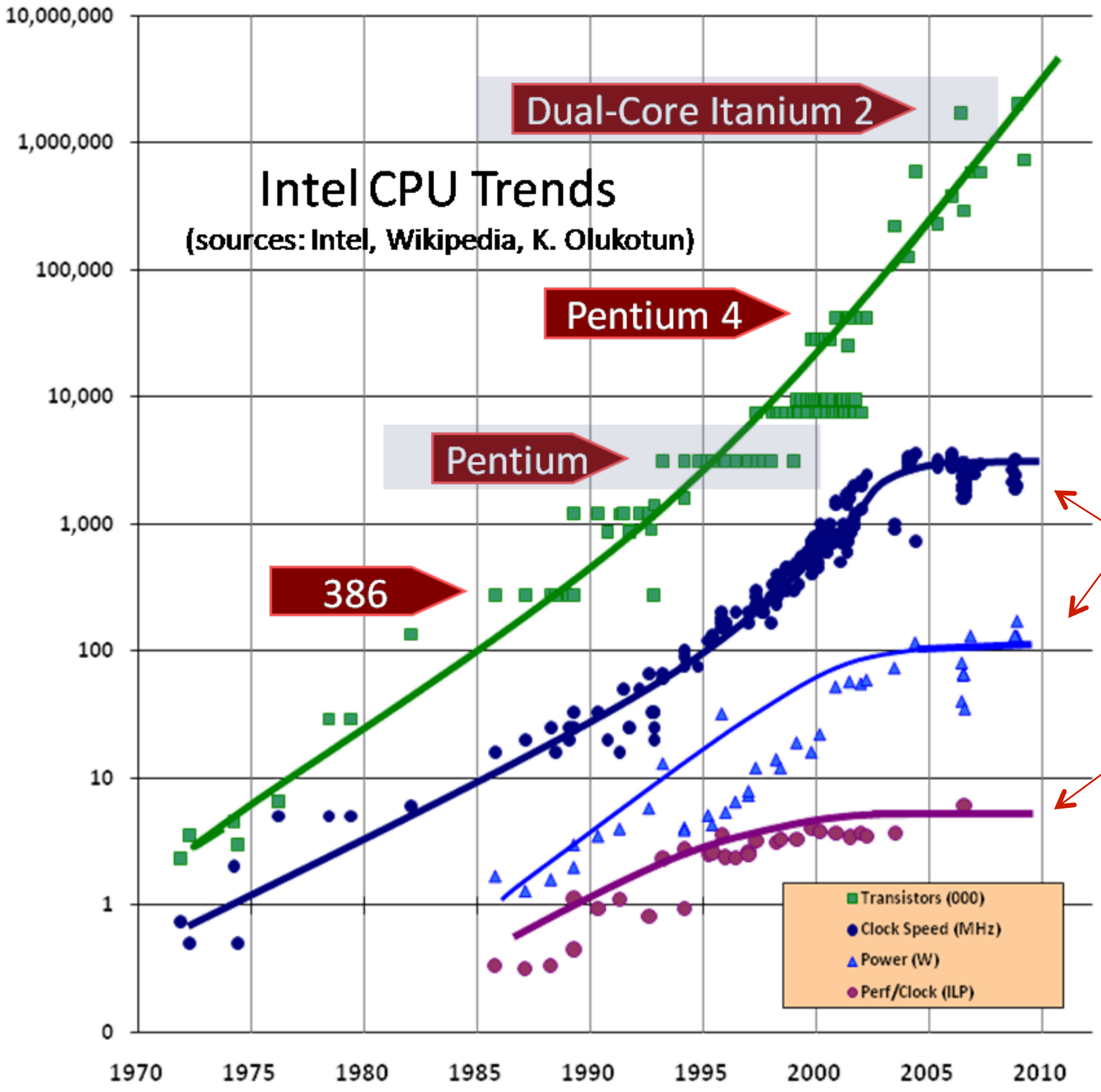
Parallelism and Concurrency

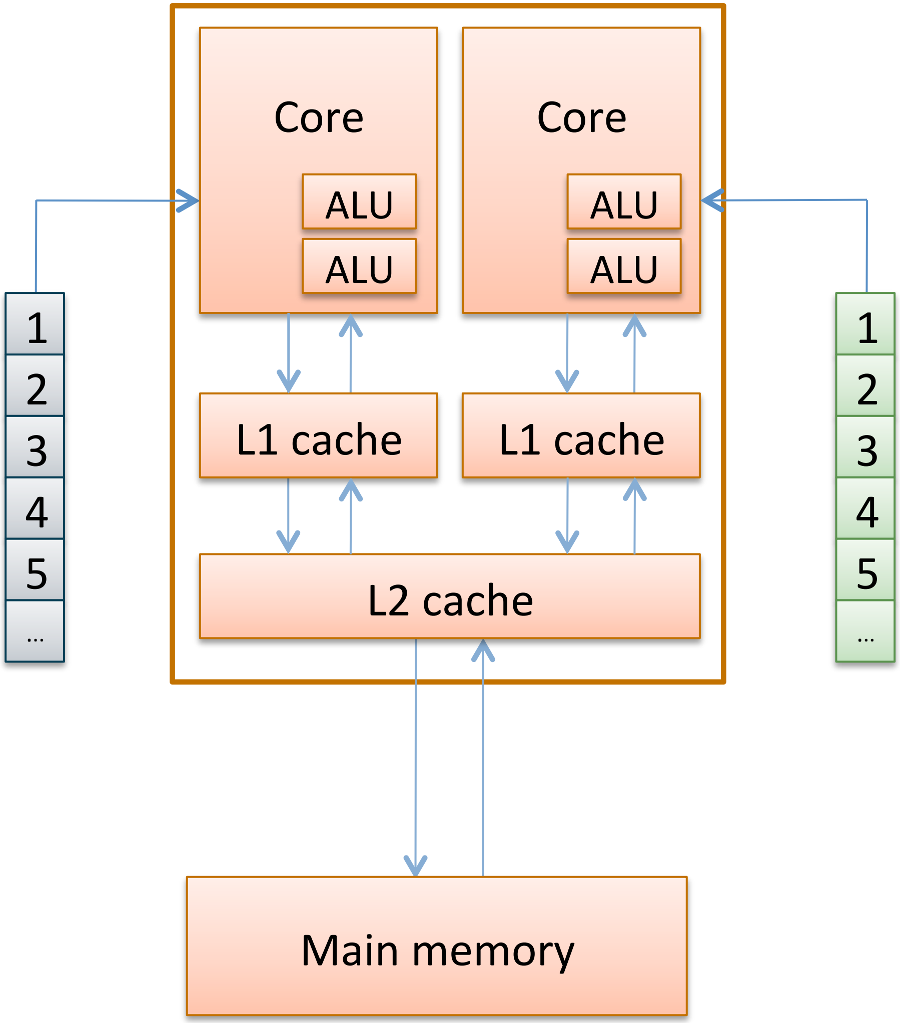
COS 326
David Walker
Princeton University



Power to chip peaking

Darn!
Intel engineers no longer optimize my programs while
I watch TV!

Last Time: Multi-core Hardware & Data Centers



1
2
3
4
5
…

Core

ALU

ALU

Core

ALU

ALU

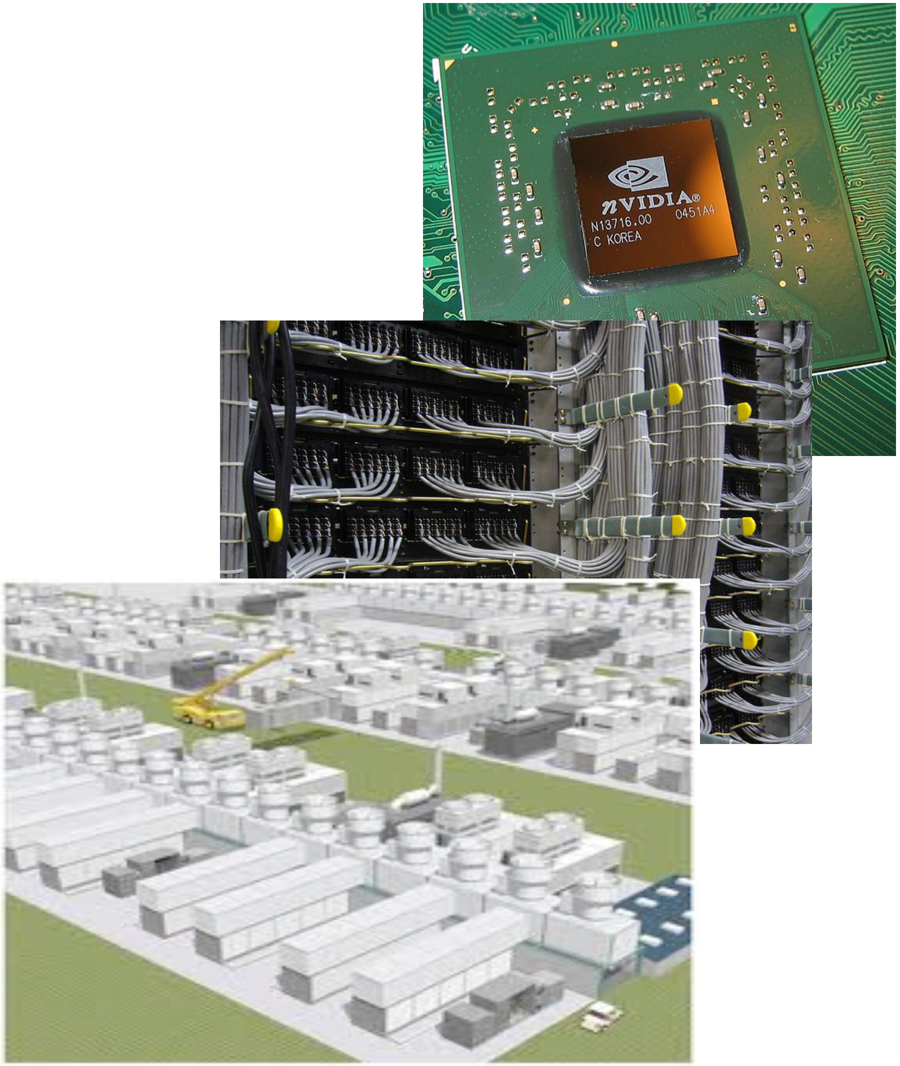
1
2
3
4
5
…

L1 cache

L1 cache

L2 cache

Main memory



PARALLEL AND CONCURRENT PROGRAMMING

Speedup

- *Speedup*: the ratio of parallel program execution time to sequential program execution time.
- If T(p) is the time it takes to run a computation on p processors

speedup(p) = T(p)/T(1)

- A parallel program has *perfect speedup* (aka *linear speedup*) if

T(p)/T(1) = speedup = p

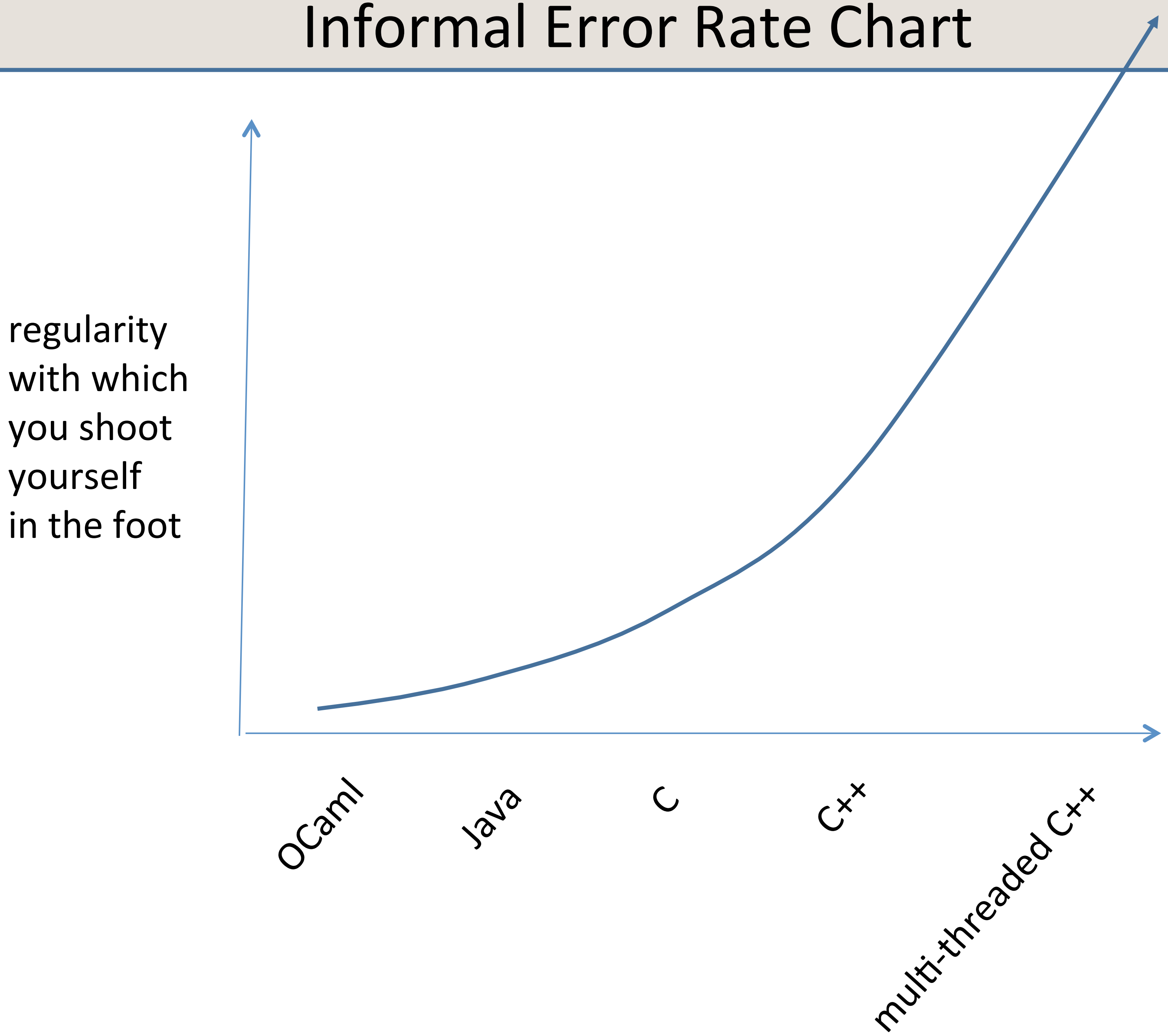
- *Bad news: Not every program can be effectively parallelized.*
  - in fact, very few programs will scale with perfect speedups.
  - we certainly can't achieve perfect speedups automatically
  - limited by sequential portions, data transfer costs, ...

Most Troubling…

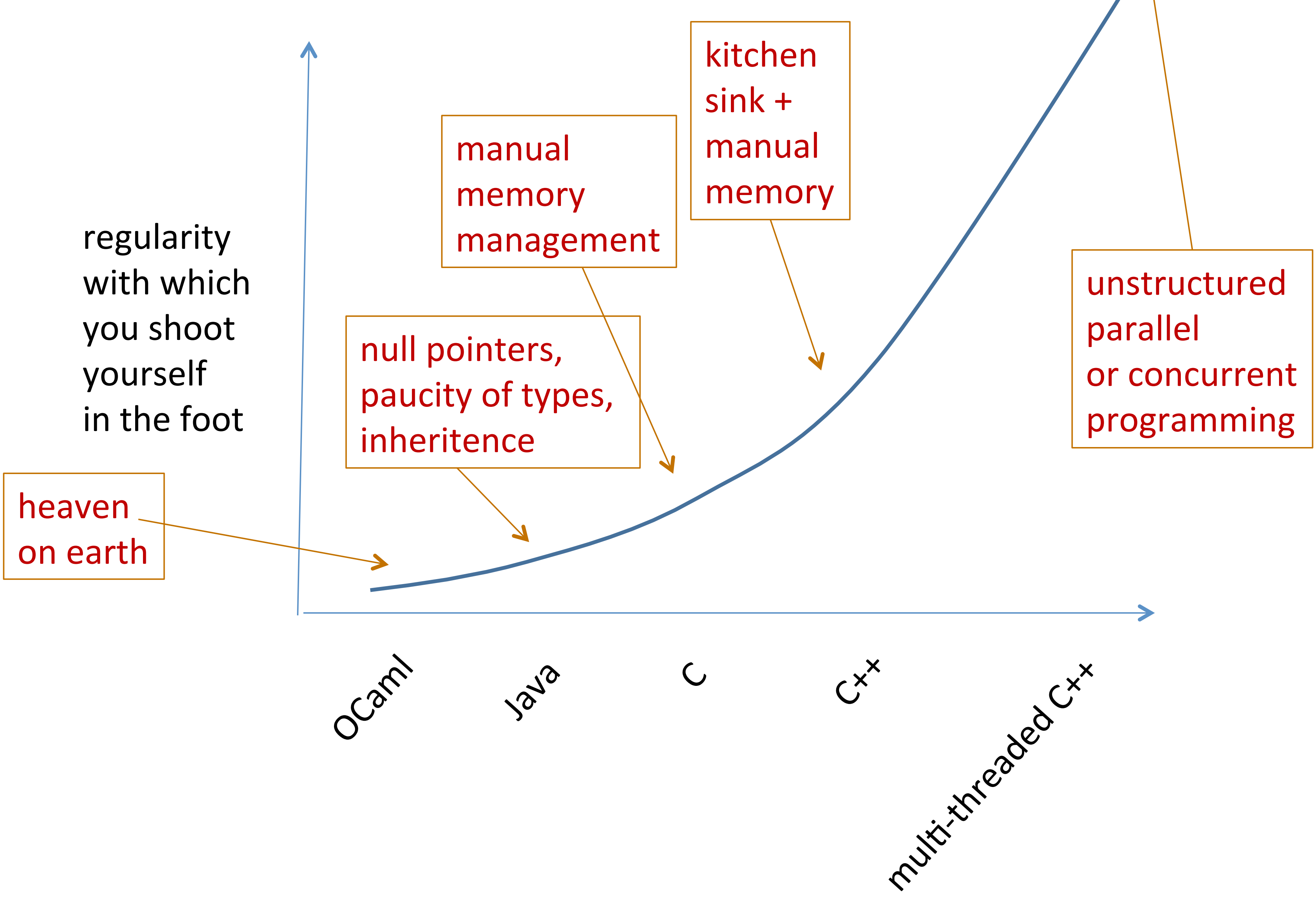
Most, *but not all*, parallel and concurrent programming models are far harder to work with than sequential ones:

- *They introduce non-determinism*
  - the root of (almost all) evil
  - program parts suddenly have many different outcomes
    - they have different outcomes *on different runs*
    - debugging requires considering *all of the possible outcomes*
    - horrible *heisenbugs* hard to track down
- *They are non-modular*
  - module A implicitly influences the outcomes of module B
- *They introduce new classes of errors*
  - race conditions, deadlocks
- *They introduce new performance/scalability problems*
  - busy-waiting, sequentialization, contention,

Informal Error Rate Chart



Informal Error Rate Chart



heaven on earth

null pointers, paucity of types, inheritence

manual memory management

kitchen sink + manual memory

unstructured parallel or concurrent programming

Solid Parallel Programming Requires

1. Good sequential programming skills.
   - all the things we've been talking about: use modules, types, ...

2. Deep knowledge of the application.

3. Pick a correct-by-construction parallel programming model
   - whenever possible, a parallel model with semantics that coincides with sequential semantics
     - whenever possible, reuse well-tested libraries that hide parallelism
   - whenever possible, a model that cuts down non-determinism
   - whenever possible, a model with fewer possible concurrency bugs
   - if bugs can arise, know and use safe programming patterns

4. Careful engineering to ensure scaling.
   - unfortunately, there is sometimes a tradeoff:
     - reduced non-determinism can lead to reduced resource utilization
   - synchronization, communication costs may need optimization

OUR FIRST PARALLEL PROGRAMMING MODEL: THREADS

Threads: A Warning

- *Concurrent Threads: the classic shoot-yourself-in-the-foot concurrent programming model*
  - all the classic error modes

- Why Threads?
  - almost all programming languages will have a threads library
    - OCaml in particular!
  - you need to know where the pitfalls are
  - the assembly language of concurrent programming paradigms
    - we'll use threads to build several higher-level programming models

Threads

- Threads: an abstraction of a processor.
  - programmer (or compiler) decides that some work can be done in parallel with some other work, e.g.:

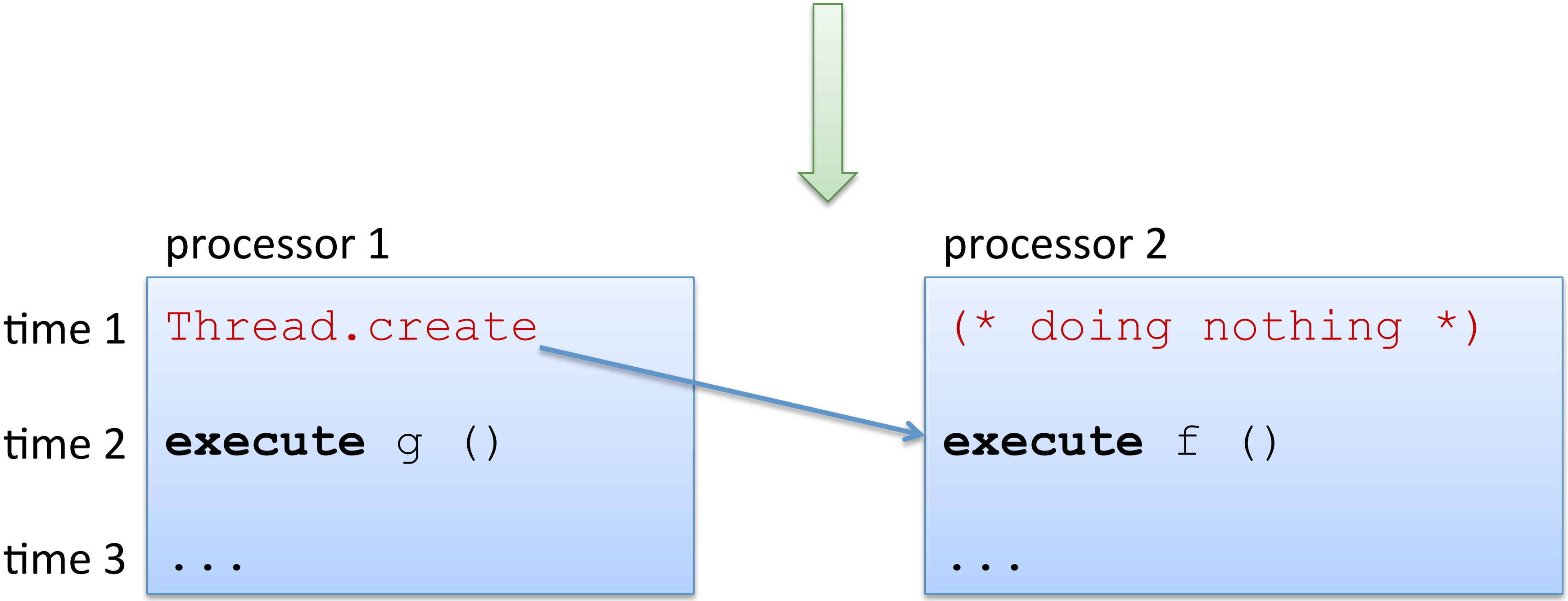
```
let _ = compute_big_thing() in
let y = compute_other_big_thing() in
...
```

  - we *fork* a thread to run the computation in parallel, e.g.:

```
let t = Thread.create compute_big_thing () in
let y = compute_other_big_thing () in
 ...
```

Intuition in Pictures

```
let t = Thread.create f () in
let y = g () in
 ...
```



	processor 1	processor 2
time 1	`Thread.create`	`(* doing nothing *)`
time 2	`execute g ()`	`execute f ()`
time 3	`...`	`...`

Of Course…

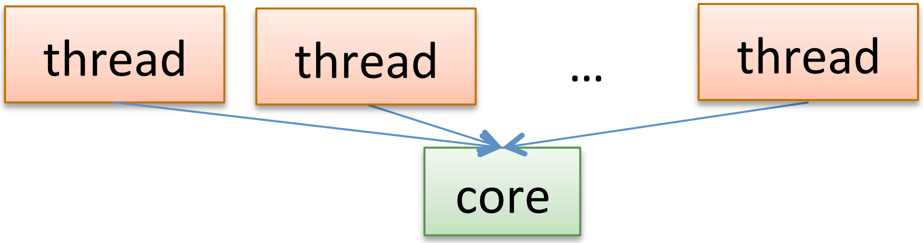
Suppose you have 2 available cores and you fork 4 threads. In a typical multi-threaded system,

- the operating system provides *the illusion* that there are an infinite number of processors.
  - not really: each thread consumes space, so if you fork too many threads the process will die.

- it *time-multiplexes* the threads across the available processors.
  - about every 10 msec, it stops the current thread on a processor, and switches to another thread.
  - so a thread is really a *virtual processor*.

OCaml, Concurrency and Parallelism

Unfortunately, even if your computer has 2, 4, 6, 8 cores, OCaml cannot exploit them. It multiplexes all threads over a single core



Hence, OCaml provides concurrency, but not parallelism. *Why?* Because OCaml (like Python) has no parallel run time or garbage collector. Lots of other functional languages (Haskell, F#, ...) do.

Fortunately, when thinking about *program correctness*, it doesn't matter that OCaml is not parallel -- I will often pretend that it is. But we won't be able to get the same kinds of speedups. :-(

Coordination

```
Thread.create : ('a -> 'b) -> 'a -> Thread.t

let t = Thread.create f () in
let y = g () in
  ...
```

How do we get back the result that `t` is computing?

First Attempt

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
  match !r with
    | Some v -> (* compute with v and y *)
    | None -> ???
```

What’s wrong with this?

Second Attempt

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

Two Problems

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

First, we are *busy-waiting*.

- consuming cpu without doing something useful.
- the processor could be either running a useful thread/program or power down.

Two Problems

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

Second, an operation like r := Some v may not be *atomic.*

- r := Some v requires us to copy the bytes of Some v into the ref r
- we might see part of the bytes (corresponding to Some) before we've written in the other parts (e.g., v).
- So the waiter might see the wrong value.

Atomicity

Consider the following:

```
let inc(r:int ref) = r := (!r) + 1
```

and suppose two threads are incrementing the same ref r:

Thread 1	Thread 2
`inc(r);`	`inc(r);`
`!r`	`!r`

If r initially holds 0, then what will Thread 1 see when it reads r?

Atomicity

The problem is that we can’t see exactly what instructions the compiler might produce to execute the code.

It might look like this:

Thread 1	Thread 2
`EAX := load(r);`	`EAX := load(r);`
`EAX := EAX + 1;`	`EAX := EAX + 1;`
`store EAX into r`	`store EAX into r`
`EAX := load(r)`	`EAX := load(r)`

Atomicity

But a clever compiler might optimize this to:

Thread 1	Thread 2
`EAX := load(r);`	`EAX := load(r);`
`EAX := EAX + 1;`	`EAX := EAX + 1;`
`store EAX into r`	`store EAX into r`
~~`EAX := load(r)`~~	~~`EAX := load(r)`~~

Atomicity

Furthermore, we don’t know when the OS might interrupt one thread and run the other.

Thread 1	Thread 2
`EAX := load(r);`	`EAX := load(r);`
`EAX := EAX + 1;`	`EAX := EAX + 1;`
`store EAX into r`	`store EAX into r`
`EAX := load(r)`	`EAX := load(r)`

(The situation is similar, but not quite the same on multi-processor systems.)

The Happens Before Relation

We don't know exactly when each instruction will execute, but there are some constraints: the *Happens Before* relation

Rule 1: Given two expressions (or instructions) in sequence, e1; e2 we know that e1 happens before e2.

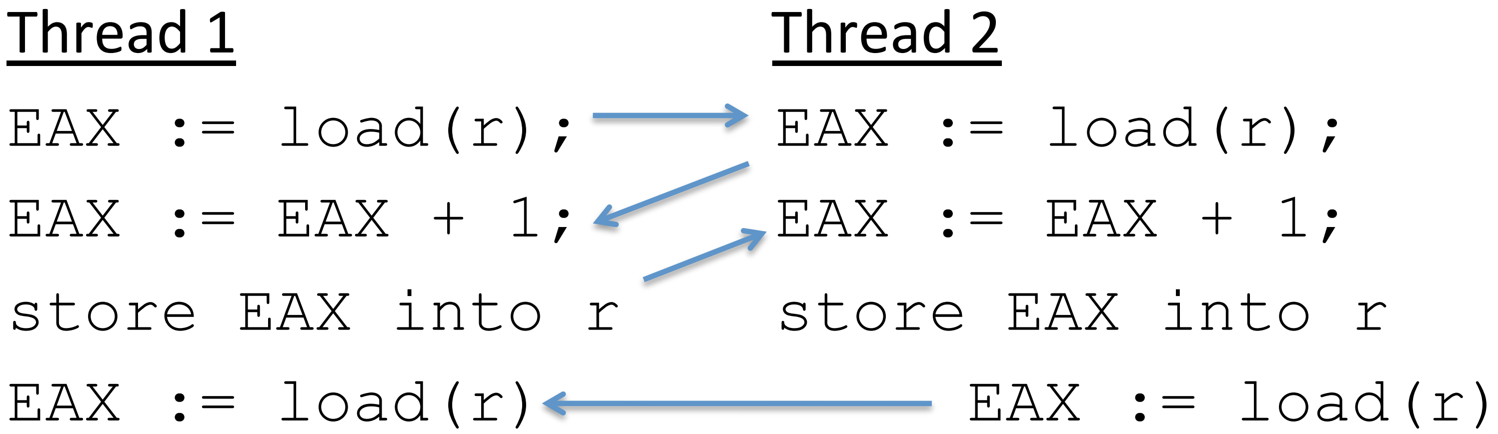
Rule 2: Given a program:

```
let t = Thread.create f x in
....
Thread.join t;
e
```

we know that (f x) happens before e.

Atomicity

One possible interleaving of the instructions:

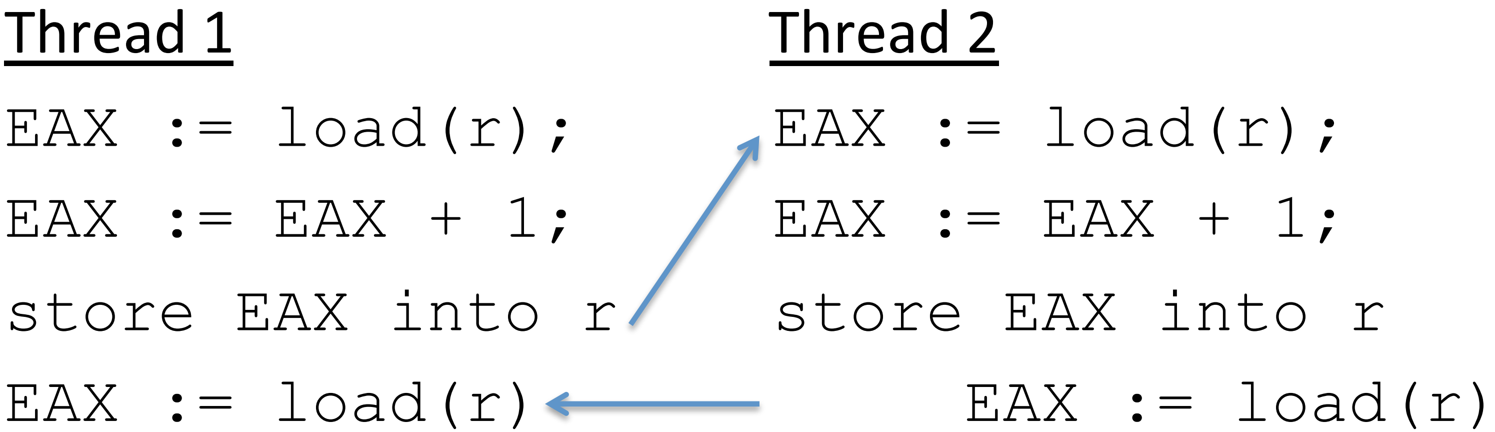


```
Thread 1                    Thread 2
EAX := load(r);             EAX := load(r);
EAX := EAX + 1;             EAX := EAX + 1;
store EAX into r            store EAX into r
EAX := load(r)                     EAX := load(r)
```

What answer do we get?

Atomicity

Another possible interleaving:

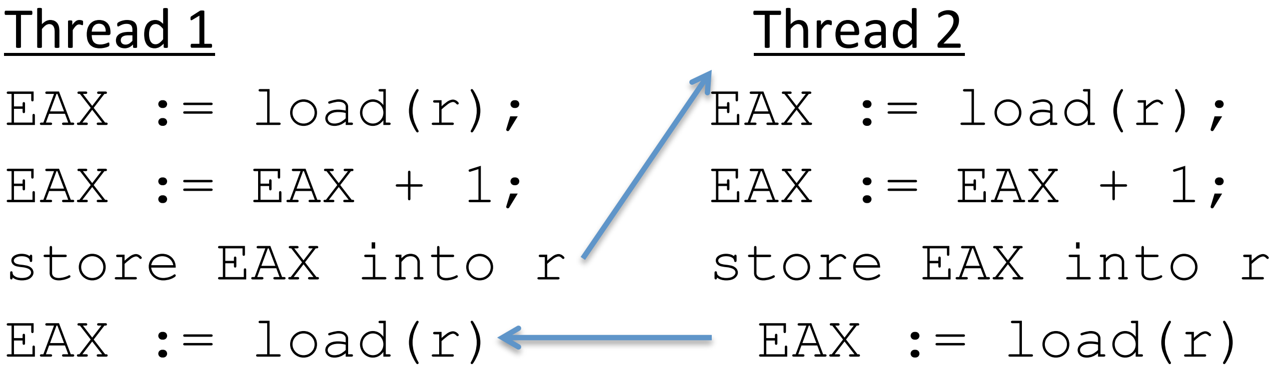


```
Thread 1                   Thread 2
EAX := load(r);            EAX := load(r);
EAX := EAX + 1;            EAX := EAX + 1;
store EAX into r           store EAX into r
EAX := load(r)                   EAX := load(r)
```

What answer do we get this time?

Atomicity

Another possible interleaving:



Thread 1

```
EAX := load(r);
EAX := EAX + 1;
store EAX into r
EAX := load(r)
```

Thread 2

```
EAX := load(r);
EAX := EAX + 1;
store EAX into r
 EAX := load(r)
```

What answer do we get this time?

Moral: The system is responsible for *scheduling* execution of instructions.

Moral: This can lead to an enormous degree of *non-determinism*.

Atomicity

In fact, today's multi-core processors don't treat memory in a *sequentially consistent* fashion.

```
Thread 1            Thread 2
EAX := load(r);     EAX := load(r);
EAX := EAX + 1;     EAX := EAX + 1;
store EAX into r    store EAX into r
EAX := load(r)          EAX := load(r)
```

That means that *we can't even assume that what we will see corresponds to some interleaving of the threads' instructions!*

Beyond the scope of this class (and my brain…)

Summary: Interleaving & Race Conditions

Calculate possible outcomes for a program by considering all of the possible interleavings of the *atomic* actions performed by each thread.

- Subject to the *happens-before* relation.
  - can't have a child thread's actions happening before a parent forks it.
  - can't have later instructions execute earlier *in the same thread.*
- Here, *atomic* means indivisible actions.
  - For example, on most machines reading or writing a 32-bit word is atomic.
  - But, writing a multi-word object is usually *not* atomic.
  - Most operations like "b := b - w" are implemented in terms of a series of simpler operations such as
    - r1 = read(b); r2 = read(w); r3 = r1 – r2; write(b, r3)

Reasoning about all interleavings is ~~*hard*~~. just about impossible for people

- Number of interleavings grows exponentially with number of statements.
- It's hard for us to tell what is and isn't atomic in a high-level language.
- YOU ARE DOOMED TO FAIL IF YOU HAVE TO WORRY ABOUT THIS STUFF!

Back to the Coordination Problem

```
Thread.create : ('a -> 'b) -> 'a -> Thread.t

let t = Thread.create f () in
let y = g () in
  ...
```

How do we get back the result that `t` is computing?

One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
    Thread.join t ;
    match !r with
    | Some v -> (* compute with v and y *)
    | None -> failwith “impossible”
```

One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
    Thread.join t ;
    match !r with
    | Some v -> (* compute      ith v and y *)
    | None -> failwith “impos        e”
```

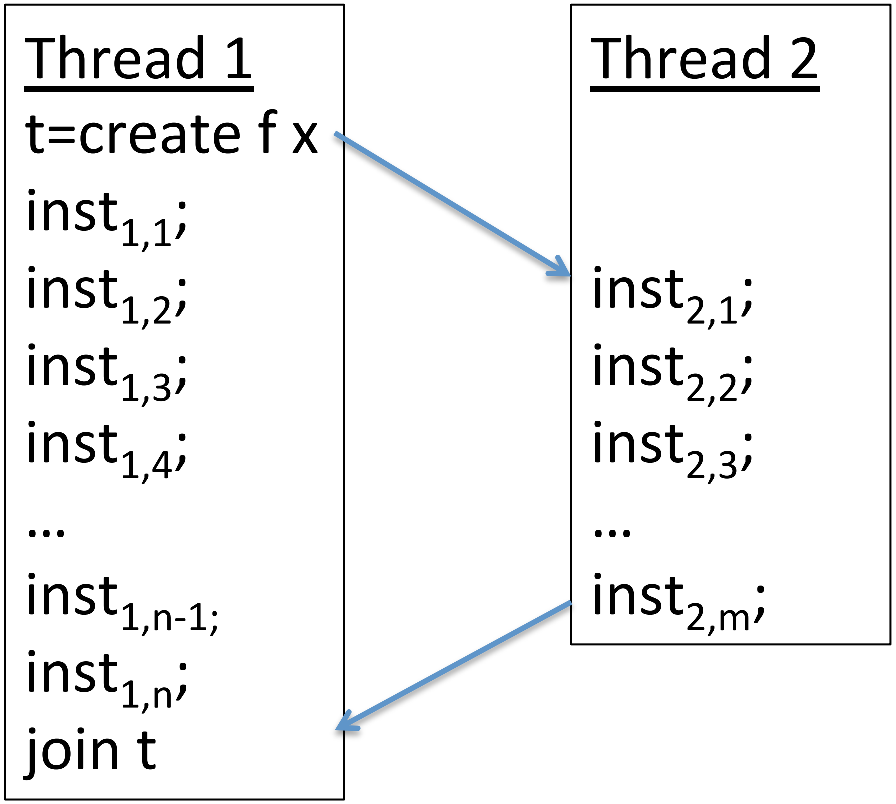
`Thread.join t` causes the current thread to *wait* until the thread `t` terminates.

One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
    Thread.join t ;
    match !r with
    | Some v -> (* compute with v and y *)
    |   ne -> failwith "impossible"
```

So after the join, we know that any of the operations of t have *completed*.

In Pictures



Thread 1

t=create f x

inst$_{1,1}$;

inst$_{1,2}$;

inst$_{1,3}$;

inst$_{1,4}$;

…

inst$_{1,n-1;}$

inst$_{1,n}$;

join t

Thread 2

inst$_{2,1}$;

inst$_{2,2}$;

inst$_{2,3}$;

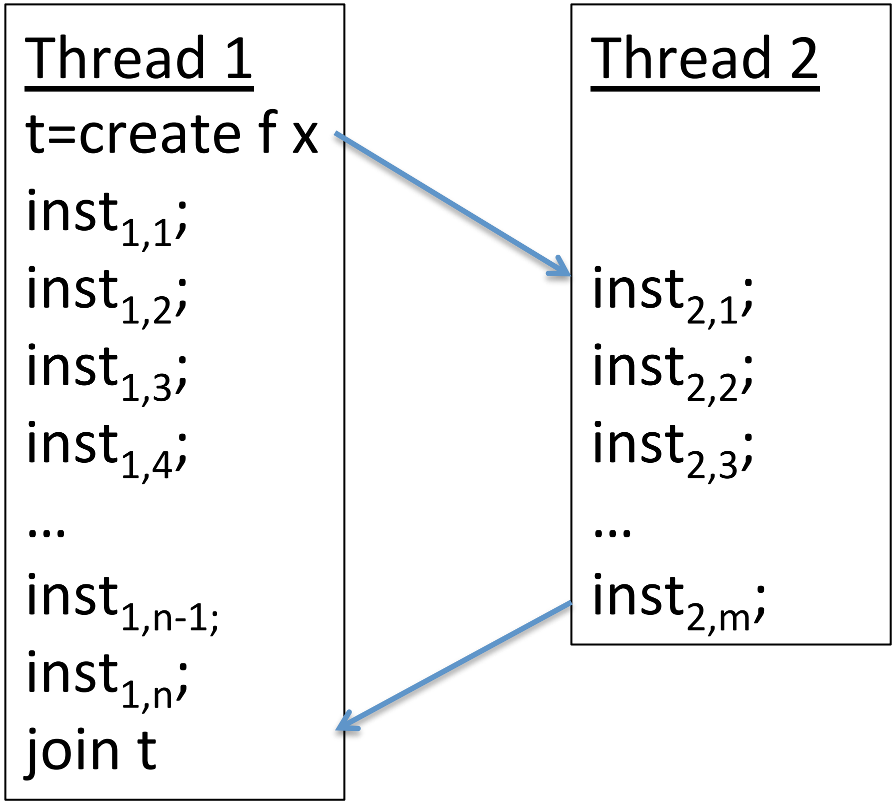
…

inst$_{2,m}$;

We know that for each thread the previous instructions must happen before the later instructions.

So for instance, inst$_{1,1}$ must happen before inst$_{1,2}$.

In Pictures



Thread 1

t=create f x

inst$_{1,1}$;

inst$_{1,2}$;

inst$_{1,3}$;

inst$_{1,4}$;

…

inst$_{1,n-1;}$

inst$_{1,n}$;

join t

Thread 2

inst$_{2,1}$;

inst$_{2,2}$;

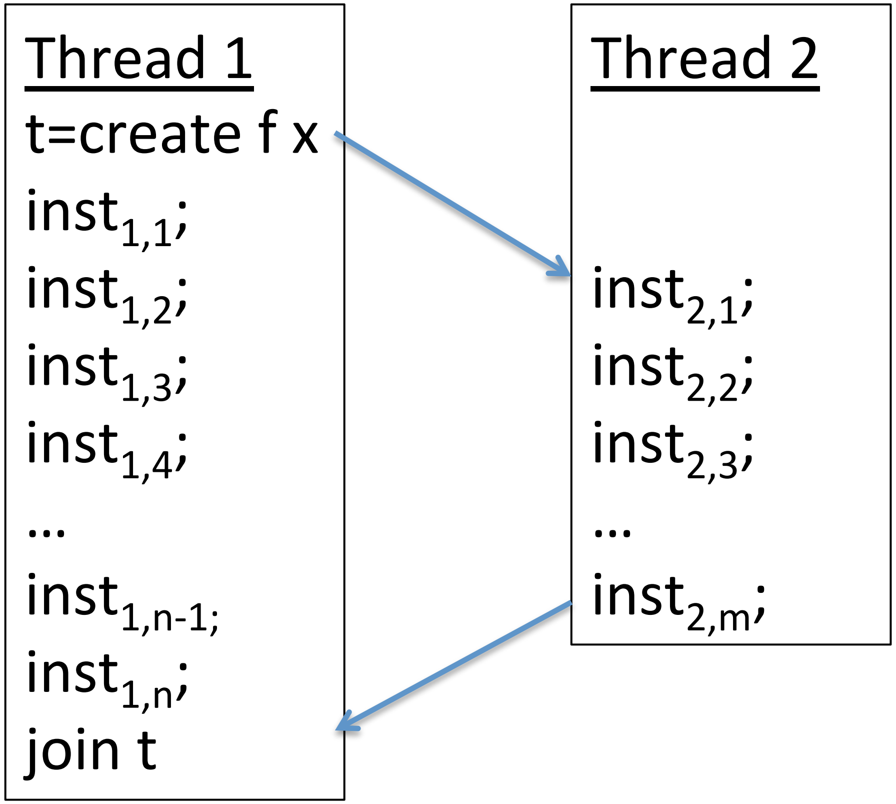
inst$_{2,3}$;

…

inst$_{2,m}$;

We also know that the fork must happen before the first instruction of the second thread.

In Pictures



Thread 1

t=create f x

$\text{inst}_{1,1}$;

$\text{inst}_{1,2}$;

$\text{inst}_{1,3}$;

$\text{inst}_{1,4}$;

…

$\text{inst}_{1,n-1;}$

$\text{inst}_{1,n}$;

join t

Thread 2

$\text{inst}_{2,1}$;

$\text{inst}_{2,2}$;

$\text{inst}_{2,3}$;

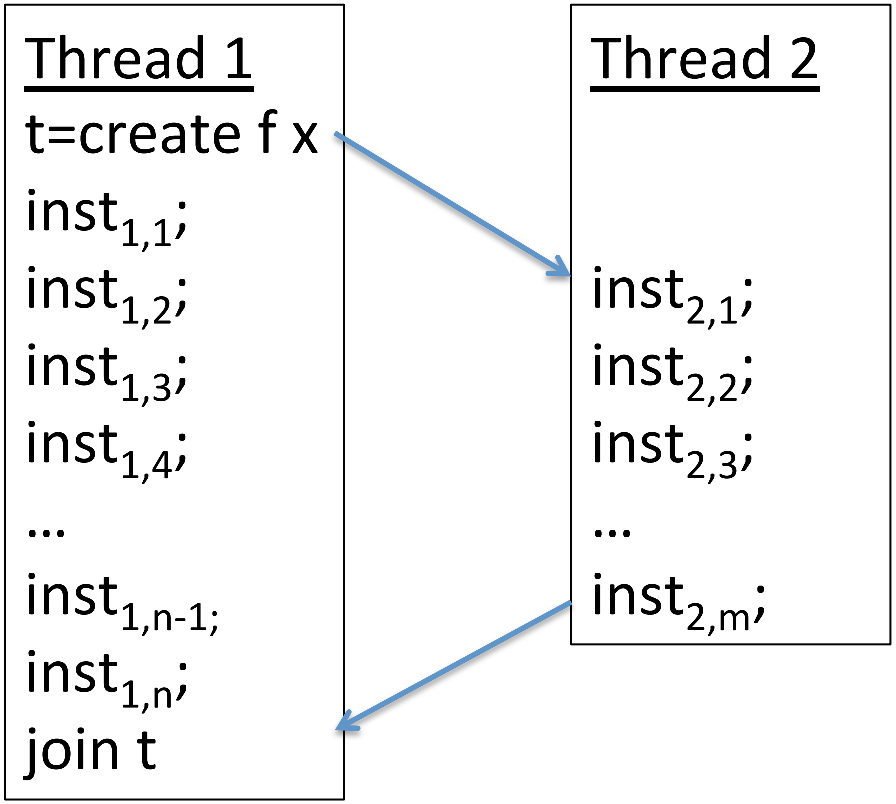
…

$\text{inst}_{2,m}$;

We also know that the fork must happen before the first instruction of the second thread.

And thanks to the join, we know that all of the instructions of the second thread must be completed before the join finishes.

In Pictures



However, in general, we do not know whether $inst_{1,i}$ executes before or after $inst_{2,j}$.

In general, *synchronization instructions like fork and join reduce the number of possible interleavings.*

Synchronization cuts down non-determinism.

In the absence of synchronization we don't know anything…

FUTURES: A PARALLEL PROGRAMMING ABSTRACTION

Futures

The fork-join pattern we just saw is so common, we’ll create an abstraction for it:

```
module type FUTURE =
sig
  type ‘a future

  (* future f x forks a thread to run f(x)
     and stores the result in a future when complete *)
  val future : (’a->’b) -> ’a -> ‘b future

  (* force f causes us to wait until the
     thread computing the future value is done
     and then returns its value. *)
  val force :’a future -> ‘a
end
```

Does that interface looks familiar .... ?

Future Implementation

```
module Future : FUTURE =
struct
  type ‘a future = {tid   : Thread.t       ;
                    value : ’a option ref }









end
```

Future Implementation

```
module Future : FUTURE =
struct
  type ‘a future = {tid   : Thread.t      ;
                    value : ’a option ref }

  let future(f:’a->’b)(x:’a) : ‘b future =
    let r = ref None in
    let t = Thread.create (fun () -> r := Some(f x)) ()
    in
    {tid=t ; value=r}

end
```

Future Implementation

```
module Future : FUTURE =
struct
  type ‘a future = {tid   : Thread.t      ;
                    value : ’a option ref }

  let future(f:’a->’b)(x:’a) : ‘b future =
    let r = ref None in
    let t = Thread.create (fun () -> r := Some(f x)) ()
    in
    {tid=t ; value=r}

  let force (f:’a future) : ‘a =
    Thread.join f.tid ;
    match !(f.value) with
    | Some v -> v
    | None -> failwith “impossible!”
end
```

Now using Futures

```
let x = future f () in
let y = g () in
let v = force x in
(* compute with v and y *)
```

Back to the Futures

```
module type FUTURE =
sig
  type ‘a future

  val future : (’a->’b) -> ’a -> ‘b future
  val force :’a future -> ‘a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + v
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith “impossible”
```

Back to the Futures

```
module type FUTURE =
sig
  type ‘a future

  val future : (’a->’b) -> ’a -> ‘b future
  val force :’a future -> ‘a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + v
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith “impossible”
```

what happens if we delete these lines?

Back to the Futures

```
module type FUTURE =
sig
  type ‘a future

  val future : (’a->’b) -> ’a -> ‘b future
  val force :’a future -> ‘a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
            (fun _ -> r := Some(f ()))
            ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith “impossible”
```

what happens if we use x and forget to force?

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

Moral: Futures + typing ensure entire categories of errors can't happen -- you protect yourself from your own stupidity

Back to the Futures

```
module type FUTURE =
sig
  type ‘a future

  val future : (’a->’b) -> ’a -> ‘b future
  val force :’a future -> ‘a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let v = force x in
let y = g () in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
Thread.join t ;
let y = g() in
match !r with
    Some v -> y + v
  | None -> failwith “impossible”
```

what happens if you relocate force, join?

Back to the Futures

```
module type FUTURE =
sig
  type ‘a future

  val future : (’a->’b) -> ’a -> ‘b future
  val force :’a future -> ‘a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let v = force x in
let y = g () in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
Thread.join t ;
let y = g() in
match !r with
    Some v -> y + v
  | None -> failwith “impossible”
```

Moral: Futures are not a universal savior

An Example: Mergesort on Arrays

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) : 'a array =
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1 = msort start half in
             let a2 = msort (start + half)
                            (len - half) in
               merge a1 a2

  and merge (a1:'a array) (a2:'a array) : 'a array =
```

An Example: Mergesort on Arrays

```
let mergesort (cmp:'a->'a->int) (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) : 'a array =
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1 = msort start half in
             let a2 = msort (start + half) (len - half) in
               merge a1 a2
  and merge (a1:'a array) (a2:'a array) : 'a array =
    let a = Array.make (Array.length a1 + Array.length a2) a1.(0) in
    let rec loop i j k =
      match i < Array.length a1, j < Array.length a2 with
        | true, true -> if cmp a1.(i) a2.(j) <= 0 then
                          (a.(k) <- a1.(i) ; loop (i+1) j (k+1))
                        else (a.(k) <- a2.(j) ; loop i (j+1) (k+1))
        | true, false -> a.(k) <- a1.(i) ; loop (i+1) j (k+1)
        | false, true -> a.(k) <- a2.(j) ; loop i (j+1) (k+1)
        | false, false -> ()
    in
      loop 0 0 0 ; a
  in
    msort 0 (Array.length arr)
```

An Example: Mergesort on Arrays

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) :
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1 = msort start half in
             let a2 = msort (start + half)
                            (len - half) in
               merge a1 a2

  and merge (a1:'a array) (a2:'a array) : 'a array =
```

Opportunity for parallelization

Making Mergesort Parallel

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) : 'a array =
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1_f =
               Future.future (msort start) half
             in
             let a2 = msort (start + half)
                            (len - half) in
               merge (Future.force a1_f) a2

  and merge (a1:'a array) (a2:'a array) : 'a array =
```

Divide-and-Conquer

This is an instance of a basic *divide-and-conquer* pattern in parallel programming

- take the problem to be solved and divide it in half
- fork a thread to solve the first half
- simultaneously solve the second half
- synchronize with the thread we forked to get its results
- combine the two solution halves into a solution for the whole problem.

Warning: the fact that we only had to rewrite 2 lines of code for mergesort made the parallelization transformation look deceptively easy

- we also had to verify that any two threads did not touch overlapping portions of the array -- if they did we would have to again worry about scheduling non-determinism

Caveats

There is some overhead for creating a thread.

- On a uni-processor, parallel code will run *slower* than the sequential code.

Even on a multi-processor, we probably do *not always* want to fork a thread

- when the sub-array is small, faster to sort it than to fork a thread to sort it.
  - similar to using insertion sort when arrays are small vs. quicksort
- this is known as a *granularity problem*
  - more parallelism than we can effectively take advantage of.

Caveats

In a good implementation of futures, a compiler and run-time system might look to see whether the cost of doing the fork is justified by the amount of work that will be done. Today, it's up to you to figure this out… ☹

- typically, use parallel divide-and-conquer until
  - (a) we have generated *at least* as many threads as there are processors
    - often *more threads* than processors because different jobs take different amounts of time to complete and we would like to keep all processors busy
  - (b) the sub-arrays have gotten small enough that it's not worth forking.

We're not going to worry about these performance-tuning details but rather focus on the distinctions between *parallel* and *sequential algorithms*.

Another Example

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {left  : 'a tree ;
               value : 'a      ;
               right : 'a tree }

let rec fold (f:'a -> 'b -> 'b -> 'b) (u:'b)
             (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
     f n.value (fold f u n.left) (fold f u n.right)

let sum (t:int tree) = fold (+) 0 t
```

Another Example

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {left  : 'a tree ;
               value : 'a      ;
               right : 'a tree }

let rec pfold (f:'a -> 'b -> 'b -> 'b) (u:'b)
              (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
     let l_f = Future.future (pfold f u) n.left in
     let r = pfold f u n.right in
     f n.value (Future.force l_f) r

let sum (t:int tree) = pfold (+) 0 t
```

Note

- If the tree is imbalanced, then we're not going to get the same speedup as if it's balanced.
- Consider the degenerate case of a list.
  - The forked child will terminate without doing any useful work.
  - So the parent is going to have to do all that work.
  - Pure overhead… ☹
- In general, lists are a horrible data structure for parallelism.
  - *we can't cut the list in half in constant time*
  - for arrays and trees, we can do that (assuming the tree is balanced.)

Side Effects?

```
type 'a tree = Leaf | Node of 'a node
and 'a node = { left  : 'a tree ;
                value : 'a      ;
                right : 'a tree }

let rec pfold (f:'a -> 'b -> 'b -> 'b) (u:'b)
              (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
     let l_f = Future.future (pfold f u) n.left in
     let r = pfold f u n.right in
     f n.value (Future.force l_f) r

let print (t:int tree) =
  pfold (fun n _ _ -> Printf.print “%d\n” n) ()
```

Huge Point

If code is purely functional, then it never matters in what order it is run.

If f () and g () are pure then all of the following are equivalent:

```
let x = f() in
let y = g() in
e
```

```
let x_f = future f () in
let y   = g ()        in
let x   = force x_f   in
e
```

```
let y = g () in
let x = f () in
e
```

```
let y_g = future g () in
let x   = f ()        in
let y   = force y_g   in
e
```

As soon as we introduce *side-effects*, the order starts to matter.

- This is why, IMHO, *imperative* languages where even the simplest of program phrases involves a side effect, are doomed.
- Of course, we've been saying this for 30 years!
- See J. Backus's Turing Award paper, *"Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs."*

  http://www.cs.cmu.edu/~crary/819-f09/Backus78.pdf

END