

# High-level concurrency concepts

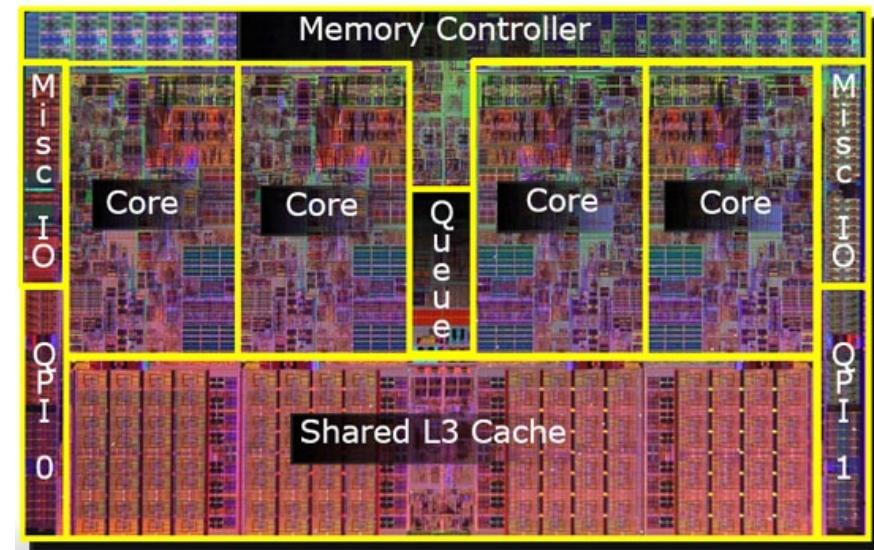


**Václav Pech**  
*NPRG014 2025/2026*



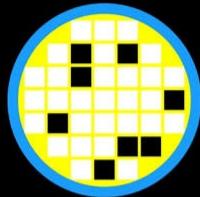
<http://www.vaclavpech.eu>  
@vaclav\_pech

# Why concurrency?



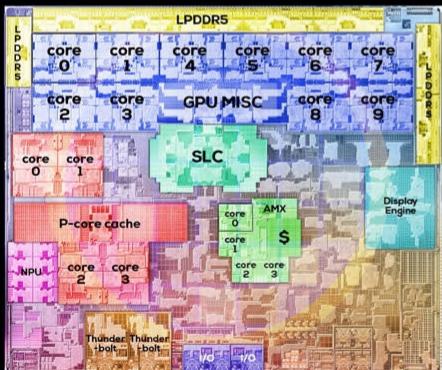
We're all in the parallel computing business!

# Apple M3 family TSMC N3B (3nm)



High  
Yield

25bn transistors

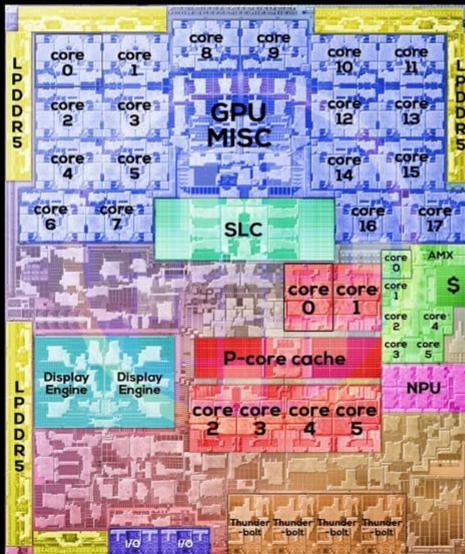


**M3**



CPU 4 P-cores  
CPU 4 E-cores  
GPU 10 cores  
128-bit LPDDR5

37bn transistors

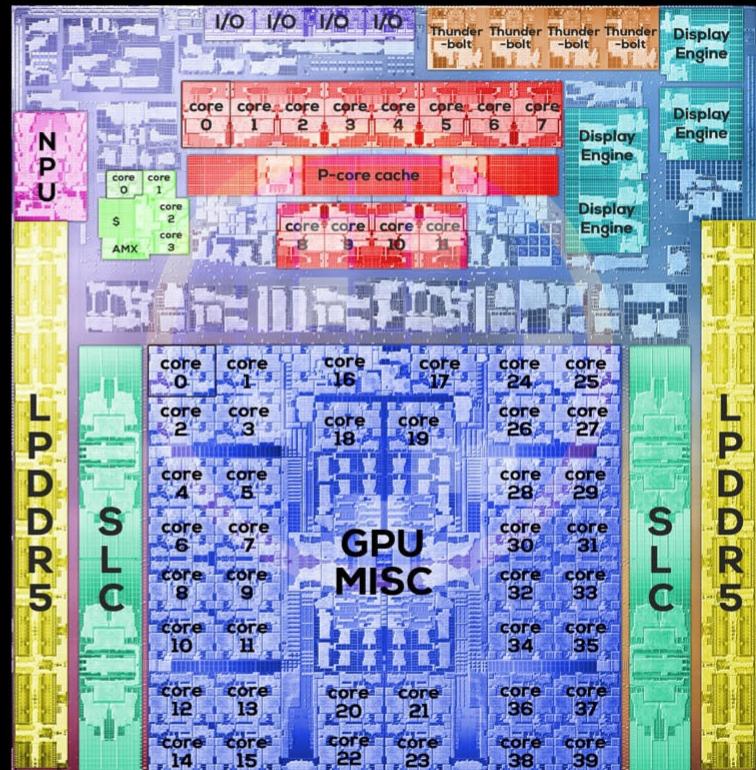


**M3 Pro**



CPU 6 P-cores  
CPU 6 E-cores  
GPU 18 cores  
192-bit LPDDR5

92bn transistors



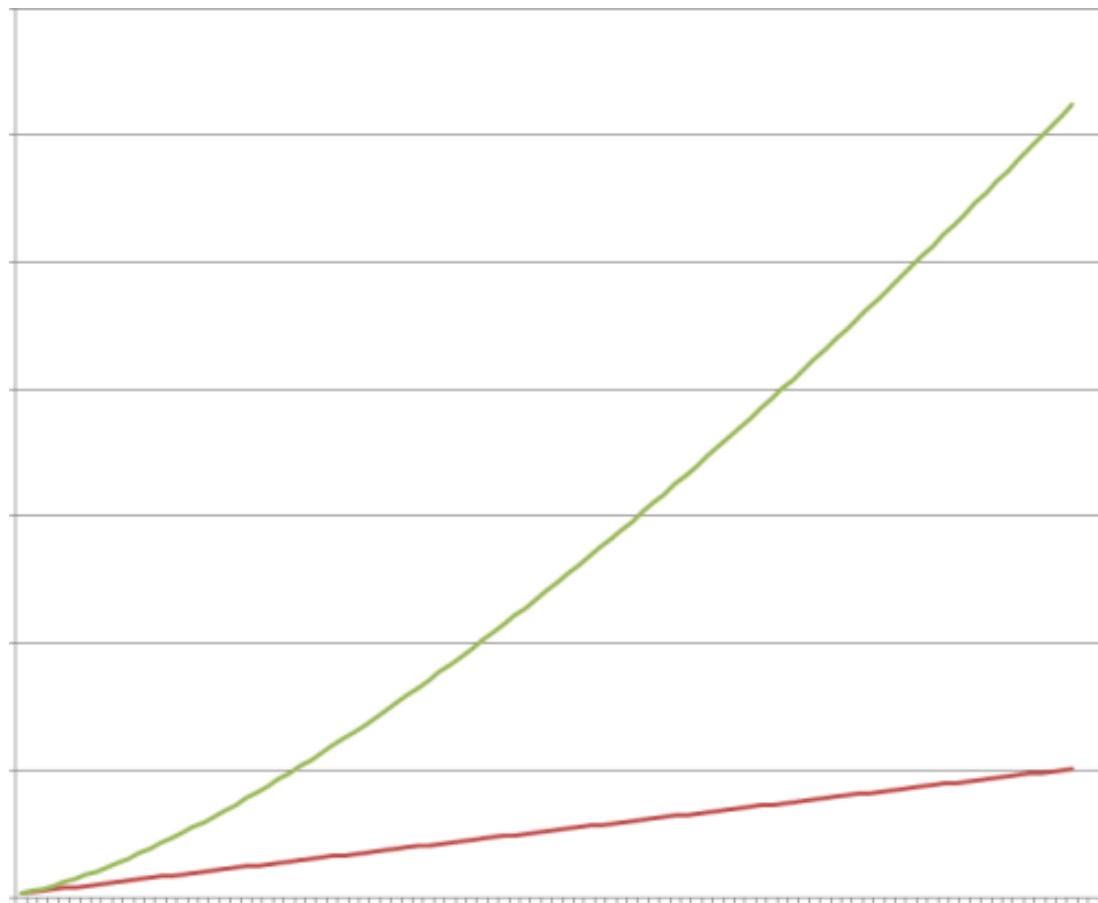
**M3 Max**



CPU 12 P-cores  
CPU 4 E-cores  
GPU 40 cores  
512-bit LPDDR5

NPU  
SLC

# # of cores



# JVM low-level machinery

Thread, Runnable, Thread Pools, Virtual Threads

- *to run things in parallel*

# JVM low-level machinery

Thread, Runnable, Thread Pools, Virtual Threads

- *to run things in parallel*

Synchronized blocks

Volatile

Locks

Atomic

- *to run things sequentially again*

# Threads

```
def myRunnable = new Runnable {  
    public void run() {  
        ...  
    }  
}  
  
new Thread(myRunnable).start()
```

# Threads

```
def myRunnable = new Runnable {  
    public void run() {  
        ...  
    }  
}  
  
new Thread(myRunnable).start()
```

# Thread Pool

```
def pool = Executors.newFixedThreadPool(6)
def myCallable = new Callable<T> {
    public T call() {...}
}
Future<T> f = pool.submit(myCallable)
println f.get()
```

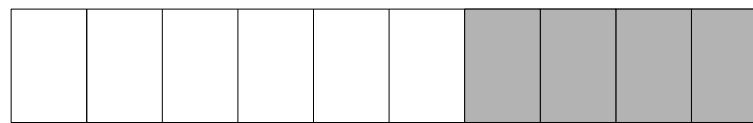
# Thread Pool

```
def pool = Executors.newFixedThreadPool(6)
def myCallable = new Callable<T> {
    public T call() {...}
}
Future<T> f = pool.submit(myCallable)
println f.get()
```

# Thread Pool

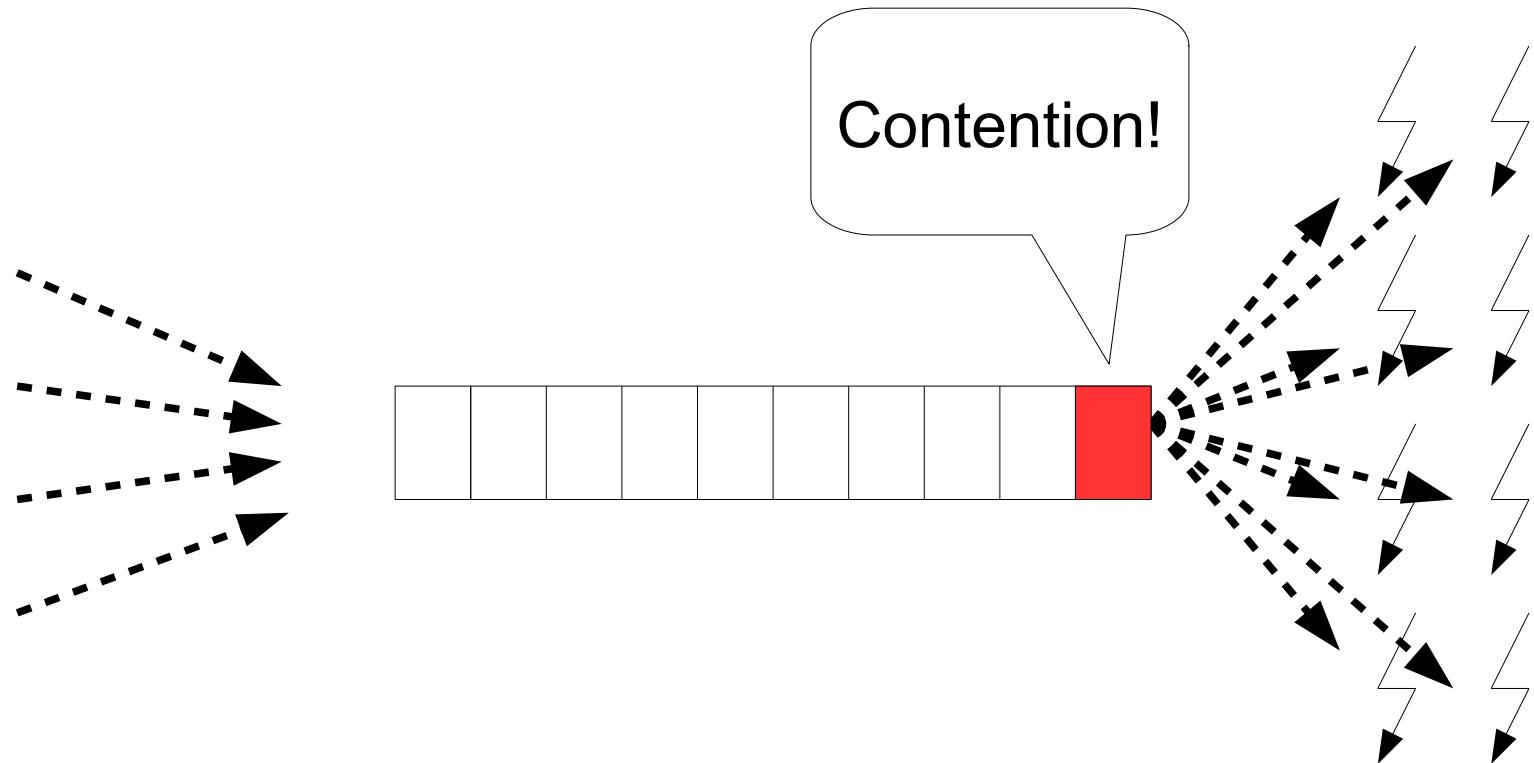
Tasks

Queue

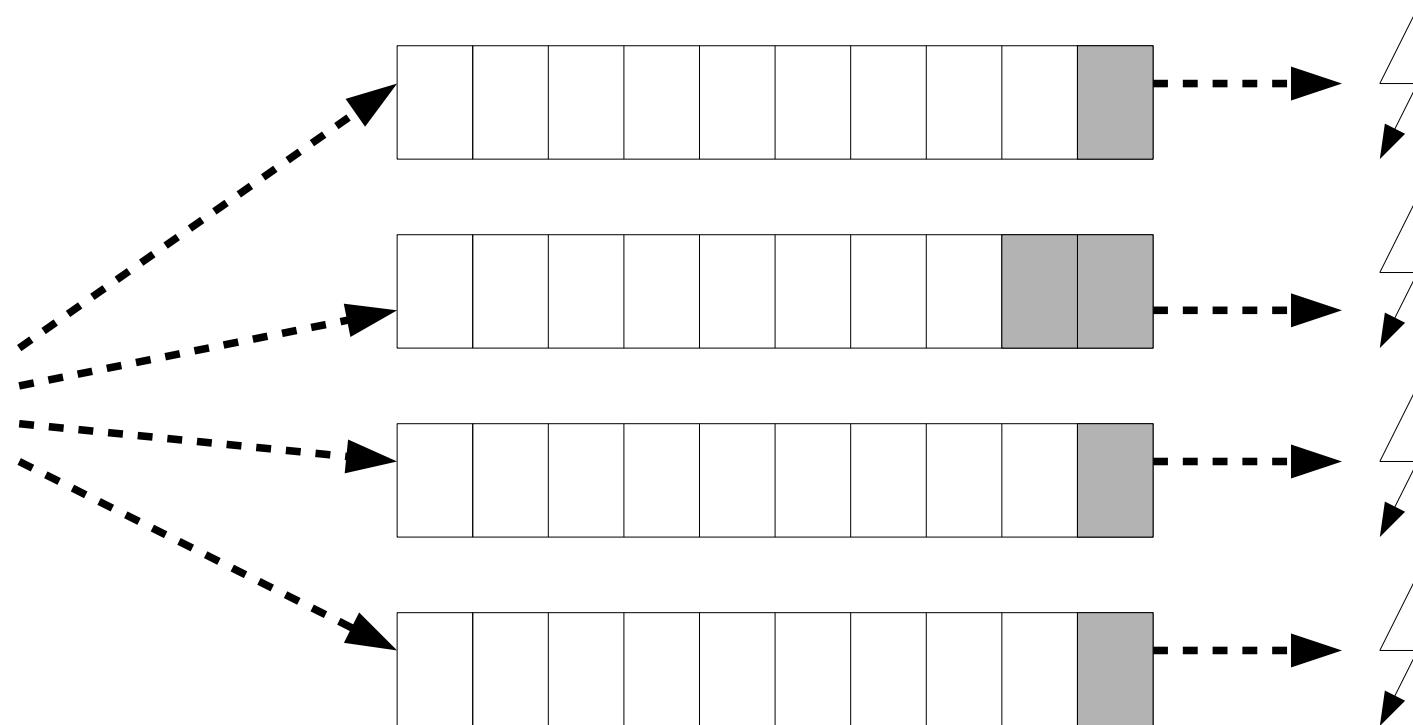


Worker threads

# Thread Pool

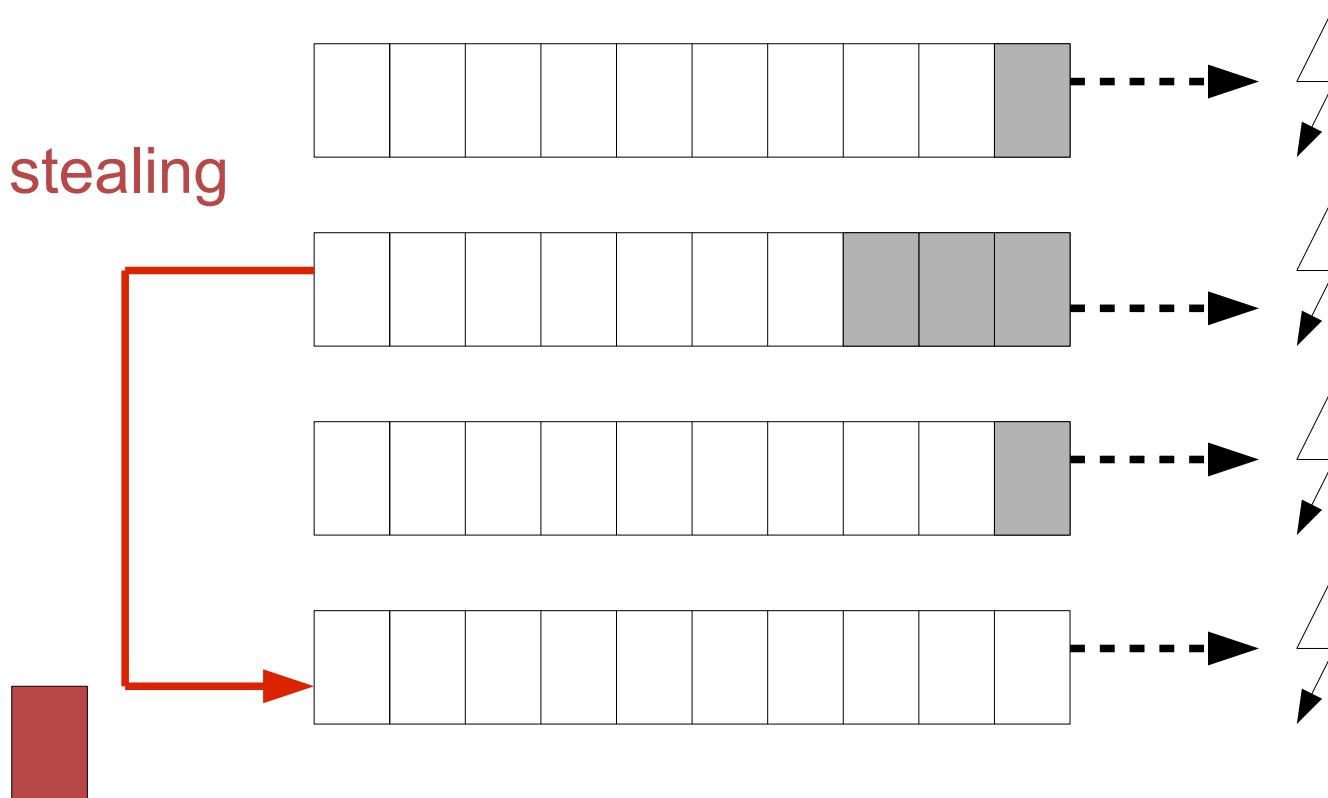


# Fork/Join Thread Pool



# Fork/Join Thread Pool

Work stealing



# Dealing with threads sucks!

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

# Dealing with threads sucks!

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

# Dealing with threads sucks!

```
public class Counter {  
    private volatile static long count = 0;  
  
    public Counter() {  
        count = count + 1;  
    }  
}
```

# 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 Counter {  
    private static Long count = 0;  
  
    public Counter() {  
        synchronized (count) {  
            count++;  
        }  
    }  
}
```

# Dealing with threads sucks!

```
public class Counter {  
    private static Long count = 0;  
  
    public Counter() {  
        synchronized (count) {  
            count = new Long(count.longValue() + 1);  
        }  
    }  
}
```

# Dealing with threads sucks!

```
public class ClickCounter implements ActionListener {  
    private int numberOfClicks;  
  
    public ClickCounter(JButton button, int initialValue) {  
        button.addActionListener(this);  
        numberOfClicks = initialValue;  
    }  
  
    public void actionPerformed(final ActionEvent e) {  
        numberOfClicks++;  
    }  
}
```

# Dealing with threads sucks!

```
public class ClickCounter implements ActionListener {  
    private int numberOfClicks;  
  
    public ClickCounter(JButton button, int initialValue) {  
        button.addActionListener(this);  
        numberOfClicks = initialValue;   
    }  
  
    public void actionPerformed(final ActionEvent e) {  
        numberOfClicks++;   
    }  
}
```

# Low-level concurrency

- Monitors
- Volatile variables
- Atomic variables
- Locks – R/W, ReentrantLock
- Semaphors
- CountDownLatches
- Barriers

# Stone age of parallel SW

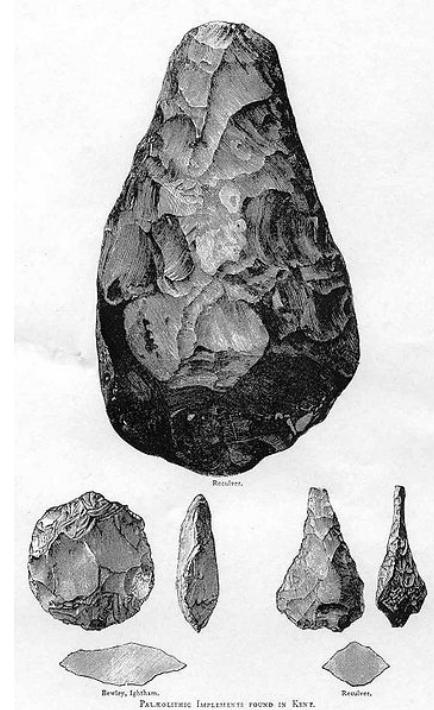
Dead-locks

Live-locks

Race conditions

Starvation

Shared Mutable State



PALÆOLITHIC IMPLEMENTS FOUND IN KENT.

# The root problem

Shared mutable state

# Approach 1

Shared **mutable** state

*Use read-only data*

# Approach 2

**Shared mutable state**

*Use only local thread-contained data*

# Structured concurrency

- Keep state **local** to a computation
  - Virtual threads
  - Coroutines
- CSP provides the cornerstone theory

# CSP

## Communicating Sequential Processes

Focus on composable processes more than on data

# Two key components

## Tasks (aka coroutines, goroutines)

- Implemented as (pooled) threads or virtual, lightweight threads

## Data exchange between tasks

- Tasks' *promised* return value
- Dataflow variables
- Dataflow channels

# Two key components

## Tasks (aka coroutines, goroutines)

- Implemented as (pooled) threads or virtual, lightweight threads

## Data exchange between tasks

- Tasks' *promised* return value
- Dataflow variables
- Dataflow channels

# Virtual Threads (Java 21)

```
def t = Thread.ofVirtual.start(myRunnable)  
println f.join()
```

# Virtual Threads (Java 21)

```
def t = Thread.ofVirtual.start(myRunnable)  
println f.join()
```

```
def pool = Executors  
.newVirtualThreadPerTaskExecutor()  
Future<T> f = pool.submit(myCallable)  
println f.get()
```

# Scopes (Java 21)

```
Response handle() throws ExecutionException, InterruptedException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        Supplier<String> user = scope.fork(() -> findUser());
        Supplier<Integer> order = scope.fork(() -> fetchOrder());

        scope.join()          // Join both subtasks
            .throwIfFailed(); // ... and propagate errors

        // Here, both subtasks have succeeded, so compose their results
        return new Response(user.get(), order.get());
    }
    //...
}
```

# Scope

Provides virtual threads with:

- a thread pool
- a context
- *life-cycle management*
- *control for sub-groups*
- *error handling for tasks and sub-groups*

# Thread pools the Groovy way

```
task {  
    calculation.process()  
}
```



# Scopes the Groovy way

```
def group = new NonDaemonPGroup(10)
```

```
group.task {  
    calculation.process()  
}
```



# Task variants

```
group.task( {->...} )
```

```
group.task( new Runnable() {...} )
```

```
group.task( new Callable<V>() {...} )
```



# Not implemented yet

group.cancelAll()

group.onFailureRestart()

group.onFailureCancelAll()

def subgroup = group.subgroup()



# Control State Sharing

```
List registrations = []  
submissions.each {form →  
    group.task {  
        if (form.process().valid) {  
            registrations << form  
        }  
    }  
}
```

# Control State Sharing

Needs protection

```
List registrations = []
```



```
submissions.each {form →
```

```
    group.task {
```

```
        if (form.process().valid) {
```

```
            registrations << form
```

```
        }
```

```
}
```

```
}
```

# Control State Sharing

```
List registrations = []
submissions.each {form →
  group.task {
    synchronized(registrations) {
      if (form.process().valid) {
        registrations << form
      }
    } } }
```

# Control State Sharing

```
List registrations = []
submissions.each {form →
  group.task {
    synchronized(registrations) {
      if (form.process().valid) {
        registrations << form
      }
    } }}
```

WRONG!  
No possible  
parallelism for  
process()

←  
Needs no lock,  
works on local data  
only

# Control State Sharing

```
List registrations = []
```

```
submissions.each {form →
```

```
    group.task {
```

```
        if (form.process().valid) {
```

```
            synchronized(registrations) {
```

```
                registrations << form
```

```
}
```

```
} }}
```

Keep the scope of  
lock minimal

# Two key components

## Tasks (aka coroutines, goroutines)

- Implemented as (pooled) threads or virtual, lightweight threads

## Data exchange between tasks

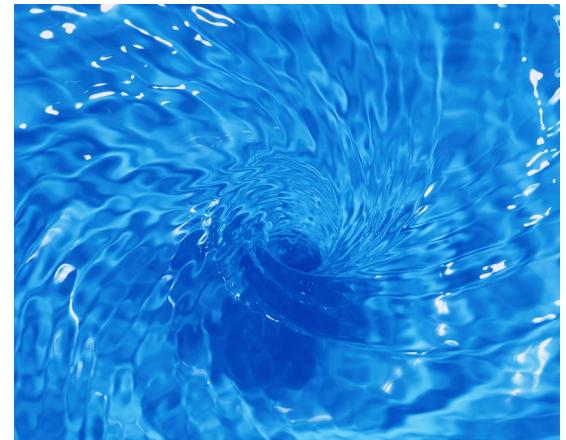
- Tasks' *promised* return value
- Dataflow variables
- Dataflow channels

# Dataflow Variables

No race-conditions

No live-locks

Deterministic deadlocks



# Dataflow Variables

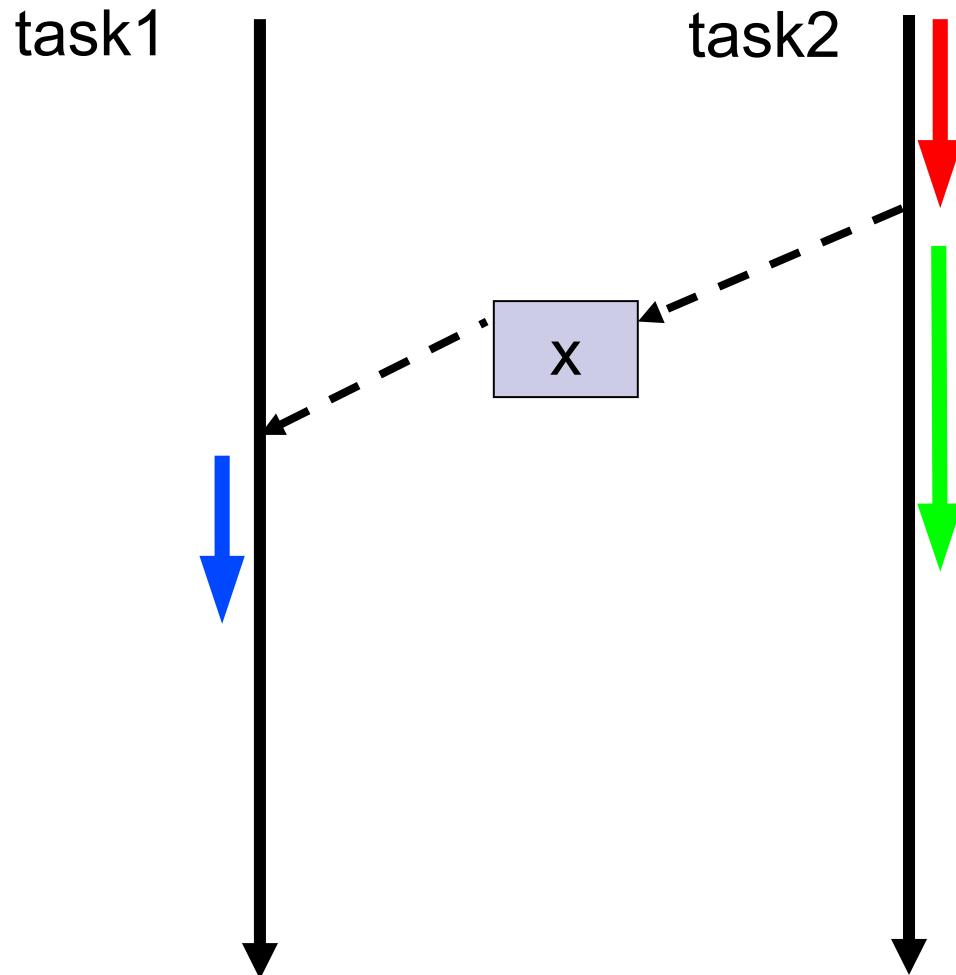
Start empty

*set()* / *bind()* / *<<*

- write a value, if still empty
- make **read-only** since then

*get()* / *val* – wait for a value, then read it

# Dataflow Variables / Promises



# DataflowVariables/Promises

```
def x = new DataflowVariable()
```

```
task { println x.get() }
```

```
task {  
    storeInDB(x.get())  
}
```

```
task { x << 10 }  
x.join()
```

# Promises to exchange data

```
def x,y,z = new DataflowVariable()
```

```
task { z << x.get() + y.get() }
```

```
task { x << 10 }
```

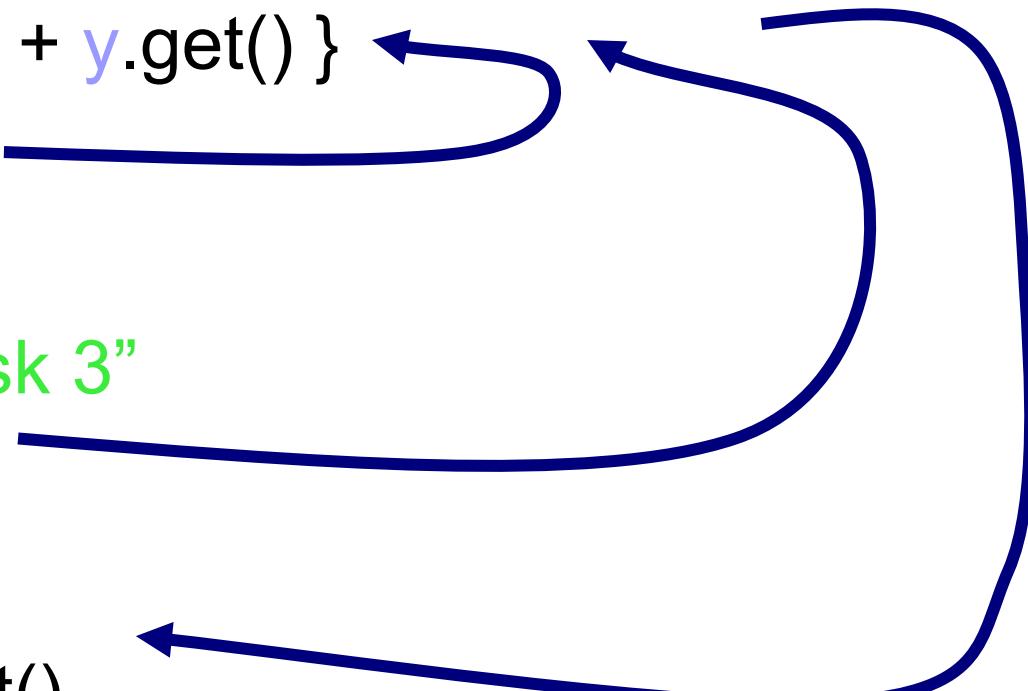
```
task {
```

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

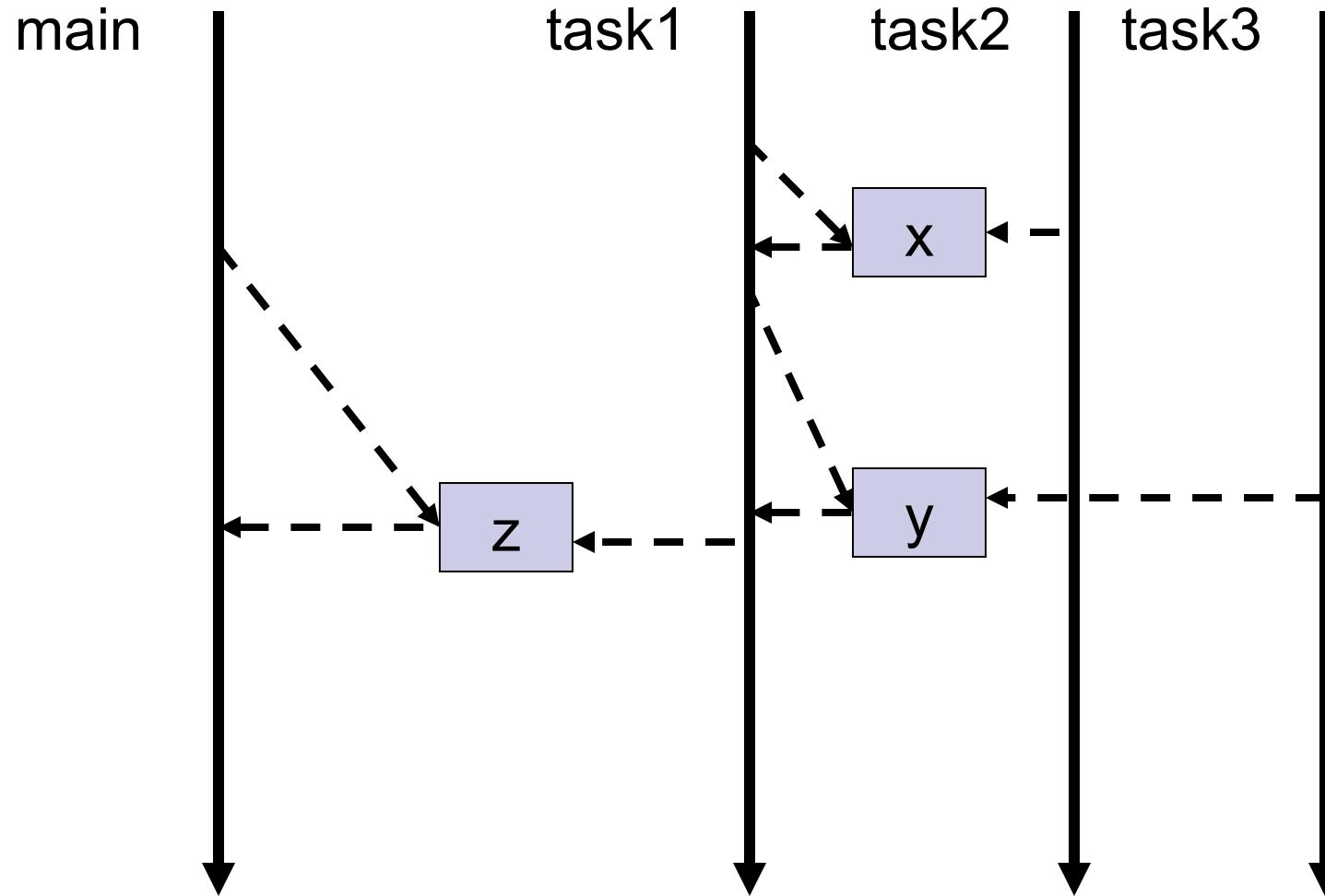
```
    y << 5
```

```
}
```

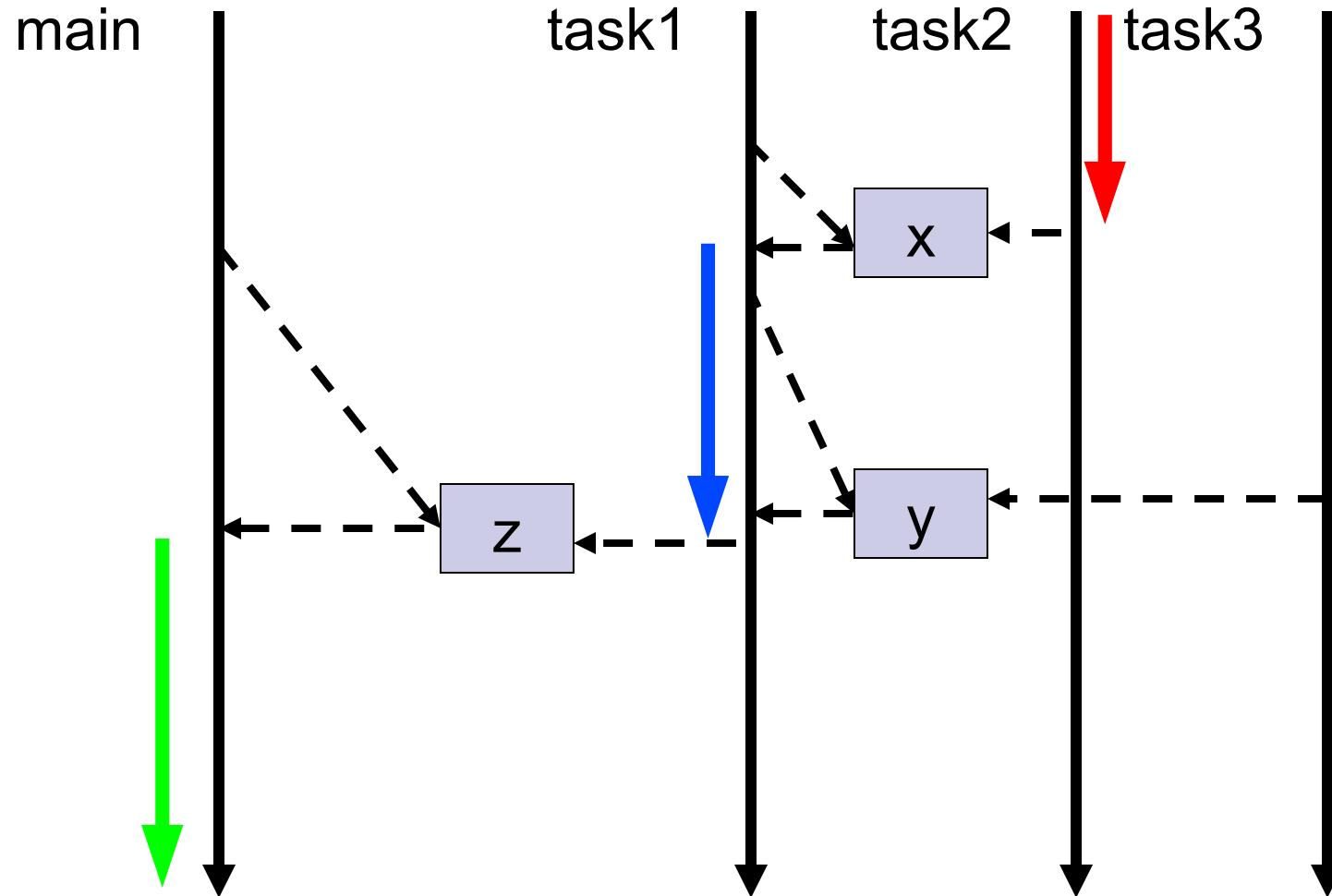
```
assert 15 == z.get()
```



# Dataflow Variables / Promises



# Dataflow Variables / Promises



# Promises as return values

```
Promise x = new DataflowVariable()  
task { x << 10 }
```

vs.

```
Promise x = task { return 10 }
```

# Promises as return values

```
Promise x = task { return 10 }
```

```
Promise y = task {  
    println "I am task 3"  
    return 5
```

```
}
```

```
Promise z = task {x.get() + y.get() }  
assert 15 == z.get()
```

# Promises as return values

```
Promise t1 = task {  
    def data = loadDataFromDB("Joe")  
    data.optimize()  
    return data  
}  
  
Promise t2 = task { printData(t1.get()) }  
t2.join()
```

# Kotlin coroutines



```
val time = measureTimeMillis {  
    val one = async { doSomethingUsefulOne() }  
    val two = async { doSomethingUsefulTwo() }  
    println("The answer is ${one.await() + two.await()}")  
}  
println("Completed in $time ms")
```



# Reactive/callback approach

- Needed when tasks are implemented on top of system threads (*unlike virtual/lightweight threads*)
- Blocking in `Promise.get()` is too expensive

Allows for callbacks to be registered on `Promises`

# Callbacks

```
Promise x = new DataflowVariable()
```

```
x.then { println it }
```

```
task {
```

```
  x.then { storeInDB(it) }
```

```
    .then {dbStatus → println dbStatus}
```

```
}
```

```
task { x << 10 }
```

```
x.join()
```

# Chaining promises

```
def h1 = download('url') then {text → text.trim()} then hash
```

# Chaining promises

```
def h1 = download('url') then {text → text.trim()} then hash
```

```
def h1 = download('url') | {text → text.trim()} | hash
```

# Error handling

```
url.then(download)  
  .then(calculateHash)  
  .then(formatResult)  
  .then(printResult, printError)  
  .then(sendNotificationEmail);
```

# CompletableFuture (Java)

```
final CompletableFuture<Integer> f = CompletableFuture.supplyAsync(() -> 10);

final CompletableFuture<Integer> f2 = f.thenApplyAsync(x -> 2 * x)
    .handleAsync((x, e) -> 3 * x);

final CompletableFuture<Integer> f3 = f.thenComposeAsync(x -> CompletableFuture.supplyAsync(() -> 2 * x))
    .handleAsync((x, e) -> 3 * x);

final CompletableFuture<Integer> f4 = f2.thenCombineAsync(f3, (x, y) -> x + y);
```

# Glue tasks together

```
Promise c1 = task { compile(module1) }
```

```
Promise c2 = task { compile(module2) }
```

# Glue tasks together

```
Promise c1 = task { compile(module1) }
```

```
Promise c2 = task { compile(module2) }
```

```
Promise j1 = c1.then {jar it}
```

```
Promise j2 = c2.then {jar it}
```

# Glue tasks together

```
Promise c1 = task { compile(module1) }
```

```
Promise c2 = task { compile(module2) }
```

```
Promise j1 = c1.then {jar it}
```

```
Promise j2 = c2.then {jar it}
```

```
whenAllBound(j1, j2) {m1, m2 → deploy(m1, m2)}
```

```
j1.then {pushToRepo it}
```

# Glue tasks together

```
Promise c1 = task { compile(module1) }
```

```
Promise c2 = task { compile(module2) }
```

```
Promise j1 = c1.then {jar it}
```

```
Promise j2 = c2.then {jar it}
```

```
whenAllBound(j1, j2) {m1, m2 → deploy(m1, m2)}
```

```
j1.then {pushToRepo it}
```

```
iWillSendEmailWhenJarred(j1)
```

# Glue tasks without callbacks

```
Promise c1 = task { compile(module1) }
```

```
Promise c2 = task { compile(module2) }
```

```
Promise j1 = task { jar c1.get() }
```

```
Promise j2 = task { jar c2.get() }
```

```
task { deploy(j1.get(), j2.get()) }
```

```
task {pushToRepo j1.get()}
```

```
iWillSendEmailWhenJarred(j1.get())
```

# Dataflow variables wrap-up

- *Promise* is the reading side
- Blocking read – *val*, *get()*, *await()*
- Callbacks – *then*, *|*, *whenAllBound*
- Callback handlers can be chained
- Promises are typically returned from `async` calls – *task*, *async*

# Abstractions

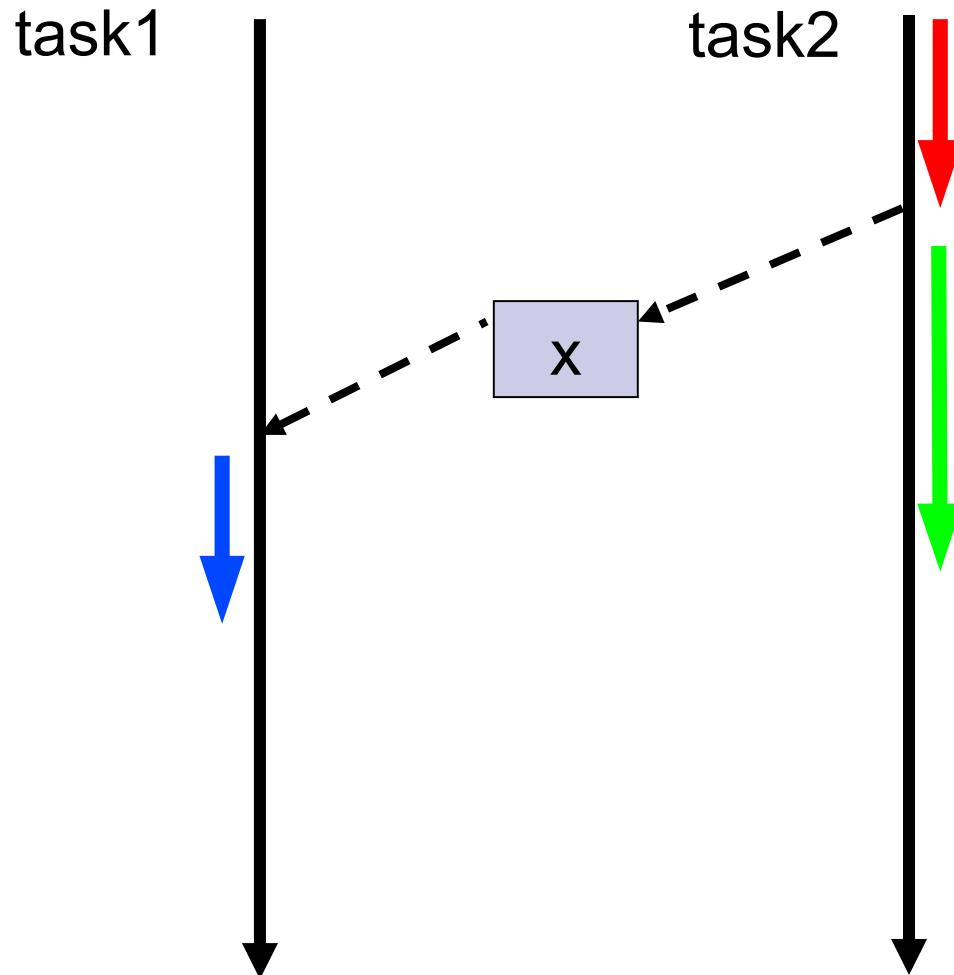
## Tasks (aka coroutines, goroutines)

- Implemented as (pooled) threads or virtual, lightweight threads

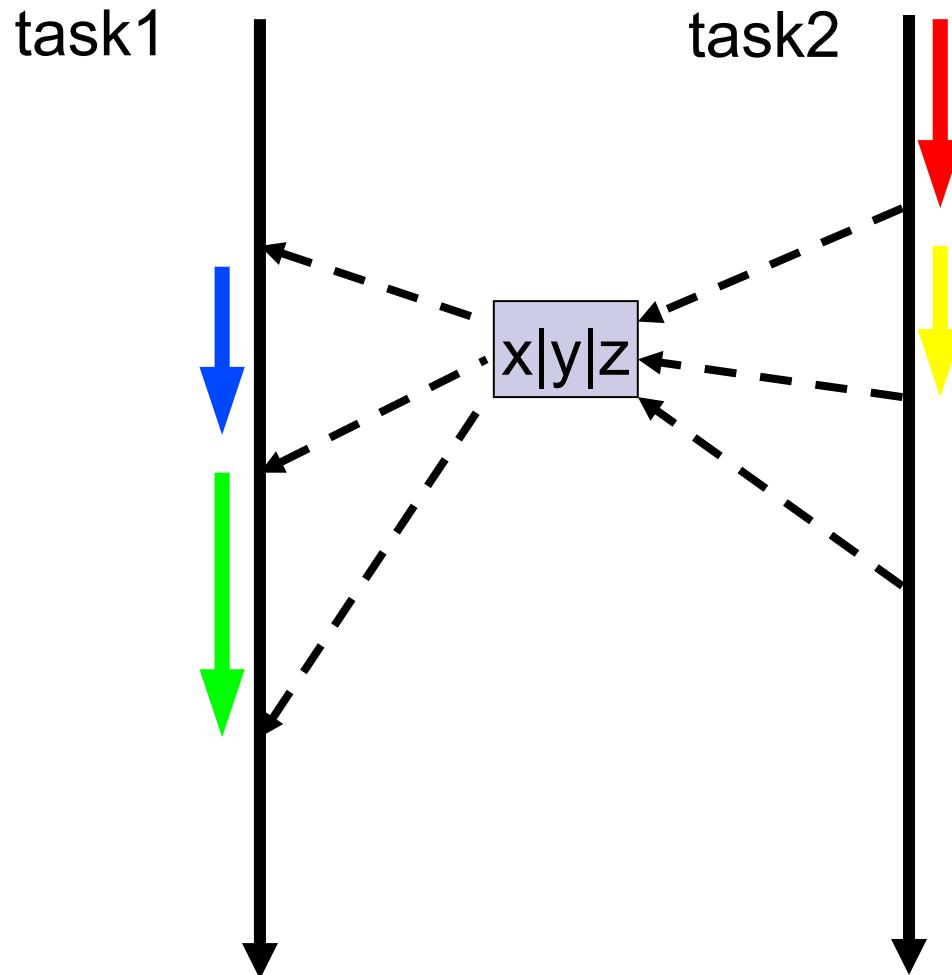
## Data exchange between tasks

- Tasks' return value
- Dataflow variables
- **Dataflow channels**

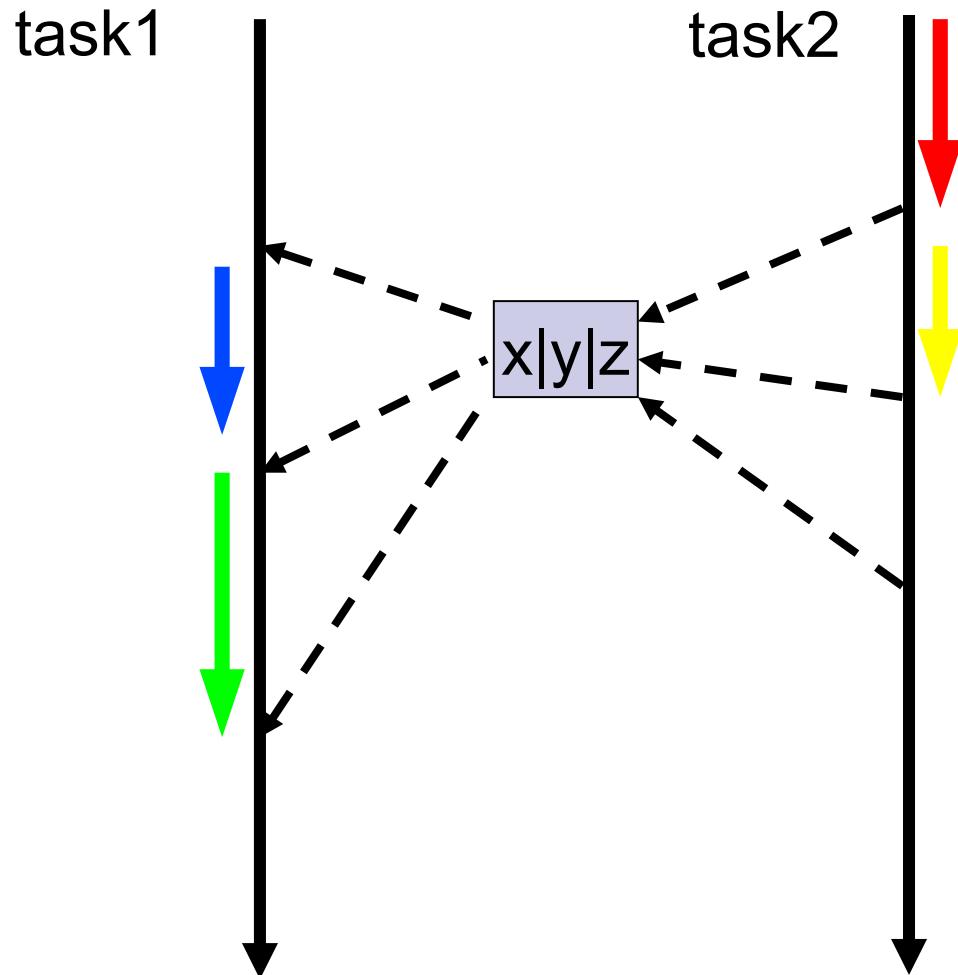
# Dataflow Variables / Promises



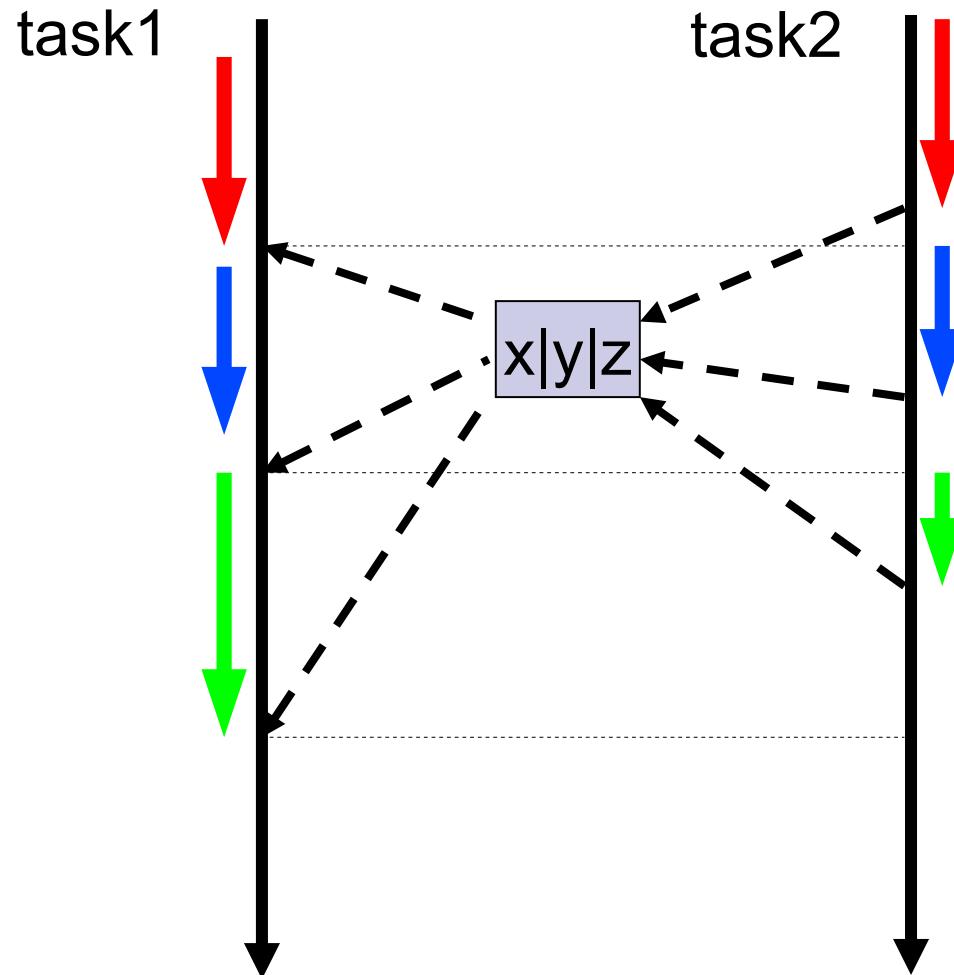
# Dataflow Channels



# Buffered Channels



# Synchronous Channels



# Tasks as processes (CSP)

```
group.task {  
    doStuff()  
  
    logChannel << 'initialized'  
  
    def result = doWork(workQueue.get())  
    if (result.isError) errors << result  
    else results << result  
  
    logChannel << 'finished'  
}
```

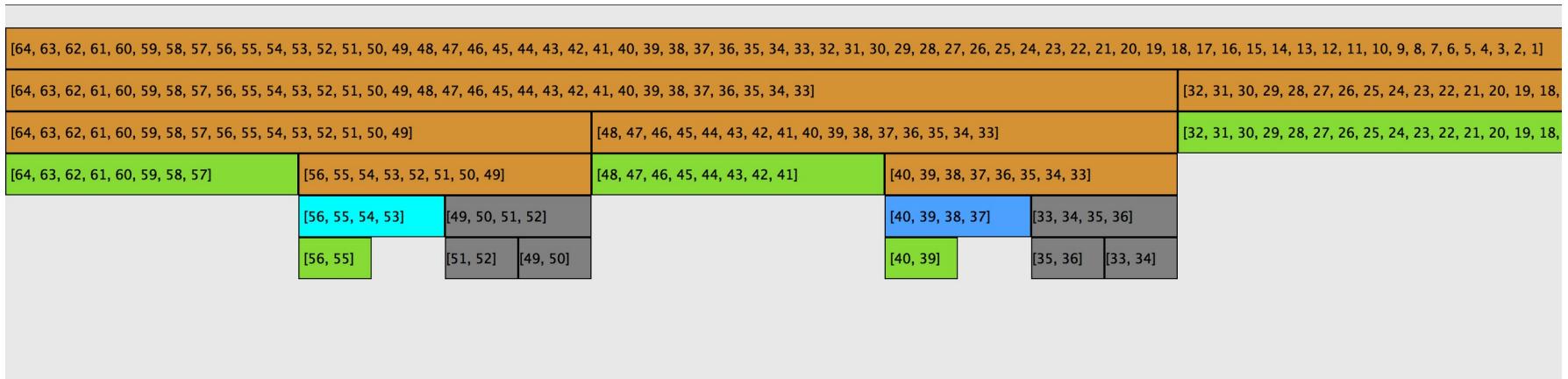
# Channel Selection

```
Select alt = group.select(validForms, invalidForms)
```

```
SelectResult selectResult = alt.select() //alt.prioritySelect()
```

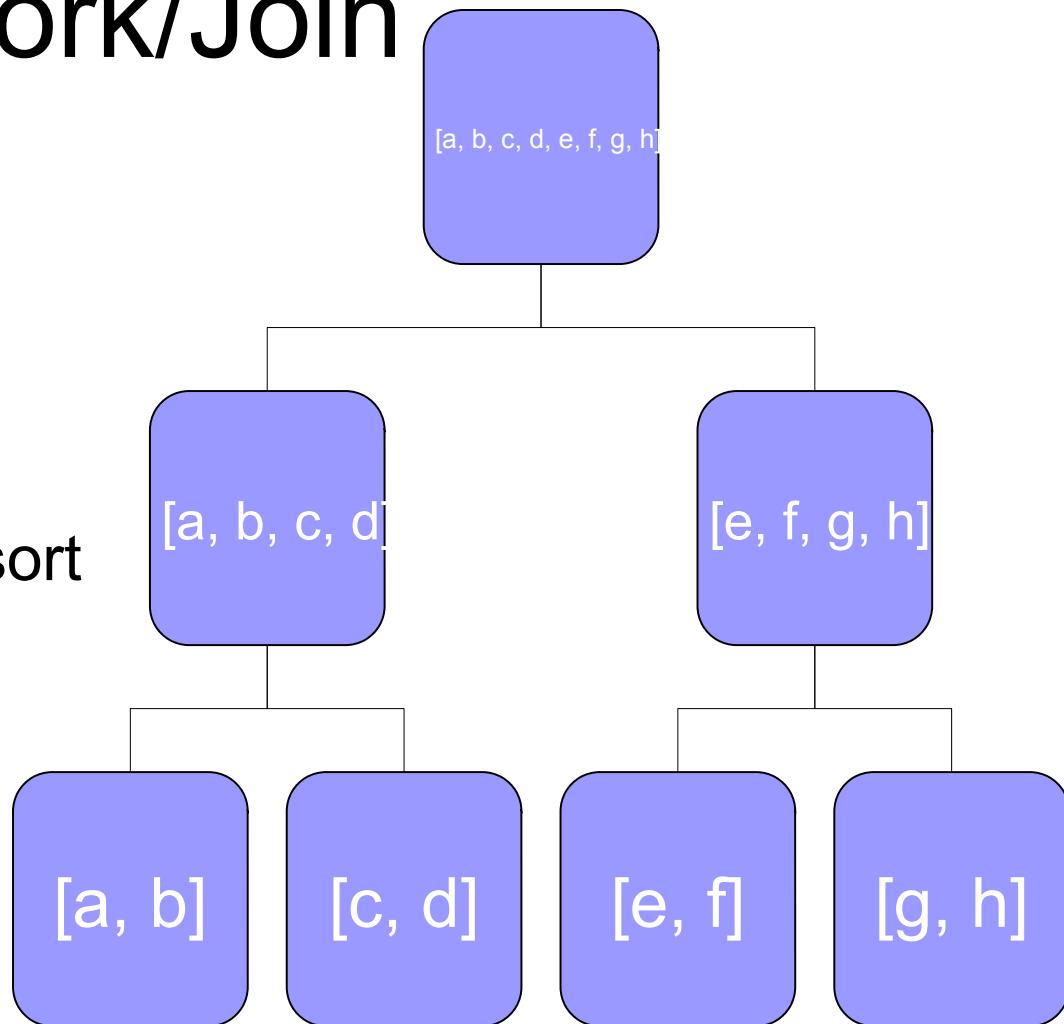
```
switch (selectResult.index) {  
    case 0: registrations << selectResult.value; break  
    case 1: ...  
}
```

# Hierarchical decomposition



# Fork/Join

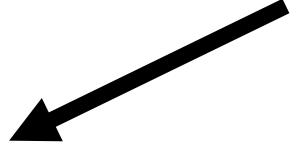
- Solve hierarchical problems
  - Divide and conquer
  - Merge sort, Quick sort
  - Tree traversal
  - File scan / search
  - ...



# Fork/Join (GParS)

```
{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!



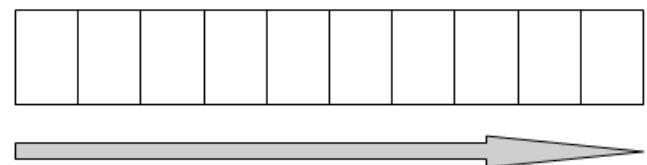
# Geometrical decomposition

images.eachParallel {it.process()}

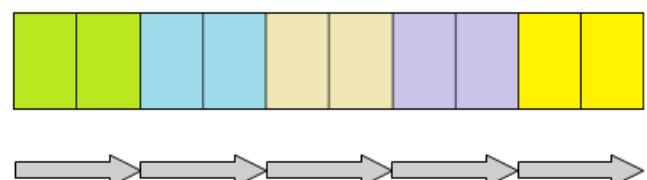
documents.sumParallel()

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

1 thread



5 threads



# Geometric decomposition

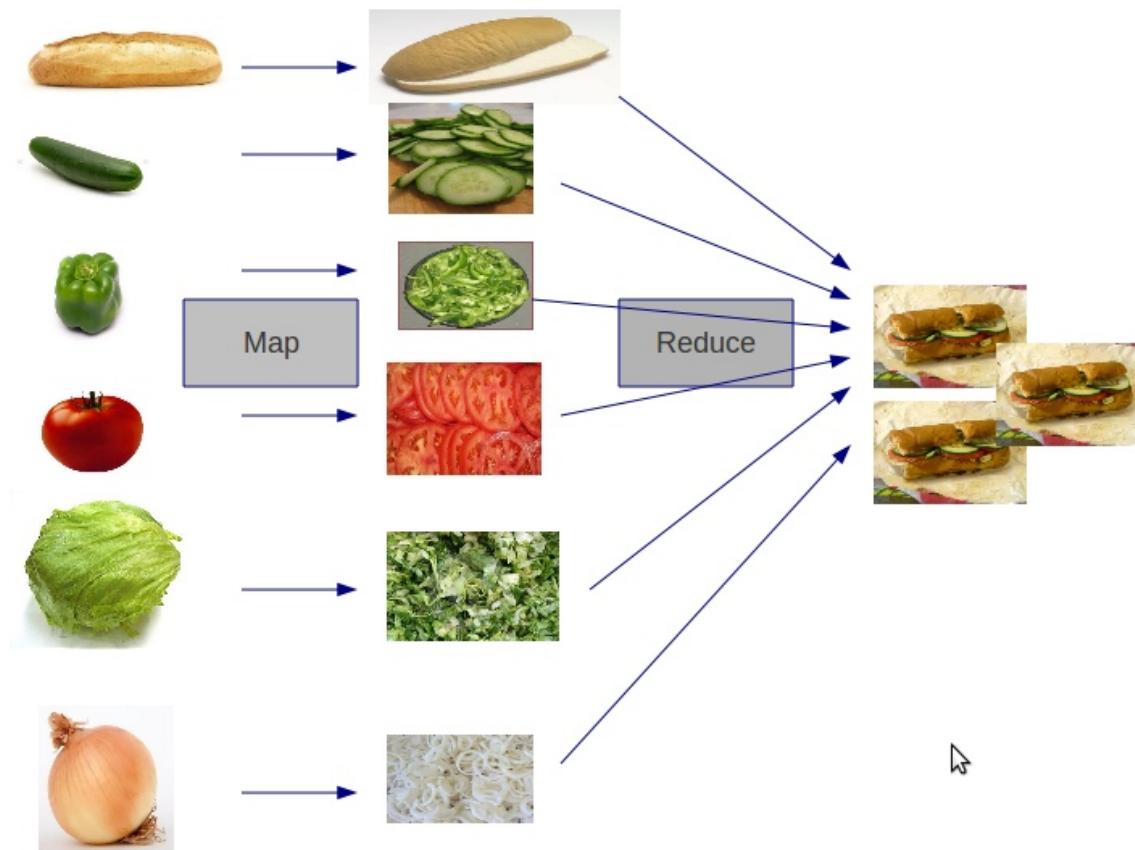
registrations = submissions

```
.collectParallel { form -> form.process() }  
.findAllParallel { it.valid }
```

registrations = submissions.parallel

```
.map { form -> form.process() }  
.filter { it.valid }.collection
```

# Map - reduce



# Frequent confusion



## parallel quick sort outdone by single threaded quicksort

---

-  I've been reading , here is the example in the book using futures to implement parallel quick sort.
-  0 But I found this function is more than twice slower than the single threaded quick sort function without using any asynchronous facilities in c++ standard library. Tested with g++ 4.8 and visual c++ 2012.
-  I used 10M random integers to test, and in visual c++ 2012,this function spawned 6 threads in total to perform the operation in my quad core PC.

I am really confused about the performance. Any body can tell me why?

# GPU



# Improper use 1

```
def accumulator = 0

myCollection.eachParallel {
    accumulator += calculate(it)
}

}
```

# Do not accumulate, map-reduce!

```
def accumulator = myCollection.parallel  
    .map {calculate(it)}  
    .reduce {a, b → a + b}
```

# Improper use 2

```
new File("/file.txt").withReader{reader ->  
    reader.eachParallel {  
        def r1 = step1(r)  
        def r2 = step2(r1)  
        def r3 = step3(r2)  
    }  
}
```

# Unroll iterations into streams

```
def pipeline = data | step1 | step2 | step3
```

```
new File("/file.txt").withReader{reader ->  
    reader.each {  
        data << it  
    }  
}
```

# Unroll iterations

```
[project1, project2, project3, project4].eachParallel {
```

*checkout it*

*compile it*

*package it*

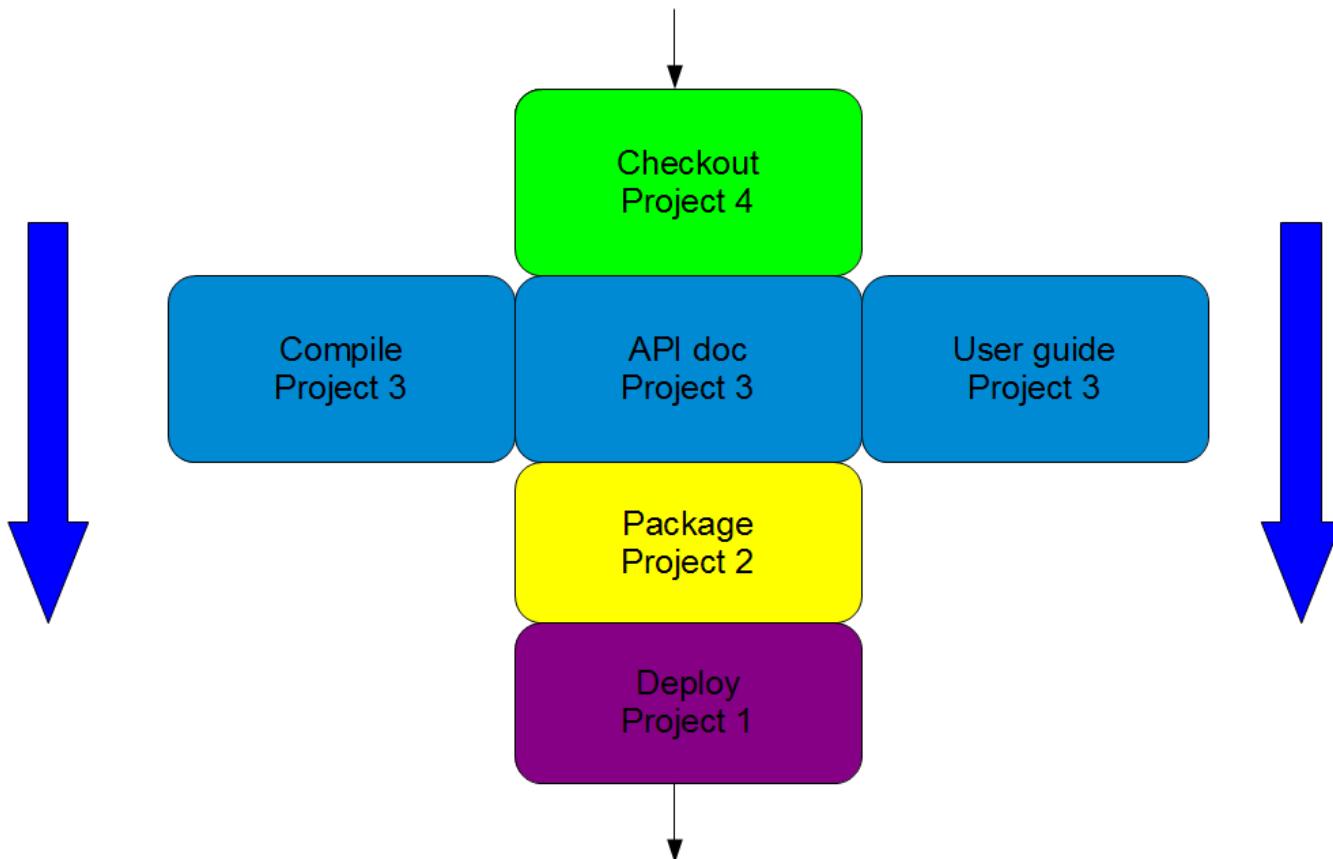
*deploy it*

```
}
```

vs.

```
def results = projects | checkout | compile | package | deploy
```

# Unroll iteration



# Reactive streams

# Pipeline DSL

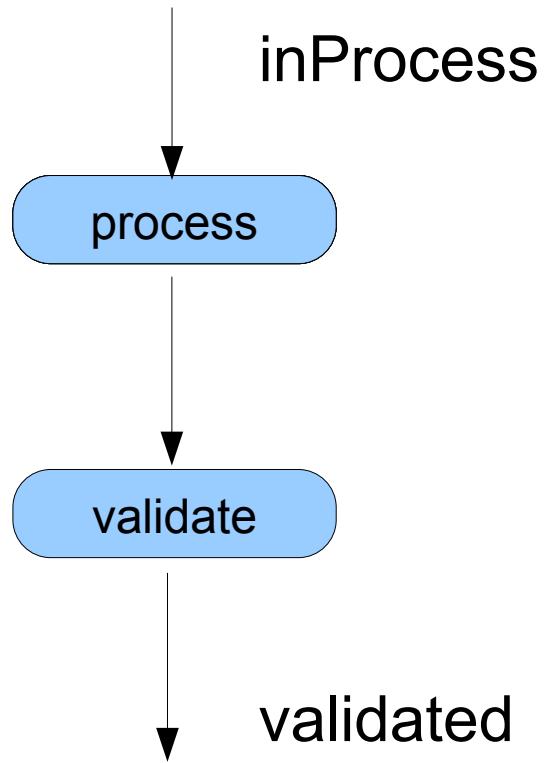
```
def toProcess = new DataflowQueue()
```

```
def validated = new DataflowQueue()
```

```
toProcess | {form -> process(form)} |
```

```
{processedForm -> validate(processedForm)} | validated
```

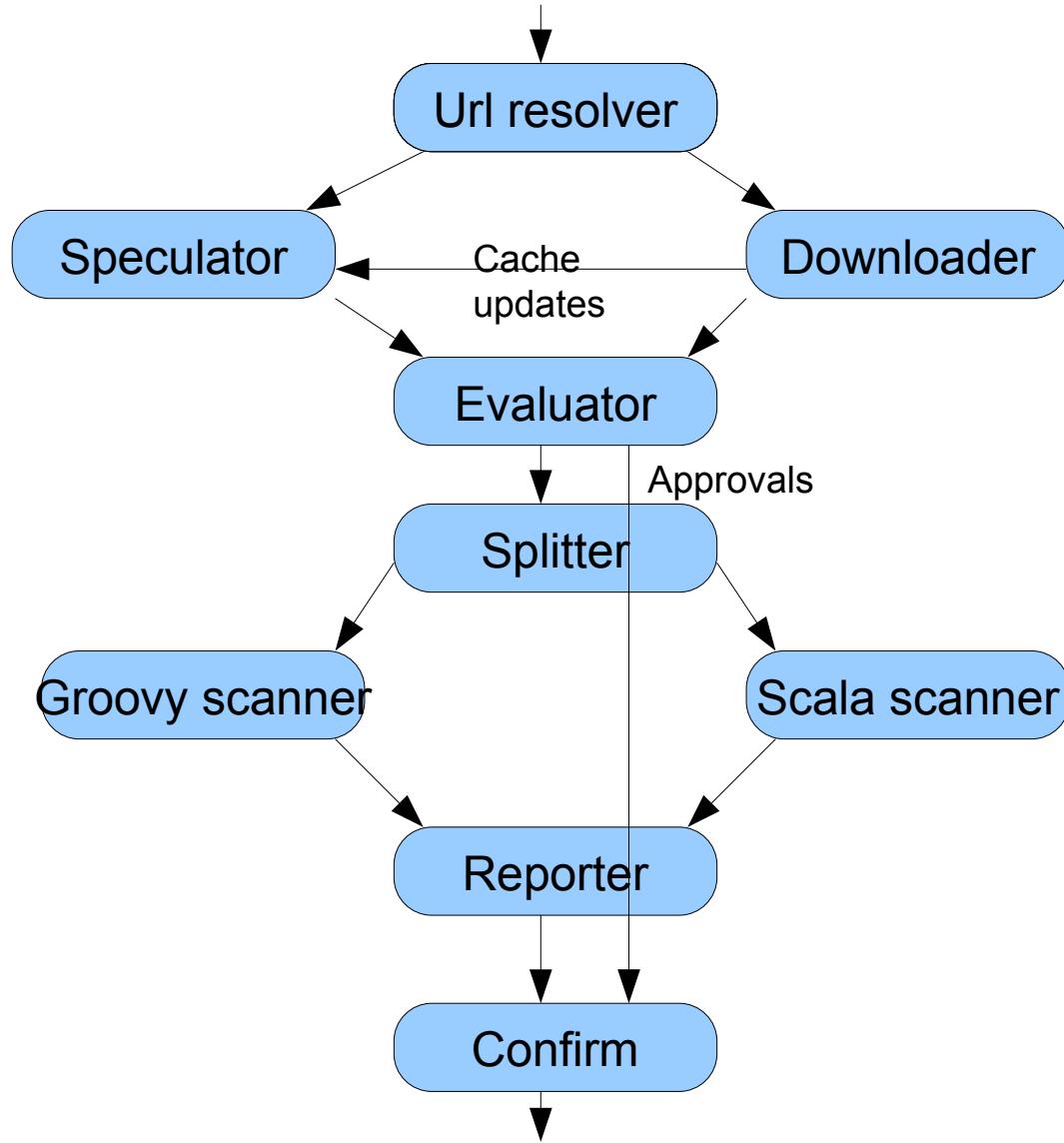
```
submissions.each {toProcess << it} //feed data to the pipe
```



# Generalization

An operation can take multiple inputs and outputs

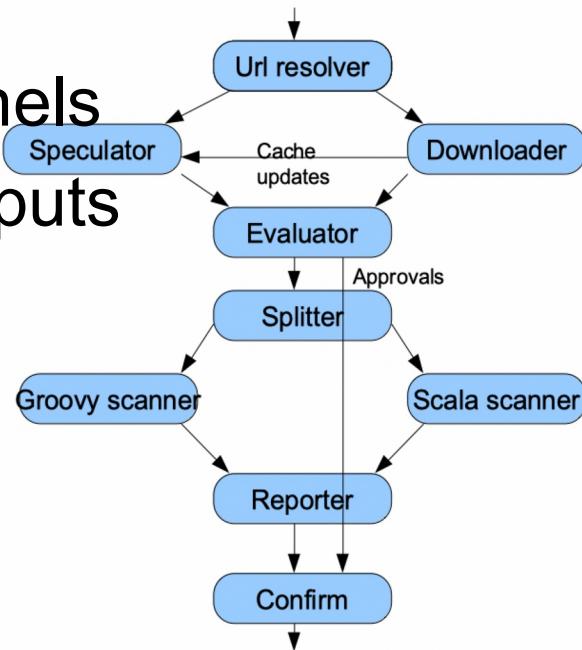
Operations addressed indirectly using channels



# Dataflow Operators

Like functions

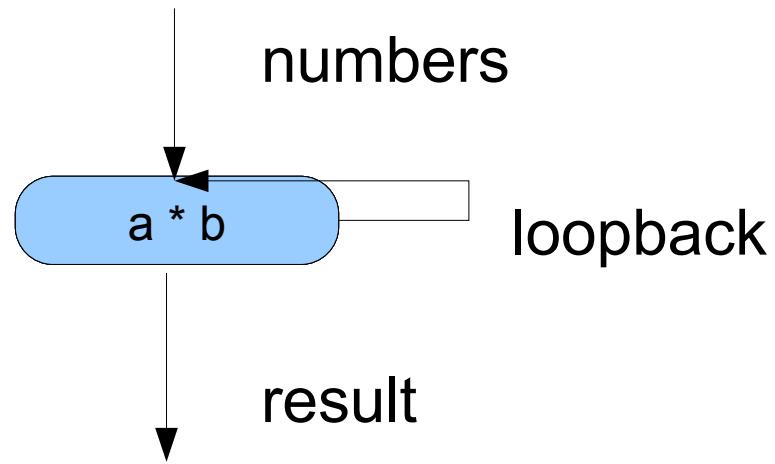
- wired together with channels
- possibly with multiple outputs
- may be recursive

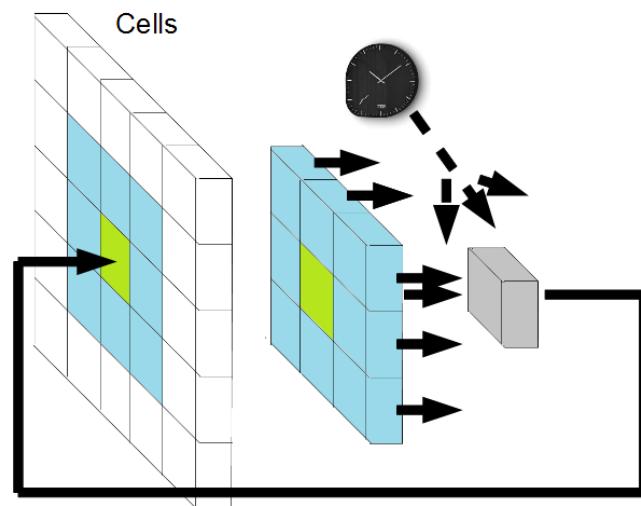
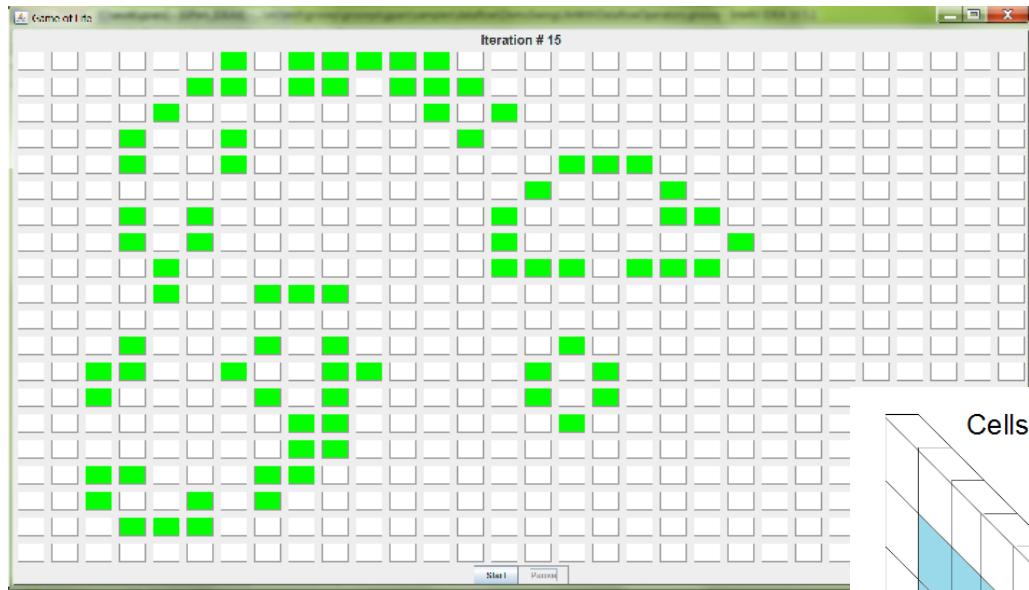


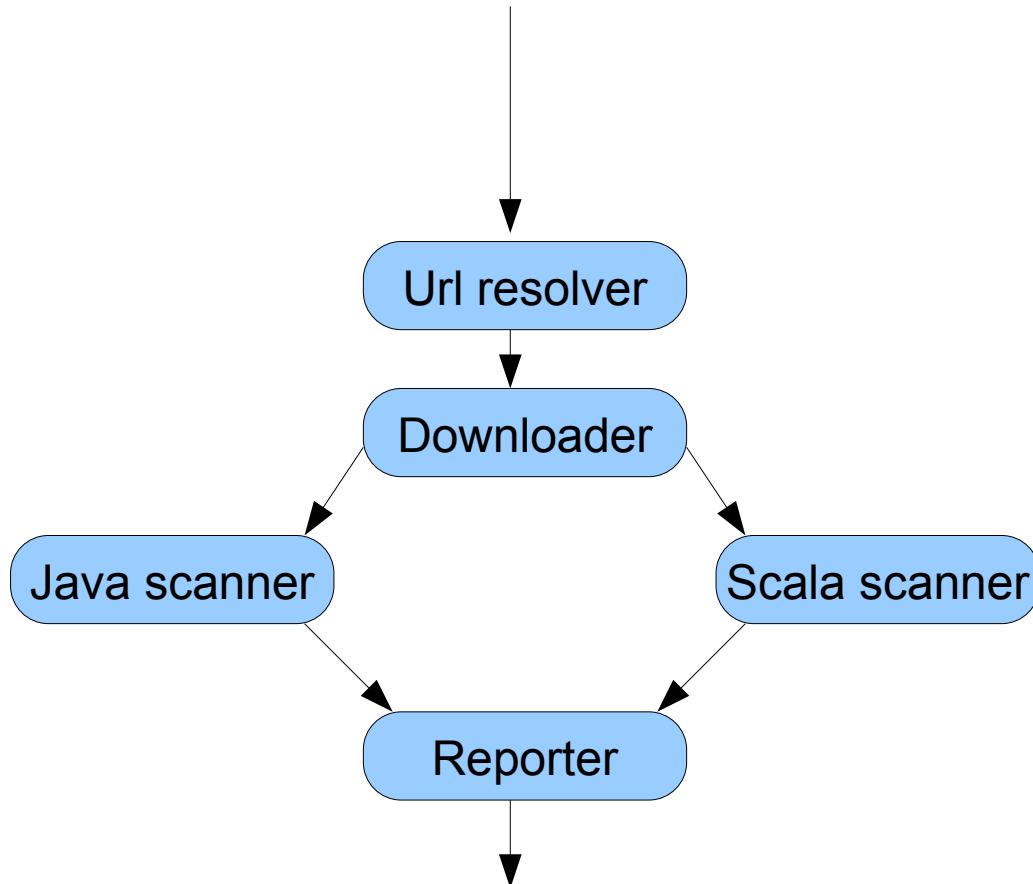
# Dataflow Operators

```
operator(inputs: [headers, bodies, footers],  
        outputs: [articles, summaries])  
{header, body, footer ->  
    def article = buildArticle(header, body, footer)  
    bindOutput(0, article)  
    bindOutput(1, buildSummary(article))  
}
```

# Factorial







# Vision/trend

- Structured concurrency and reactive streams go well together

# Shared mutable state

# High-level abstractions

- STM
- Agents
- Actors

# 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>

# Control State with STM

Needs protection

```
List registrations = []
```



```
submissions.each {form →
```

```
    group.task {
```

```
        if (form.process().valid) {
```

```
            registrations << form
```

```
        }
```

```
}
```

```
}
```

# Control State with STM

```
List registrations = []  
submissions.each {form →  
    group.task {  
        atomic {  
            if (form.process().valid) {  
                registrations << form  
            }  
        }  
    }  
}
```

# Control State with STM

Which List class?

List registrations = [] 

submissions.each {form →

group.task {

***atomic*** {

if (form.process().valid) {

    registrations << form

}

} }

# Persistent Data Structures

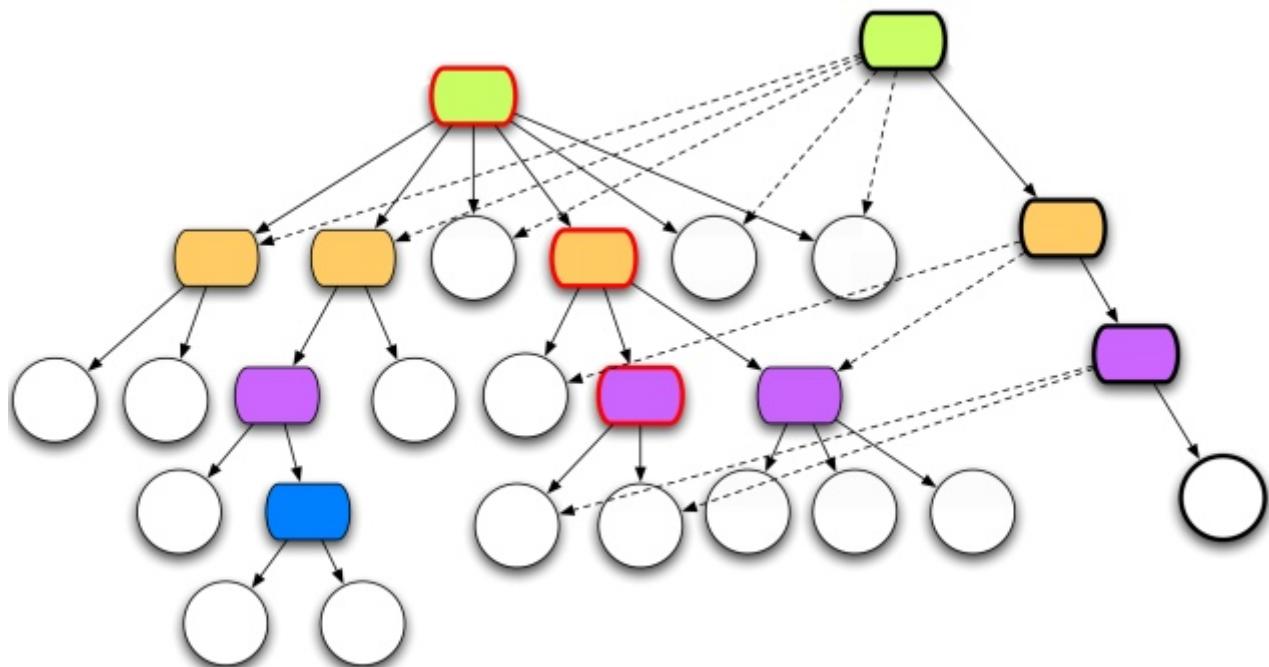


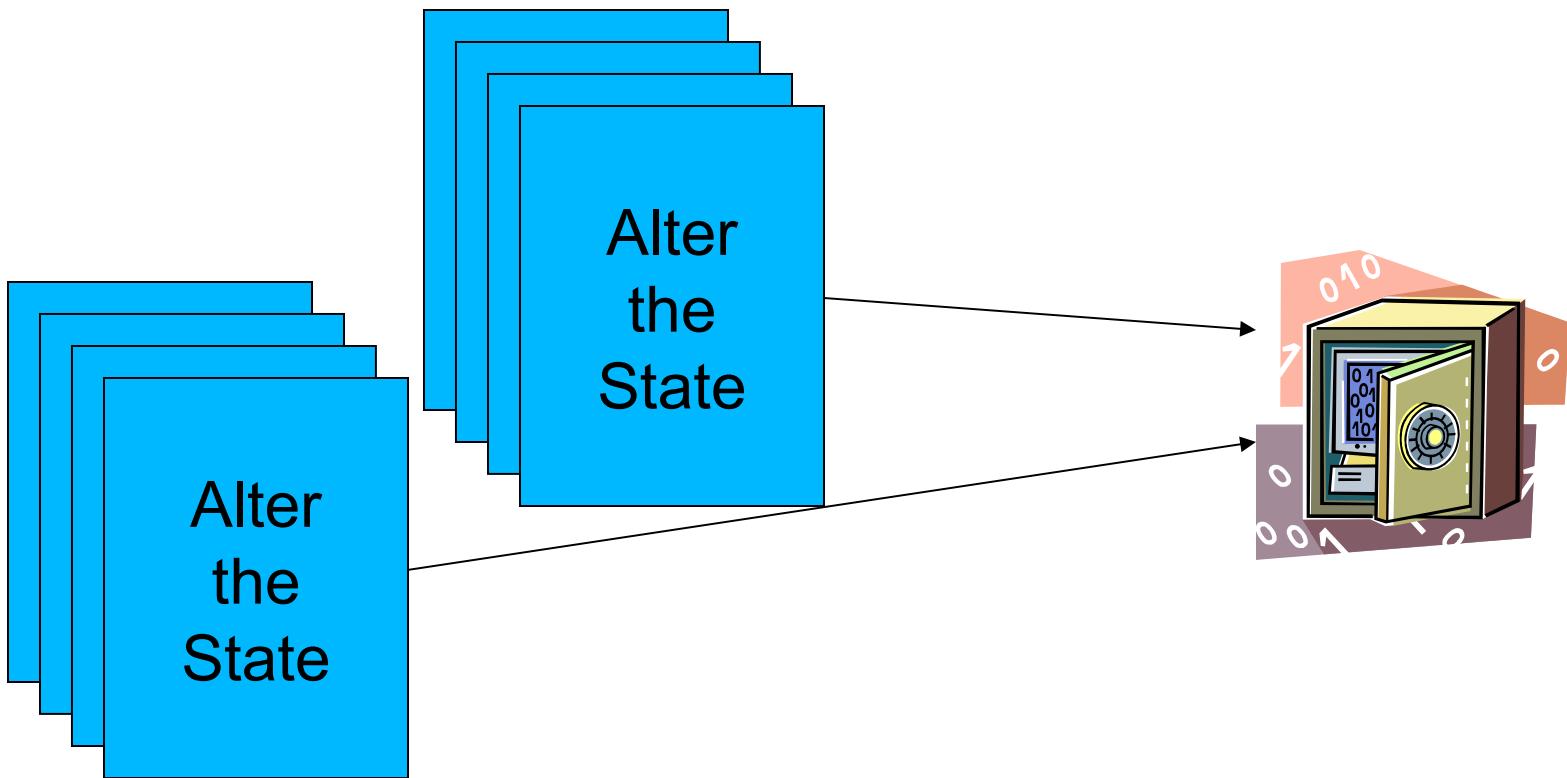
Illustration taken from Rich Hickey's presentation. Copyright Rich Hickey 2009

# High-level abstractions

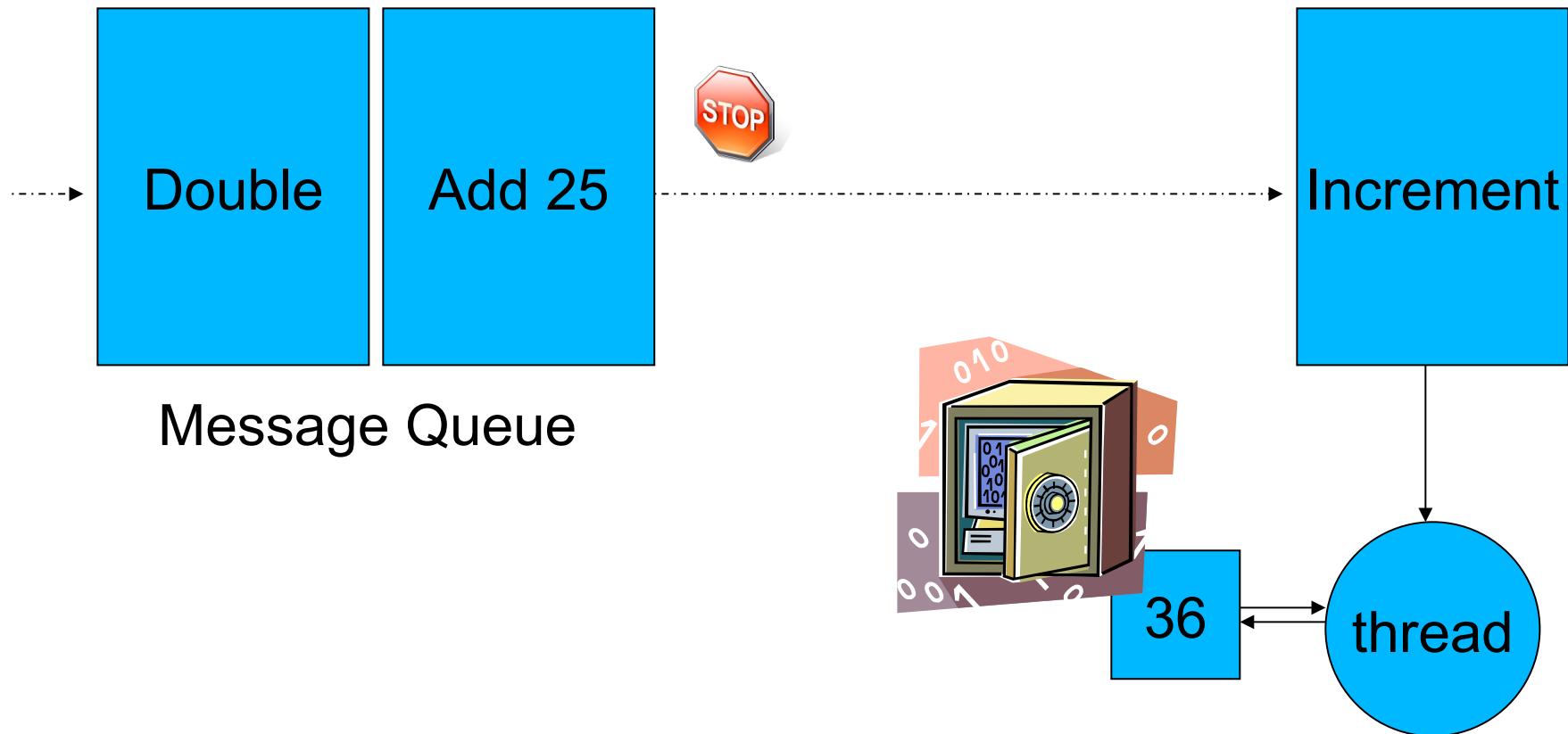
- STM
- Agents

# Agent

Lock Shared Mutable State in a Safe



# Agent inside



# Control State with Agents

Needs protection

```
List registrations = [] ←  
submissions.each {form →  
    group.task {  
        if (form.process().valid) {  
            registrations << form  
        }  
    }  
}
```

# Control State with Agents

```
Agent regAgent = new Agent( [] )  
submissions.each {form →  
    task {  
        if (form.process().valid) {  
            regAgent.send {List regs → regs.add form}  
        }  
    }  
}
```

# Actors

Isolated state

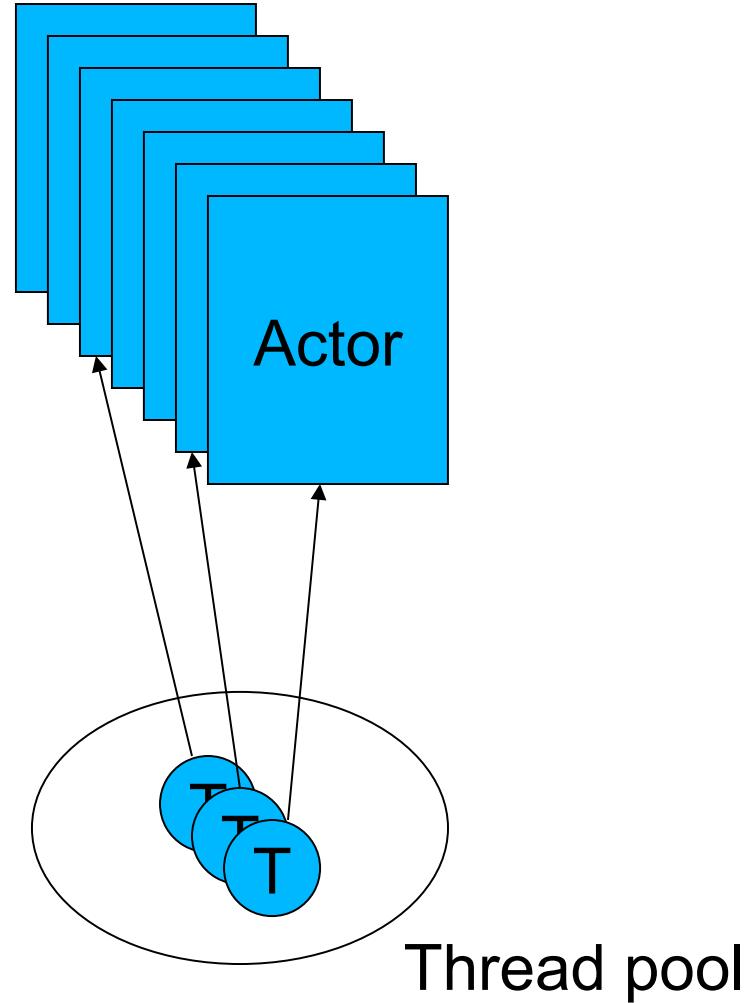
Active - shared threads

Async Communication

- Direct addressing
- Immutable messages

Activities:

- Create a new actor
- Send a message
- Receive a message



# Actors vs. Agents

Agents send commands

Actors send data

*Both use direct addressing (no channels)*

# Actor definition

```
class MyActor extends DynamicDispatchActor {  
    private int counter = 0  
  
    public void onMessage(String msg) {  
        this.counter += msg.size()  
    }  
    public void onMessage(Integer number) {  
        this.counter += number  
    }  
    public void onMessage(Money cash) {  
        this.counter += cash.amount  
        reply 'Thank you'  
    }  
}
```

# Sending messages

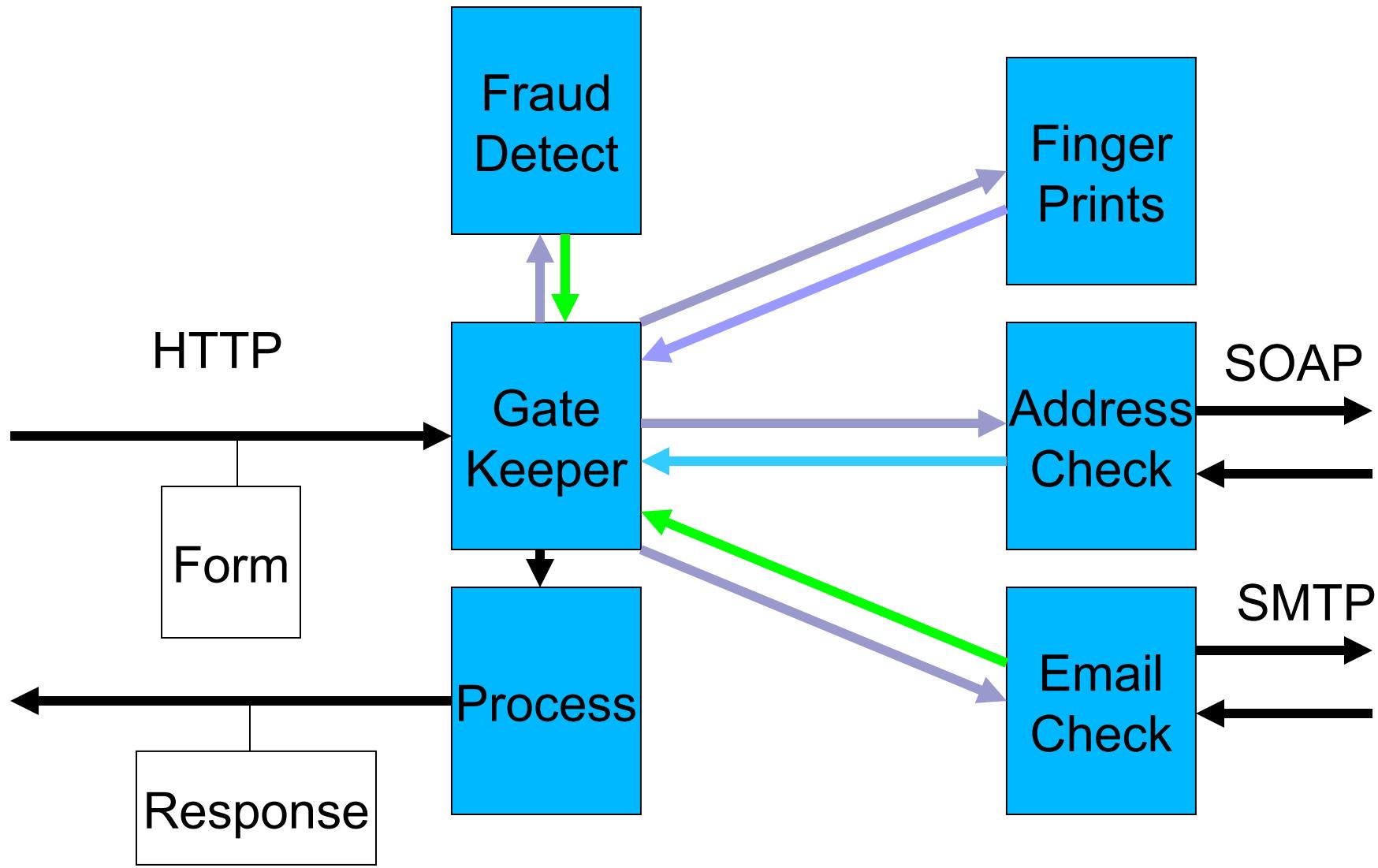
```
buddy.send 10.eur
```

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

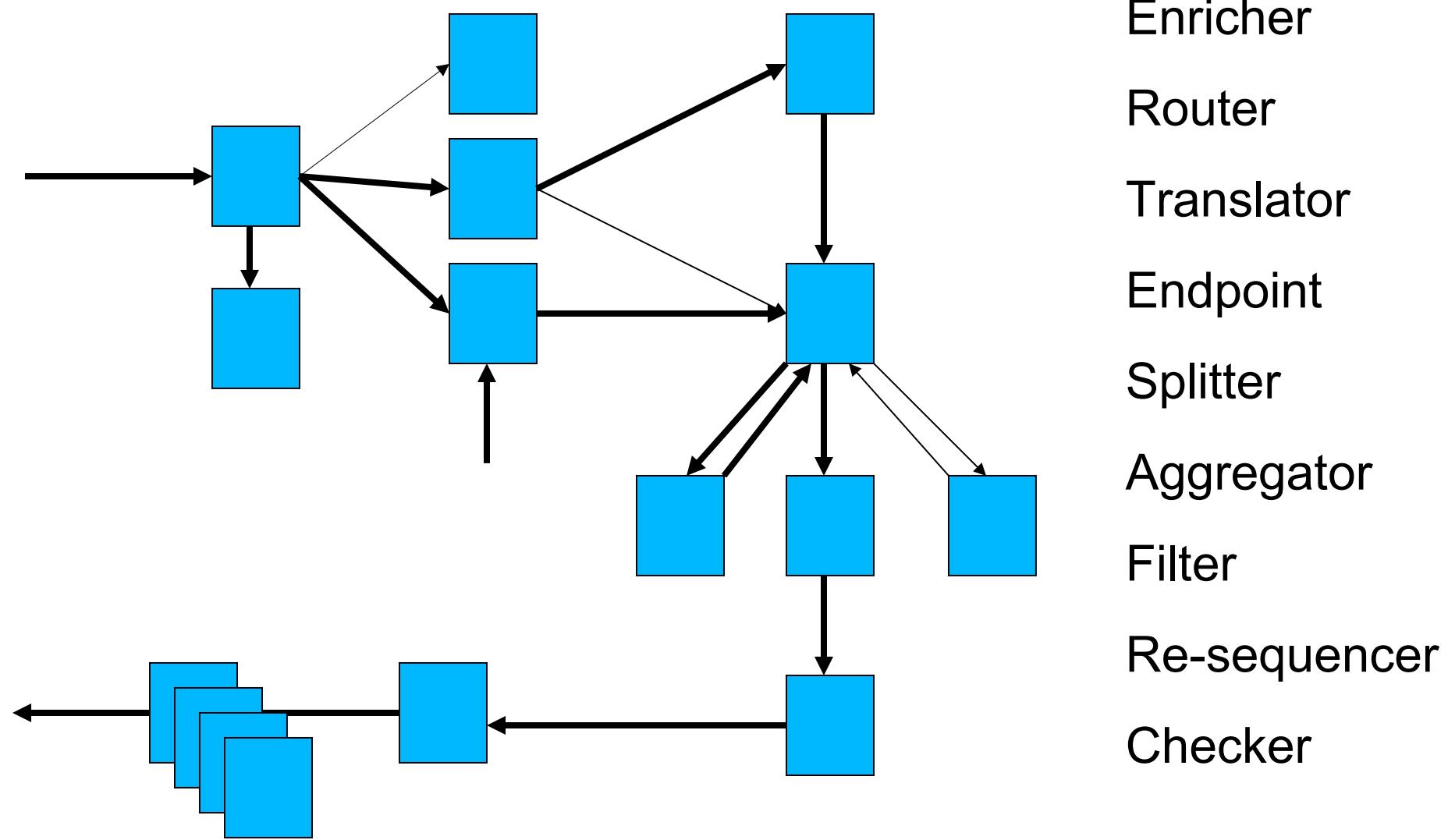
```
def canChat = buddy.sendAndWait 'Got time?'
```

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

# Actors use



# Actors patterns



# Active objects

```
@ActiveObject
```

```
class MyCounter {
```

```
    private int counter = 0
```

```
@ActiveMethod
```

```
def incrementBy(int value) {
```

```
    println "Received an integer: $value"
```

```
    this.counter += value
```

```
}
```

```
}
```

# Summary

Parallelism is not hard, multi-threading is

Jon Kerridge, Napier University

# References

<http://groovy-lang.org>

<http://grails.org>

<https://docs.oracle.com/en/java/javase/20/core/virtual-threads.html>

<https://akka.io/>