Synchronous/Asynchronous Parallelism

Marc Pouzet

École normale supérieure
Marc.Pouzet@ens.fr

April 4, 2019
Interlude: Parallelism and time sharing

- It is possible to launch several processes on a computer even if it has a single processor...
- The computer fundamentally does a single thing at a time.
- It gives the impression of doing several because it goes faster than our perception of change.
- Things are easy when applications are independent from each others.

Is this parallelism useful for programming a real-time system?

Read absolutely: [“The problem with threads”, Edward Lee, 2006] ¹

¹https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
Let us draw a swing in OCaml...

```ocaml
let swing center radius alpha_init speed =
  let alpha = ref alpha_init in
  while true do
    alpha := !alpha +. speed;
    draw center radius !alpha;
  done

let main = (swing c1 r pi) speed
```

(See OCaml code on the web page.)
Two swings?

Put two in parallel.

```plaintext
let swing center radius alpha_init speed =
  let alpha = ref alpha_init in
  while true do
    alpha := !alpha +. speed;
    draw center radius !alpha;
  done

let main =
  Thread.create (swing c1 r pi) speed;
  Thread.create (swing c2 r 0.) speed
```
It does not work: why?

• The OS scheduler decides to execute some steps of the first then some of the second, then some of the first, etc.
• What happen if an extra swing is added?
• The programmer has no simple mean to control the scheduling policy of the OS such that swing move at the same pace.
• No precise time synchronization.
Synchrony/Asynchrony

Can we fix this by giving a hint to the scheduler so that it switches from a task to the other?

```ocaml
let swing center radius alpha_init speed =
  let alpha = ref alpha_init in
  while true do
    alpha := !alpha +. speed;
  draw center radius !alpha;
  Thread.yield ()
  done

let main =
  Thread.create (swing c1 r pi) speed;
  Thread.create (swing c2 r 0.) speed
```
Synchrony/Asynchrony

Parallelism by interleaving

- `swing1; swing1; swing1; swing2;...`
- `swing1; swing2; swing1; swing2;...`
- Non reproducible behavior (depend on the scheduler and number of running processes).
- Add explicit synchronizations: expensive (at run-time), subtle (dead-locks, starvation), obfuscate the program.

This is even worst on multi-core processors

The semantics by interleaving is wrong: putting two threads running on a shared memory machine exhibits extra behaviors. \(^2\)

Remark

OCaml threads do not run in parallel.

\(^2\)Cf. work by Zappa Nardelli [9].
Synchrony/Asynchrony
Add extra code to explicitly synchronise processes and ensure that every process do a single step.

```ocaml
let swing center radius alpha_init speed m1 m2 =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw center radius !alpha;
    Mutex.unlock m2; Mutex.lock m1
  done

let main =
  let m1, m2 = Mutex.create (), Mutex.create () in
  Mutex.lock m1;
  Mutex.lock m2;
  Thread.create (swing c1 r a1) speed m1 m2;
  Thread.create (swing c2 r a2) speed m2 m1
```

Synchrony/Asynchrony

A solution with a synchronization barrier.

```ocaml
let barriere n =
  let mutex, attente =
    Mutex.create (), Mutex.create () in
  Mutex.lock attente;
  let nb_att = ref 0 in
  fun () ->
    Mutex.lock mutex;
    incr nb_att;
    if !nb_att = n then begin
      for i = 1 to n-1 do Mutex.unlock attente done;
      nb_att := 0; Mutex.unlock mutex
    end else begin
      Mutex.unlock mutex; Mutex.lock attente
    end
```
Synchrony/Asynchrony

```ocaml
let stop = barriere 3

let swing center radius alpha_init speed =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw center radius !alpha;
    stop ()
  done

let main =
  Thread.create (swing c1 r a1) speed;
  Thread.create (swing c2 r a2) speed;
  Thread.create (swing c3 r a3) speed
```

Not modular; error-prone (deadlock/starvation) and inefficient when systems are tightly coupled (at lot of synchronization between them).
Design domain specific languages (Berry’89 [4])

What’s the matter with general purpose sequential languages and concurrency from the OS?

Sequential Programming:

- Turing complete, too much expressiveness, too hard to verify.
- Complexity is not where it is needed: pointer arithmetic, dynamic allocation, etc.
- **Parallelism while ensuring determinism** is absent whereas it is fundamental.
Concurrent (asynchronous) programming

- Time is not taken into account, both in language and in the semantics, non determinism is unavoidable:
  E.g., \texttt{wait 2 second; wait 3 second} \neq \texttt{wait 5 second}

- No precise synchronization, no instantaneous broadcast:
  E.g., \texttt{wait 60 second; send minute to B? (send minute to B; send minute to C)?}

- Parallel composition and rendez-vous communication compose poorly: adding an extra listening process changes the semantics.
- Observing/debugging a program may change its behavior.
What if time was made logical?
Suppose that the machine is infinitely fast...
Do as if there is an global time scale, shared by all processes. Do not try to go as fast as possible but synchronise/agree on it.

The swing in ReactiveML [8]

```plaintext
let process swing center radius alpha_init speed =
  let alpha = ref alpha_init in
  while true do
    alpha := !alpha +. speed;
    drawcenter radius !alpha;
    pause
  done

let process main =
  run (swing c1 r pi speed)
|| run (swing c2 r 0. speed)
```

Synchrony

It is possible to add as many swings as we want without modifying the overall behavior.

```haskell
let process main =
    run (swing c1 r a1 speed)
  || run (swing c2 r a2 speed)
  || run (swing c2 r a3 speed)
```

The Synchronous Model of Time

A global **logical time** that is shared by all processes. This defines instants on which processes can synchronize.

A **sequence of ticks**. Instruction `pause` means: *wait for the next tick.*
A bit of History

In the 80’s, several team invented (at about the same time) languages dedicated to the design/implementation of control-systems.

- **Lustre** (Caspi & Halbwachs, Grenoble): data-flow (block-diagrams), functional model (deterministic);
- **Signal** (Benveniste & Le Guernic, Rennes): data-flow but relational (to define also non-deterministic systems);
- **Esterel** (Berry & Gonthier, Sophia): hierarchical automata and process algebra

Base it on the **mathematical culture and models** of the field of embedded control-systems.
The Synchronous Model of Time

Simplify the programming of real-time system by considering first that exact time can be neglected, i.e, What is the worst case?

Time becomes logical: it is a sequence of instantaneous reactions.

1. Do as if the machine is infinitely fast: all processes listen to each others and see the very same inputs. E.g., the radio.
2. Check a posteriori the correspondance between logical time and real-time: is the machine fast enough? What is the worst case?

Worst case execution time (WCET): $\max_{n \in \mathbb{N}}(t_n - t_{n-1}) \leq \text{bound}$. This is checked on the actual platform.
Is it that original?

The conductor
All musician share a global time scale, that of the conductor.

Dancers:
They synchronise on music. This is the way several dancers agree and do the same thing.

Synchronous circuits:
A global clock shared by all gates/registers.

That is:
First reason ideally, neglecting light speed (orchestra), sound (dancers), electricity (circuits).

Then measure actual computational time and transmission delays. Verify that the synchronous abstraction is reasonable.
E.A. Ashcroft and W.W. Wadge.
Lucid, a non procedural language with iteration.

The synchronous languages 12 years later.

A. Benveniste, P. LeGuernic, and Ch. Jacquemot.
Synchronous programming with events and relations: the SIGNAL language and its semantics.

G. Berry.
Real time programming: Special purpose or general purpose languages.

G. Berry and G. Gonthier.
The Esterel synchronous programming language, design, semantics, implementation.


P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice.
Lustre: a declarative language for programming synchronous systems.

Gilles Kahn.
The semantics of a simple language for parallel programming.
In *IFIP 74 Congress.* North Holland, Amsterdam, 1974.

Louis Mandel and Marc Pouzet.
ReactiveML, a Reactive Extension to ML.
x86-tso: a rigorous and usable programmer’s model for x86 multiprocessors.