canoo

# Concurrent programming for you and me

Dierk König

**GeeCon 2012**

# Welcome!

**Dierk König**

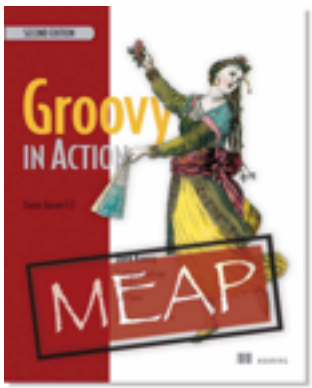
Fellow @ Canoo Engineering AG, Basel (CH)

Rich Internet Applications

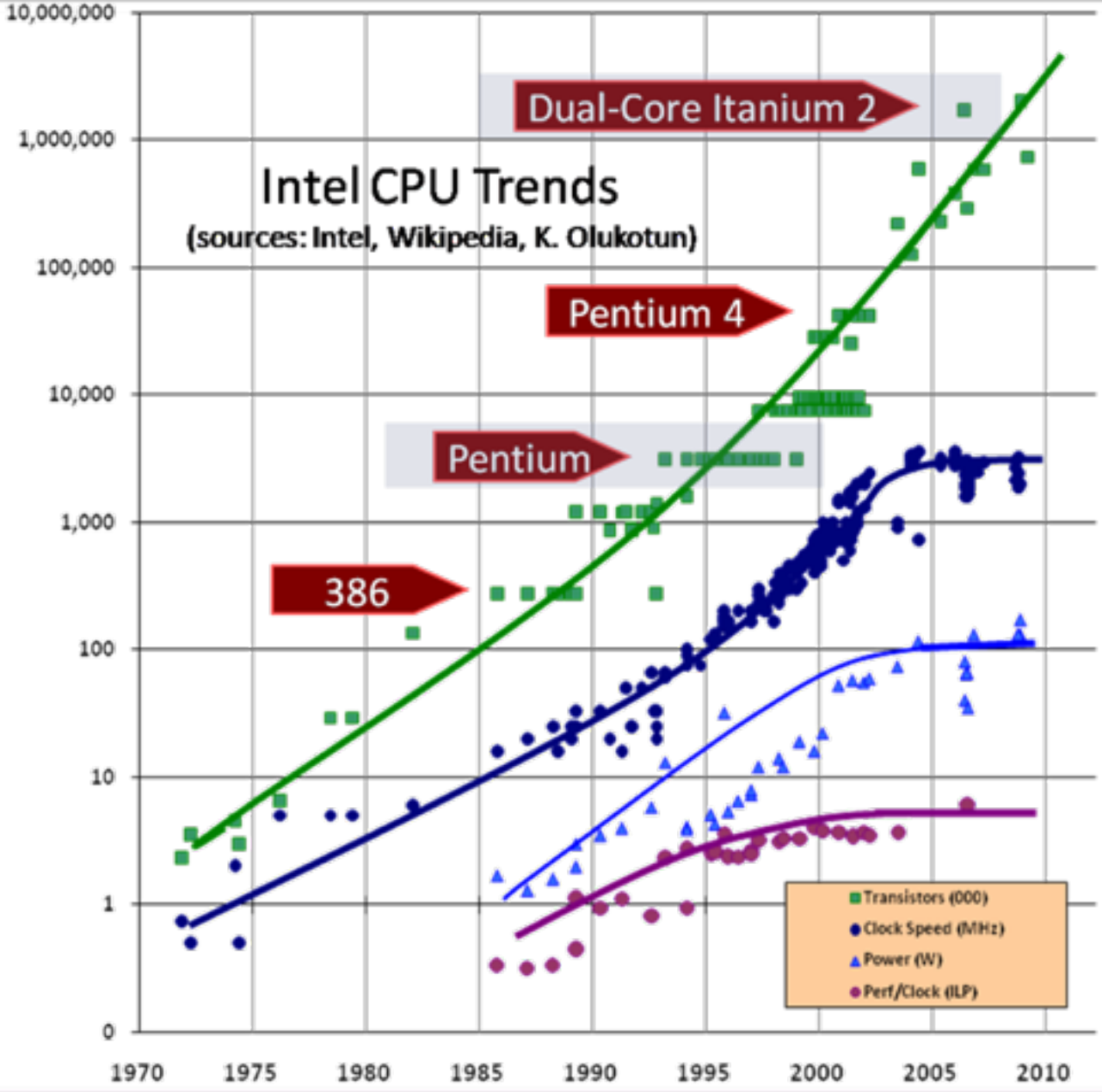
Products, Projects, Consulting

**www.canoo.com**

Open-source committer Groovy, Grails, GPars

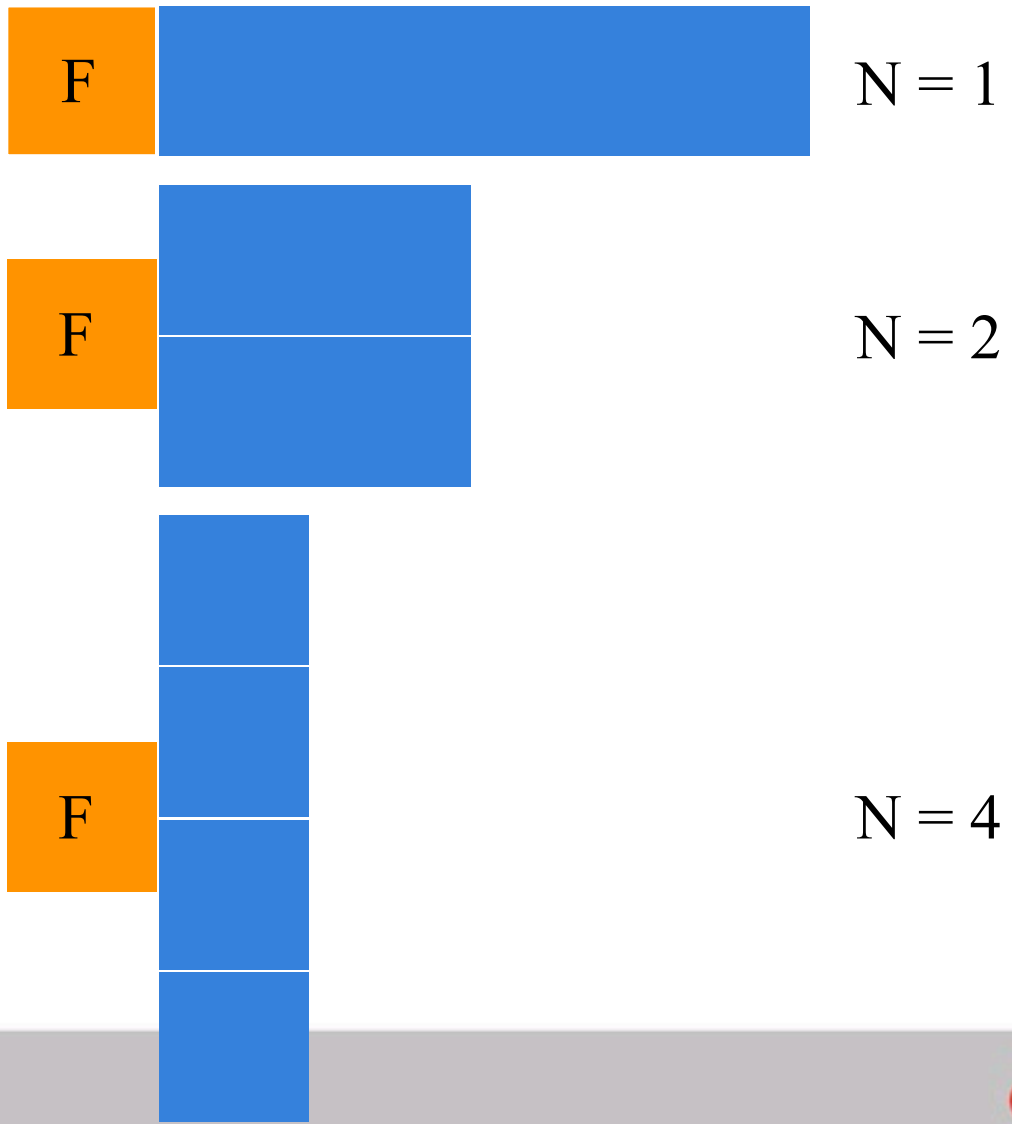

canoo


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)



canoo

# What you can expect






canoo

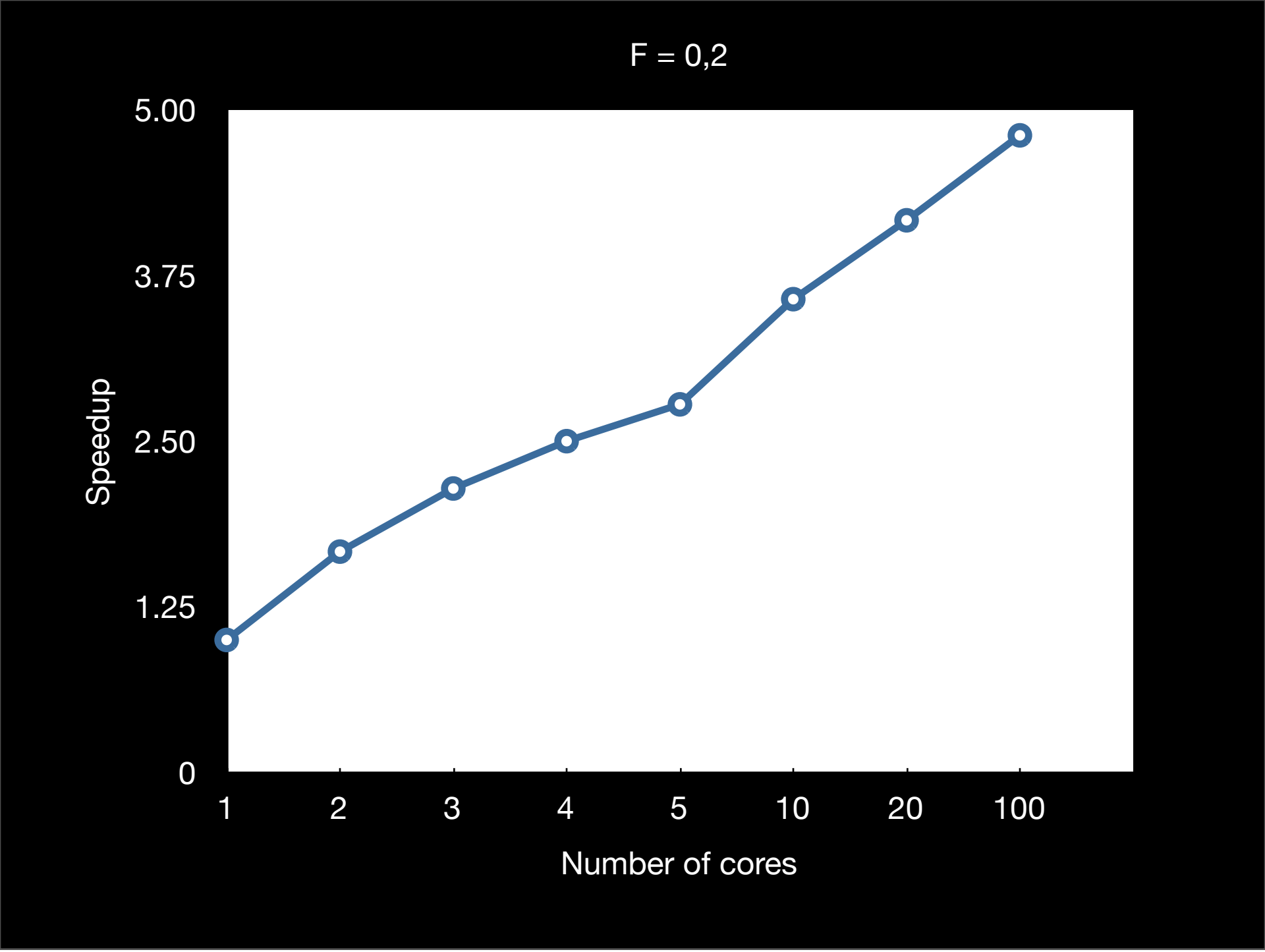
# Amdahl‘s law



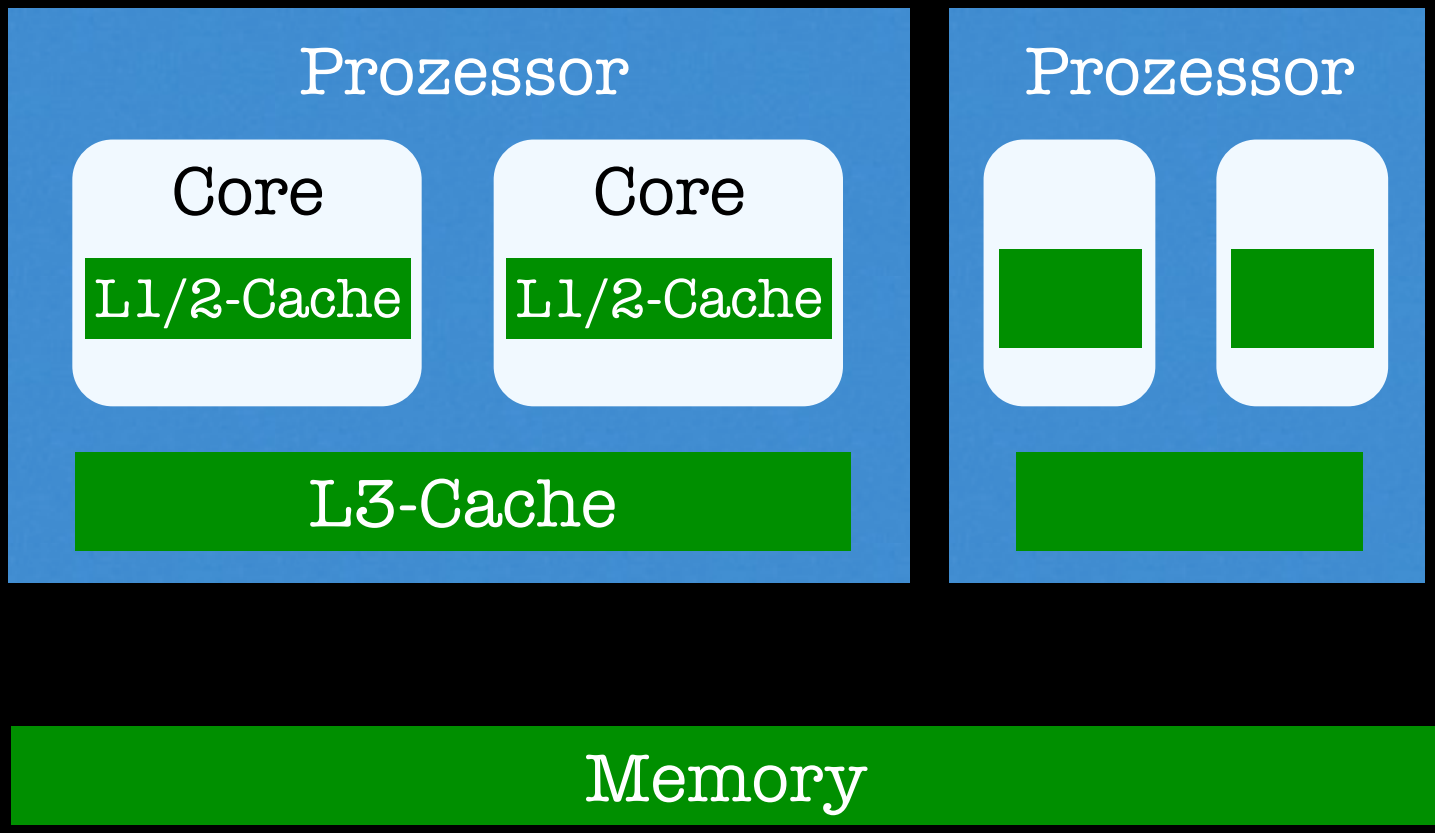
canoo

# Amdahl‘s law

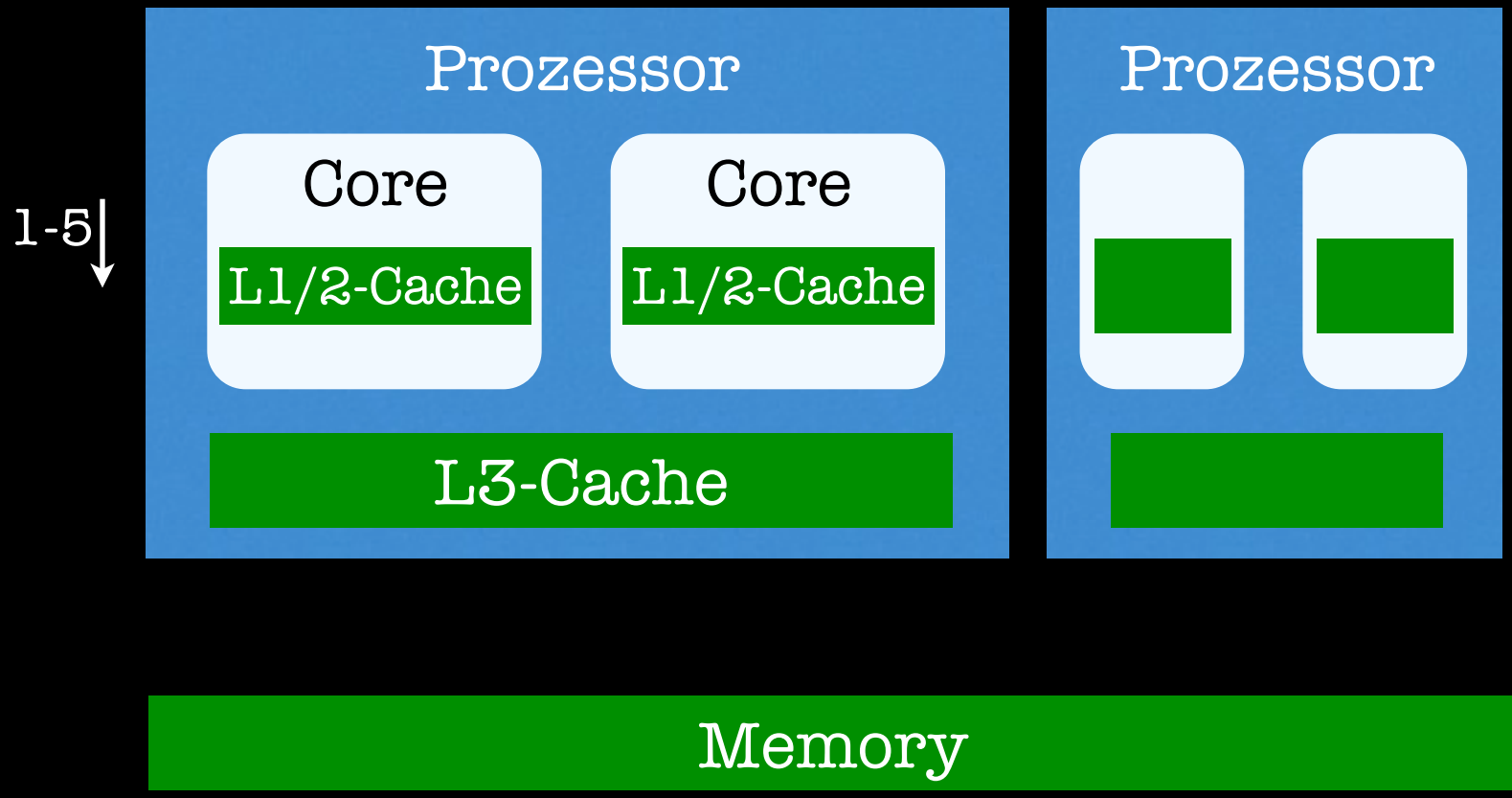
$$speedup \leq \frac{1}{\left( F + \frac{(1 - F)}{N} \right)}$$


F = 0,2
5.00
3.75
2.50
1.25
0
Speedup
1
2
3
4
5
10
20
100
Number of cores

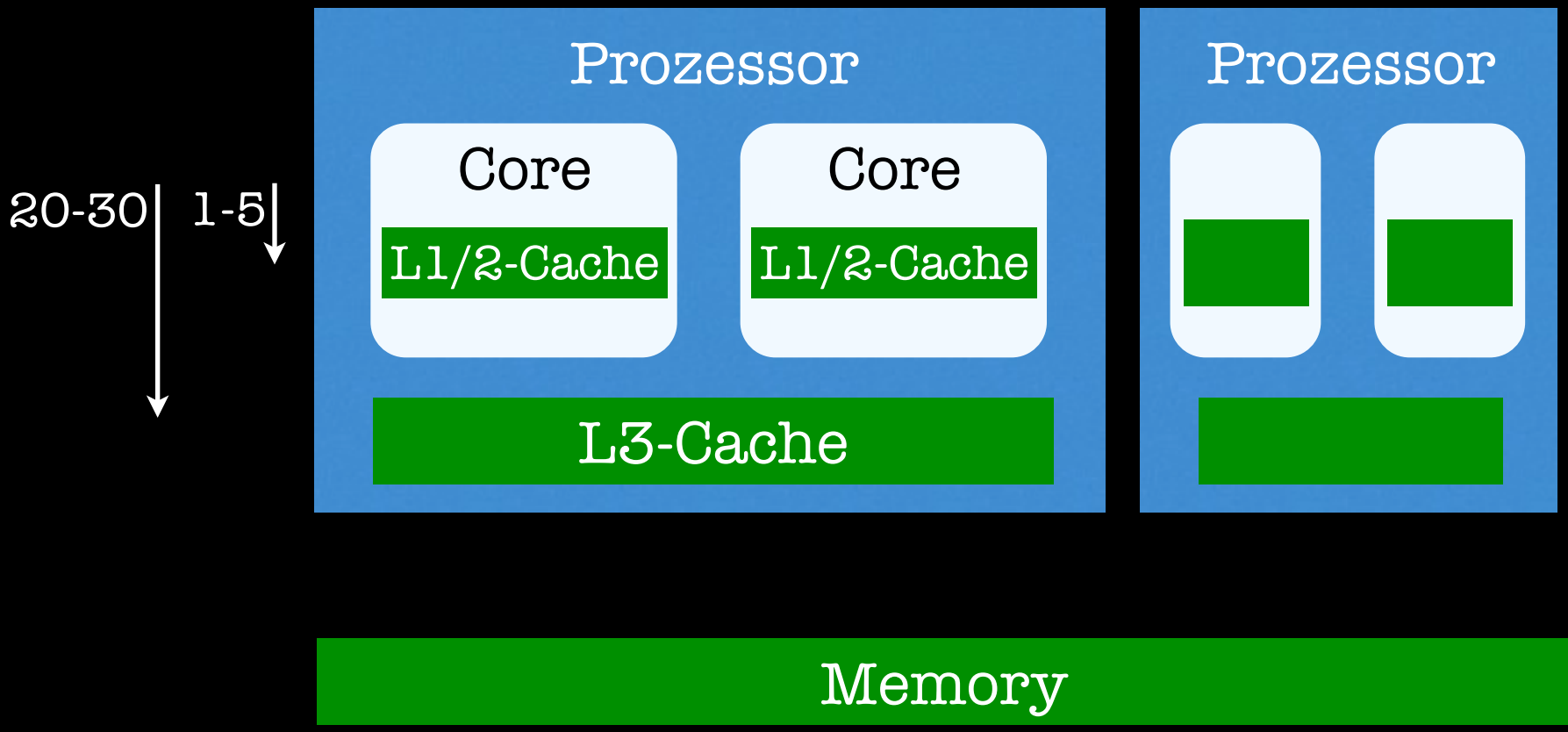
# Caching, Memory & Co



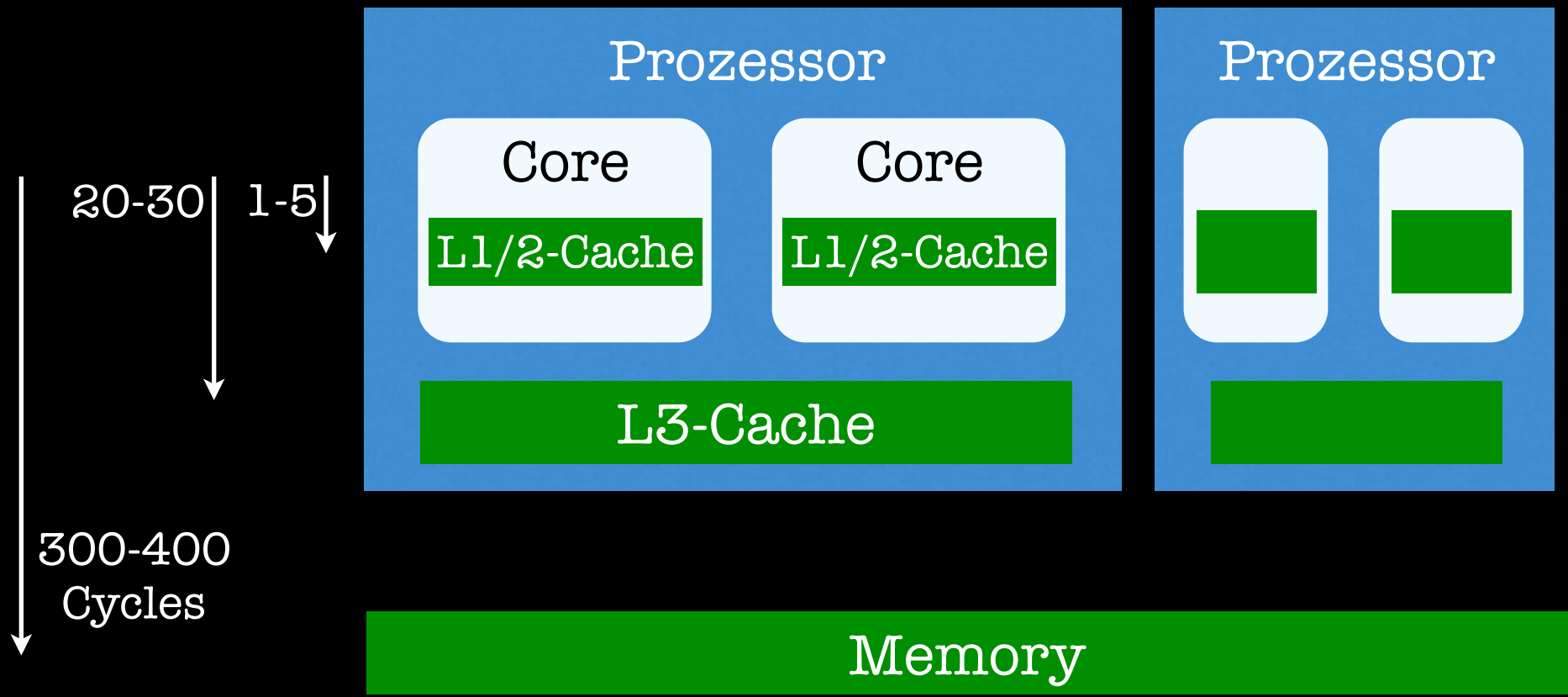
# Caching, Memory & Co



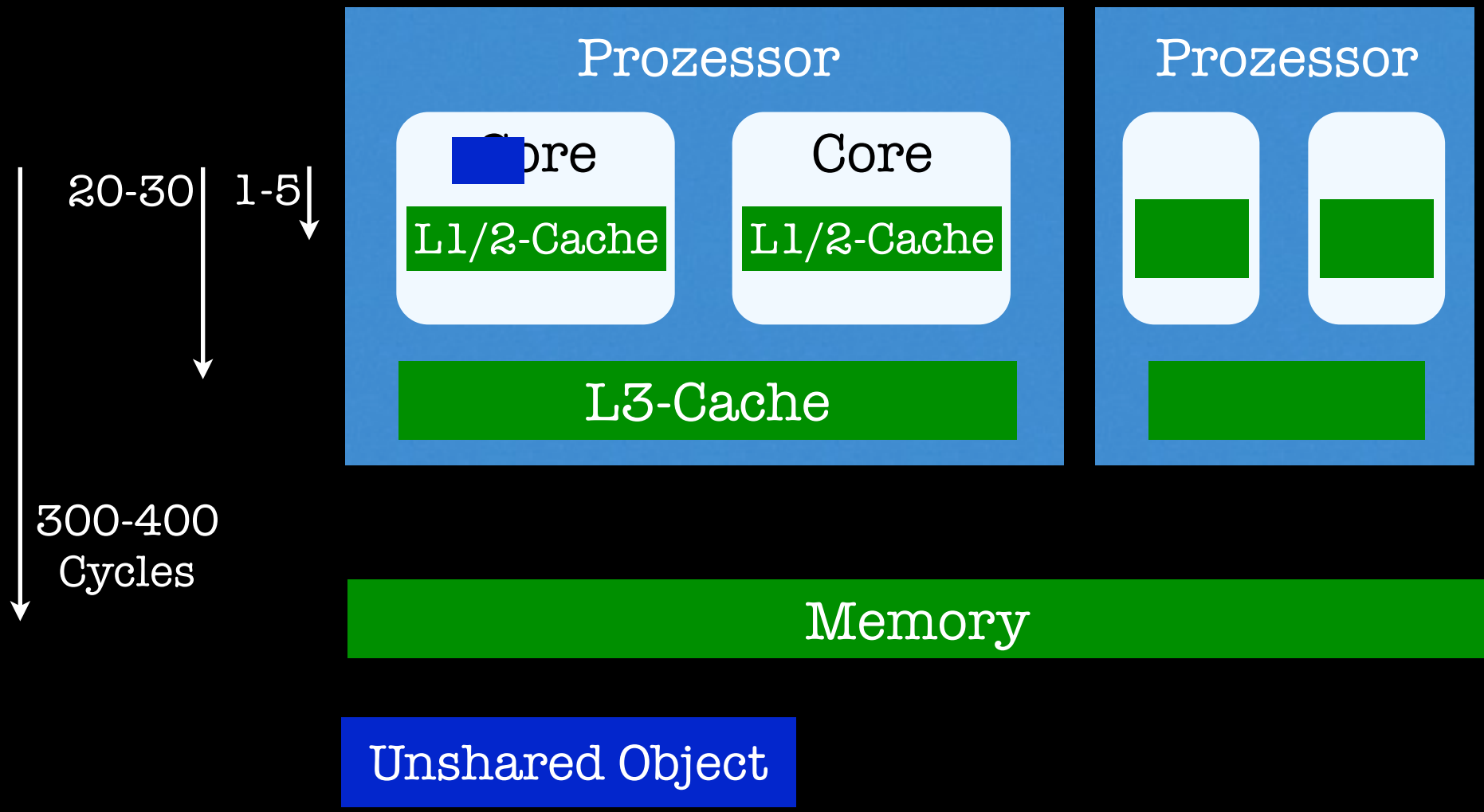
# Caching, Memory & Co



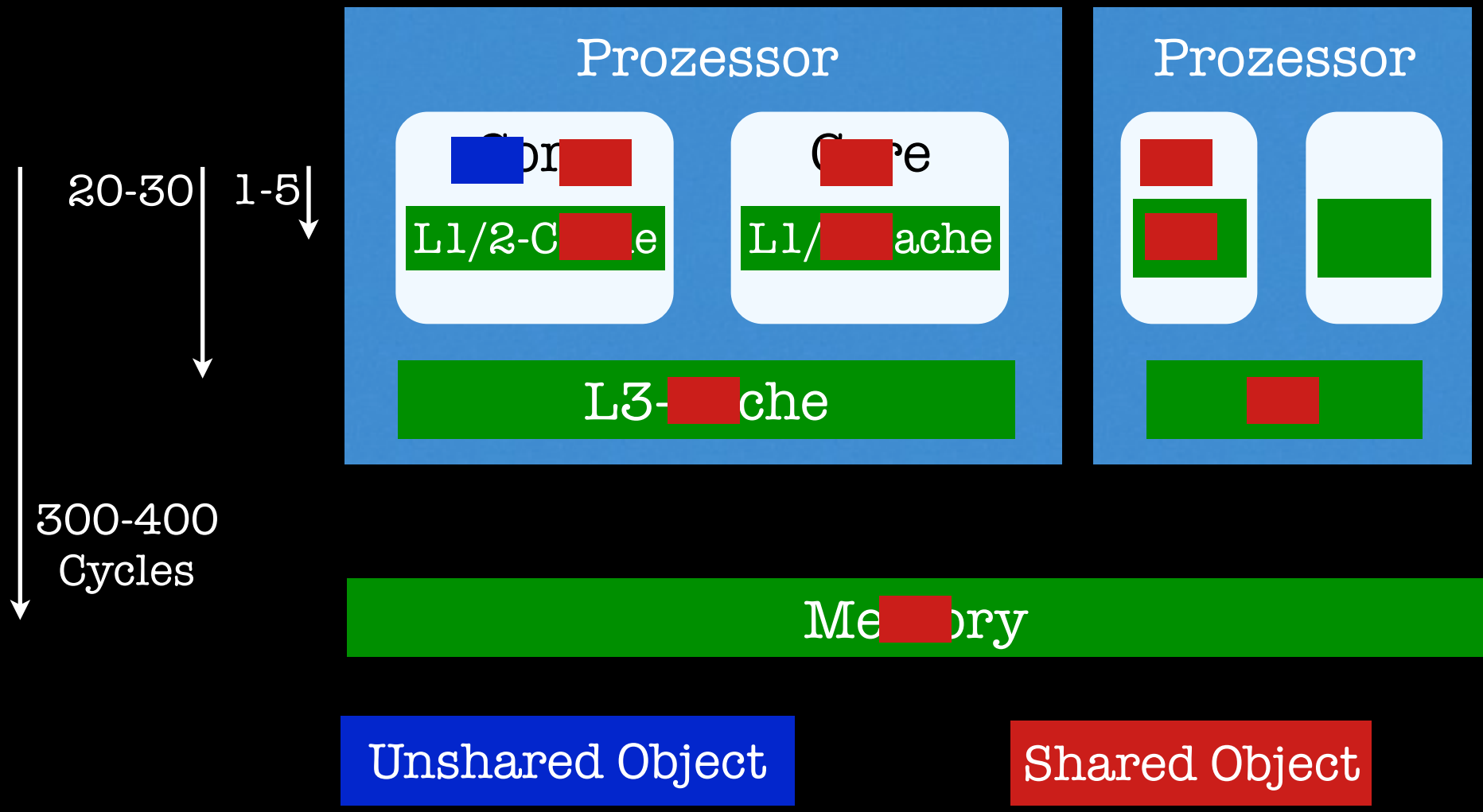
# Caching, Memory & Co



# Caching, Memory & Co



# Caching, Memory & Co



# The Java state of affairs

Starting new threads is easy.
Some real goodies in java.util.concurrent.* & Java 7

Manual thread-coordination is difficult.
Access to shared state is error-prone.

Scheduling issues for many threads with bad concurrency characteristics.
Good use of pooling is not obvious.
Concepts are rather „low level“.

# For the application programmer

The servlet model works!

Swing, JavaFx? Oh, well...

We should learn from what works.



canoo

# It‘s all about coordination

| | |
|---|---|
| Fork/Join | Working on collections with |
| Map/Filter/Reduce | fixed coordination |
| Actor | Explicit coordination |
| Agent | Delegated coordination |
| Dataflow | Implicit coordination |
| Kanbanflow | Balanced coordination |


1.8.6

canoo

# Fork/Join on collections

```
import static groovyx.gpars.GParsPool.withPool

def numbers = [1, 2, 3,  4,  5,  6]
def squares = [1, 4, 9, 16, 25, 36]

withPool {
 assert squares == numbers.collectParallel  { it * it }
}

// in reality, find better chunks of work!
```

# Fork/Join on collections

```
import static groovyx.gpars.GParsPool.withPool

def numbers = [1, 2, 3,  4,  5,  6]
def squares = [1, 4, 9, 16, 25, 36]

withPool {
 assert squares == numbers.collectParallel  { it * it }
}

// in reality, find better chunks of work!
```




canoo

# More such methods

any { ... }  collect { ... }  count(filter)
each { ... }  eachWithIndex{ ... }
every { ... }
find { ... }  findAll { ... }  findAny { ... }
fold { ... }  fold(seed) { ... }
grep(filter)
groupBy { ... }
max { ... }  max()
min { ... }  min()
split { ... }  sum()



canoo

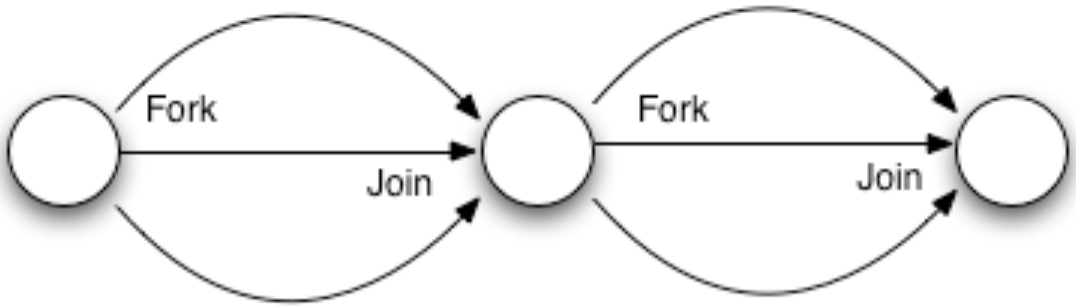
# Map/Filter/Reduce on collections

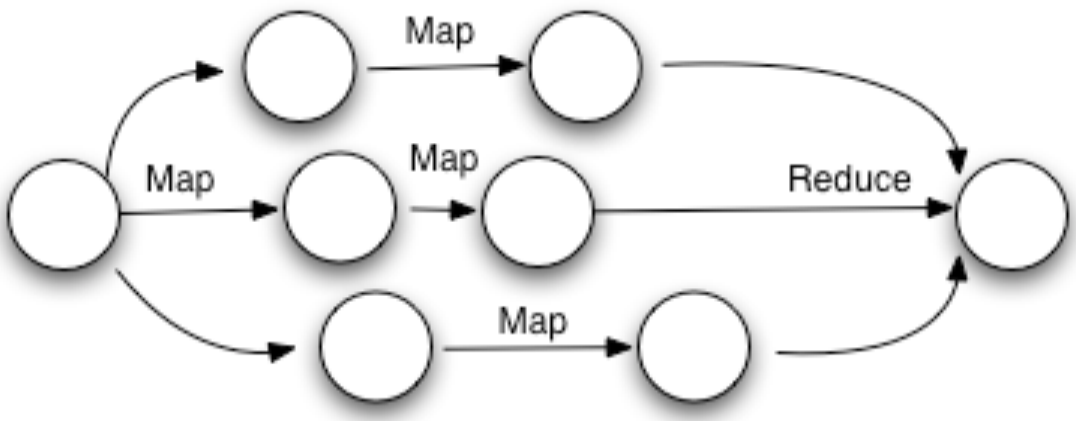
```
import static groovyx.gpars.GParsPool.withPool

withPool {
    assert 55 == [0, 1, 2, 3, 4].parallel
        .map    { it + 1  }
        .map    { it ** 2 }
        .reduce { a, b -> a + b }
}
```



canoo

# Fork/Join vs Map/Filter/Reduce

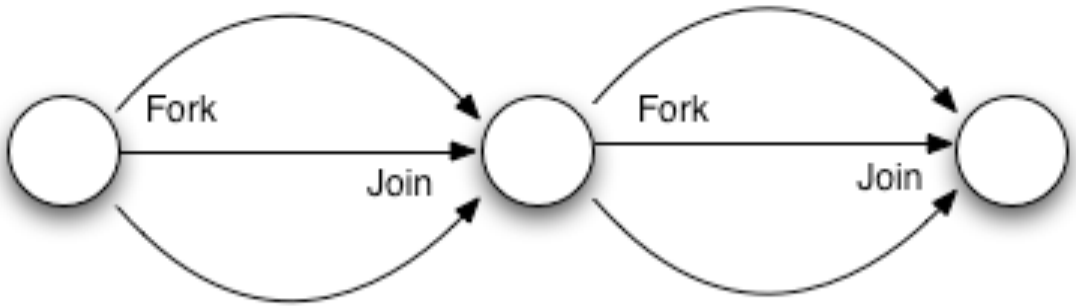


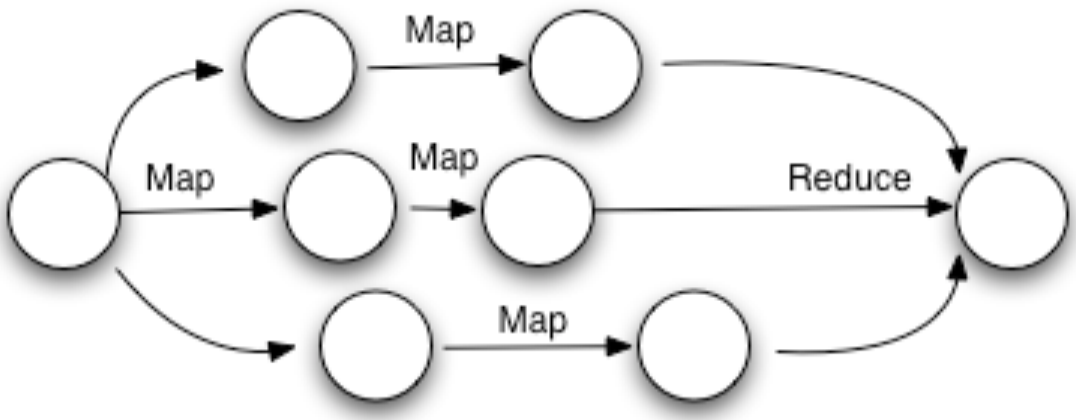



canoo

# Fork/Join vs Map/Filter/Reduce




fixed coordination






canoo

# Explicit coordination with Actors

```
import static groovyx.gpars.actor.Actors.*

def printer   = reactor { println it }
def decryptor = reactor { reply it.reverse() }

actor {
    decryptor.send    'lellarap si yvoorG'
    react {
        printer.send  'Decrypted message: ' + it
        decryptor.stop()
        printer.stop()
    }
}.join()
```



canoo

# Actors

Process one message at a time.

Dispatch on the message type,
which fits nicely with dynamic languages.

Are often used in composition,
which can lead to further problems down the road.

Personal note:
Actors are overrated



canoo

# Delegate to an Agent

```
import groovyx.gpars.agent.Agent

def safe = new Agent<List>( ['GPars'] )

safe.send { it.add 'is safe!'  }
safe.send { updateValue it * 2 }

println safe.val
```



canoo

# Agents

Analogous to Clojure agents (atoms, refs, ...)

Implementations differ much in efficiency.



canoo

# DataFlow for implicit coordination

```
import groovyx.gpars.dataflow.Dataflows
import static groovyx.gpars.dataflow.Dataflow.task

final flow = new Dataflows()
task { flow.result = flow.x + flow.y }
task { flow.x = 10 }
task { flow.y =  5 }

assert 15 == flow.result
```



canoo

# Dataflow

Flavors: variables, streams, operators, tasks, flows

Write-Once, Read-Many (non-blocking)

Feel free to use millions of them

Fast, efficient, safe, and testable!

Model the flow of data,
not the control flow!



canoo

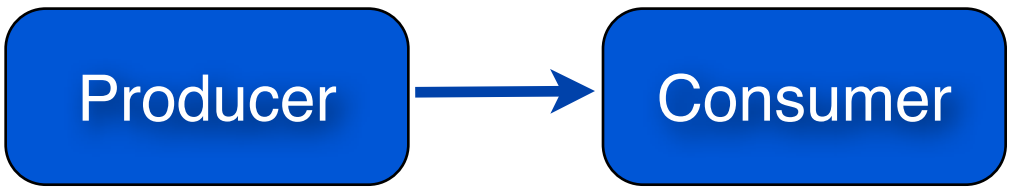
# KanbanFlow in code

```
@Grab('org.codehaus.gpars:gpars:1.0-beta-1')
import groovyx.gpars.dataflow.*
import static ProcessingNode.node

def producer = node { below -> below << 1 }
def consumer = node { above -> println above.take() }

new KanbanFlow().with {
    link producer to consumer
    start()
    // run for a while
    stop()
}
```



canoo

# Kanbanflow scenarios






canoo

# Kanbanflow chaining



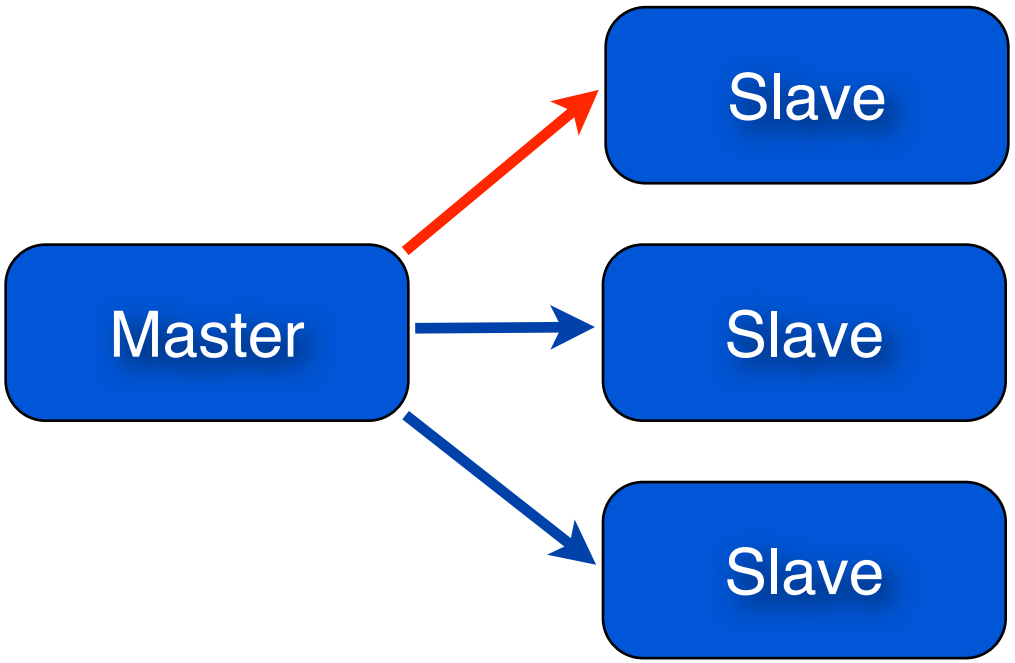

canoo

# Kanbanflow cycles




canoo

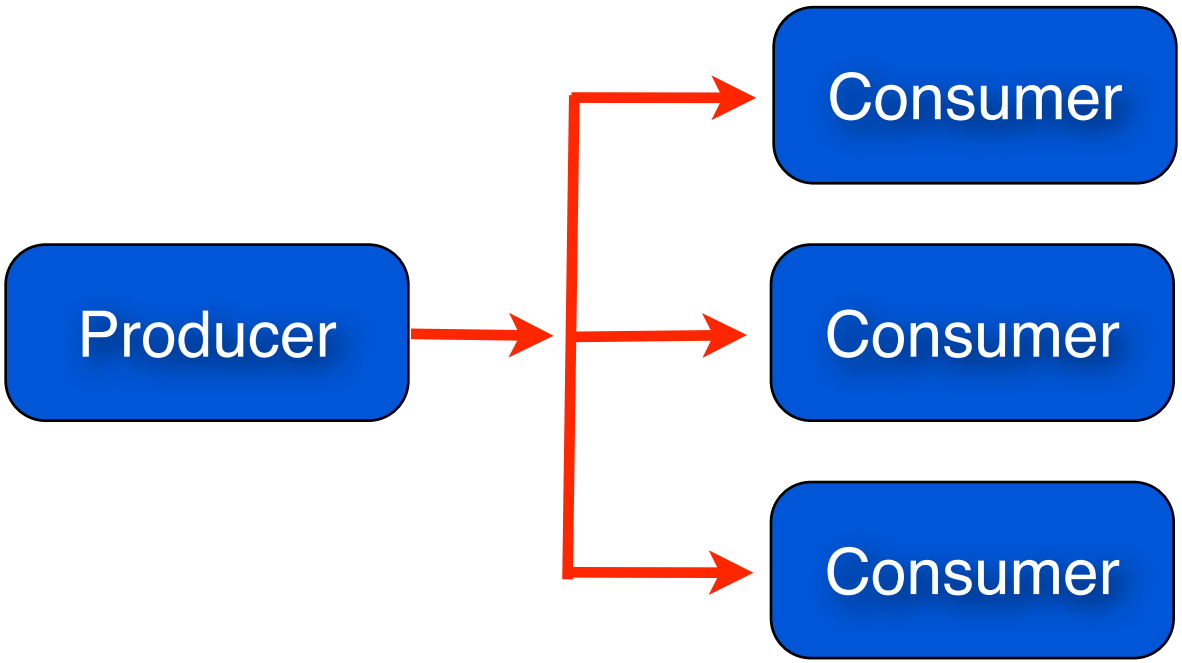
# Kanbanflow load balancer






canoo

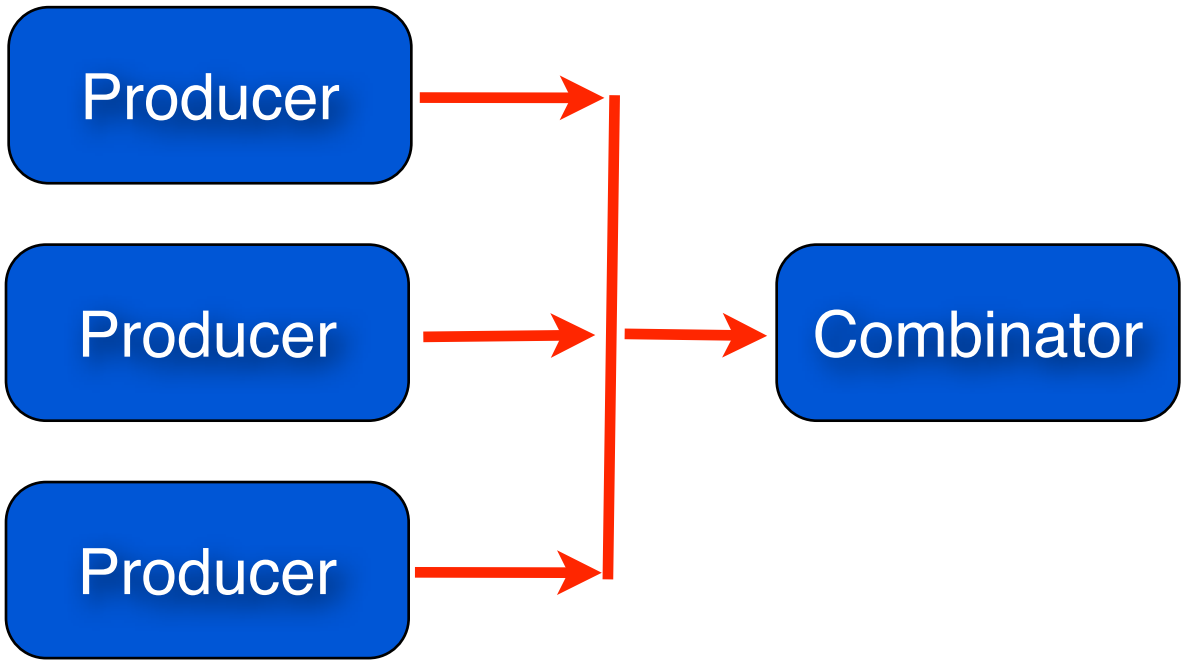
# Kanbanflow broadcast






canoo

# Kanbanflow combinator






canoo

# Efficient Producer-Consumer

KanbanFlow pattern by /me

http://people.canoo.com/mittie/kanbanflow.html

Simple idea, amazing results

Resource efficient, composable, testable

Non-blocking writes,
Deadlock-free by design

# Takeaways

1. Learn the concepts
2. Experiment with GPars
3. Validate

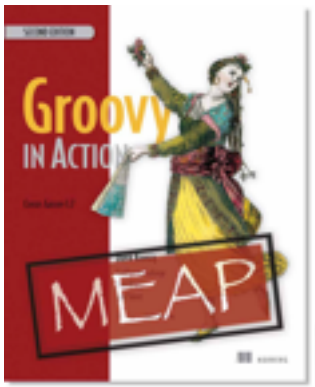


canoo

# Further reading

- **Groovy in Action**,
  2nd Edition
  chapter 17

- gpars.codehaus.org





canoo




canoo




canoo

# Discussion


credits:
Paul King

# Discussion

dierk.koenig@canoo.com
@mittie


credits:
Paul King