

High-level concurrency concepts

Václav Pech



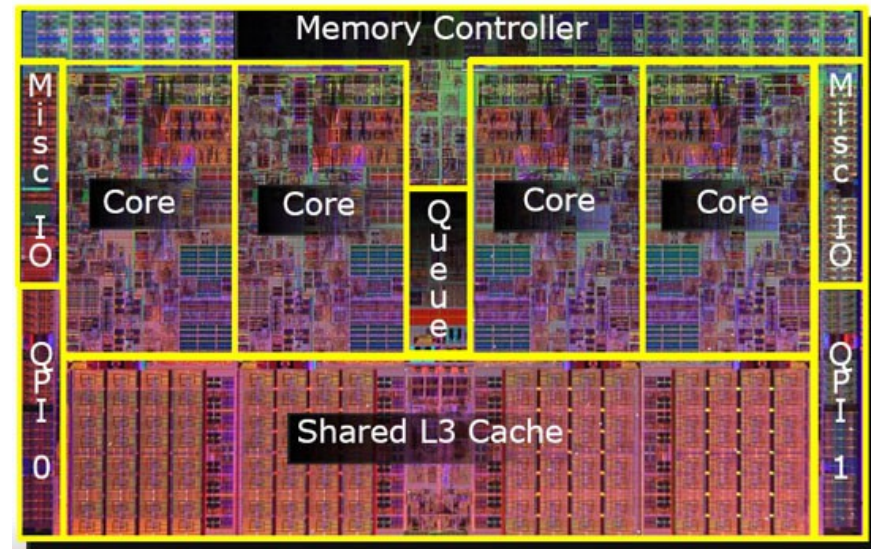
<http://jroller.com/vaclav>

<http://www.vaclavpech.eu>

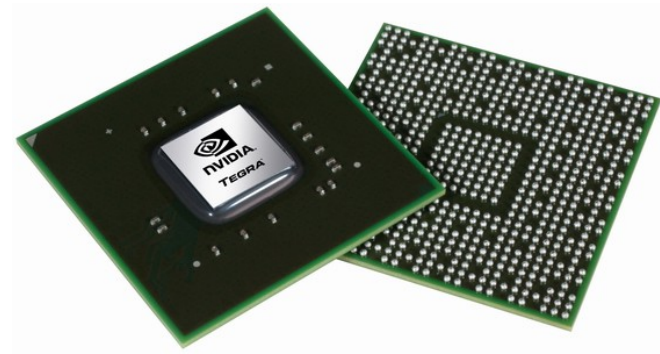
@vaclav_pech

Agenda for today

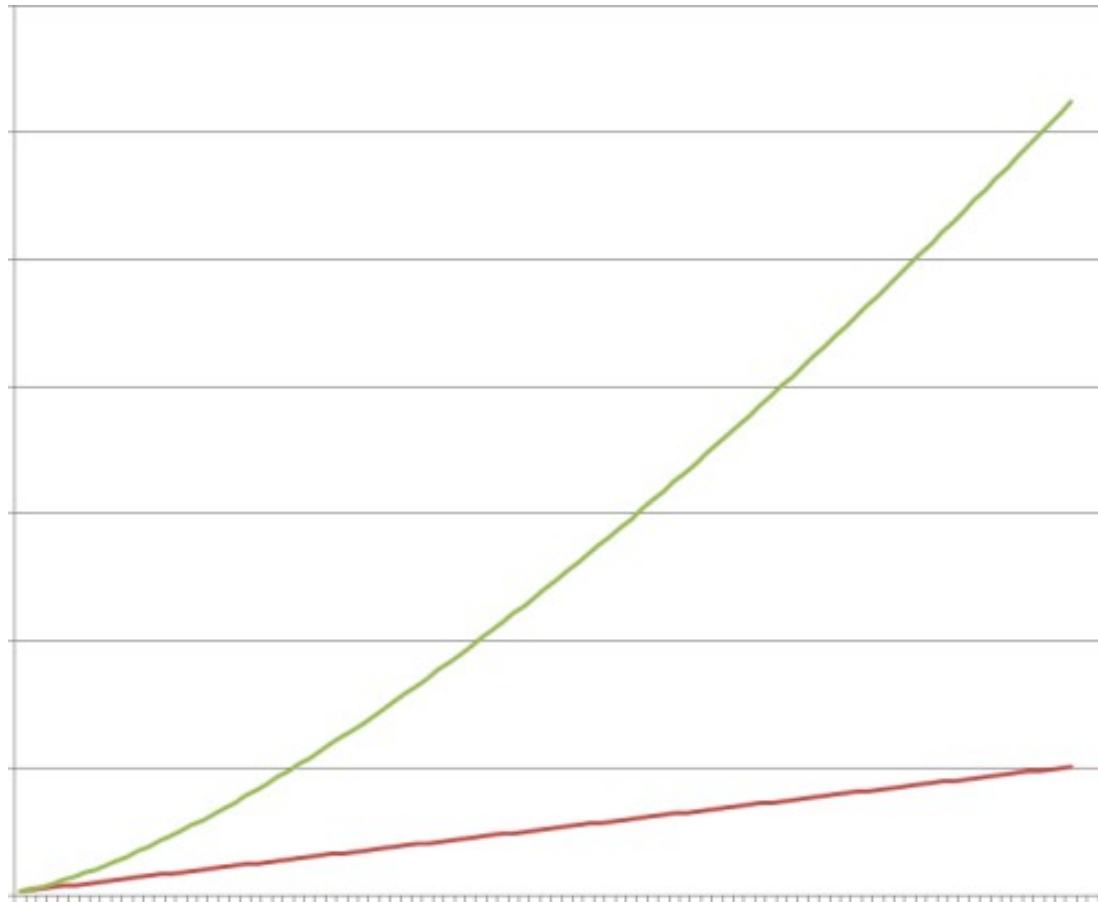
- Threads, thread pools
- Fork/Join
- Parallel collections
- Dataflow
- Message passing
- State management



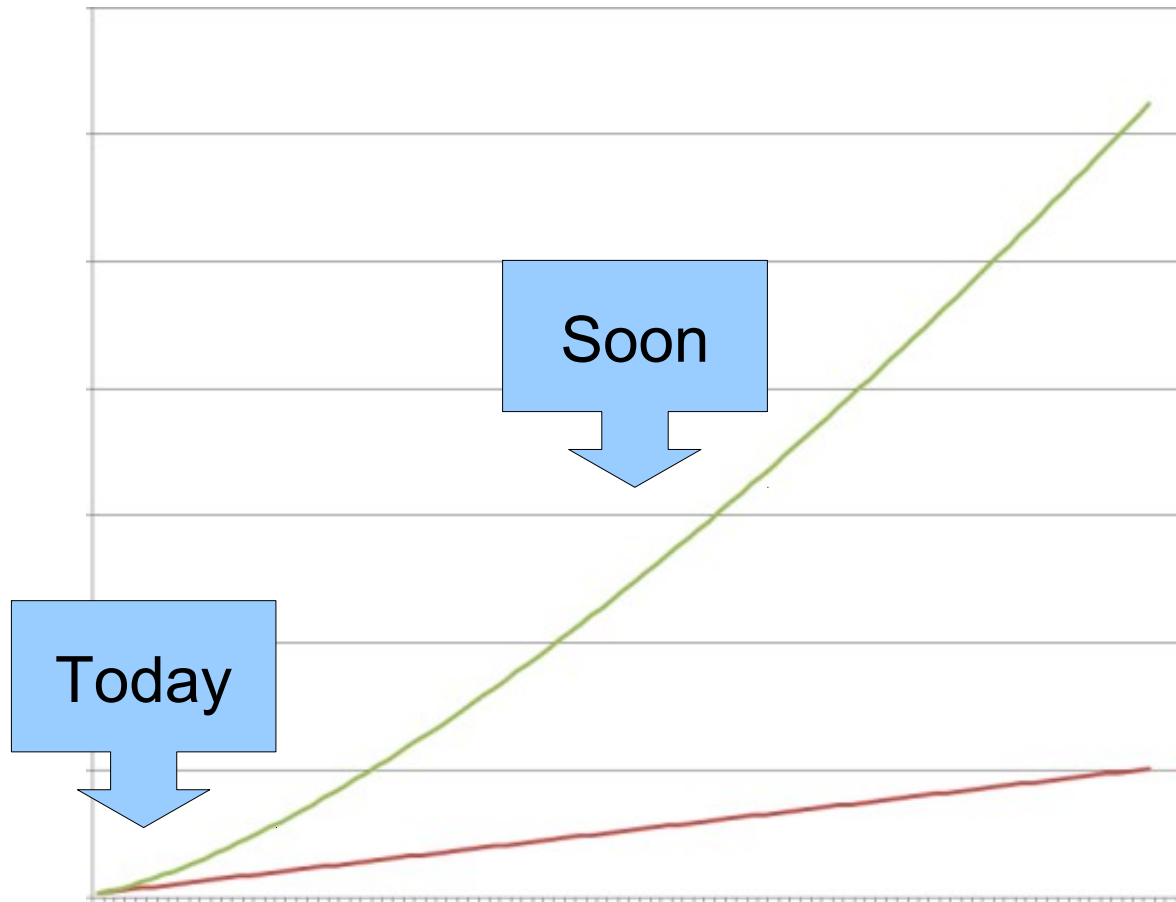
We're all in the parallel computing business!



of cores



of cores



Dealing with threads sucks!

```
public class Counter {  
    private static long count = 0;  
  
    public Counter() {  
  
        count++;  
  
    }  
}
```

Dealing with threads sucks!

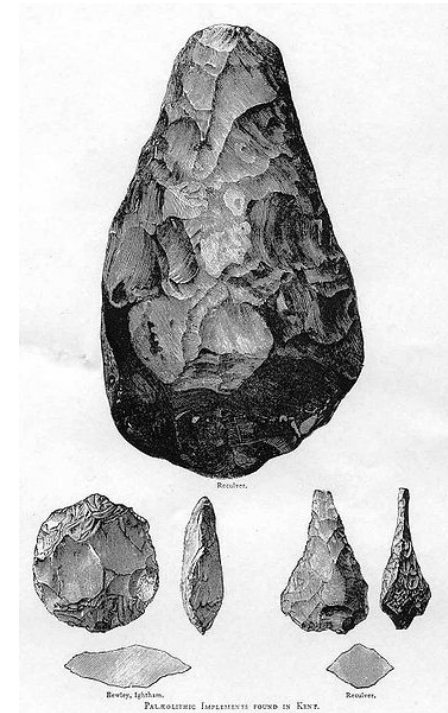
```
public class Counter {  
    private static long count = 0;  
  
    public Counter() {  
        synchronized (this) {  
            count++;  
        }  
    }  
}
```


Dealing with threads sucks!

```
public class Counter {  
    private static long count = 0;  
  
    public Counter() {  
        synchronized (this.getClass()) {  
            count++;  
        }  
    }  
}
```

Dealing with threads sucks!

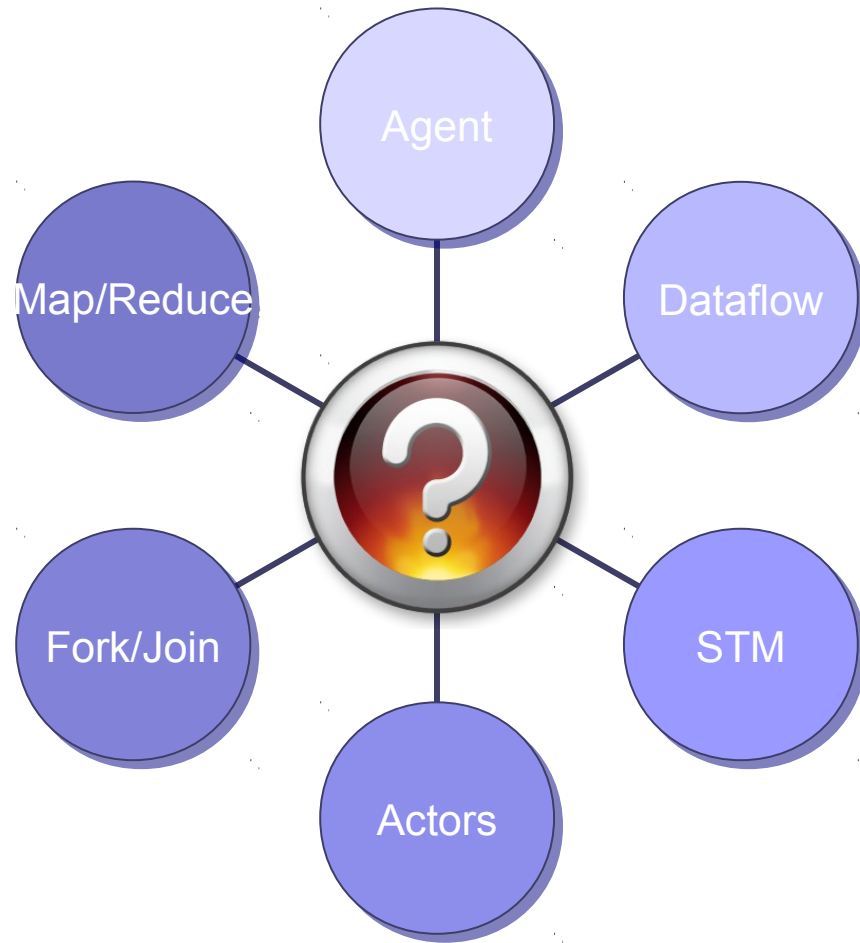
```
public class ClickCounter implements ActionListener {  
    public ClickCounter(JButton button) {  
        button.addActionListener(this);  
    }  
  
    public void actionPerformed(final ActionEvent e) {  
        ...  
    }  
}
```



Multithreaded programs today work mostly by accident!



Can we do better?



Part 1

Thread management



Asynchronous invocation

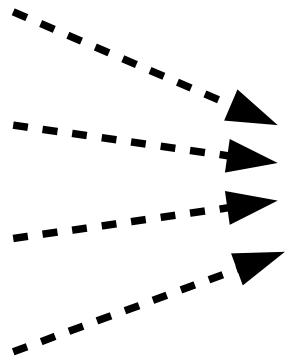
```
Future f = threadPool.submit(calculation);
```

```
...
```

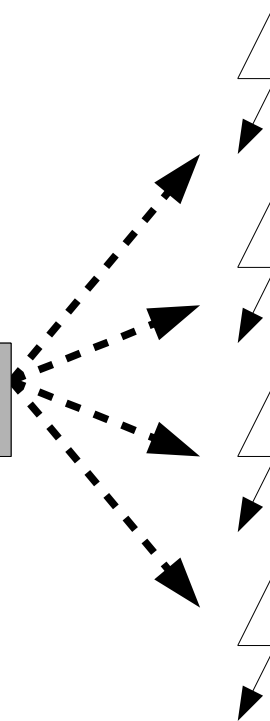
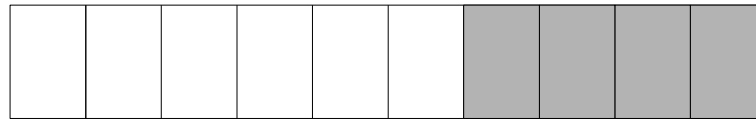
```
System.out.println("Result: " + f.get());
```

Thread Pool

Tasks

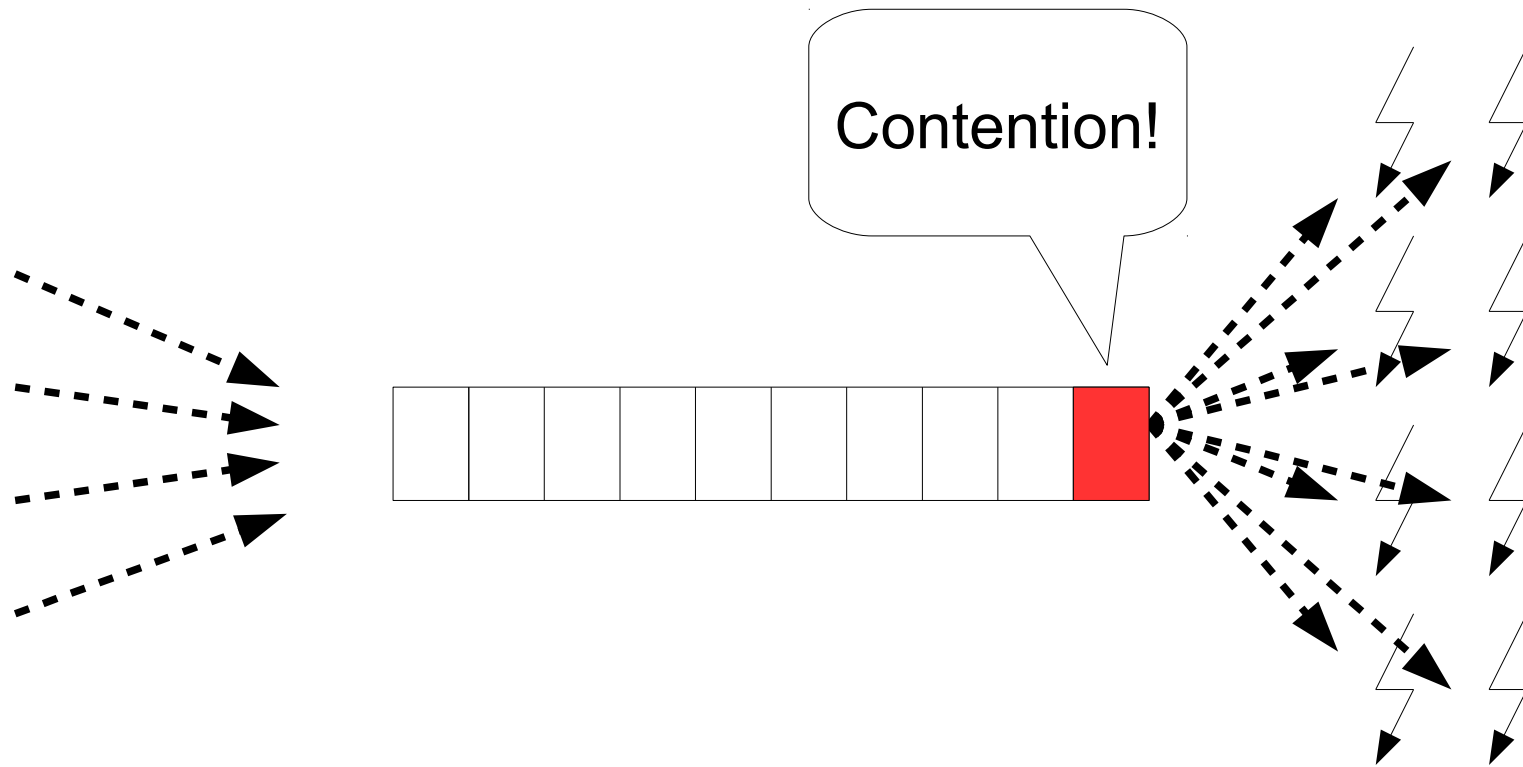


Queue

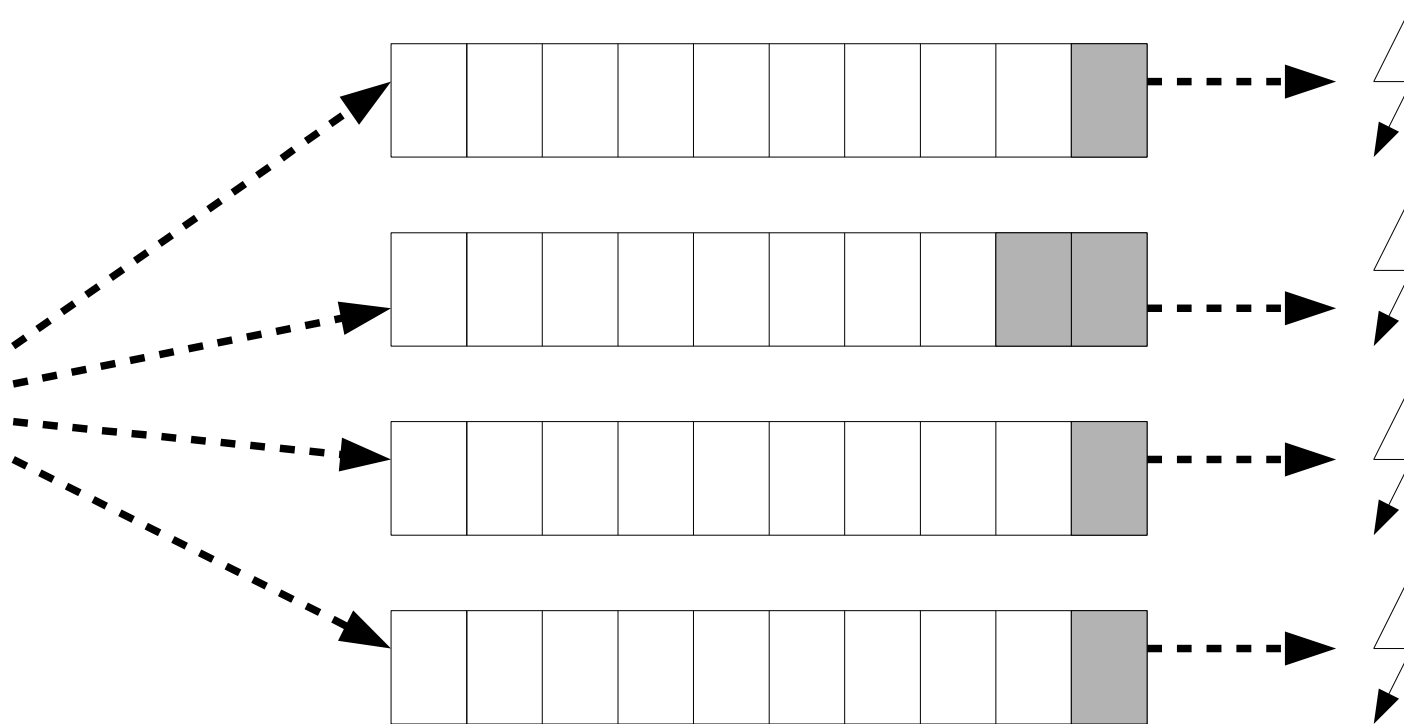


Worker threads

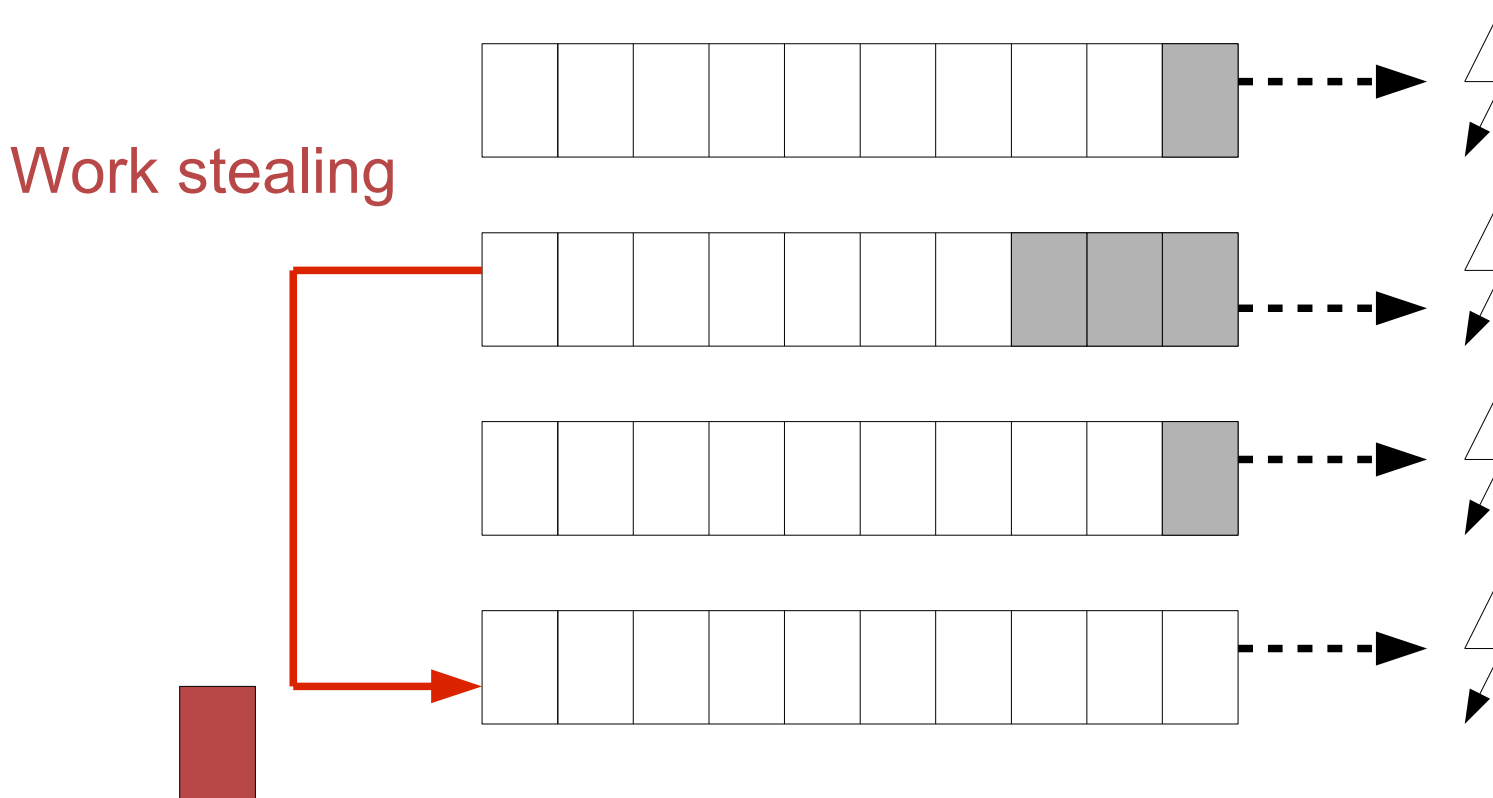
Thread Pool



Fork/Join Thread Pool



Fork/Join Thread Pool

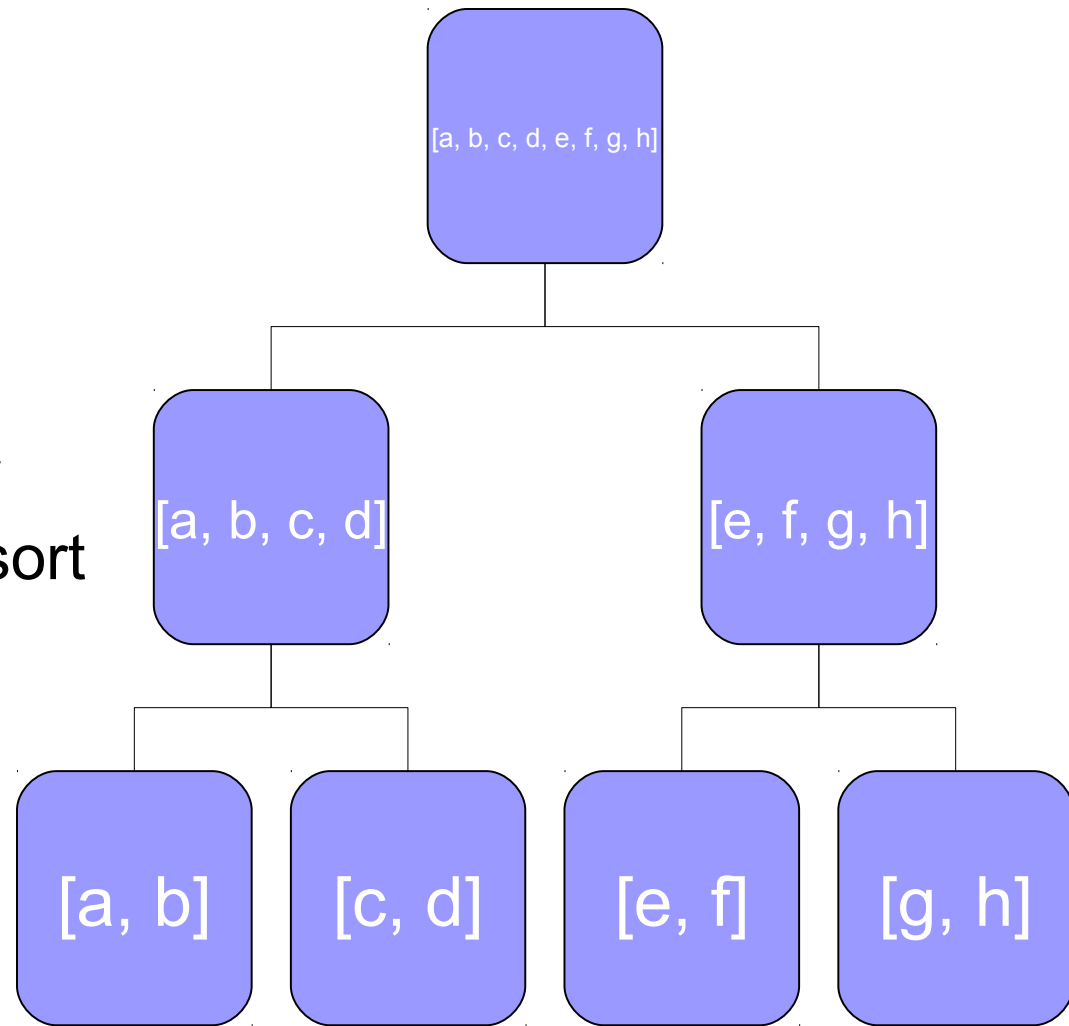


Part 2

Fork/Join

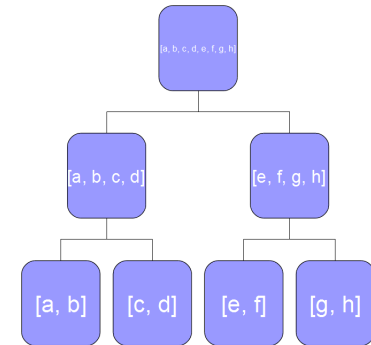
Fork/Join

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

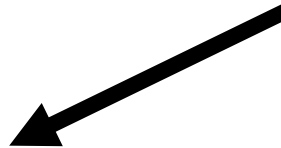


Fork/Join (GPars)

```
runForkJoin(new File("./src")) {currentDir ->
  long count = 0;
  currentDir.eachFile {
    if (it.isDirectory()) {
      forkOffChild it
    } else {
      count++
    }
  }
  return count + childrenResults.sum(0)
}
```



Waits for children
without blocking the
thread!

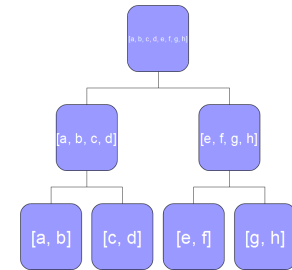


Part 3

Parallel collections

Collections (Groovy/GPars)

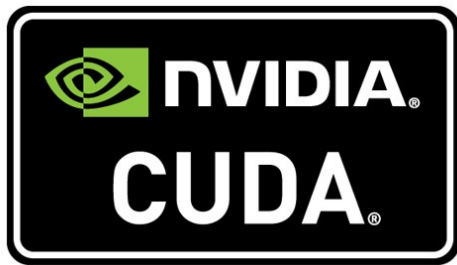
images.`eachParallel` {it.process()}



documents.`sumParallel`()

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

GPU



Java watch list:

<https://github.com/pcpratts/rootbeer1/>

<http://openjdk.java.net/projects/sumatra/>

Part 4

Dataflow



Composing async functions

```
int hash1 = hash(download('http://www.gpars.org'))  
int hash2 = hash(loadFile('/gpars/website/index.html'))  
boolean result = compare(hash1, hash2)  
println result
```

Composing async functions

@AsyncFun *hash = oldHash*

@AsyncFun *compare = oldCompare*

@AsyncFun *download = oldDownload*

@AsyncFun *loadFile = oldLoadFile*

def hash1 = hash(download('http://www.gpars.org'))

def hash2 = hash(loadFile('/gpars/website/index.html'))

def result = compare(hash1, hash2)

println result.get()

Composing async functions

@AsyncFun hash = oldHash

@AsyncFun(blocking = true) *compare = oldCompare*

@AsyncFun download = oldDownload

@AsyncFun loadFile = oldLoadFile

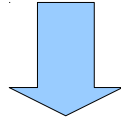
def hash1 = hash(download('http://www.gpars.org'))

def hash2 = hash(loadFile('/gpars/website/index.html'))

boolean result = compare(hash1, hash2)

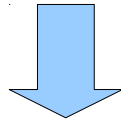
println result

*int hash(String **text**) {...}*

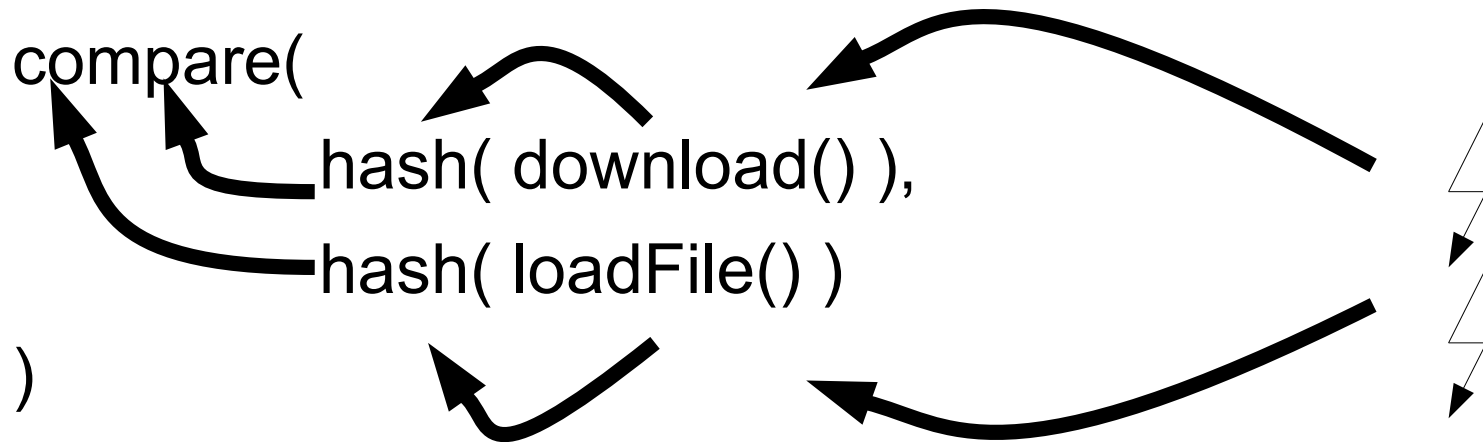


*Promise<int> hash(Promise<String> | String **text**)*

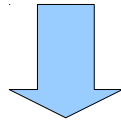
*int hash(String **text**) {...}*



*Promise<int> hash(Promise<String> | String **text**)*



*int hash(String **text**) {...}*



*Promise<int> hash(Promise<String> | String **text**) {*

1. Return a Promise for the **result**
 2. Wait (non-blocking) for the **text** param
 3. Call the original *hash()*
 4. Bind the **result**
- }*



Composing async functions

Combine functions as usual

Parallelism is detected automatically

Composing async functions

@AsyncFun *hash = oldHash*

@AsyncFun *compare = oldCompare*

@AsyncFun *download = oldDownload*

@AsyncFun *loadFile = oldLoadFile*

def hash1 = hash(download('http://www.gpars.org'))

def hash2 = hash(loadFile('/gpars/website/index.html'))

def result = compare(hash1, hash2)

println result.get()

Composing async functions

@AsyncFun *download = oldDownload*

@AsyncFun *loadFile = oldLoadFile*

def h1 = download('http://www.gpars.org') **then** hash

def h2 = loadFile('/gpars/website/index.html') **then** hash

whenAllBound([h1, h2], compare) **then** {println it}

Composing async functions

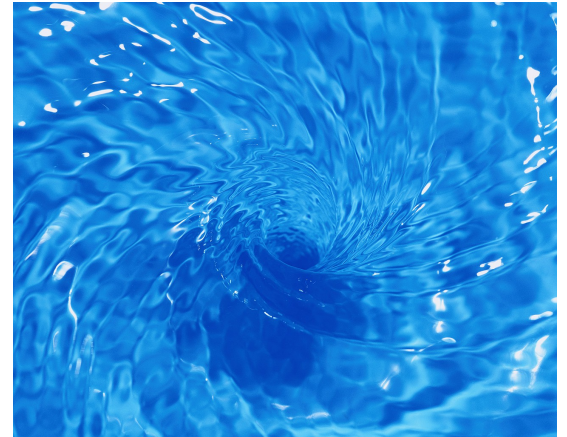
@AsyncFun *download = oldDownload*

@AsyncFun *loadFile = oldLoadFile*

```
whenAllBound ([  
    download('http://www.gpars.org') >> hash,  
    loadFile('/gpars/website/index.html') >> hash  
, compare) >> {println it}
```

Dataflow Concurrency

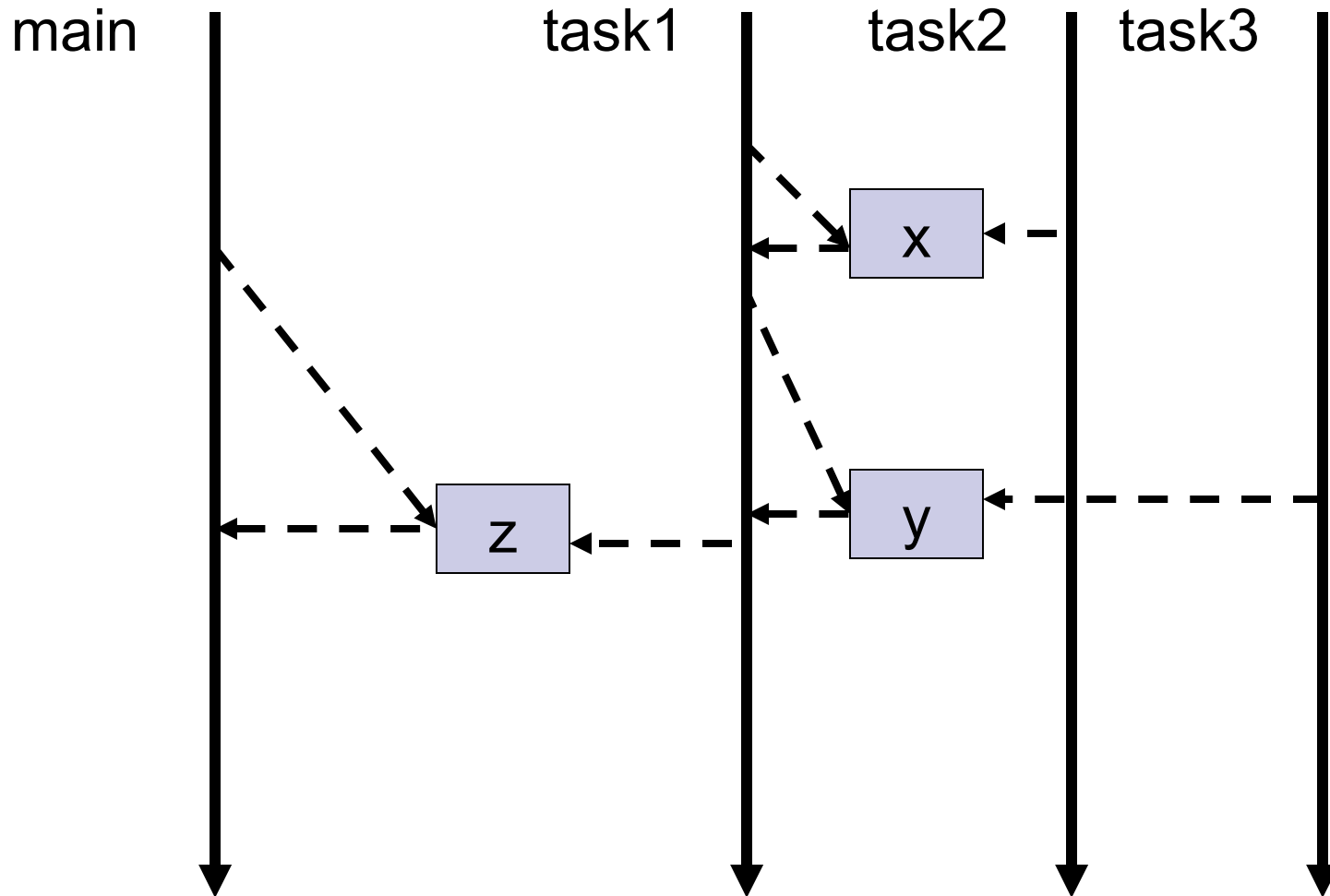
- No race-conditions
 - No live-locks
 - Deterministic deadlocks
- Completely deterministic programs



BEAUTIFUL code

(Jonas Bonér)

Dataflow Variables / Promises



Dataflows

```
def df = new Dataflows()
```

```
task { df.z = df.x + df.y }
```

```
task { df.x = 10 }
```

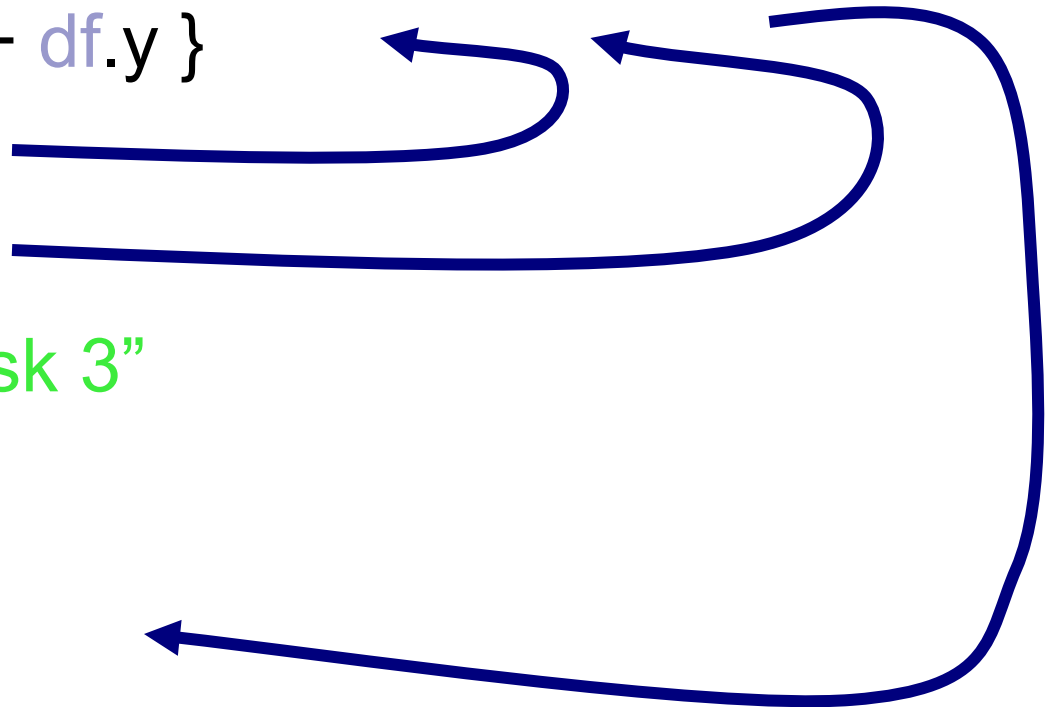
```
task {
```

```
    println "I am task 3"
```

```
    df.y = 5
```

```
}
```

```
assert 15 == df.z
```



Dataflow Operators

```
operator(inputs: [headers, bodies, footers],  
         outputs: [articles, summaries])
```

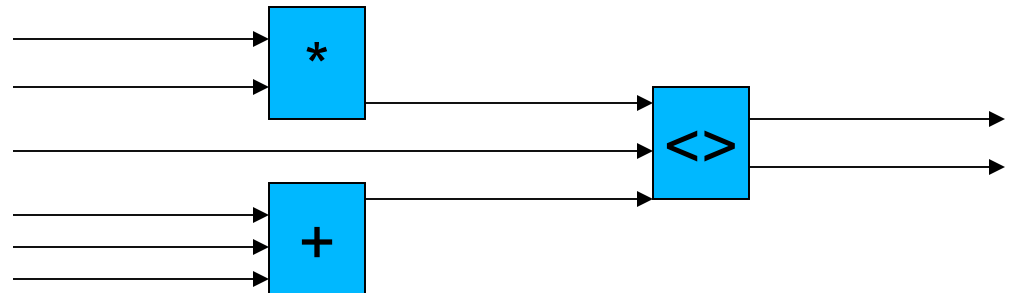
```
{header, body, footer ->
```

```
  def article = buildArticle(header, body, footer)
```

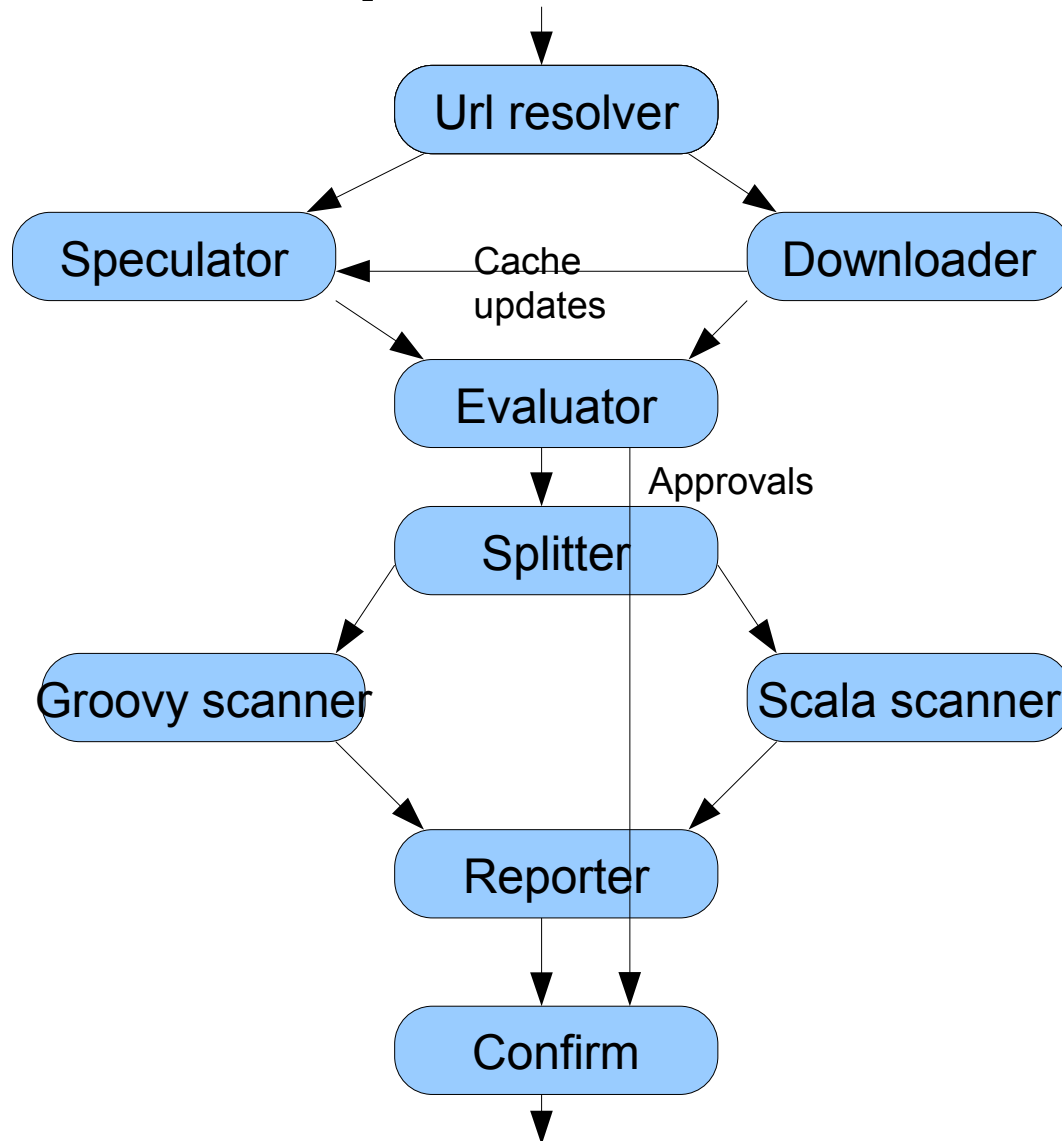
```
  bindOutput(0, article)
```

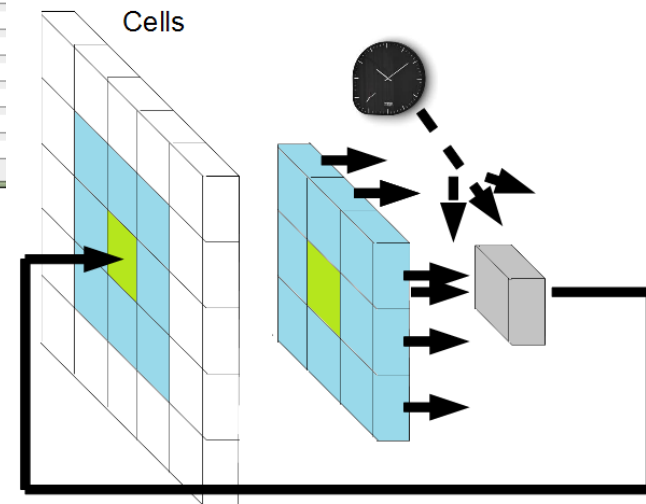
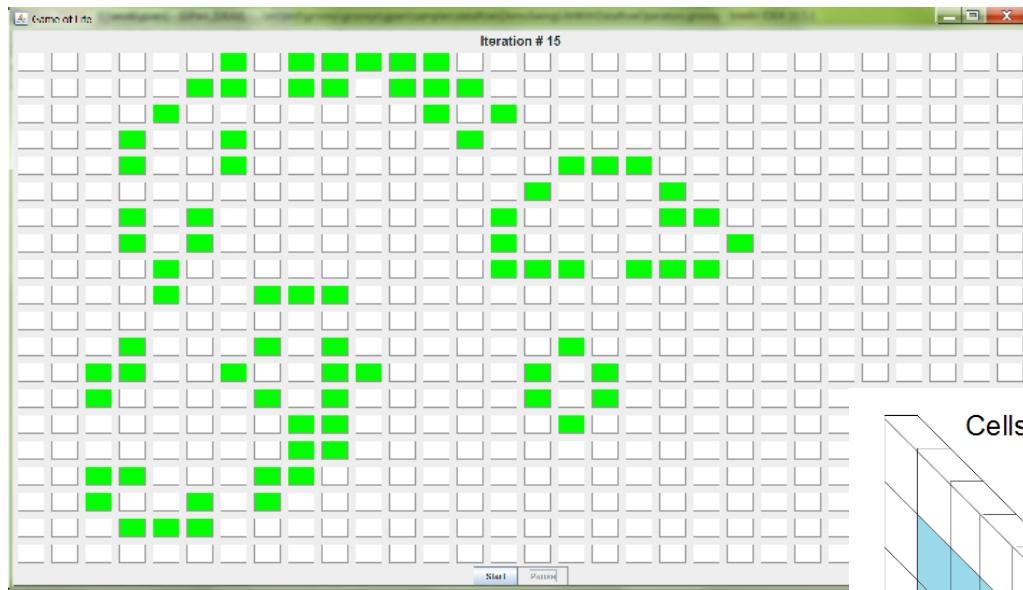
```
  bindOutput(1, buildSummary(article))
```

```
}
```



Dataflow Operators







Milestone

- Asynchronous background tasks
- Fork/Join
- Parallel collection processing
- Dataflow variables/streams

Part 5

Message passing



Message Passing

Communicating Sequential Processes (CSP)

Dataflow Operators

Actors



Actors

Processes with a mail-box

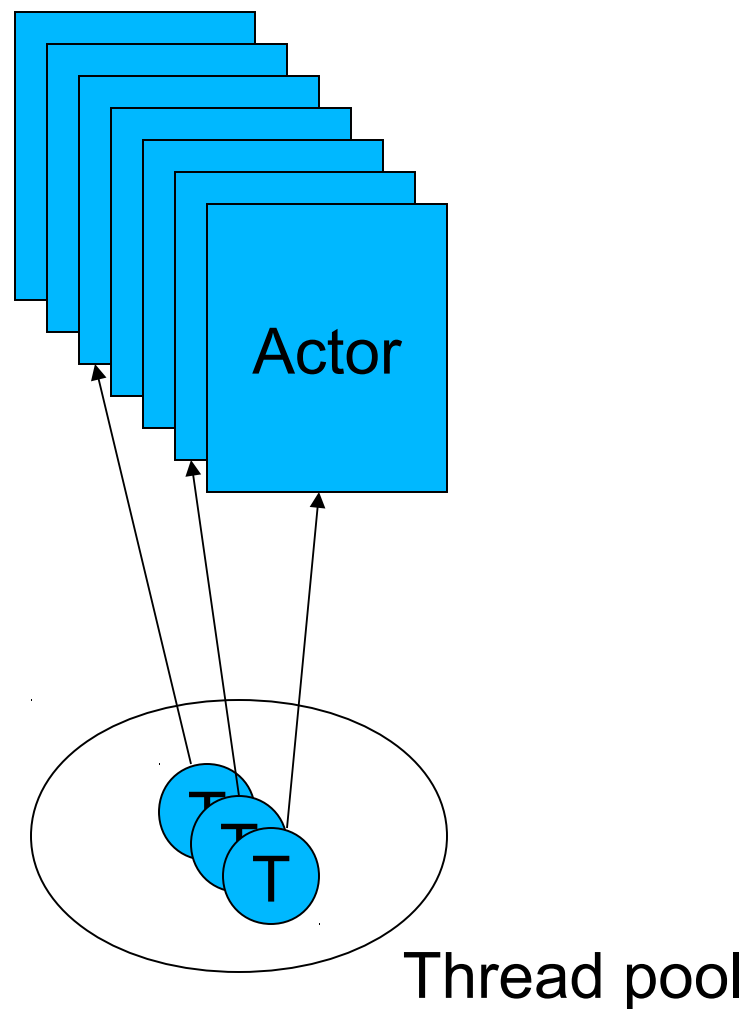
Share no data

Communicate by sending messages

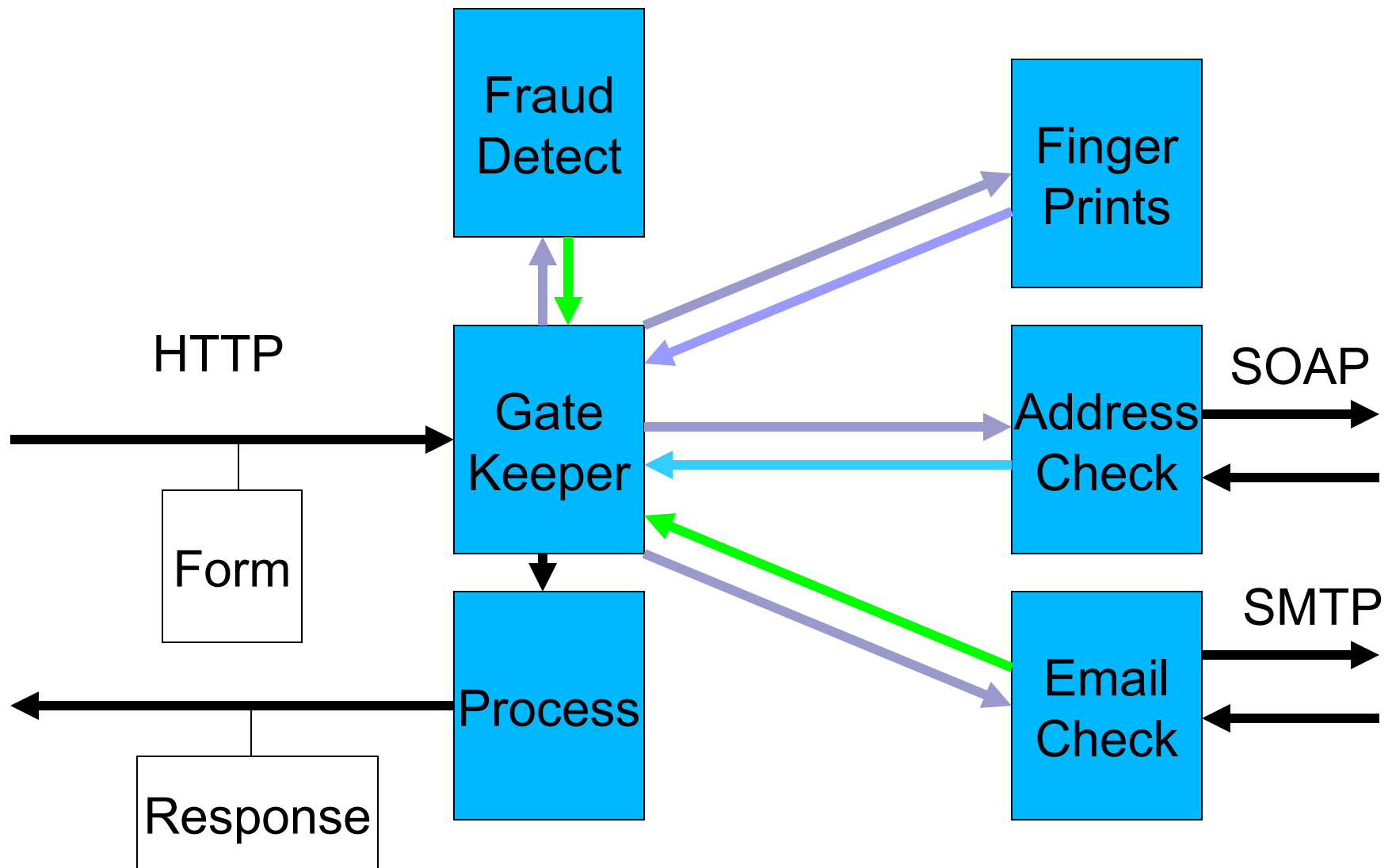
Use a thread-pool

Actors

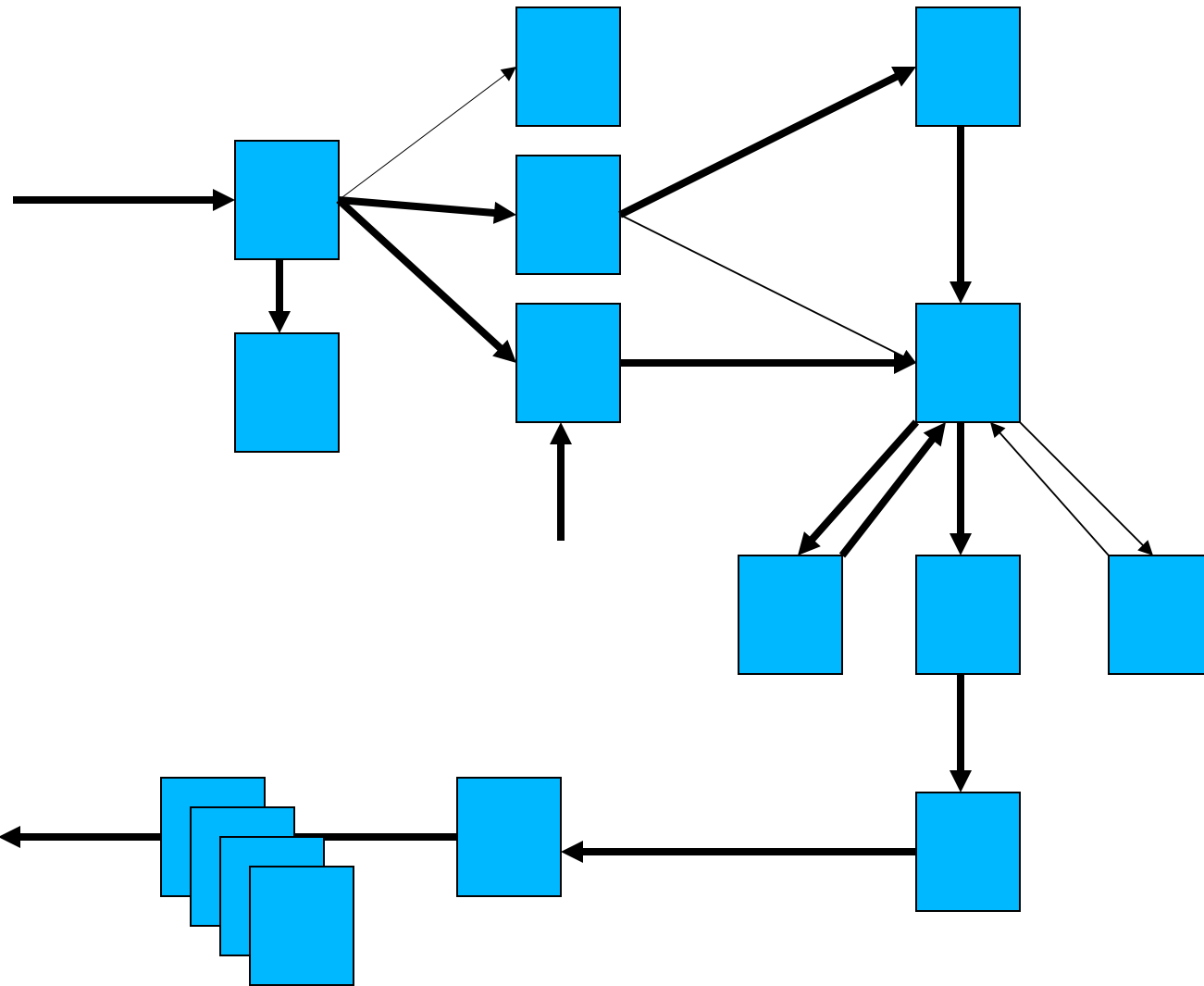
- Isolated
- Communicating
 - Immutable messages
- Active
 - Pooled shared threads
- Activities
 - Create a new actor
 - Send a message
 - Receive a message



Actors use



Actors patterns



Enricher

Router

Translator

Endpoint

Splitter

Aggregator

Filter

Resequencer

Checker

Sending messages

```
buddy.send 10.eur
```

```
buddy << new Book(title:'Groovy Recipes',  
                    author:'Scott Davis')
```

```
def canChat = buddy.sendAndWait 'Got time?'
```

```
buddy.sendAndContinue 'Need money!', {cash->  
    pocket.add cash  
}
```

Stateless Actors (pure Java)

```
class MyActor extends DynamicDispatchActor {  
    Account account = ...  
    public void onMessage(String msg) {  
        String encrypted = encrypt(msg);  
        reply(encrypted);  
    }  
    public void onMessage(Integer number) {  
        reply(2 * number);  
    }  
  
    public void onMessage(Money cash) {  
        System.out.println("Received a donation " + cash);  
        account.deposit(cash);  
    }  
}
```

Stateful Actors

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

Implicit State in Actors

```
val me = actor {  
  react {msg1 ->  
    switch (msg1) {  
      case Work: reply "I don't work so early" ; stop();  
      case Breakfast:  
        msg1.eat()  
        react {msg2 ->  
          switch (msg2) {  
            case Work: reply "OK, time to work"; msg2.do()  
            case Lunch: ...  
          }  
        }  
      }  
}
```

Continuation Style

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

Active Objects

@ActiveObject

```
class MyCounter {  
    private int counter = 0
```

@ActiveMethod

```
    def incrementBy(int value) {  
        println "Received an integer: $value"  
        this.counter += value  
    }  
}
```

Part 6

State management



Shared Mutable State

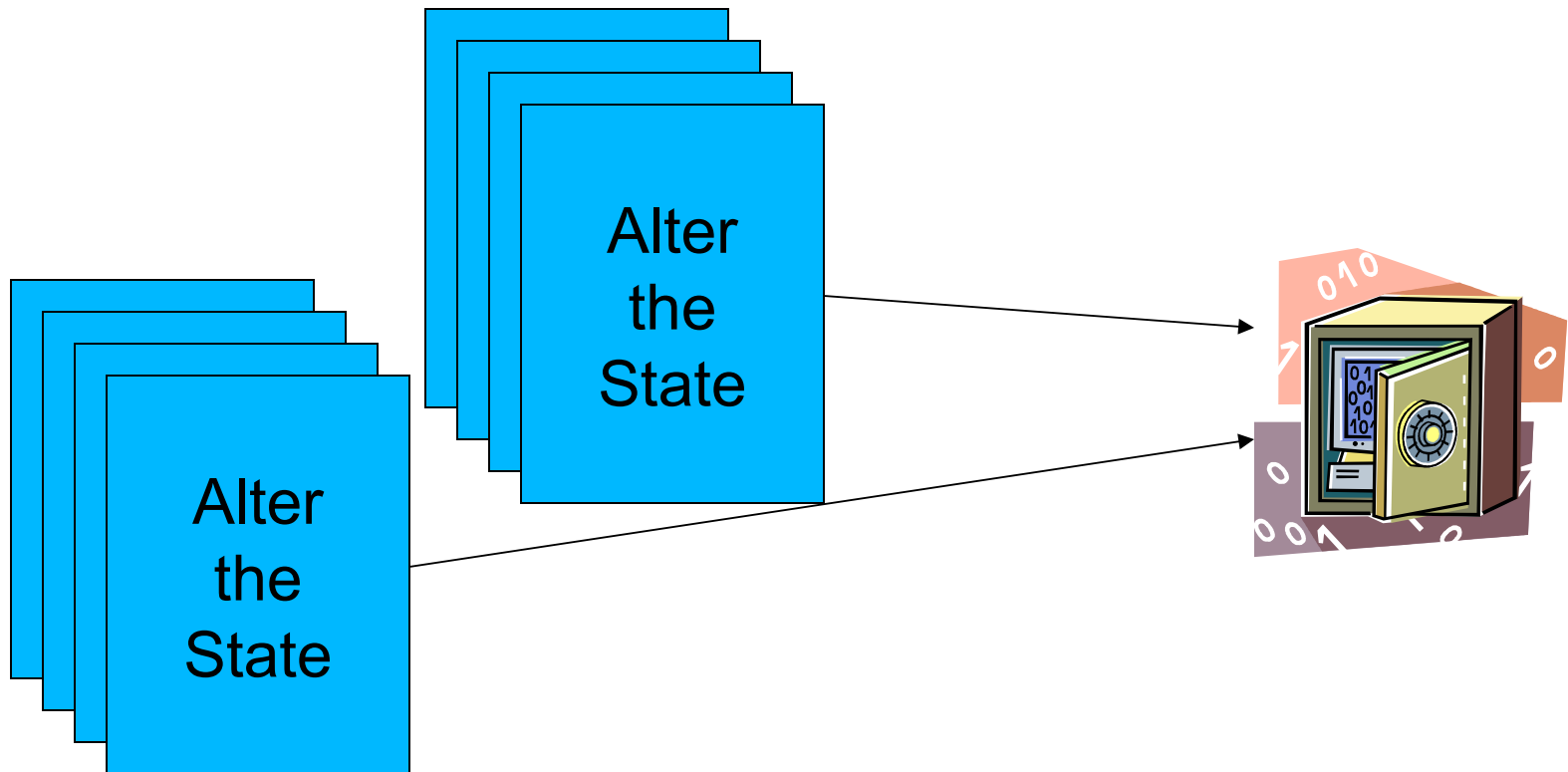
Misused most of the time

When really needed, use

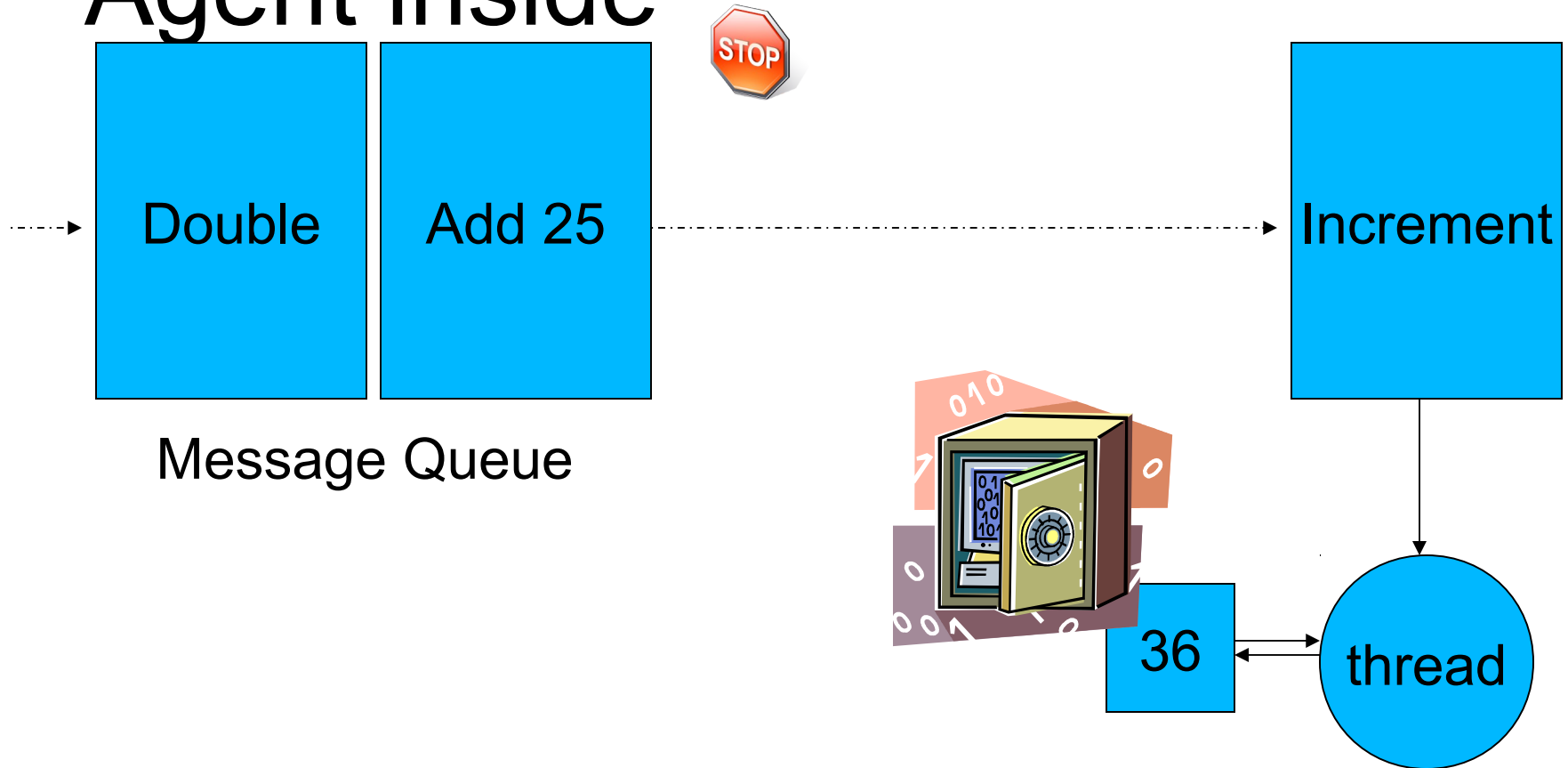
- Agents
- Software Transactional Memory
- Locks

Agent

- Lock **Shared Mutable State** in a **Safe**



Agent inside



Agent List

```
def jugMembers = new Agent(['Me']) //add Me
```

```
task {  
  jugMembers.send {it.add 'Joe'} //add Joe  
}
```

```
task {  
  jugMembers << {it.add 'Dave'} //add Dave  
  jugMembers << {it.add 'Alice'} //add Alice  
}
```

```
...
```

```
println jugMembers.val
```

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  
}
```

Persistent Data Structures

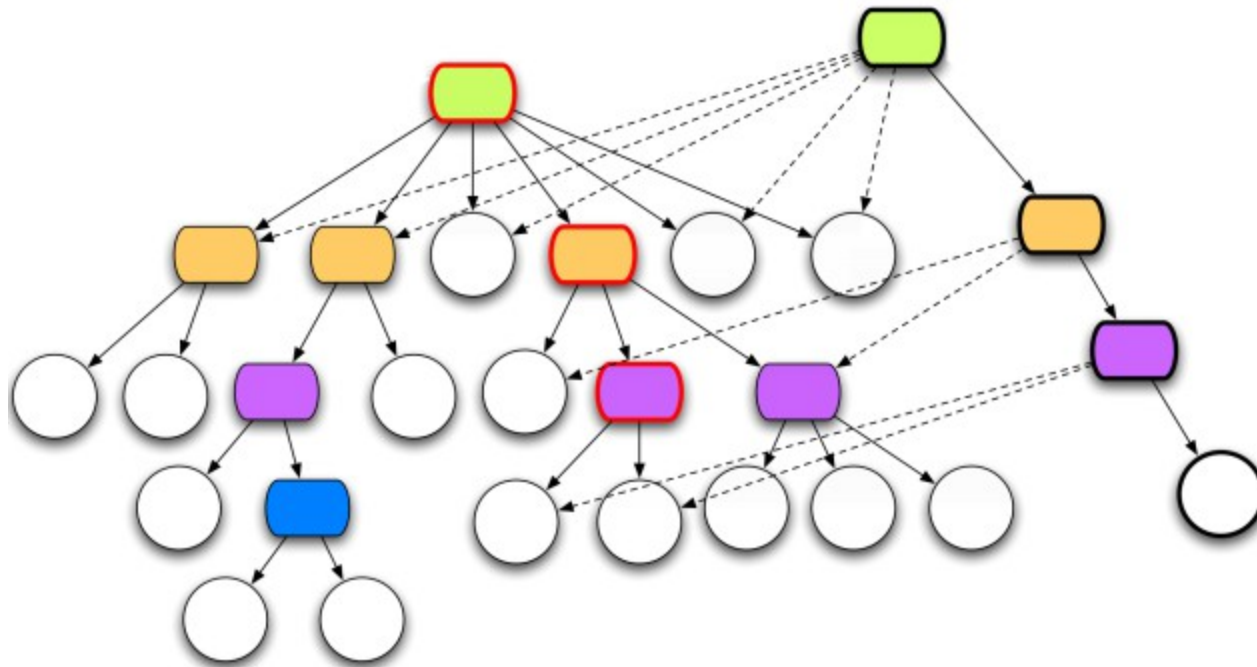


Illustration taken from Rich Hickey's presentation. Copyright Rich Hickey 2009

No more threads and locks

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

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

```
atomic { /*concurrency agnostic code here*/ }  
...
```


Summary

Parallelism is not hard, multi-threading is

Jon Kerridge, Napier University



References

<http://www.groovy.cz>

<http://gpars.codehaus.org>

<http://gparsconsole.appspot.com/>

<http://groovy.codehaus.org>

<http://www.manning.com/koenig2/>