \& \text{finished}()) \& \text{wait()}\) states that when there are no more tasks, either active \((\text{state}(0,_))\), or to be allocated \((\text{finished}())\), then the call to \text{wait()} can be answered.

We now present the pool implementation:

```plaintext
let create_pool () =
  def loop(monitor, enum) & agent(worker) =
    match step enum with
    | Some(x, next) ->
      monitor.enter() ;
      loop(monitor, next) & call_worker(monitor, x, worker)
    | None -> monitor.finished() & agent(worker)
    and call_worker(monitor,x,worker) =
      let v = worker(x) in monitor.leave(v) & agent(worker)
  in
  let fold x combine init =
    let monitor = create_monitor combine init
    in
    spawn loop(monitor, start x) ;
    monitor.wait ()
  in
  { fold=fold ; register=agent ; }
```

Let us examine the definition of \text{fold}: it first creates a monitor, then starts the iteration, and finally calls the \text{wait} function of the monitor.

The \text{loop}/\text{agent} definition is essentially a combination of the previous pool implementation and of \text{par_iter}, with worker calls being put aside for clarity (channel \text{call_worker}). The combination has the effect that various instances of a given \text{worker} agent now execute in sequence, following iteration order. Additionally, calls to the monitor are inserted at appropriate places. A remarkable point is that we can be sure that all calls to \text{monitor.enter} have been performed before the message on \text{monitor.finished} is sent. This is almost obvious by considering that the recursive sending on \text{loop} is performed only once the call \text{monitor.enter()} has returned, by the virtue of the sequencing operator ";\". Moreover, the internal counter of the monitor indeed counts active tasks: as "\&" binds more tightly than ";\", the process \text{call_worker(monitor,x,worker)} executes once \text{monitor.enter()} has returned, and thus once the monitor counter has been incremented. Similarly, the counter is decremented (by \text{monitor.leave(v)}) only once the worker has returned.

The new pool is quite powerful, since it can serve as a meeting place between several agents that offer computational power and several agents that exploit it. Let us define the former agents and register them.

```plaintext
let double x = print x; 2*x
let double_bis x = print_bis x; x+x
let pool = create_pool()
let () = spawn (pool.register(double) & pool.register(double_bis))
```

Exploiting agents are two functions, with different combination behavior.

```plaintext
let sum x = pool.fold x (+) 0
let prod x = pool.fold x ( * ) 1
```

Finally all agents meet through the pool.

```plaintext
def echo(c,x) = print_char c ; print_int x ; print_char c ; 0
let () = spawn echo(‘+’,sum(1,3)) & echo(‘*’,prod(4,5))
```

A possible output is \texttt{<1>(2)<4>(5)*80*<3>+12+}.
4.3 Distributed computations

In JoCaml, concurrent and distributed computations are based on the same model: different programs (abstracted as sites) may communicate by the means of channels. More precisely, site A may send messages on a channel of which definition resides on another site B. In that situation, guarded processes execute on site B. One practical problem in distributed applications is for the communicating partners, first to know one another, and then to have at least a few channels in common. To solve the issue, JoCaml provides library calls to connect sites and a name service, which basically is a repository for values (more specifically channel names) indexed by plain strings.

As an example, this first program runs on machine A.

```ocaml
let pool = create_pool();
let () = Join.Ns.register Join.Ns.here "reg"
    (pool.register: (int -> int) Join.chan)
let () = Join.Site.listen
    (ADDR_INET (Join.Site.get_local_addr(), 12345))
let () = print_int (pool.fold (1,3) (+) 0)
```

This program creates a pool. Then, by calling `Join.Ns.register`, it stores the channel `pool.register` associated to the name "reg" in the local name service (`Join.Ns.here`). Then, `Join.Site.listen` starts to listen for connections on the default Internet address of the local site (`Join.Site.get_local_addr()`) on port 12345. Finally, the program calls `pool.fold`. This call blocks, since no computing agent has entered the pool yet.

To become such a computing agent, machine B runs the following program.

```ocaml
(* master_addr is the Internet address of A *)
let a_site = Join.Site.there (ADDR_INET(master_addr,12345))
let ns = Join.Ns.of_site a_site
let register = (Join.Ns.lookup ns "reg": (int -> int) Join.chan)
def double(x) =
    print_char '(' ; print_int x ; print_char ')
    reply x+x to double
let () = spawn register(double)
```

Here, B first gets the site identity of the program running on A, with the function `Join.Site.there`, and its name service with `Join.Ns.of_site`. Then, it retrieves the (synchronous) channel associated to the key "reg". The name service is *not* type safe. For instance, the type of `Join.Ns.lookup` is `Join.Ns.t -> string -> 'a`. As a minimal precaution, we insert explicit type constraints. Finally, B defines and registers the synchronous channel `double`. The effect of A calling the registered `double` is the one of a remote function call. Hence, console output is 12 on A and (1)(2)(3) on B.

Another issue deserves mention. The program of B above is not complete: as `spawn register(double)` returns immediately, execution goes on. For the program not to terminate by reaching its end, we deadlock it purposely.

```ocaml
let () = def dead() & lock() = reply () to dead in dead()
```

\(^3\)A weakness of JoCaml, we agree.
But B is now blocked for ever, whereas a desirable behavior is for B to be released when A does not need B anymore, or at least when the program running on A terminates.

The function Join.Site.at_fail provides a convenient solution. It takes a site A and a channel (of type unit Join.chan) as arguments, and returns (). When it is detected that A has failed, then a message is sent on the channel. Thus, we replace the code above by:

```ocaml
let () =
  def wait() & release() = reply () to wait in
  Join.Site.at_fail a_site release ;
  wait()
```

## 5 Ray tracing and its parallelization

Ray tracing [?, ?] is a now classical technique for computing 2d-images from 3d-scenes. The general principle of ray tracing is to cast light rays from the viewpoint of an observer into the scene. To render a scene as a \(w \times h\) bitmap image, one casts the \(w \times h\) rays defined by the observer viewpoint and the position of the \(w \times h\) picture elements (pixels) of the image plane, which stands in front of the observer. One then computes the intersection of this primary ray with the scene. Reflections on object and the computation of illumination imply casting more rays. The whole process involves much computation.

This simple introduction should be sufficient to grasp the basic structure of a ray tracer. Our sequential ray tracer has not been written by us, but by the team “Camls R’Us” as an entry [?] for the ICFP programming contest of year 2000. Although written in three days, the program is of significant size (1729 lines, disregarding comments and empty lines).

Scenes are represented internally as the following (OCaml) record type, a field of module Scene.

```ocaml
type t =
  { file : string ; wid : int ; ht : int ; obj : Obj.t ; ... }
```

In the type above, file is the name of the file where to save the image, wid and ht are the image width and height respectively. The obj field holds the internal representation of the scene. We omit some of the fields in the scene record, such as the definition of lights, observer field of view, etc. Bitmap images are PPM images, a simple image format, where an image file basically is a sequence of colors encoded as three eight-bits values.

In the original ray tracer, rendering operations are performed by the module Render that exports only one function render, as shown by its interface file.

```ocaml
val render : Scene.t -> unit
```

Notice that Render.render returns () because it saves the bitmap it computes.

Let us assume a function Ray.render_pxl of type Scene.t -> int -> int -> color of type Scene.t -> int -> int -> color that performs the casting of a primary ray and all subsequent operations, finally returning the color of the pixel of the image plane of which coordinates

\(^{4}\text{color is the type of colors, in practice a color is an integer of which 24 bits are used.}\)
are given as arguments. Then, one easily writes `Render.render` by two nested for loops.

```ocaml
let render sc =
  let oc = open_out_bin sc.file in (* open output, ppm, file *)
  Printf.fprintf oc "P6%n%d%n%d\n255\n" sc.wid sc.ht; (* Ppm header *)
  for j = sc.ht - 1 downto 0 do
    for i = 0 to sc.wid-1 do
      let color = Ray.render_pxl sc i j in
      output_color oc color
    done
  done;
  close_out oc
```

The problem statement of the contest defines a language, GML, for both describing the scene and the rendering conditions. This specification naturally impacts the design of the Camls'R Us ray tracer.

1. The GML program is parsed.

2. The resulting abstract syntax tree is evaluated by an interpreter (module `Eval`), with the following peculiarities:

   (a) To execute the instruction `render` that commands the production of an image, the interpreter calls `Render.render`, all scene components being retrieved from the interpreter stack. It is to be noticed that there can be several `render` instructions in a given program.

   (b) Other instructions are Constructive Solid Geometry operations, 3D-transforms, arithmetics, `if` instructions, function calls, etc.

3. The ray tracer ends once the interpreter has returned.

Our strategy for distributed execution is straightforward: all programs involved will execute the same GML program, however they will react to the `render` instruction differently. A distinguished program, the master, controls the work of others. All other programs are slaves and they perform the rendering operations. For the master to distribute the work to the slaves, we need to define an unit of work smaller than the image. We call this unit a subimage. A simple choice for the subimage is a line (or several lines), since images are easily encoded as arrays of lines. More precisely master and slaves behave as follow...
Master

1. Parse the GML file and make the abstract syntax tree available to slaves.

2. Interpret the GML program. Instruction render starts the concurrent process of allocating subimages to slaves and of collecting their results. When all subimages are present, the image is saved.

3. End when all the images have been saved.

Slave

1. Connect to the master, register as a potential worker, and retrieve the GML abstract syntax tree.

2. Interpret the GML program. Instruction render starts accepting subimages descriptions from the master, computing subimages, and sending them back to the master.

3. End when the master is done.

Observe that master and slaves agree on the scenes because they execute the same GML program, GML being a deterministic language.

Our design has the merit of simplicity: alterations to the original, sequential, ray tracer are minimized. In particular, the source of the sequential ray tracer includes the compilation unit Eval of which idealized code is as follows: a definition of the values of GML (type value), a recursive function that implements a stack-based interpreter (function eval), and a function that starts the evaluation (function eval_program) — see Figure 1 for the original implementation and signature file. The new implementation of Eval simply abstracts out module Render, that is, it defines a functor — see Figure 2. Master and slave apply the functor Eval.Make to different modules, RenderMaster and RenderSlave respectively, yielding two different interpreters.

Another modification is worth signaling: as subimages are made of lines, we separate the nested loops of the sequential render. The inner loop goes into the Ray module, yielding a new Ray.render_line function of type Scene.t -> int -> string that takes a scene and a line number as arguments, and that returns a compact representation of the colors of the pixels in the line.

6 The Render modules

6.1 Master side

The iterator on scenes in defined in the Scene module. For simplicity we present iteration by the line.

type elt = string * int and enum = string * int
let start sc = (sc.file, sc.ht-1)
let step (file,n) = if n < 0 then None else Some ((file,n),(file,(n-1)))

The master uses the fold pool of section 4.2 of which register function is stored in the local name service.

let img_pool = create_pool ()
let () = Join.Ns.register Join.Ns.here "reg"
  (img_pool.register : (Scene.elt -> int * string) Join.chan)
Figure 1: Original source files for Eval

Implementation, file eval.ml

type value =
| ... |
| I of integer |
| S of string |
| O of Obj.t (* 3d object *) |
| ... |

let rec eval env stack code = match code, stack with
| ... 
(* Example of an instruction: integer addition *)
| Op_addi::code, I i2::I i1::stack -> eval env (I (i1+i2)::stack) code
(* Render instruction *)
| Op_render::code, S file::I ht::I wid::O obj::...::stack ->
  let sc = { file=file ; wid=wid ; ht=ht ; obj=obj ; ... } in
  Render.render sc ;
  eval env stack code
| ...

(* Entry point *)
let eval_program code = eval [] [] code

Interface, file eval.mli

val eval_program : Gml.tok list -> unit

Figure 2: New source files for Eval

Implementation, file eval.ml

module Make (Render : sig val render : Scene.t -> unit end) = struct
  ... Original code ...
end

Interface, file eval.mli

module Make :
  functor (Render : sig val render : Scene.t -> unit end) ->
  sig val eval_program : Gml.tok list -> unit end

RR n° 6261
As specified by the type constraint above, remote workers return pairs of a line number and of a line of pixels.

Finally, the following function commands the rendering of scene \(sc\) and saves the resulting image.

```ml
let render_image sc =
  let img = Array.create sc.ht "" in
  let combine (n,line) = img.(n) <- line in
  img_pool.fold sc combine () ;
  save_image sc.file img
```

Notice that the combination function performs an update of the bitmap \(img\).

It should be noticed that \(render_image\) above returns only when the image is saved. We can then define \(RenderMaster.render\) as being \(render_image\).

```ml
let render = render_image
```

As a consequence, all images are saved when the interpreter terminates and the master can terminate.

In our implementation, we in fact save the image asynchronously, for disk operations not to delay the interpreter. Termination of the master is then controlled by a simple monitor (page \(\square\) that counts images.

```ml
let monitor = create_monitor (fun () r -> 1+r) 0
```

```ml
let render_image sc =
  monitor.enter () ;
  let img = Array.create sc.ht "" in
  let combine (n,line) = img.(n) <- line in
  img_pool.fold sc combine () ;
  spawn begin
    save_image sc.file img ;
    monitor.leave ()
  end
```

The monitor is made public by the module \(RenderMaster\), for the code after the call to the interpreter to wait on it.

```ml
(* Call interpreter, gml is the abstract syntax tree *)
let module E = Eval.Make(RenderMaster) in
E.eval_program gml ;
(* Interpreter is over *)
let m = RenderMaster.monitor in
spawn m.finished() ;
let nimages = m.wait() in
Printf.eprintf "My slaves have computed %d images\n" nimages ;
exit 0
```

### 6.2 Slave side

A slave executes two tasks concurrently. It (1) interprets the GML program so as to build the scenes and (2) computes subimages on master demand. For the
two agents to communicate, we introduce a suitable mapping from filenames to scenes.

type ('a,'b) hashtbl = { find : 'a -> 'b; add : ('a * 'b) -> unit; }

let create_hashtbl () =
  let t = Hashtbl.create 17 in
  def state(blocked) & add(k,v) =
    Hashtbl.add t k v ;
    List.iter (fun release -> spawn release()) blocked ;
  state([]) & reply () to add
  or state(blocked) & find(k) =
    let v = try Some (Hashtbl.find t k) with Not_found -> None in
    match v with
    | Some v -> state(blocked) & reply v to find
    | None ->
      def release() & wait() = reply () to wait in
        state(release::blocked) & reply wait() ; find(k) to find in
        spawn state([]) ;
      { find=find; add=add; }

let scenes = create_hashtbl ()

The code above is a wrapper of the (OCaml) hashtable t. There are two points to notice. First, by very existence of state the internal hashtable t is protected against concurrent modification, as usual. Second, the synchronous channel find blocks when the table t does not associate any value to the key k, until a value for k is added into the table. This block/release process implies a form of communication between find and add operations. Such a communication is by the means of the message that is pending over the internal channel state.

More precisely, the code of find first changes the interface to Hashtbl.find “return value v or raise exception Not_found” into the new interface “return Some v or return None”. In the first, successful, case the caller of find gets v as a reply. In the second, failed, case the reply to find is delayed, by inserting a call wait() before calling find again. The call to wait will return when a message is sent on the asynchronous channel release, which the code for add does. A find operation is then attempted again. The coding is slightly inefficient, but it suffices here. Namely, the pool of the master does not allow several concurrent calls to find in a given slave. Thus the list blocked cannot hold more than one element.

Actual subimage computations are performed by the following definition.

def compute_subimage (tag,n) =
  let pxls = Ray.render_line (scenes.find tag) n in
  reply (n,pxls) to compute_subimage

Additionally, compute_subimage is registered into the pool of the master, we omit the code which is the same as the one of section 4.3. Finally, RenderSlave.render simply stores the scene at the intention of compute_subimage.

let render sc = scenes.add (sc.file,sc)

As regards termination, the simplest solution is for slaves to terminate when the master does — see the end of section 4.3.
7 Failures

We claimed that local and remote message sendings were the same. Obviously we over-simplified the issue: the remote site may crash or become unreachable.

7.1 Detected failures

In the case of our ray tracer, the remote message sending is a synchronous one, \textit{i.e.} a remote function call, which is performed by \texttt{call\_worker} (section 4.2).

\[
\ldots \quad \text{and } \texttt{call\_worker(monitor,x,worker)} = \\
\quad \text{let } v = \texttt{worker(x)} \text{ in } \texttt{monitor\_leave(v)} \& \texttt{agent(worker)}
\]

The definition of \texttt{worker} resides on a remote site. If the remote site fails and that the failure is detected by the JoCaml runtime system, then the call to \texttt{worker} will result in raising the exception \texttt{Join\_Exit} and \texttt{monitor\_leave(v)} will never execute. As a very untimely consequence, the image will never be completed.

To correct this misbehavior, it suffices to re-issue a failed task, as performed by the following, new definition of \texttt{call\_worker}.

\[
\ldots \quad \text{or } \texttt{agent(worker)} \& \texttt{compute(monitor,x)} = \\
\quad \texttt{call\_worker(monitor,x,worker)}
\]

\[
\text{and } \texttt{call\_worker(monitor,x,worker)} = \\
\quad \text{let } v = \texttt{try Some (worker(x)) with _ -> None in} \\
\quad \texttt{match } v \texttt{ with} \\
\quad \quad | \texttt{None} \rightarrow \texttt{compute(monitor,x)} \\
\quad \quad | \texttt{Some } v \rightarrow \texttt{monitor\_leave(v)} \& \texttt{agent(worker)}
\]

The re-issued task is made available to other agents by the means of a new channel \texttt{compute}, and of a new, straightforward, join-pattern. Additionally the worker that failed is forgotten about, since there is no \texttt{agent(worker)} process when \texttt{v} is \texttt{None}.

Observe that all exceptions are caught, not only \texttt{Join\_Exit}. Here, the master/slave protocol does not rely on exceptions and we can thus consider any exception to express a failure. This can occur in practice, for instance if the remote site consumes all available memory (exception \texttt{Out\_of\_memory}), since the JoCaml runtime system transmits exceptions.

7.2 Undetected failures

Unfortunately not all failures are detected. More concretely, we cannot assume that \texttt{worker(x)} will always either return a value or raise an exception. To solve the problem, we keep a record of all active tasks \textit{i.e.} of all tasks that are being computed. Then, near the end of image computation, we re-issue active tasks until the image is completed.

This technique requires a new kind of monitor, of which join-definition is as follows.
The code above is a refinement of the previous monitor (page 7). The message on state is now a triple, of an identifier (next_id, an integer), of a mapping from identifiers to task descriptions (active, an association list of which keys are identifiers), and of a partial result (r, as before). Identifiers permit the safe identification of task descriptions. They can be avoided when we are sure that tasks descriptions are pairwise distinct, which need not be the case with general enumerators.

The new monitor exports two additional synchronous channels: is_active, a predicate to test if a given task is active, and get_active that returns the list of active tasks. The guarded processes for these new channels are straightforward (List.mem_assoc is from the OCaml library and has obvious semantics). The exported channels enter, leave, finished and wait are still here, with a few changes. Channel enter now takes a task description x as argument and returns a fresh identifier next_id. The counter increment performed by the previous monitor is now replaced by adding (next_id,x) to the internal association list. Channel leave now takes an identifier id as an extra argument, which it uses to remove the completed task from the list of active tasks (by calling the library function List.remove_assoc). Notice that, as a given task can now be computed by several slaves, we take some care not to combine the result of a given task more than once. Finally the reaction rule for wait undergoes a small, but important, change: the message on state is re-emitted. Otherwise, subsequent calls to is_active would block.

The pool is also modified. The crucial modification regards re-issuing tasks when iteration has come to an end.

def loop(monitor,enum) & agent(worker) =
  match step enum with
    | Some(x, next) ->
      let id = monitor.enter(x) in
      loop(monitor,next) & call_worker(monitor, id, x, worker)
    | None -> do_again(monitor) & agent(worker)

When iteration is over (step enum returns None), a message on the internal channel do_again is sent. The worker that has not been called is also released.
The guarded process for do\_again is in charge of retrieving active tasks from the monitor.

\[
\text{or do\_again(monitor) \& agent(worker) =}
\begin{align*}
&\text{begin match monitor.get\_active() with} \\
&| [] \rightarrow \text{monitor.finished()} \\
&| xs \rightarrow \text{again(monitor, xs)} \\
&\end{align*}
\\& \text{agent(worker)}
\]

The synchronization on agent(...) above is not necessary. Nevertheless, it is clearly a good idea to wait for at least one slave to be available before re-issuing active tasks. The available slave is not used yet and the message on agent is re-emitted. If there are no active tasks left, (get\_active() returns the empty list), then the pool informs the monitor that it will not allocate any additional task (by monitor\_finished()). In fact, from all calls to wait being performed before do\_again is called for the first time, it can be deduced that the image is now complete. Hence the join-pattern for wait in the monitor could have avoided testing that active is empty.

If there are some active tasks left, then channel again is in charge of re-allocating them to available slaves.

\[
\text{or again(monitor,(id,x)::xs) \& agent(worker) =}
\begin{align*}
&\text{again(monitor, xs)} \& \\
&\text{if monitor.is\_active(id) then} \\
&\text{call\_worker(monitor, id, x, worker)} \\
&\text{else agent(worker)} \\
&\text{or again(monitor, []) = do\_again(monitor)}
\end{align*}
\]

The code above basically scans the list of active tasks. However, before calling call\_worker a last check is made. Indeed it can be that the task id has been completed while again was scanning the list. Observe that when the scanning is over (join-pattern again(..., [])), then do\_again is called again, resulting in another re-allocation of active tasks to slaves, if there still are active tasks.

It may seem that our solution is a waste of processing power. However, if we compute one image only, there is little waste. Having n slaves computing the same subimage is not less efficient than having one slave computing the subimage and n - 1 slaves being idle, up to communication costs. Furthermore, it can be more efficient on an heterogeneous network. If a slow slave is allocated a task at the end of the image, then other slaves will be allocated the same task quickly. As a result, image completion is delayed by the fastest amongst the slaves that are working on the last subimages.

If there are several images to compute, one can lower the amount of useless work by having the master to control the rendering of several images at a time. Namely, remember that the fold pool of section 4.2 can manage several exploiting agents. So as to control several images concurrently, we need change the function render of the module RenderMaster. The new definition of render simply stores the freshly computed scene in an instance of the buffer of section 3.

\[
\text{let buffer = create\_buffer ()}
\]

\[
\text{let render sc = buffer.put sc}
\]

\footnote{We omit the code, it is almost the same as in the previous section.}
An exploiting agents is a simple asynchronous channel definition that repeatedly calls the function `render_image` of page 14.

```
def render_images() =
    render_image (buffer.get()) ;
    render_images()
```

It remains to start several such agents, how many depending on some user setting $a_{\text{max}}$.

```
let () =
    for _k = 1 to $a_{\text{max}}$ do
      spawn render_images()
    done
```

An alternative is unconstrained concurrency: an exploiting agent is spawned as soon as an image is available.

```
def render_images() =
    let sc = buffer.get() in
    spawn begin render_image (sc) ; 0 end ;
    render_images()

let () = spawn render_images()
```

Notice that, with respect to the previous definition of `render_images`, the function `render_image` is called asynchronously. Now, we have three versions of `RenderMaster.render`, that respectively control the rendering of one image at a time, of at most $a_{\text{max}}$ images at a time, and of as many images as possible at a time. Preliminary experiments show that setting $a_{\text{max}}$ to be 2 or 3 is a reasonable choice. However, we list all these possibilities to demonstrate the flexibility of JoCaml. In particular, master termination is controlled by the same counting monitor (see page 14) in all cases.

### 8 Results

The performance of parallel programs is best illustrated by speedup measures. Speedup is the ratio of sequential execution time by parallel execution time. By sequential, we here mean the sequential ray tracer, not the concurrent ray tracer with one slave running. Of course, the measured times are wall-clock time. So as to minimize noise, any data we present is the median of three measures.

We perform most of our experiments on a cluster machine, the machine has 11 nodes, each node being a $2\times2$GHz AMD-64 bi-processor. Each node has 6 Gb of physical memory and the nodes are connected by a fast Gigabit dedicated network. Our main experiment consists in running two instances of the slave program on 1, 2, ... , 11 nodes, yielding 2, 4, ... , 22 participating slaves. Expressing speedup as a function of the number of slaves involved here makes sense, since cluster nodes are identical.

We also perform the additional experiment of running one slave per processor on 111 processors from the students computer lab at École polytechnique — we use 68 machines, some of which are bi-processors. The machines involved are of various CPU model and memory size, but all are rather modern. By contrast
with the cluster, speedup as a function of the number of slaves involved does not make sense on such an heterogeneous network. Instead, we introduce \emph{processing power}, a crude estimate computed by considering running times for a small image. We take one processor of the cluster as a base (1.0), cluster power is thus 22.0, while the local network power is about 80.0. Network conditions are standard for such a local network. All machines involved run Linux of one kind or another.

In all experiments, we run the master program on a distinct, “front-end”, machine, from which we launch the slave programs on the other machines by the means of \texttt{ssh}. More precisely, we start slaves in advance and measure the running time of the master.

We perform two sets of experiments, “urchin” (U) and “Sierpinski” (S) — see Appendix A. In both experiments, inputs to the ray tracer are scenes of identical structure and increasing complexity. By contrast, rendering parameters do not change: images are of size 1000 \times 1000, the subimage unit is the default setting of one line, and adaptive antialiasing is enabled, yielding between 2 and 27 primary rays per pixel.

Experiments U yield excellent results: here, we basically achieve linear speedup on the cluster (left of Figure 3). To provide an idea of the concrete benefits of concurrency, we also give some execution times (columns “Seq” and “x22” at the right of Figure 3). For instance, in the case of U11, image production time is reduced from more of 2 hours and a half to less than 7 minutes. It is still possible to go faster by using the local network (column “x111”), with a better result for the most difficult test U11.

Indeed, we here achieve a speedup of 71.8, which is surprisingly good, considering that the estimated processing power the network is 80.0 and that some of the machines involved are not idle at the time we use them. We shall thus assume that experiment U11 somehow validates our crude estimate of processing power. Namely, the (relatively) poor performance of U8 originates in the important setup times we observe while 111 slaves attempt their first connection to the master. The setup of \texttt{ssh} tunnels may partly explain this delay, but we cannot reject the hypothesis of a master overflow here.

We think that the good performance of experiment U stems from two main factors.
1. Rendering operations account for the vast majority of computations, almost the totality. They literally dwarf other operations which, by contrast, are executed sequentially by the master (parsing and image saving) or are executed once by every slave (scene building).

2. The subimage unit of one line is adequate.

(a) It is large enough. That is, due to scene complexity and antialiasing, most of the 1000 lines computed represent sufficient work for messaging and task management costs to remain unnoticed. In particular, by observing instantaneous CPU load, we had the confirmation that slaves are seldom idle because they are waiting for the master to allocate a subimage to them.

(b) It is small enough. Obviously, dividing the images into 1000 subimages to be computed by no more than 22 (or even 111) slaves allows a distribution of tasks that adapts well to varying task complexity and slave load. Besides, near the end, image completion is delayed by no more than a small fraction of the total work to be performed.

Figure 4: Decaying Sierpinski cube

<table>
<thead>
<tr>
<th>Running times, in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
</tr>
<tr>
<td>S4</td>
</tr>
<tr>
<td>S5</td>
</tr>
</tbody>
</table>

*Unstable results

Our second series (Figure 4) illustrates some of the limitations inherent to concurrent ray tracing. Series S includes simple scenes (S1 is made of a few cubes) and scenes that take time to compute (S5 is made of almost 580,000 cubes).

Experiments S1 and S2 are here for completeness, there is little point in attempting to compute such simple images faster than 7 s and 36 s. However, we achieve decent speedups, at least for S2. As regards the other experiments, significant sequential running times make relevant the idea of distributed execution. And indeed, we achieve good speedups on the cluster for experiments S3 and S4. However, the most demanding S5 experiment yields poor speedups. Those are easily explained once one knows that the sequential ray tracer starts to cast rays after about 100 s of computing time, which is mostly devoted to the interpretation of the GML program. About 100 s is a small fraction (3.8%) of the sequential running time of 2620 s, but a significant fraction of
204 s. More precisely, we can approximate the expected speedup for $N$ slaves as $N/(0.038 \times N + 0.962)$ (Amdahl’s law). For $N = 22$ we have a theoretical speedup of 12.2, which is rather close to the observed speedup of 12.8.

Experiments S on the local network are not very rewarding. In particular, computing image S5 on the local network of 111 processors is actually slower than on the cluster. Here also, poor performance originates from non-parallelized work dominating parallelized work, but on a more dramatic scale. Direct measures showed us that slaves engage in computing subimages no sooner than 200 s after they start. The expected delay inferred from processing powers should be about 140–160 s. Obviously, the computers at École polytechnique are not as fast as we expect them to be, perhaps due to significant memory traffic. Furthermore, about 20 processors are so slow that they do not take any part to the concurrent computation. This is mostly due to the presence of other users.

In spite of those unfavorable results, we claim to have demonstrated the efficiency of our concurrent ray tracer. Namely, for complex images of reasonable memory footprint and building time, we achieve dramatic speedups. Additionally, in a less favorable situation, we still make a decent profit out of the cluster.

9 Related works and conclusion

We are not computer graphics specialists and cannot claim that our work is a contribution to the study of parallel ray tracing. The survey [?] cites the humorous quote that “the more experience the writer of the parallel algorithm has in sequential algorithms, the less parallelism that algorithm is likely to exhibit”. Thus, as novices, we are likely to discover a lot of parallelism.

Rather, our contribution resides in the design and implementation of a concurrent language. Since the pioneering languages of the eighties (e.g. [?]) there have been many such languages. We restrict our attention to recent works that are close to ours as regards design and availability. We directly compare with \( C_\omega \) [?] that extends \( C^\# \) with join-definitions. In numerous aspects, the \( C_\omega \) design and development effort is similar to ours. Some differences exist though, most of which reflect the differences in the language extended. For instance, \( C^\# \) being an object language, channels appear as methods of the objects that define the join-patterns (called \texttt{chords}). This often leads to a natural and concise style, more than our technique of explicitly bundling channels in records. As to \texttt{JoCaml}, \texttt{OCaml} being a functional language, \texttt{JoCaml} offers pattern-matching of channel arguments, a feature that we use in several occasions here. Another extension inspired by the join-calculus is Join Java [?]. Finally, it is to be noticed that the principles of the join-calculus also inspire recent concurrency libraries: \texttt{Joins} [?] for .NET, and \texttt{Boost.Join} [?] for C++. On the one hand such libraries provide an easier approach to the high-level abstractions of the join-calculus than the linguistic approach, both on the psychic (no need to try a new language) and technical level (no need to install a new compiler). On the other hand, they offer less static checks and integrated features, such as convenient syntax.

A recent extension of (standard) ML is \texttt{Alice} [?]. As regards distributed programming, \texttt{Alice} component model allows one program to transmit a module to another in a type-safe manner, and for the receiver to link the received

\footnote{Other causes are possible though: in one case, the machine was over-heating...}
module in a controlled manner. This comes in sharp contrast with our type-less name server, and with our renunciation of code migration. As regards the basic concurrency primitives, Alice relies on explicit threads and support for data-flow synchronization. Scala is a language in its own right, not an extension, which is both object-oriented and functional. Support for concurrency in Scala is library based and using actors as the basic abstraction is encouraged [?]. The language Erlang [?] primarily targets the concurrent and distributed systems of the telecommunications industry. Its model for concurrency is reminiscent of actors, with provisions for fault tolerance; while its functional core language is rather simple. Erlang is being used for industrial developments of important size.

Clearly, there is a steady trend in programming language design: mature implementations are released that offer serious support for concurrency and distribution. The systems we have cited are based upon a variety of models. It certainly does not belong to us to claim that the join-calculus is the best amongst those models. Rather, we hope that our presentation of JoCaml at work demonstrate its elegance and expressive power.

A  Scenes used during experiments

A.1  Urchin

The scenes in experiments U are made of small cylinders arranged into a spheric shape, yielding a stylized sea urchin — see the first image of Figure 5. At order \( n \), there are \( 2^n \) cylinders. The “urchin” scenes achieve a reasonably high level of complexity in rendering, since there are many intersections to compute, still at a moderate price in memory and scene building time.

A.2  Decaying Sierpinski cube

The scenes in experiments S are decaying Sierpinski cubes of increasing order — see the second image of Figure 5. The Sierpinski cube of order \( n \) normally contains \( 20^n \) small cubes. In our “decaying” version we erase 2 sub-elements out of 7 at every induction step, and apply pseudo-random rotation and scaling. As a result, we obtain a slightly chaotic image made of approximatively \( 14.2^n \) elementary cube.
Figure 5: Images “urchin (8)” and “Sierpinski (3)”