Distributed Computing in JoCaml

Luc Maranget

Inria Paris-Rocquencourt

What is JoCaml, exactly

JoCaml *is* OCaml, plus:

- Join-Definitions (compiler-change).
- Negligible additional runtime support (we build on `Thread`).
- Significant library extensions for :
  - Concurrent programming (mostly invisible).
  - Distributed programming.

Concretely:

- Limited source incompatibility:
  - New keywords, `def`, `spawn` and `reply`.
  - New usage of `or` and `&` (you knew you'd better use `||` and `&&` in OCaml).
- Binary compatibility for JoCaml/OCaml matching versions.

JoCaml for writing concurrency utilities

Counting n events:

```
let create n =
  def st(rem) & tick() = st(rem-1)
  or st(0) & wait() = reply to wait in
  spawn st(n) ; { tick=tick; wait=wait; }
```

Available in library[a].

```
module C = JoinCount.Down
let c = C.create n
(* Asynchronous print of 0..9 *)
let () =
  for k=0 to 9 do
    spawn begin printf "%i" k ; c.C.tick() end
  done
```

[a] http://jocaml.inria.fr/manual/libref/JoinCount.Down.html

```
(* Print newline at the end *)
let () = c.C.wait () ; printf "\n%!"
```

Asynchronous print of a list of length n

```
module C = JoinCount.Down

let loop xs =
  let c = C.create (List.length xs) in
  let rec loop_rec = function
    | [] -> ()
    | x::xs ->
        spawn begin printf "%i" x ; c.C.tick() end ;
        loop_rec xs in
  loop_rec xs ;
  c.C.wait() ;
  printf "\n%!"

let () = loop [0;1;2;3;4;5;6;7;8;9;]
```

Here, using `List.length` is inelegant. In some situations (reading a file), n may not be known in advance.

Counting n events, dynamic version:

```
let create () =
  def st(n) & enter() = st(n+1) & reply to enter
  or st(n) & leave() = st(n-1)
  or st(0) & finished() & wait() = reply to wait in
  spawn st(0) ; { enter; leave; over; wait; }
```

Usage

```
def loop([]) = c.finished()
or loop(k::ks) =
  let () = c.enter() in
  begin printf "%i" k ; c.leave() end & loop(ks)

let () = spawn loop [0;1;2;...;9;]
let () = c.wait() ; printf "\n%!"
```

What can I do with JoCaml ?

Some (useful) JoCaml programs.

- Master/Slave computations:
  - Ray-tracing (hedgehogs...).
  - Running a slow Power memory model simulator on many inputs
  - Memory model testing.
- Opening shells on many distant machines.

Master and slave computations

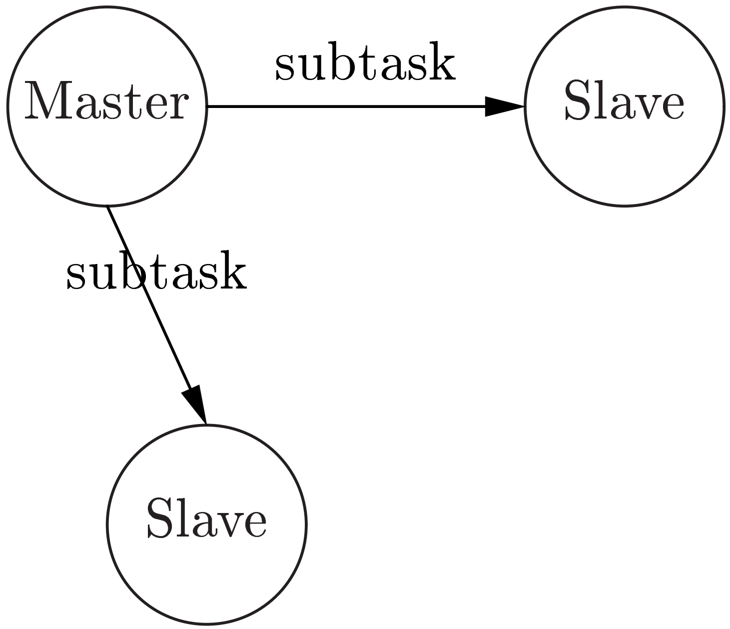
- Some "collection" $x_1, \ldots x_n$
- Some result to compute (*e.g.* some images, $\sum_{k=1}^{N} k^2$):
- That can be divided in subtasks (*e.g.* lines, k^2): $y_k = w(x_k)$.
- Subtasks are combined easily (*e.g.* store lines, $\sum$), regardless of order.

That is we compute:

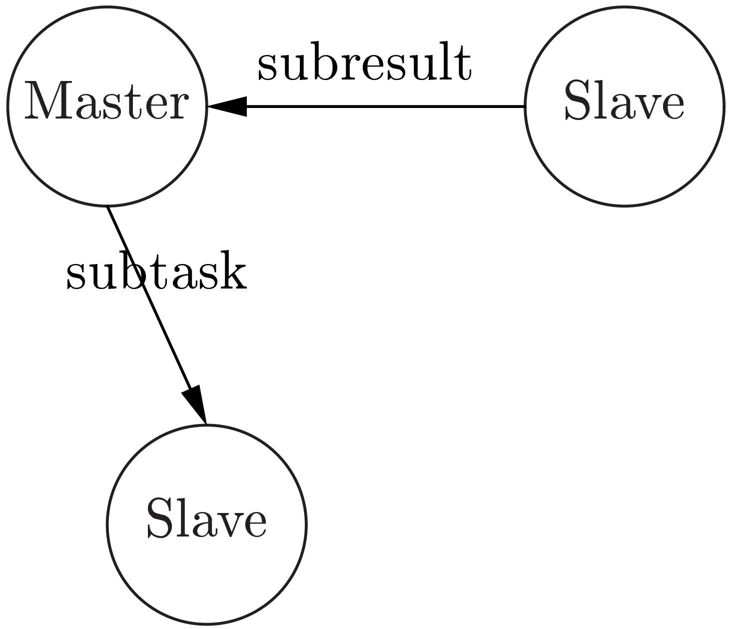
$$add(y_n, add(y_{n-1}), \ldots add(y_1), r_0)))))$$

Up to order, of course.

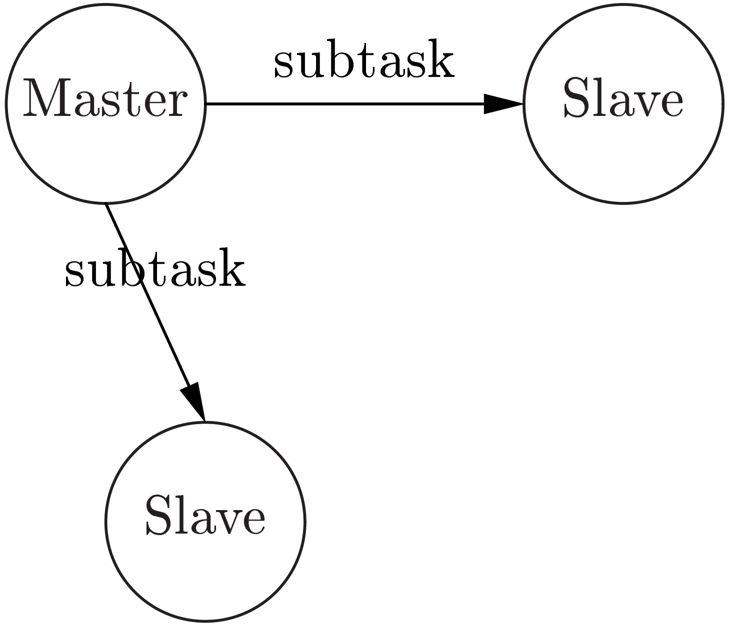
Master and slaves, giving work



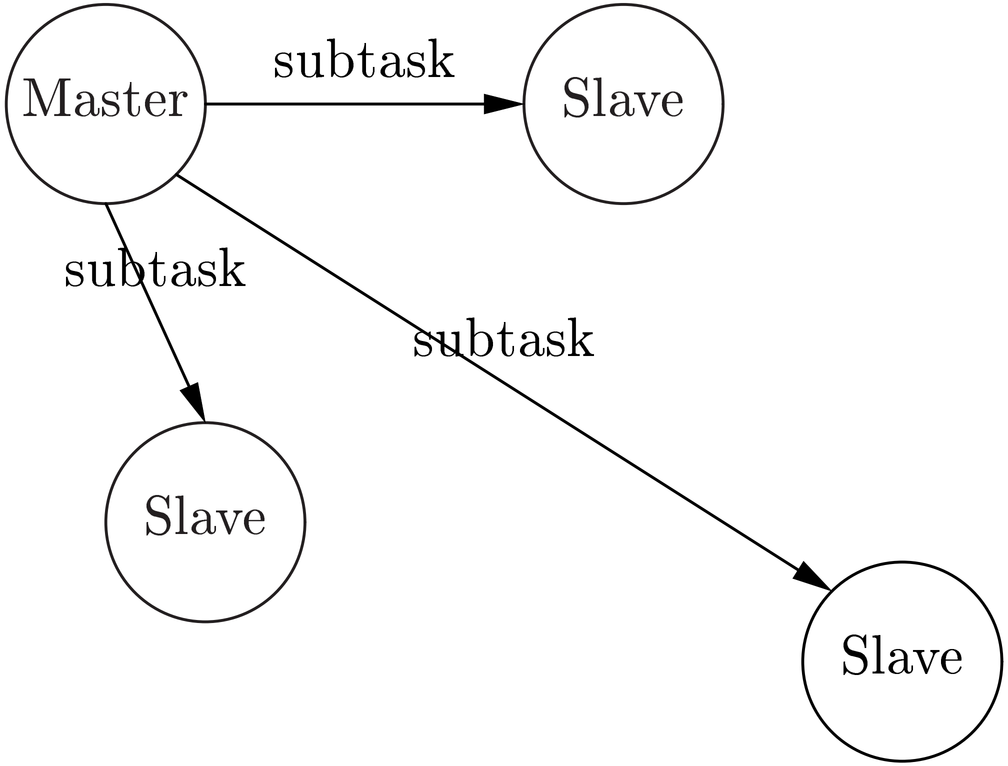
Master and slaves, one slave returns



Master and slaves, giving work



Master and slaves, one slave joins



Master and slave, program view

Slaves offer computing power, as function w.

```
type worker : subtask -> subresult
```

(Slaves will *register* their w function, for the master to call it).

Master decomposes a `collection` into substaks, distributes them to slaves and combines sub-results into result.

All this is performed by a "fold" function:

```
val fold :
  collection ->
    (subresult -> result -> result) (* add *) ->
       result (* r0 *) -> result
```

Cf. `List.fold_right`

Schematic master code

The master code is the most complex, it combines:

- The pool that organises slaves production;
- And the collector[a] that combines sub-results.

Collector, a refinement of countdown

```
let create add r0 =
  def st(n,r) & enter() = st(n+1,r) & reply to enter
  or st(n,r) & leave(y) = st(n-1,add y r)
  or st(0,r) & finished() & wait() = reply r to wait in
  spawn st(0,r0) ;
  { enter; ... ; wait; }
```

[a] http://jocaml.inria.fr/manual/libref/JoinCount.Dynamic.html

Interlude — Collections

The input collection is J.-C. Filliâtre's enumerators:

```
type t (* Collection *)
type elt (* Of elements *)
type enum (* Stateful enumerator *)
(* Start enumeration *)
val start : t -> enum
(* One step *)
val step : enum -> (elt * enum) option
```

Example: integers from n to m

```
type t = {n:int ; m:int;} type elt = int
type enum = {next:int; max:int;}
let start t = { next=t.n; max=t.m; }
let step e =
  if e.next > e.max then None
  else Some (e.next,{ e with next=e.next+1; })
```

Pool interface

```
module Make(E:Enumerable) = struct
  type ('subresult,'result) t = {
    (* Slaves register here *)
    register : (E.elt -> 'subresult) Join.chan ;
    (* Master's "fold" *)
    fold :
      E.t ->
      ('subresult -> 'result -> 'result) ->
      'result ->
      'result
  }
end
```

This is a simplified interface, complete interface in library[a]

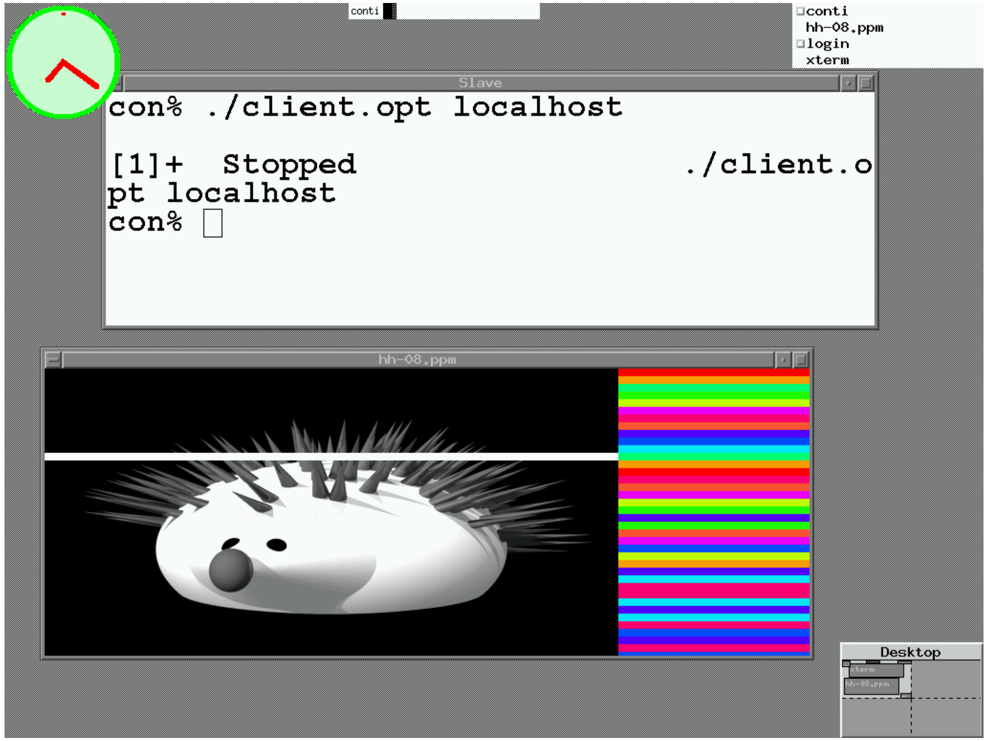
[a] http://jocaml.inria.fr/manual/libref/JoinPool.Shared.S.html

Simplified pool

```
module C = JoinCount.Dynamic

let create () =

  def worker(w) & st(e,c) = match E.step e with
    | None -> m.C.finished() & worker(w)
    | Some (x,e) ->
        st(e,c) &
        let () = c.C.enter() in
        let y = w(x) in
        m.C.leave(y) & worker(w) in

  let fold add r0 =
    let c = C.create add r0 in
    c.C.wait() in

  { register=worker; fold; }
```

Non-simplified pool

The pool from the library[a] takes *failures* into account.



The library pool handle failures by the (master) principle:

[a] `http://jocaml.inria.fr/manual/libref/JoinPool.Shared.S.html`

Better have several slaves working on the same subtask than one slave working while the others are idle.

How master and slave meet

The *name service* `Ns`, a type unsafe repository:

Master:

```
let addr = ... (* IP address + port *)
let () = Join,Site.listen addr

module P = JoinPool.Shared(...)
let pool = P.create()

let ns = Ns.here (* Ny name service *)
let () = Join.Ns.register ns "register" pool.P.register
```

Slave:

```
let addr = ... (* IP address + port *)

let ns = Ns.there addr (* Master's name service *)
let register = Join.Ns.lookup ns "register"
```

Coping with type unsafety

- Write the type of messages in a dedicated `Config` module:

```
type worker = subtask -> subresult
type register = worker Join.chan
let magic = "XXX000"
```

- Master: register magic in name service, use type cast:

```
let () = Join.Ns.register ns "magic" Config.magic
let () =
  Join.Ns.register ns "register" (register:Config.register)
```

- Slave: check magic from name service, use type cast:

```
let magic = Join.Ns.lookup ns "magic"
let () = if magic <> Config.magic then failwith "Bad magic"
let (register:Config.register) = Join.Ns.lookup ns "register"
```

Will work when master and slave share `Config`.

How slave stops

An abstraction for sites `Site`.

Master: does nothing, will terminate normally.

```
let result = pool.P.fold ...

let () = print_result result ; exit 0
```

Slave: register a *guard* on master's site:

```
...
let () = register worker

let master = Join.Site.there addr
def wait() & go() = reply to wait
let () = Join.Site.at_fail master go
let () = wait() ; exit 0
```

Proved to be convenient: killing the master kills all slaves!

Coping with OCaml thread implementation

We focus on distributed applications. Nevertheless, a given machine may have several cores…

It is well known that OCaml threads do not run concurrently. We fork/exec on a given machine, as we do for several machines.

In practise, `slave -nclients` n `-host` a

- Forks/execs slave n times, and dies. A shorthand for:

$$\underbrace{\texttt{slave -host } a \texttt{ \& } \cdots \texttt{ slave -host } a \texttt{ \&}}_{\times n}$$

- Or follows "coordination" idiom:
  - Get external program from master.
  - Register n workers.
  - Each worker call will fork/exec the external program.

Coordination — Fork/Exec

The library features `JoinProc`[a] (basic interface) and `JoinTextProc`[b] (text processing). As replacements for `Unix.open_`... functions.

The synchronous text processing `JoinTextProc` is by far the simplest:

```
type text = string list (* List of lines *)
type result = {
   st : Unix.process_status; (* Child status *)
   out : text; (* Standard output of child *)
   err : text; (* Standard error of child *)
}
type t = {
  wait : unit -> result; (* Get result (will block) *)
```

[a] http://jocaml.inria.fr/manual/libref/JoinProc.html

[b] http://jocaml.inria.fr/manual/libref/JoinTextProc.html

```
  kill : int -> unit;  (* Kill child *)
}

val open_full : string -> string array -> text -> t
```

Example, forking a shell

```
module P = JoinTextProc.Sync

let shell cmds =
  let proc = P.open_full "/bin/sh" [| "/bin/sh"; "+e";|] cmds in
  let r = proc.P.wait() in
  match r.P.st with
  | Unix.WEXITED 0 -> r.P.stdout
  | _ ->
      eprintf "Shell failed:\n" ;
      List.iter (eprintf "%s\n%!") (r.P.err) ;
      raise Error
```

Demo: show simple master and slave that follow the coordination idiom.

Example, `ppcmem`

`ppcmem` is a simulator of the memory model of Power machines.

Our claim: our model is not unvalidated by experiments on hardware (which are running independently).

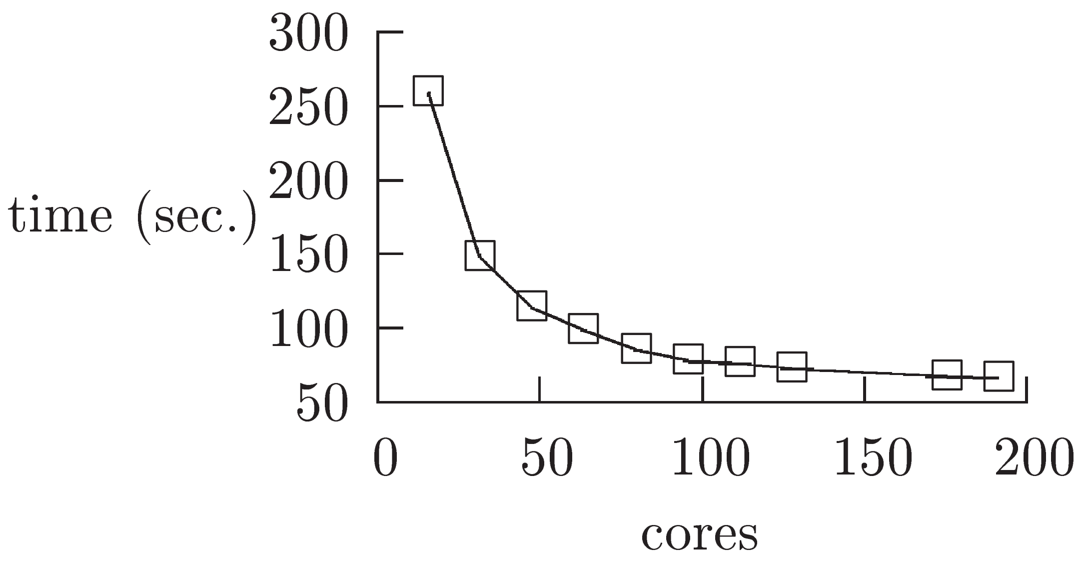
We need to run `ppcmem` as much as possible.

- The deadline is two weeks away and `ppcmem` is so slow.
- We have 1382 tests to run.
- We have a 16 nodes $\times$ 12 cores cluster (192 cores).
- We managed to get the results for the 648 tests that run in less than 12 hours and 8Mb.

Performance

Setting up regression test suite for `ppcmem`.

- Finding the 487 tests (out of 1382) that run in less than one minute: 330 sec. (using 192 cores)
- Re-running those tests:

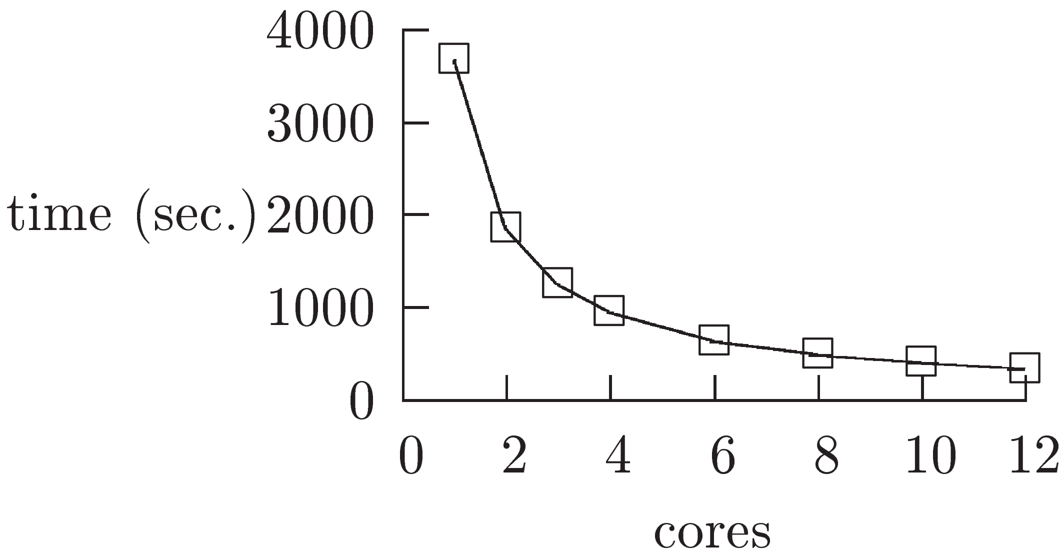


A little more than 1 min. for the whole batch.

Using one multicore machine

My English friends have no cluster and have not installed JoCaml.

They use `make -j` n . But they have to be patient...

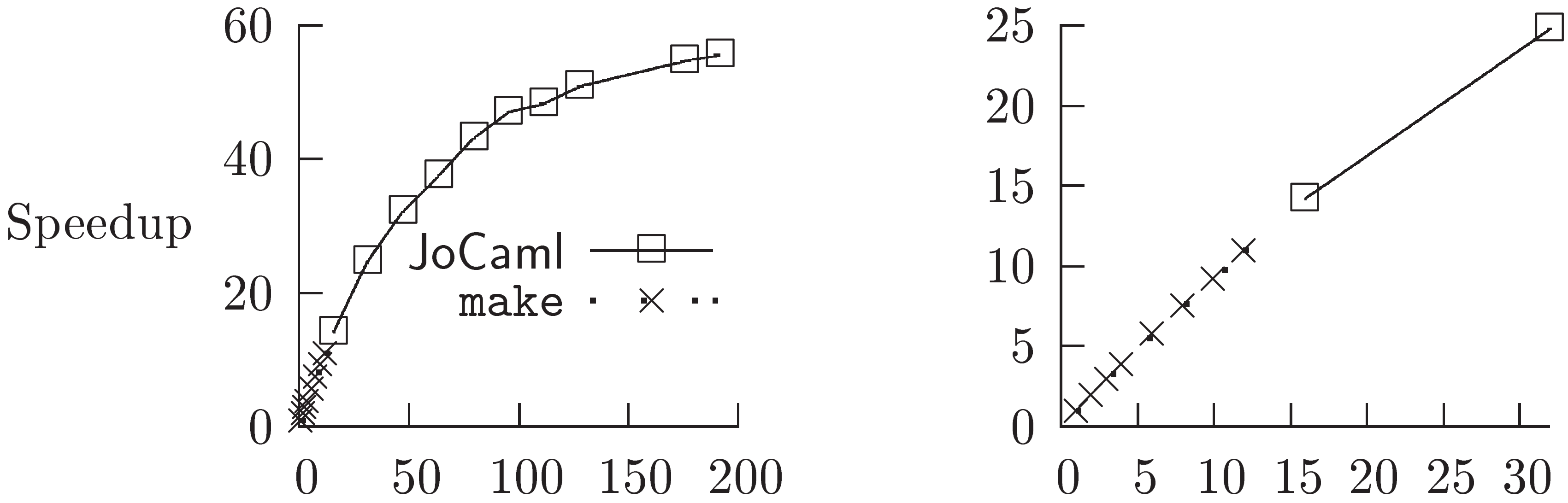


Hence, with 12 cores we wait for 5 min 30 sec.

And my English friends have 4 cores only (about 16 min.)

Running the tests sequentially takes about 61 min.

Speed(up) and beyond



The JoCaml solution is also rather convenient and flexible:

- Takes care of installing `ppcmem` on nodes;
- Shares code from other tools;
- One easily adds supplementary slaves or kill some;
- One easily runs several batches of tests concurrently.

Conclusion

I have presented a field of applications for JoCaml:

- Some applications are "embarrassingly" parallel.
- Some machines are "massively" parallel (or some networks consists in many machines).
- It does not mean that coding them is easy (failures, synchronisations, ...).
- JoCaml (and its library) helps in running "embarrassingly" parallel applications on "massively" parallel machines.

More involved situations (distributed algorithms, less favourable compute/communicate ratio) are another story.