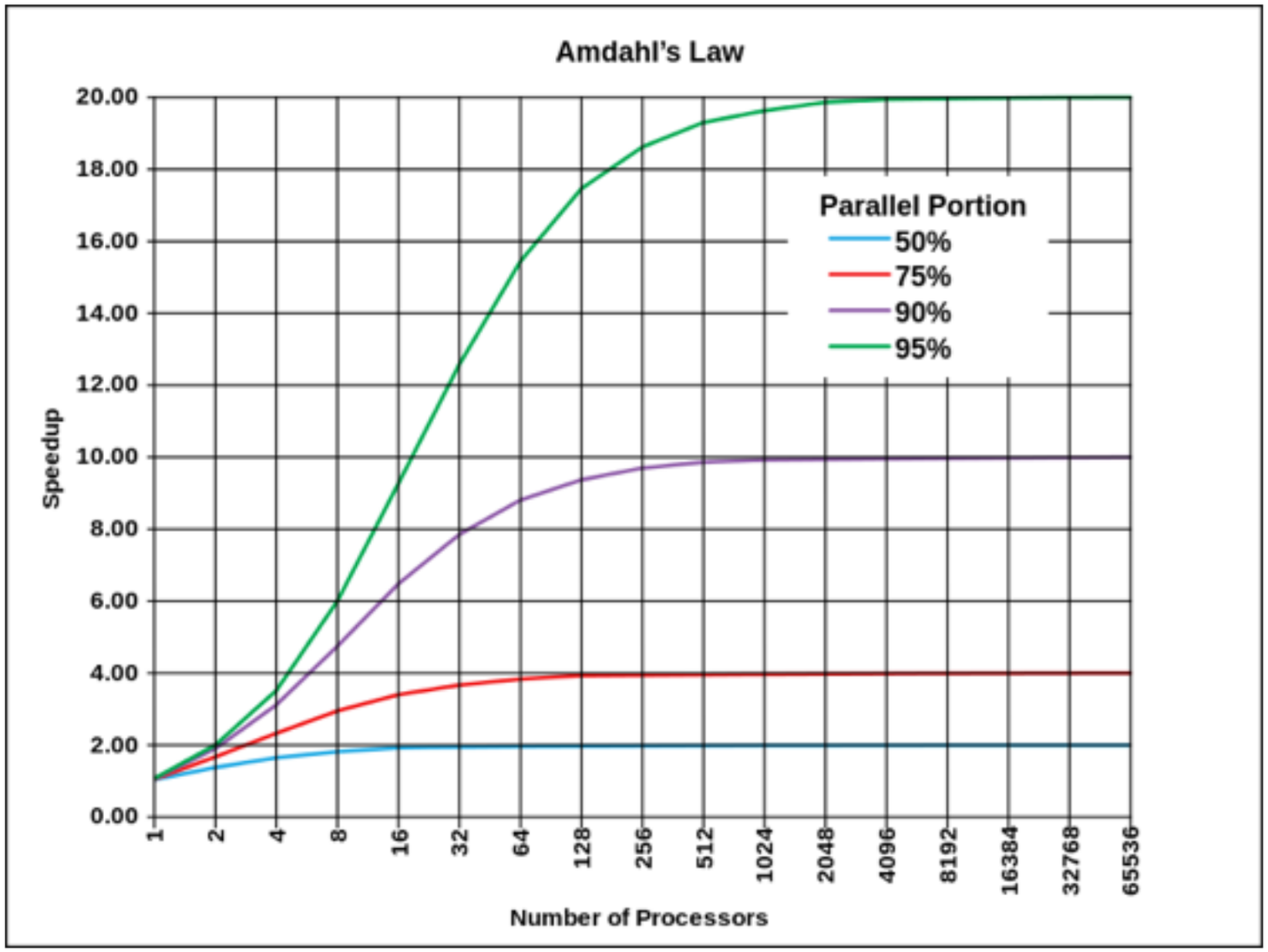
codecentric

Akka_

What’s the problem?

Amdahl's Law



https://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/AmdahlsLaw.svg/800px-AmdahlsLaw.svg.png

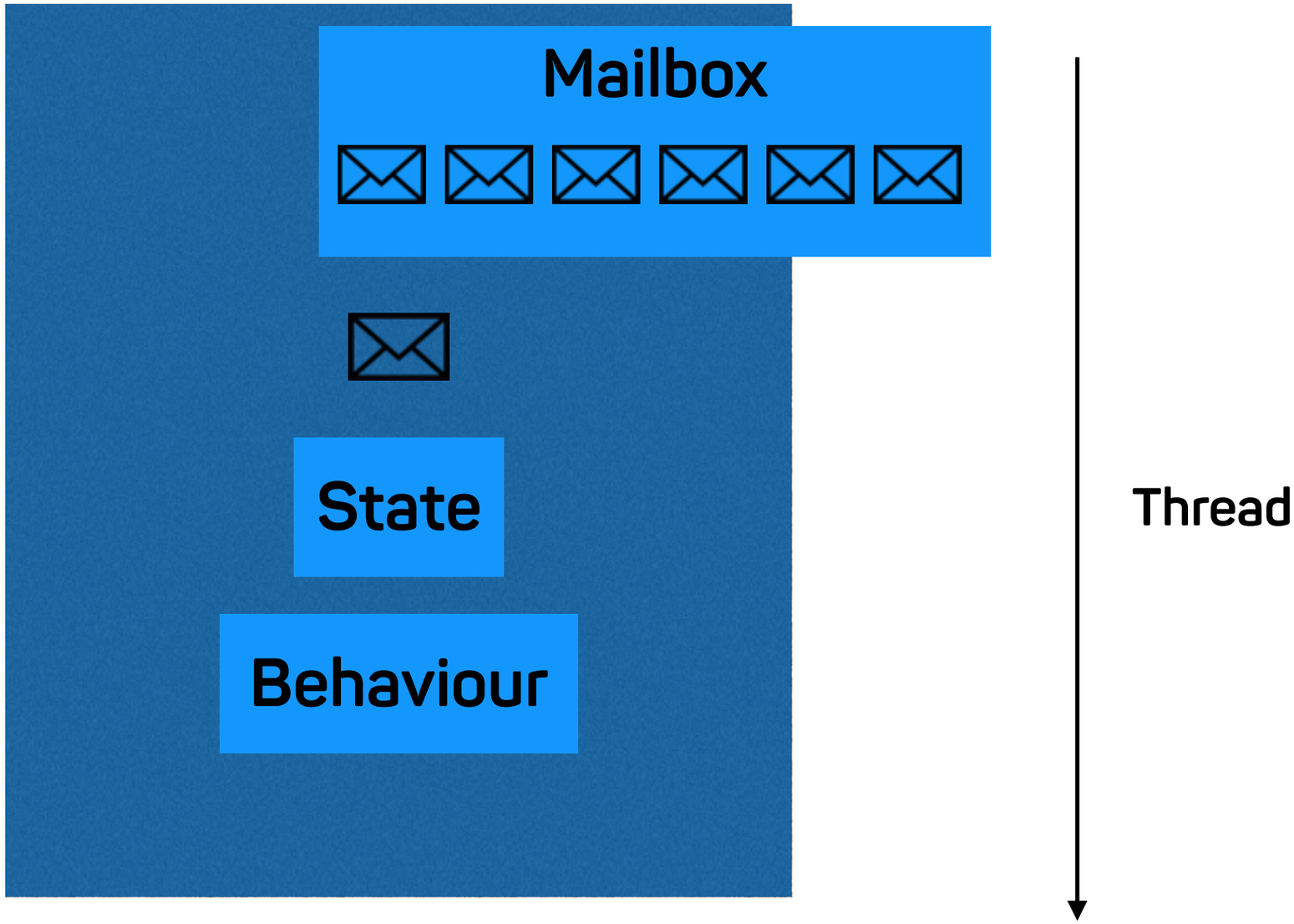
We need to go concurrent

- But concurrency is hard...
  - Race Conditions
  - Blocking Calls
  - Deadlocks

Akka

Actor-based concurrency

- Light-weight actor objects
- No shared mutable state
- Asynchronous messaging between actors
- By definition distributable
- Single-threaded computation


Mailbox
State
Behaviour
Thread

Hello World_

```
class HelloActor extends Actor {
  def receive = {
    case "hello" => println("hello back at you")
    case _       => println("huh?")
  }
}

object Main extends App {
  val system = ActorSystem("HelloSystem")
  val helloActor = system.actorOf(Props[HelloActor], name = "helloactor")
  helloActor ! "hello"
  helloActor ! "buenos dias"
}
```

There are many modules

- akka-remote
- akka-cluster
- akka-http
- akka-streams
- See akka.io

Reactive Programming and Akka Streams

A reactive system should be

- Responsive
  - Respond to clients in a timely manner if at all possible
- Resilient
  - System remains responsive during failures
- Elastic
  - Needs to be able to handle varying workloads
- Message-driven
  - Asynchronous
  - Handling back-pressure

An overloaded component must not drag down the whole system

- Components must communicate that they're under stress
- Upstream components must respect that and reduce load
- Might impact responsiveness

Streams that understand back-pressure

- http://www.reactive-streams.org
- Standardised Interfaces and semantics for reactive streams
- Akka Streams is an implementation
- Actor-based implementation that is abstracted away from the developer

Source

- Source of data
  - Database Query
  - Http Request
  - Random Number generator
  - Kafka
  - ...

Sink

- Destination of Data
  - Database
  - Web socket
  - File System
  - System.out
  - Kafka
  - ...

Flow

- Transformation of Data
  - mapping
  - flat mapping
  - filter
  - join
  - zip
  - ...

Example_

```
val source = Source((10 to 20))
val sink = Sink.foreach[Int](println _)

source.runWith(sink)
```

```
10
11
12
13
14
15
16
17
18
19
20
```

Example_

```
val source = Source((10 to 20))
val randomizer =
Flow[Int].map(Random.nextInt(_))
val sink = Sink.foreach[Int](println _)

source.via(randomizer).runWith(sink)
```

```
1
0
8
10
6
14
13
12
5
17
14
```