CONCURRENCY MODELS: GO CONCURRENCY MODEL

BY VASYL NAKVASIUK, 2014

KYIV GO MEETUP #1

CONCURRENCY AND PARALLELISM

CONCURRENCY AND PARALLELISM

THE WORLD IS OBJECT ORIENTED

THE WORLD IS PARALLEL

THE WORLD IS OBJECT ORIENTED AND PARALLEL

CONCURRENCY AND PARALLELISM

Concurrency is a composition of independently computing things.

Parallelism is a simultaniuse execution of multiple things.

Concurrency is about dealing with lots of things at once.

Parallelism is about doing lots of things at once.

Rob Pike, "Concurrency Is Not Parallelism", 2012

CONCURRENCY AND PARALLELISM

CONCURRENT

CONCURRENT AND PARALLEL

PARALLEL

CONCURRENCY AND PARALLELISM

USERS

SOFTWARE

MULTICORE

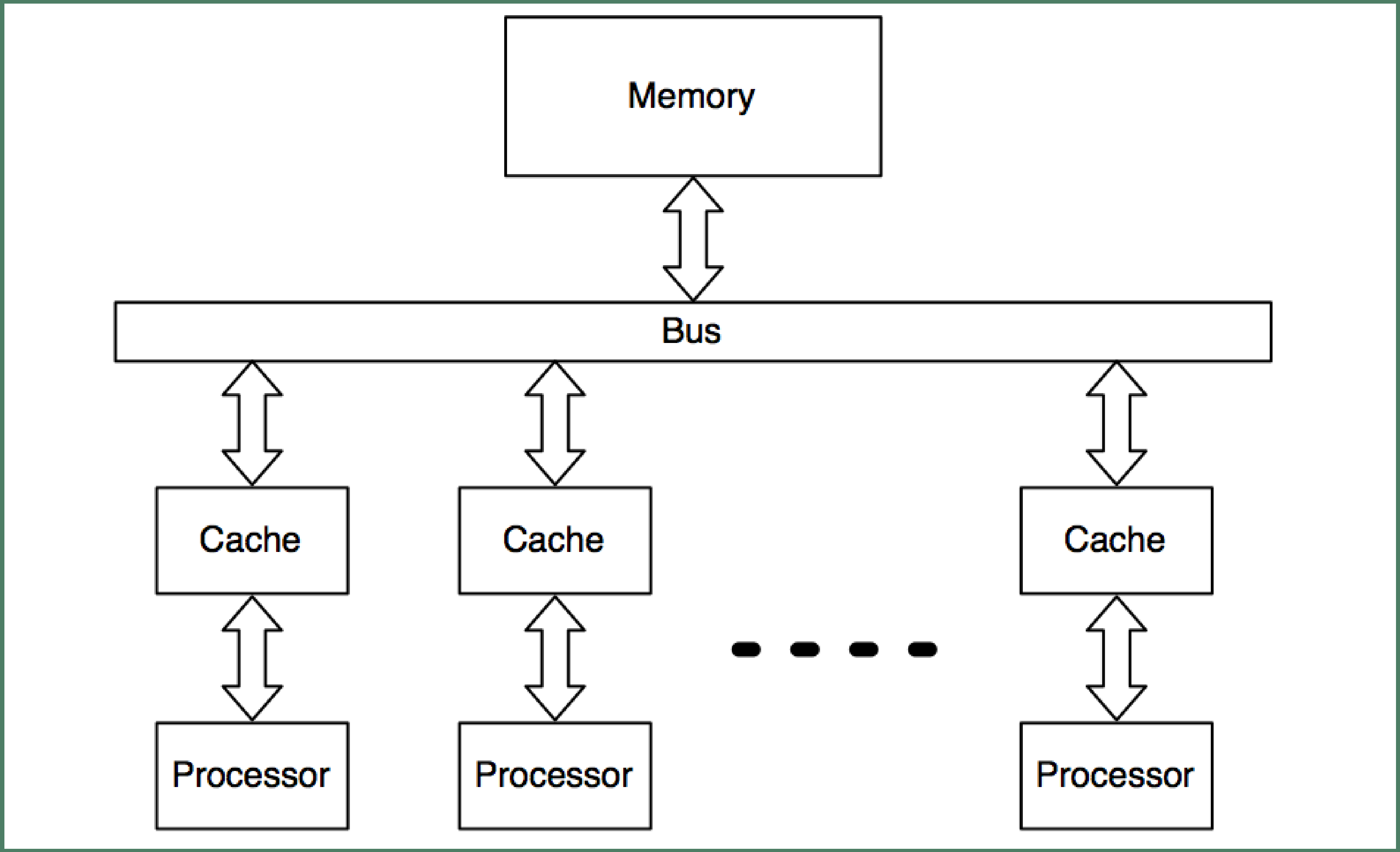
CONCURRENCY AND PARALLELISM

MOORE’S LAW

CPU: WHY ARE STALLED?

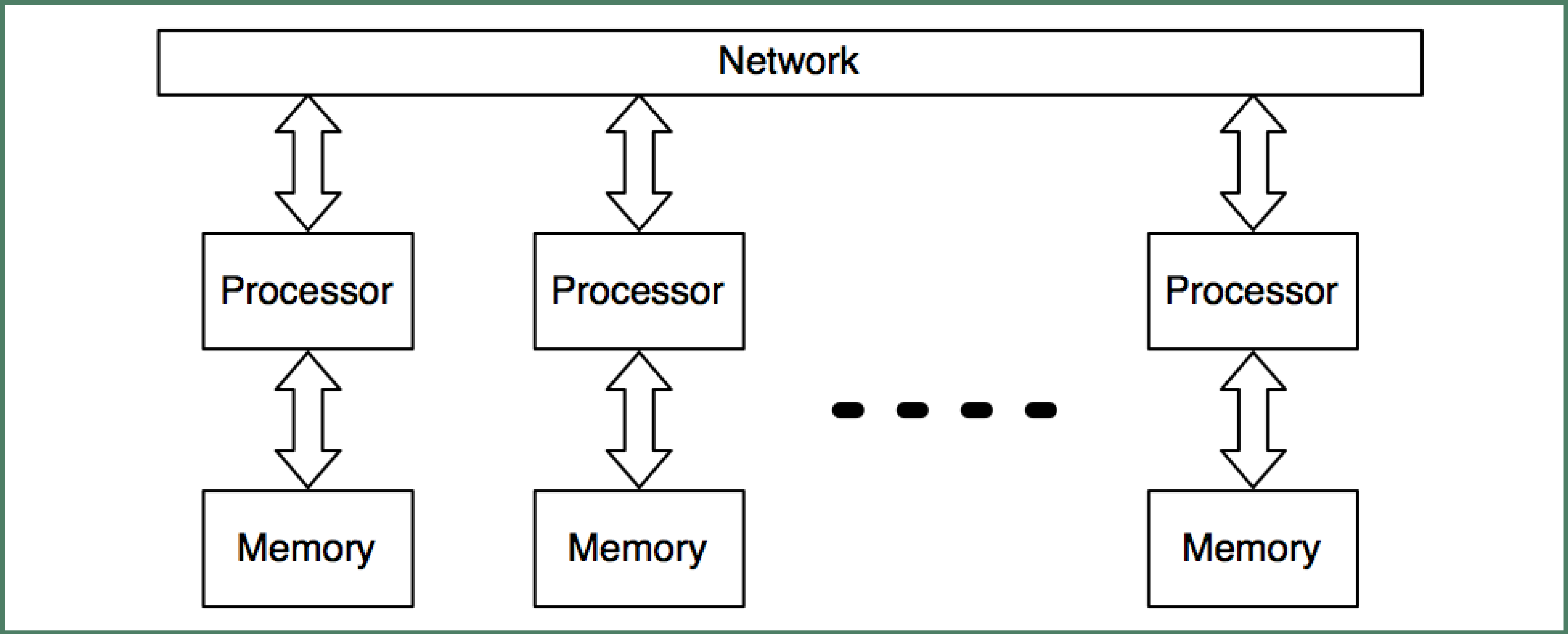
CONCURRENCY AND PARALLELISM

SHARED MEMORY



CONCURRENCY AND PARALLELISM

DISTRIBUTED MEMORY



CONCURRENCY AND PARALLELISM

CONCURRENT SOFTWARE FOR A CONCURRENT WORLD

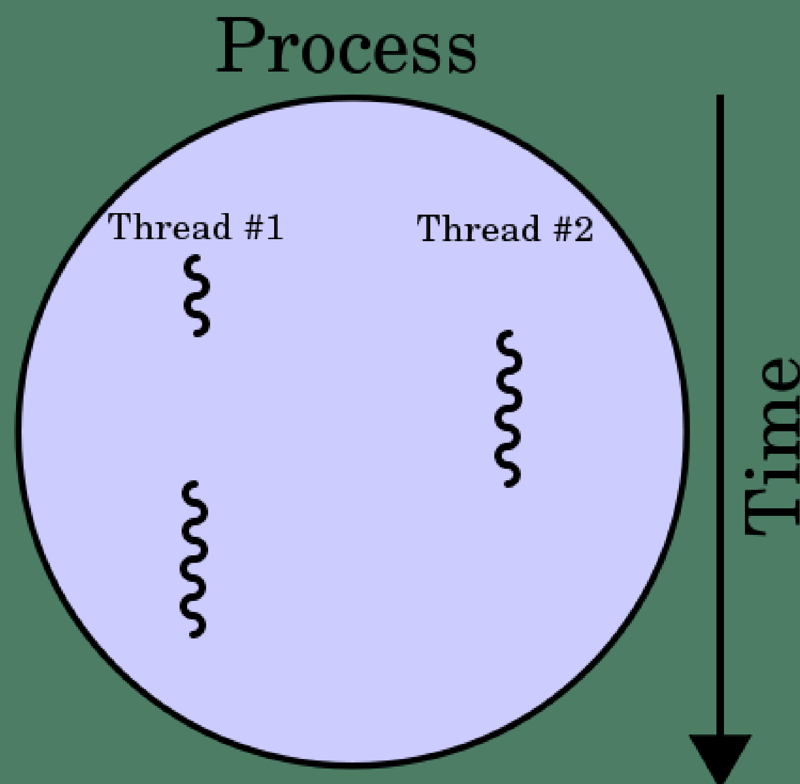
DISTRIBUTED SOFTWARE FOR A DISTRIBUTED WORLD

FAULT-TOLERANT SOFTWARE FOR AN UNPREDICTABLE WORLD

THREADS AND LOCKS

THREADS AND LOCKS

PROCESS
THREAD



```java
public class Counting {
    public static void main(String[] args) throws InterruptedException {
        class Counter {
            private int count = 0;
            public void increment() { ++count; }
            public int getCount() { return count; }
        }
        final Counter counter = new Counter();

        class CountingThread extends Thread {
            public void run() {
                for(int x = 0; x < 10000; ++x)
                    counter.increment();
            }
        }

        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();

        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println(counter.getCount());
    }
}
```

COUNT != 20000

THREADS AND LOCKS: PROBLEMS

HEISENBUGS

RACE CONDITIONS

THREADS AND LOCKS: LOCKS

MUTUAL EXCLUSION (MUTEX)

SEMAPHORE

HIGH-LEVEL SYNCHRONIZATION

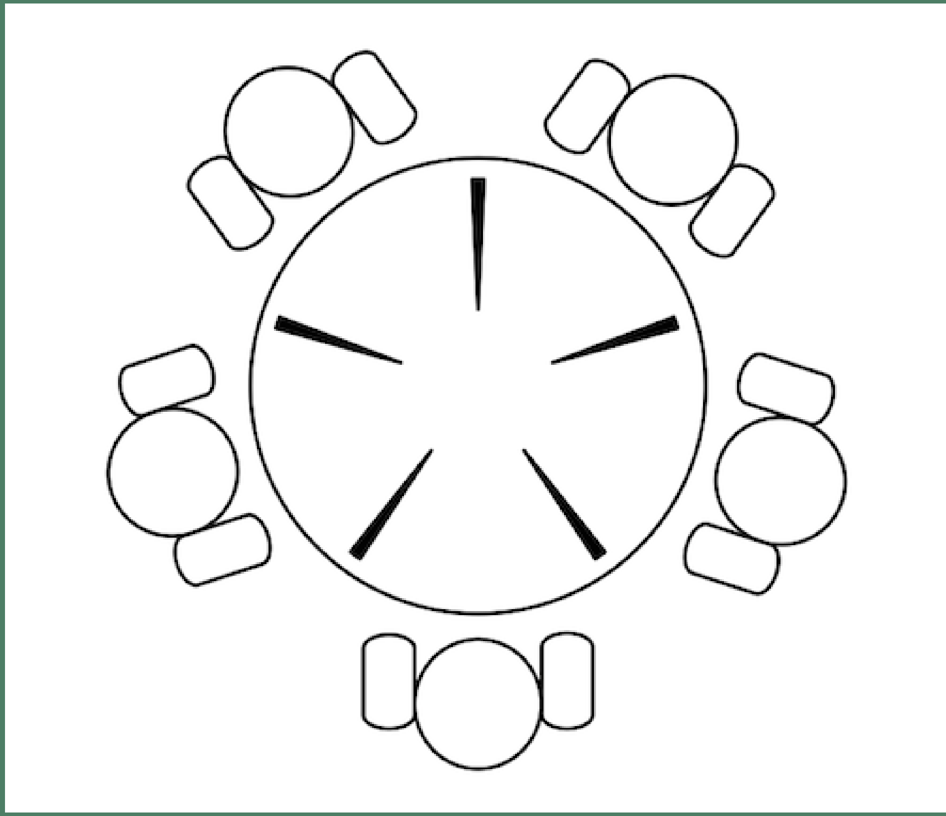
THREADS AND LOCKS: LOCKS

```
class Counter {
    private int count = 0;
    public synchronized void increment() { ++count; }
    public int getCount() { return count; }
}
```

COUNT == 20000

THREADS AND LOCKS: MULTIPLE LOCKS

"DINING PHILOSOPHERS" PROBLEM



DEADLOCK!

THREADS AND LOCKS: MULTIPLE LOCKS

DEADLOCK

SELF-DEADLOCK

LIVELOCK

THREADS AND LOCKS: MULTIPLE LOCKS

“DINING PHILOSOPHERS” SOLUTIONS

RESOURCE HIERARCHY SOLUTION

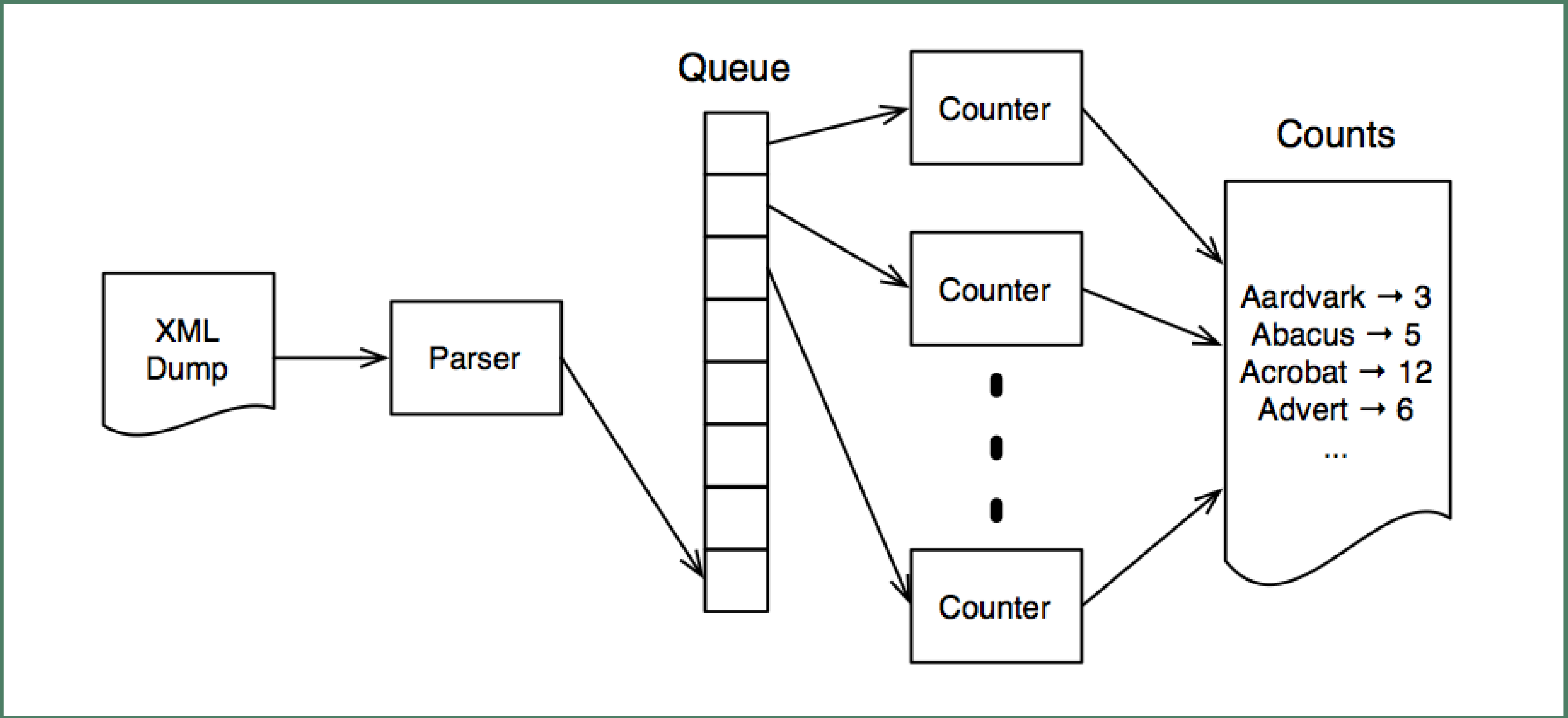
ARBITRATOR SOLUTION

TRY LOCK

THREADS AND LOCKS: WIKIPEDIA PARSER

WHAT'S THE MOST COMMONLY USED WORD ON WIKIPEDIA?

"PRODUCER-CONSUMER" PATTERN



THREADS AND LOCKS: WRAP-UP

STRENGTHS

“CLOSE TO THE METAL”

EASY INTEGRATION

WEAKNESSES

ONLY SHARED-MEMORY ARCHITECTURES

HARD TO MANAGE

HARD TO TESTING

FUNCTIONAL PROGRAMMING

FUNCTIONAL PROGRAMMING

IMMUTABLE STATE

EFFORTLESS PARALLELISM

FUNCTIONAL PROGRAMMING: SUM

```
(defn reduce-sum [numbers]
    (reduce (fn [acc x] (+ acc x)) 0 numbers))
```

```
(defn sum [numbers]
   (reduce + numbers))
```

```
(ns sum.core
    (:require [clojure.core.reducers :as r]))

(defn parallel-sum [numbers]
    (r/fold + numbers))
```

FUNCTIONAL PROGRAMMING: WIKIPEDIA PARSER

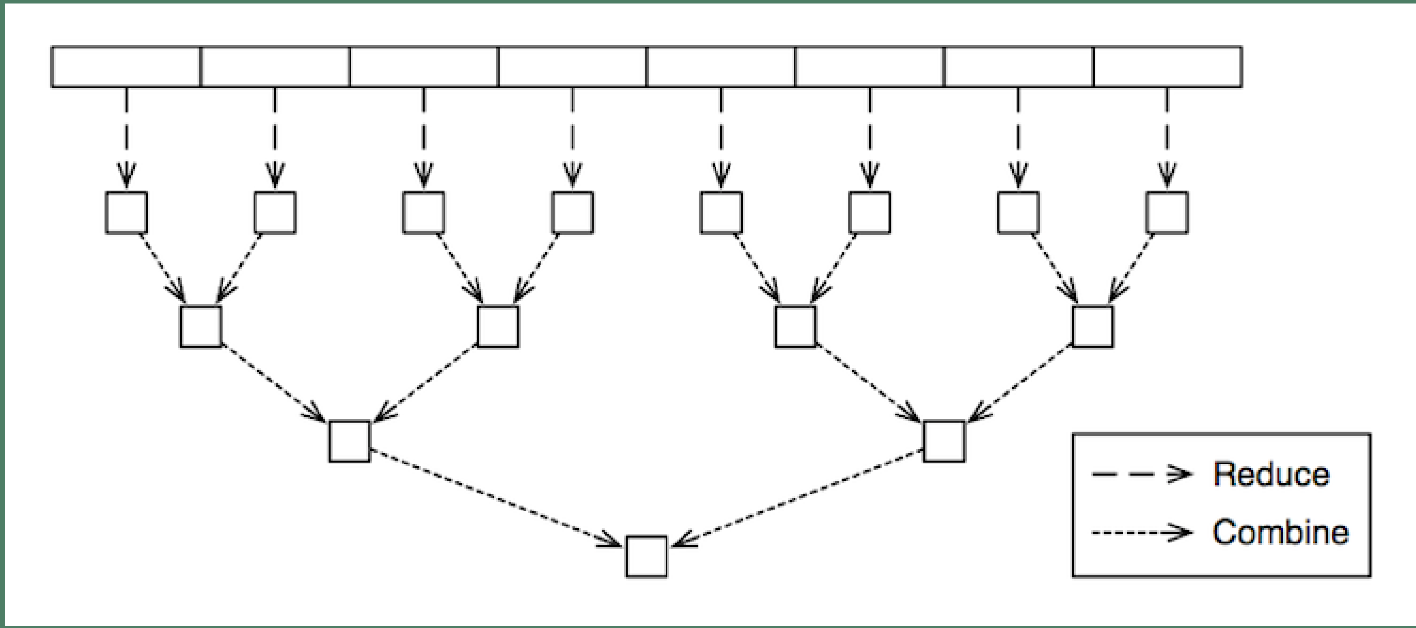
```
(defn count-words-sequential [pages]
    (frequencies (mapcat get-words pages)))
```

```
(pmap #(frequencies (get-words %)) pages)
```

```
(defn count-words-parallel [pages]
    (reduce (partial merge-with +)
        (pmap #(frequencies (get-words %)) pages)))
```

FUNCTIONAL PROGRAMMING: DIVIDE AND CONQUER

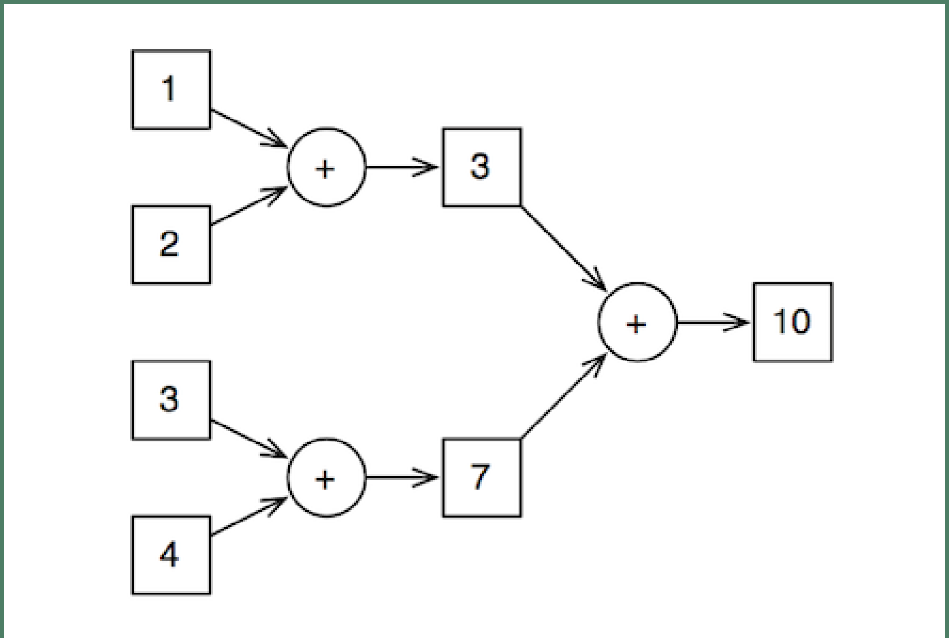
```clojure
(ns sum.core
    (:require [clojure.core.reducers :as r]))

(defn parallel-sum [numbers]
    (r/fold + numbers))
```



FUNCTIONAL PROGRAMMING: REFERENTIAL TRANSPARENCY

```
(+ (+ 1 2) (+ 3 4)) → (+ (+ 1 2) 7) → (+ 3 7) → 10
```

```
(+ (+ 1 2) (+ 3 4)) → (+ 3 (+ 3 4)) → (+ 3 7) → 10
```



FUNCTIONAL PROGRAMMING: WRAP-UP

STRENGTHS

REFERENTIAL TRANSPARENCY

NO MUTABLE STATE

WEAKNESSES

LESS EFFICIENT THAN ITS IMPERATIVE EQUIVALENT

SOFTWARE TRANSACTIONAL MEMORY (STM)

STM

MUTABLE STATE

CAS (COMPARE-AND-SWAP)

TRANSACTIONS ARE ATOMIC, CONSISTENT, AND ISOLATED

STM

```clojure
(defn transfer [from to amount]
    (dosync
        (alter from - amount)
        (alter to + amount)))
```

```clojure
=> (def user1 (ref 1000))

=> (def user2 (ref 2000))

=> (transfer user2 user1 100)
1100

=> @checking
1100

=> @savings
1900
```

STM: WRAP-UP

STRENGTHS

EASY TO USE

WEAKNESSES

RETRYING TRANSACTIONS

SPEED

ACTOR MODEL

ACTOR MODEL

CARL HEWITT (1973)

ACTOR – LIGHTWEIGHT PROCESS

MESSAGES AND MAILBOXES

ACTOR MODEL

```
defmodule Talker do
    def loop do
        receive do
            {:greet, name} -> IO.puts("Hello, #{name}")
            {:bye, status, name} -> IO.puts("Bye, #{status} #{name}")
        end
        loop
    end
end
```

```
pid = spawn(&Talker.loop/0)

send(pid, {:greet, "Gopher"})
send(pid, {:bye, "Mrs", "Pike"})

sleep(1000)
```

```
Hello, Gopher
Bye, Mrs Pike
```

ACTOR MODEL

PATTERN MATCHING

BIDIRECTIONAL COMMUNICATION

NAMING PROCESSES

SUPERVISING A PROCESS

ACTOR MODEL

DISTRIBUTION

CLUSTER

REMOTE MESSAGING

ACTOR MODEL: WRAP-UP

STRENGTHS

MESSAGING AND ENCAPSULATION

FAULT TOLERANCE

DISTRIBUTED PROGRAMMING

WEAKNESSES

WE STILL HAVE DEADLOCKS

OVERFLOWING AN ACTOR'S MAILBOX

COMMUNICATING SEQUENTIAL PROCESSES (CSP)

COMMUNICATING SEQUENTIAL PROCESSES (CSP)

SIR CHARLES ANTONY RICHARD HOARE (1978)

SIMILAR TO THE ACTOR MODEL

FOCUS ON THE CHANNELS

> *Do not communicate by sharing memory, instead share memory by communicating*

Rob Pike

CSP

GOROUTINES

IT'S VERY CHEAP

IT'S NOT A THREAD

COOPERATIVE SCHEDULER VS PREEMPTIVE SCHEDULER

MULTITHREADING, MULTICORE

```
go func()
```

> *Just looked at a Google-internal Go server with 139K goroutines serving over 68K active network connections. Concurrency wins.*

@rob_pike

CSP: CHANNELS

CHANNELS – THREAD-SAFE QUEUE

CHANNELS – FIRST CLASS OBJECT

```
// Declaring and initializing
var ch chan int
ch = make(chan int)
// or
ch := make(chan int)
// Buffering
ch := make(chan int, 100)
```

```
// Sending on a channel
ch <- 1
```

```
// Receiving from a channel
value = <- ch
```

CSP

EXAMPLE

```go
func main() {
    jobs := make(chan Job)
    done := make(chan bool, len(jobList))

    go func() {
        for _, job := range jobList {
            jobs <- job // Blocks waiting for a receive
        }
        close(jobs)
    }()

    go func() {
        for job := range jobs { // Blocks waiting for a send
            fmt.Println(job) // Do one job
            done <- true
        }
    }()

    for i := 0; i < len(jobList); i++ {
        <-done // Blocks waiting for a receive
    }
}
```

CSP: WRAP-UP

STRENGTHS

FLEXIBILITY

NO CHANNEL OVERFLOWING

WEAKNESSES

WE CAN HAVE DEADLOCKS

GO CONCURRENCY: WRAP-UP

STRENGTHS

MESSAGE PASSING (CSP)

STILL HAVE LOW-LEVEL SYNCHRONIZATION

DON'T WORRY ABOUT THREADS, PROCESSES

WEAKNESSES

NIL

WRAPPING UP

THE FUTURE IS IMMUTABLE

THE FUTURE IS DISTRIBUTED

THE FUTURE WITH BIG DATA

USE RIGHT TOOLS

DON’T WRITE DJANGO/ROR BY GO/CLOJURE/ERLANG

LINKS

BOOKS:

- “Seven Concurrency Models in Seven Weeks”, 2014, by Paul Butcher
- “Communicating Sequential Processes”, 1978, C. A. R. Hoare

OTHER:

- “Concurrency Is Not Parallelism” by Rob Pike (http://goo.gl/hyFmcZ)
- “Modern Concurrency” by Alexey Kachayev (http://goo.gl/Tr5USn)
- A Tour of Go (http://tour.golang.org/)

THE END

THANK YOU FOR ATTENTION!

- Vasyl Nakvasiuk
- Email: vaxxxa@gmail.com
- Twitter: @vaxXxa
- Github: vaxXxa

THIS PRESENTATION:

Source: https://github.com/vaxXxa/talks

Live: http://vaxXxa.github.io/talks