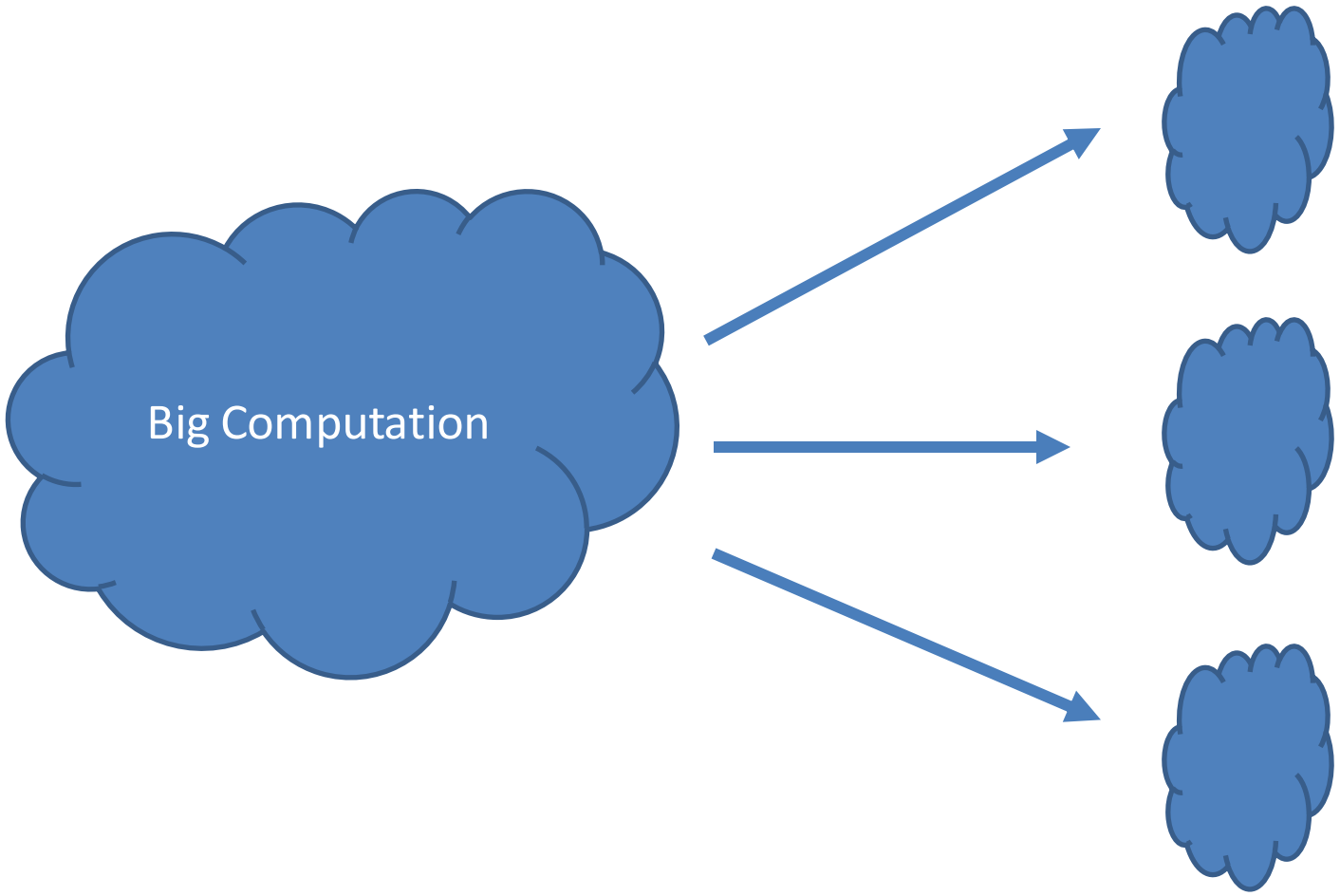
Parallelism I

COS 326

David Walker

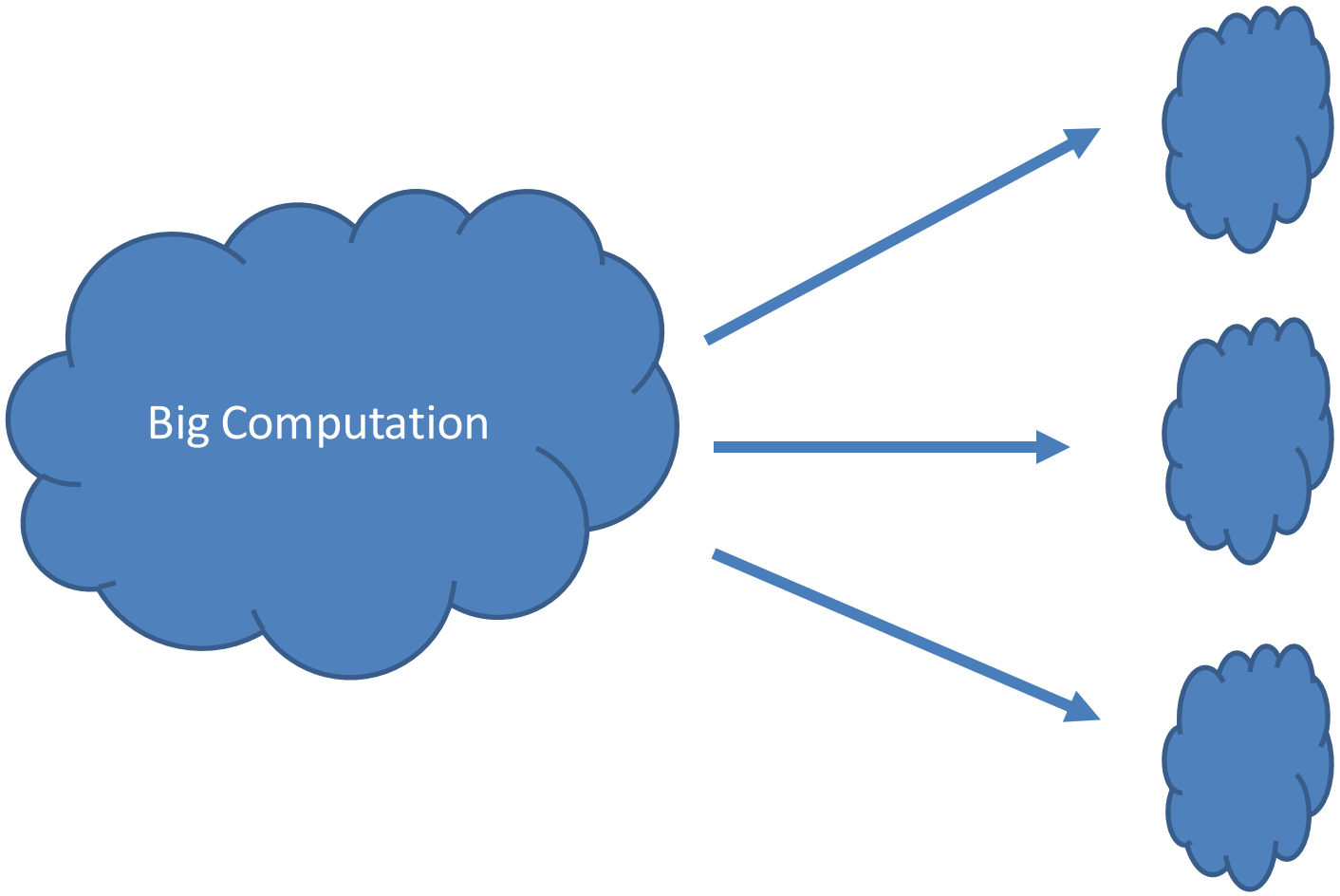
Princeton University

Parallelism



If you can split up a big computation into small chunks and do each chunk the same time, you won't save work (energy), but you will save time

Parallelism



Hardware manufacturers have been trying to help us do that for decades

Moore's Law

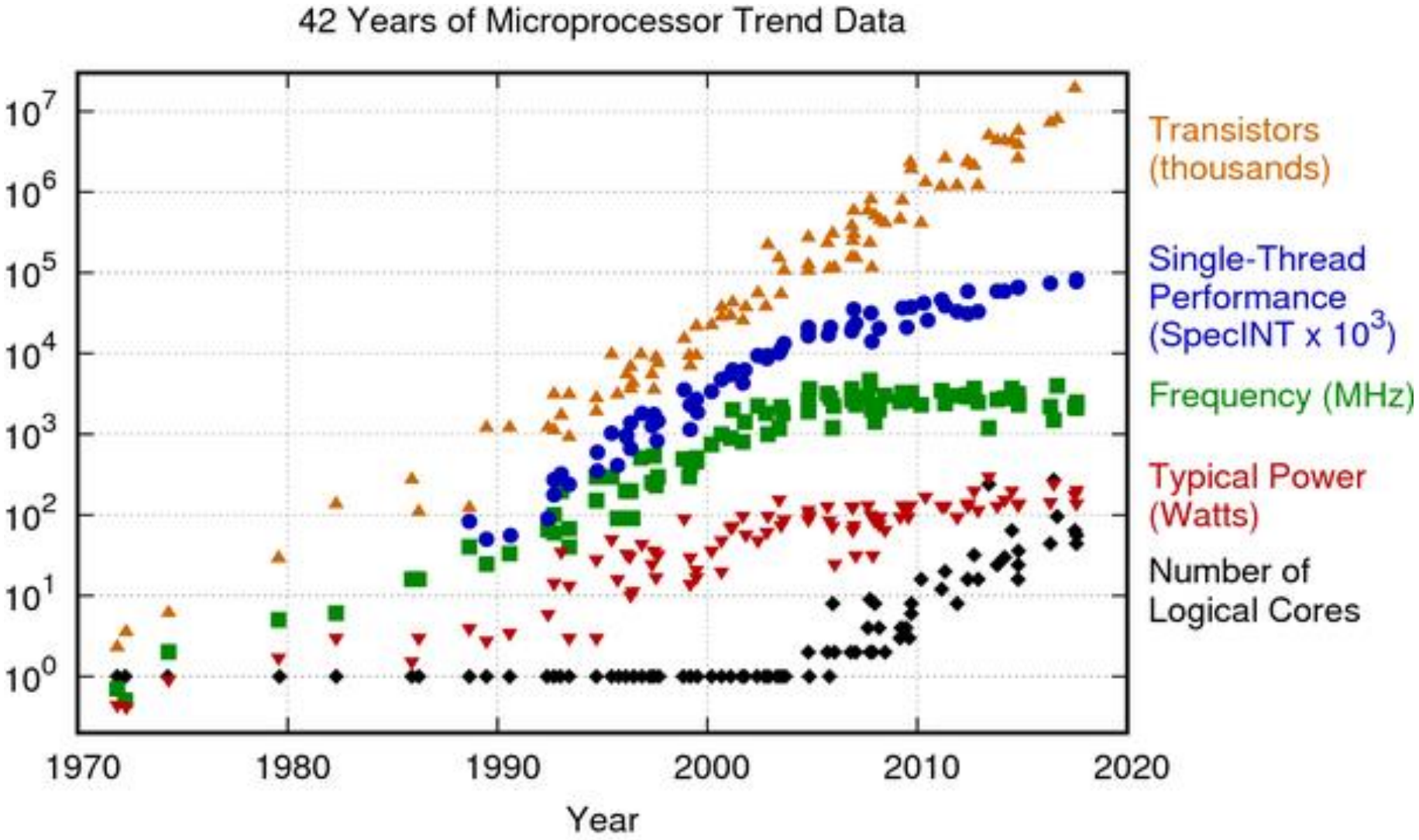
Moore's Law: *The number of transistors you can put on a computer chip doubles (approximately) every couple of years.*

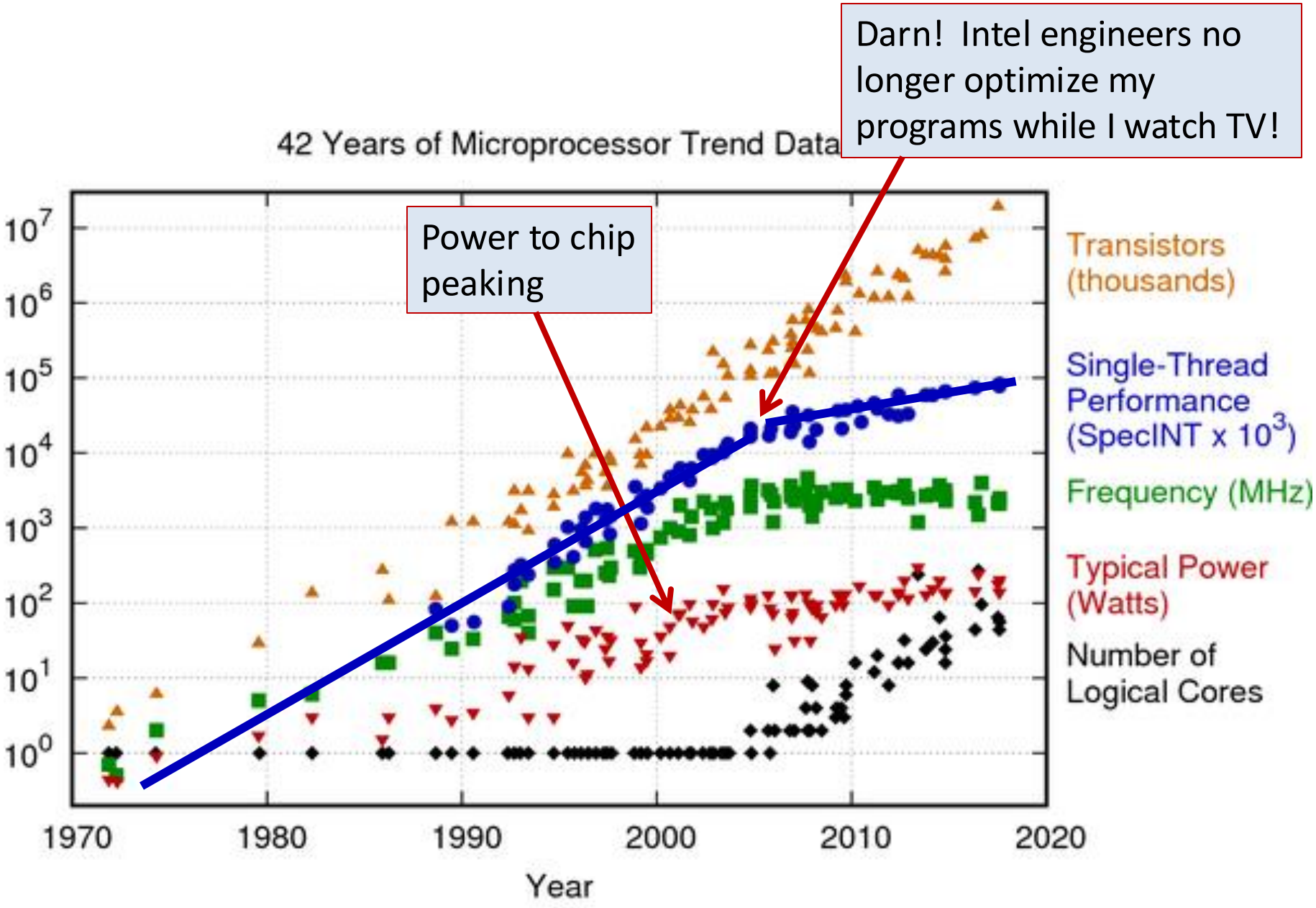
Consequence for most of the history of computing:

- *All programs double in speed every couple of years.*
- Extra transistors doubled the number of instructions executed per time unit

Consequence for application writers:

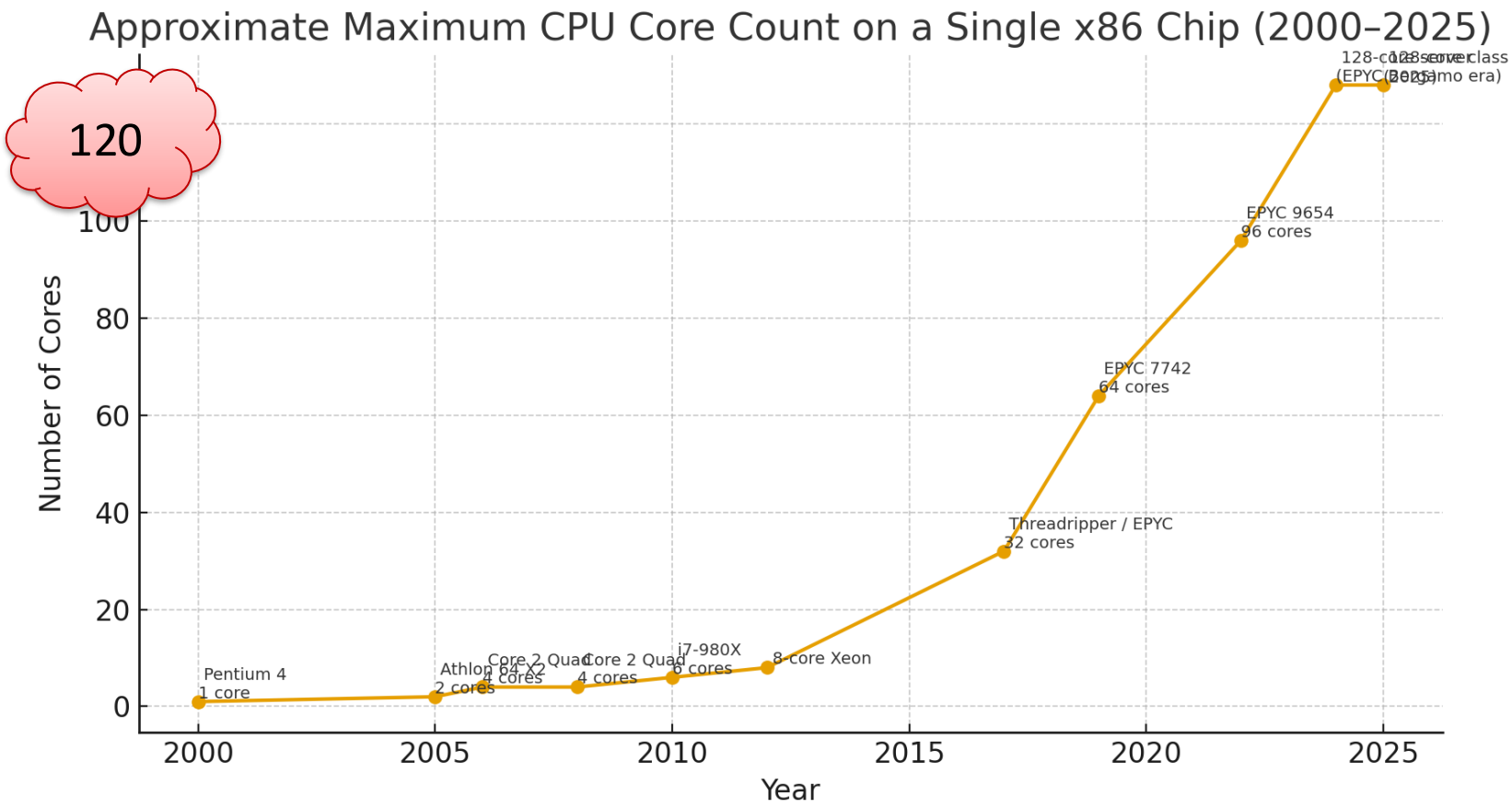
- *Watch TV for a while and your programs optimize themselves*!
- New applications thought impossible became possible because of increased computational power


42 Years of Microprocessor Trend Data
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1970
1980
1990
2000
2010
2020
Year
Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp


Darn! Intel engineers no longer optimize my programs while I watch TV!
42 Years of Microprocessor Trend Data
Power to chip peaking
Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1970
1980
1990
2000
2010
2020
Year
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

So…

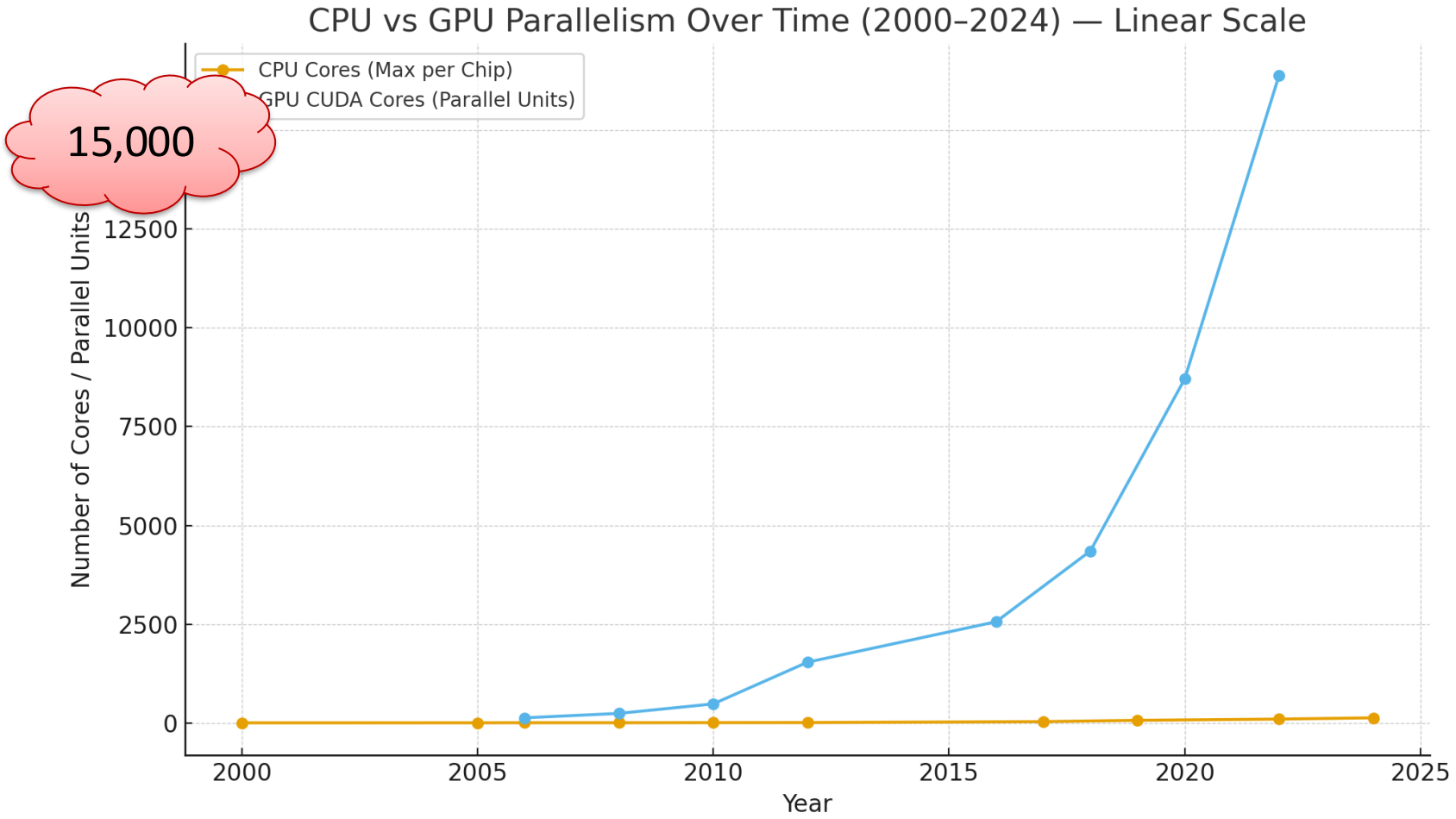
Instead of making your CPU go faster, manufacturers have been to packing more CPUs onto a chip.



* produced by ChatGPT 5.1 in 11/2025

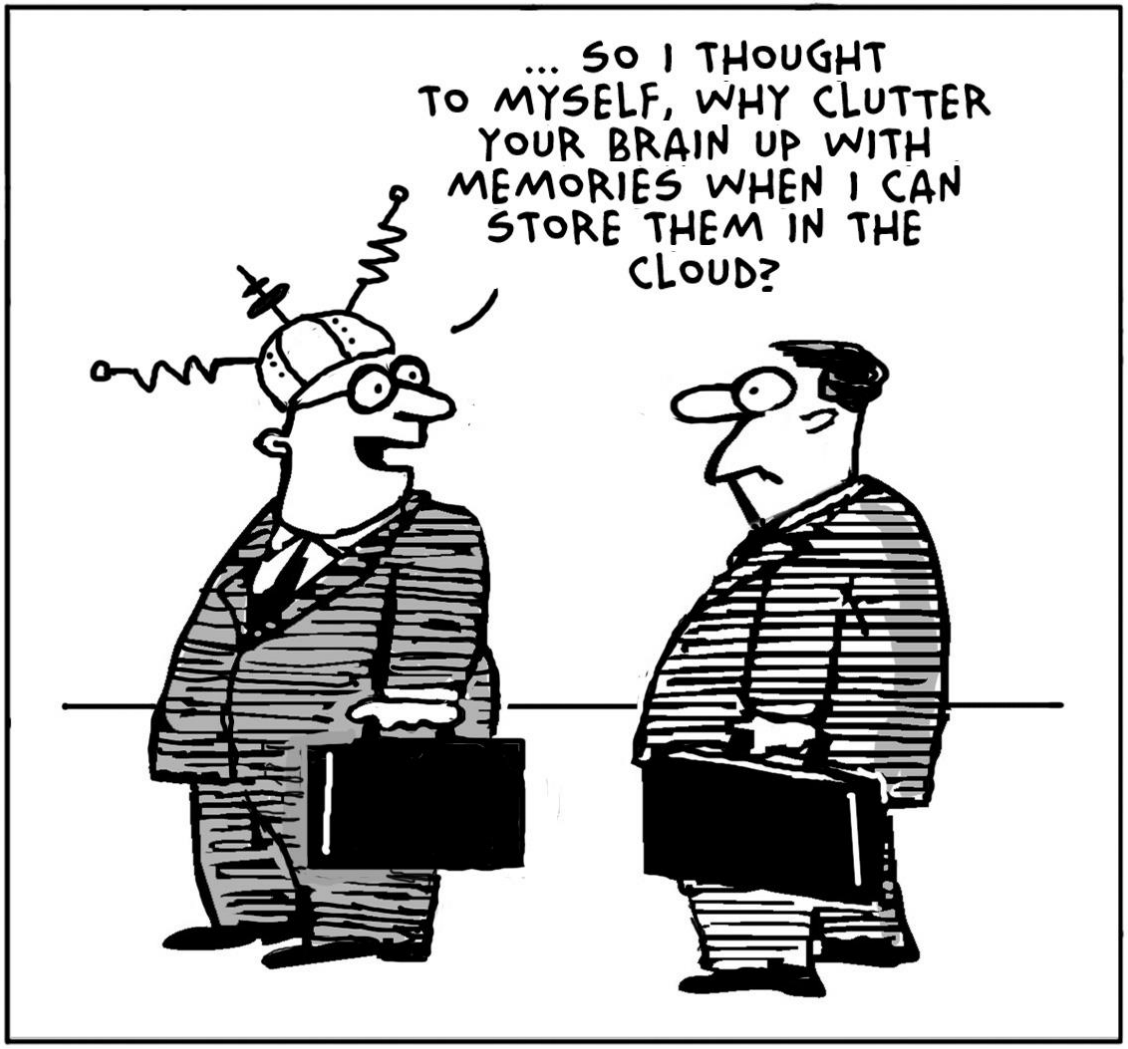
So…

Instead of making your CPU go faster, manufacturers have been to packing more CPUs onto a chip.

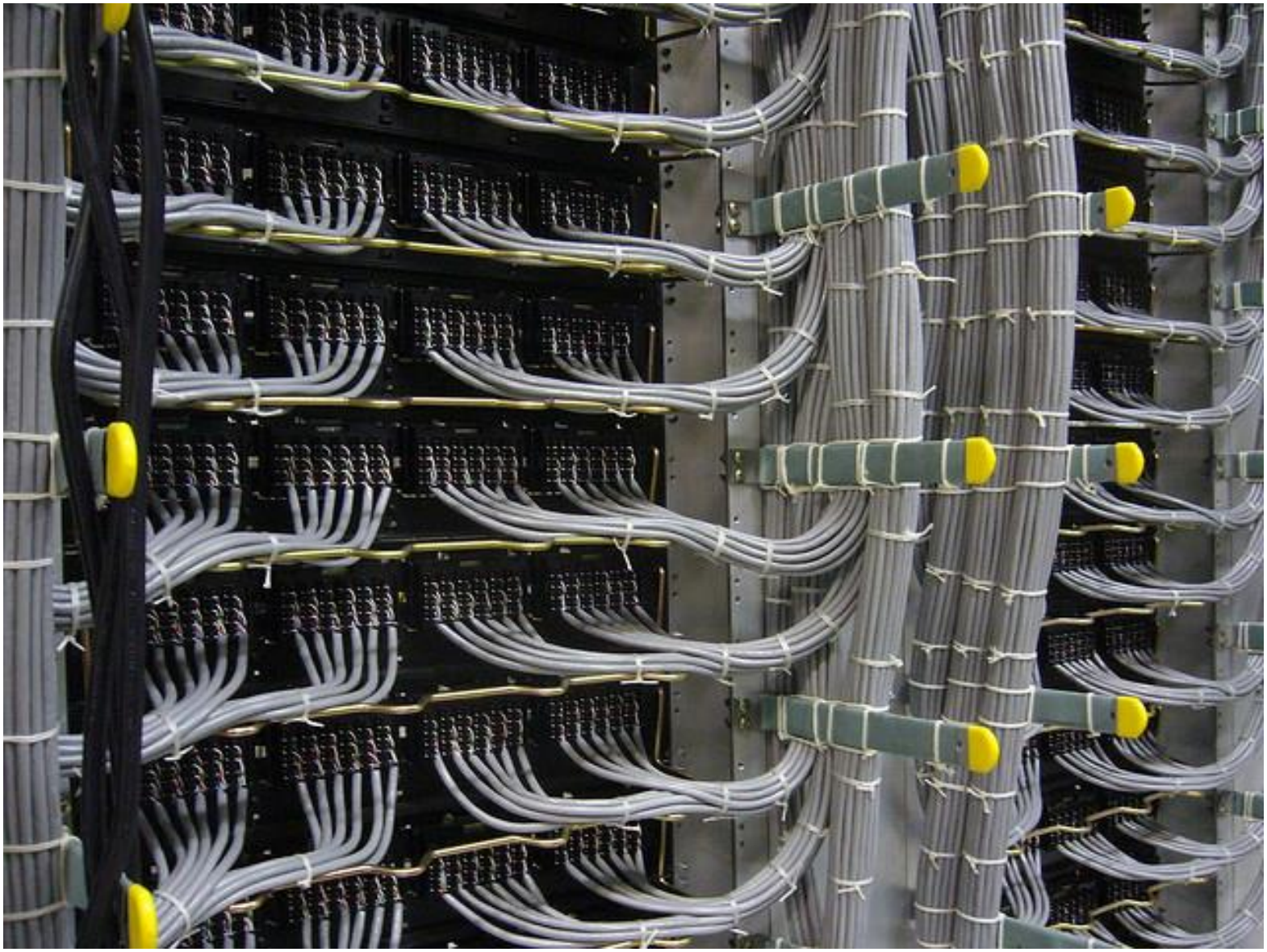


* produced by ChatGPT 5.1 in 11/2025

But there's more: Data Centers



Data Centers: *Lots* of Connected Computers!



Data Centers: Lots of Connected Computers

Computer containers for plug-and-play parallelism:

80,000 servers ?
x 20 cores/server ?
= 1.6 million cores?

How many servers?
Trade secret!
How does Microsoft estimate how many servers Google has?

Now that we have all this parallel hardware:

- many cores, on
- many computers, in
- many data centers

How do we program it effectively?

Unfortunately

Many parallel programming models:

- *Introduce nondeterminism*
  - 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
- *Are nonmodular*
  - module A implicitly influences the outcomes of module B
- *Introduce new classes of errors*
  - race conditions, deadlocks
- *Introduce new performance/scalability problems*

Fortunately

You are taking a class on functional programming ...

There are some functional abstractions for parallel programming that have all the performance but none of the downsides!

Fortunately

You are taking a class on functional programming ...

What we want: Parallel performance with sequential semantics

This gives us determinism

No worries about race conditions

But we still have to reason about the cost of parallel functional programs and that remains hard

WHY CONCURRENT PROGRAMMING IS FUNDAMENTALLY HARD

It's all about invariants

Assume f is a pure function. Is x > 0 ?

```
let x =
  if f y > 0 then
    f y
  else
    1
```

yes. If f is a function, every call to f with the same argument returns the same answer.

Reasoning about programs is (often) highly modular.

Invariants established persist forever.

It's all about invariants

Assume f is a function in a sequential application that may access mutable state. Is x > 0 ?

```
let x =
  if f y > 0 then
    f y
  else
    1
```

We don't know!

It's all about invariants

Assume f is a function in a sequential application that may access mutable state. Is x > 0 ?

```
let x =
  if f y > 0 then
    f y
  else
    1
```

But we could look at f to find out definitively. eg:

```
let r = ref 0

let f y = !r + y

let g y = (r := !r - y)
```

other functions don't matter since they weren't called

It's all about invariants

Assume f is a function in a concurrent application that may access mutable state. Is x > 0 ?

```
let x =
  if f y > 0 then
    f y
  else
    1
```

Thread 1

```
g 1;
g 2;
g 3;
g 4;
…
```

Thread 2

```
let r = ref 0

let f y = !r + y

let g y = (r := !r - y)
```

suddenly, all other functions that may reference the same state matter. And it can hard to pin down which functions might access which bits of state at which times

It's all about invariants

Assume f is a function in a concurrent application that may access mutable state. Is x > 0 ?

```
let x =
  if f y > 0 then
    f y
  else
    1
```

Thread 1

```
g 1;
g 2;
g 3;
g 4;
…
```

Thread 2

```
let r = ref 0

let f y = !r + y

let g y = (r := !r - y)
```

And you may have to reason about all interleavings of all other functions with the function you wrote.

(It may actually be worse than that)

Our Goal

To study **functional abstractions** that provide the performance but none of the semantic non-determinism that makes reasoning about concurrent programs with mutable state so hard.

THREADS:
A CONVENTIONAL PARALLEL PROGRAMMING MODEL

Threads: A Warning

Concurrent Threads with Locks: the classic shoot-yourself-in-the-foot concurrent programming model

- almost all programming languages will have a threads library
  - OCaml in particular!
- you need to know where (some of) the pitfalls are
- the assembly language of concurrent programming paradigms
  - build higher-level programming models on top of threads

https://ocaml.org/manual/5.4/api/Thread.html

Threads

A thread is an abstraction of a processor.

- programmer (or compiler) decides that some work can be done in parallel with some other work, e.g., our program is:

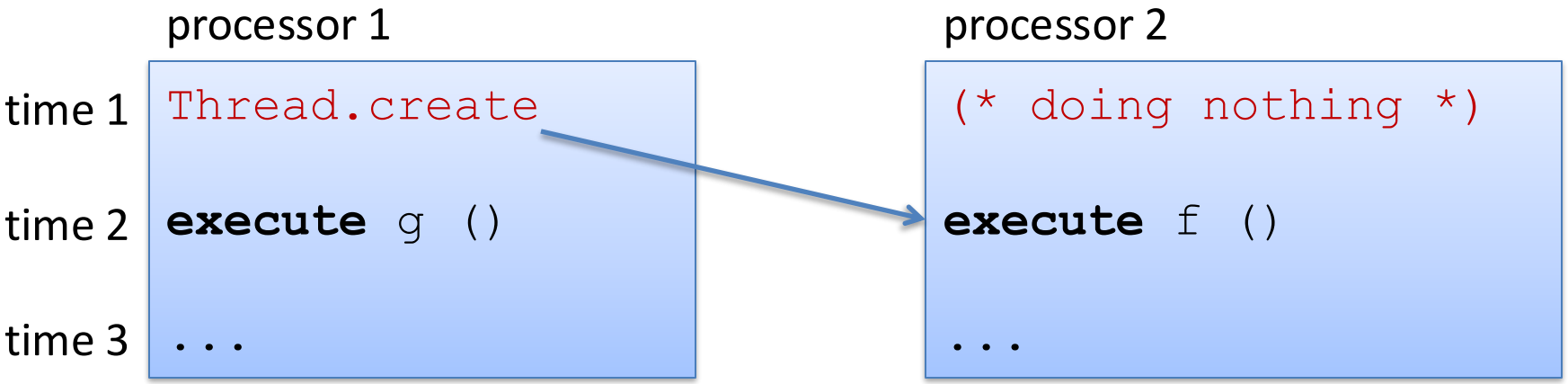
```
let _ = compute_big_thing() in
let y = compute_other_big_thing() in
...
```

- 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
 ...
```



Of Course…

Suppose you have 2 available cores and you fork 4 threads.

In a typical multi-threaded system,

- OS provides *the illusion* that there are infinite processors.
  - not really: each thread consumes space, so if you fork too many threads the process will die.
- OS *time-multiplexes* the threads across the available processors.
  - every few ms, the OS stops the current thread on a processor, and switches to another thread.

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 -> failwith "impossible"
```

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.
- CPU could either be 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 *)
```

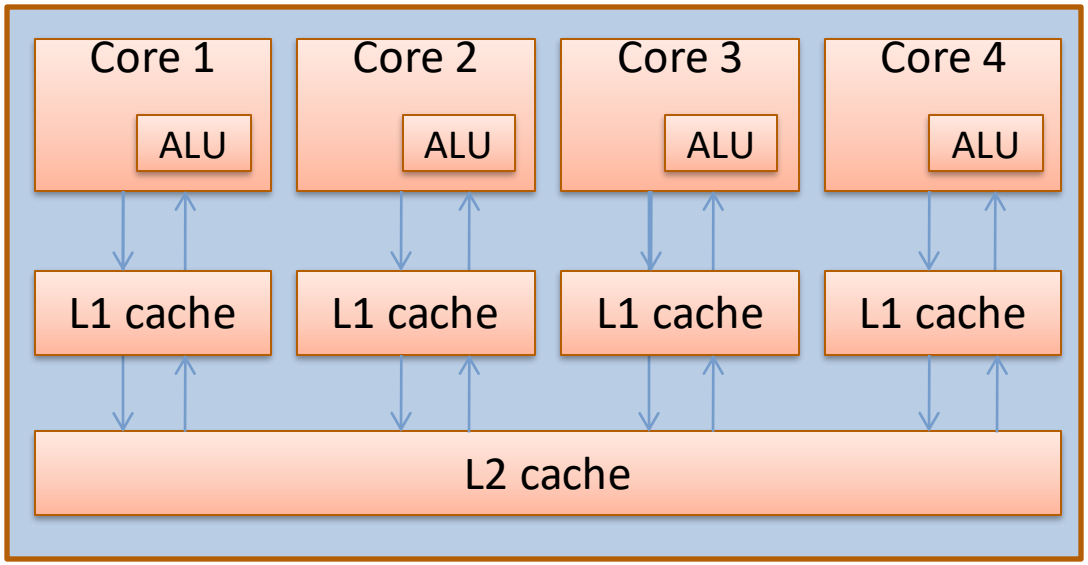
Second, an operation like r := Some v may not be *atomic*.

- 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.

Real Machines

Today's multicore processors don't even have *sequentially consistent* memory models.

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 course.

Solution: Synchronization Primitives

All systems for parallel programming have synchronization primitives.

Examples:

- locks
- semaphores
- compare-and-swap
- synchronized methods

Today:

- fork-join parallelism
  - join is the synchronization primitive

Recall our 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 with v and y *)
  | None -> failwith "impossible"
```

`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 *)
  | None -> failwith "impossible"
```

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

The happens-before relation

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

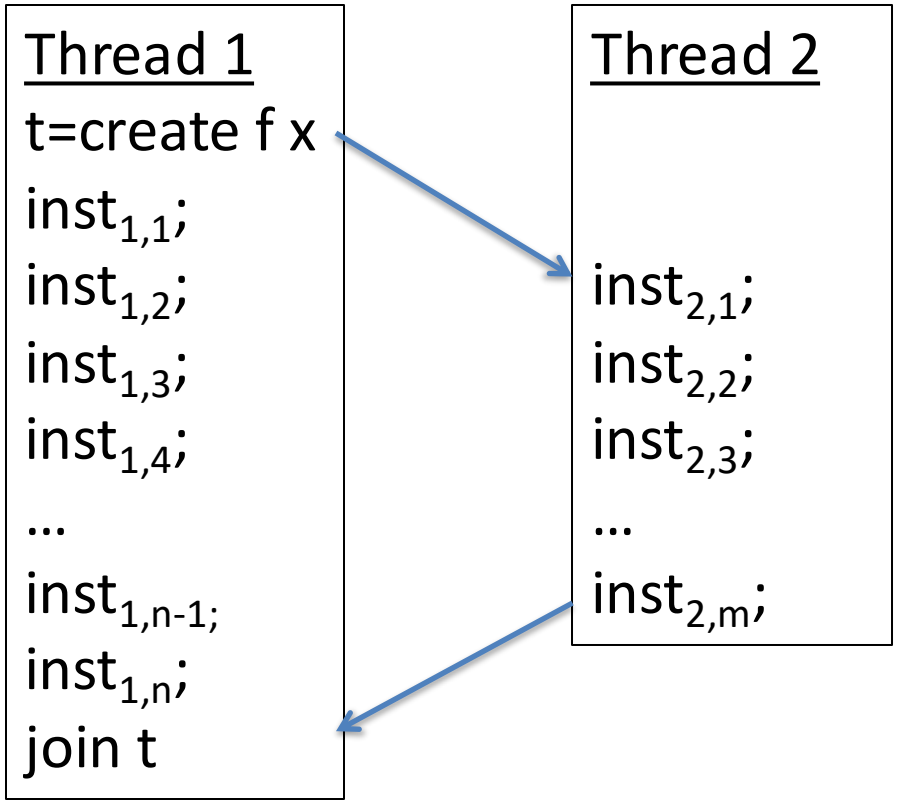
Rule 2: Given a program:

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

we know that (f x) happens before e.

Rule 3: Transitivity

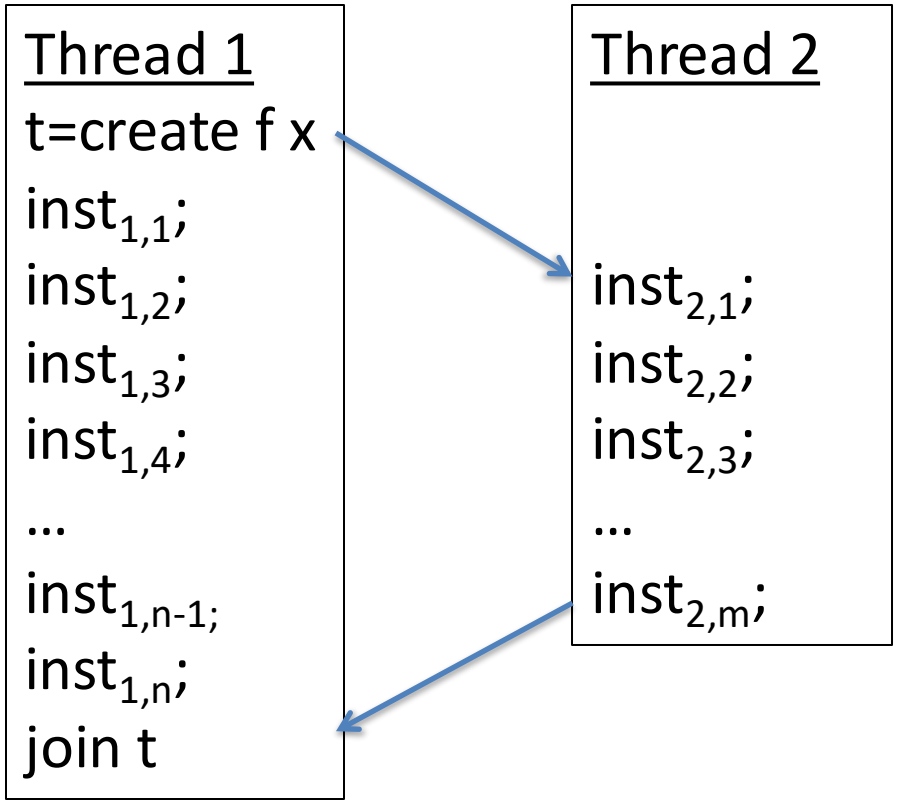
In Pictures



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

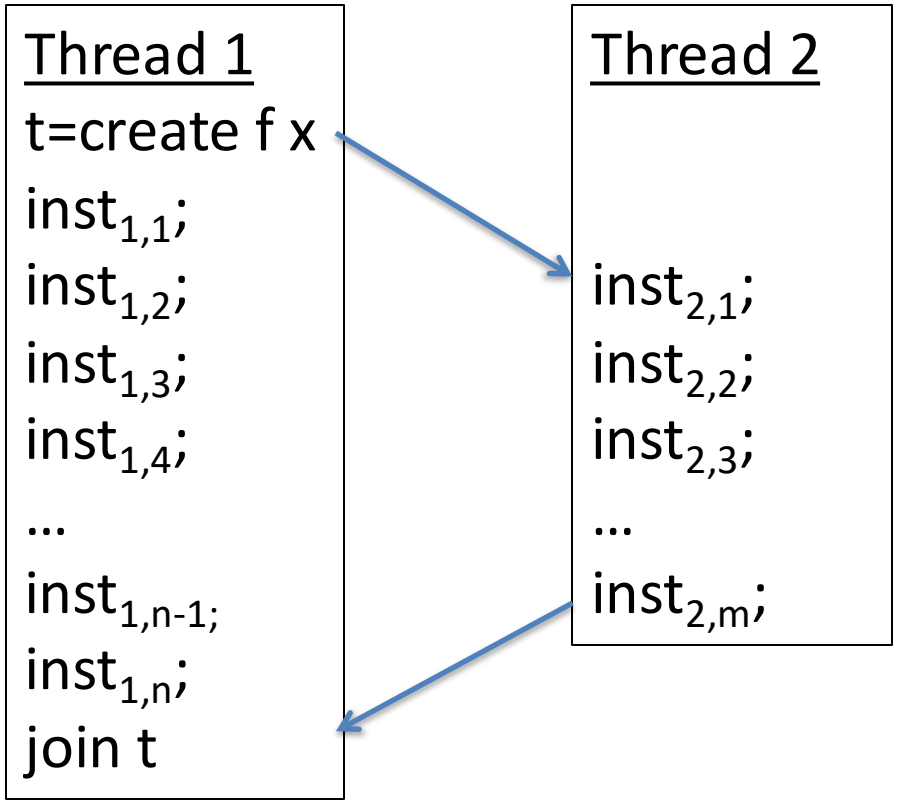
So for instance, $inst_{1,1}$ must happen before $inst_{1,2}$.

In Pictures



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

In Pictures



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.

Fork-Join

Fork-Join parallelism cuts down on the number of interleavings

Reduces non-determinism

A very useful pair of primitives

But we still do have to think about all those interleavings and how they interact with mutable references. It's very, very hard to write and debug programs in this model.

Can we avoid doing such reasoning altogether?

FUTURES: A PARALLEL PROGRAMMING ABSTRACTION

Futures

```
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 this lines? (Forgetting to synchronize)

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
```

what happens if we use x and forget to force?

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 + 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 + v
```

what happens if you relocate force, join?

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"
```

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

```
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
```

An 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
```

Performance Notes!

Creating a thread involves a lot of systems-level overhead.

You have to do a lot of work in parallel with other threads to make up for that overhead.

So the code we wrote will be slower than sequential code unless the functions we are mapping do a huge amount of work (thousands of instructions at least).

Moreover, the tree had better be balanced.

But this is really not a course about program optimization!

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, the results get from using futures is exactly the same as the result you would get from writing a sequential program

The following are equivalent when f and g are pure functions

```
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*, all bets are off!

SUMMARY

Summary: Threads & Futures

Reasoning about unstructured concurrent programs is very hard.

Futures + pure functions: an abstraction that makes reasoning about the correctness of parallel programs easy.

Performance analysis is still hard.

Good programmers write simple code.

Good programmers use abstractions like futures.

Good programmers engineer interfaces that make it harder for clients to make mistakes.