Parallelism and Concurrency

COS 326

David Walker

Princeton University

Parallelism

- What is it?
- Today's technology trends.
- Then:
  - Why is it so much harder to program?
    - (Is it actually so much harder to program?)
  - Some preliminary linguistic constructs
    - thread creation
    - our first parallel functional abstraction: futures

PARALLELISM: WHAT IS IT?

Parallelism

- What is it?
  - doing many things at the same time instead of sequentially (one-after-the-other).

Flavors of Parallelism

Data Parallelism

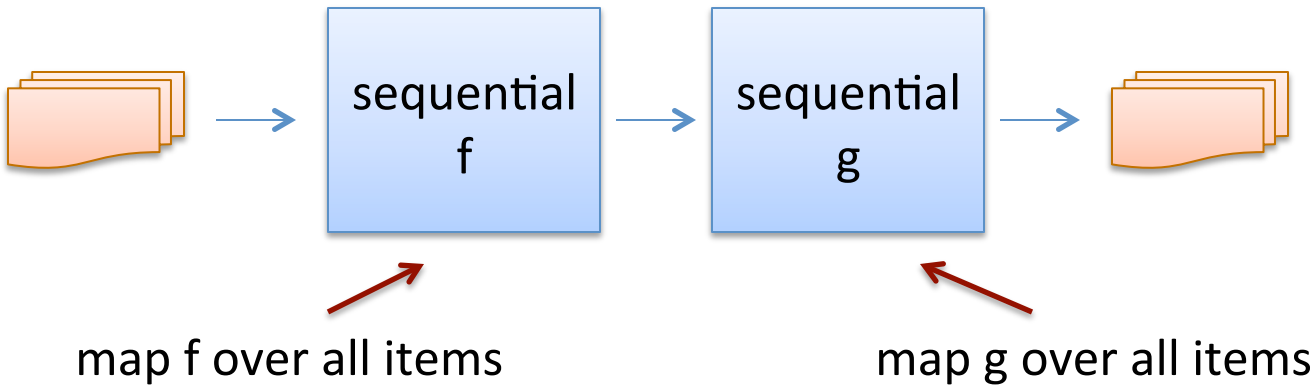
- same computation being performed on a *collection* of independent items
- e.g., adding two vectors of numbers

Task Parallelism

- different computations/programs running at the same time
- e.g., running web server and database

Pipeline Parallelism

- assembly line:



Parallelism vs. Concurrency

Parallelism: performs many tasks *simultaneously*

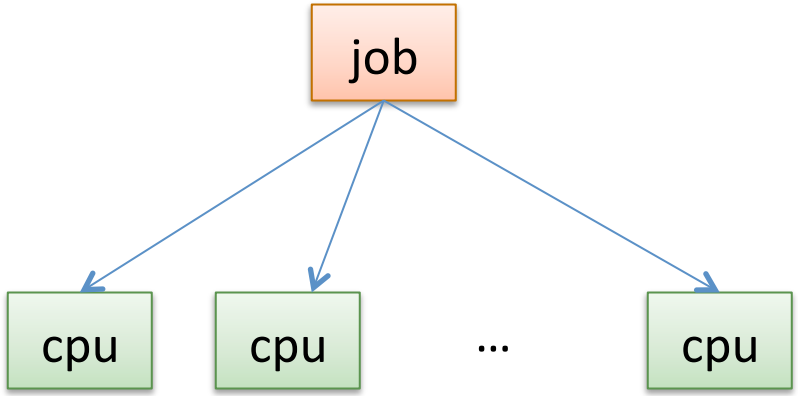
- purpose: improves throughput
- mechanism:
  - many independent computing devices
  - decrease run time of program by utilizing multiple cores or computers
- eg: running your web crawler on a cluster versus one machine.

Concurrency: mediates multi-party access to shared resources

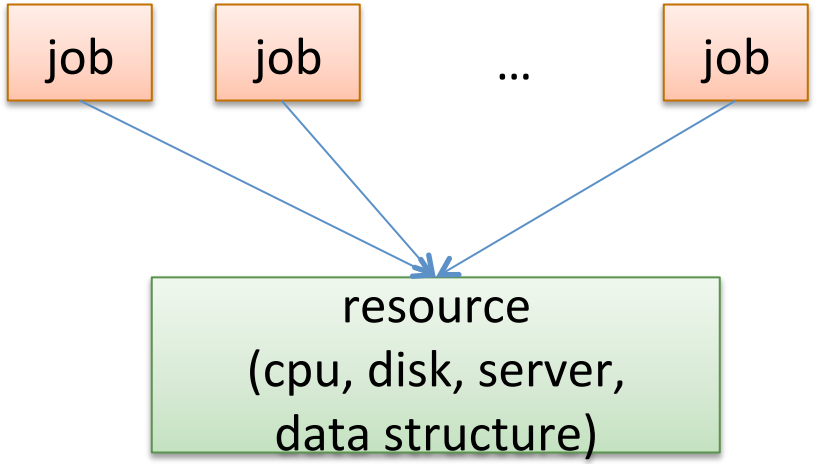
- purpose: decrease response time
- mechanism:
  - switch between different threads of control
  - work on one thread when it can make useful progress; when it can't, suspend it and work on another thread
- eg: running your clock, editor, chat at the same time on a single CPU.
  - OS gives each of these programs a small time-slice (~10msec)
  - often *slows* throughput due to cost of switching contexts
- eg: don't block while waiting for I/O device to respond, but let another thread do useful CPU computation

Parallelism vs. Concurrency

Parallelism:
perform several independent tasks simultaneously



Concurrency:
mediate/multiplex access to shared resource



many efficient programs use some parallelism and some concurrency

UNDERSTANDING TECHNOLOGY TRENDS

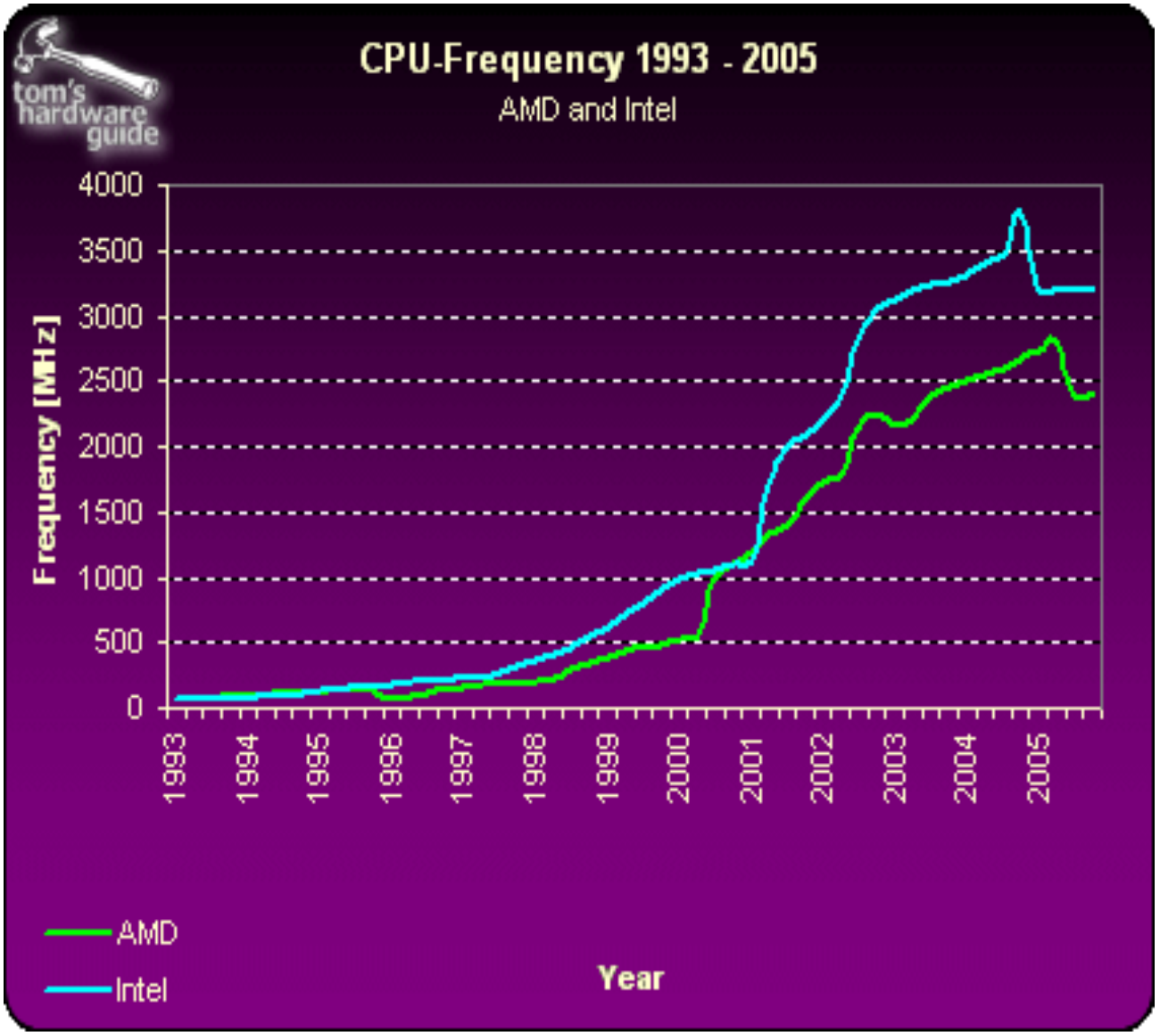
Moore's Law

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

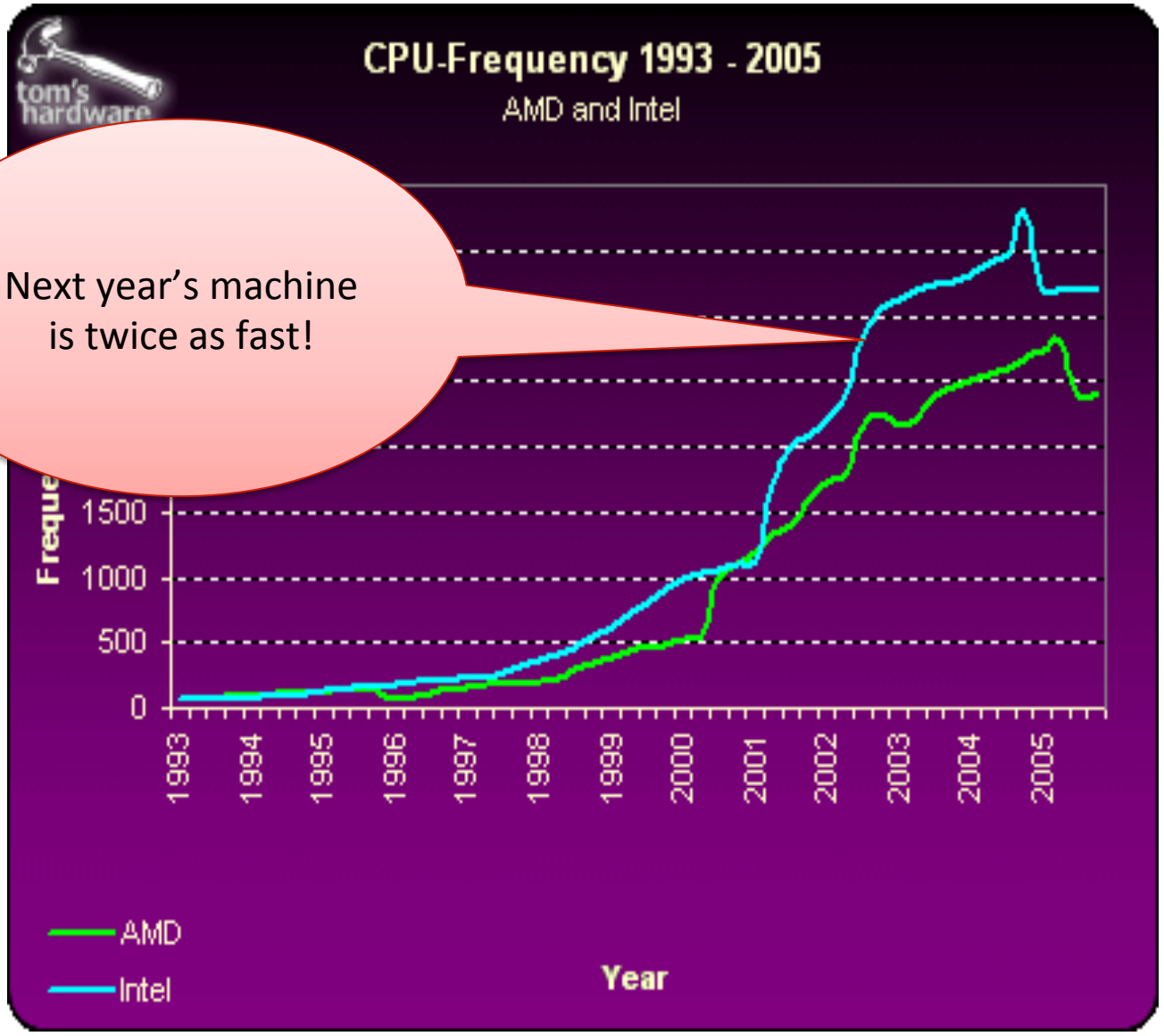
- Consequence for most of the history of computing: *All programs double in speed every couple of years.*
  - Why? Hardware designers are wicked smart.
  - They have been able to use those extra transistors to (for example) double the number of instructions executed per time unit, thereby processing speed of programs

- Consequence for application writers:
  - *watch TV for a while and your programs optimize themselves*!
  - perhaps more importantly: new applications thought impossible became possible because of increased computational power

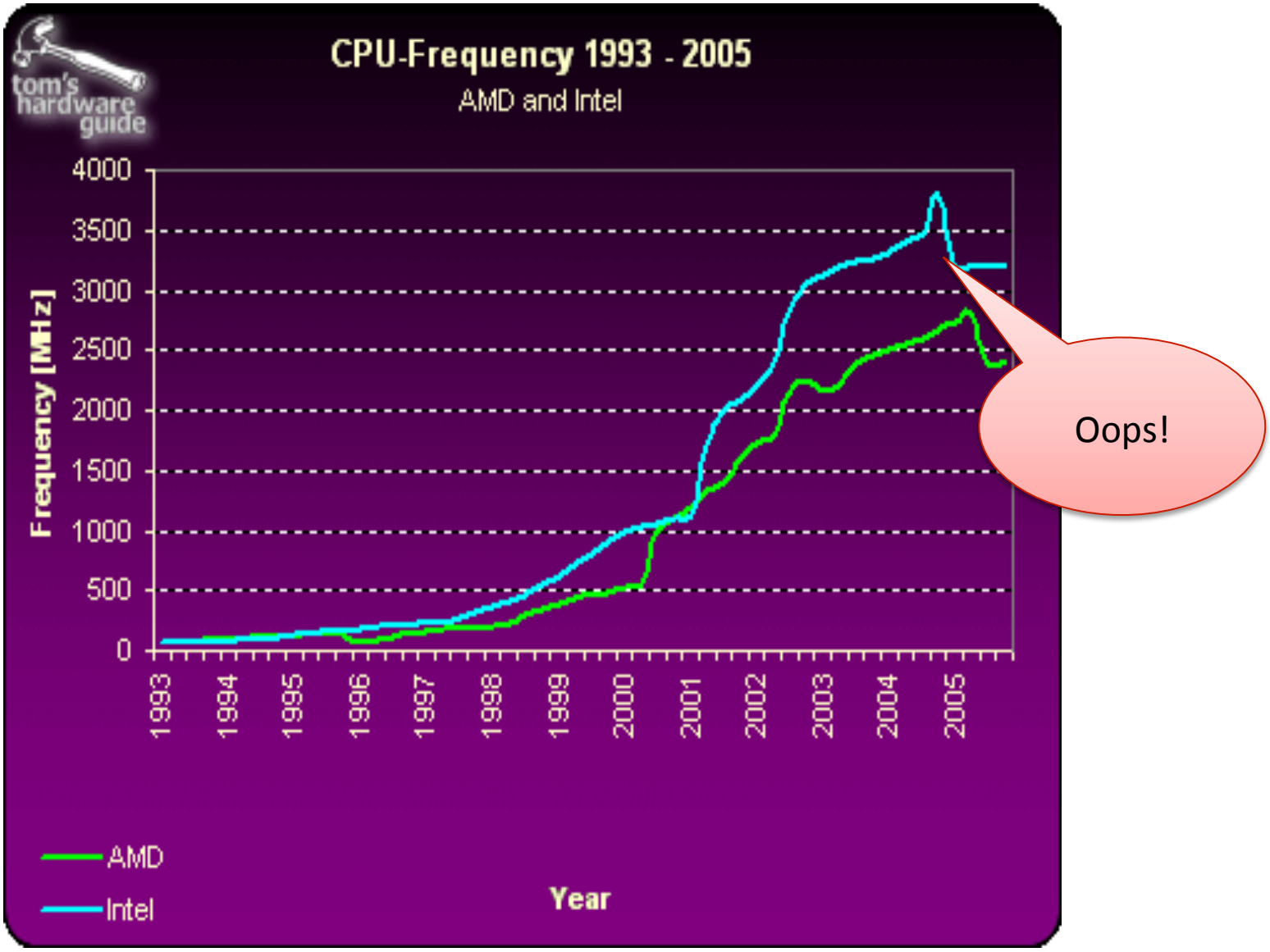
CPU Clock Speeds from 1993-2005



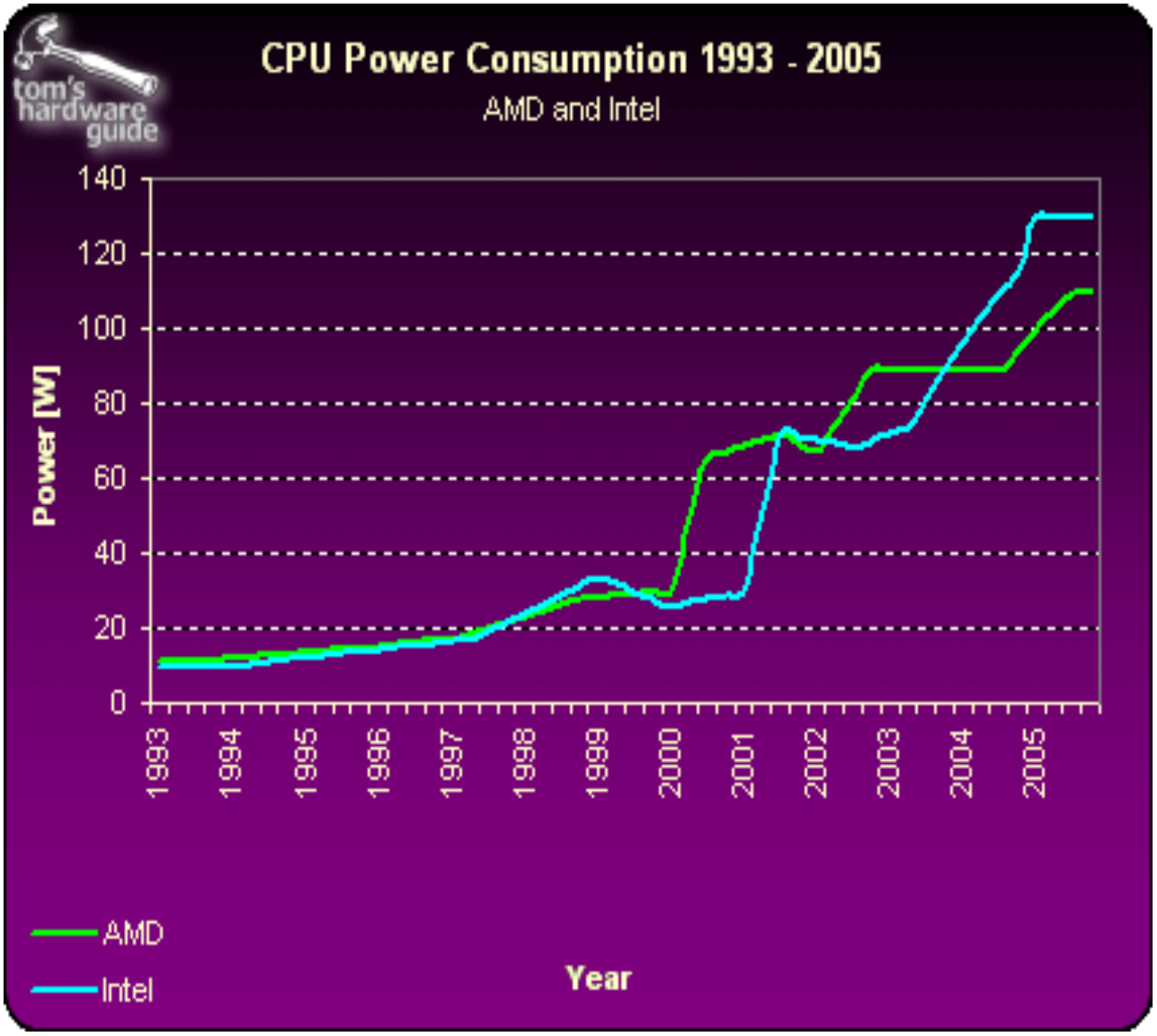
CPU Clock Speeds from 1993-2005



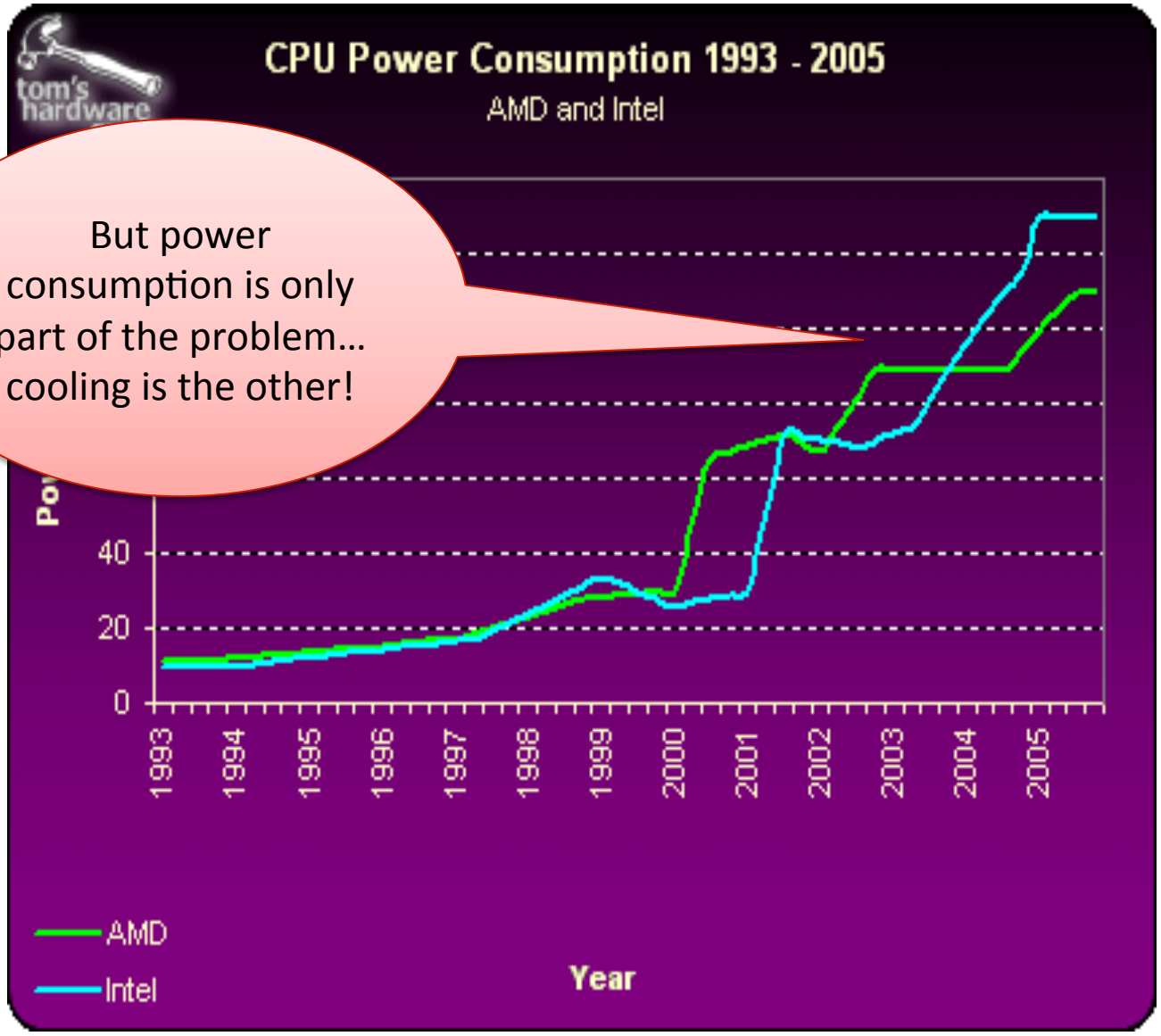
CPU Clock Speeds from 1993-2005



CPU Power 1993-2005

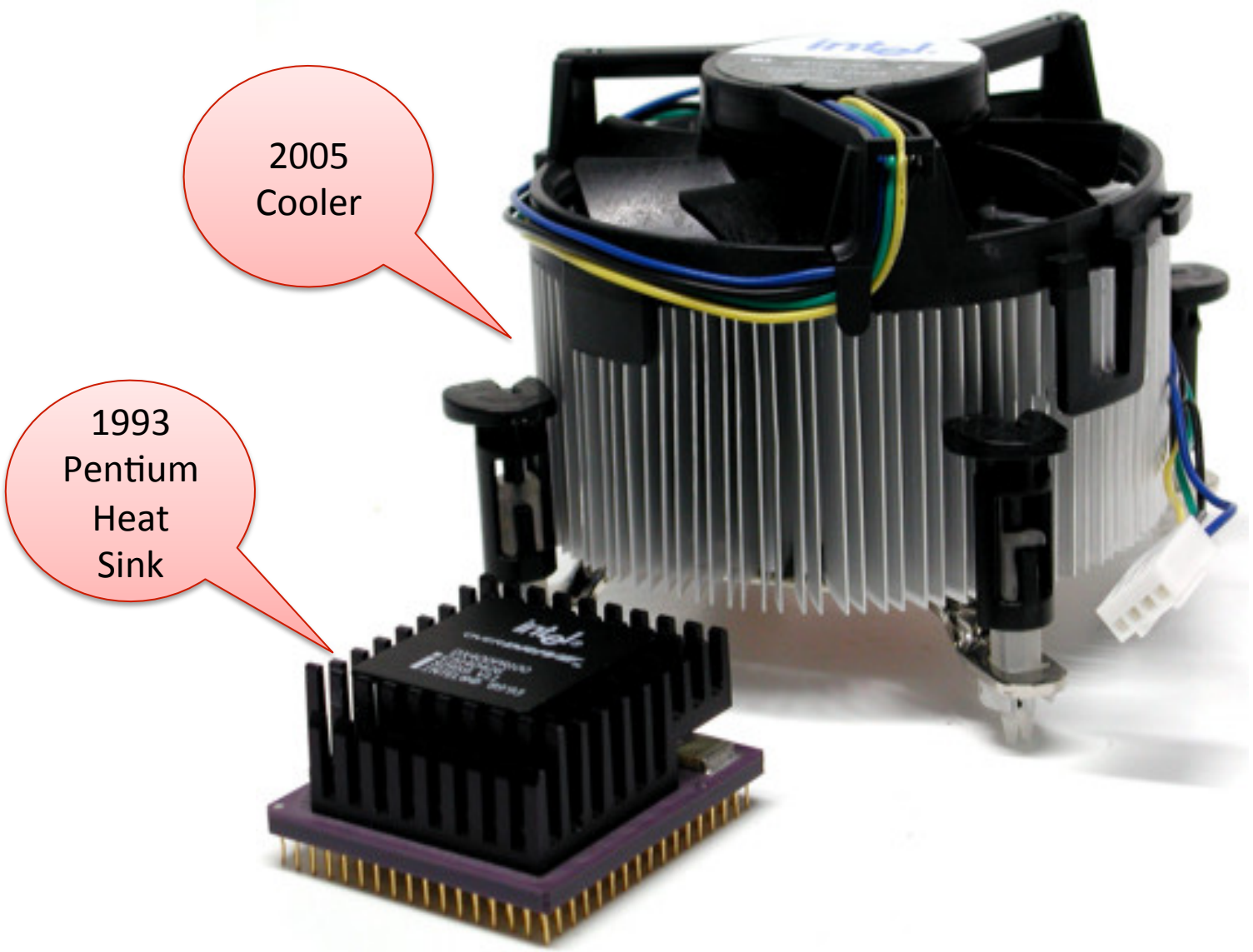


CPU Power 1993-2005



The Heat Problem

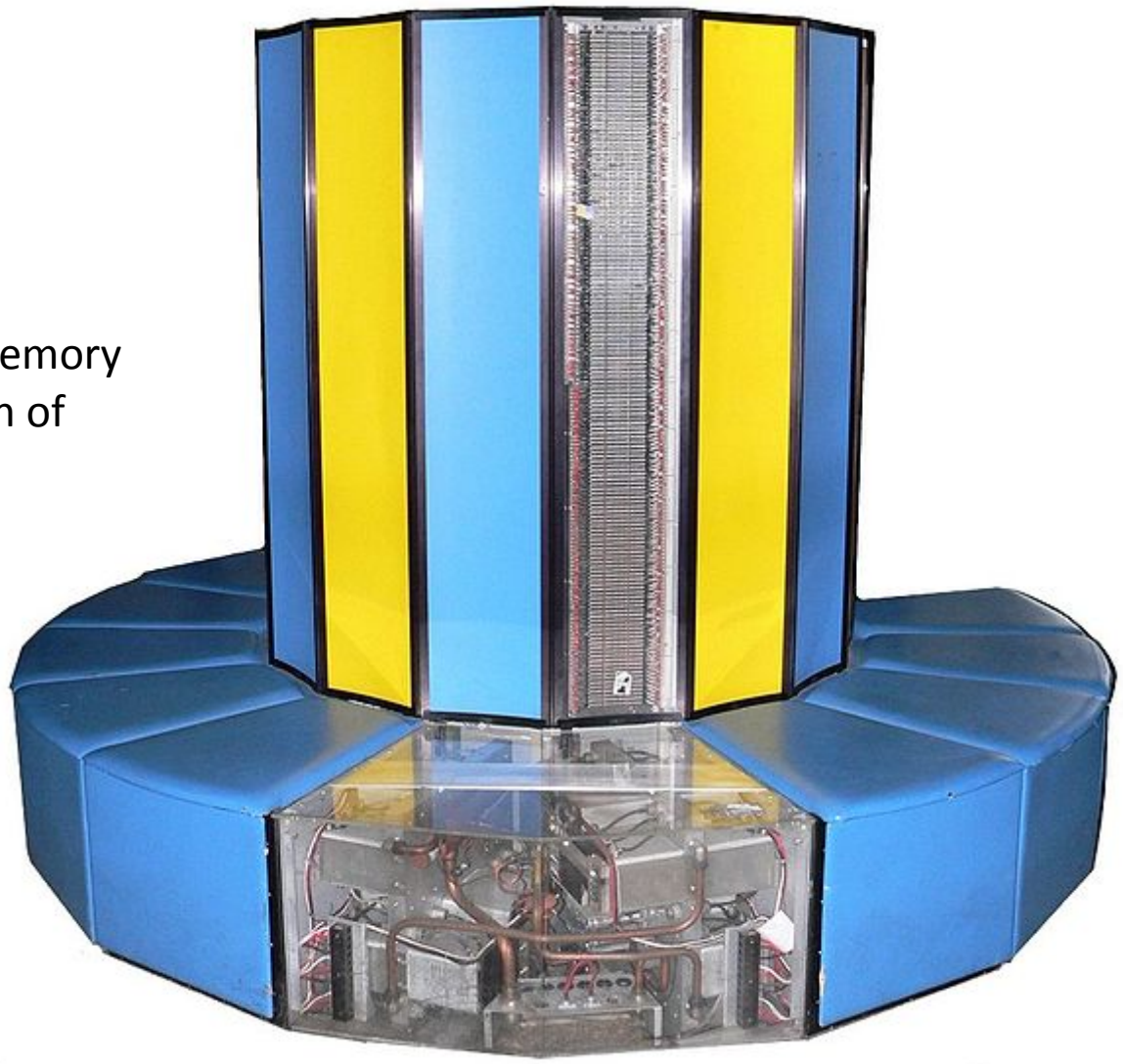
The Problem



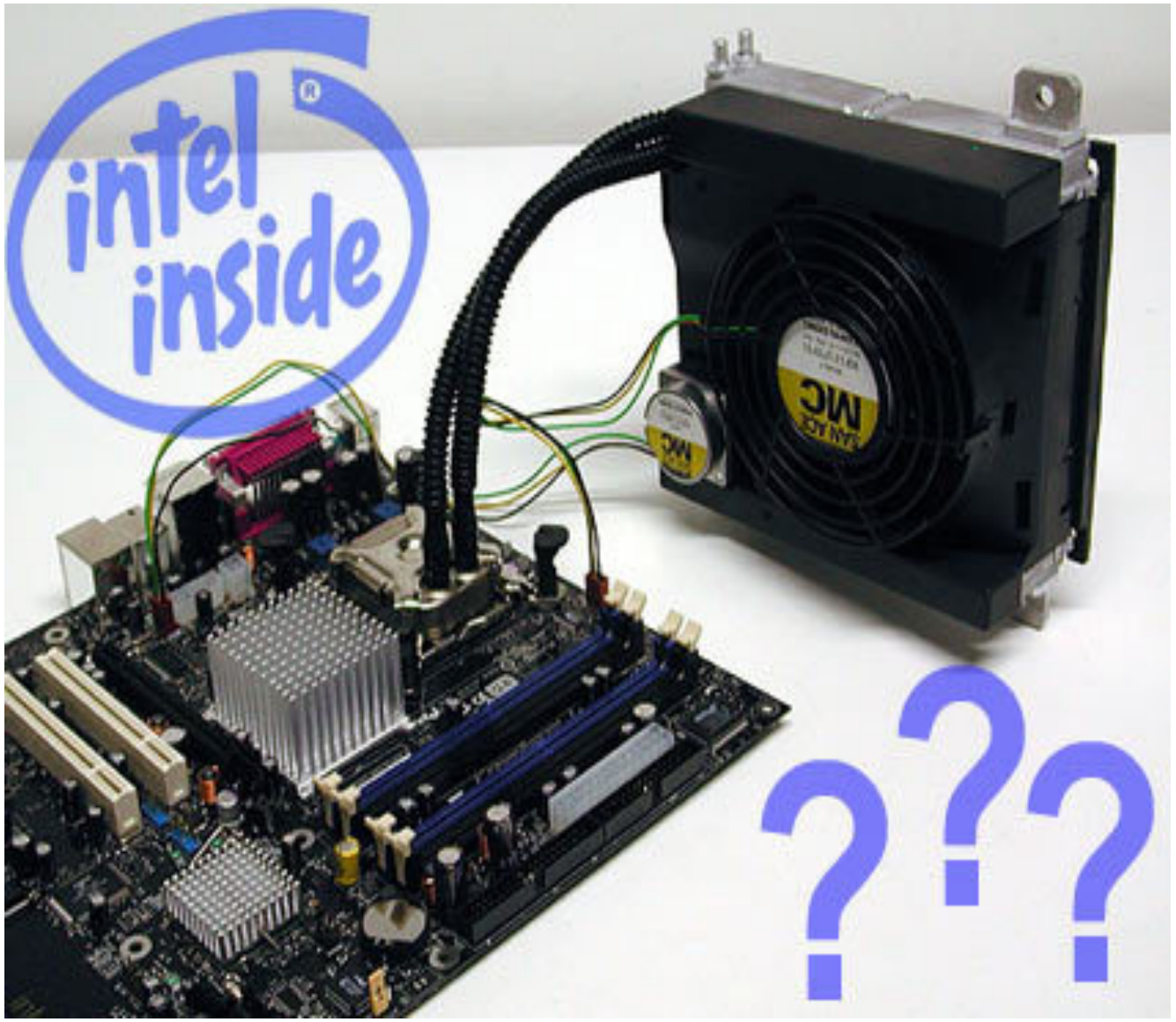
Cray-4: 1994

Up to 64 processors
Running at 1 GHz
8 Megabytes of RAM
Cost: roughly $10M

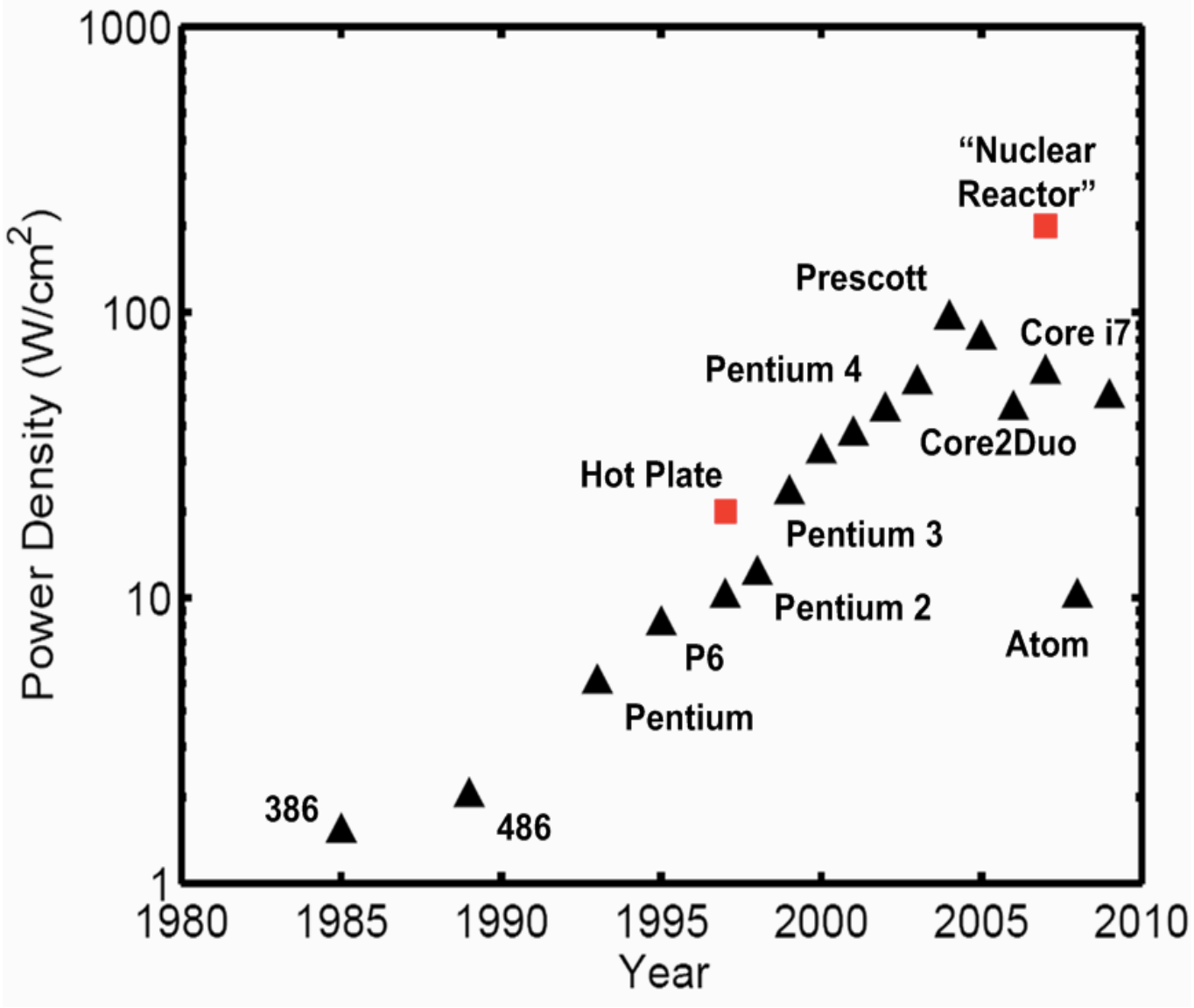
The CRAY 2,3, and 4 CPU and memory boards were immersed in a bath of electrically inert cooling fluid.

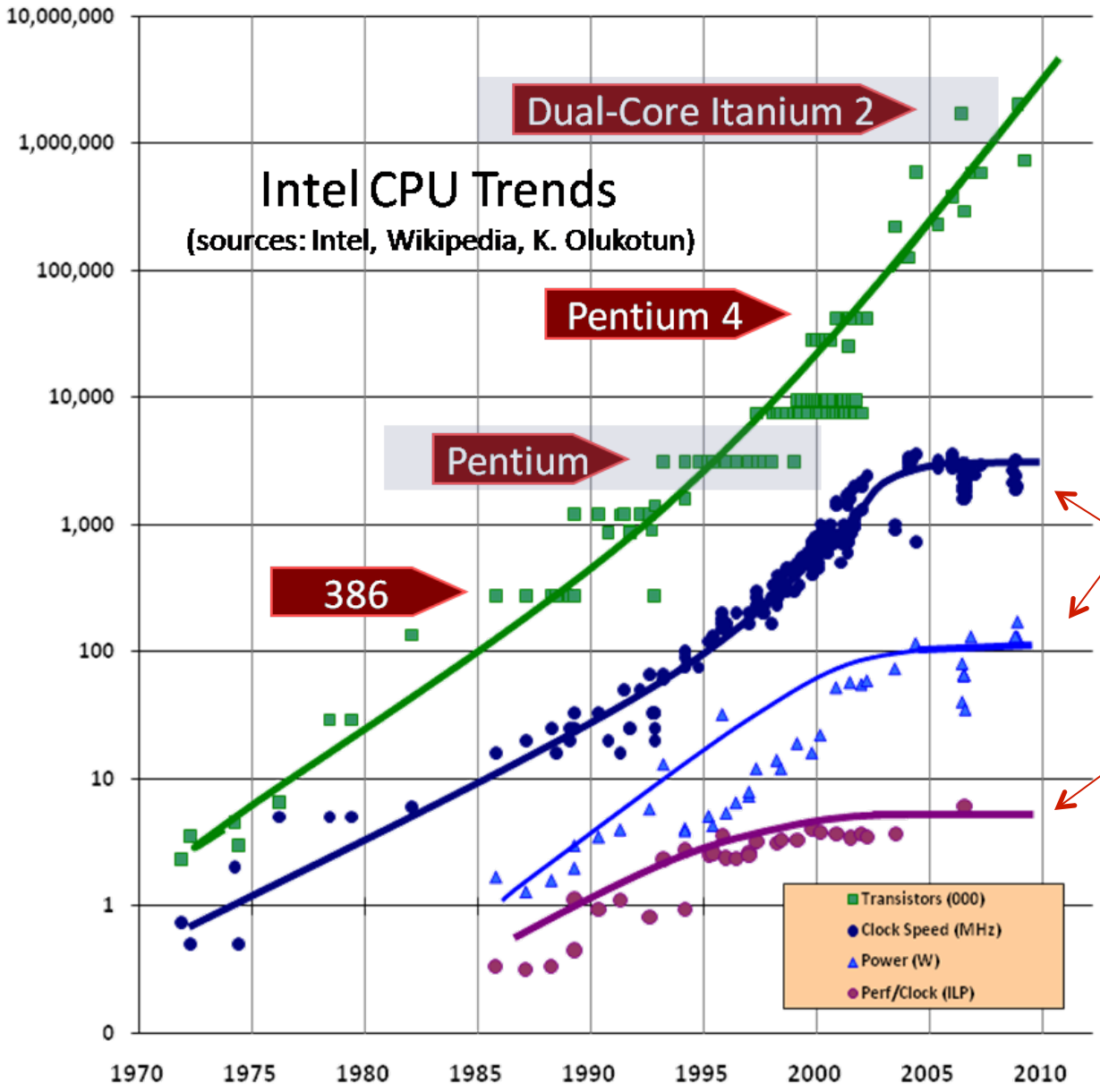


water cooled!



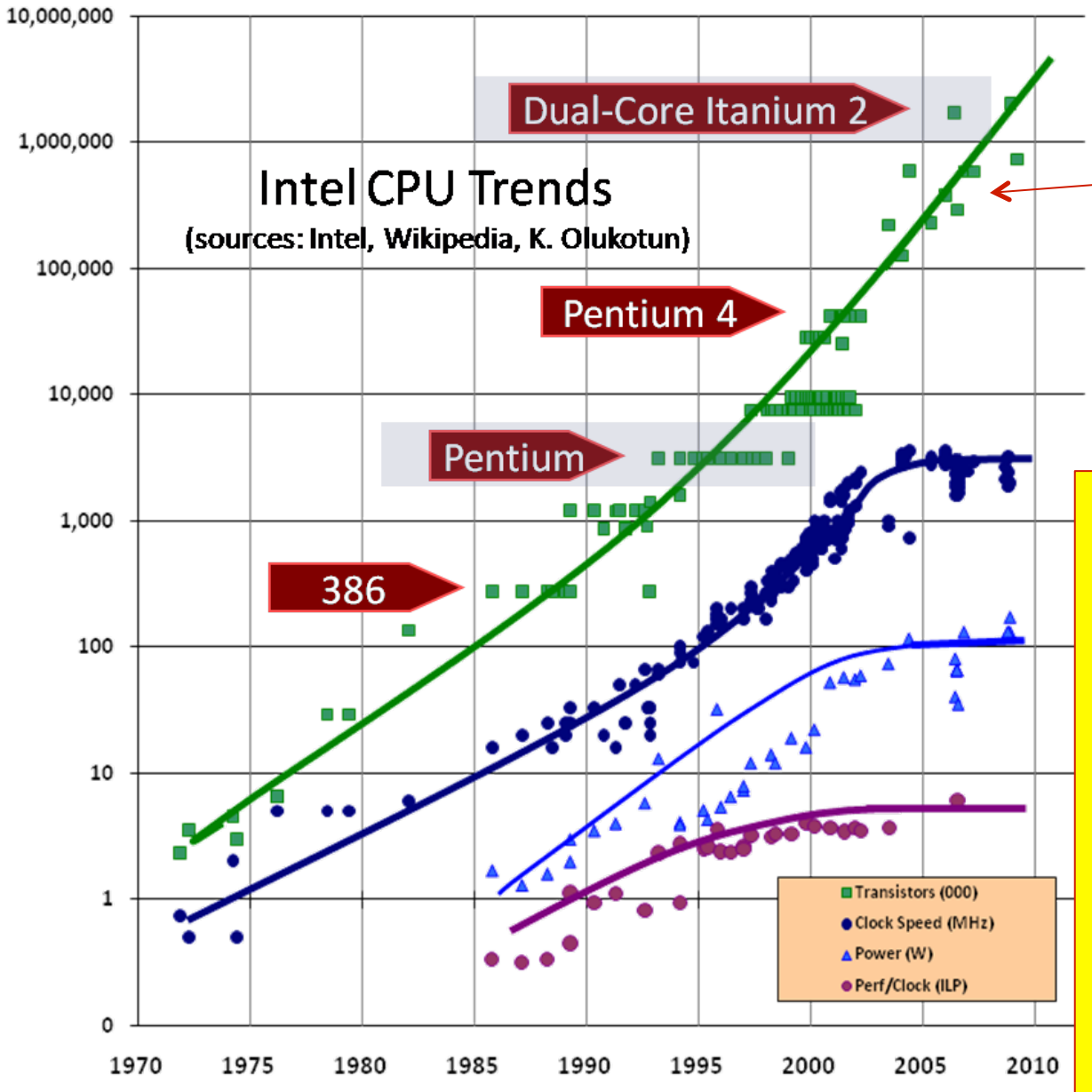
Power Dissipation





Power to chip peaking

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



But look:
Moore's Law still holds, so far, for transistors-per-chip.

What do we do with all those transistors?

1. Multicore!

2. System-on-chip with specialized coprocessors (such as GPU)

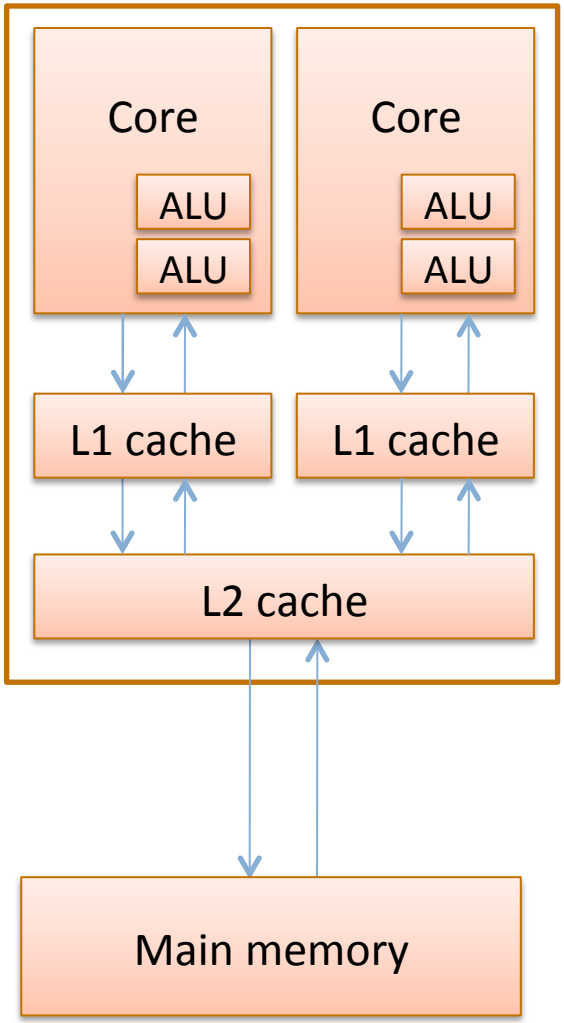
Both of those are PARALLELISM

Parallelism

Why is it particularly important (today)?

- Roughly every other year, a chip from Intel would:
  - halve the feature size (size of transistors, wires, etc.)
  - double the number of transistors
  - double the clock speed
  - this drove the economic engine of the IT industry (and the US!)
- No longer able to double clock or cut voltage: a processor won't get any faster!
  - (so why should you buy a new laptop, desktop, etc.?)
  - power and heat are limitations on the clock
  - errors, variability (noise) are limitations on the voltage
  - but we can still pack a lot of transistors on a chip… (at least for another 10 to 15 years.)

Multi-core h/w – common L2

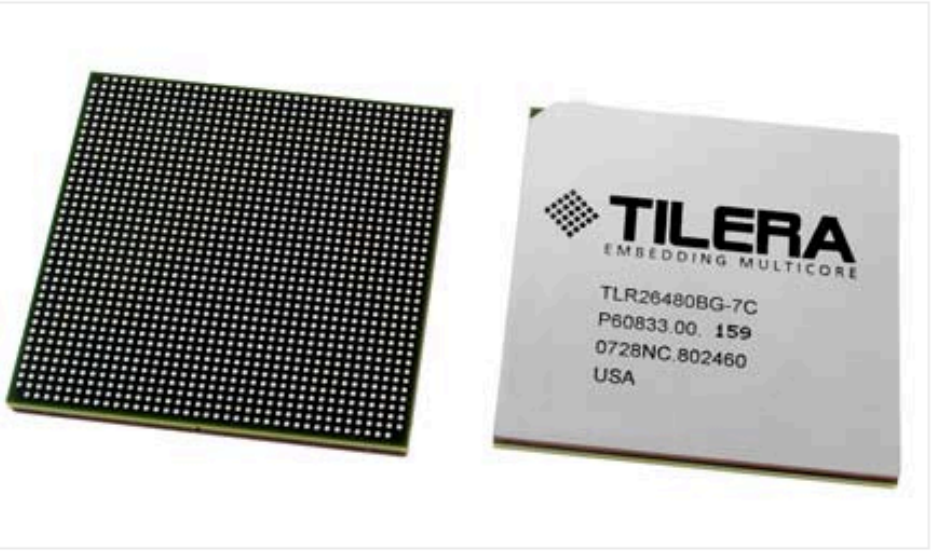


Today… (actually 9 years ago!)

Tilera announces 64-core processor

Posted on August 20, 2007 - 00:00 by **Wolfgang Gruener**

Palo Alto (CA) – Silicon Valley startup Tilera today announced the Tile64, a processor with 64 programmable cores that, according to the company, houses ten times the performance and 30 times the power efficiency of Intel's dual-core Xeon processors.

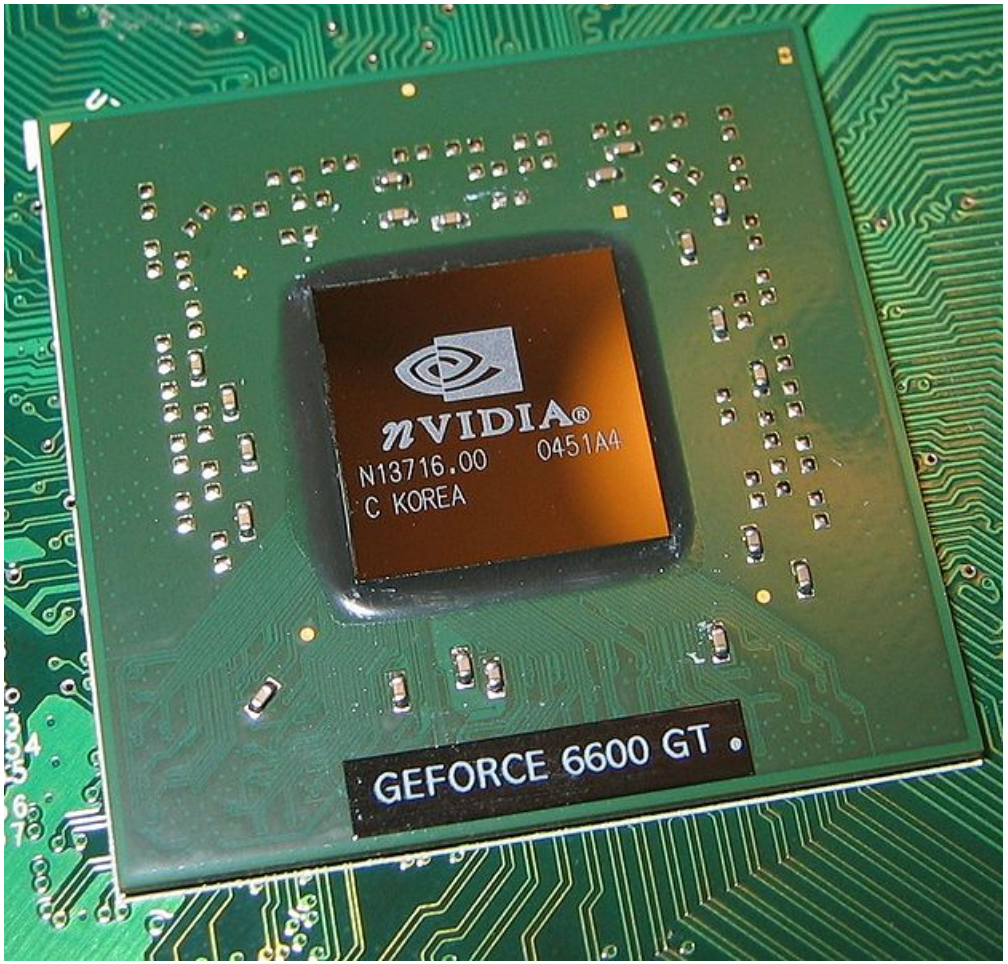


Intel may be getting tired of hearing about products performing better than its dual-core processors targeting server and embedded, as the company describes dual-core processors, at least when it comes to performance, as last year's product. However, when there's a company claiming that it can beat Intel's last year's product by a factor of 10x and 30x, depending on discipline, it's certainly worth a look.

The Tile64 is a 90 nm RISC-based processor clocked between 600 MHz and 1 GHz aiming for integration in embedded applications such as routers, switches, appliances, video conferencing systems and set-top boxes. Its manufacturer claims that the CPU solves a critical problem in multi-core scaling and opens the door to hundreds or even thousands of cores using this new architecture.

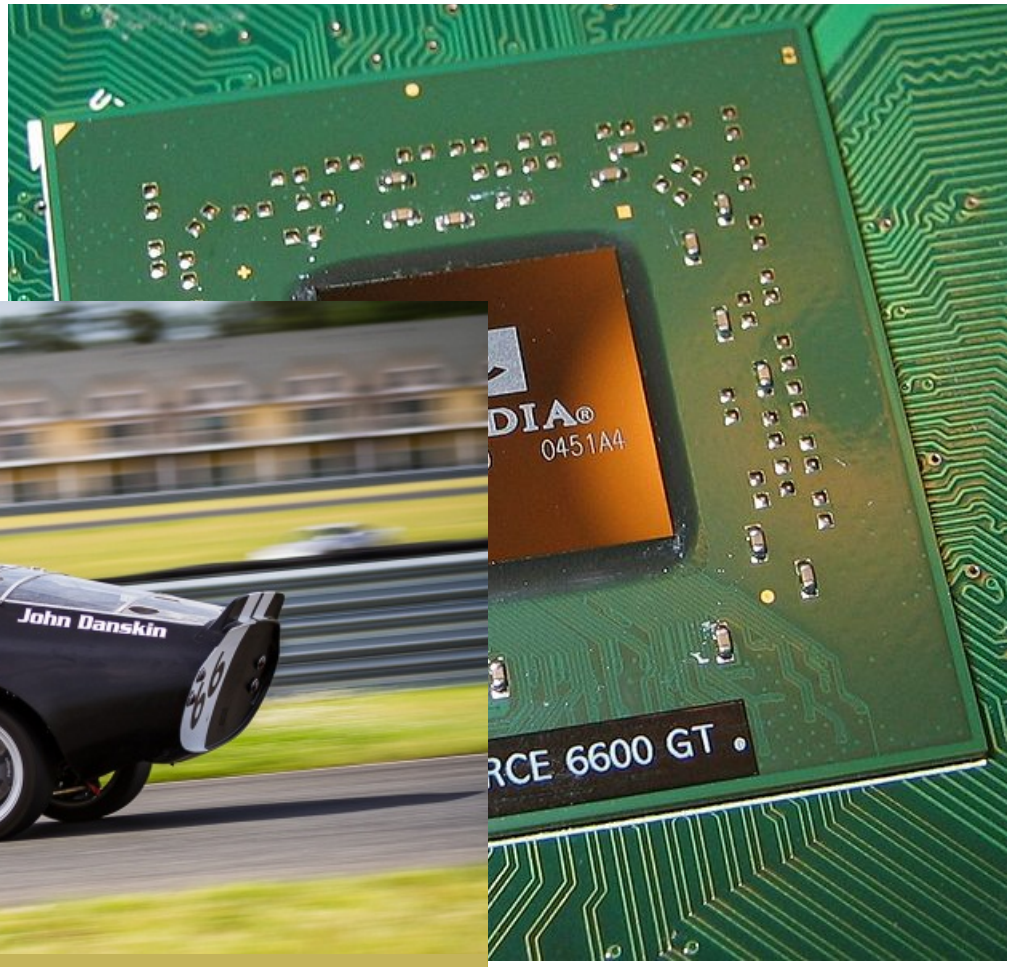
GPUs

- *There's nothing like video gaming to drive progress in computation!*
- GPUs can have hundreds or even thousands of cores
- Three of the 5 most powerful supercomputers in the world take advantage of GPU acceleration.
- Scientists use GPUs for simulation and modelling
  - eg: protein folding and fluid dynamics



GPUs

- *There's nothing like video gaming to drive progress in computation!*
- GPUs can have hundreds or even thousands of
-
-



John Danskin, PhD Princeton 1994,
Vice President for GPU architecture, Nvidia
(what he does with his spare time … built this car himself)

So…

Instead of trying to make your CPU go faster, Intel's just going to pack more CPUs onto a chip.

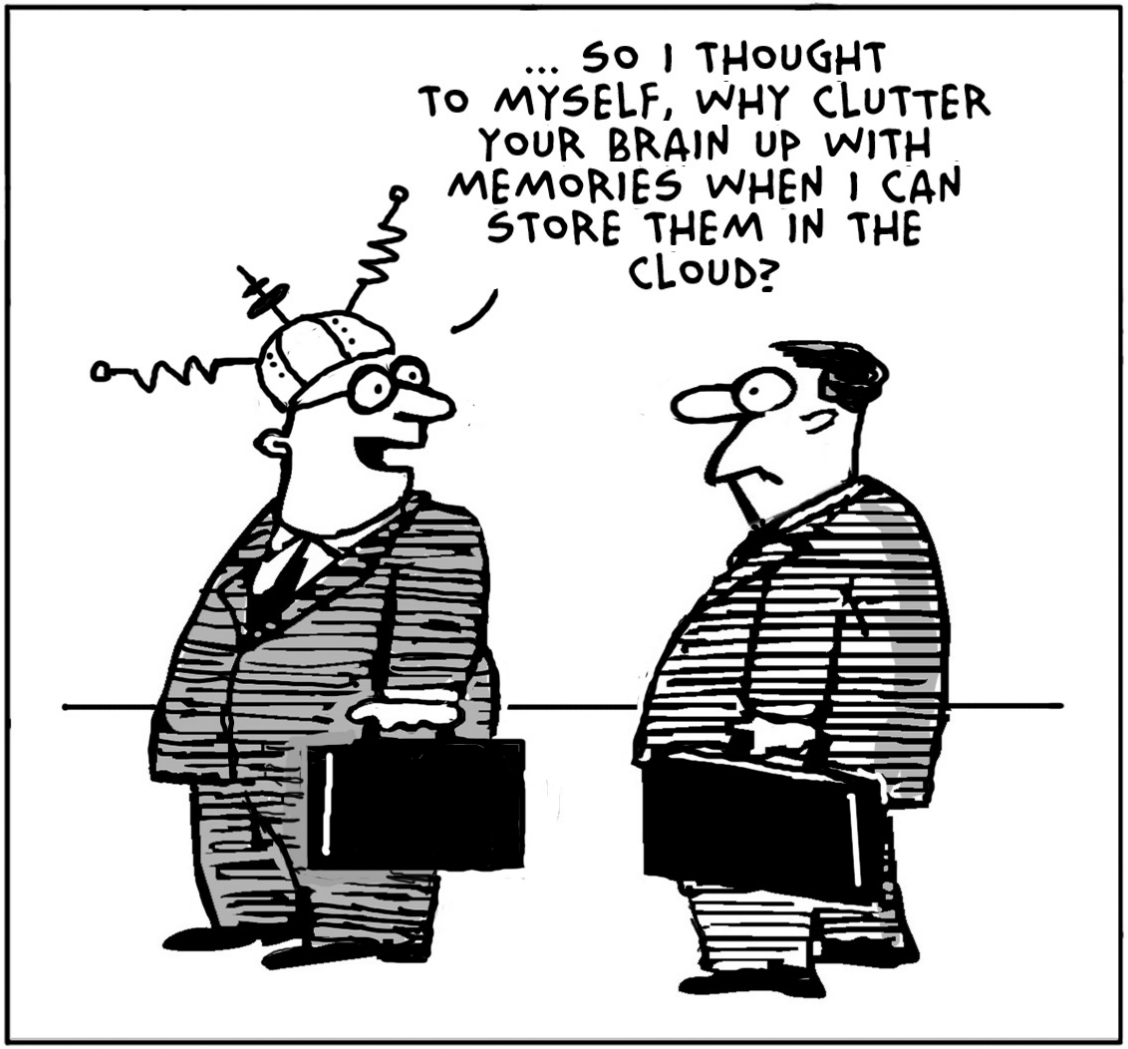
- a few years ago: dual core (2 CPUs).
- a little more recently: 4, 6, 8 cores.
- Intel is testing 48-core chips with researchers now.
- Within 10 years, you'll have ~1024 Intel CPUs on a chip.

In fact, that's already happening with graphics chips (eg, Nvidia).

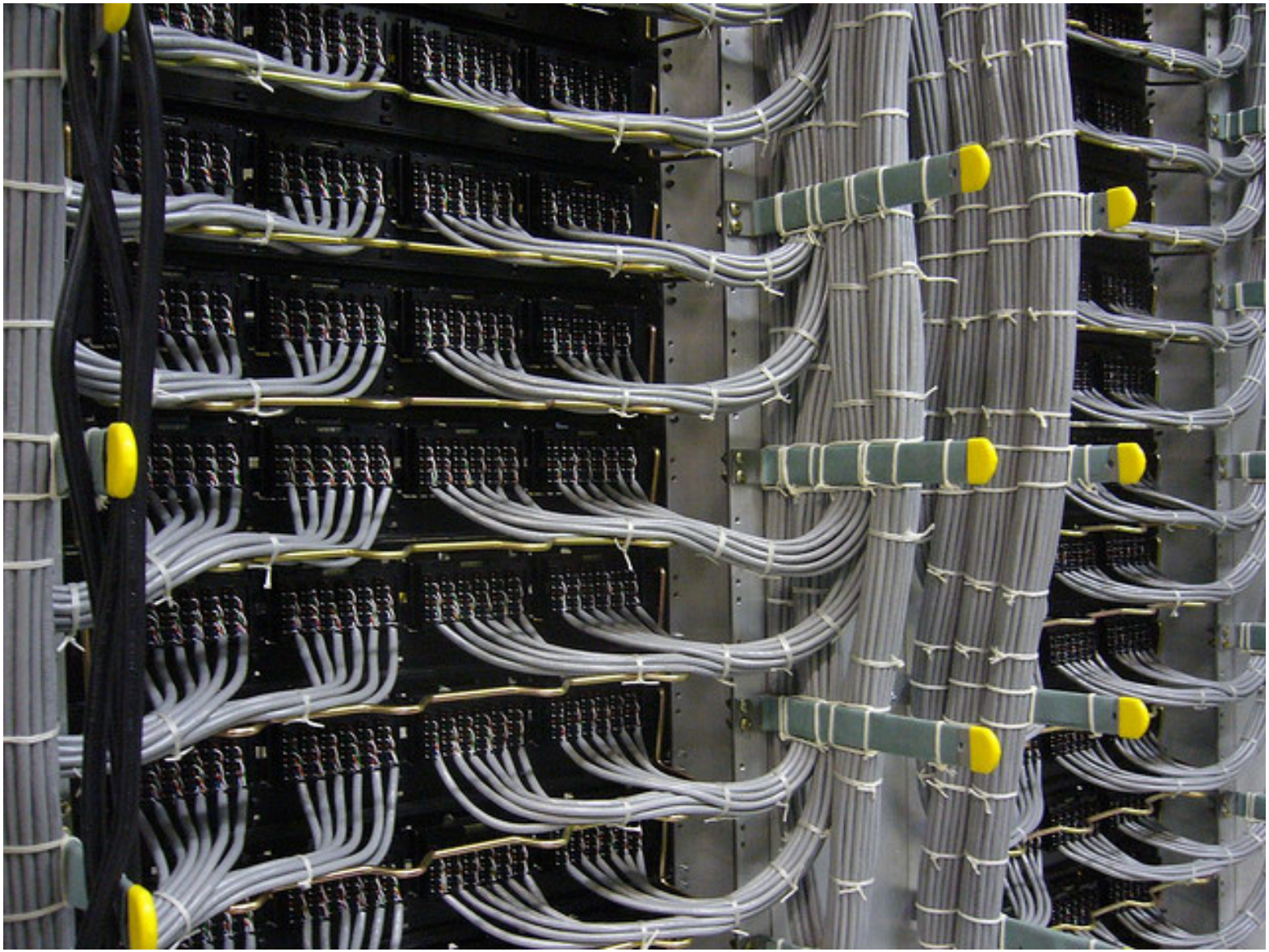
- really good at simple data parallelism (many deep pipes)
- but they are *much* dumber than an Intel core.
- and right now, chew up a *lot* of power.
- watch for GPUs to get "smarter" and more power efficient, while CPUs become more like GPUs.

STILL MORE PROCESSORS: THE DATA CENTER

Data Centers: Generation Z Super Computers



Data Centers: *Lots* of Connected Computers!



Data Centers

- *10s or 100s of thousands* of computers
- All connected together
- Motivated by new applications and scalable web services:
  - let's catalogue all N billion webpages in the world
  - let's all allow anyone in the world to search for the page he or she needs
  - let's process that search in less than a second
- It's Amazing!
- It's Magic!

Data Centers: Lots of Connected Computers

Computer containers for plug-and-play parallelism:

Sounds Great!

- So my old programs will run 2x, 4x, 48x, 256x, 1024x faster?

Sounds Great!

- So my old programs will run 2x, 4x, 48x, 256x, 1024x faster?
  - no way!

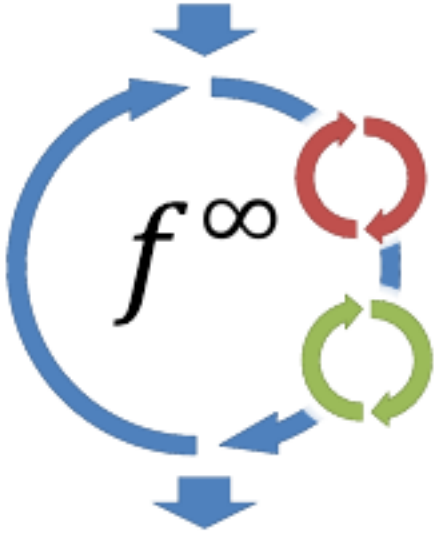
Sounds Great!

- So my old programs will run 2x, 4x, 48x, 256x, 1024x faster?
  - no way!
  - to upgrade from Intel 386 to 486, the app writer and compiler writer did not have to do anything (much)
    - IA 486 interpreted the same sequential stream of instructions; it just did it faster
    - this is why we could watch TV while Intel engineers optimized our programs for us
  - to upgrade from Intel 486 to dual core, we need to figure out how to split a single stream of instructions in to two streams of instructions that collaborate to complete the same task.
    - *without work & thought, our programs don't get any faster at all*
    - *it takes ingenuity to generate efficient parallel algorithms from sequential ones*

What’s the answer?

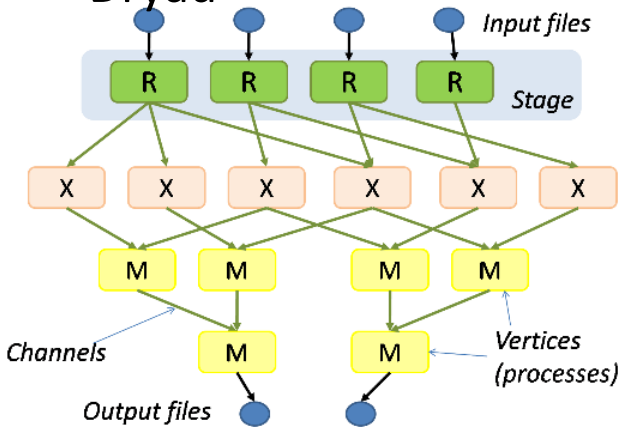
In Part: Functional Programming!

Naiad



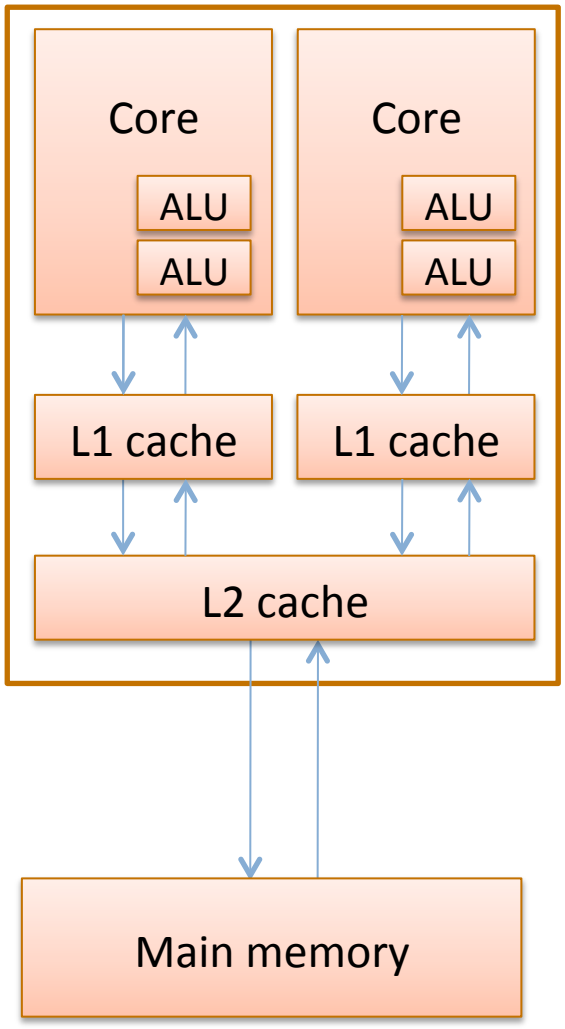
Pig

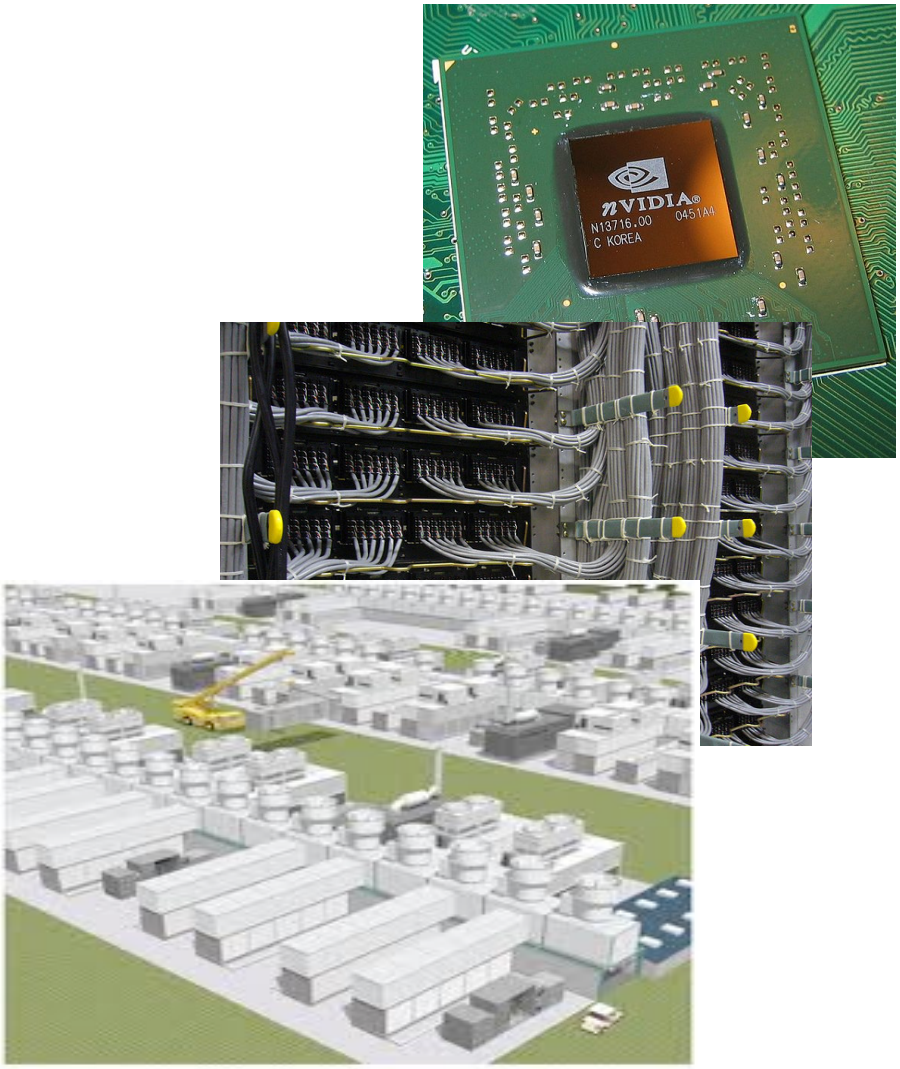
Dryad



PARALLEL AND CONCURRENT PROGRAMMING

Multicore Hardware & Data Centers





Speedup

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

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

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

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

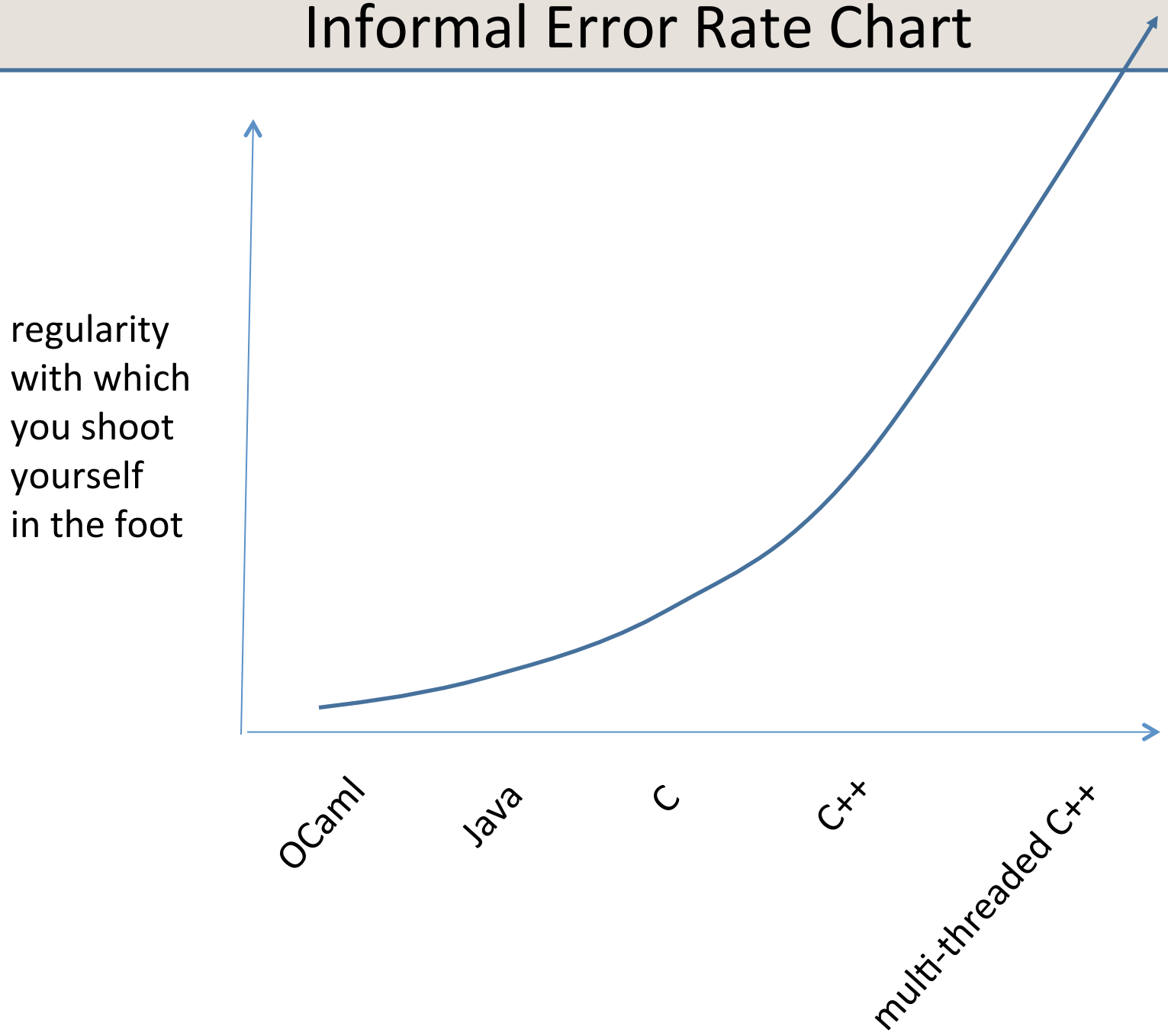
- *Bad news: Not every program can be effectively parallelized.*
  - in fact, very few programs will scale with perfect speedups.
  - we certainly can't achieve perfect speedups automatically
  - limited by sequential portions, data transfer costs, ...

Most Troubling…

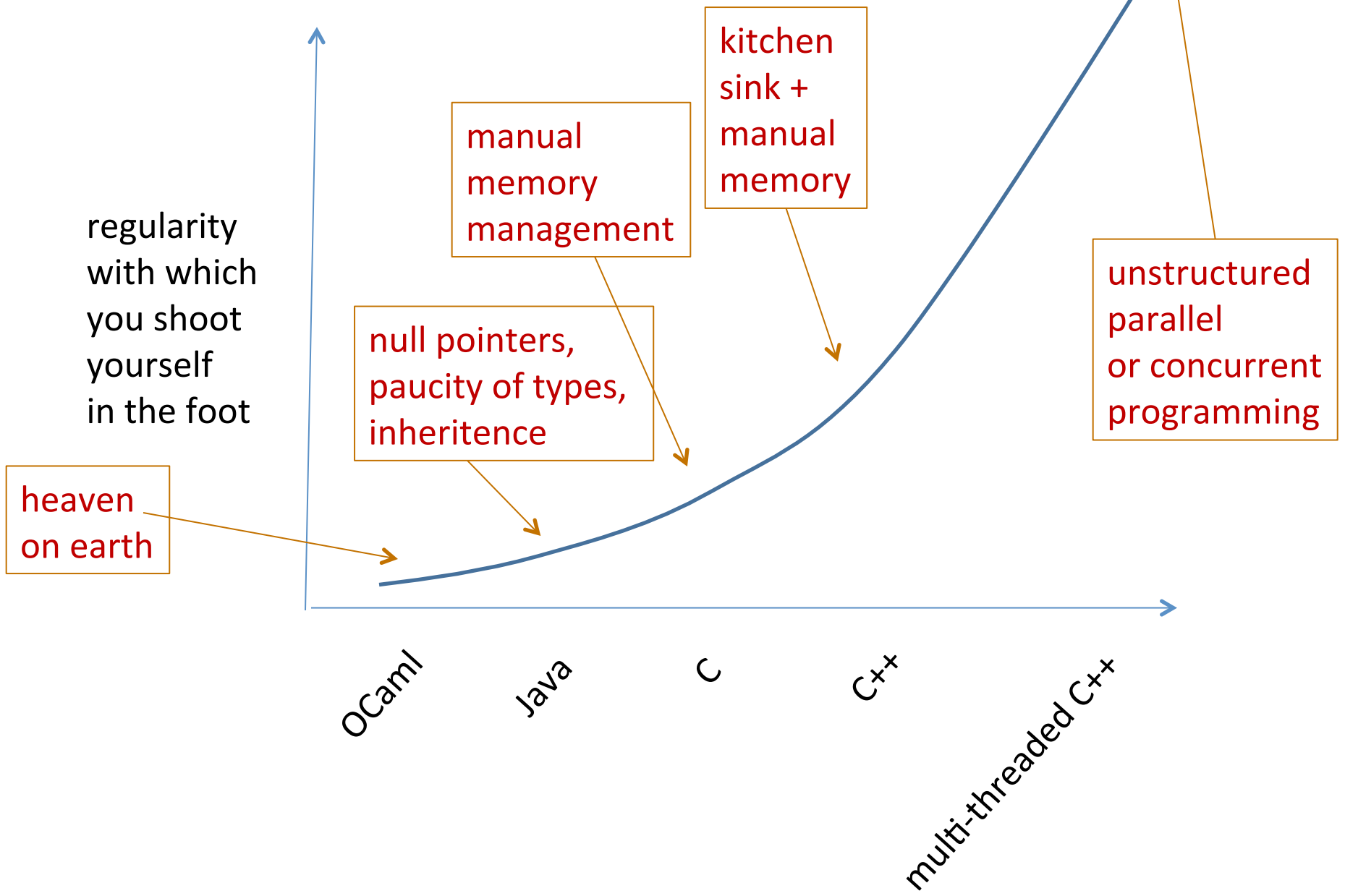
Most, *but not all*, parallel and concurrent programming models are far harder to work with than sequential ones:

- *They introduce nondeterminism*
  - 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 nonmodular*
  - 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



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 nondeterminism can lead to reduced resource utilization
   - synchronization, communication costs may need optimization

OUR FIRST PARALLEL PROGRAMMING MODEL: THREADS

Threads: A Warning

- *Concurrent Threads with Locks: 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.:

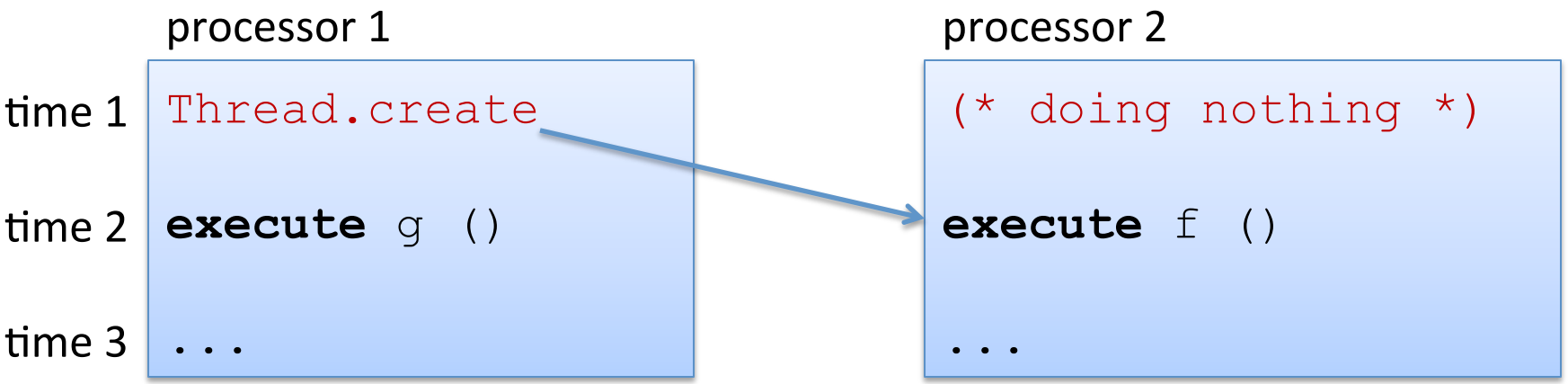
```
let _ = compute_big_thing() in
let y = compute_other_big_thing() in
...
```

  - 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
 ...
```



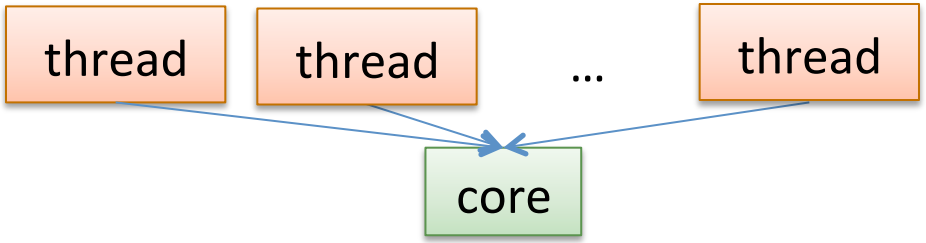
Of Course…

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 "runtime system" or garbage collector. 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.

You can hide I/O latency, do multiprocess programming or distribute tasks amongst multiple computers in OCaml.

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.

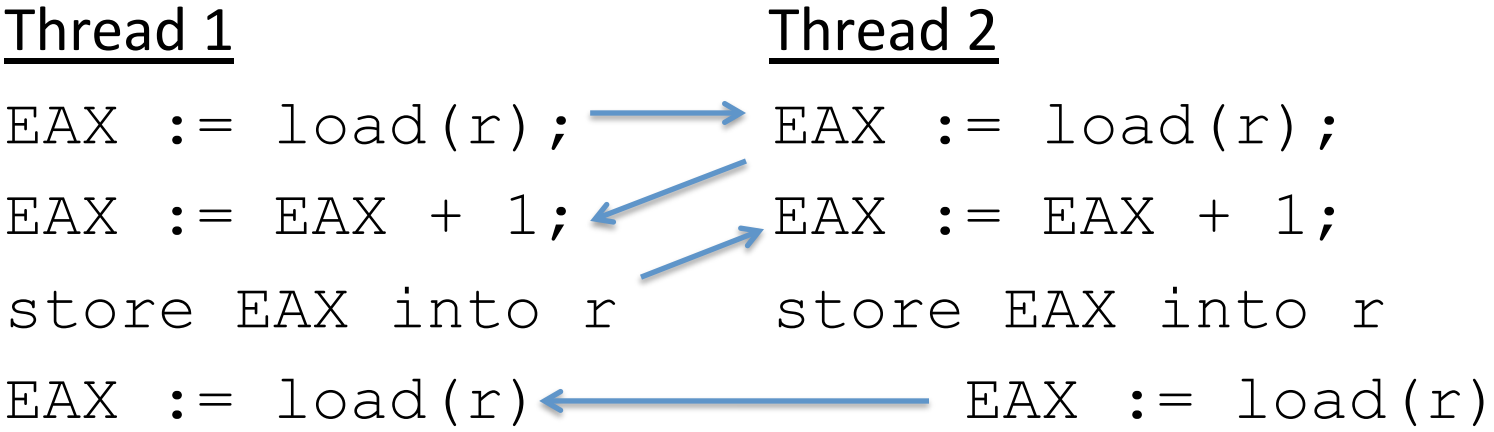
Rule 2: Given a program:

```
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:



What answer do we get?

Atomicity

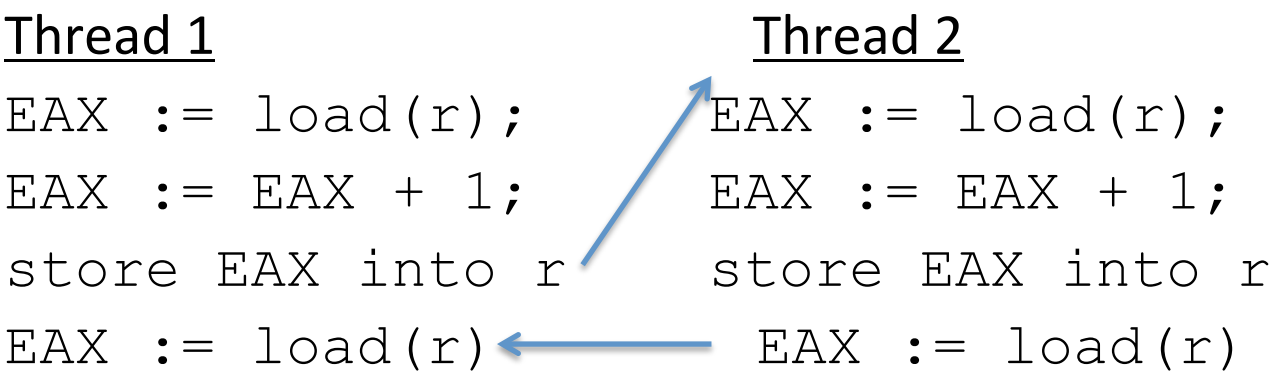
Another possible interleaving:

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:



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 *nondeterminism*.

Atomicity

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

Thread 1

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

Thread 2

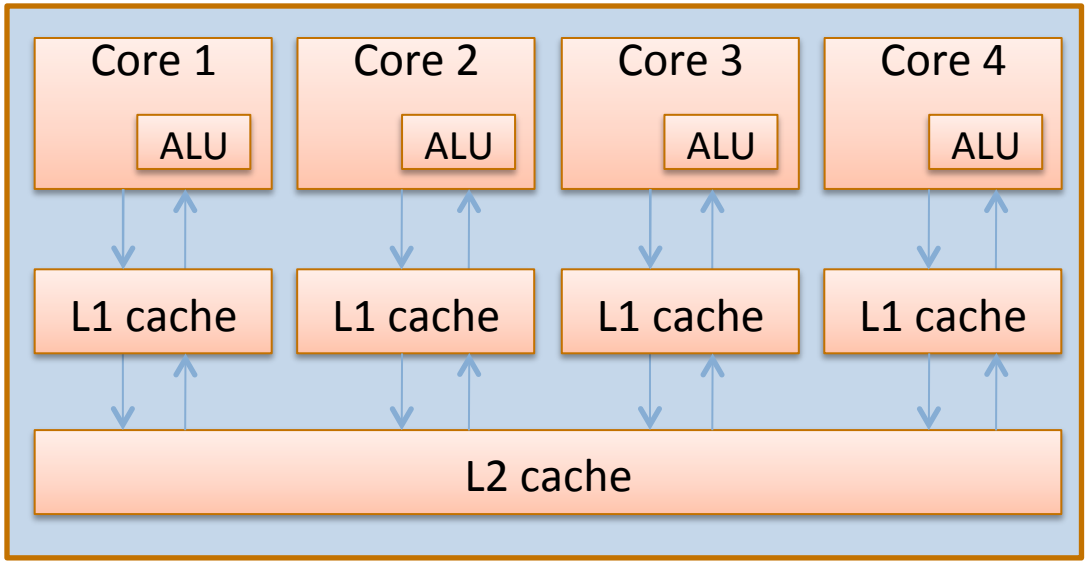
```
EAX := load(r);
EAX := EAX + 1;
store EAX into r
     EAX := load(r)
```

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! But the take-away is this: It's not a good idea to use ordinary loads/stores to synchronize threads; you should use explicit synchronization primitives so the hardware and optimizing compiler don't optimize them away.

Atomicity

In fact, today's multicore processors don't treat memory in a *sequentially consistent* fashion. That means that *we can't even assume that what we will see corresponds to some interleaving of the threads' instructions!*



When Core1 stores to "memory", it *lazily* propagates to Core2's L1 cache. The load at Core2 might not see it, unless there is an explicit synchronization.

Beyond the scope of this class! But the take-away is this: It's not a good idea to use ordinary loads/stores to synchronize threads; you should use explicit synchronization primitives so the hardware and optimizing compiler don't optimize them away.

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!

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 *happen*
  - can't have a child th… before a parent forks it.
  - …
- He…
  - …omic.
  - …
  - …s of

WARNING

If you see people talk about interleavings, BEWARE!
It probably means they're assuming
"sequential consistency,"
which is an oversimplified, naïve model of what the parallel computer really does.
It's actually more complicated than that.

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!

A conventional solution for shared-memory parallelism

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

Thread 1

```
lock(mutex);
inc(r);
!r
unlock(mutex);
```

Thread 2

```
lock(mutex);
inc(r);
!r
unlock(mutex);
```

Guarantees *mutual exclusion* of these critical sections.

This solution works (even for real machines that are not sequentially consistent), **but…**

Complex to program, subject to *deadlock*, prone to bugs, not fault-tolerant, hard to reason about.

A conventional solution for shared-memory parallelism

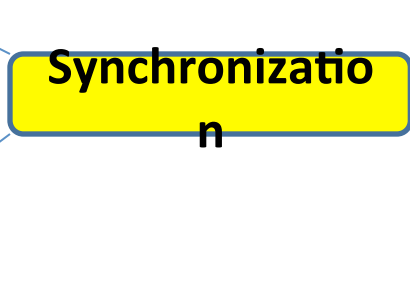
```
let inc(r:int ref) = r := (!r) + 1
```

Thread 1

```
lock(mutex);
inc(r);
!r
unlock(mutex);
```

Thread 2

```
lock(mutex);
inc(r);
!r
unlock(mutex);
```



Guarantees *mutual exclusion* of these critical sections.

This solution works (even for real machines that are not sequentially consistent), **but…**

Complex to program, subject to *deadlock*, prone to bugs, not fault-tolerant, hard to reason about.

Another approach 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 with v and y *)
    | None -> failwith "impos          "
```

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

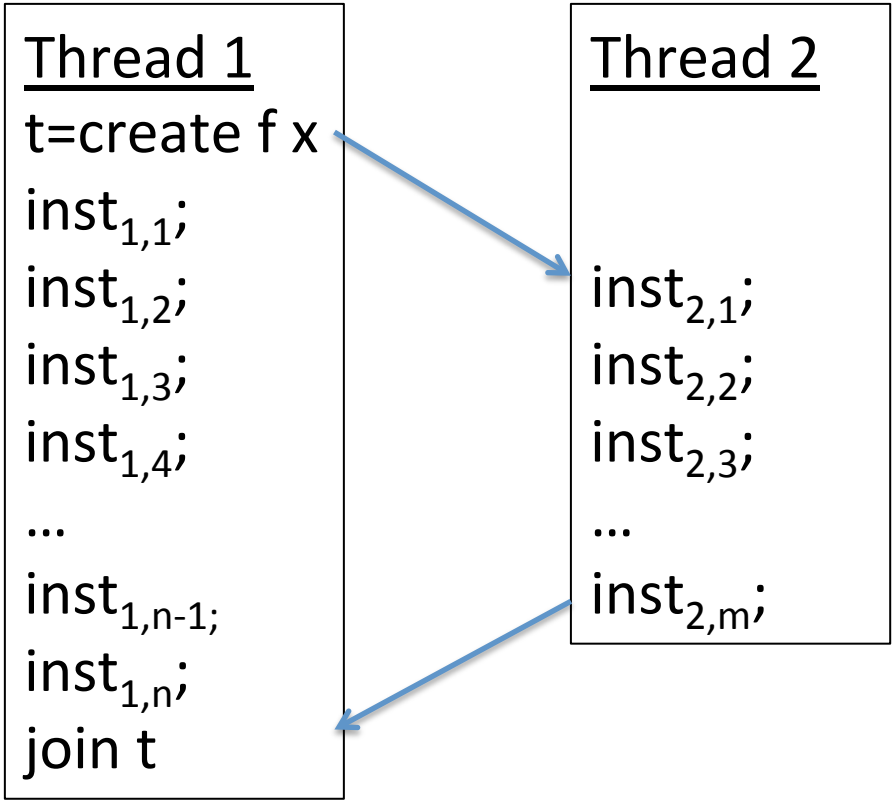
One Solution (using join)

```
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"
```

Synchronization

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

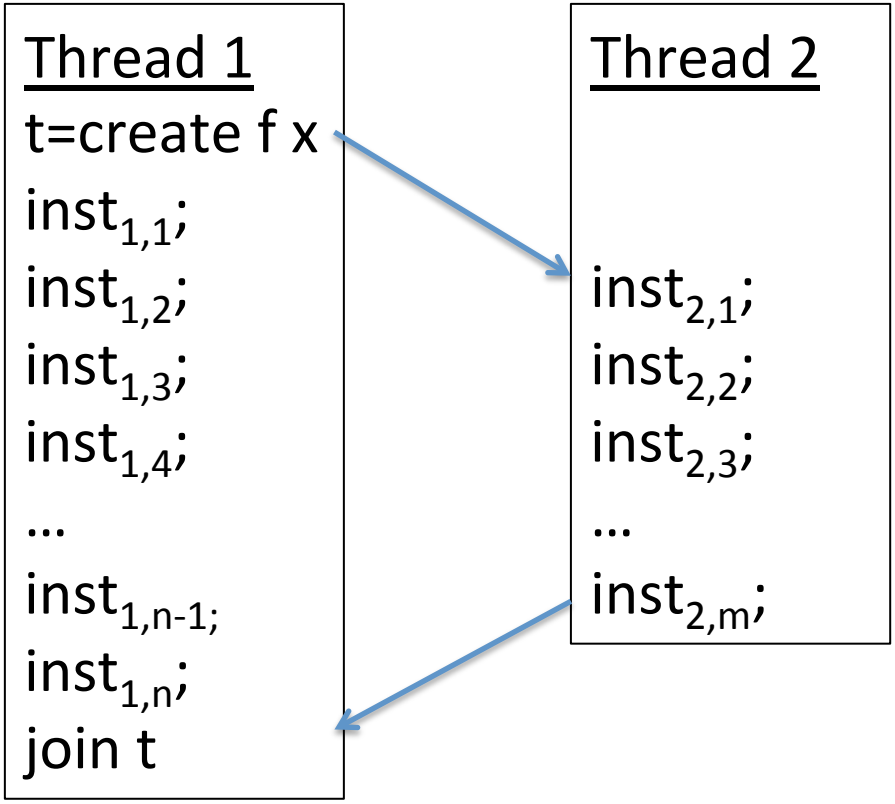
In Pictures



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

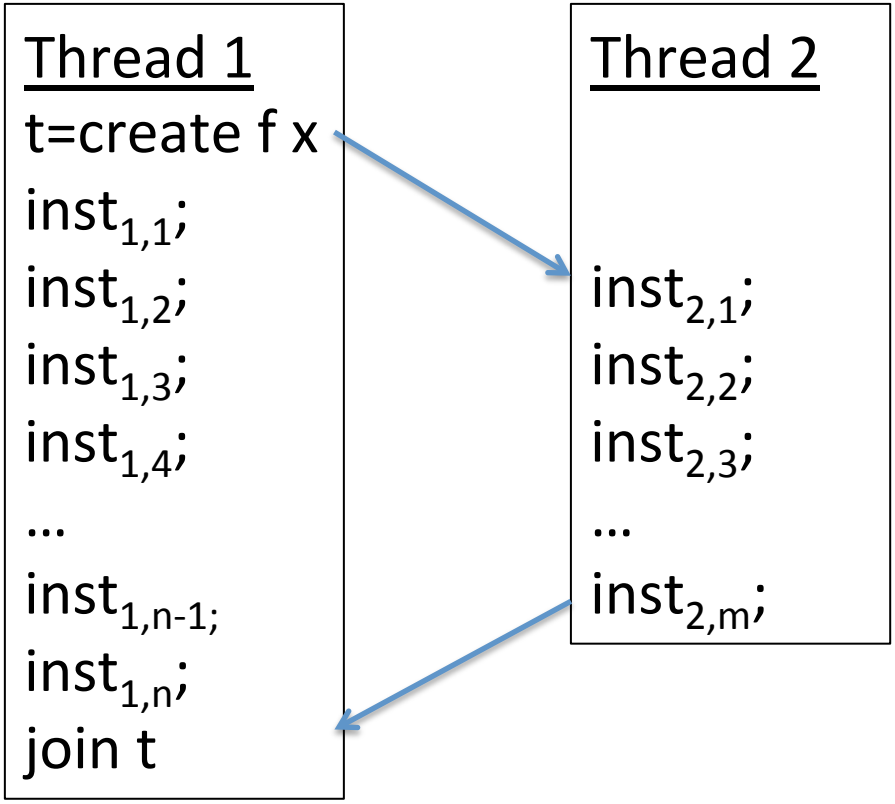
So for instance, $inst_{1,1}$ must happen before $inst_{1,2}$.

In Pictures



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

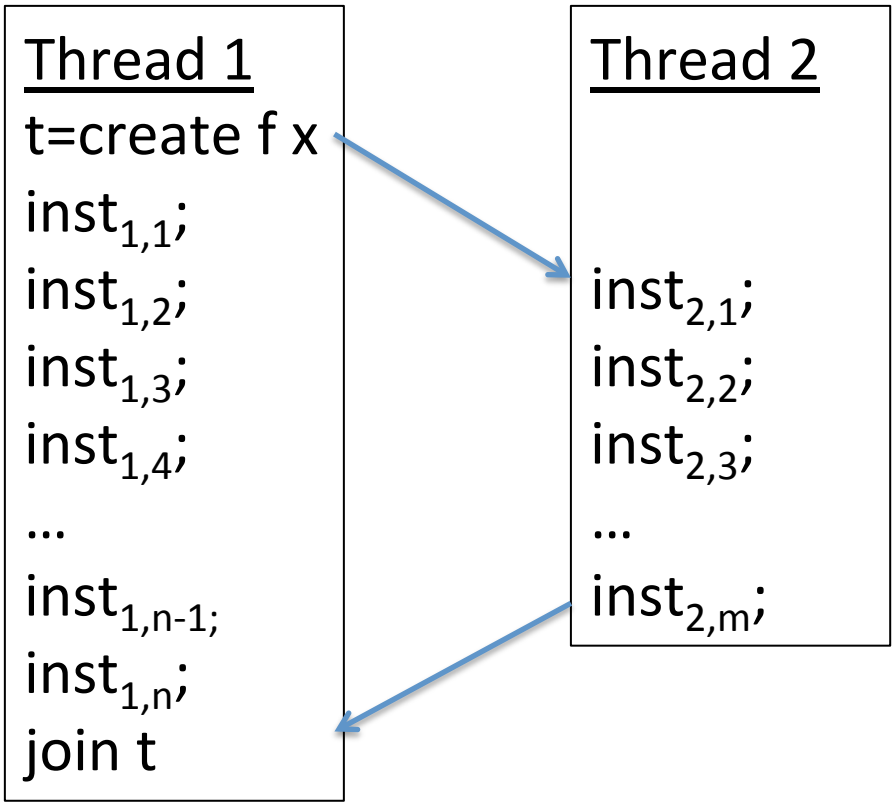
In Pictures



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.

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 nondeterminism.

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