Real World Distributed and Concurrent Programming using Erlang
Chalmers 2017-03-01

Cons T Åhs

cahs@cisco.com/cons@tail-f.com

@lisztspace

Tail-f is now part of Cisco.

Overview

- Presentation of me, Cisco and tail-f
- The two products of tail-f engineering
- Large and small scale concurrency and distribution
- Hardware trends, modern hardware and models of concurrency and distribution
- Erlang - the language and virtual machine

Cons T Åhs

- Technical Leader at Cisco since Sept, 2014
- Core developer using (almost) only Erlang (and some C)
- Previous:
    - Keeper of The Code & developer, Klarna (lots of Erlang)
    - Independent consultant (Lisp, Prolog, Java, C, C++, Actionscript, ..)
        - online poker, medical image analysis, speech synthesis, music notation, 3D graphics, real time video decoding, networking, financial systems, compilers and language implementation, teaching, ..
    - Lecturer and researcher at Uppsala University
        - teaching at all levels of Computing Science Programme (fundamentals, algorithms, compilers, functional programming, logic programming, tools, ..)
        - formal methods of programs, language implementation, theorem proving

Cisco

- Well known manufacturer of network equipment (routers, switches, firewalls, ..) mainly for enterprise use

- A large company

- Hardware is getting cheaper, more difficult to sustain on that alone

- Networks are getting larger and needs to be configured and include high lever services, e.g., VPN

Tail-f

- A small Swedish company focused on network configuration
- Two products:
  - ConfD - network device configuration (small scale)
  - NSO (NCS) - network service configuration (large scale)

Cisco + Tail-f = true

- Cisco acquired tail-f mid 2014
  - We're now called Tail-f engineering
- Main reason to strengthen area of large scale network configuration (NSO) and the requirements of service providers
- ConfD (used by competitors to Cisco) still available, even as free product (ConfD Basic), under Tail-f brand

Tail-f ConfD

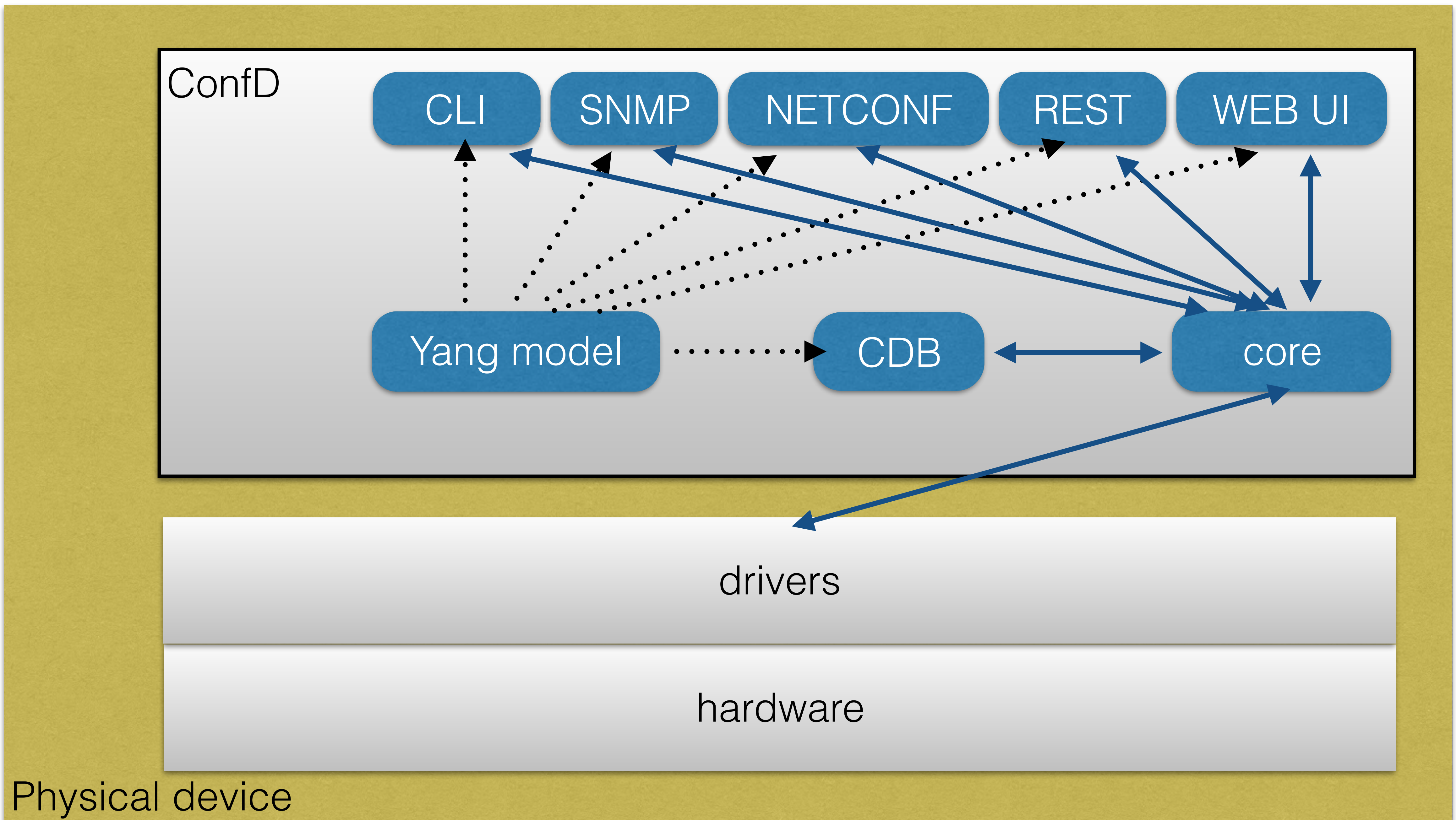
- Typical customer: network device manufacturer (Cisco and competitors to Cisco)
- Problem solved:
  - network device configuration needs interfaces (at least one of CLI, web UI, snmp, netconf, REST)
  - decouple hardware design and specifics from software
  - focus on hardware and interface between hardware and (generic) software
  - same hardware, different properties through configuration

Tail-f ConfD

- provide (generate/render) standard northbound interfaces:
  - device model is written in Yang (RFC 6020)
    - a hierarchical data model
  - Northbound interfaces are *generated* from the Yang model
    - netconf, snmp, CLI, web UI, REST
    - several interfaces and sessions can be active at the same time

ConfD architecture

- Very much is generated from the Yang model



Tail-f ConfD

- Device configuration (and operational data) stored in hierarchical database (cdb) which corresponds to data model
    - cdb is written in house (combination of Erlang and C)
- Configuration changes are done with transactions
    - crucial since several sessions (via same or different interfaces) can be active at the same time
- Subscribe to changes in data model and react on them
    - change IP in config -> reconfigure hardware
    - subscribers typically written in C and communicates directly with the hardware
- Operational data is, e.g., statistics
    - described in Yang
    - has similar, but not identical, semantics as config data
    - typically written from southbound interface, i.e., hardware drivers and reported/used in data model and other subscribers

Tail-f NSO (Network Service Orchestrator)

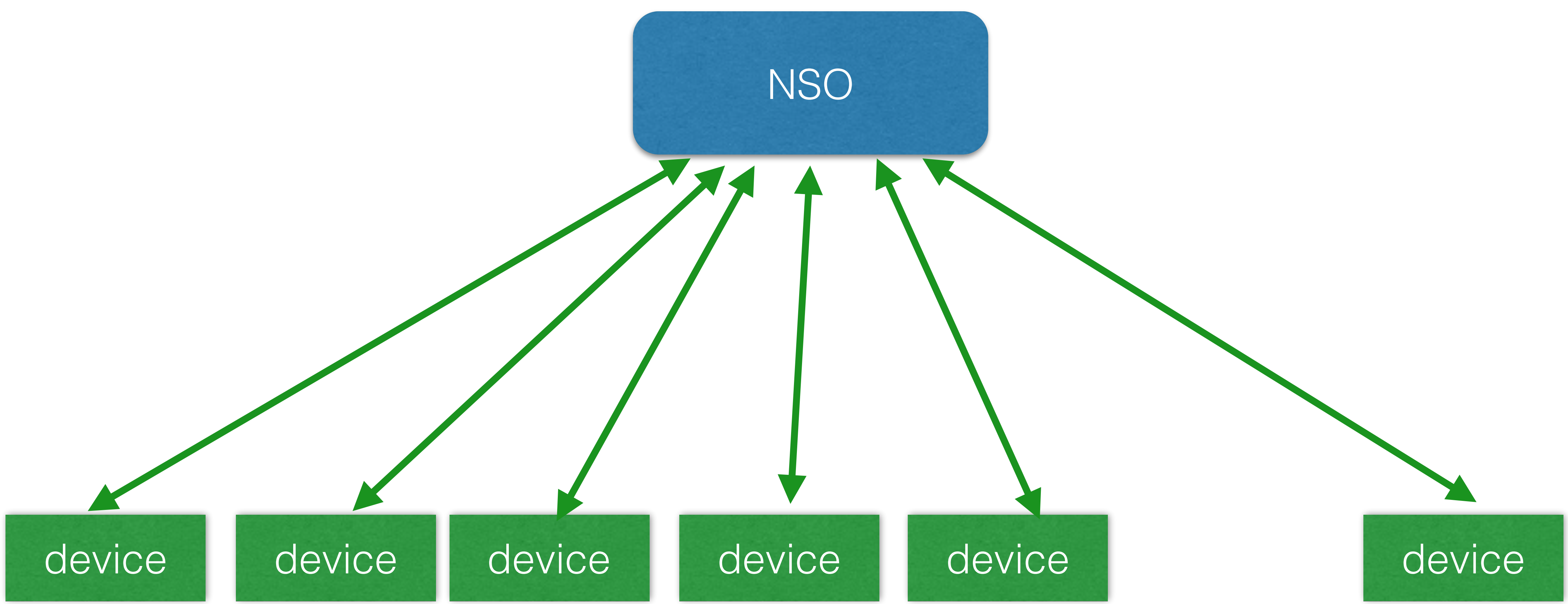
- Typical customer: ISPs, network operators, large enterprises
- Problem solved:
    - configuring services, e.g., a VPN, in networks entails configuring a large number (hundreds, thousands..) of individual devices
    - slow and error prone to do this manually
        - whole network might end up in faulty or unusable state
        - installation of new services can go from weeks to minutes
    - Describe services (with Yang) and reconfigure large sets of devices in transactions
- Uses cdb as well, both to describe the internal state of NSO and the state of the devices it manages

NSO and ConfD

- NSO is the natural generalization of ConfD
- NSO uses standardised interfaces on the devices it manages; these devices are often (already) using ConfD
- If not, a device can be described in Yang together with interface/driver code. NSO uses the Yang model and the driver code communicates with the device.
  - NSO sees it as any “device”
- Shares a large part of code base with ConfD - NSO is essentially a large service written on top of ConfD

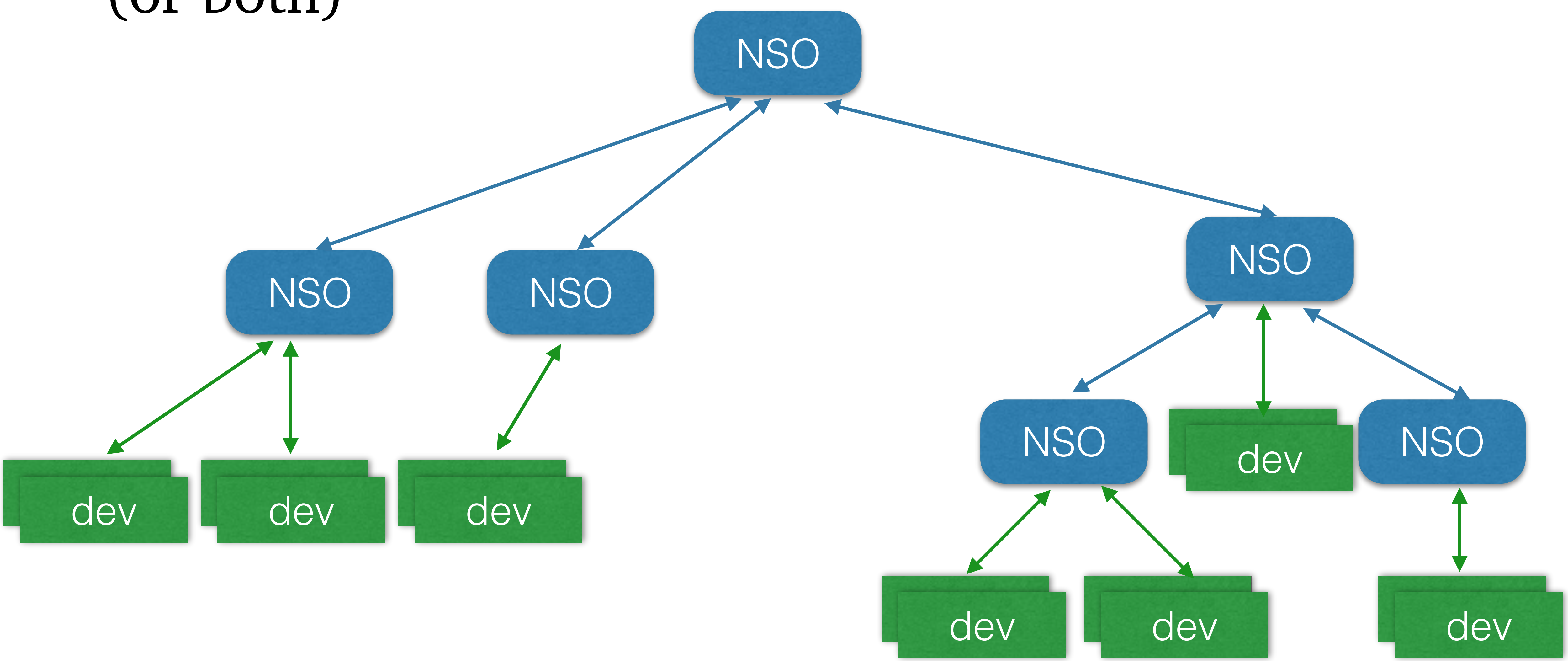
Distribution in NSO

- Large number of devices to manage
- Talk to several devices at the same time
- Concurrency needed more to handle latency rather than parallel computation



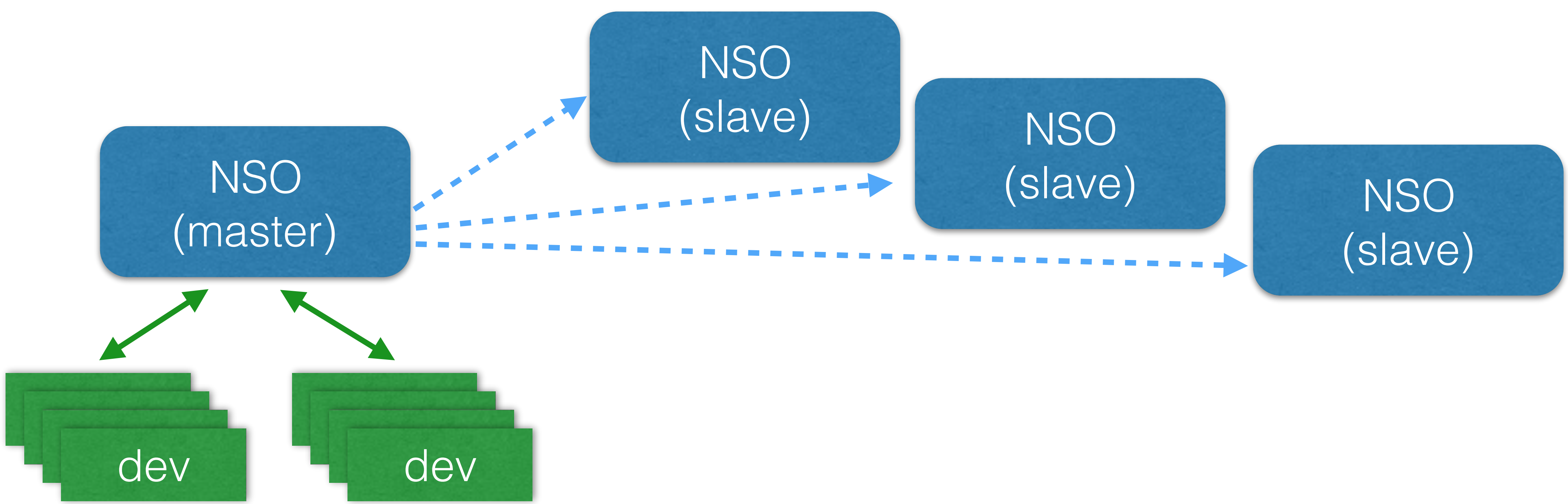
Distribution in NSO

- Large distances, latency, large number of devices
- make clusters of NSO instances - an NSO instance can manage either other NSO instances or devices (or both)



Distribution in NSO

- requirements on high availability (HA)
  - master and several standby slaves
  - a slave will take over when the master fails
  - current state of data model must be distributed from master (read/write mode) to slaves (read only mode)



Distribution in NSO

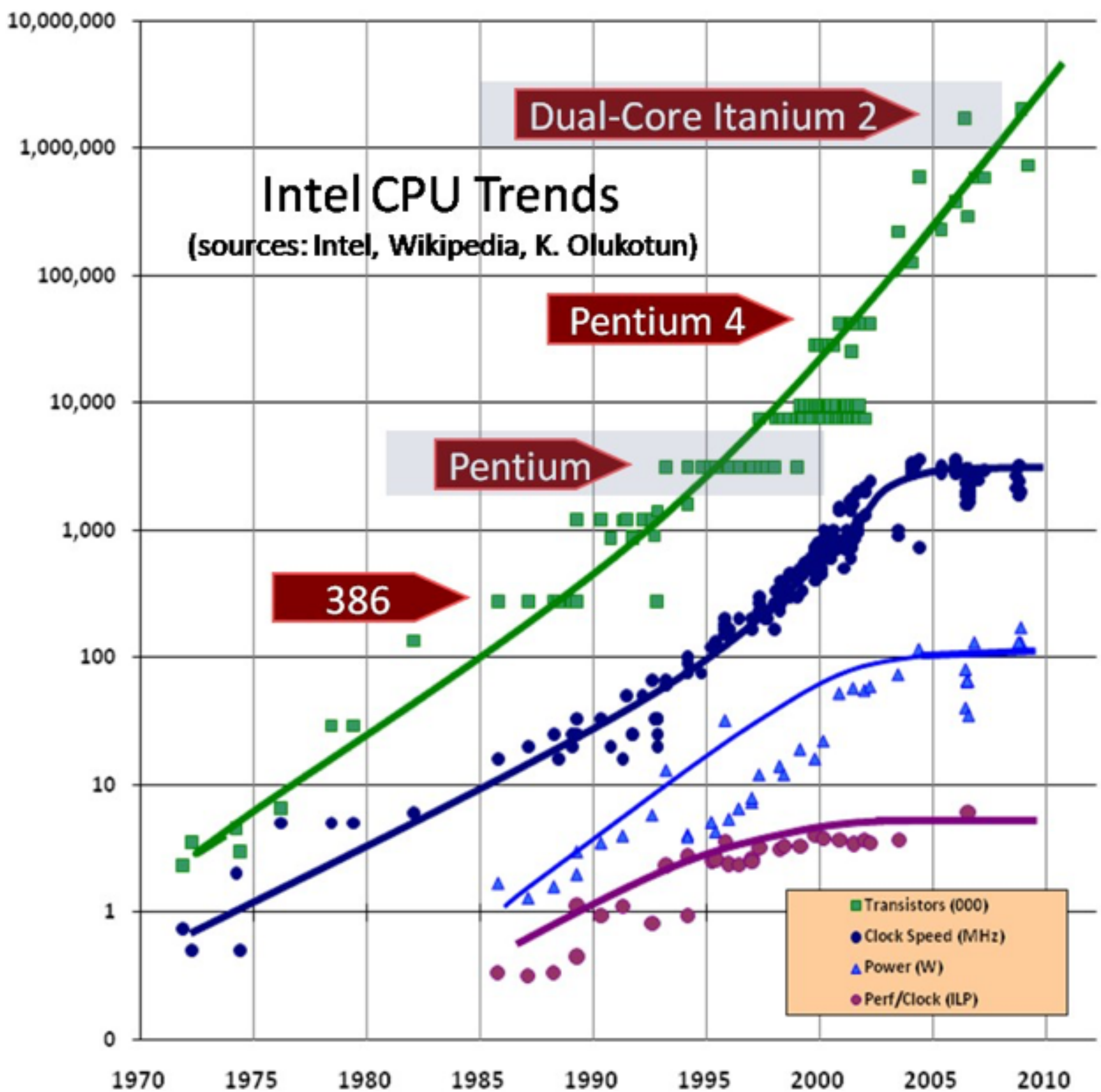
- All modes (multiple devices, clusters and HA) can be combined to form a scalable and robust network management system
- Problems to solve on the development side are
    - data consistency between models on (several) NSO instances and actual devices
    - resource management, e.g., connections and bandwidth
    - customers doing unexpected things

Joe Armstrong says:

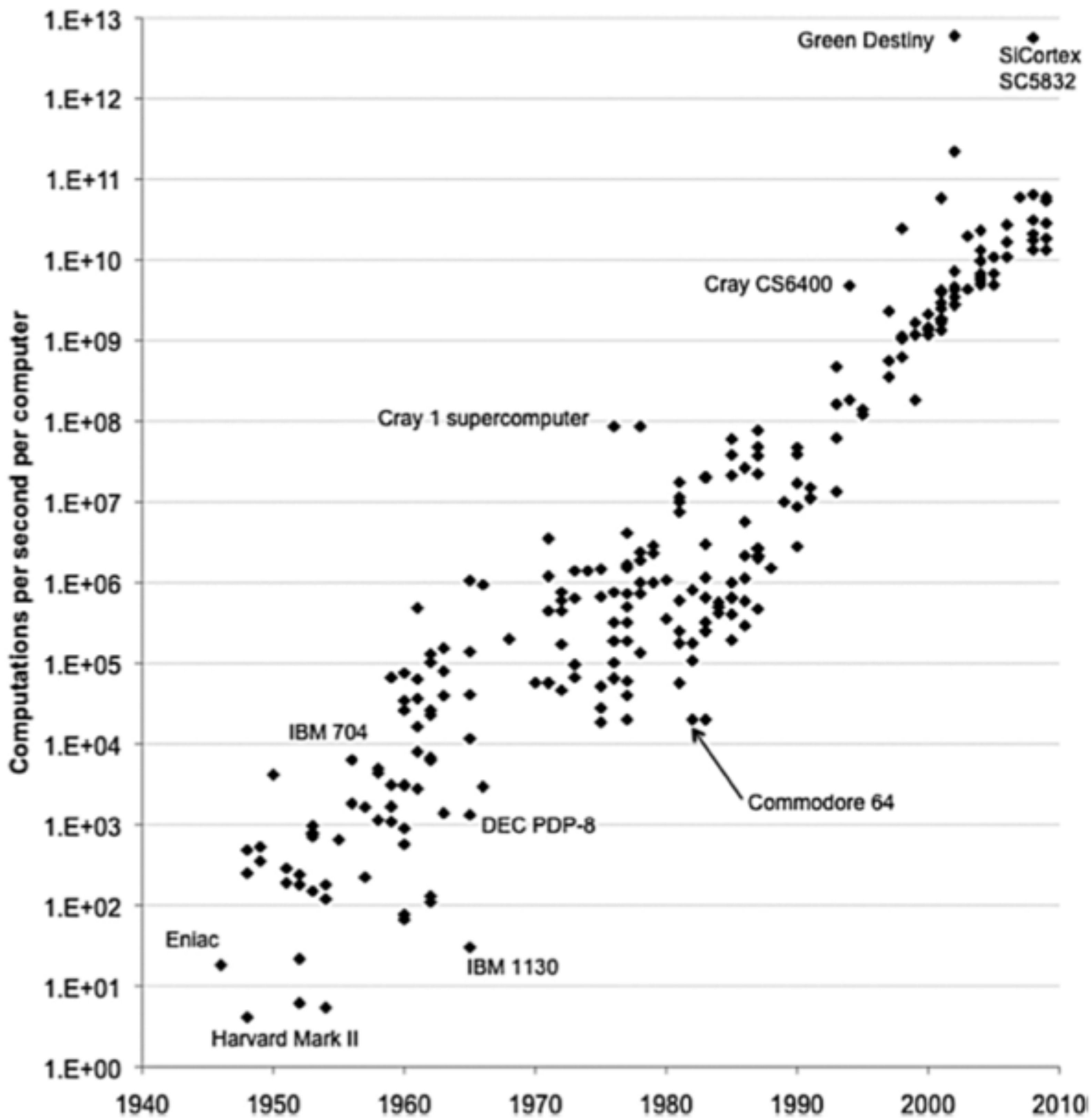
Each year your sequential programs will go slower.

Each year your concurrent programs will go faster.

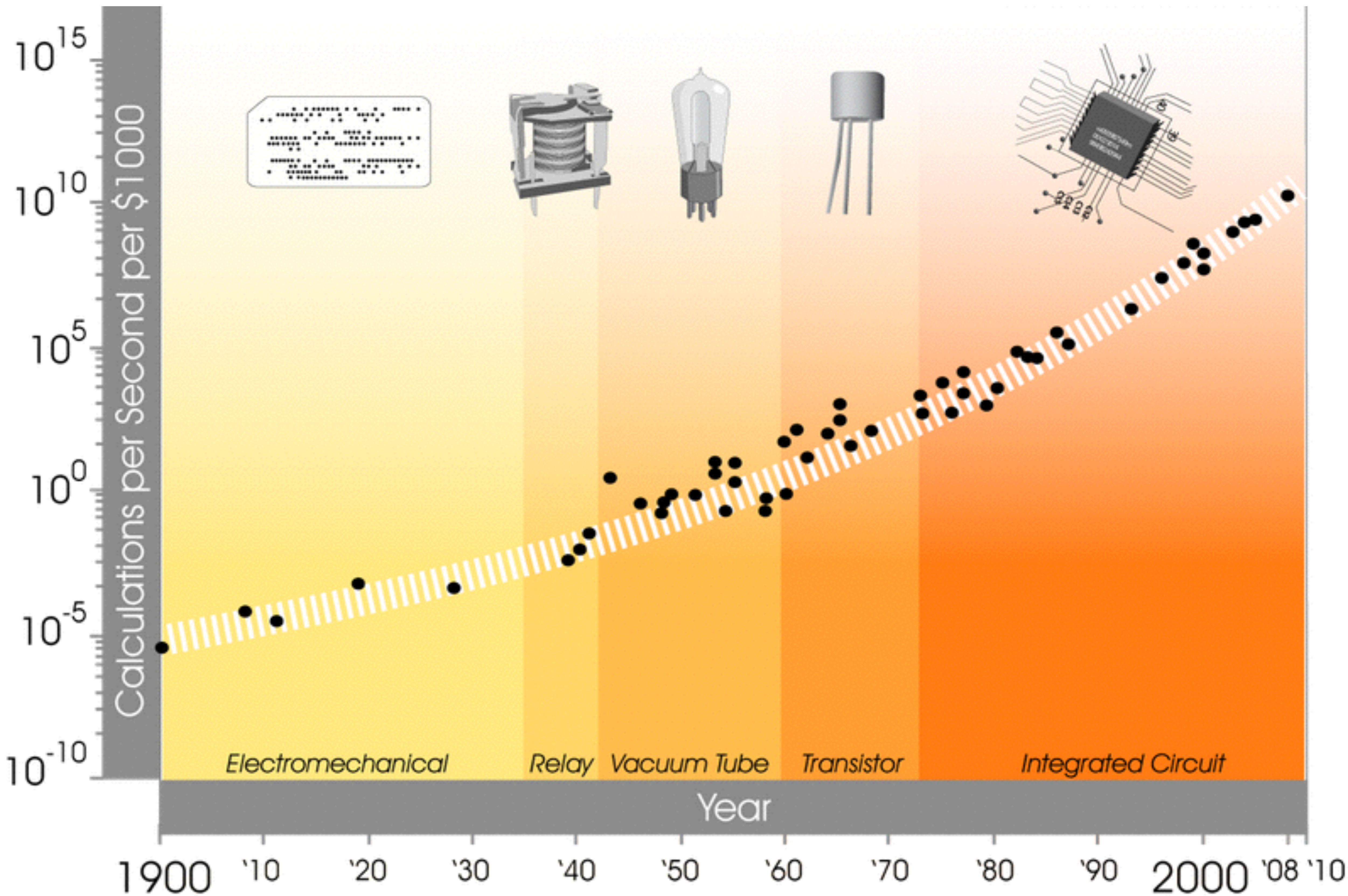
- (Single core) processors are not really getting faster
- Processors grow more cores instead



Moore's law (1965) backwards in time



Computation growth over technological shifts



Humans, on the other hand..

- The cost of developers are growing
- Compare the amount of computational power that can be bought for a man month over the years.
- Are we really getting more productive over the years?
    - Faster machines do not enable us to think faster
    - We can reuse more (due to sheer availability), but find the right tool is time consuming and the fit might nor be perfect anyway
    - Better tools and languages are probably more of an enabler than faster hardware.
    - Faster hardware is an enabler for the better tools and languages
    - Higher level languages with simple semantics and no low level memory management are very attractive

Brace yourself
more cores are coming..

- Sequential programming is not enough
- What you need:
  - A language that supports threads and/or processes
  - Proficiency at writing concurrent and distributed programs
- Every interesting language today supports multithreading
  - Actual model differs
  - Different models give rise to different problems
  - Code that works is more interesting than fast non working code

Distribution and concurrency at different levels

- Concurrency on single core machines
  - time sharing, OS scheduled processes
- Multi processor machines
- Distribution among machines
- Multi core processors
- Combinations of above

Potential problems with concurrent and distributed programming

- Shared **and** mutable state
  - low level problems, low level solutions
  - can happen on single core processors
- Time
  - computations can literally happen at the same time
  - if computations happen on different nodes, how do you know which was first?
  - consistency of data view on different nodes
- No general solutions - depends on domain

Why Erlang?

- Erlang is just like any other tool or language
- Choice of language/technology is seldom a deep process
- Stronger correlation to the developers than the problem, especially if the set of developers is given from the start (the comfort zone)
- Choose the language that you feel most comfortable with!
- There are good and bad fits in terms of language and problem
- Erlang is a good fit for networking, but not an obvious good fit for configuration
- The founders of Tail-f had an extensive knowledge of Erlang

Erlang - the good parts

- Small language (in terms of language definition)
  - Single assignment
  - Functional sequential semantics
  - Higher order functions
- Beam - the virtual machine is an extraordinary work of technology
  - Start fast and small
  - Grow large and robustly handle a very large number of processes and very large memory spaces

Erlang - the good parts

- Process semantics
  - Few and simple primitives
    - `spawn`, `send`, `receive` and `link`
  - *No* shared or mutable state - take away *both* sources of problems
  - A process starts fast and small, but can grow very large
- Memory management
  - Each process has its *own* memory - simple life cycle management

Erlang - the good parts

- Hot code loading
  - Nice for systems that have a low tolerance for downtime, e.g., telecom and financial systems
- Easy distribution from the start
  - makes scaling “simpler” - it is never trivial, especially when you need to maintain state across several instances
  - Recall Brewer’s CAP “theorem” (Consistency, Availability, Partition tolerance - choose any two (at most))

Erlang - the good parts

- OTP - Open Telecom Platform
  - not really part of Erlang - you can avoid using/loading (parts of) OTP, but will probably end up rewriting at least parts of OTP anyway
  - set of libraries and utilities
    - add generic components, e.g., `gen_server`, with behaviours
  - robust and battle proven

Erlang - the bad parts

- The syntax
  - ever mix up , ; . ?
  - awkward syntax for closures/lambdas/funs (whatever you want to call them)
  - leftover from the initial implementation i Prolog
- No real strings (also a leftover from Prolog)
- The broken if..
  - admittedly one of the least used constructs in Erlang
- No scoping rules or actually just one - the whole clause! (often leads to hard to find bugs)
- Being a dynamically typed language, static type checking is a very difficult problem
  - dialyzer exists, but results in the a type checker generating a type system which is not always consistent with runtime behavior

Erlang - the bad parts

- libraries are inconsistent - evolution vs design..
- No obvious support for abstraction, with some support bolted on afterwards, e.g., records and maps
- “too easy” to build complex applications fast
  - technical debt might build fast by using libraries causing too tight coupling
  - causes large problems later
  - normal software engineering principles still apply

Interesting problems for distributed 24/7 systems

- [CAP] Which two of consistency, availability and partition tolerance do you focus on?
- How do you upgrade (software or hardware) your system without being unavailable? [No downtime allowed]
  - Lifespan of system larger than individual components
- How do you physically relocate your system without being unavailable? [no downtime allowed]
- How do you change your persistent representation (database) without being unavailable? [no downtime allowed]
- How do you design your system architecture to be failure resistent and without domino effects?

An interesting problem in parallelisation and distribution

- Background:
  - When communicating with several devices you want to do it in parallel, not sequentially
  - We had code for this, but we wanted to make it better in terms of behaving better on crashes and timeouts (this is reality)
  - During this we found bugs in the parallel utilization - it was lower than expected and sequential in the extreme
- Having the collection of devices as a list and mapping a function over them is a reasonable and simple model.
  - Also accurate - this is what we use

Sequential map

- Sequential version

```
map(_F, []) -> [];
map( F, [E | Es]) -> [F(E) | map(F, Es)].
```

- Simple and straight forward
  - Will be “slow” on multi core machine by using only one core
  - Make it “faster” by utilizing more cores

Parallel map (straightforward solution)

```
pmap(F, List) ->
  I = self(),
  S = fun(E) ->
          spawn(fun() -> I ! {self(), F(E)} end)
      end,
  C = fun(Pid) ->
          receive {Pid, Res} -> Res end
      end,
  lists:map(C, lists:map(S, List)).
```

- Spawn one process for each element (returns pid of newly spawned process "directly")
- Collect results in same order using selective receive
- [No, you can't do both maps at the same time - why?]

Parallel map

- Naive or troublesome
  - Directly spawns one process for each element - this will be wasteful when the number of elements is large (`length(List) >>` N_{cores}) and depending on what is done for each element
  - What happens when a process for an element crashes? It's lost and the initial call to `pmap/2` will never return since the result is never sent.
  - Selective receive hides complexity (simple code, but need to search mailbox for every run) [not really a big issue]
  - Does order of results really matter?

Parallel map

- Write a parallel map that
  - can impose resource restrictions, i.e., the number of processes run in parallel
  - utilises maximum parallelism in the light of the above restriction, i.e., only slack off when there are few results left
  - handles processes that crash in a reasonable way, i.e., at least does not hang, but might also make it possible to distinguish between successful and crashed processes
  - lets the user determine if the order of the results matter, i.e., either return results in the same order as the initial list or in the order they arrive

Questions?