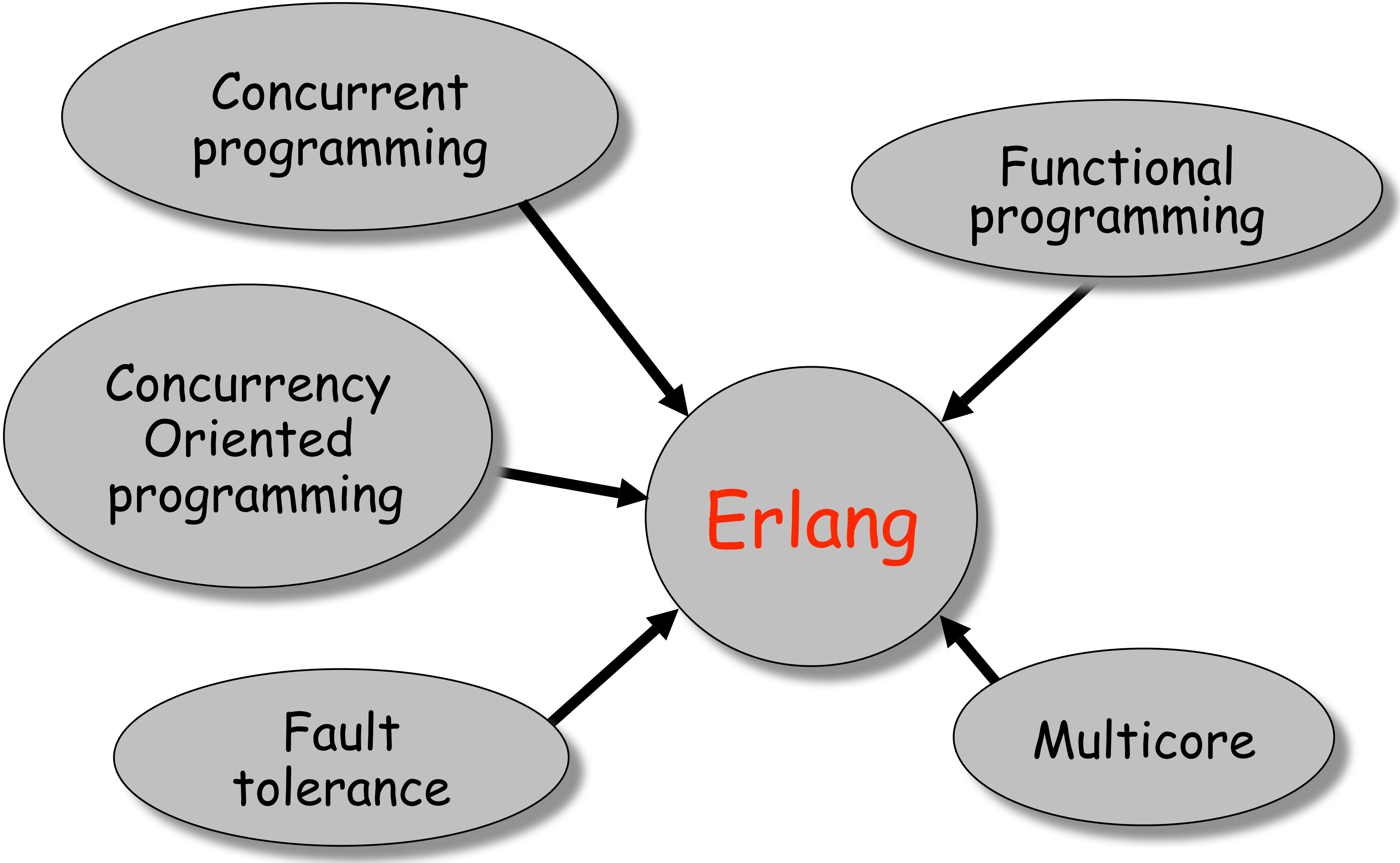
Functions + Messages + Concurrency = Erlang

Joe Armstrong


Concurrent programming
Functional programming
Concurrency Oriented programming
Erlang
Fault tolerance
Multicore

Problem domain

Highly concurrent (hundreds of thousands of parallel activities)

Real time

Distributed

High Availability (down times of minutes/year – never down)

Complex software (million of lines of code)

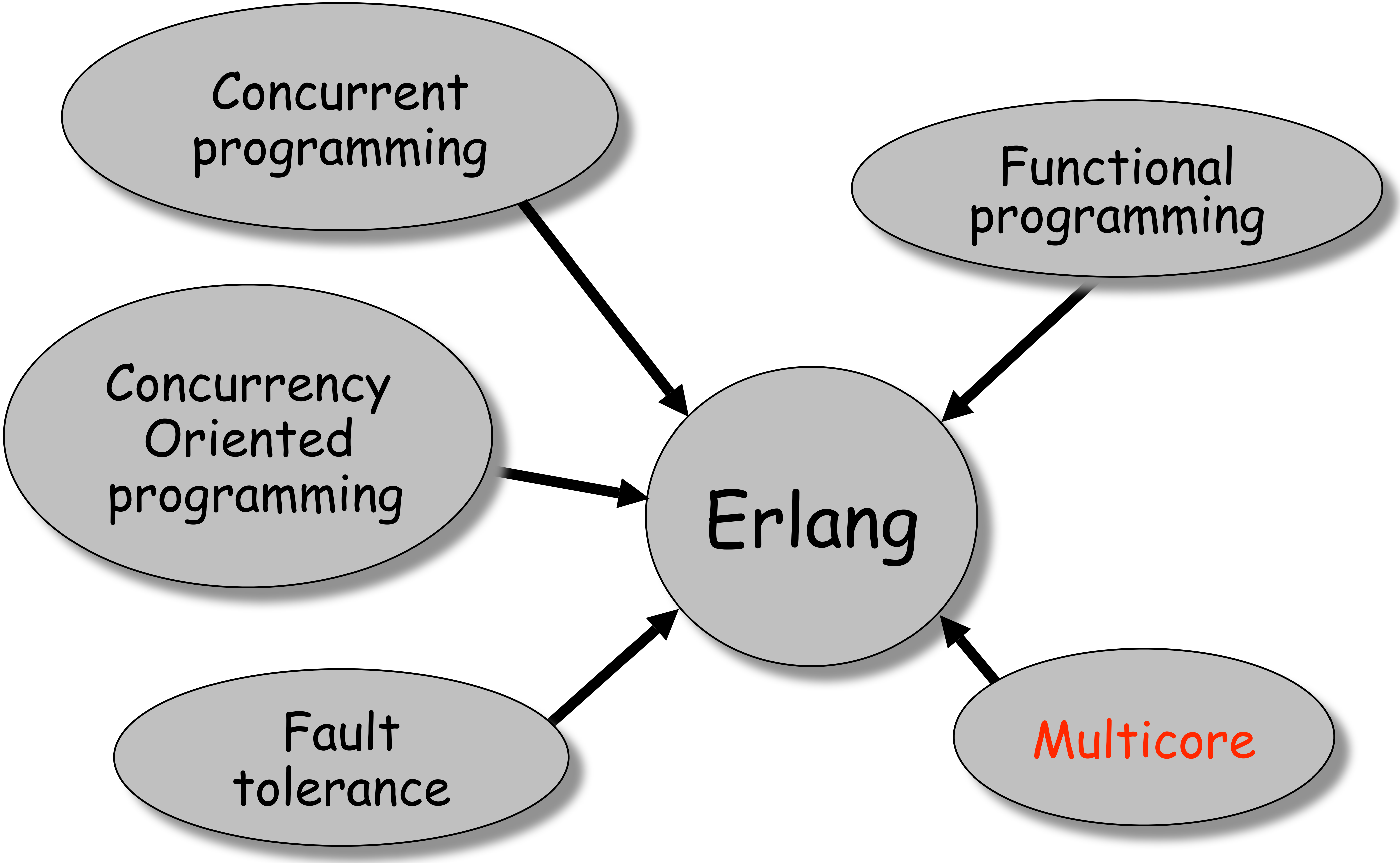
Continuous operation (years)

Continuous evolution

In service upgrade

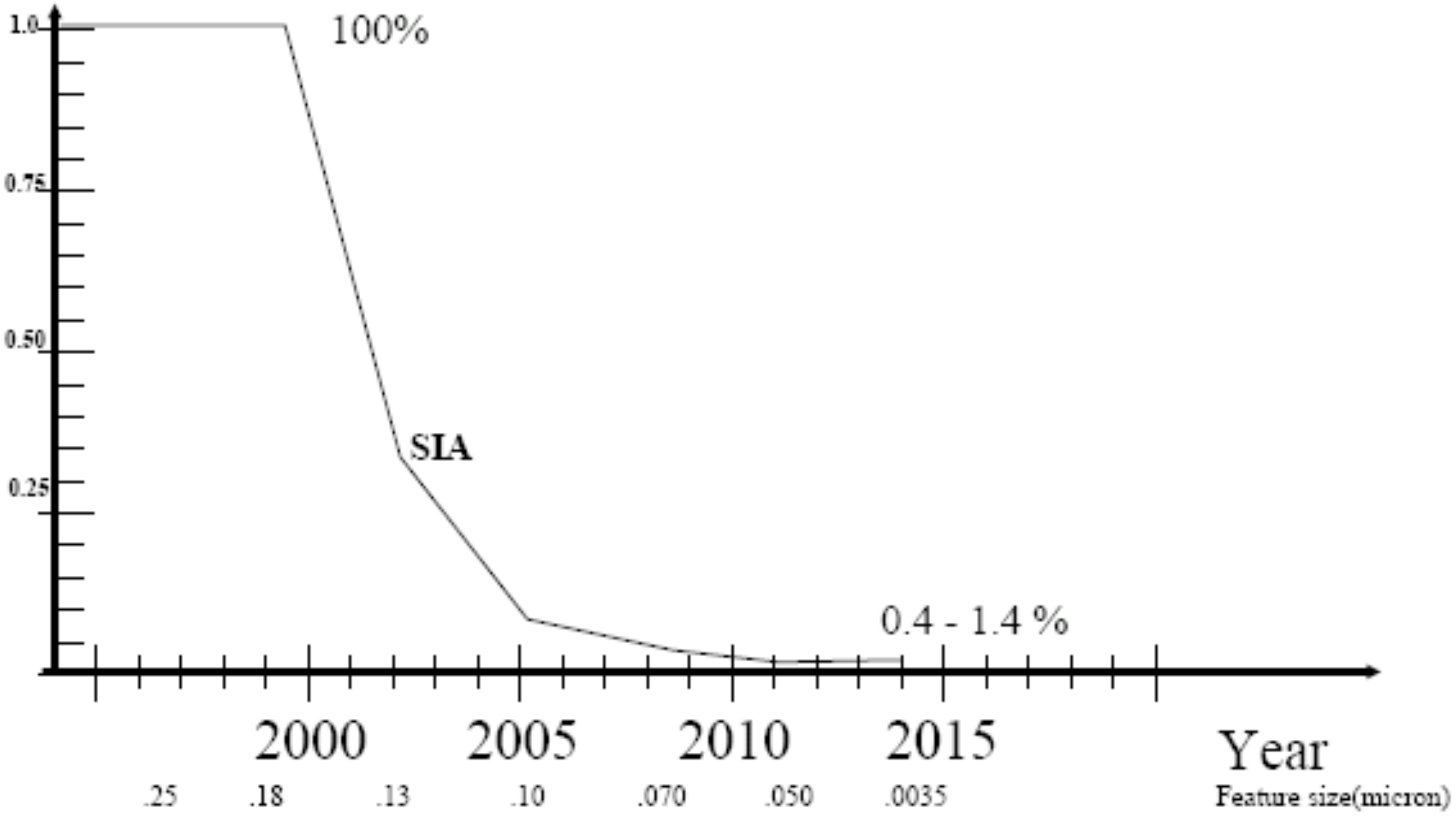
Erlang

- Very light-weight processes
- Very fast message passing
- Total separation between processes
- Automatic marshalling/demarshalling
- Fast sequential code
- Strict functional code
- Dynamic typing
- Transparent distribution
- Compose sequential AND concurrent code


Concurrent programming
Functional programming
Concurrency Oriented programming
Erlang
Fault tolerance
Multicore

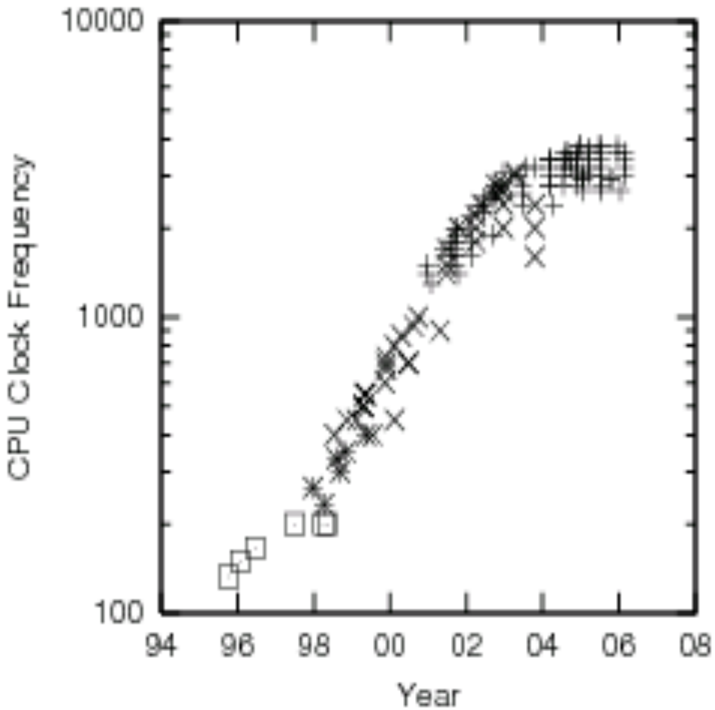
2002

Fraction of Chip reachable in one clock cycle



[source] Erik Hagersten http://www.sics.se/files/projects/multicore/day2007/ErikH-intro.pdf

Clock Frequency



Clock frequency trend for Intel Cpus (Linux Journal)

Read: Clock rate verses IPC. The end of the road for Conventional Microarchitectures. Agarwal et.al 2000

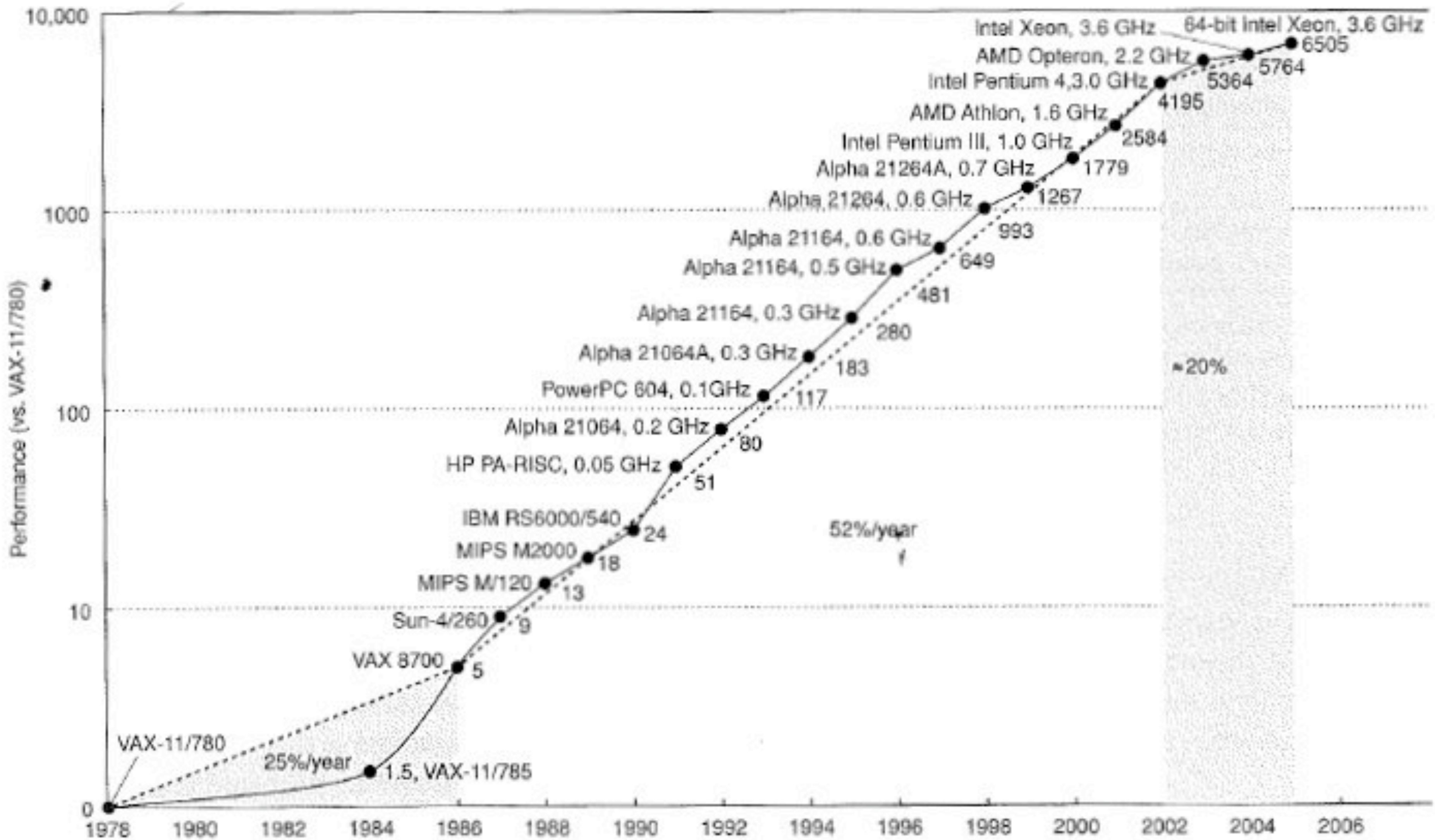


Figure 1.1 Growth in processor performance since the mid-1980s.

Due to hardware changes:

Each year your sequential programs will go slower

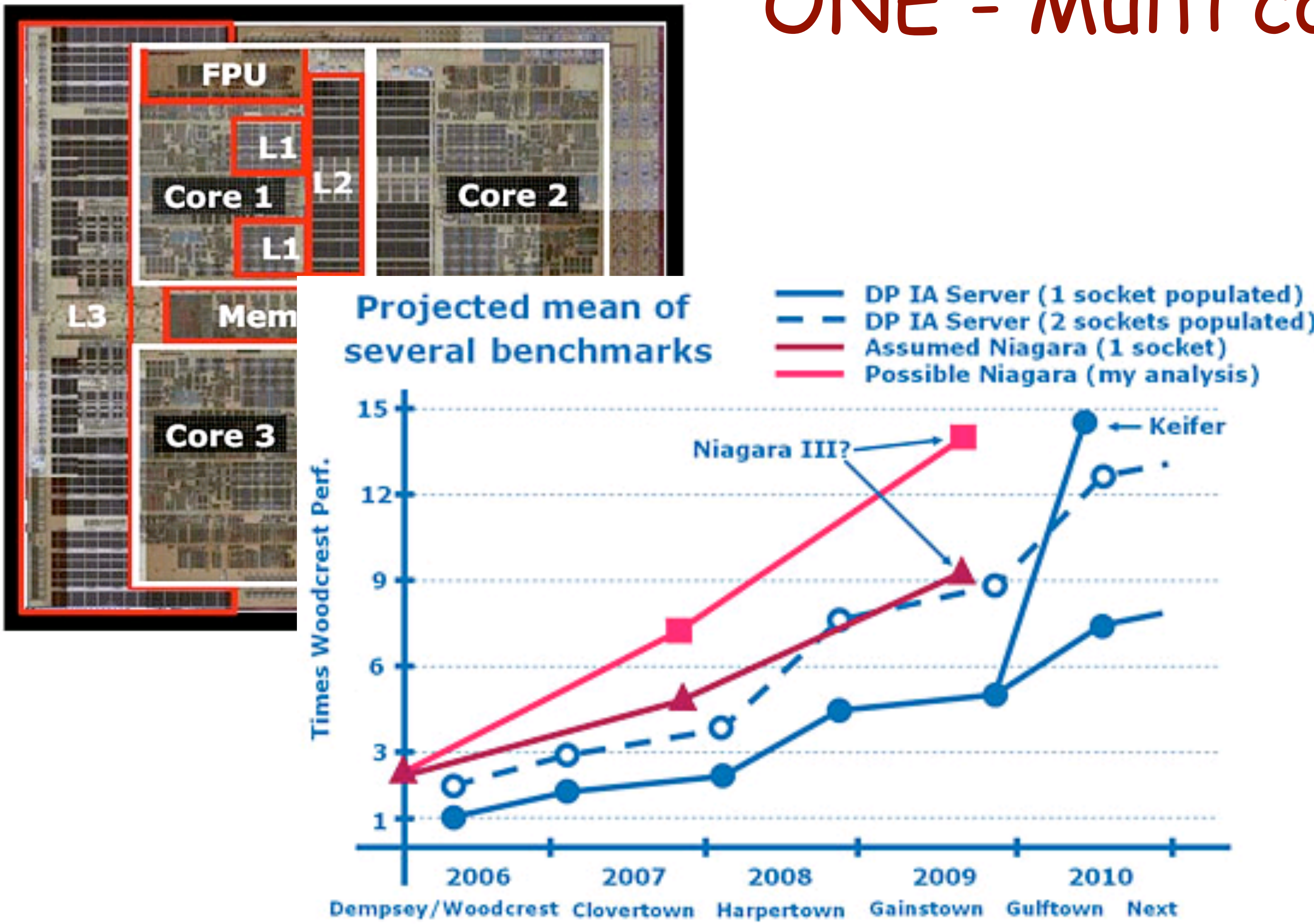
Each year your concurrent programs will go faster

2005 - 2015

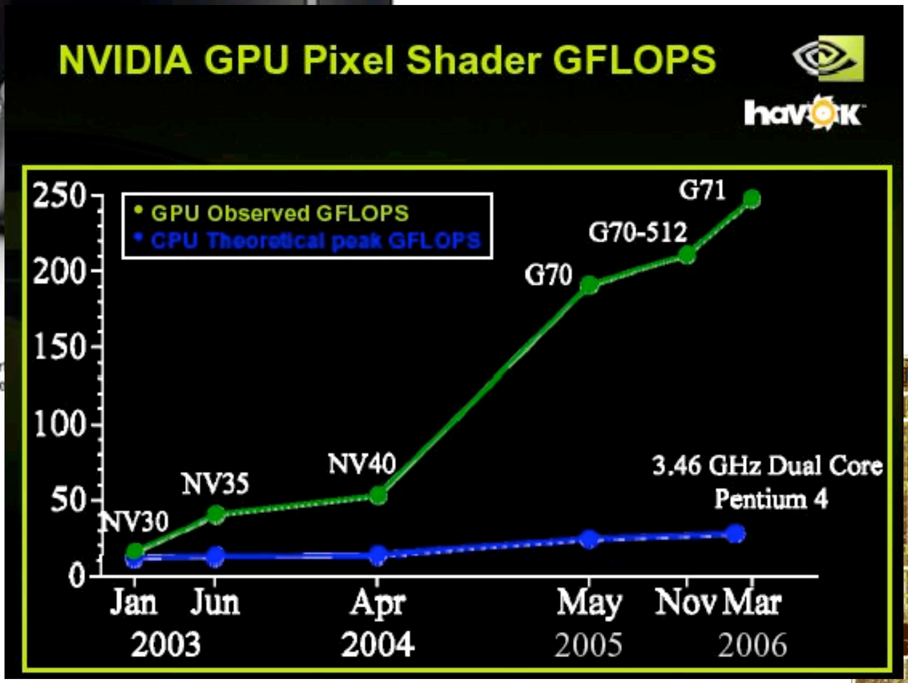
Paradigm shift in CPU architectures

Three New Architectures

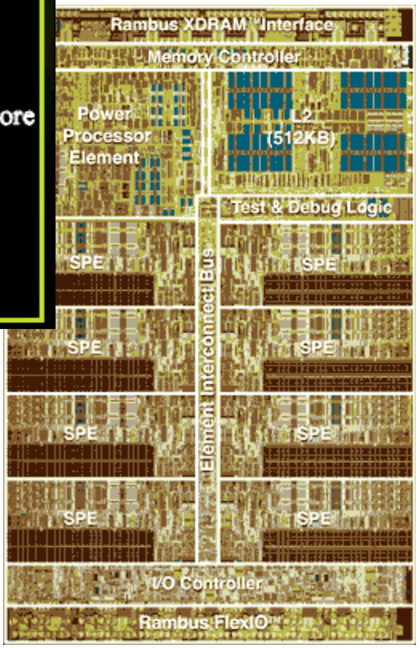
ONE - Multi core



TWO - GPUs



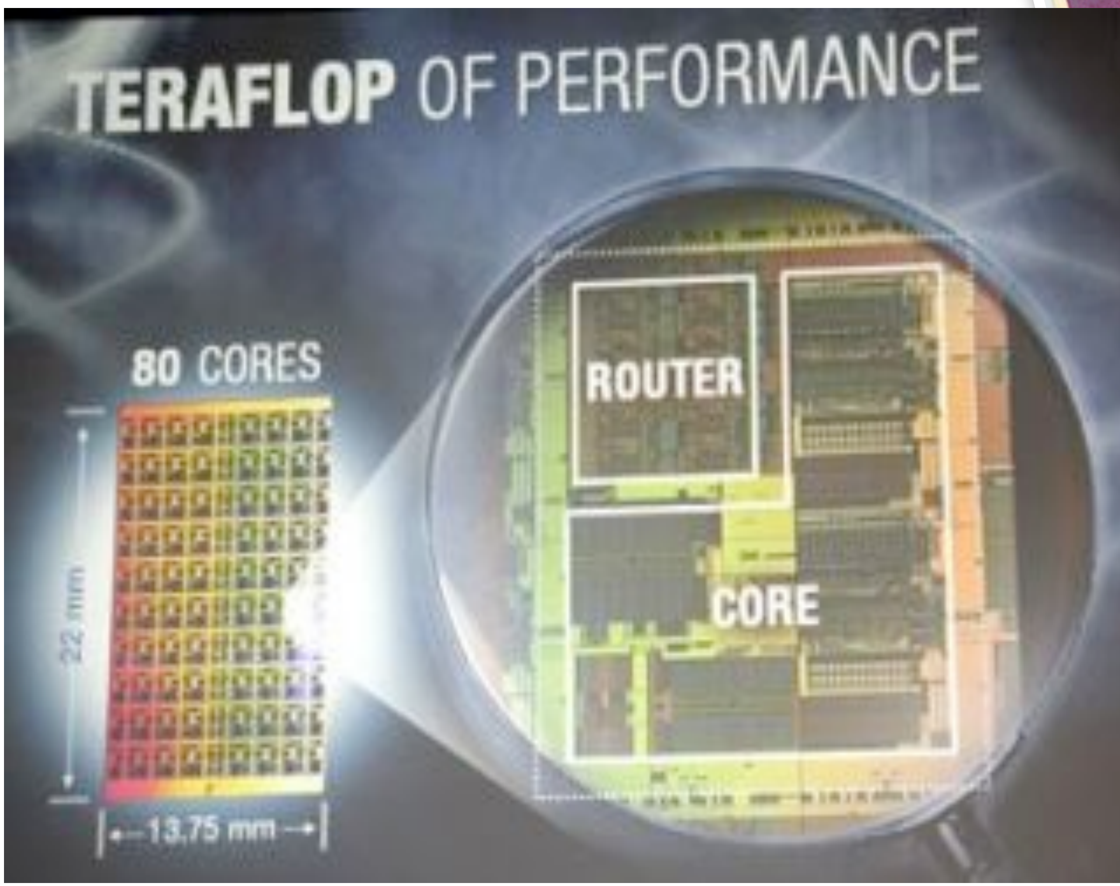
Cell Computers



THREE – network on Chip (NOC)

Intel Polaris – 2007

1 Tflop at 24 Watts



ASCI RED- 1997

- 1997
- First machine over 1 Tera Flop
- 2,500 sq ft floor space 104 cabinets
- 9326 pentium pro processors
- 850 KW

2 cores won't hurt you

4 cores will hurt a little

8 cores will hurt a bit

16 will start hurting

32 cores will hurt a lot (2009)

...

1 M cores ouch (2019)

(complete paradigm shift)

1997 1 Tflop = 850 KW

2007 1 Tflop = 24 W (factor 35,000)

2017 1 Tflop = ?

Goal

Make my program run N times faster on an N core CPU with

- no changes to the program
- no pain and suffering

Can we do this?

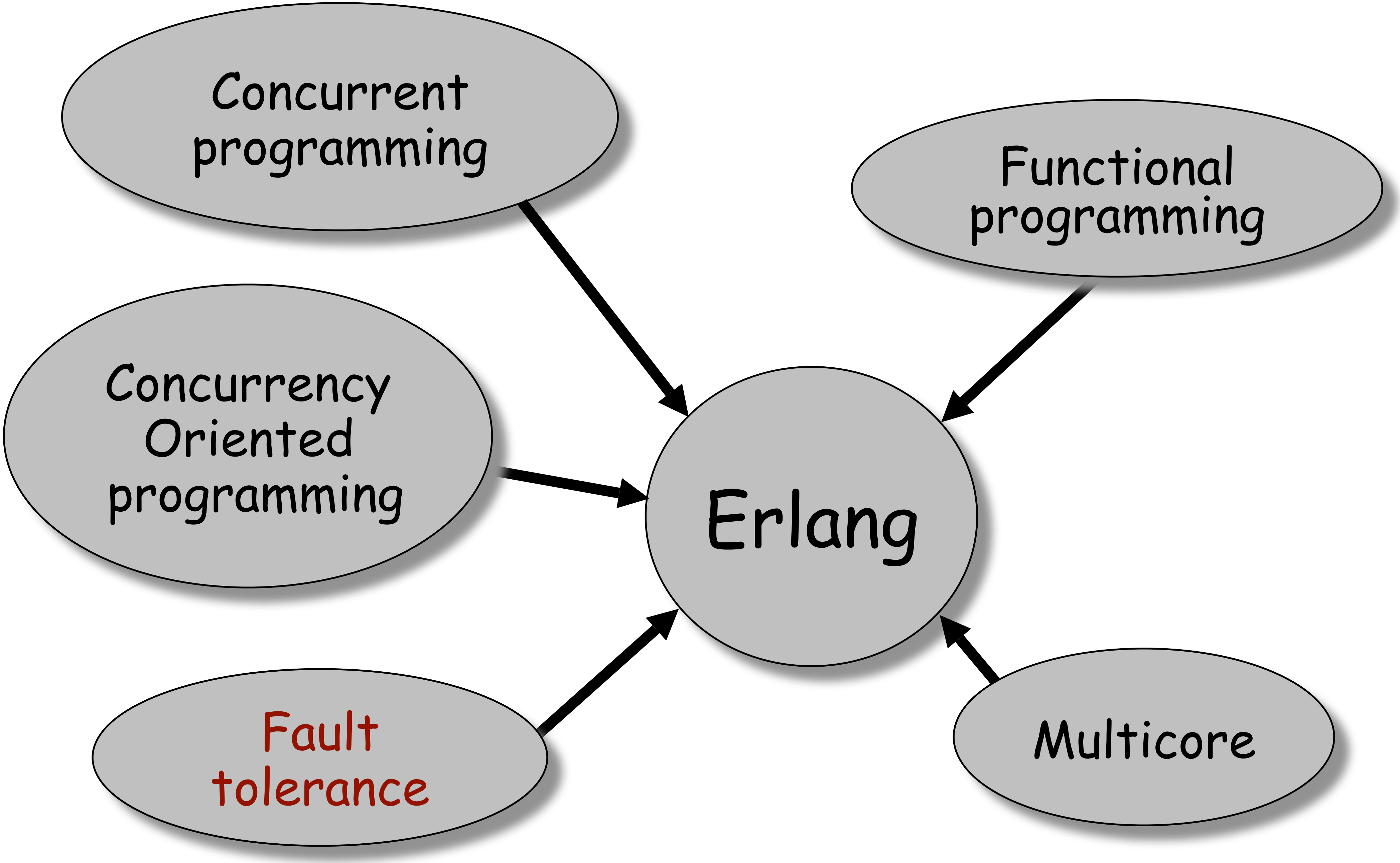
Yes

Sometimes (often)

Due to hardware changes:

Each year your sequential programs will go slower

Each year your concurrent programs will go faster

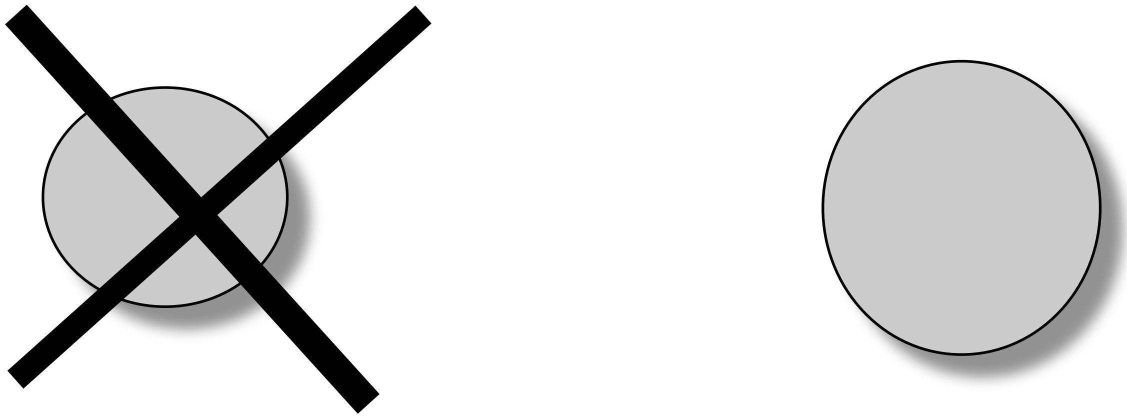

Concurrent programming
Functional programming
Concurrency Oriented programming
Erlang
Fault tolerance
Multicore

To make
a fault-tolerant system
you need at least

two

computers

If one computer crashes the other must take over



= No Shared data

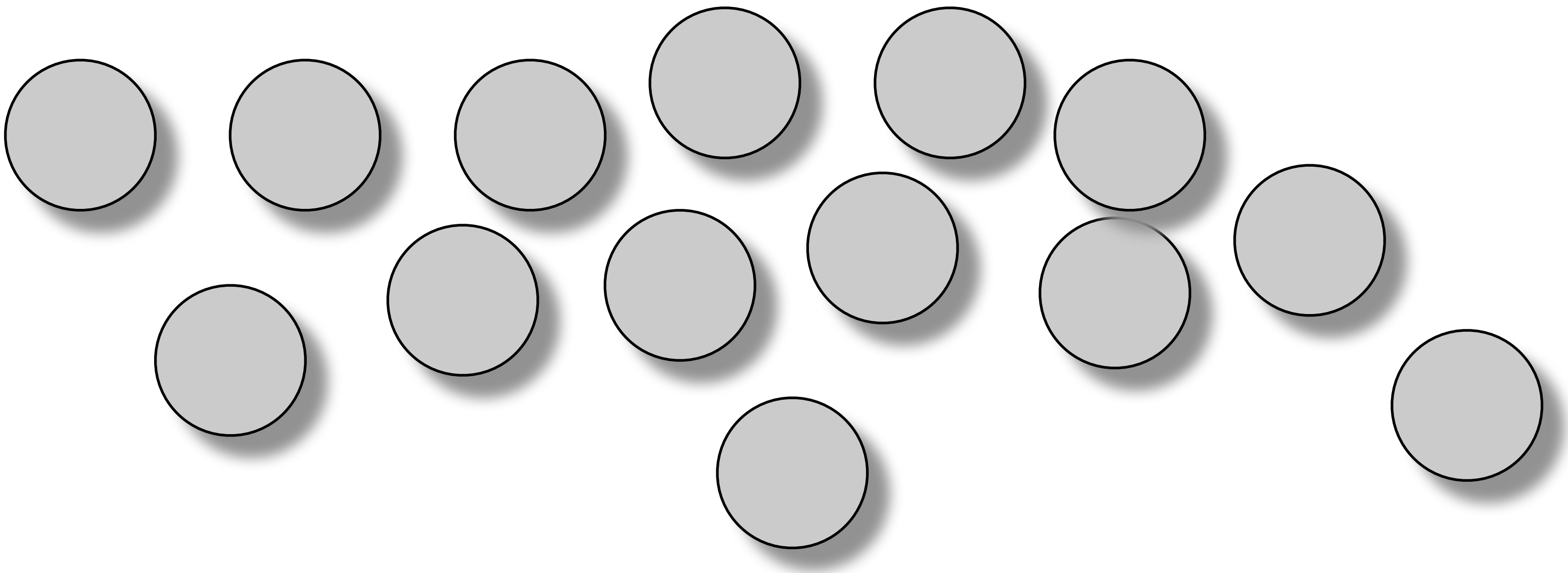
= Distributed programming

= Pure Message passing

To do fault tolerant computing we need at least two isolated computers

= Concurrent programming
with pure message passing

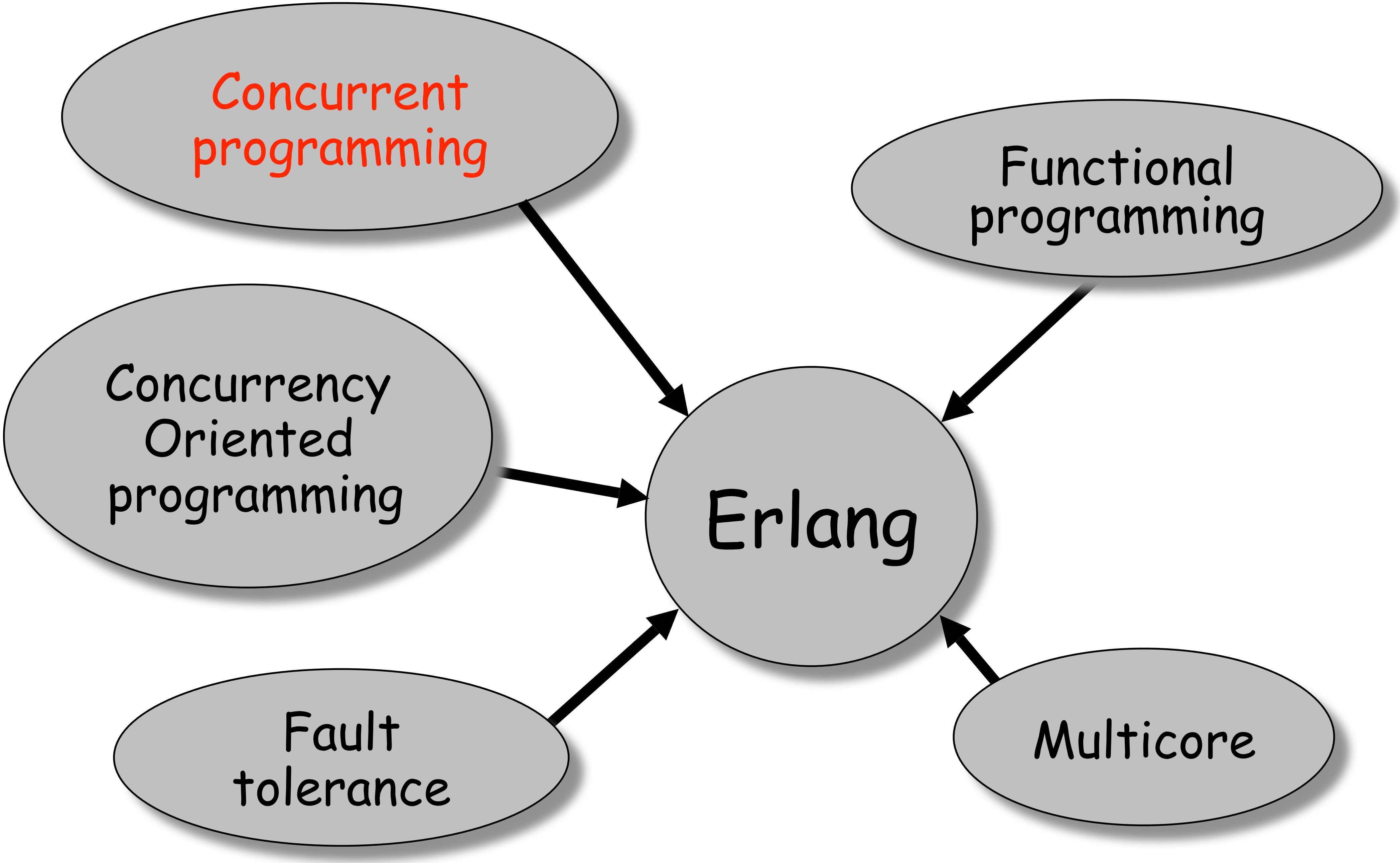
To do very fault tolerant computing we need lots of isolated computers



= Scalable

Fault tolerance
Distribution
Concurrency
Scalability

are inseparable


Concurrent programming
Functional programming
Concurrency Oriented programming
Erlang
Fault tolerance
Multicore

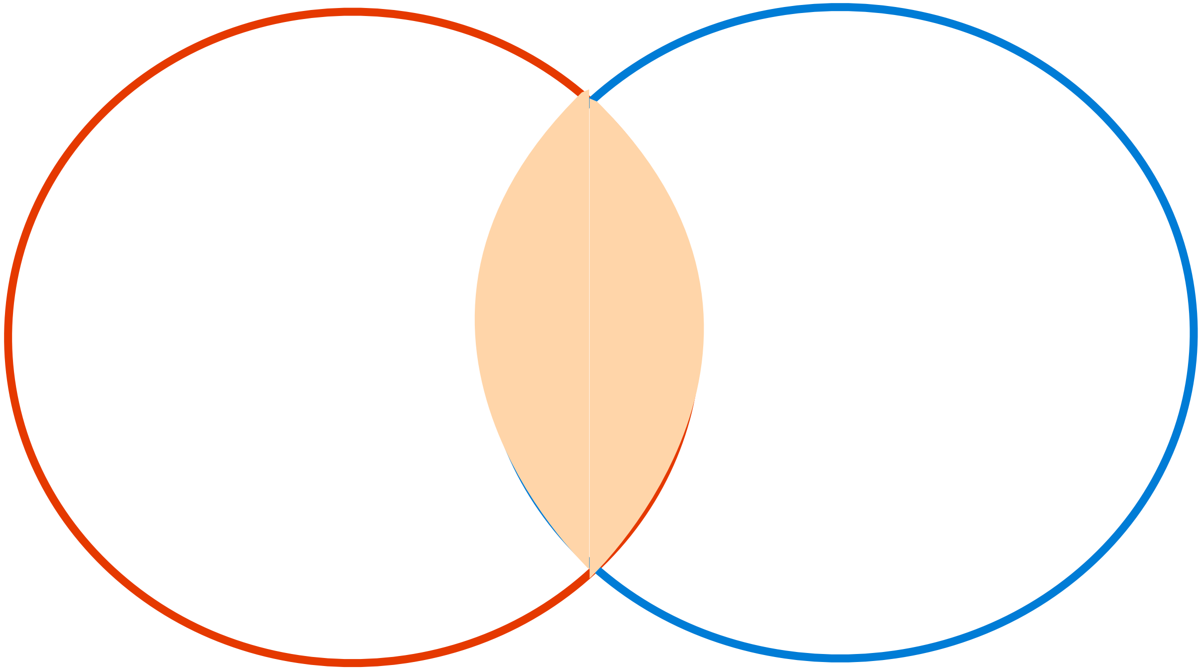
Two models of Concurrency

Shared Memory
- mutexes
- threads
- locks

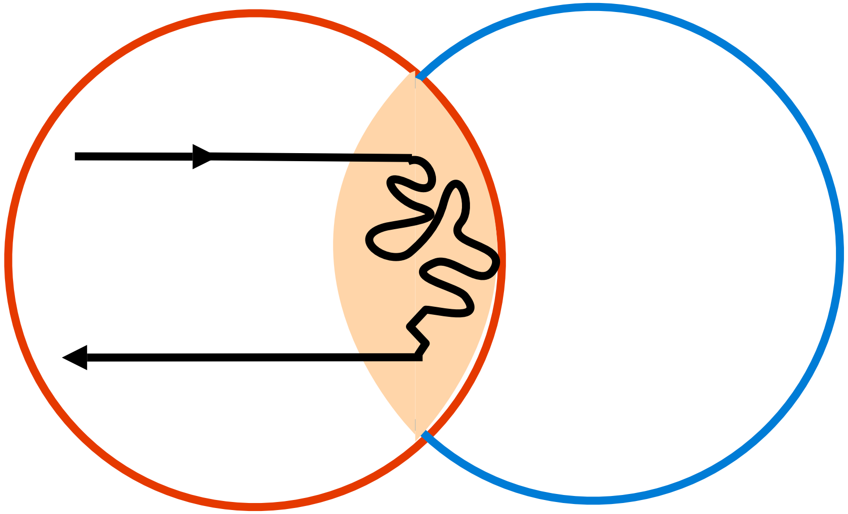
Message Passing
- messages
- processes

Shared Memory Programming

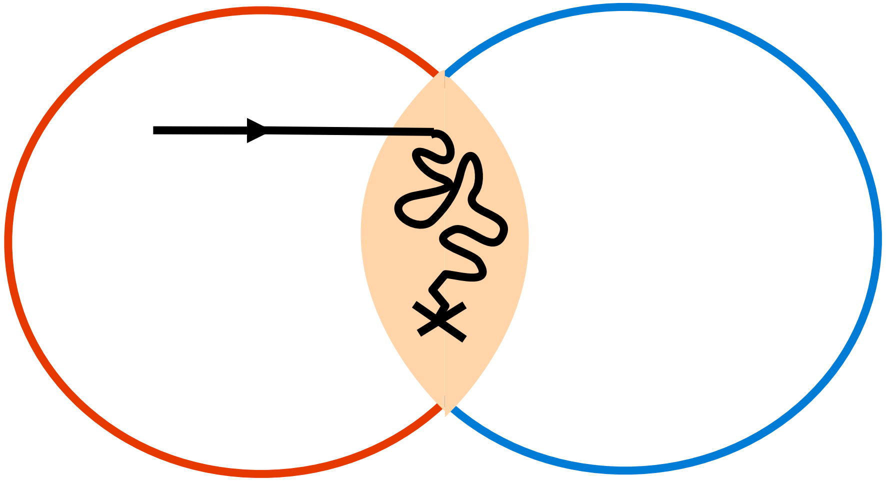
Shared memory



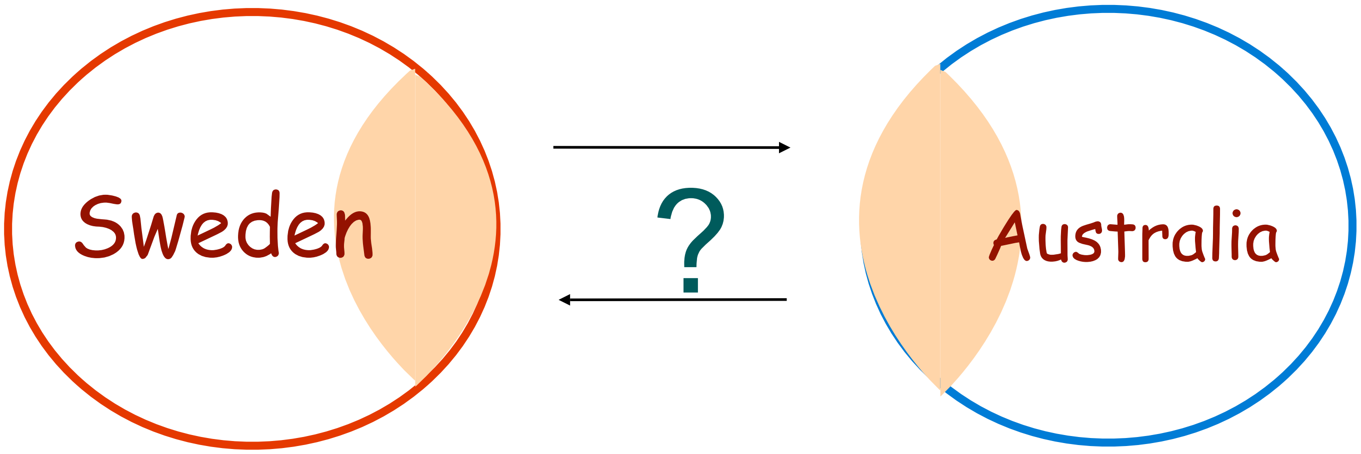
Problem 1



Your program crashes in the critical region having corrupted memory



Problem 2



Where do we (physically) locate the shared memory?

Impossible to get low-latency and make consistent (violates laws of physics)


HARDENED
NINE
NINE


HARDENED
NINE
NINE


skullandbone.com


HARDENED
NINE

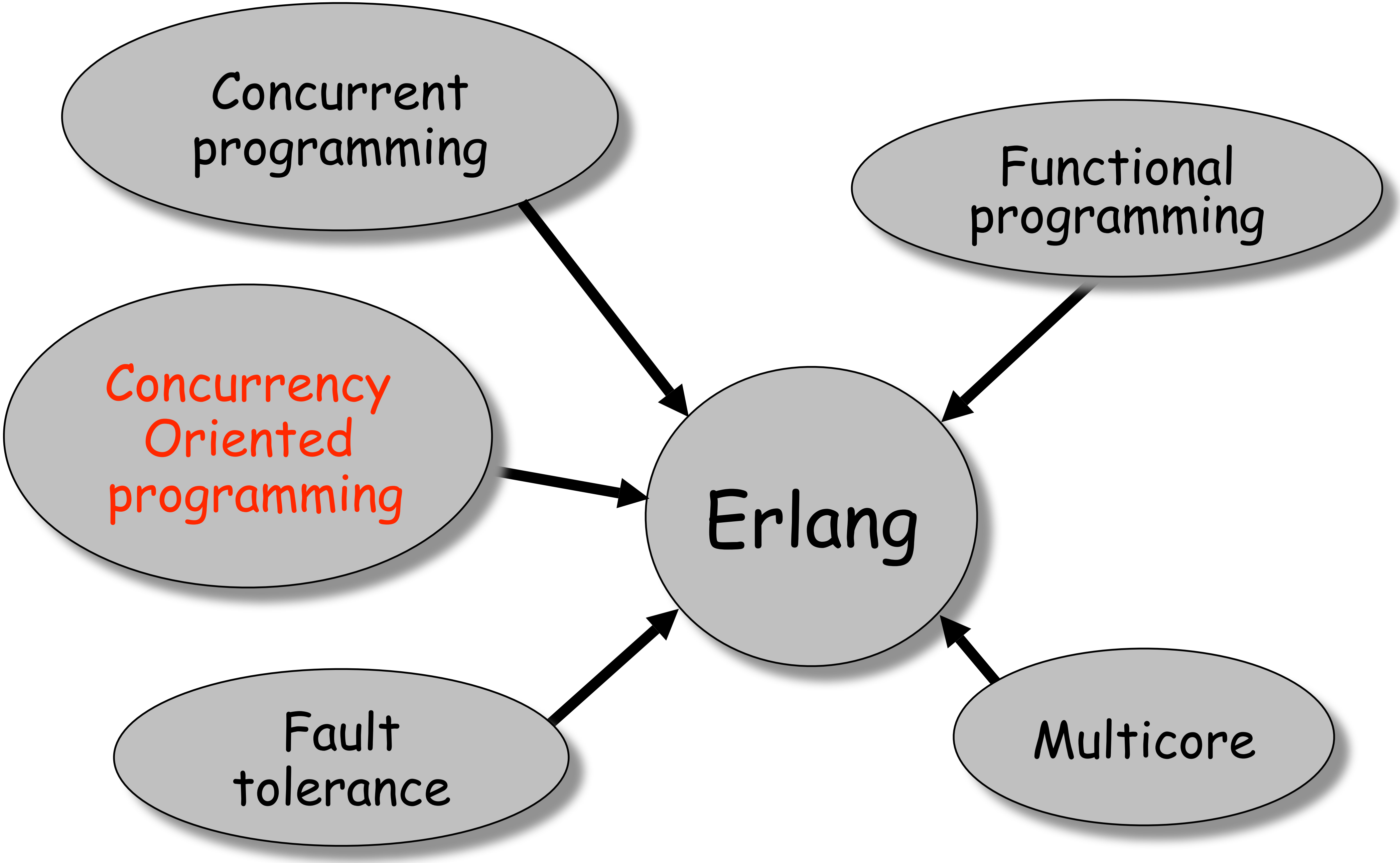
Thread Safety

Erlang programs are automatically thread safe if they don't use an external resource.

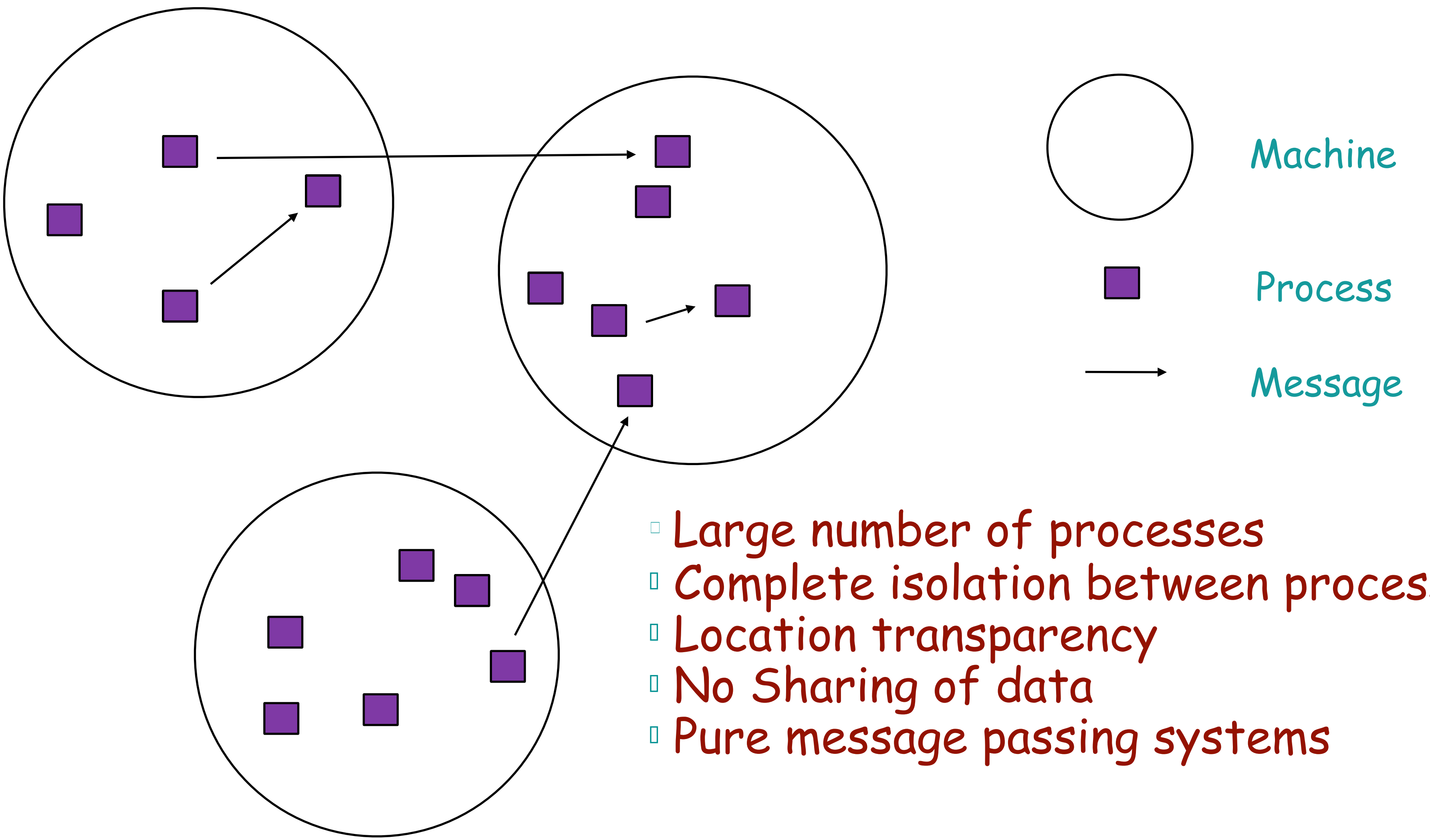
Sharing is the property that prevents fault tolerance and Thread safety

Message Passing Concurrency

No sharing
Pure message passing
No locks
Lots of computers (= fault tolerant scalable ...)
Functional programming (no side effects)


Concurrent programming
Functional programming
Concurrency Oriented programming
Erlang
Fault tolerance
Multicore

What is COP?



- Large number of processes
- Complete isolation between processes
- Location transparency
- No Sharing of data
- Pure message passing systems

Why is COP nice?

- We intuitively understand concurrency
- The world is parallel
- The world is distributed
- Making a real-world application is based on observation of the concurrency patterns and message channels in the application
- Easy to make scalable, distributed applications

Concurrency Oriented Programming

- A style of programming where concurrency is used to structure the application

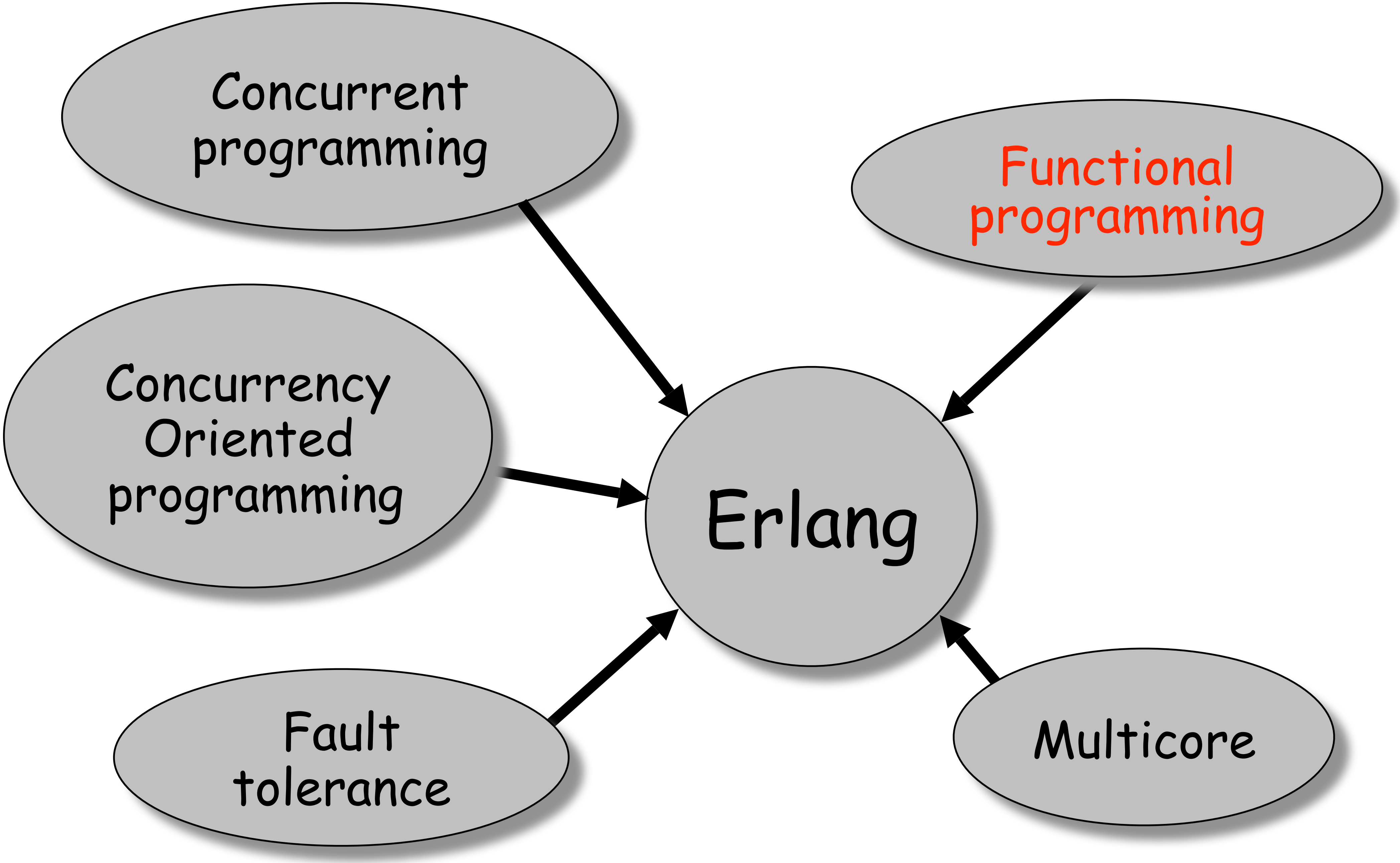
- Large numbers of processes
- Complete isolation of processes
- No sharing of data
- Location transparency
- Pure message passing

> My first message is that concurrency
> is best regarded as a program structuring principle"
>
> Structured concurrent programming
> – Tony Hoare

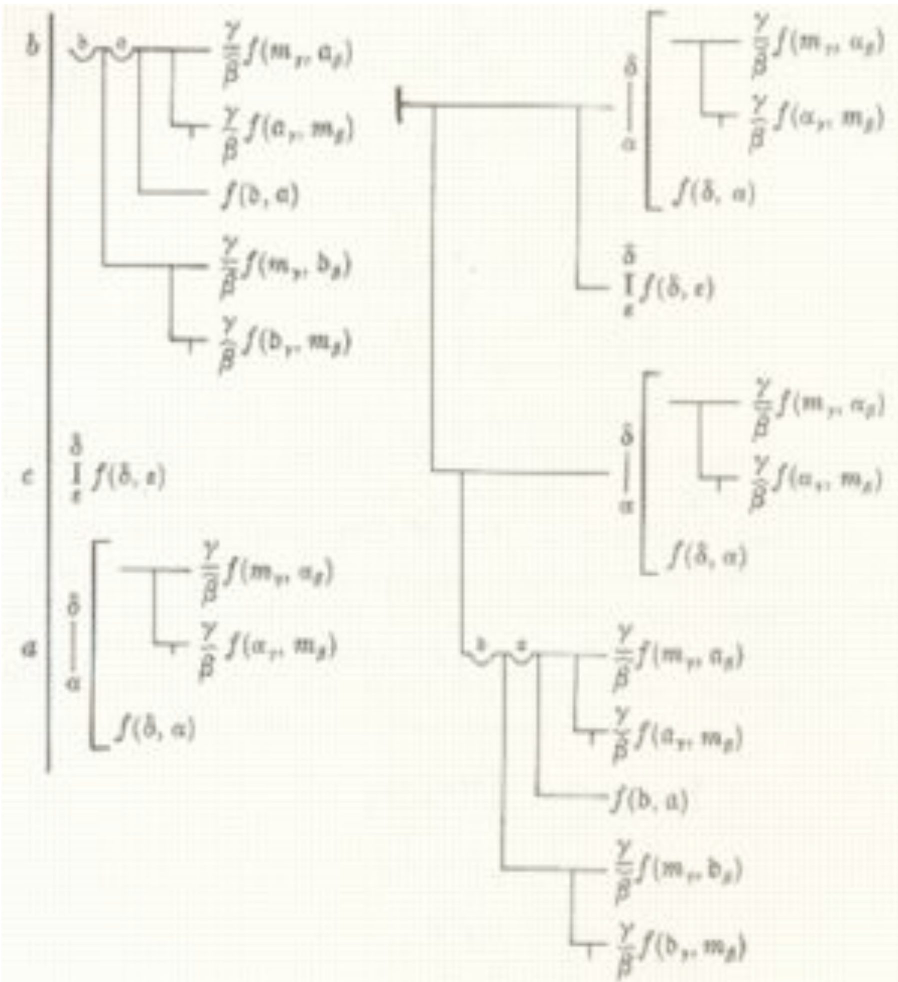
Examples of COP architectures

remember - no shared memory
- pure message passing

Email
Google - map - reduce (450,000 machines)
People (no shared state, message passing via voiceGrams, waving arms, non-reliable etc.)


Concurrent programming
Functional programming
Concurrency Oriented programming
Erlang
Fault tolerance
Multicore

Functional programming



Scary stuff

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\{z : B \wedge A\} \vdash z : B \wedge A}\,\text{Id}}{\{z : B \wedge A\} \vdash z.snd : A}\,\wedge\text{-E}_2 \qquad \dfrac{\dfrac{}{\{z : B \wedge A\} \vdash z : B \wedge A}\,\text{Id}}{\{z : B \wedge A\} \vdash z.fst : B}\,\wedge\text{-E}_1}{\dfrac{\{z : B \wedge A\} \vdash \langle z.snd,\ z.fst\rangle : A \wedge B}{\{\} \vdash \lambda z.\ \langle z.snd,\ z.fst\rangle : (B \wedge A) \longrightarrow (A \wedge B)}\,\longrightarrow\text{-I}}\,\wedge\text{-I} \qquad \dfrac{\dfrac{}{\{y : B\} \vdash y : B}\,\text{Id} \qquad \dfrac{}{\{x : A\} \vdash x : A}\,\text{Id}}{\{x : A,\ y : B\} \vdash \langle y,\ x\rangle : B \wedge A}\,\wedge\text{-I}}{\{x : A,\ y : B\} \vdash (\lambda z.\ \langle z.snd,\ z.fst\rangle)(\langle y,\ x\rangle) : A \wedge B}\,\longrightarrow\text{-E}
$$

$$\Downarrow$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\{y : B\} \vdash y : B}\,\text{Id} \qquad \dfrac{}{\{x : A\} \vdash x : A}\,\text{Id}}{\{x : A,\ y : B\} \vdash \langle y,\ x\rangle : B \wedge A}\,\wedge\text{-I}}{\{x : A,\ y : B\} \vdash \langle y,\ x\rangle.snd : A}\,\wedge\text{-E}_2 \qquad \dfrac{\dfrac{\dfrac{}{\{y : B\} \vdash y : B}\,\text{Id} \qquad \dfrac{}{\{x : A\} \vdash x : A}\,\text{Id}}{\{x : A,\ y : B\} \vdash \langle y,\ x\rangle : B \wedge A}\,\wedge\text{-I}}{\{x : A,\ y : B\} \vdash \langle y,\ x\rangle.fst : B}\,\wedge\text{-E}_1}{\{x : A,\ y : B\} \vdash \langle\langle y,\ x\rangle.snd,\ \langle y,\ x\rangle.fst\rangle : A \wedge B}\,\wedge\text{-I}
$$

$$\Downarrow$$

$$
\dfrac{\dfrac{}{\{x : A\} \vdash x : A}\,\text{Id} \qquad \dfrac{}{\{y : B\} \vdash y : B}\,\text{Id}}{\{x : A,\ y : B\} \vdash \langle x,\ y\rangle : A \wedge B}\,\wedge\text{-I}
$$

Figure 6: A reduction sequence with type derivations

Or easy?

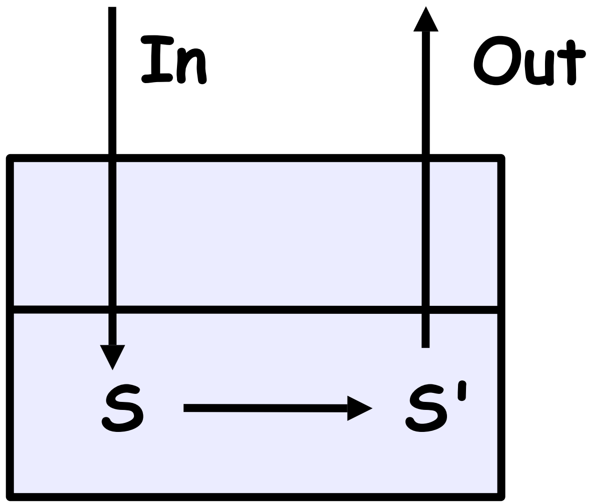
```
fac(0) -> 1;
fac(N) -> N*fac(N-1).
```

Why is FP good?

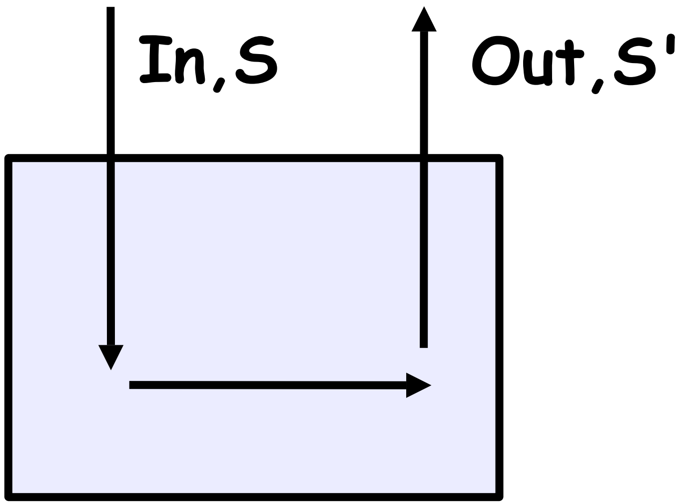
- Side effects are strictly controlled

> If you call the
> same function twice with
> the same arguments
> it should return the same value

Referential transparency



OOP



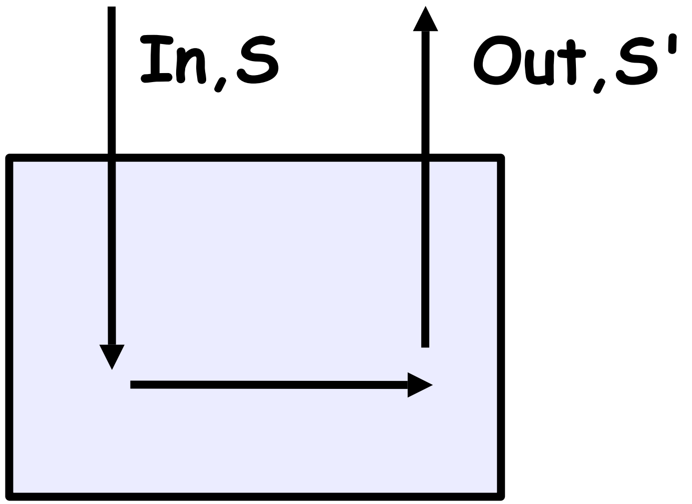
FP

Functional programming languages

FLPs carry state with them wherever the flow of control goes. Different FPLs provide different notations and mechanisms for hiding this from the user.

In Erlang we hide the state in a process. In Haskell in a monad

FLPs have are based on a formal mathematical model
Lambda calculus (Pi calc, CSP)



FP

Why is this important?

- Compositional properties
- Output of one function must be input to next
- f(g(h(i(k(X)))))
- Echo “foo” | k | i | h | g | f
- No mutable state means nothing to lock and automatic thread safety when parallelised
- Can reuse pure functions

FP is on the rise

- Haskell
- Erlang
- O'Caml, F#

BAD STUFF

Very very bad

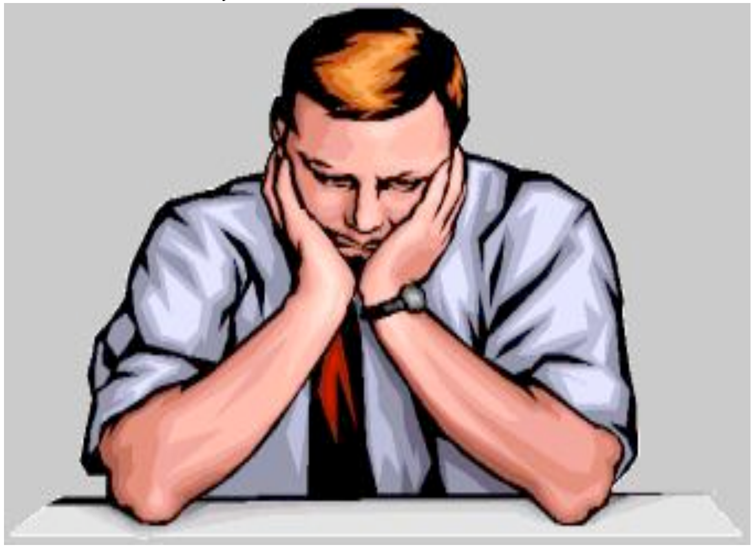
Threads

Sharing

Mutexes - Locks

Synchronized methods

Mutable state

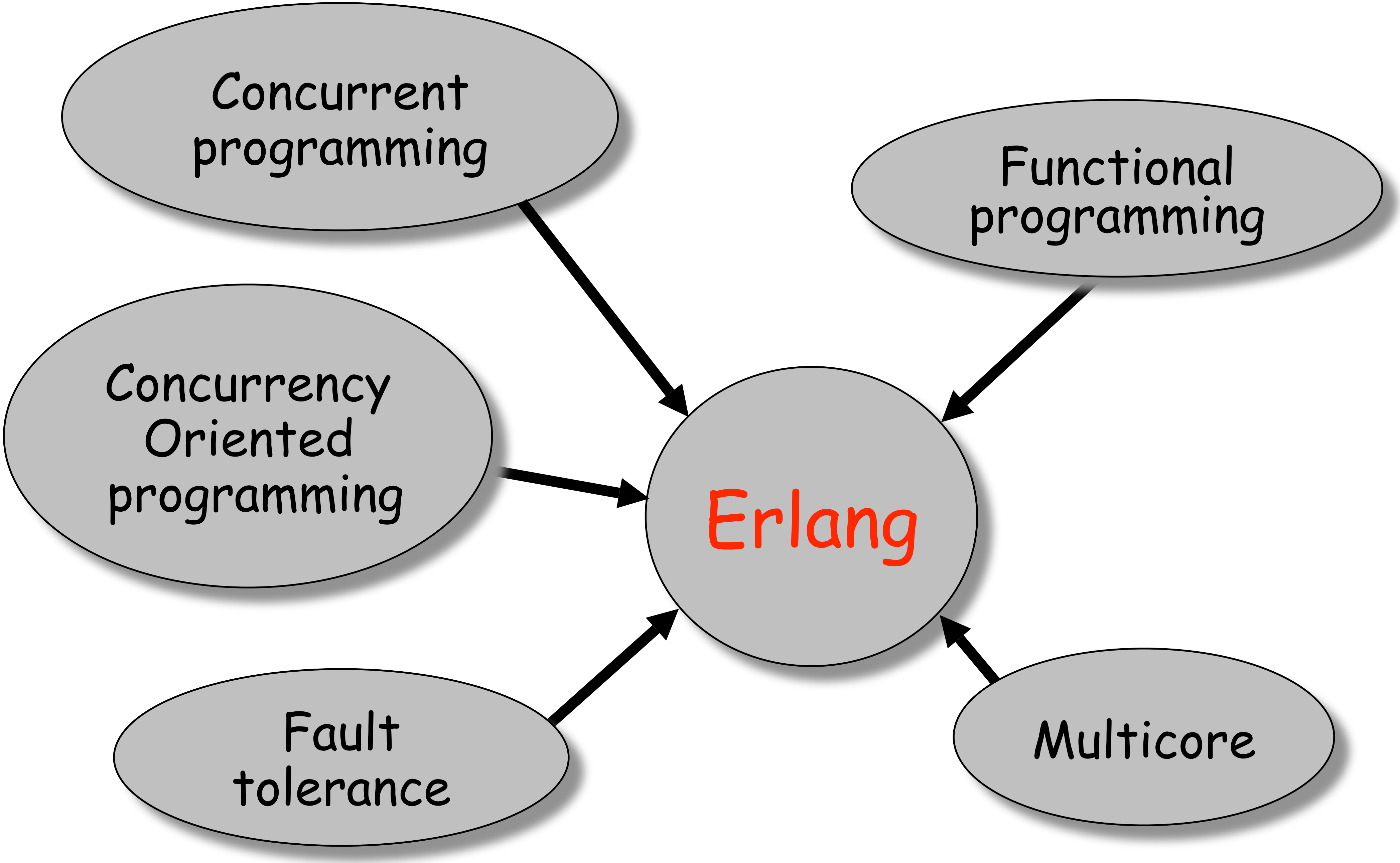


Mutable state is the root of all evil

FPLs have no mutable state

GOOD STUFF

Processes
Controlled side effects
Pure functions
Copying
Pure Message passing
Failure detection


Concurrent programming
Functional programming
Concurrency Oriented programming
Erlang
Fault tolerance
Multicore

The Pragmatic Programmers

Programming Erlang

Software for a Concurrent World

Joe Armstrong

Erlang in 11 Minutes

Sequential Erlang 5 examples

Concurrent Erlang 2 examples

Distributed Erlang 1 example

Fault-tolerant Erlang 2 examples

Bit syntax 1 example

Sequential Erlang

Factorial

Dynamic types
Pattern matching
No mutable data structures

```
-module(math).
-export([fac/1]).

fac(N) when N > 0 -> N*fac(N-1);
fac(0) -> 1

> math:fac(25).
 15511210043330985984000000
```

Binary Tree Search

```
lookup(Key, {Key, Val,_,_}) -> {ok, Val};
lookup(Key, {Key1,Val,S,B}) when Key < Key1 ->
   lookup(Key, S);
lookup(Key, {Key1, Val, S, B})->
   lookup(Key, B);
lookup(key, nil) ->
   not_found.
```

Sequential Erlang

append

```
append([H|T], L) -> [H|append(T, L)];
append([],    L) -> L.
```

sort

```
sort([Pivot|T]) ->
    sort([X||X <- T, X < Pivot]) ++
    [Pivot] ++
     sort([X||X <- T, X >= Pivot]);
sort([]) -> [].
```

adder

```
> Adder = fun(N) -> fun(X) -> X + N end end.
#Fun
> G = Adder(10).
#Fun
> G(5).
15
```

Concurrent Erlang

spawn

send

receive

```
Pid = spawn(fun() -> loop(0) end)

Pid ! Message,
.....

receive
    Message1 ->
        Actions1;
    Message2 ->
        Actions2;
        ...
    after Time ->
        TimeOutActions
end
```

The concurrency is in the language NOT the OS

Distributed Erlang

```
Pid = spawn(Fun@Node)

alive(Node),
.....

not_alive(Node)
```

Fault-tolerant Erlang

```
...
case (catch foo(A, B)) of
    {abnormal_case1, Y} ->
        ...
    {'EXIT', Opps} ->
        ...
    Val ->
        ...
end,
...

foo(A, B) ->
    ...
    throw({abnormal_case1, ...})
```

Monitor a process

```
...
process_flag(trap_exit, true),
Pid = spawn_link(fun() -> ... end),
receive
    {'EXIT', Pid, Why} ->
    ...
end
```

Bit Syntax - parsing IP datagrams

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN,5).
DgramSize = size(Dgram),
case Dgram of
 <<?IP_VERSION:4, HLen:4,
  SrvcType:8, TotLen:16, ID:16, Flgs:3,
  FragOff:13, TTL:8, Proto:8, HdrChkSum:16,
  SrcIP:32, DestIP:32, Body/binary>> when
   HLen >= 5, 4*HLen =< DgramSize ->
    OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
    <<Opts:OptsLen/binary,Data/binary>> = Body,
  ...
```

This code parses the header and extracts the data from an IP protocol version 4 datagram

Bit syntax – unpacking MPEG data

An MPEG header starts with an 11-bit *frame sync* consisting of eleven consecutive 1 bits followed by information that describes the data that follows:

```
AAAAAAAA AAABBCCD EEEEFFGH IIJJKLMM
```

AAAAAAAAAAA	The sync word (11 bits, all ones)
BB	2 bits is the MPEG Audio version ID
CC	2 bits is the layer description
D	1 bit, a protection bit

Download mp3_sync.erl

```
decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->
    Vsn = case B of
              0 -> {2,5};
              1 -> exit(badVsn);
              2 -> 2;
              3 -> 1
          end,
```

The magic lies in the amazing expression in the first line of the code.

```
decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->
```

This pattern matches eleven consecutive 1 bits,[1] 2 bits into B, 2 bits into C, and so on. Note that the code exactly follows the bit-level specification of the MPEG header given earlier. More beautiful and direct code would be difficult to write.

Some code

```
loop() ->
    receive
        {email,From,Subject,Text} = Email ->
            {ok, S} = file:open("inbox",[append,write]),
            io:format(S, "~p.~n",[Email]),
            file:close(S);
        {msg, From, Message} ->
            io:format("msg (~s) ~s~n", [From, Message]);
        {From, get, File} ->
            From ! file:read_file(File)
    end,
    loop().

Mike ! {email, "joe", "dinner", "see you at 18.00"}.

Helen ! {msg, "joe", "Can you buy some milk on your way
home?"}
```

Programming Multicore computers is difficult because of shared mutable state.

Functional programming languages have no shared state and no mutable state

Erlang has the right intrinsic properties for programming multicore computers (concurrency maps to the multiple CPUs, non-mutability means we don't get any problems with memory corruption)

Chapter 20

Programming Multicore CPUs

Here's the good news for Erlang programmers:

Your Erlang program might run N times faster on an N core processor—*without any changes to the program.*

If, that is, you've followed a simple set of rules....

If you want your application to run faster on a multicore CPU, you'll have to make sure that it has lots of processes, that the processes don't interfere with each other, and that you have no sequential bottlenecks in your program.

If instead you've written your code in one great monolithic clump of sequential code and never used `spawn` to create a parallel process, your program might not go any faster.

Don't despair: even if your program started as a gigantic sequential program there are several rather simple things we can do to the program to parallelize it.

In this chapter we'll look at the following topics.

1. What we have to do to make our programs run efficiently on a multicore CPU.
2. How to parallelize a sequential program.
3. The problem of sequential bottlenecks.

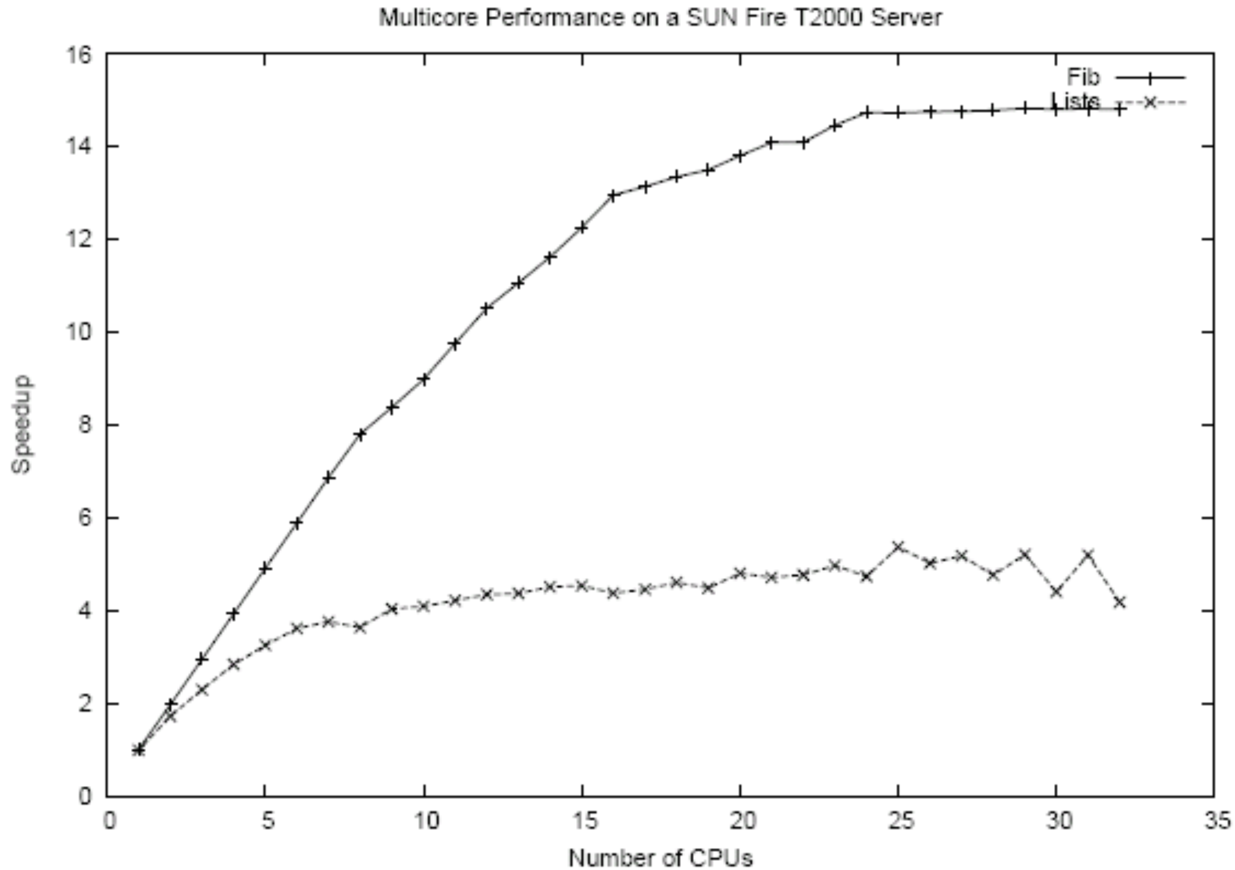


Figure 20.1: Speedup on multicore CPU

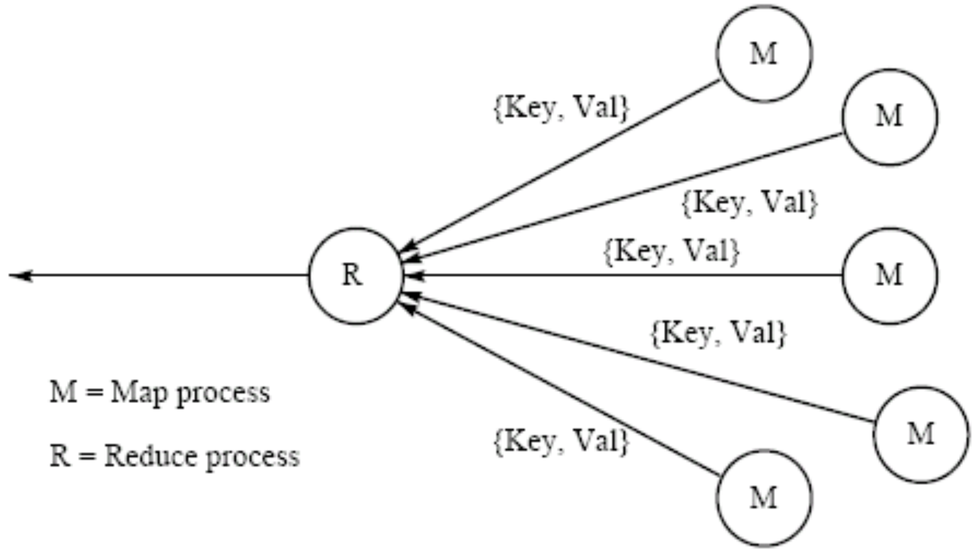


Figure 20.2: Mapreduce

- Use “lots” of processes
- Avoid sequential bottlenecks
- Use “large computation“ small data transfer (if possible)
- New abstractions (pmap, mapreduce)

Commercial projects

Ericsson AXD301 (part of "Engine")
Ericsson GPRS system
Alteon (Nortel) SSL accelerator
Alteon (Nortel) SSL VPN
Teba Bank (credit card system – South Africa)
T-mobile SMS system (UK)
Kreditor (Sweden)
Synapse
Tail-f
jabber.org /uses ejabberd)
Twitter (uses ejabberd)
Lshift (RabbitMQ) AMQP (Advanced Message Queuing protocol)

Finally

We've known how to program parallel computers for the last twenty years

We can make highly reliable fault tolerant distributed real-time systems

ww.erlang.org

Questions?