Models for concurrent programming for the future

Tobias Ivarsson

hacker @ Neo Technology

tobias@neotechnology.com
twitter: @thobe
web: http://thobe.org/ http://neo4j.org/

Common misconceptions

More threads

More throughput

Finite number of cores

I/O is only one pipe

Locking always impedes performance

Concurrency on the track, only one in the station

Amdahl's law

$$\text{Speedup} \leq \frac{1}{F + \frac{1-F}{N}}$$

N: Number of processors

F: Serial fraction

Throughput with synchronized regions

Throughput: 3

Throughput: 6

Throughput: 9

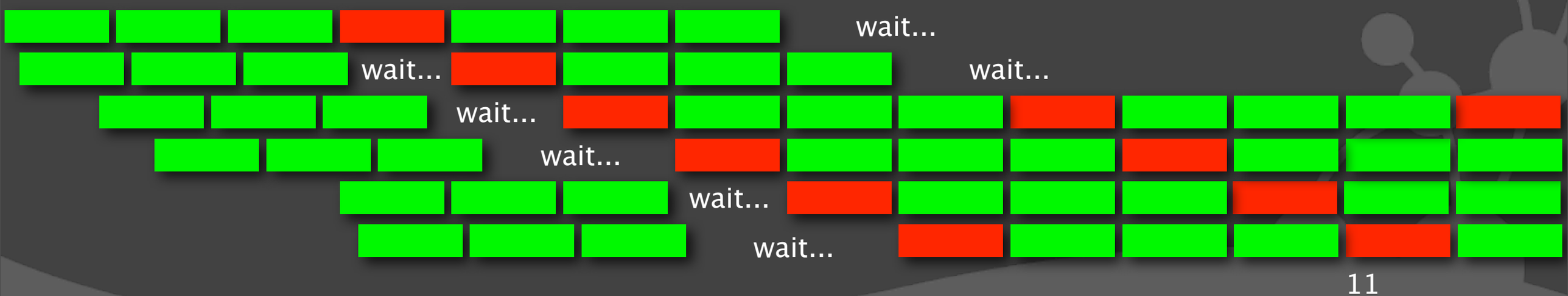
Throughput: 11

Throughput with synchronized regions

Throughput: 11

Throughput with synchronized regions

Throughput: 11



Concurrency abstractions

Java of today

Threads

Monitors

```
synchronized (someMonitor)
{  read barrier
     // single threaded region
}  write barrier
```

Volatile read/write

```
class Thing {
     volatile String name;
     String getName()  {
          return this.name; read barrier
     }
     void setName( String newName ) {
          this.name = newName; write barrier
     }
}
```

java.util.concurrent

```
ConcurrentMap<String,Thing> map =
new ConcurrentHashMap();
```

```
List<Thing> list = new
CopyOnWriteArrayList();
```

```
ExecutorService executor = new
  ThreadPoolExecutor();


executor.submit(new Runnable()  {
      void run()  {
          doWork();
      }
});
```

```
final CyclicBarrier barrier = new
  CyclicBarrier(4);
Thread[] testThreads = new Thread[4];
for(int i = 0;i<testThreads.length;i++)
   testThreads = new Thread(){
      void run() {
        barrier.await();
        // all threads will exit await
        // (roughly) at the same time
        doWork();
   }
}.start();
```

java.util.concurrent.locks

```
LockSupport.park();

LockSupport.unpark();
```


```
Lock lock = new ReentrantLock();

ReadWriteLock rwLock = new ReentrantReadWriteLock();

Lock read = rwLock.readLock();
Lock write = rwLock.writeLock();
```

java.util.concurrent.atomic

```
AtomicReference<Thing> ref = new AtomicReference();

AtomicReferenceArray<Thing> array = new AtomicReferenceArray();

class MyAtomicThingReference {
    volatile Thing thing;
    static AtomicReferenceFieldUpdater
        <MyAtomicThingReference,Thing> THING = newUpdater(...);

    Thing swap( Thing newThing ) {
        return THING.getAndSet( this, newThing );
    }
}
```

Java of Tomorrow

ForkJoin

Models not in Java today

ParallelArray


```
ParallelArray<Student> students = ...

double highestScore = students
    .filter( #{ Student s -> s.gradYear == 2010 } )
    .map(    #{ Student s -> s.score } )
    .max();
```

Transactional Memory

```
void transfer( TransactionalLong source,
               TransactionalLong target,
               long amount ) {

  Transaction tx = txManager.beingTransaction();
  try {

    long sourceFunds = source.getValue();
    if (sourceFunds < amount) {
      throw new InsufficientFundsException();
    }
    source.setValue( sourceFunds - amount );
    target.setValue( target.getValue() + amount );

    tx.success();
  } finally {
    tx.finish();
  }
}
```

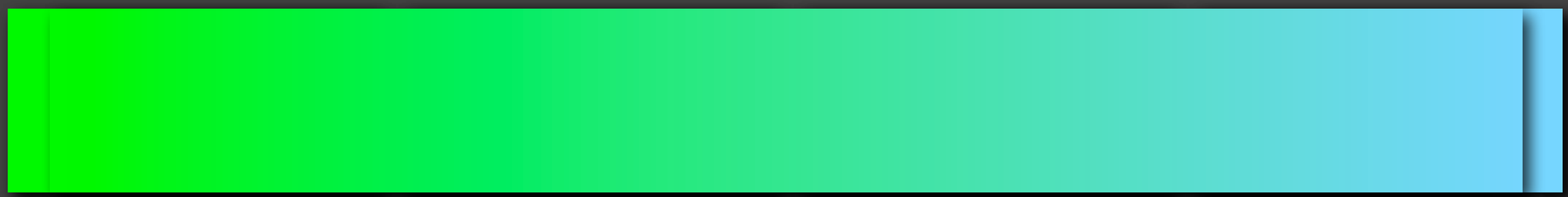
Actors


```
class HelloWorldActor extends Actor {
  def receive = {
    case msg => self reply ("Hello " + msg)
  }
}
```

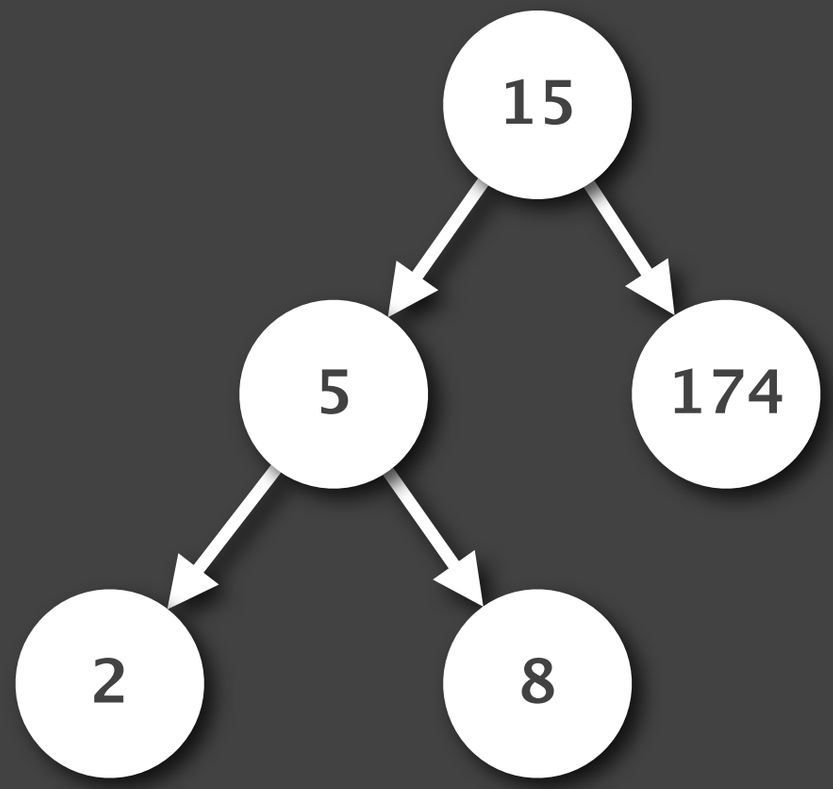
```
val response = helloActor !! "World"
```

Scala Parallel Collection Framework

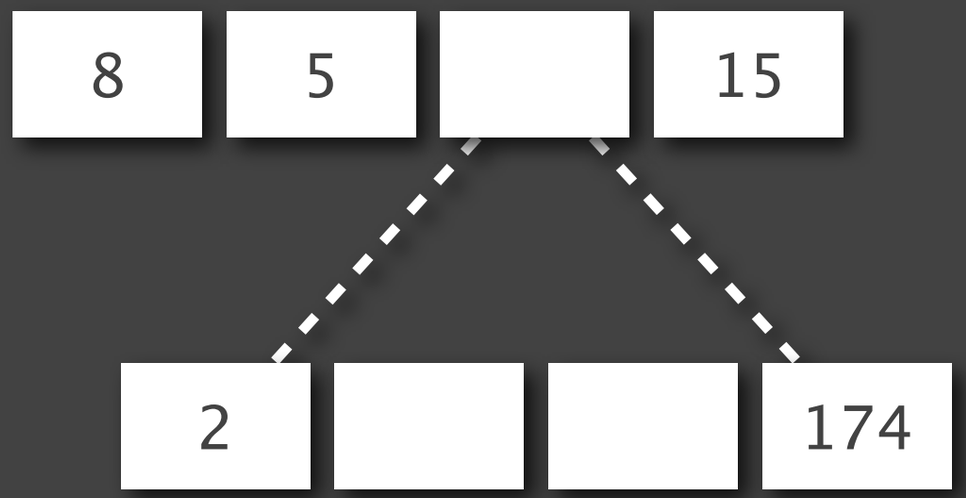
Efficient Collection splitting & combining



Splittable maps



Splittable maps



Hash tries

- Similar performance to hash tables
- Splittable (like arrays)
- Memory characteristics more like trees
- No resizing races

Clojure

Immutable by default

Thread local (and scoped) override

```clojure
(def x 10) ; create a root binding

(def x 42) ; redefine the root binding

(binding [x 13] ...) ; create a thread local override
```

Explicit *atomic* mutability

Transactional memory

(ref) *and* (dosync ...)


```
(def street (ref))
(def city (ref))


(defn move [new-street new-city]
  (dosync ; synchronize the changes
    (ref-set street new-street)
    (ref-set city new-city)))
```

```
(defn print-address []
  (dosync ; get a snapshot state of the refs
    (printf "I live on %s in %s%n" @street @city)))
```

(atom)

easier API for AtomicReference


```
(defstruct address-t :street :city)

(def address (atom))

(defn move [new-street new-city]
  (set address (struct new-street new-city)))
```

```
(defn print-address []
  (let [addr @address] ; get snapshot
    (printf "I live on %s in %s%n"
            (get addr :street)
            (get addr :city))))
```

Explicit *asynchronous* updates

(agent)

A simpler cousin to actors

```
(def account (agent))
```

```
(defn deposit [balance amount]
  (+ balance amount))

(send account deposit 1000)
```

Questions?