Concurrency

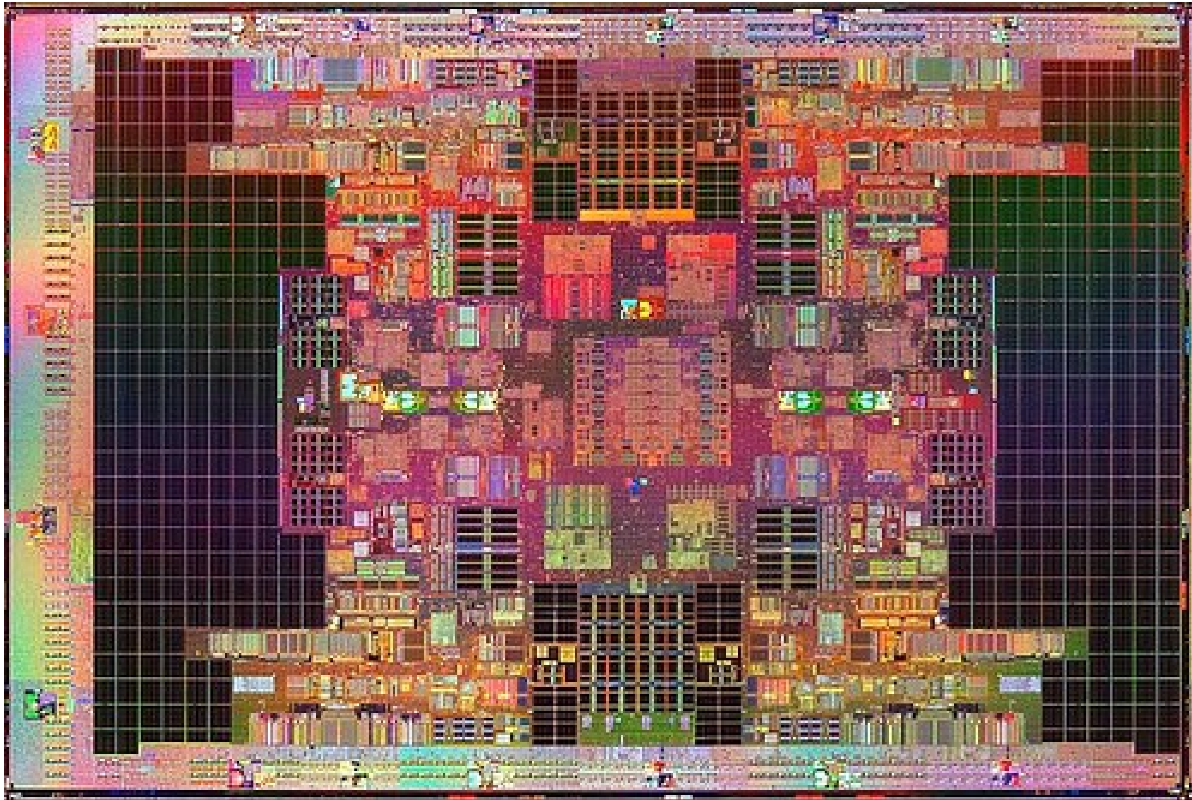
Unleash your processor(s)

Václav Pech

The evil accumulator

```
int sum=0;
for(int i : collection) sum += i;
```

We’re quad core already

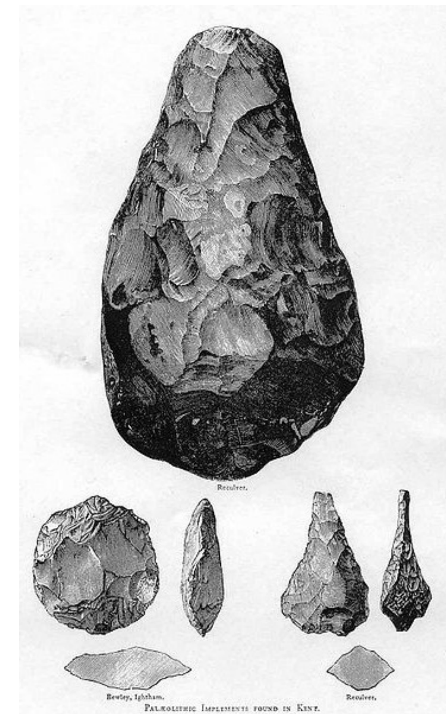


Beware: More cores to come shortly!

Stone age of parallel SW

- Dead-locks
- Live-locks
- Race conditions
- Starvation

- Shared Mutable State



Lock and synchronize

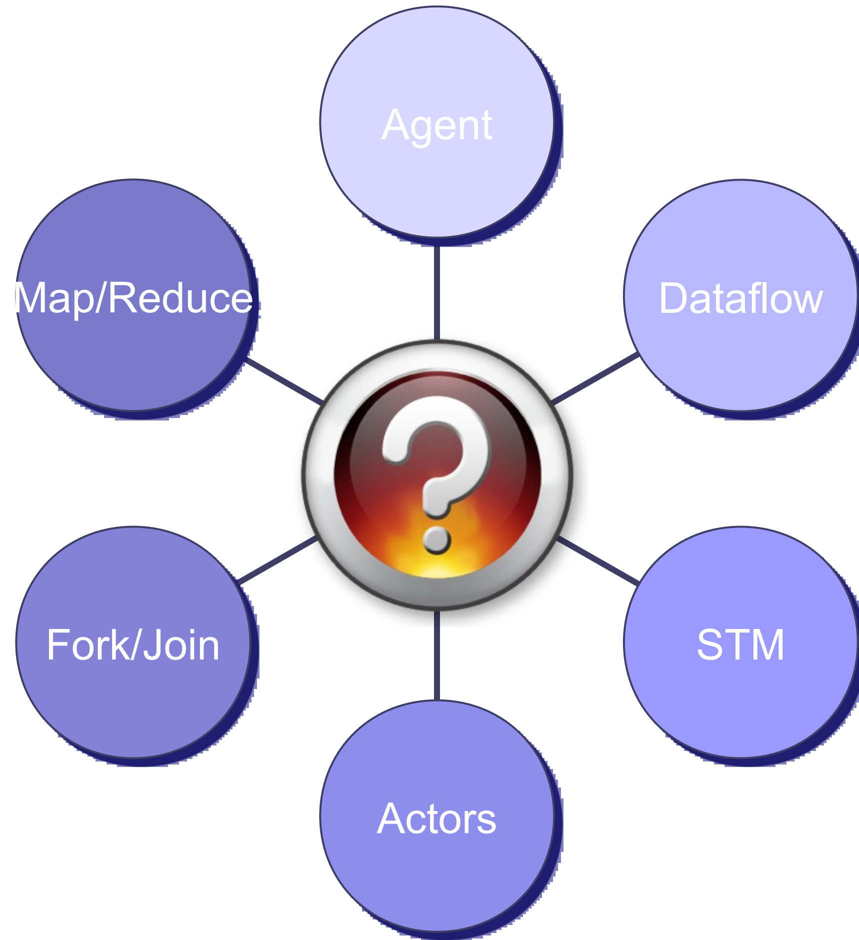
Multithreaded programs today work mostly by accident!

Making code transactional

```
@Transactional
public void findAllFoosAround() {
    //tx agnostic code here
}
```

```
withTransaction {
    //tx agnostic code here
}
```

Can we do better?



Collections (GPars)

```
images.eachParallel {it.process()}

documents.sumParallel()

candidates.maxParallel {it.salary}.marry()
```

Parallel Arrays (jsr-166y)

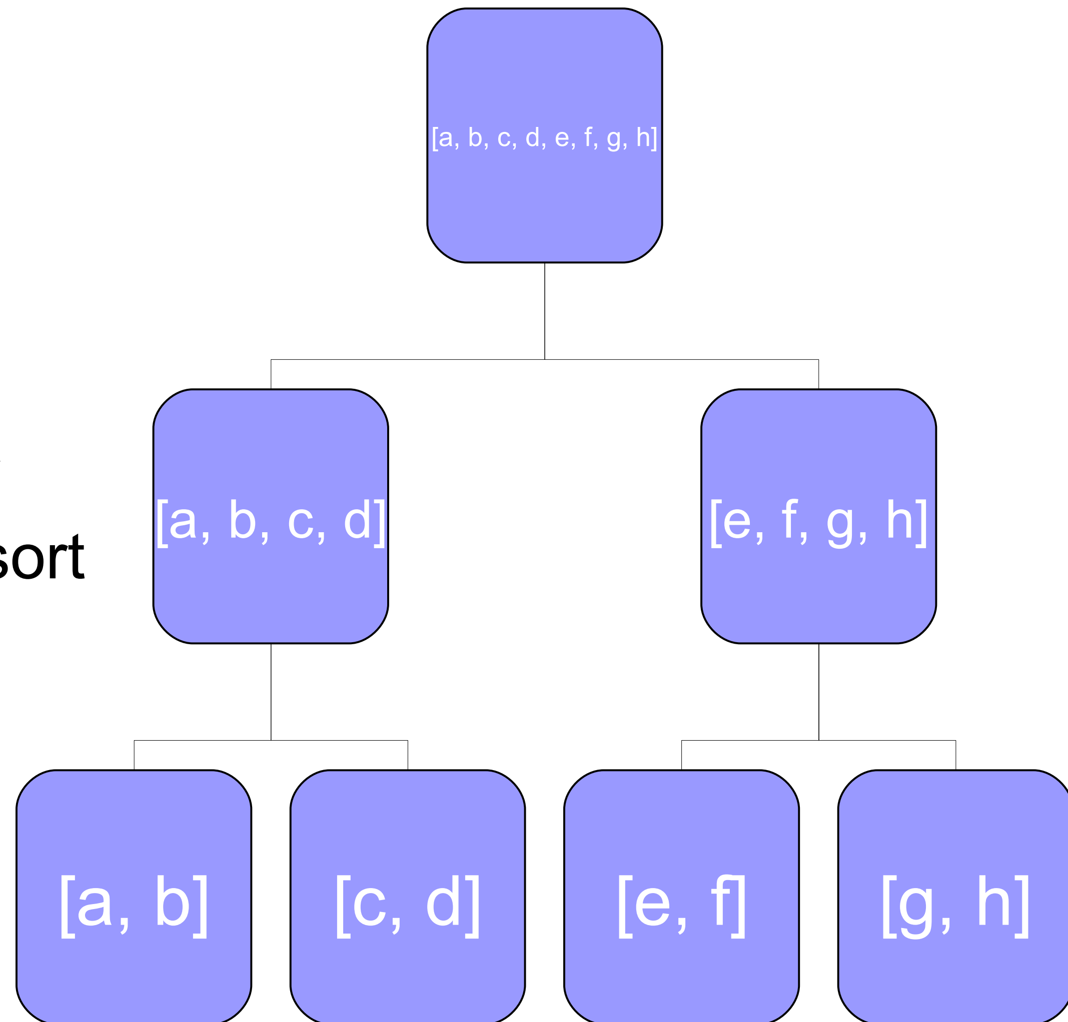
```
ParallelArray namesOfWomen =
   people.withFilter(aWoman).withMapping(retrieveName).all();
```

```
Ops.Predicate aWoman = new Ops.Predicate() {
  public boolean op(Person friend) {return !friend.isMale();}
};

Ops.Op retrieveName = new Ops.Op() {
  public Object op(Person friend) {return friend.getName();}
};
```

Fork/Join

- Solve hierarchical problems
  - Divide and conquer
    - Merge sort, Quick sort
    - Tree traversal
    - File scan / search
    - …

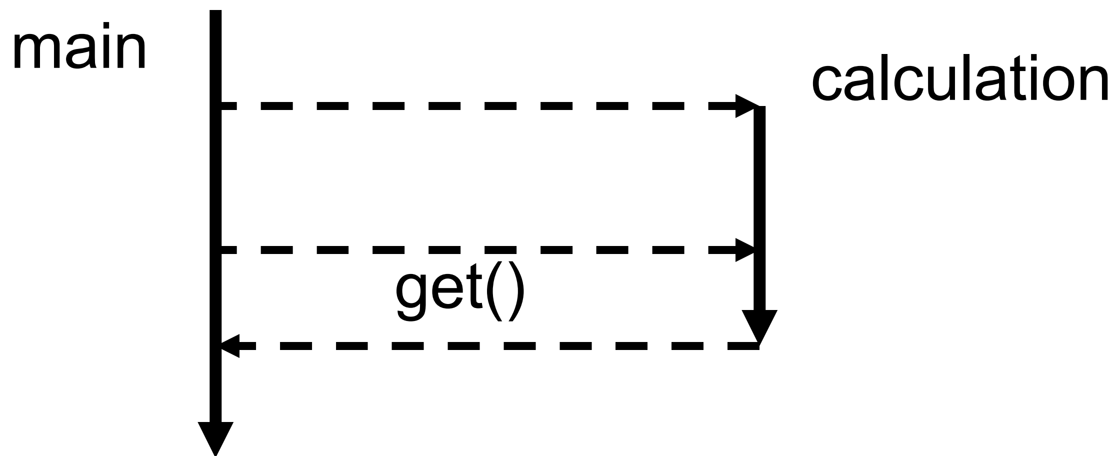


Fork/Join (GPars)

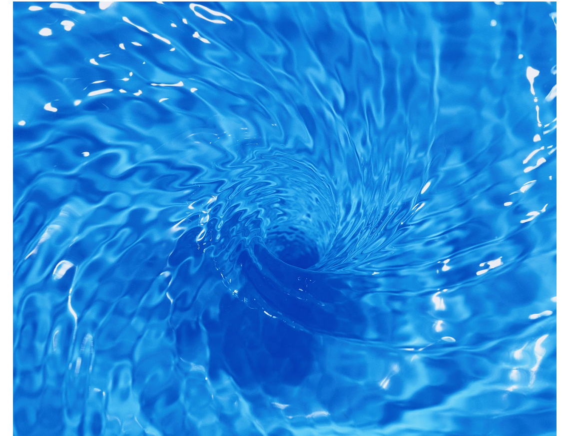
```
protected void computeTask() {
    long count = 0;
    file.eachFile {
        if (it.isDirectory()) {
            println "Forking a thread for $it"
            forkOffChild(new FileCounter(it))
        } else {
            count++
        }
    }
    return count + childrenResults.sum()
}
```

Asynchronous invocation

```
Future f = threadPool.submit(calculation);
…
System.out.println("Result: " + f.get());
```



Dataflow Concurrency



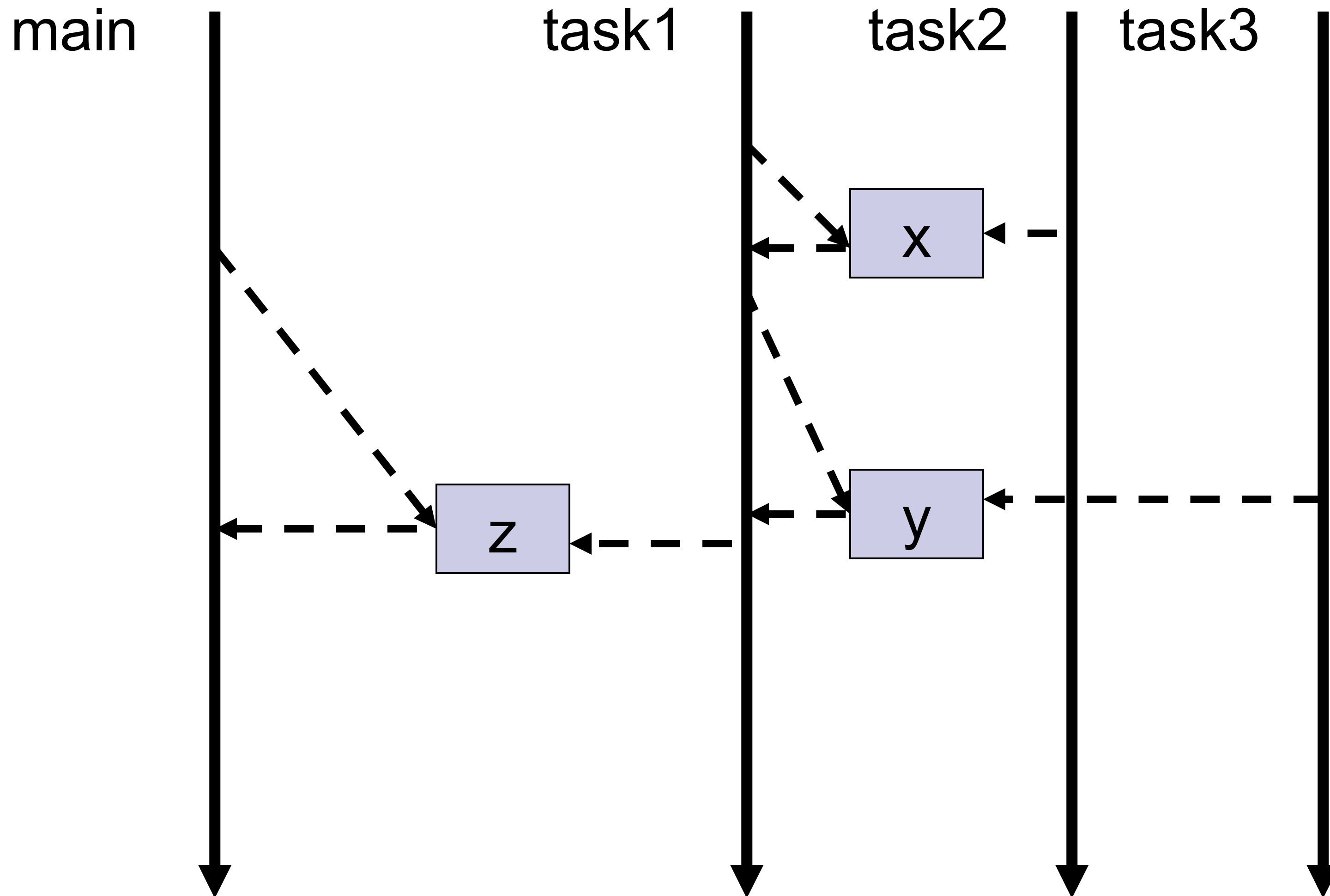
- No race-conditions
- No live-locks
- Deterministic deadlocks

Completely deterministic programs

BEAUTIFUL code

(Jonas Bonér)

Dataflow Variables / Promises

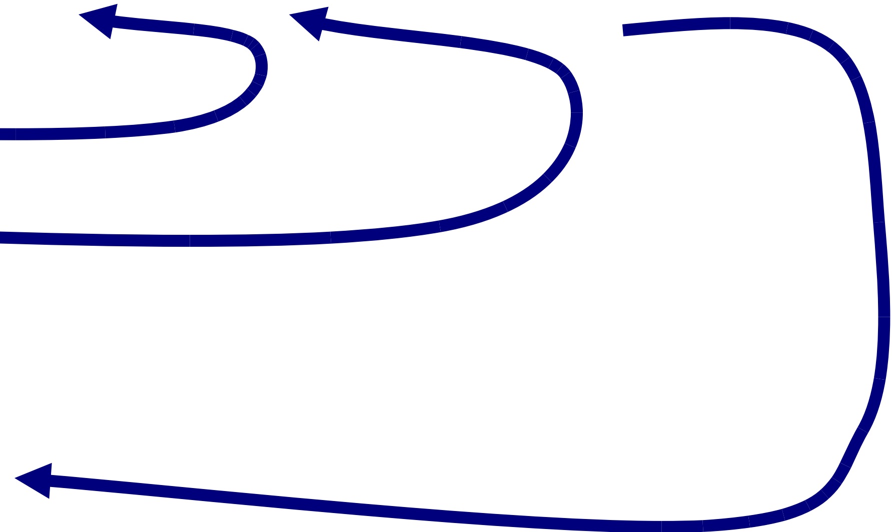


DataFlows (GPars)

```
def df = new DataFlows()

task { df.z = df.x + df.y }
task { df.x = 10 }
task { df.y = 5 }

assert 15 == df.z
```



Promises (Clojure)

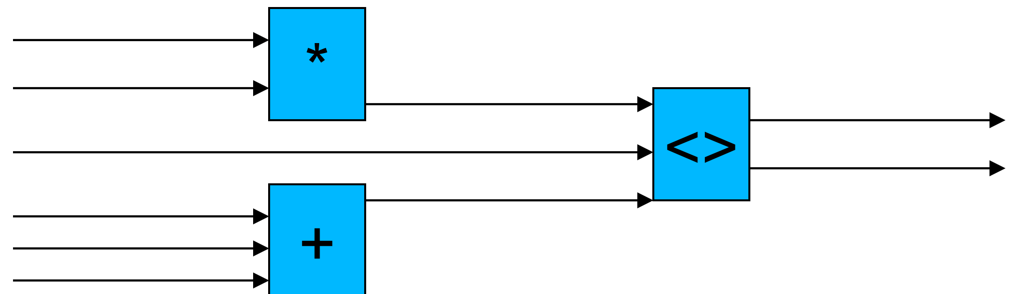
```
(def x (promise)) (def y (…))(def z (…))

(future (deliver z (+ @x @y)))
(future (deliver x 10))
(future (deliver y 5))

(println @z)
```

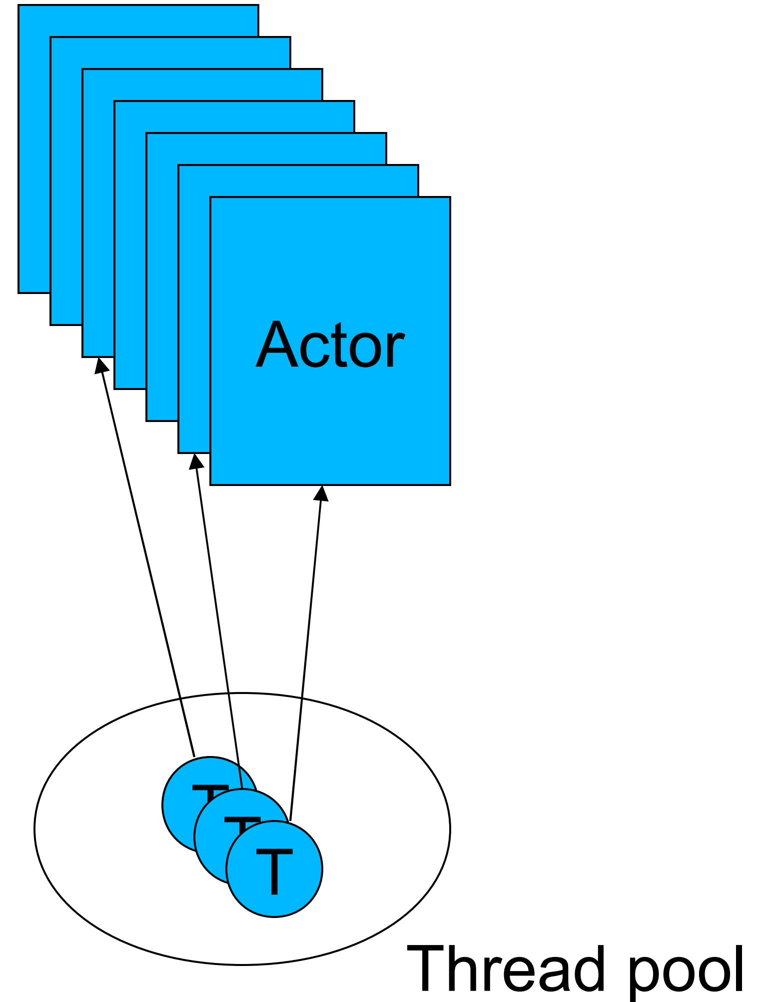
Dataflow Operators (GPars)

```
operator(inputs: [stocksStream],
            outputs: [pricedStocks])
  {stock ->
    def price = getClosing(stock, 2008)
    bindOutput(0, [stock: stock, price: price])
  }
```

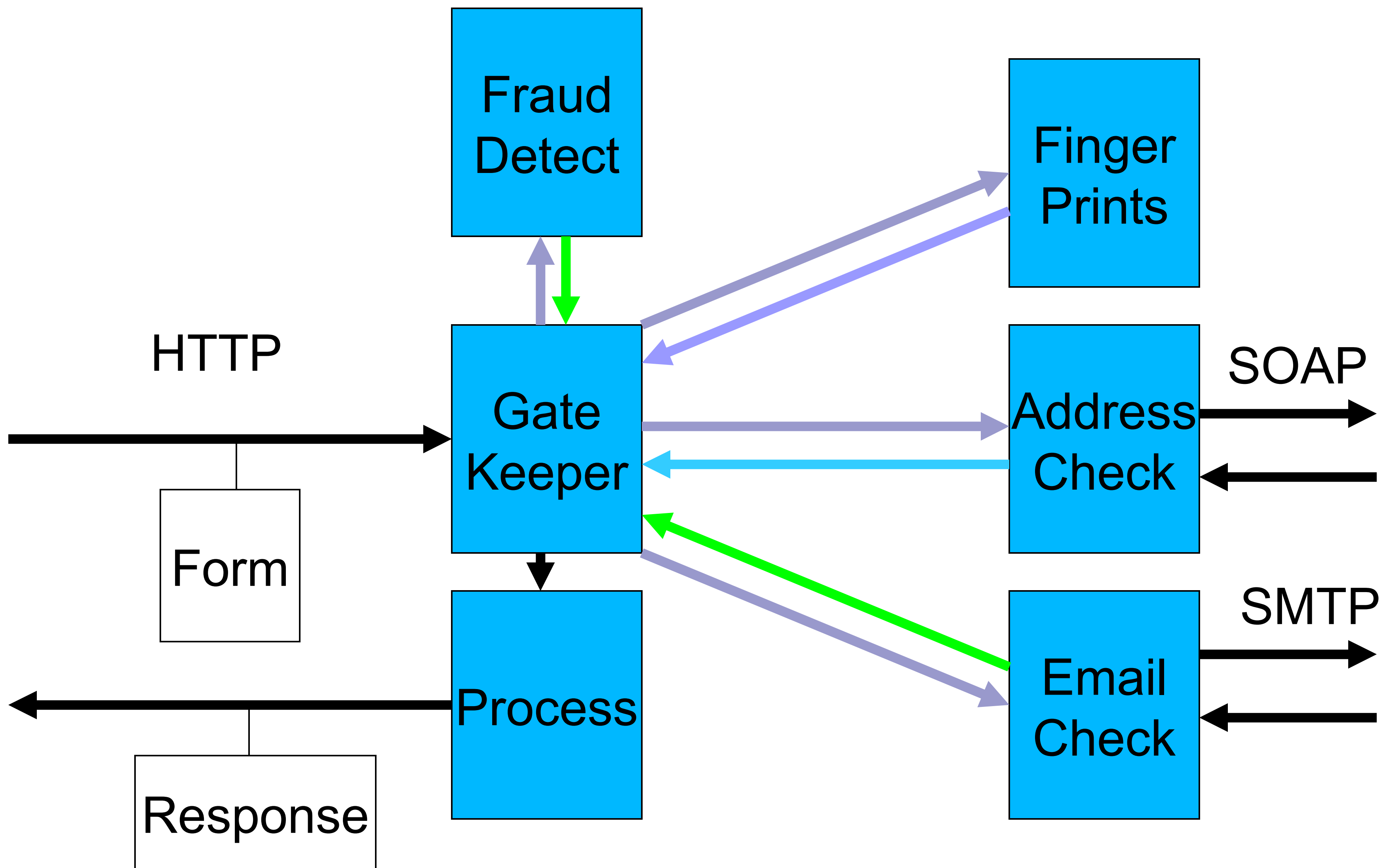


Actors

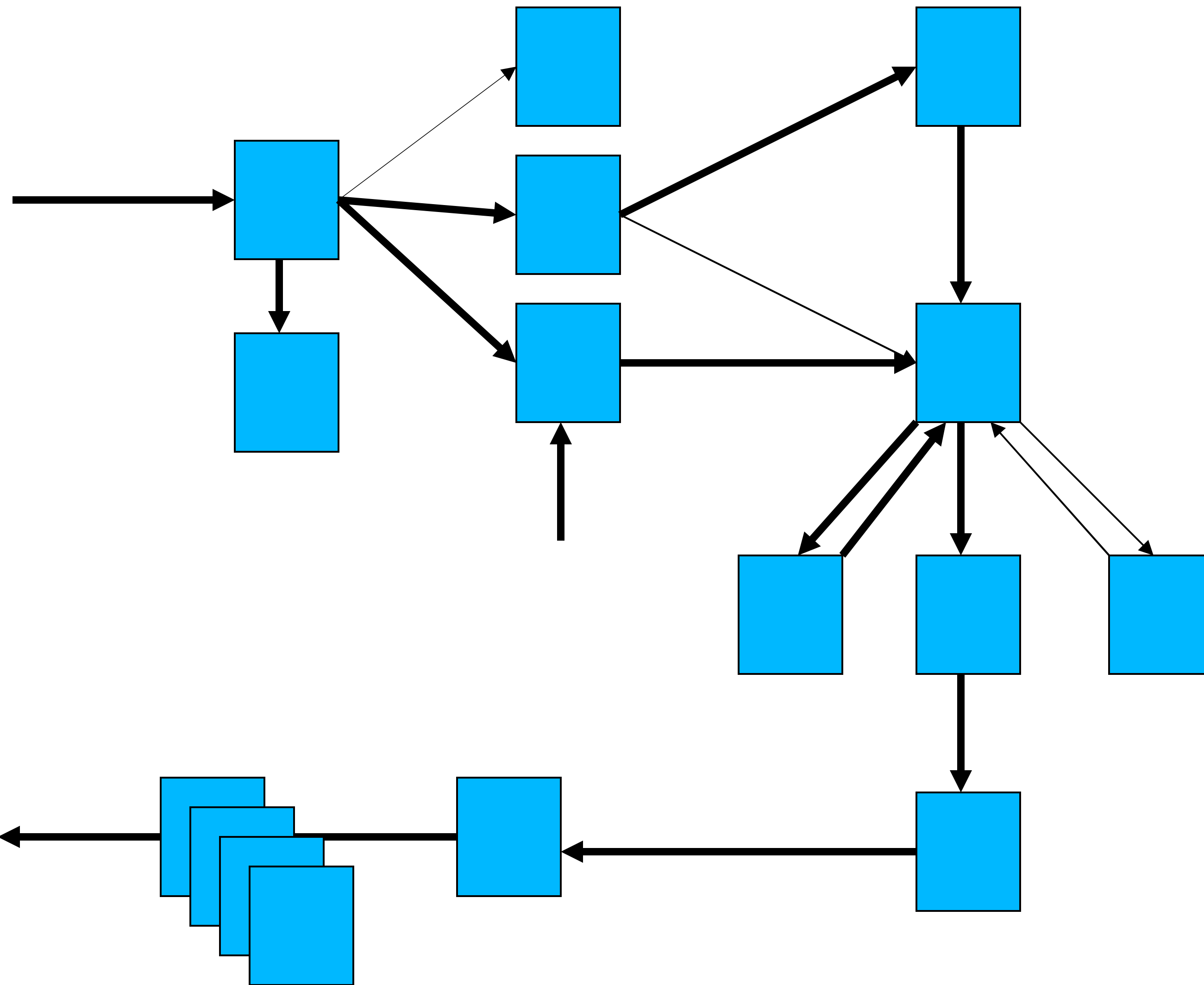
- Isolated
- Communicating
  - Immutable messages
- Active
  - Pooled shared threads
- Activities
  - Create a new actor
  - Send a message
  - Receive a message



Actors use



Actors patterns



Enricher

Router

Translator

Endpoint

Splitter

Agregator

Filter

Resequencer

Checker

Creating Actors (GPars)

```
class MyActor extends AbstractPooledActor {
    void act() {
        def buddy = new YourActor()
        buddy << ‘Hi man, how\’re things?’
        def response = receive()
    }
}
```

Creating Actors (Scala)

```
val myActor = actor {
    loop {
        react {
            case x:Int=>reply 2*x
            case s:String=>reply s.toUpperCase()
        }
    }
}
myActor ! 10
```

Creating Actors (Akka - Scala)

```
class MyActor extends Actor {
  def receive = {
    case "Hello" => reply("World")
    case Register(user, sender) =>
      val registrationCode = register(user)
      sender ! SuccessfulRegistration(registrationCode)
    case _ => throw new RuntimeException("unknown
  message")
} }
```

Example form http://doc.akkasource.org/actors

Continuation Style

```
loop {
  …
  react {
    …
    react {/*schedule the block; throw CONTINUE*/
      …
    }
    //Never reached
  }
  //Never reached
}
//Never reached
```

Creating Actors (Jetlang)

```
Fiber fiber = new ThreadFiber();
fiber.start();
Channel<String> channel = new MemoryChannel<String>();

Callback<String> runnable = new Callback<String>() {
    public void onMessage(String msg) {
        log(msg.toUpperCase());
    }
};
channel.subscribe(fiber, runnable);
channel.publish("hello");
```

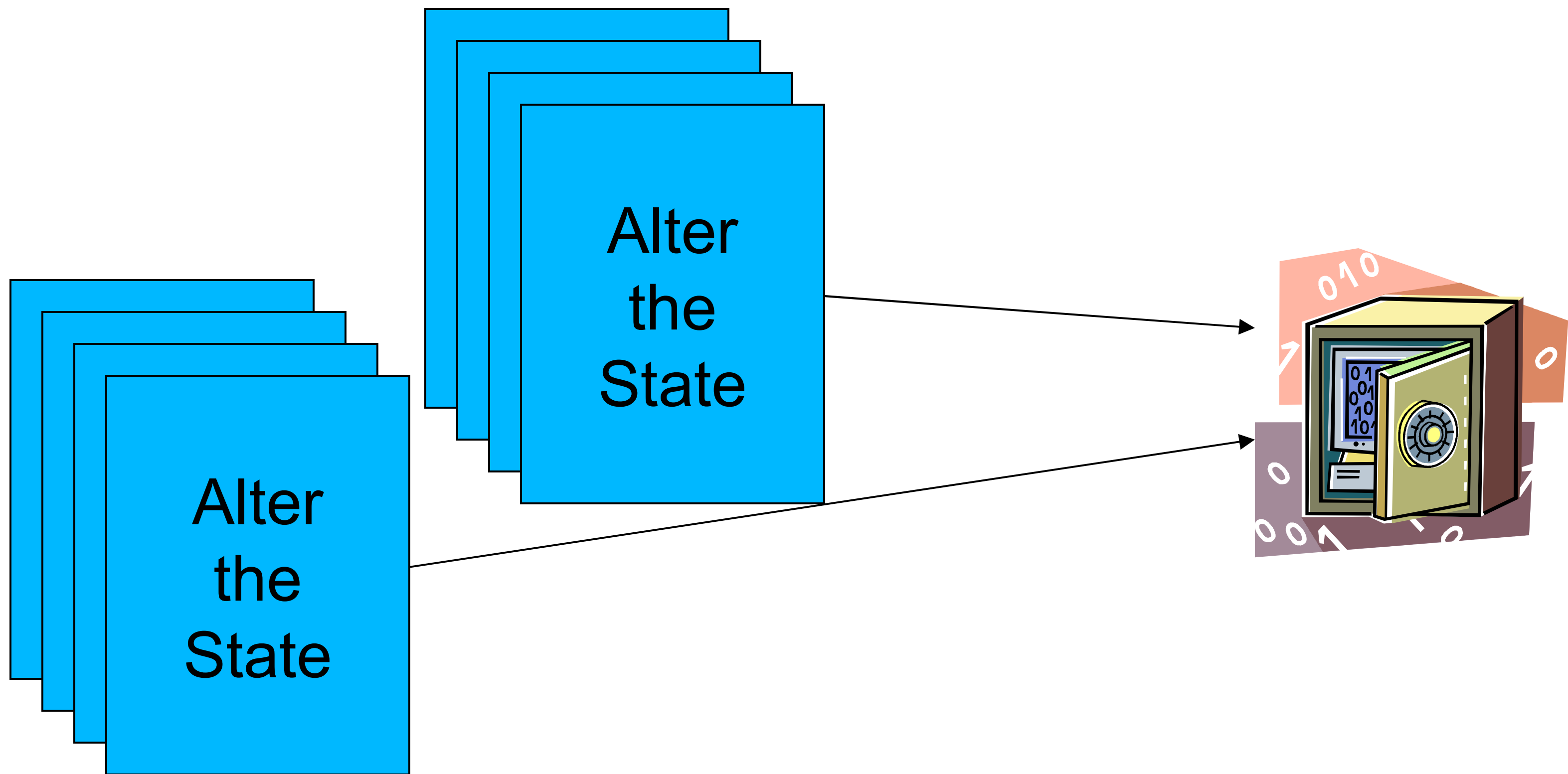
Based on sample code from http://code.google.com/p/jetlang/

Java actor frameworks

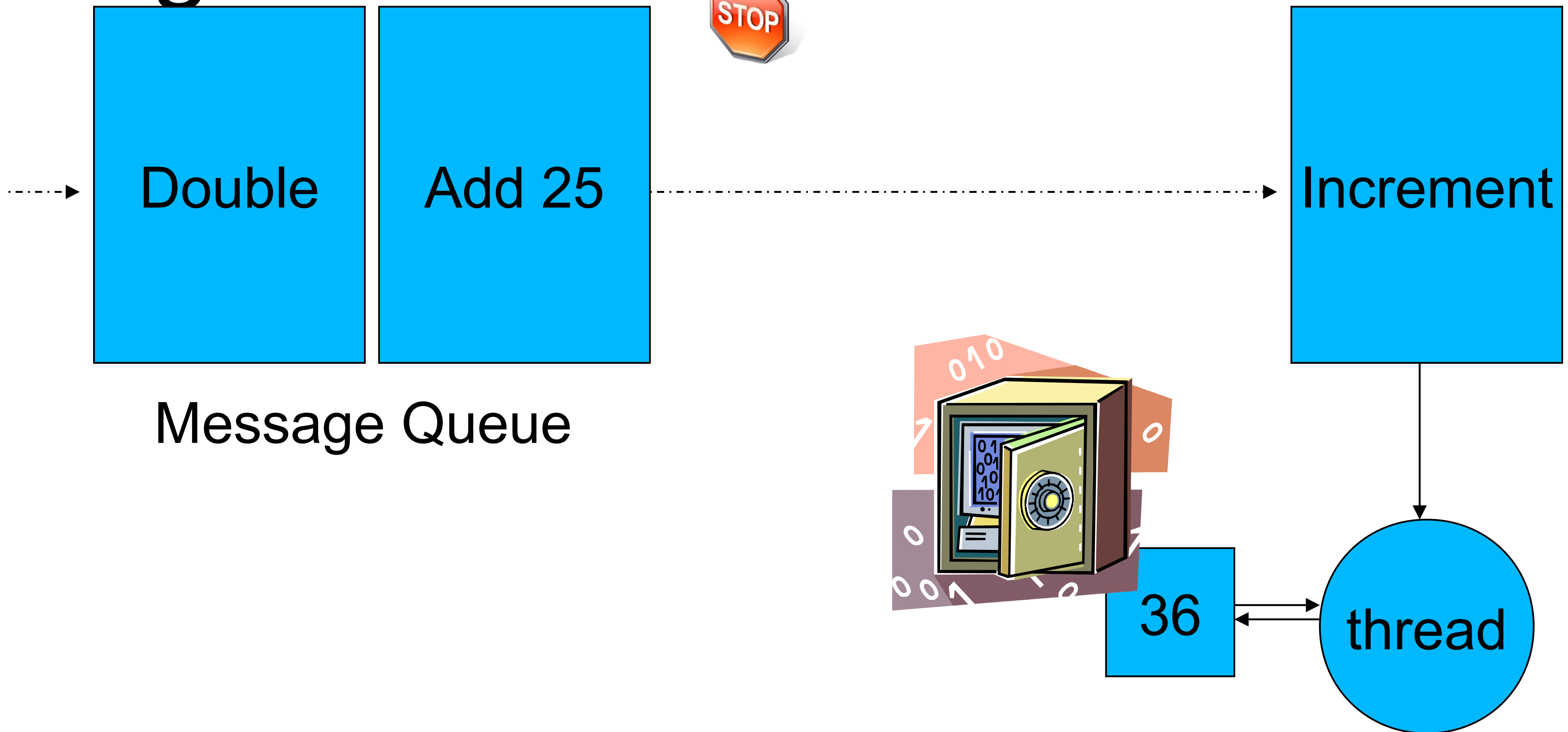
Jetlang
Kilim
ActorFoundry
Actorom
Akka
…

Agent

- Lock Shared Mutable State in a Safe



Agent inside



Agent (Clojure)

```
(defn increment  [c] (+ c 1))
(defn decrement [c] (- c 1))
(defn add [c delta] (+ c delta))

(def a (agent 0))

(send a increment) | (send a decrement) | (send a add 25) | …

(await a)         (println @a)
```

Agent (ScalaAgent)

```
def increment(x: Long) = x + 1
def decrement(delta : Long)(x: Long) = x - delta
val agent = Agent(0L)

agent(increment _)
agent(decrement(3) _)
agent{_ + 100}

println(agent get)
```

STM (Clojure)

```
(defstruct person :name :gender :age)
(def people (ref #{}))

(dosync
  (alter people conj (struct person "Joe" "male" 39))
  (alter people conj (struct person "Alice" "female" 27))
  (alter people conj (struct person "Dave" "male" 41))
)
```

STM (Akka - Scala)

```
atomic {
    .. // do something within a transaction
}

atomic(maxNrOfRetries) { .. }
atomicReadOnly { .. }

atomically {
    .. // try to do something
} orElse {
    .. // if tx clash; try do do something else
}
```

Sample taken from Akka documentation - http://doc.akkasource.org/stm

Persistent Data Structures

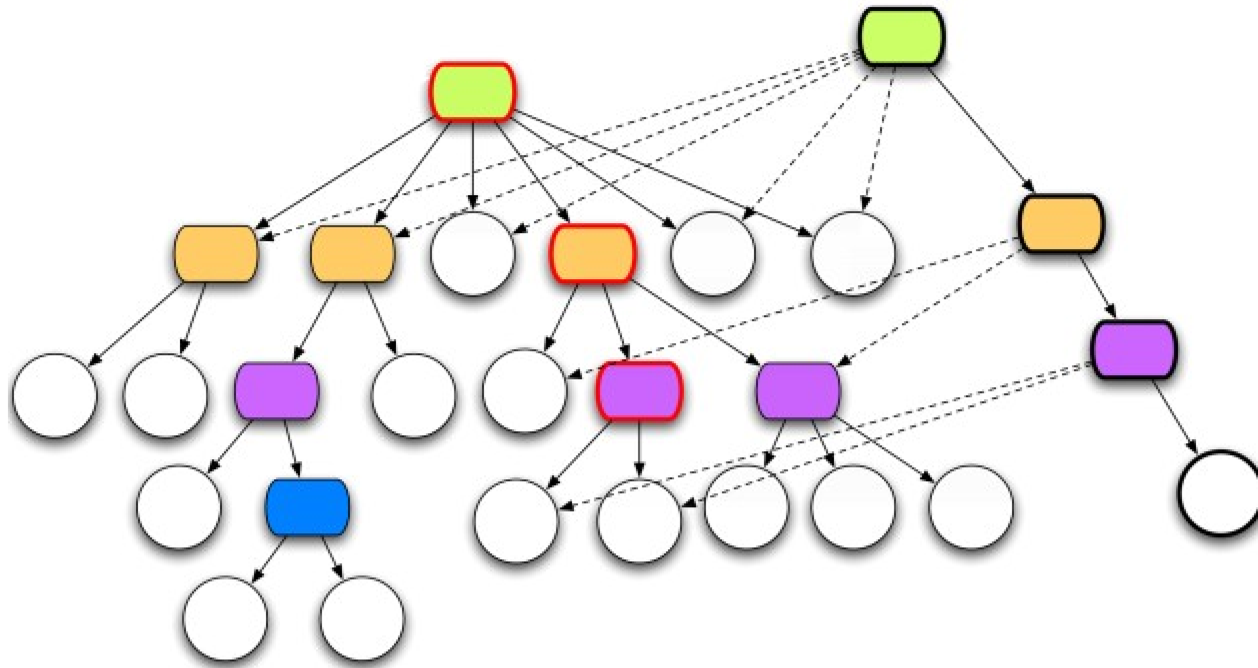


Illustration taken from Rich Hickey's presentation. Copyright Rich Hickey 2009

Our original intention

```
@Transactional
public void findAllFoosAround() {
    //tx agnostic code here
}
```

```
withTransaction {
    //tx agnostic code here
}
```

The reality

```
images.eachParallel {
    //concurrency agnostic code here
}

def myActor = actor {
    //concurrency agnostic code here
}

(dosync /*concurrency agnostic code here*/)
…
```

- Parallelism is not hard, multi-threading is

- Systems that interface to the real world are easier to design in parallel

Tips

- Java
  - JSR-166y - Fork/Join, Parallel Arrays, Multiverse
- Clojure
  - STM, Atoms, Agents, Promises
- Scala
  - Actors, Akka, ScalaAgent
- Groovy
  - GPars – actors, agent, dataflow, fork/join, map/reduce

Questions?