

# High-level concurrency concepts



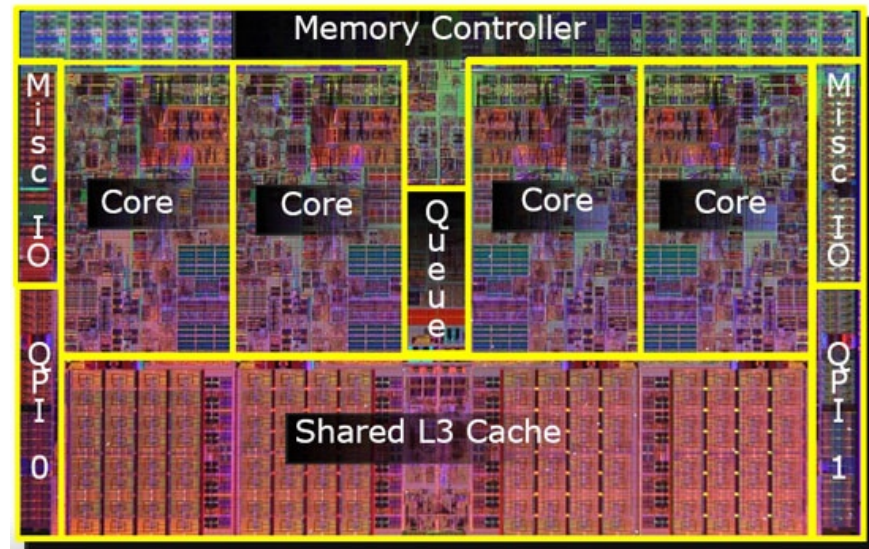
**Václav Pech**

*NPRG014 2020/2021*

<http://www.vaclavpech.eu>

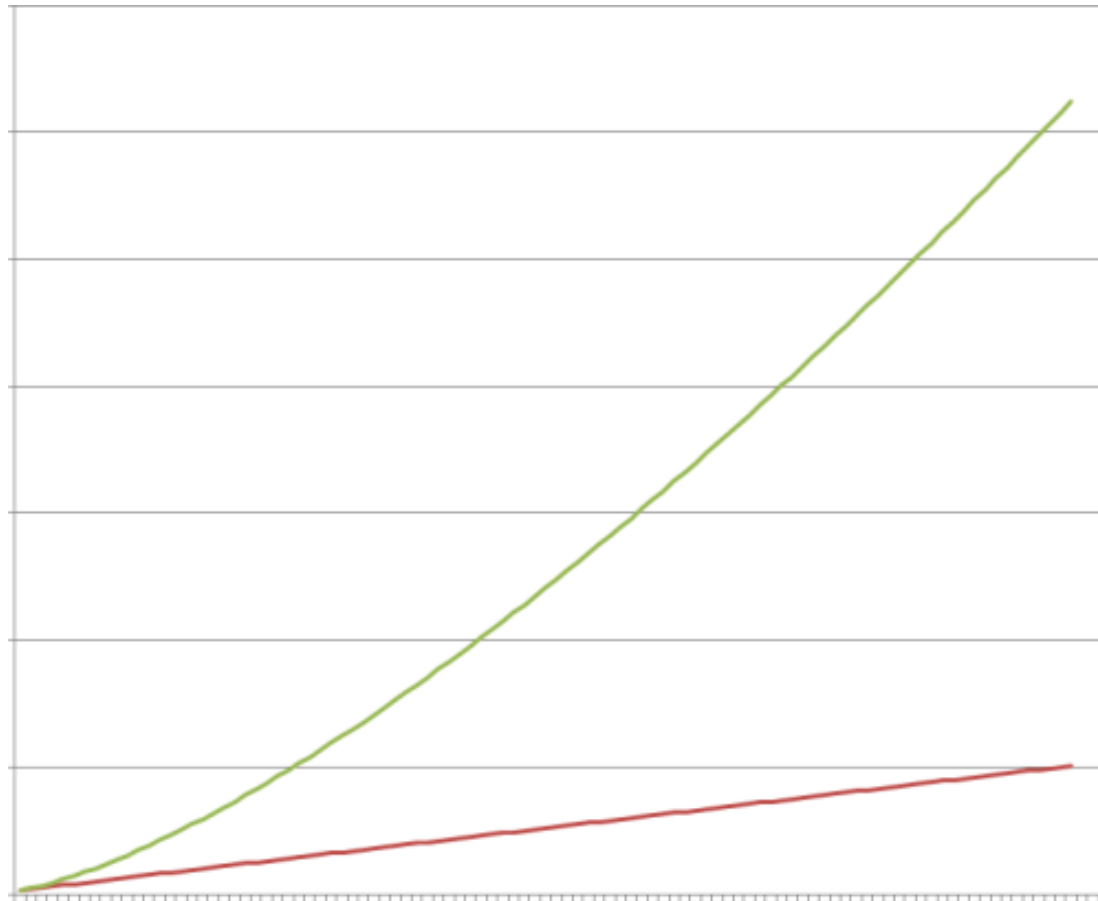
@vaclav\_pech

# Why concurrency?



We're all in the parallel computing business!

# # of cores



# JVM machinery

Thread, Runnable, Thread Pools

# JVM machinery

Thread, Runnable, Thread Pools

Synchronized blocks

Volatile

Locks

Atomic

# 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 Long count = 0;  
  
    public doSomething() {  
        synchronized (count) {  
            count++;  
        }  
    }  
}
```

# Dealing with threads sucks!

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

# Dealing with threads sucks!

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

# Stone age of parallel SW

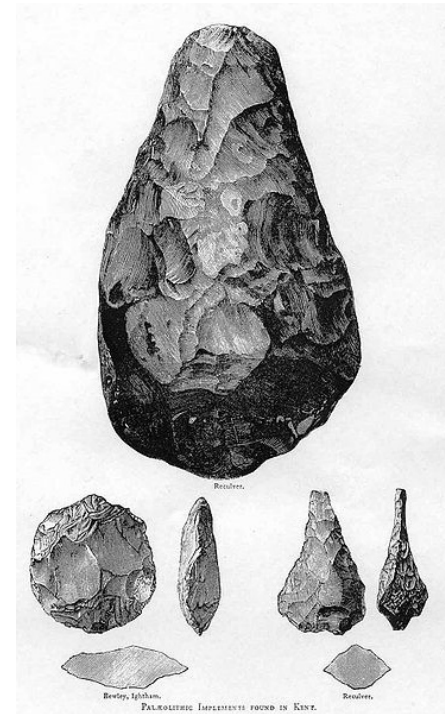
Dead-locks

Live-locks

Race conditions

Starvation

Shared Mutable State

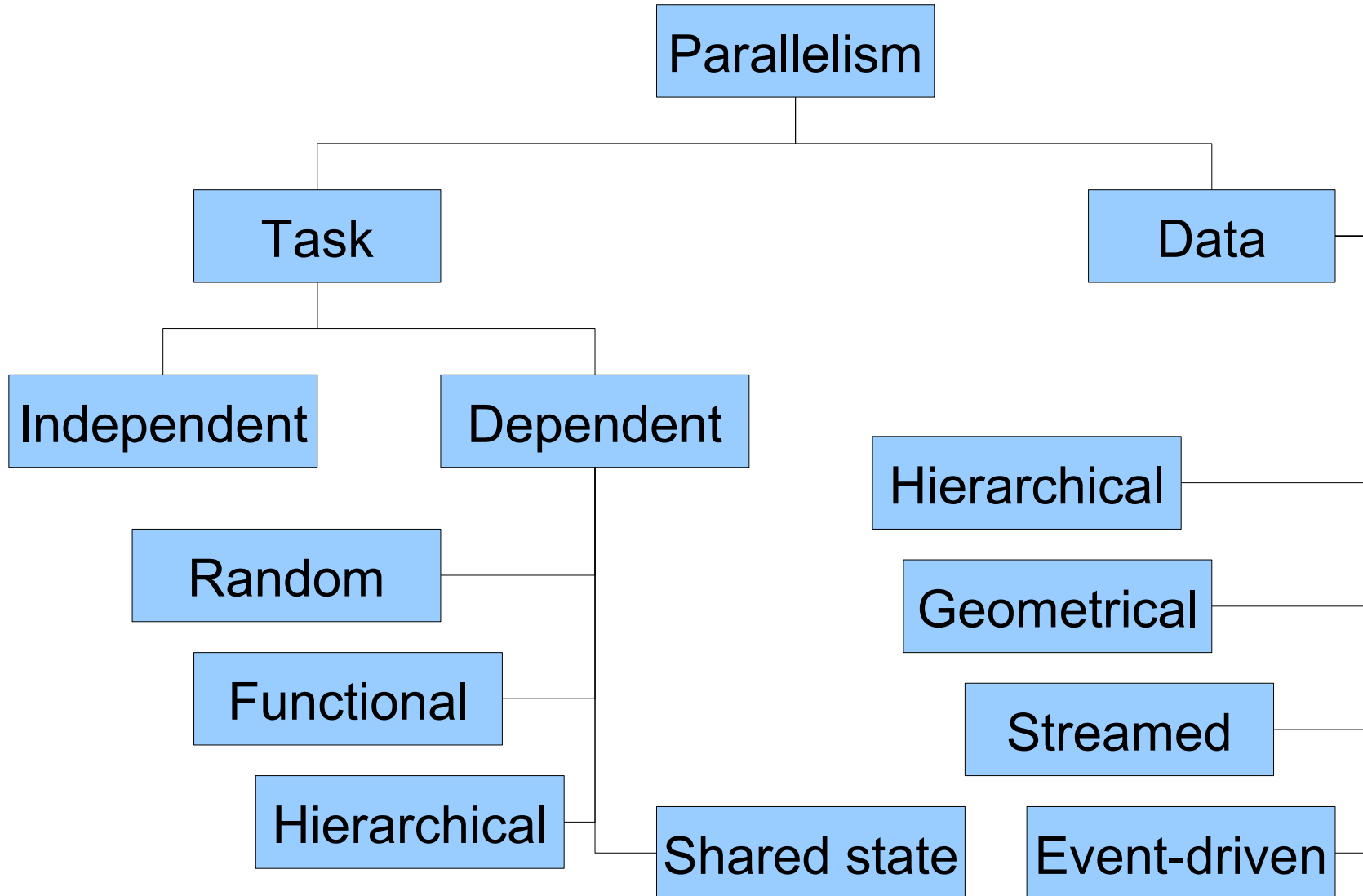


# Why high-level concurrency?

Multithreaded programs today work mostly by accident!

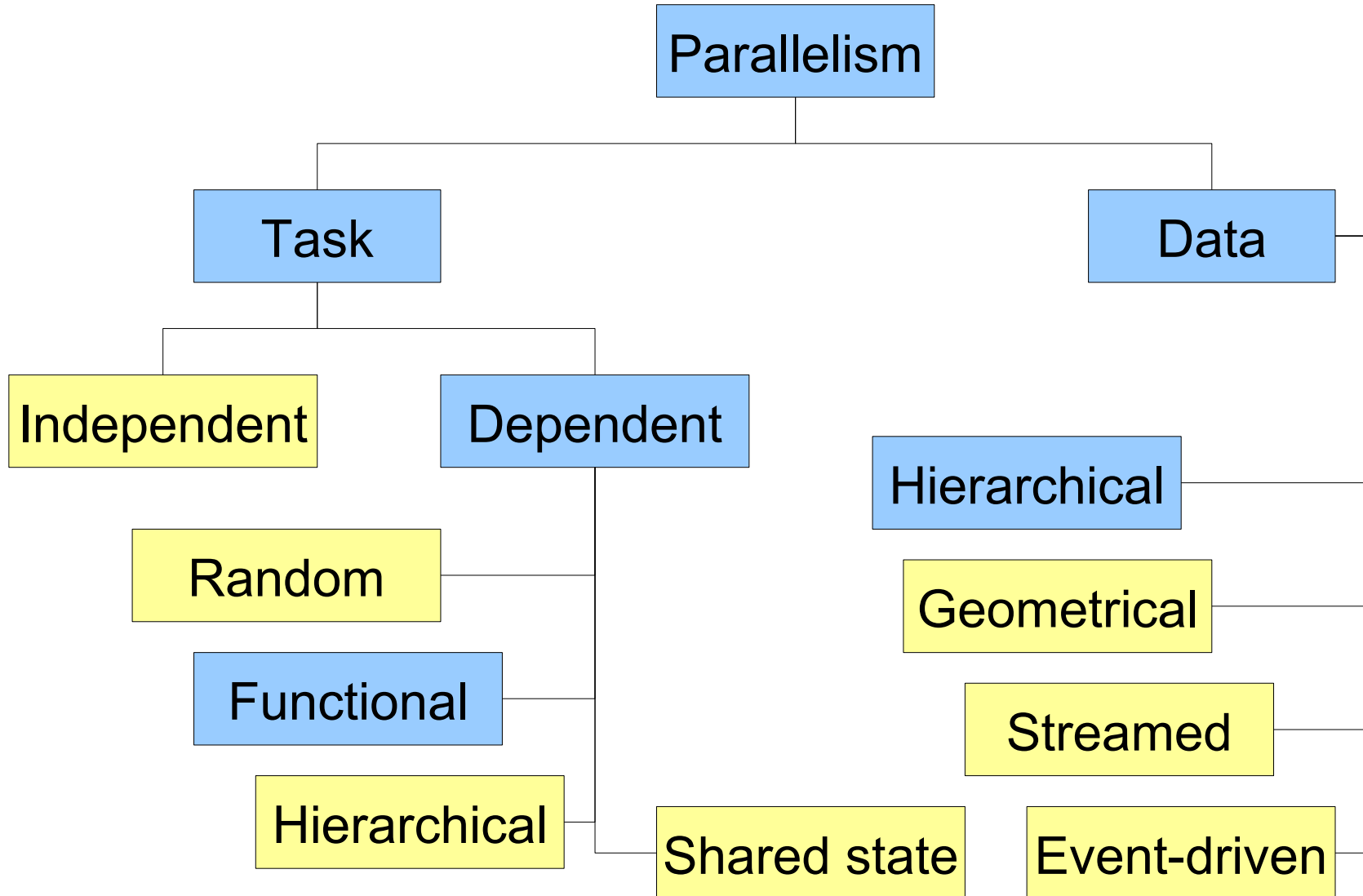


# Structure

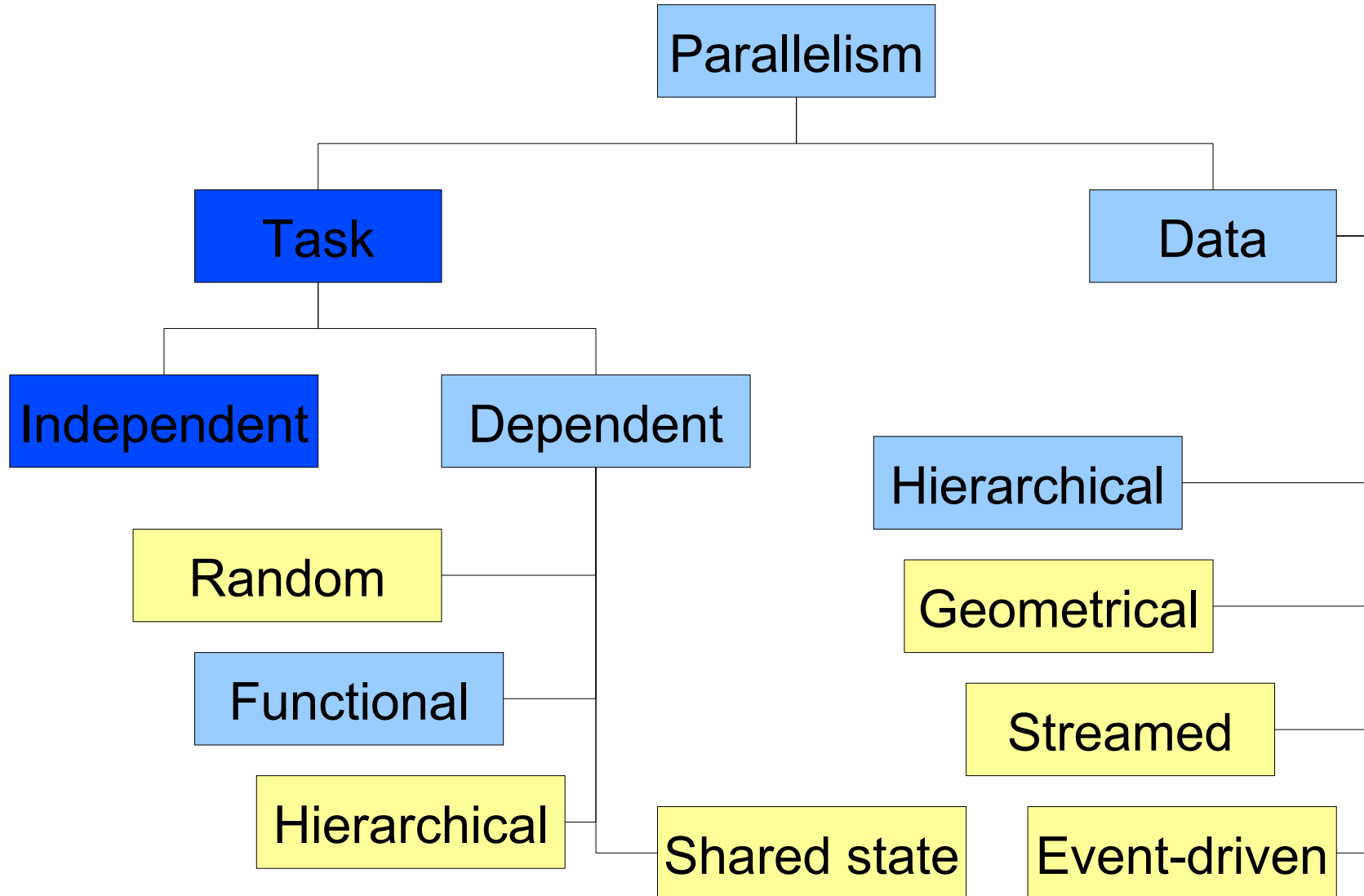




# Structure



# Task parallelism



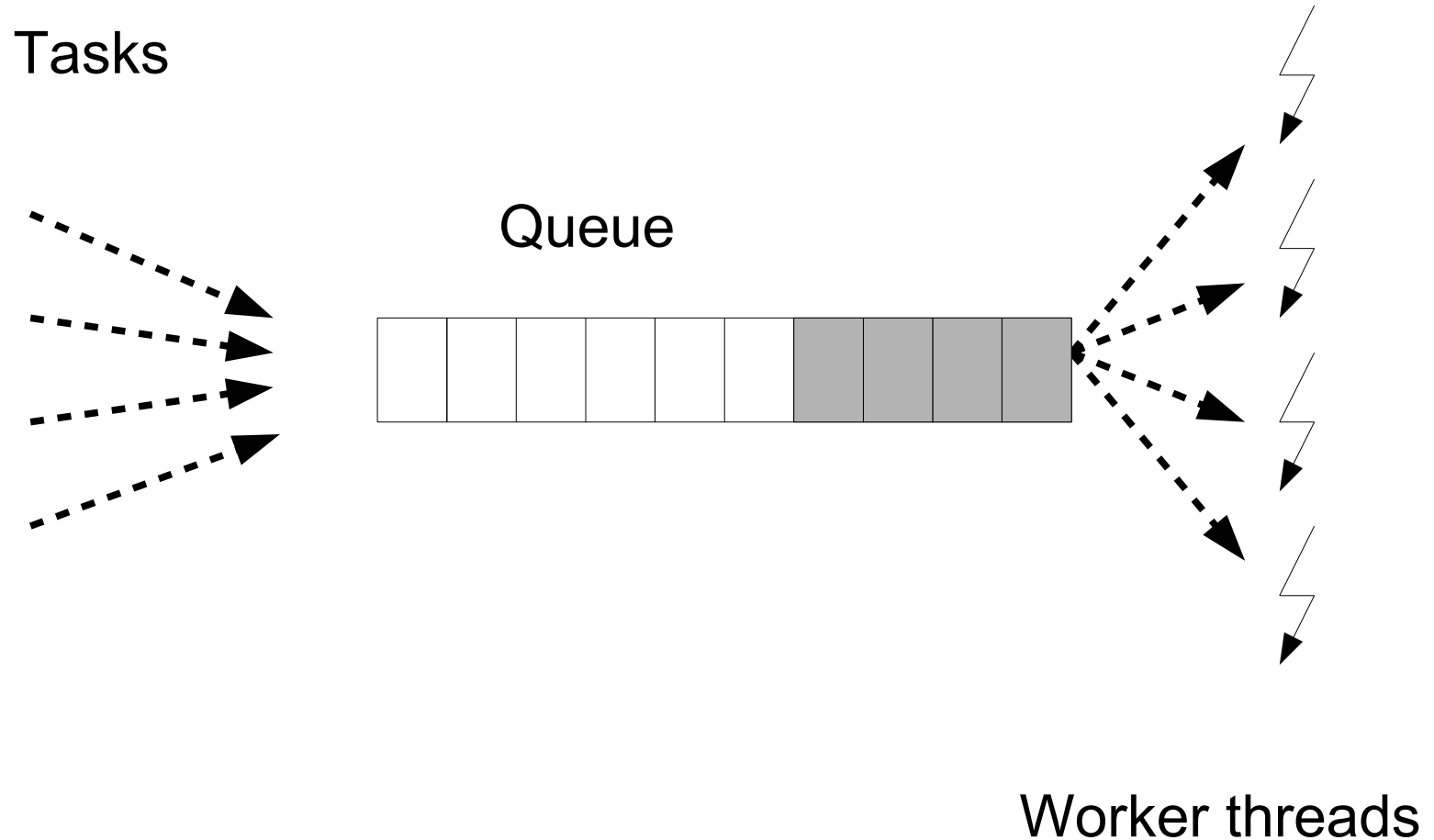
# Asynchronous invocation

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

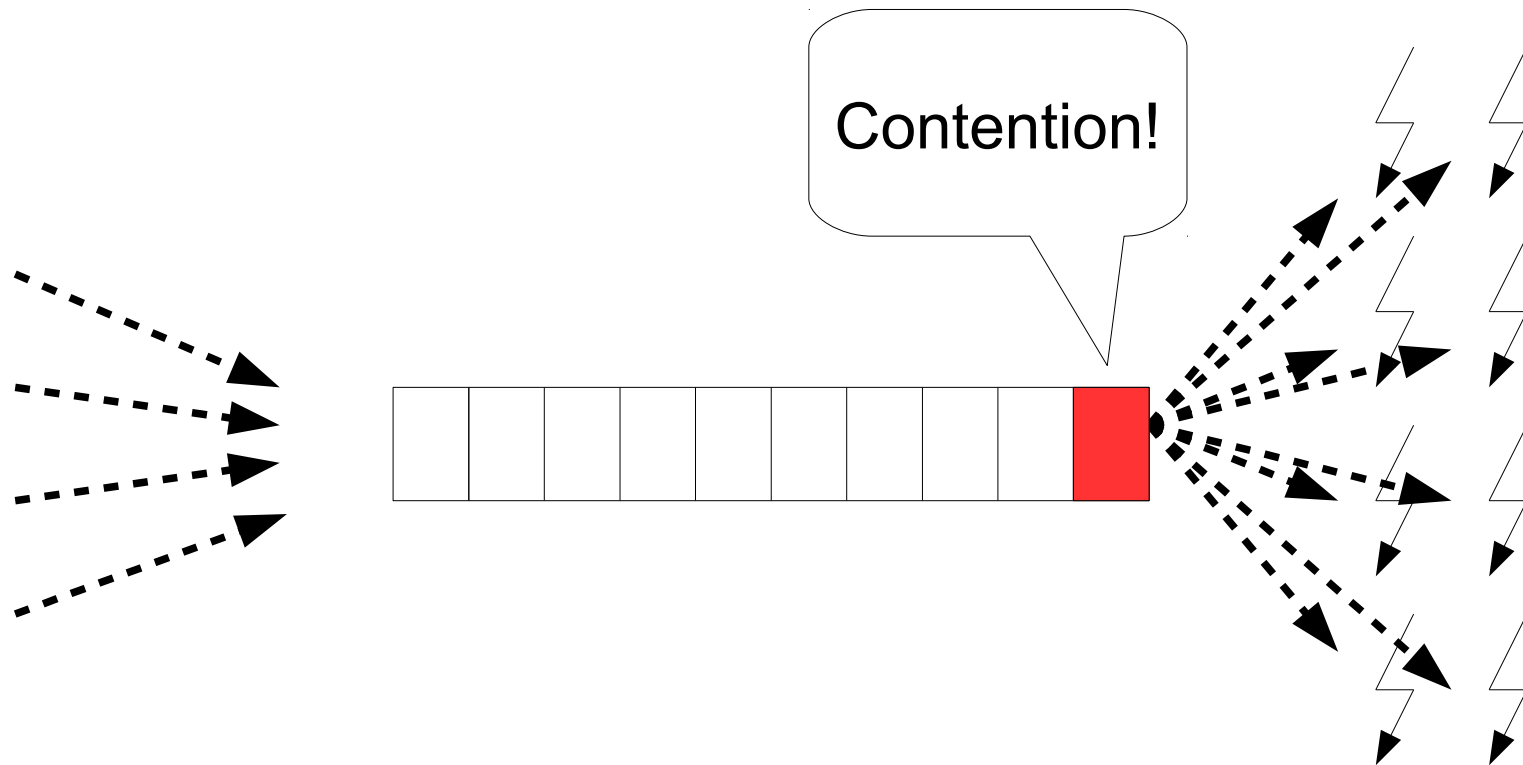
```
...
```

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

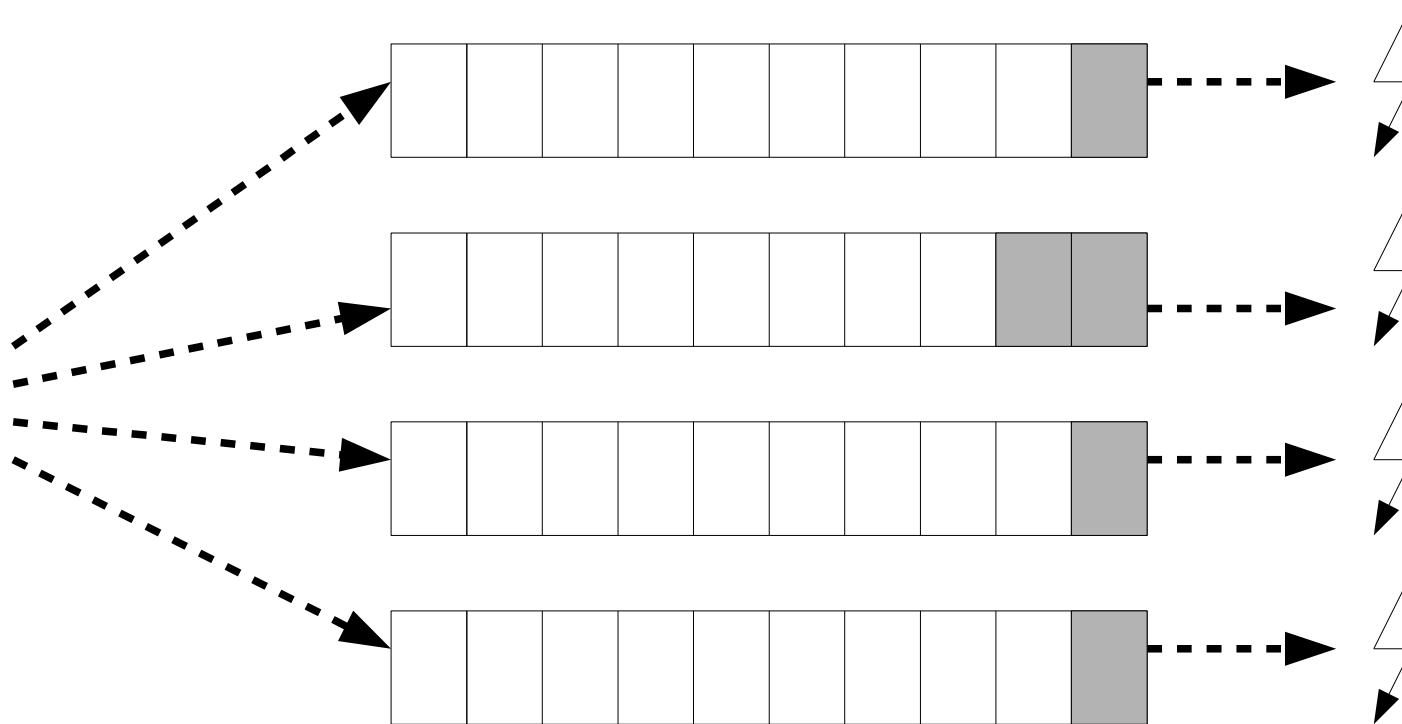
# Thread Pool



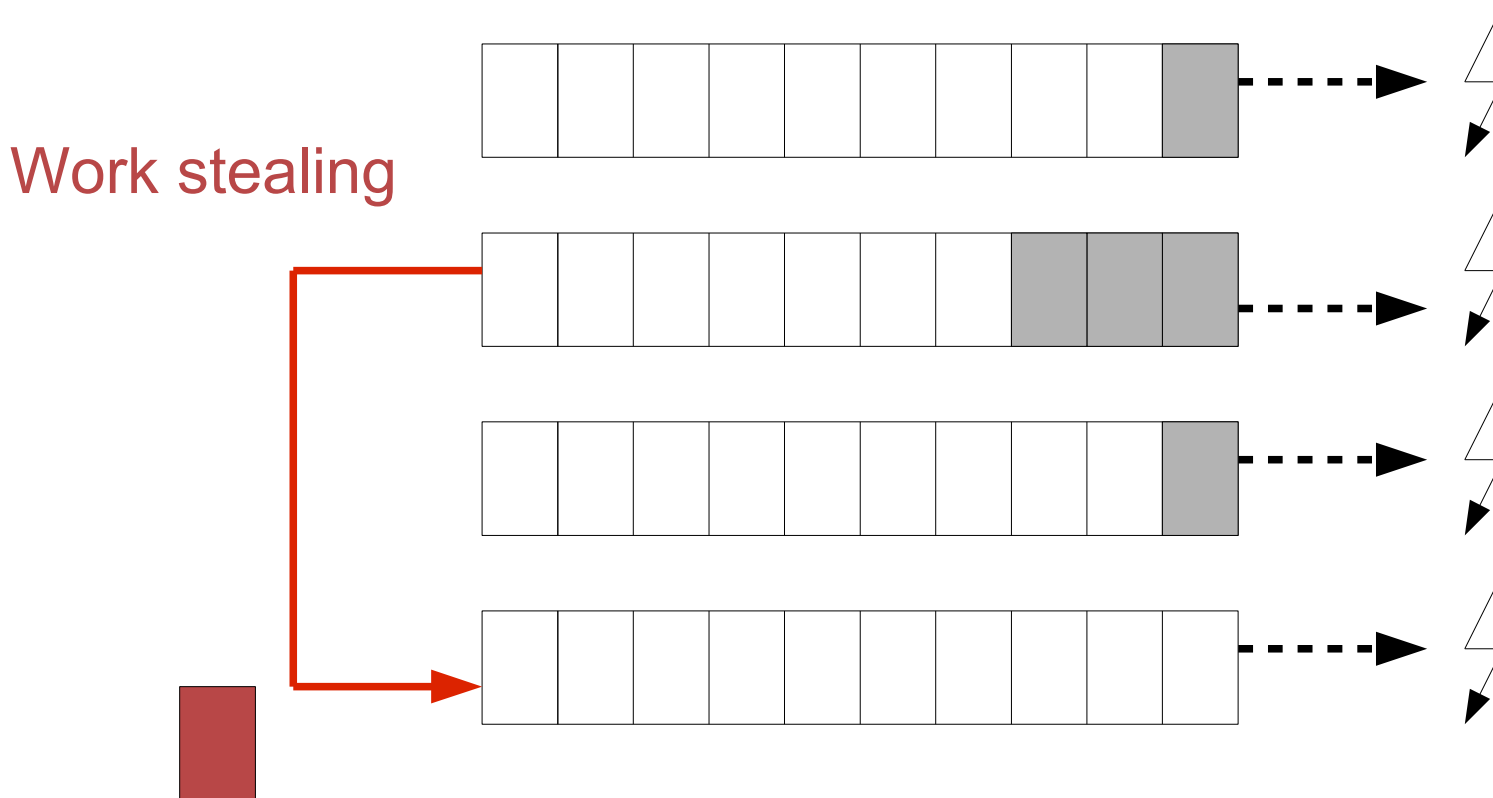
# Thread Pool



# Fork/Join Thread Pool



# Fork/Join Thread Pool



# Async the Groovy way

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





# Async the Groovy way

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

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



# Async the Groovy way

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

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

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



# Independent tasks

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

```
submissions.each {form →
```

```
    group.task {
```

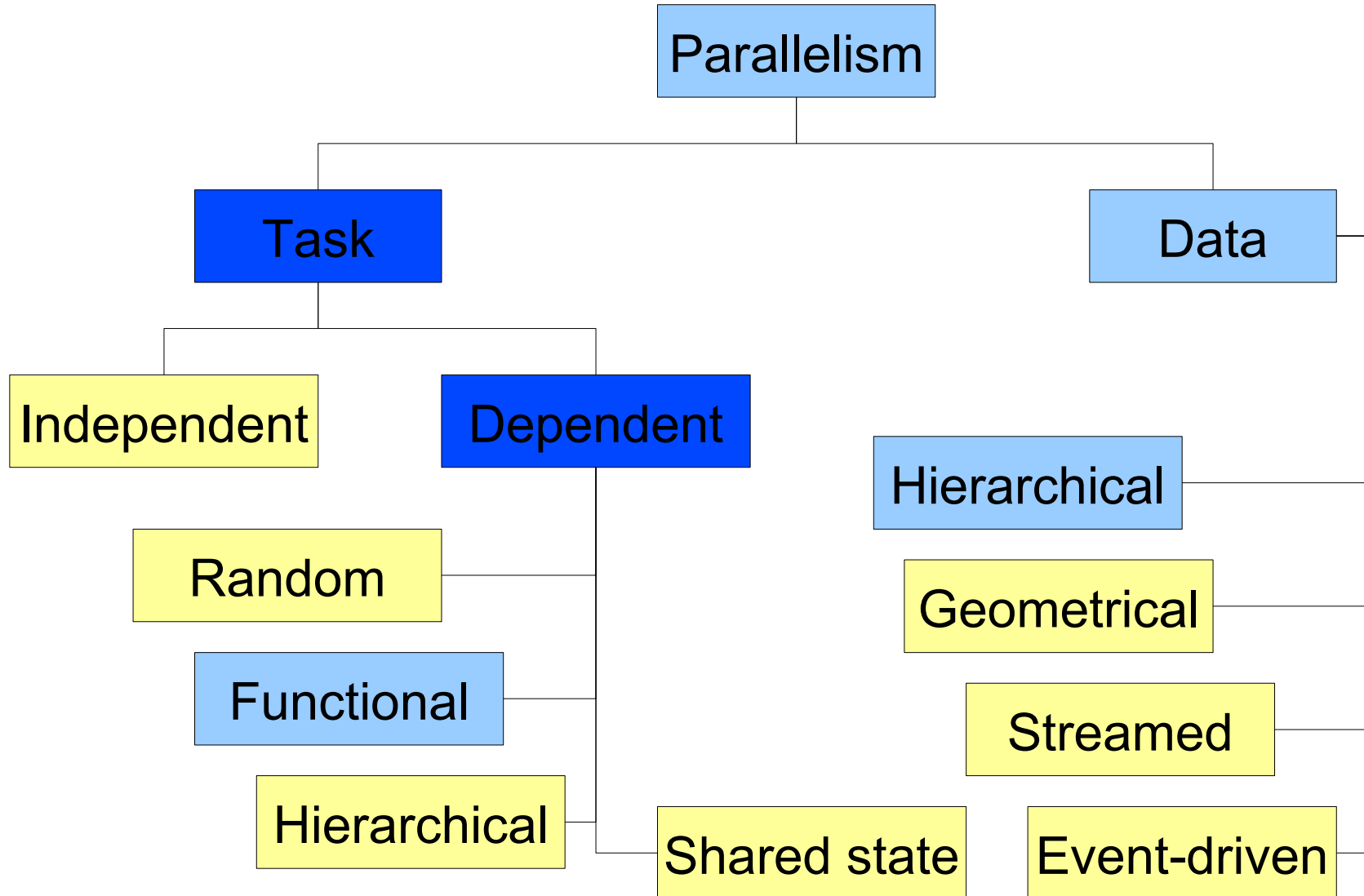
```
        form.process()
```

```
    }
```

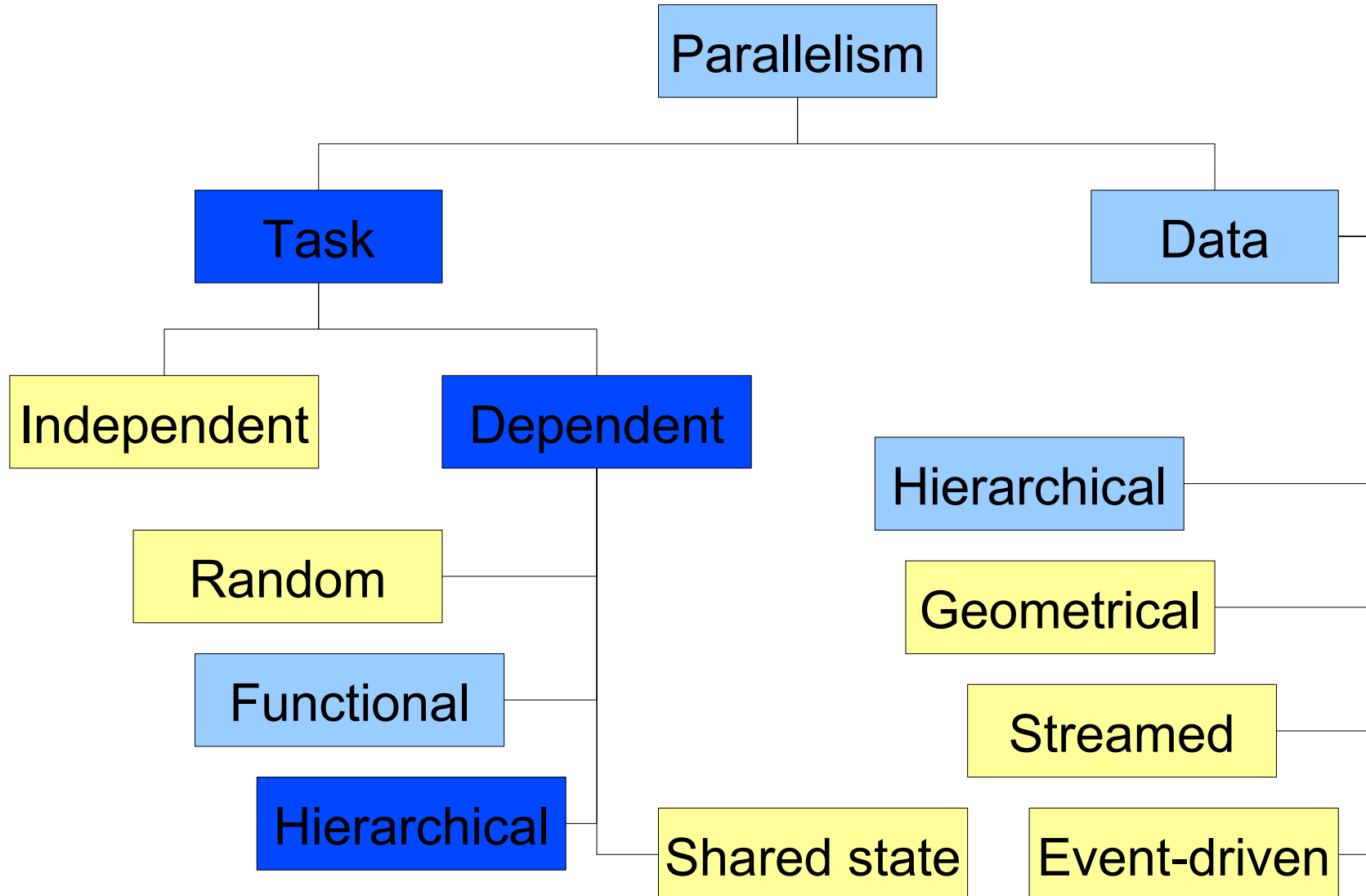
```
}
```



# Dependent tasks



# Dependent tasks

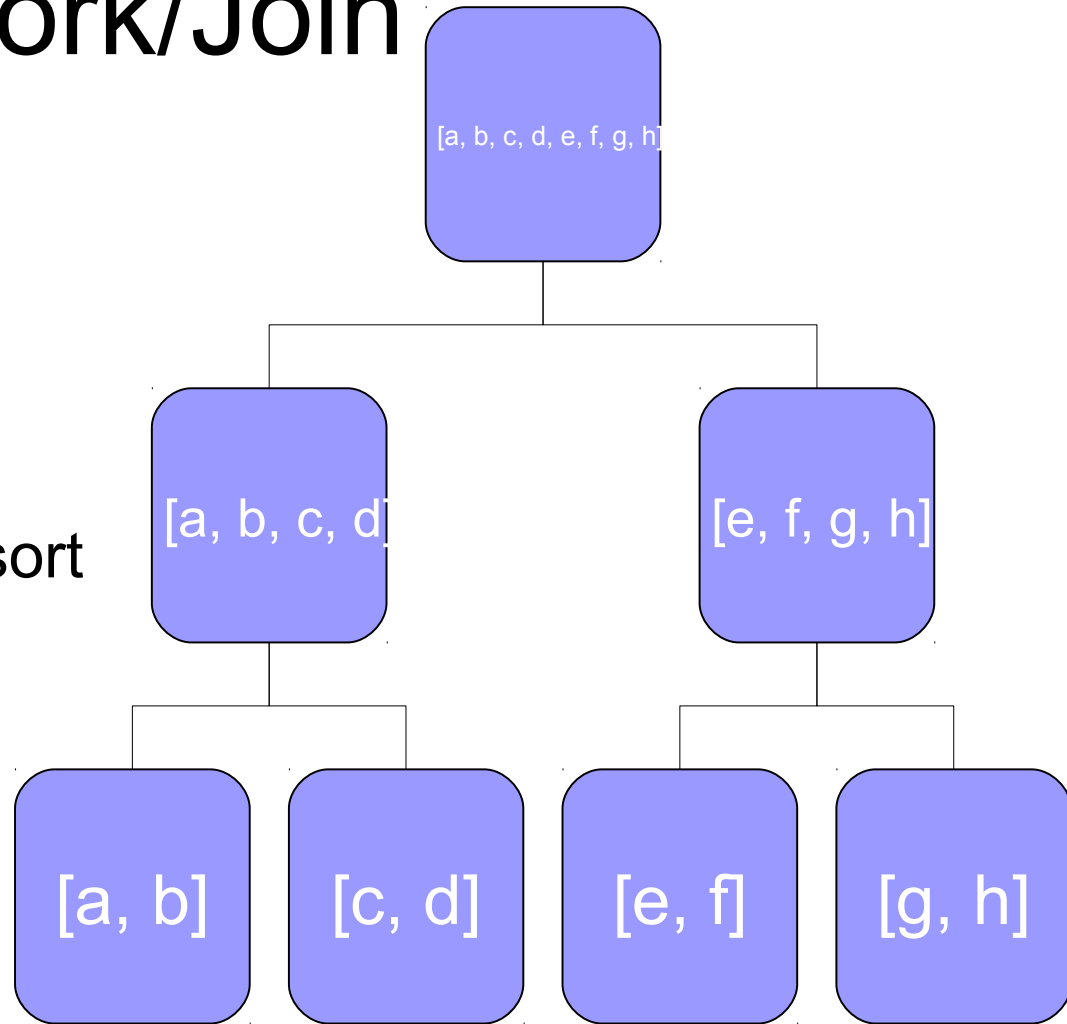


# Hierarchical decomposition

[64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]														
[64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33]										[32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18,				
[64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49]						[48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33]				[32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18,				
[64, 63, 62, 61, 60, 59, 58, 57]			[56, 55, 54, 53, 52, 51, 50, 49]			[48, 47, 46, 45, 44, 43, 42, 41]			[40, 39, 38, 37, 36, 35, 34, 33]					
			[56, 55, 54, 53]		[49, 50, 51, 52]					[40, 39, 38, 37]		[33, 34, 35, 36]		
[56, 55]				[51, 52]		[49, 50]					[40, 39]		[35, 36]	[33, 34]

# Fork/Join

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



# Fork/Join (GPars)

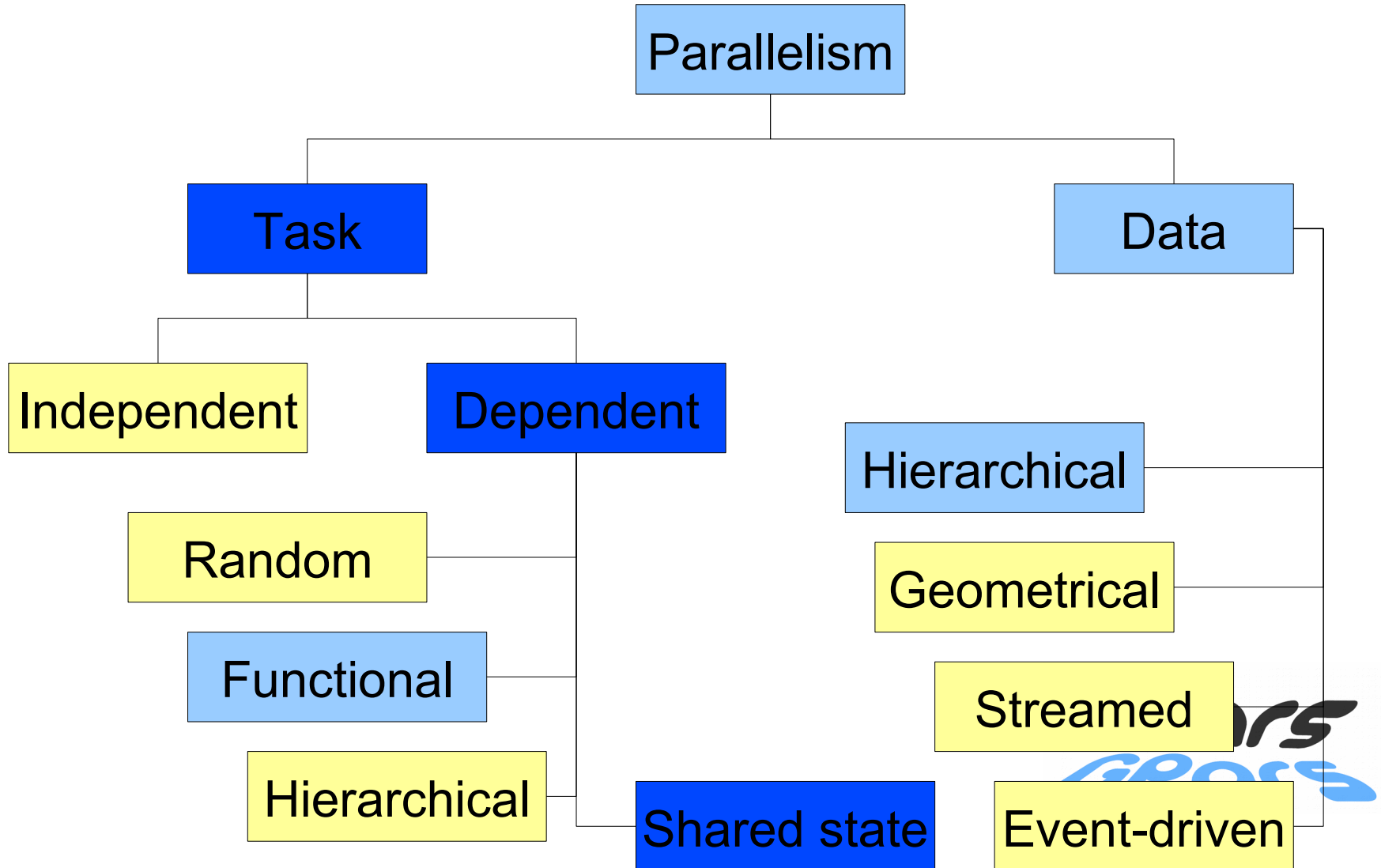
```
{currentDir ->
  long count = 0;
  currentDir.eachFile {
    if (it.isDirectory()) {
      forkOffChild it
    } else {
      count++
    }
  }
  return count + childrenResults.sum(0)
}
```

Waits for children  
without blocking the  
thread!





# State sharing




# State sharing

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

# State sharing

Needs protection

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



# Shared Mutable State

Frequently over- or mis-used

When really needed, use

- Locks
- Software Transactional Memory
- Agents

# 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

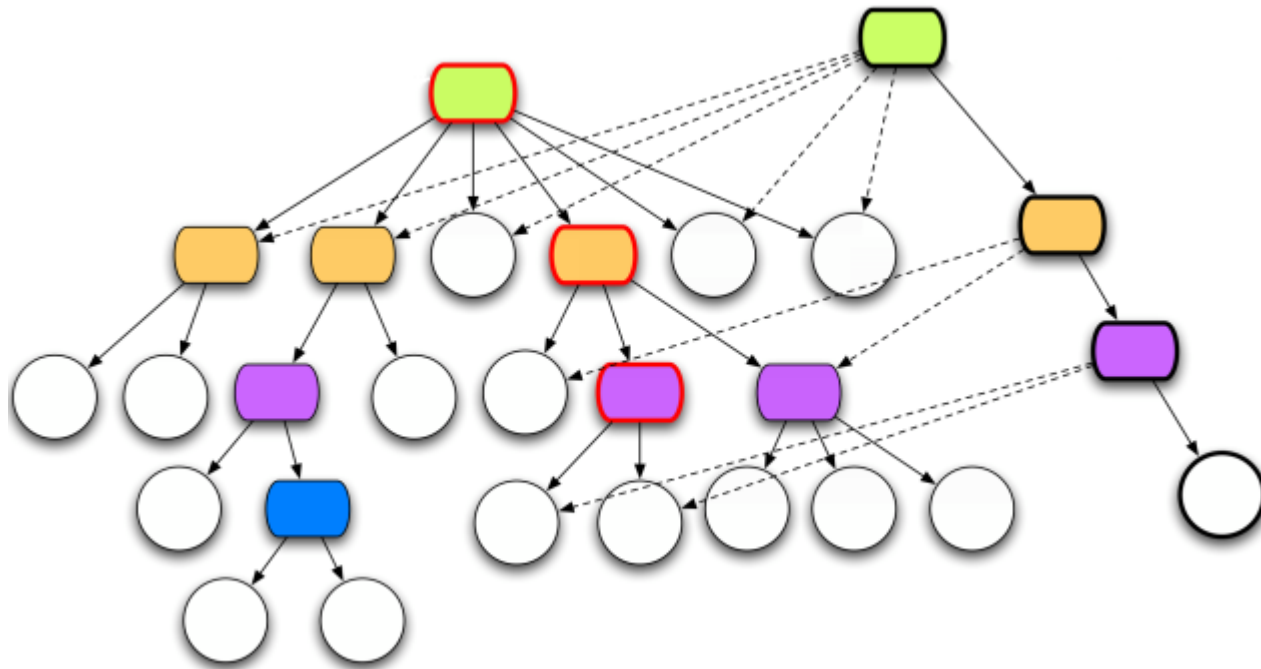
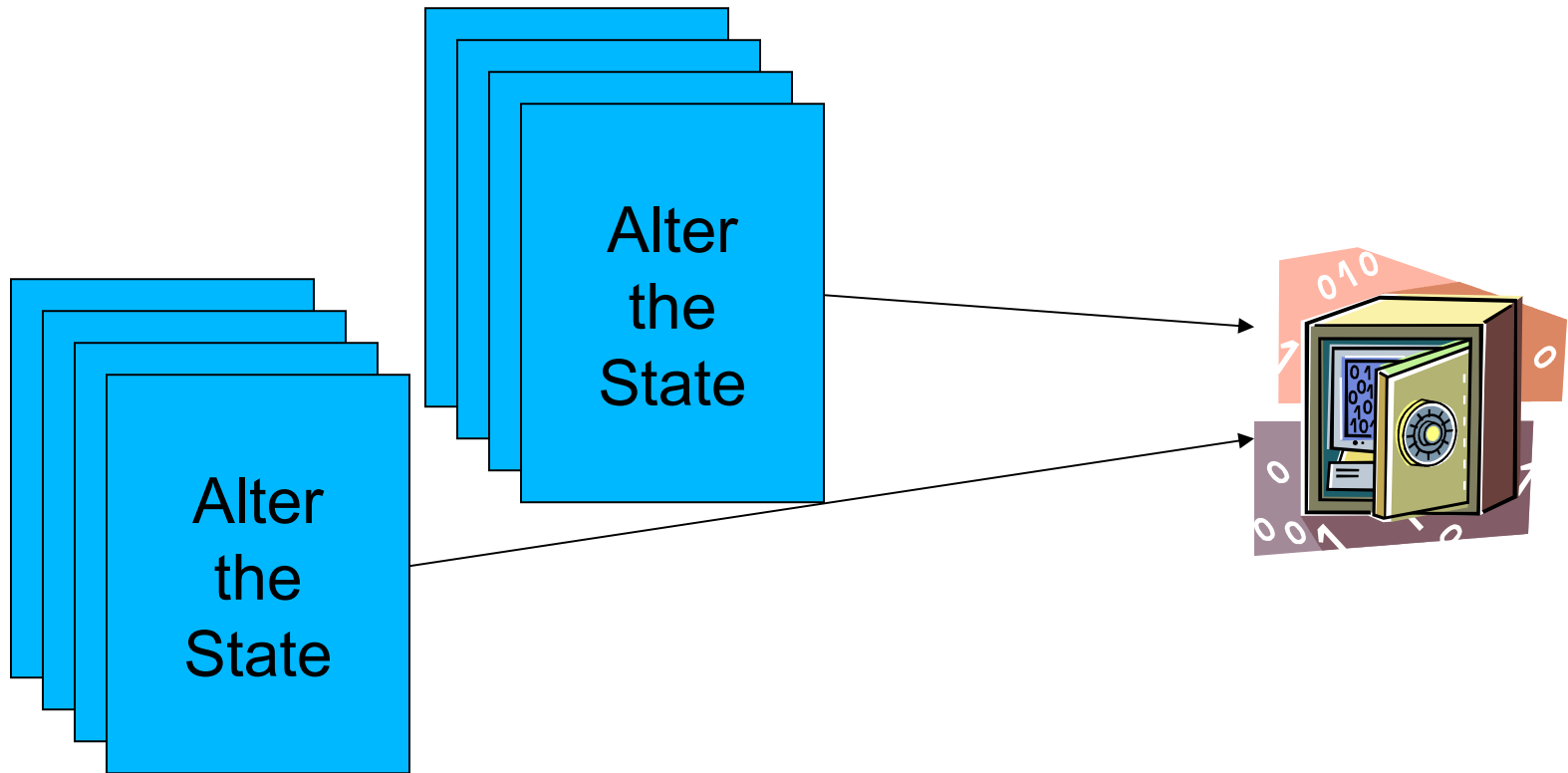


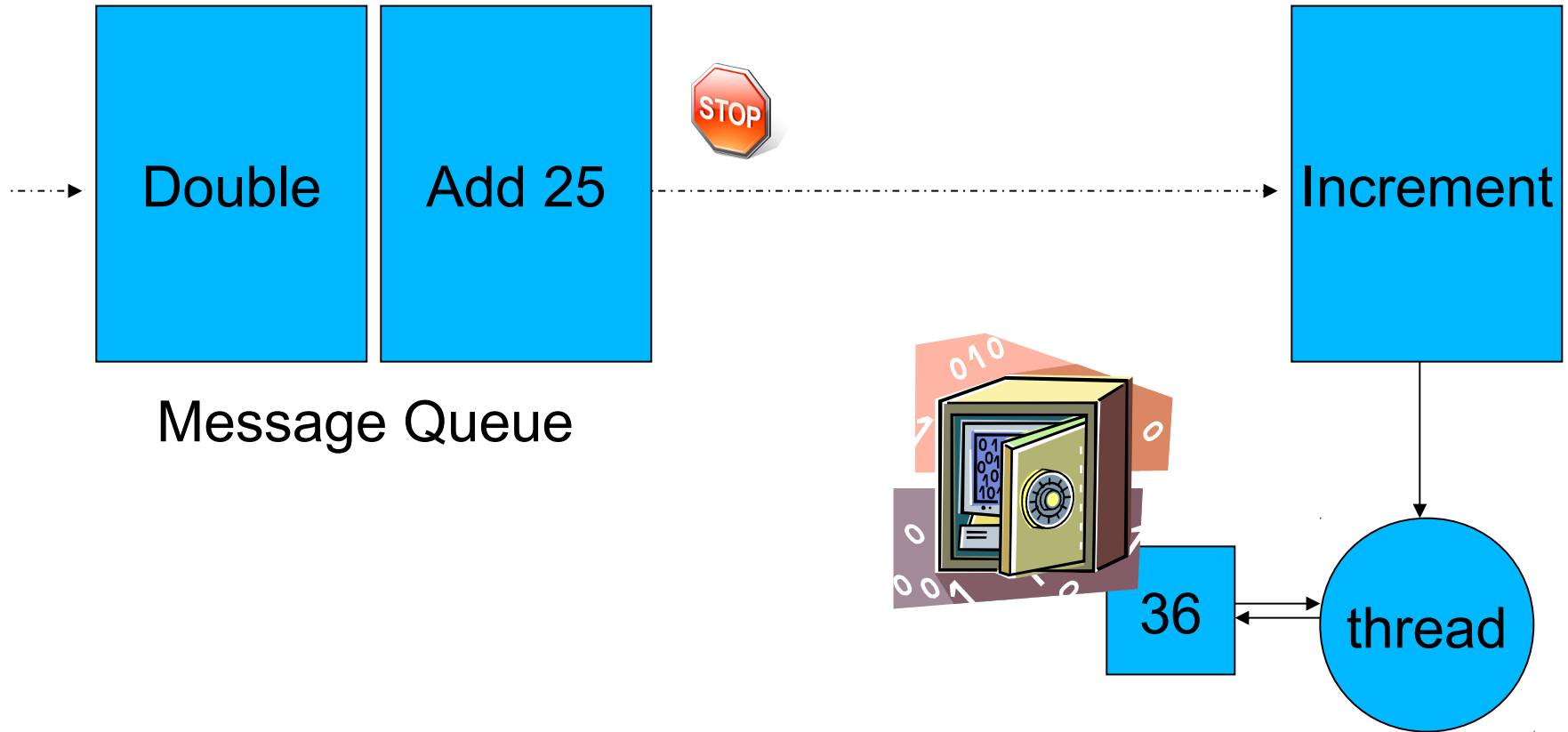
Illustration taken from Rich Hickey's presentation. Copyright Rich Hickey 2009

# Agent

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



# Agent inside





# Sharing through agents

```
Agent registrations = new Agent( [] )
```

```
submissions.each {form →
```

```
  task {
```

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

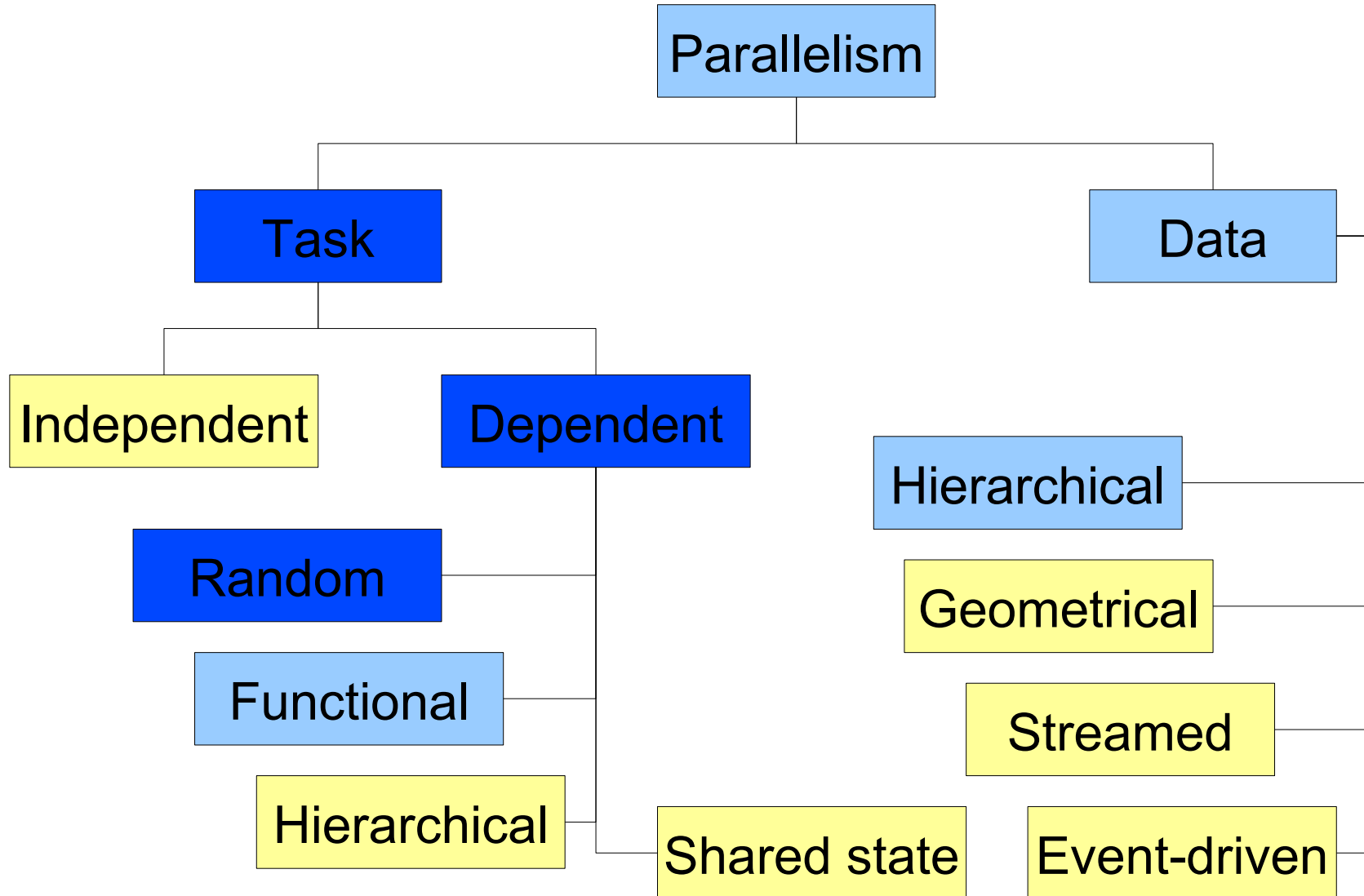
```
      registrations.send {it << form}
```

```
    }
```

```
  }
```

```
}
```

# Random task dependency

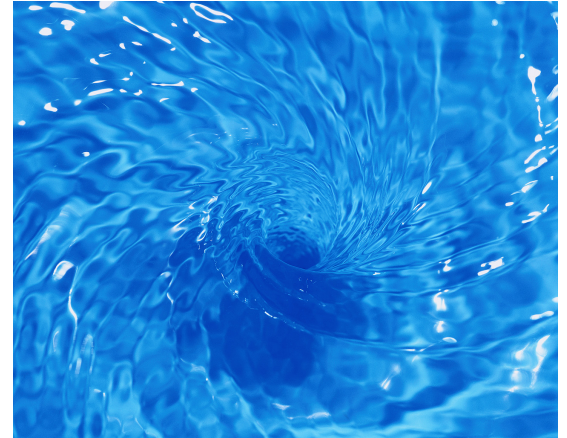


# Dataflow Concurrency

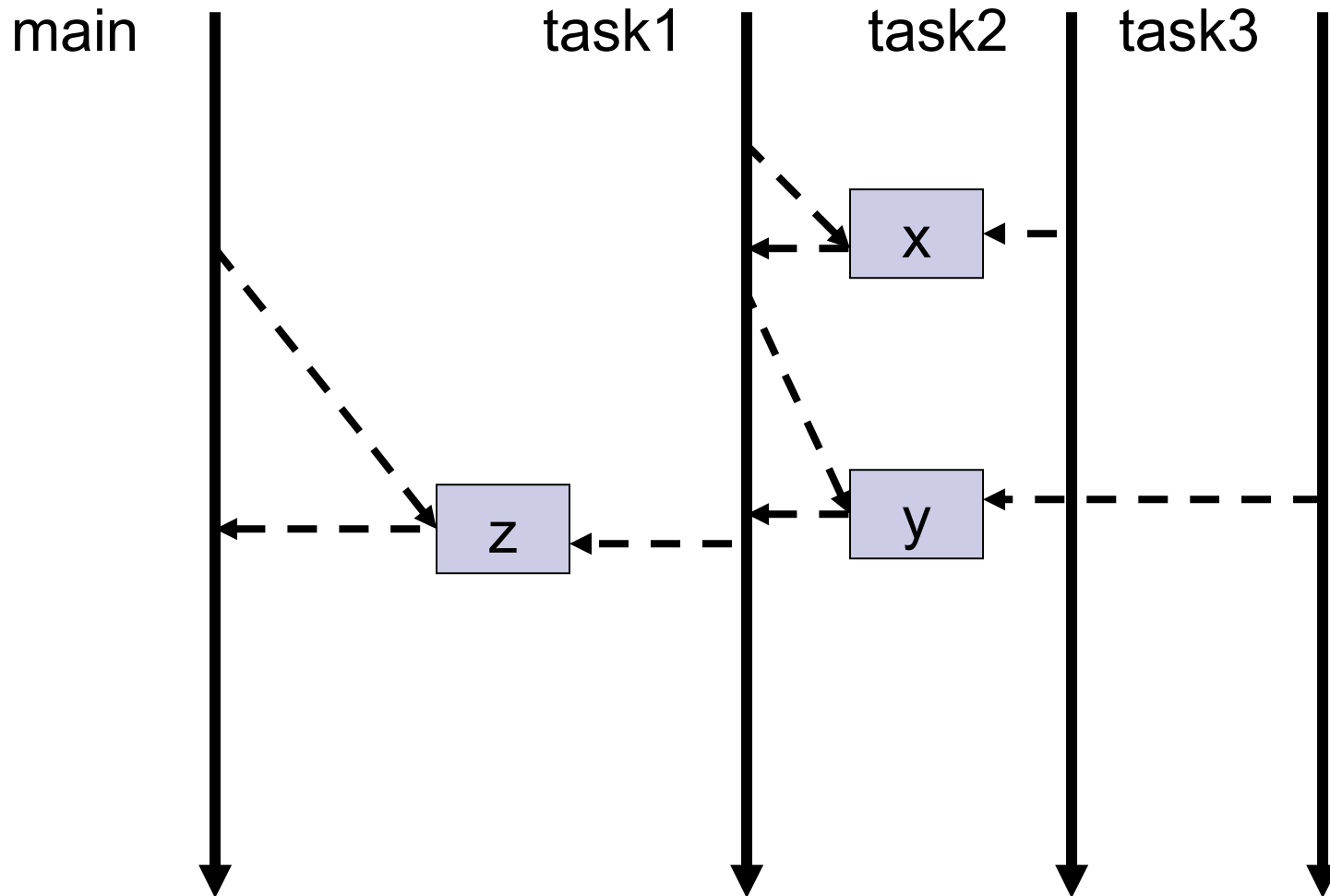
No race-conditions

No live-locks

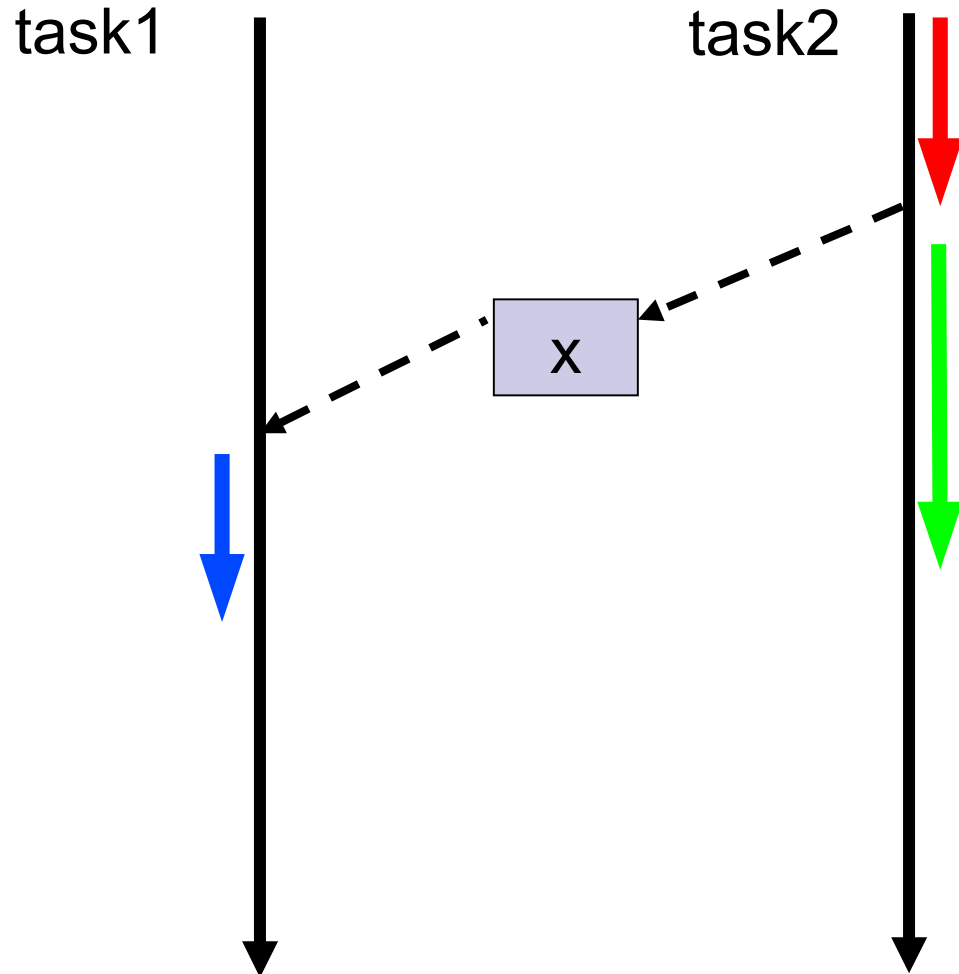
Deterministic deadlocks



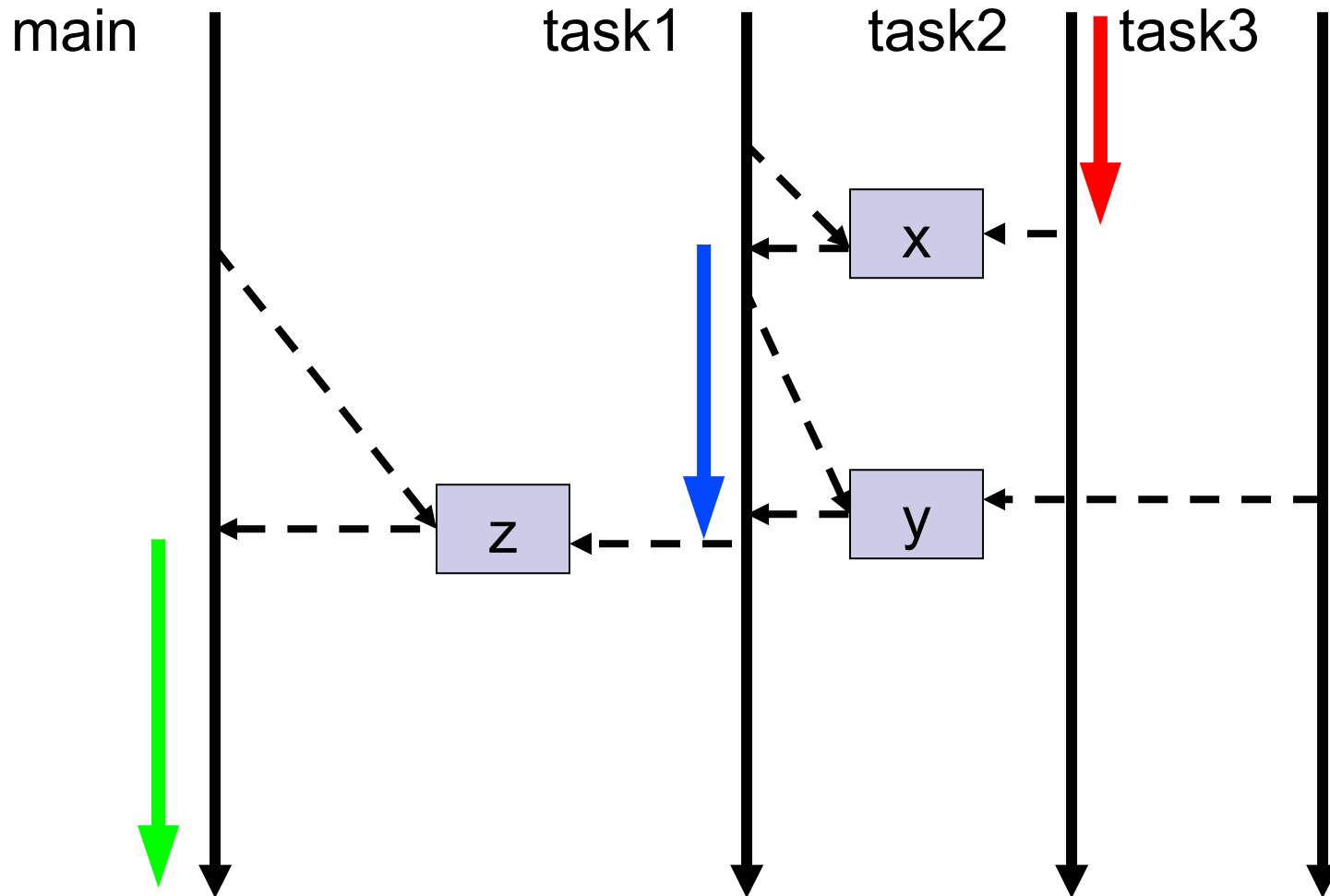
# Dataflow Variables / Promises



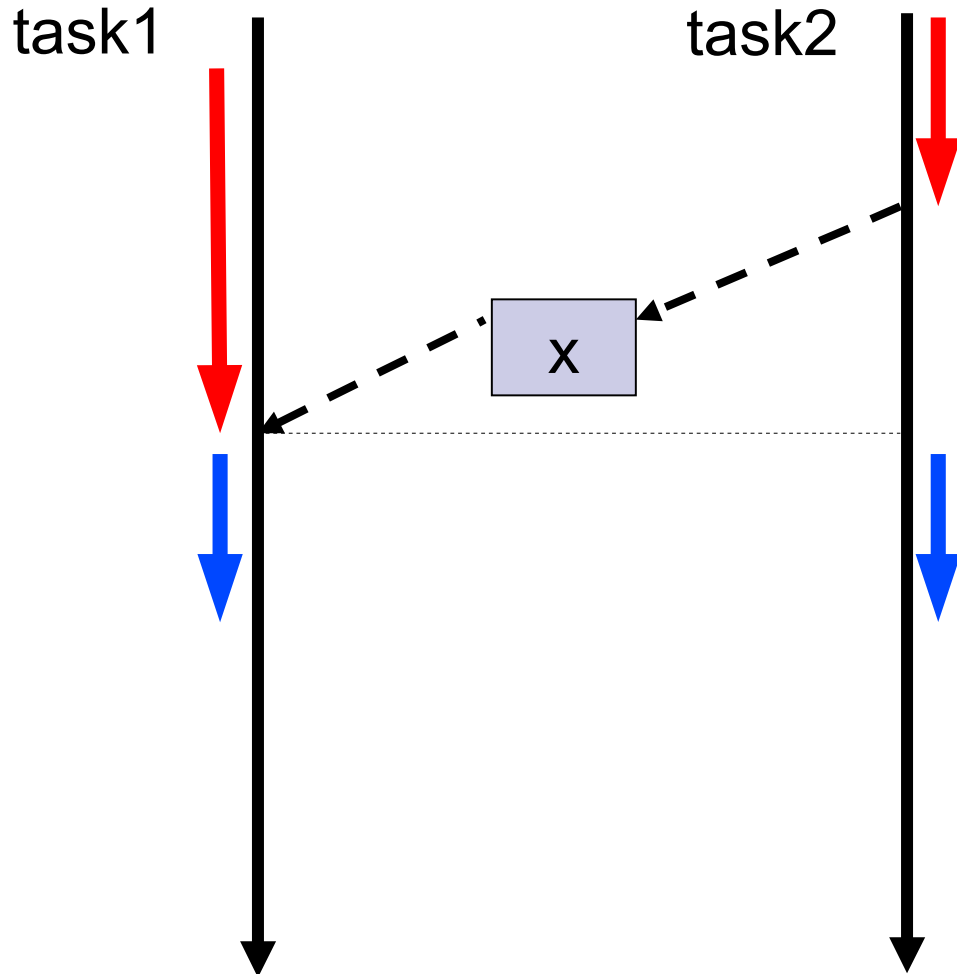
# Dataflow Variables / Promises



# Dataflow Variables / Promises



# Synchronous Variables



# Promises to exchange data

```
task { z << x.val + y.val }
```

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

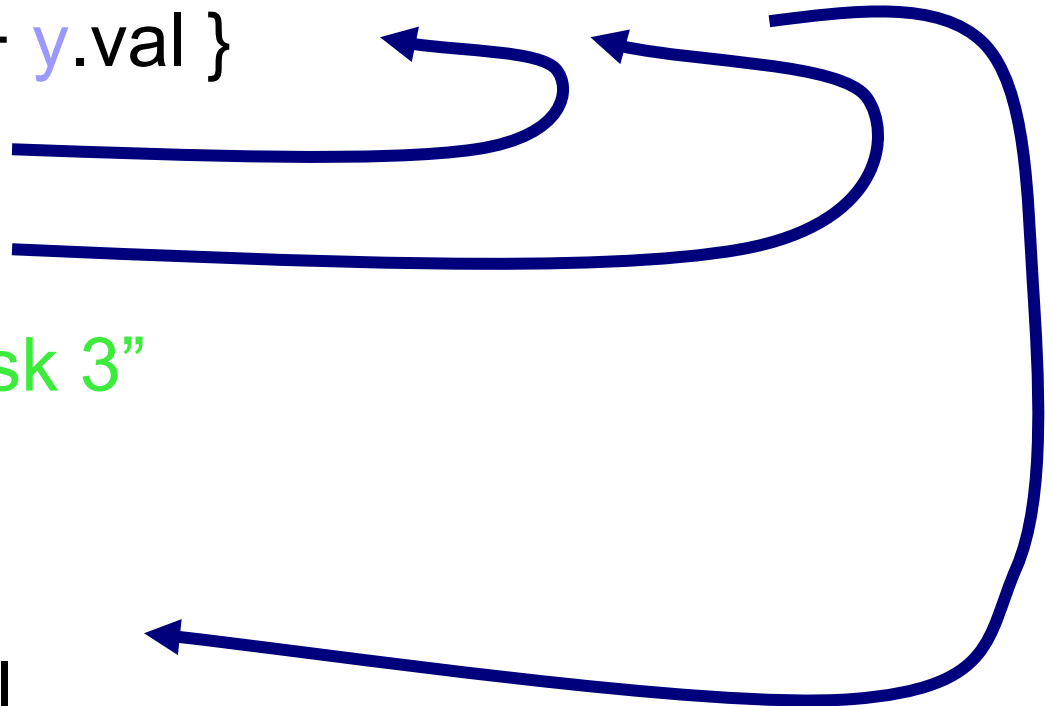
```
task {
```

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

```
    y << 5
```

```
}
```

```
assert 15 == z.val
```





# 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)
```

# 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);
```

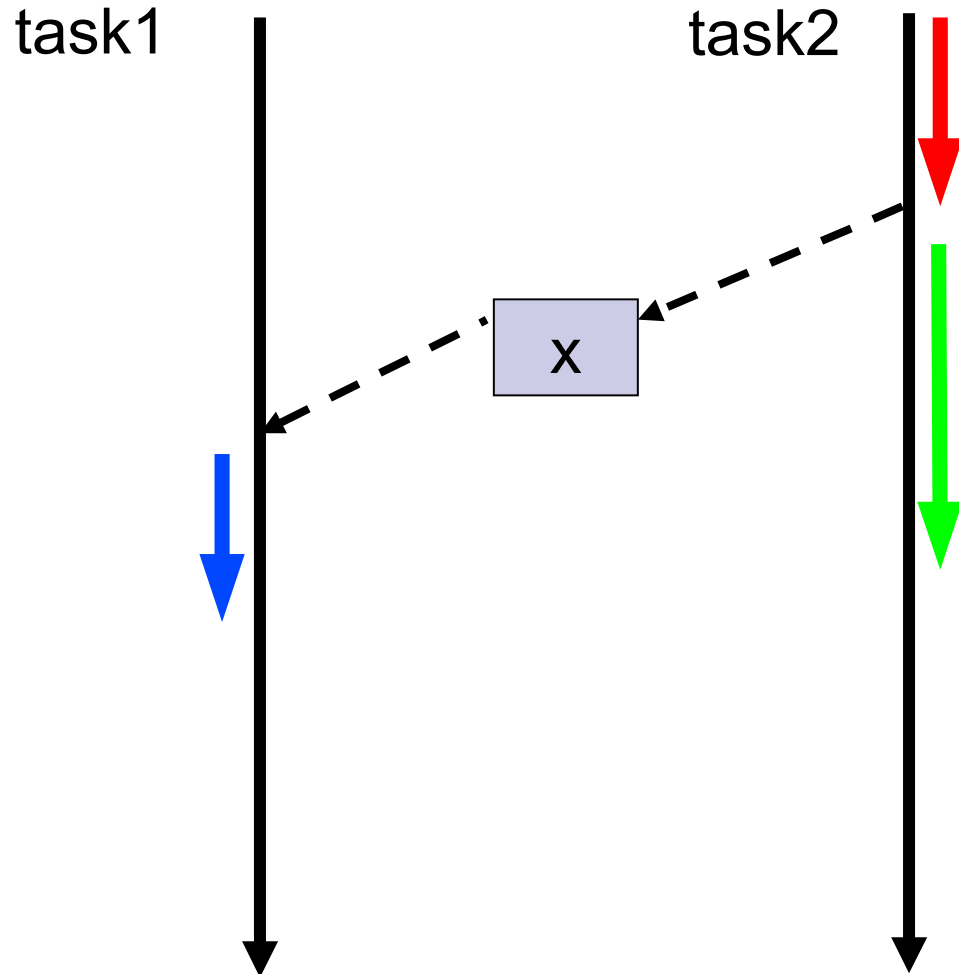
# Lazy promises

Only calculated when needed the first time

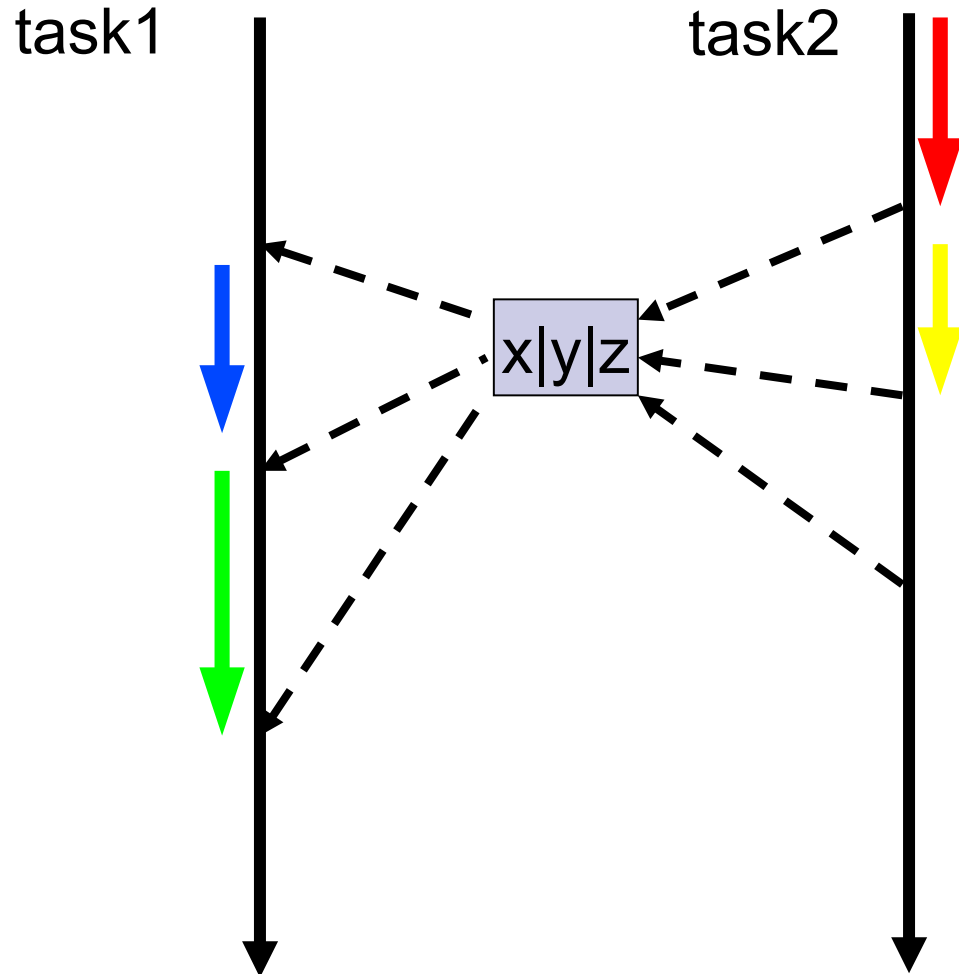
```
def mostPopularLang = new LazyDataflowVariable({->  
    return longLastingCalculation()  
})
```



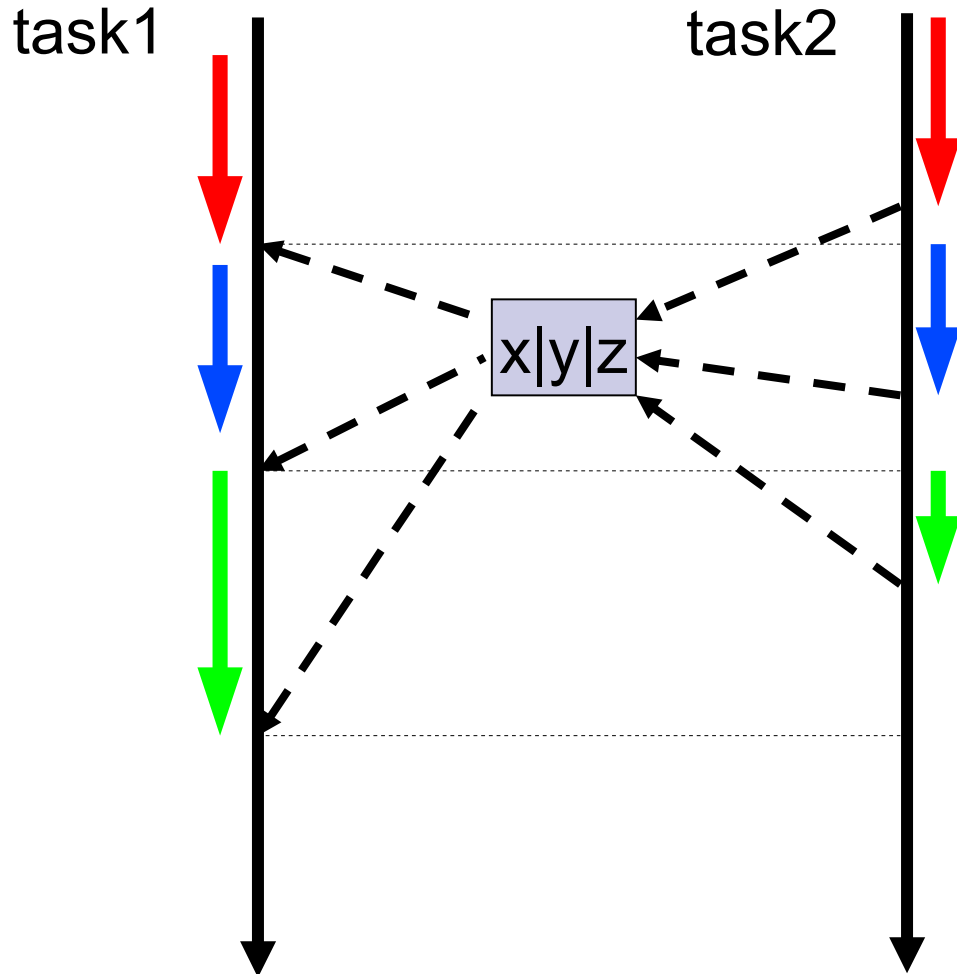
# Dataflow Variables / Promises



# Dataflow Channels



# Synchronous Channels



# Async progress indication

```
List<Promise> forms=submissions.collect {form →  
  group.task {  
    def result = form.process()  
    progressQueue << 1  
    if (result.valid) {  
      return form  
    }  
  }  
}
```

# Async result reporting

submissions.each {form →

  group.task {

    if (form.process().valid) queue << form

  }

}

# 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: ...  
}
```

# Tasks as processes

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

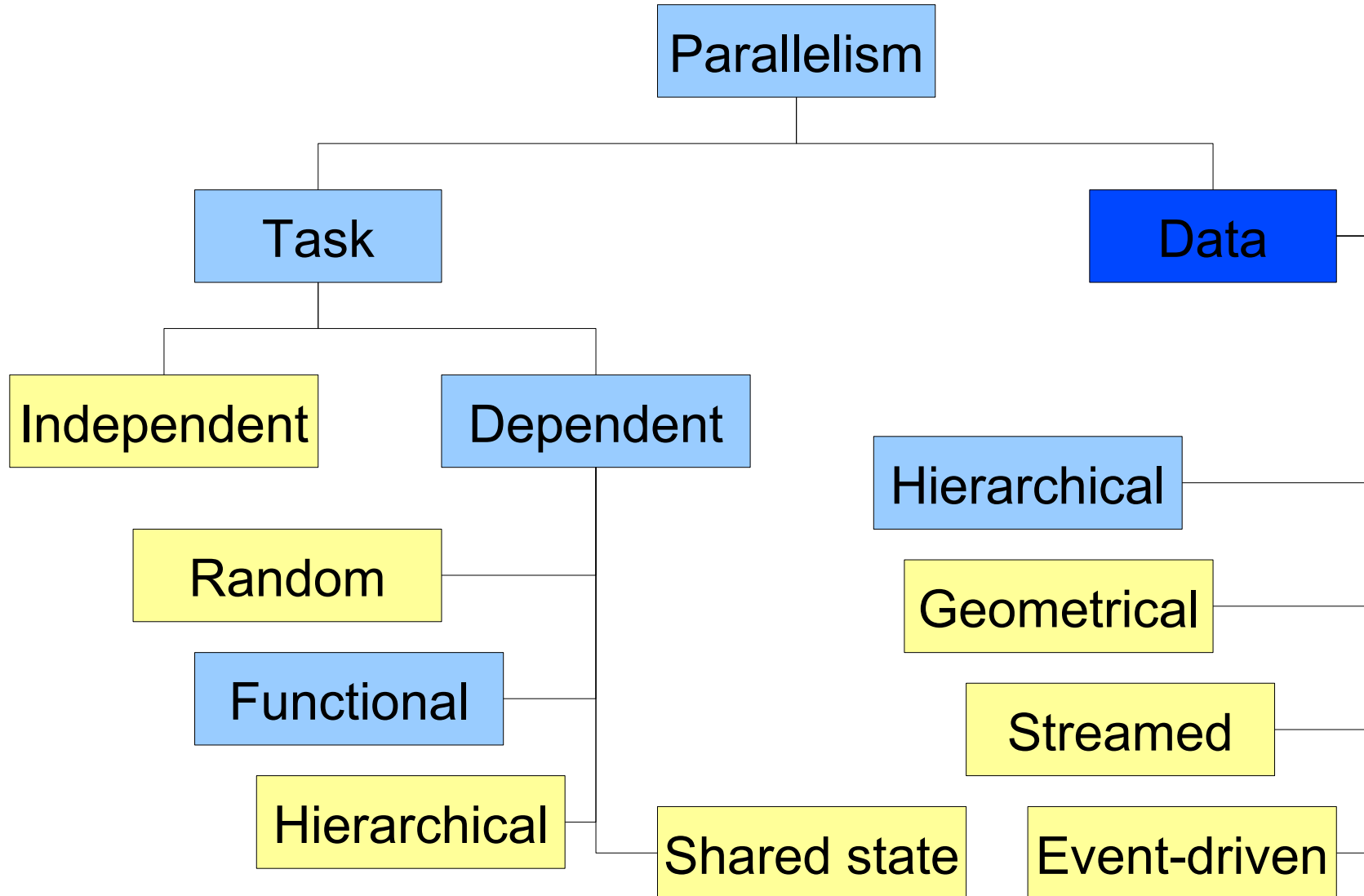
# A CSP flavour

## Communicating Sequential Processes

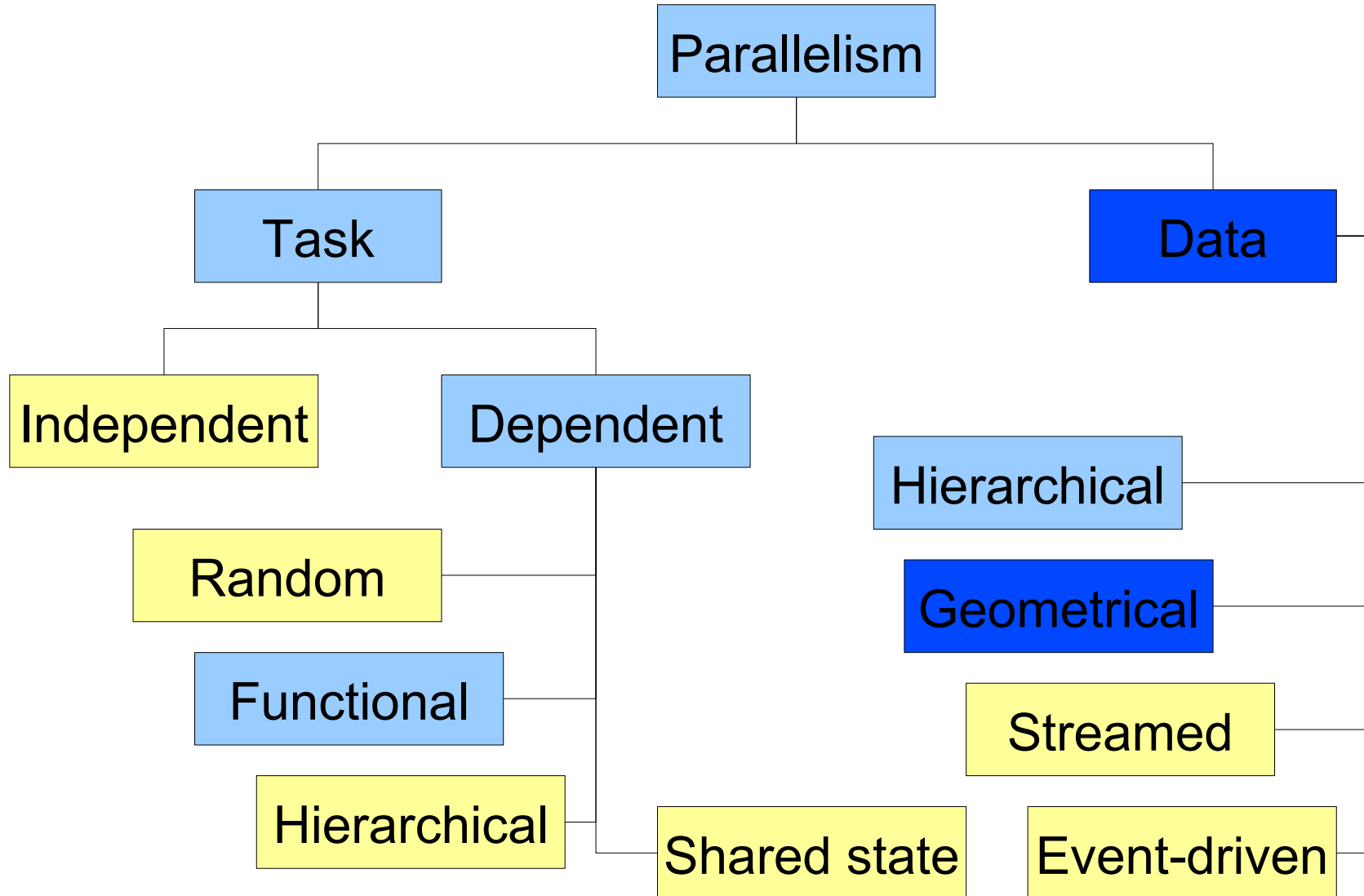
Focus on composable processes more than on data



# Data parallelism



# Data parallelism



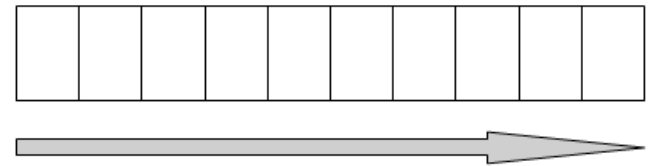
# Geometric decomposition

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

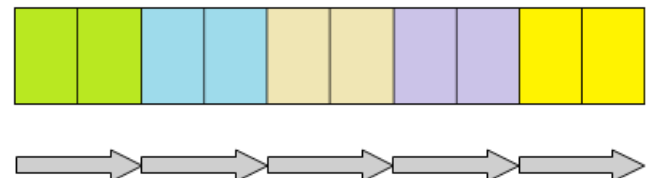
documents.`sumParallel`()

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

1 thread



5 threads



# Geometric decomposition

```
registrations = submissions
```

```
  .collectParallel { form -> form.process() }
```

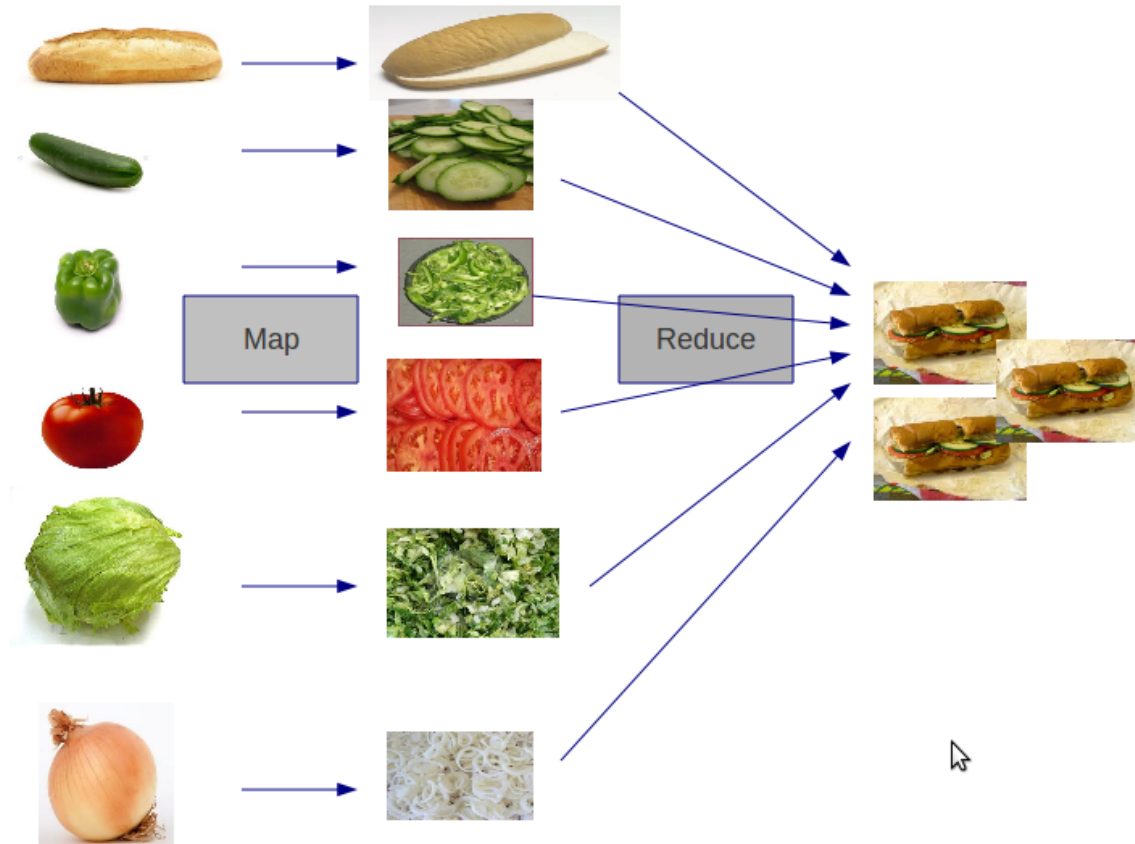
```
  .findAllParallel { it.valid }
```

```
registrations = submissions.parallel
```

```
  .map { form -> form.process() }
```

```
  .filter { it.valid }.collection
```

# Map - reduce



# Frequent confusion

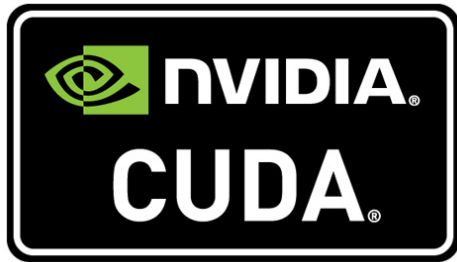
[Questions](#)[Tags](#)[Users](#)[Badges](#)[Unanswered](#)

## 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

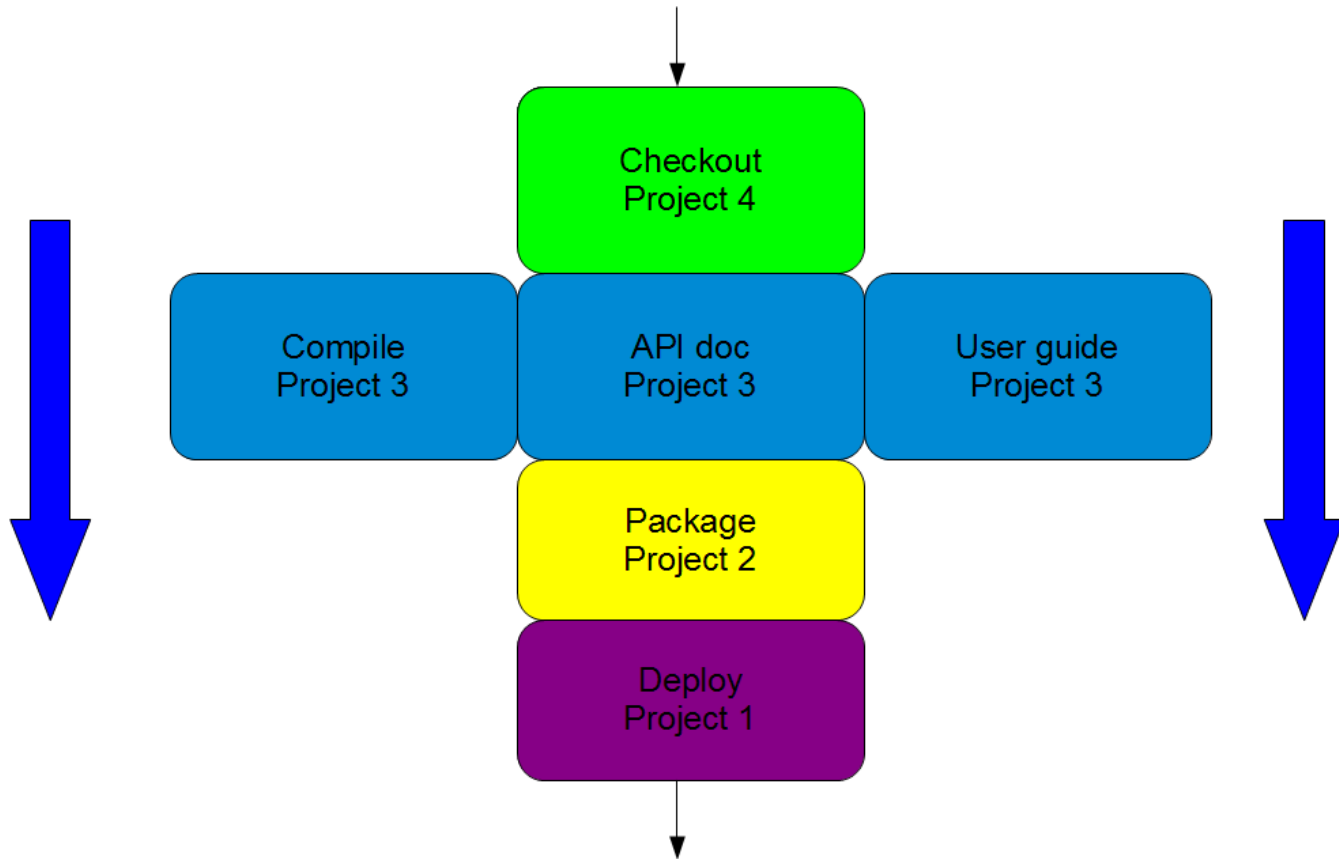
```
new File("/file.txt").withReader{reader ->
  reader.eachParallel {
    def r1 = step1(r)
    def r2 = step2(r1)
    def r3 = step3(r2)
  }
}
```

# Unroll iteration

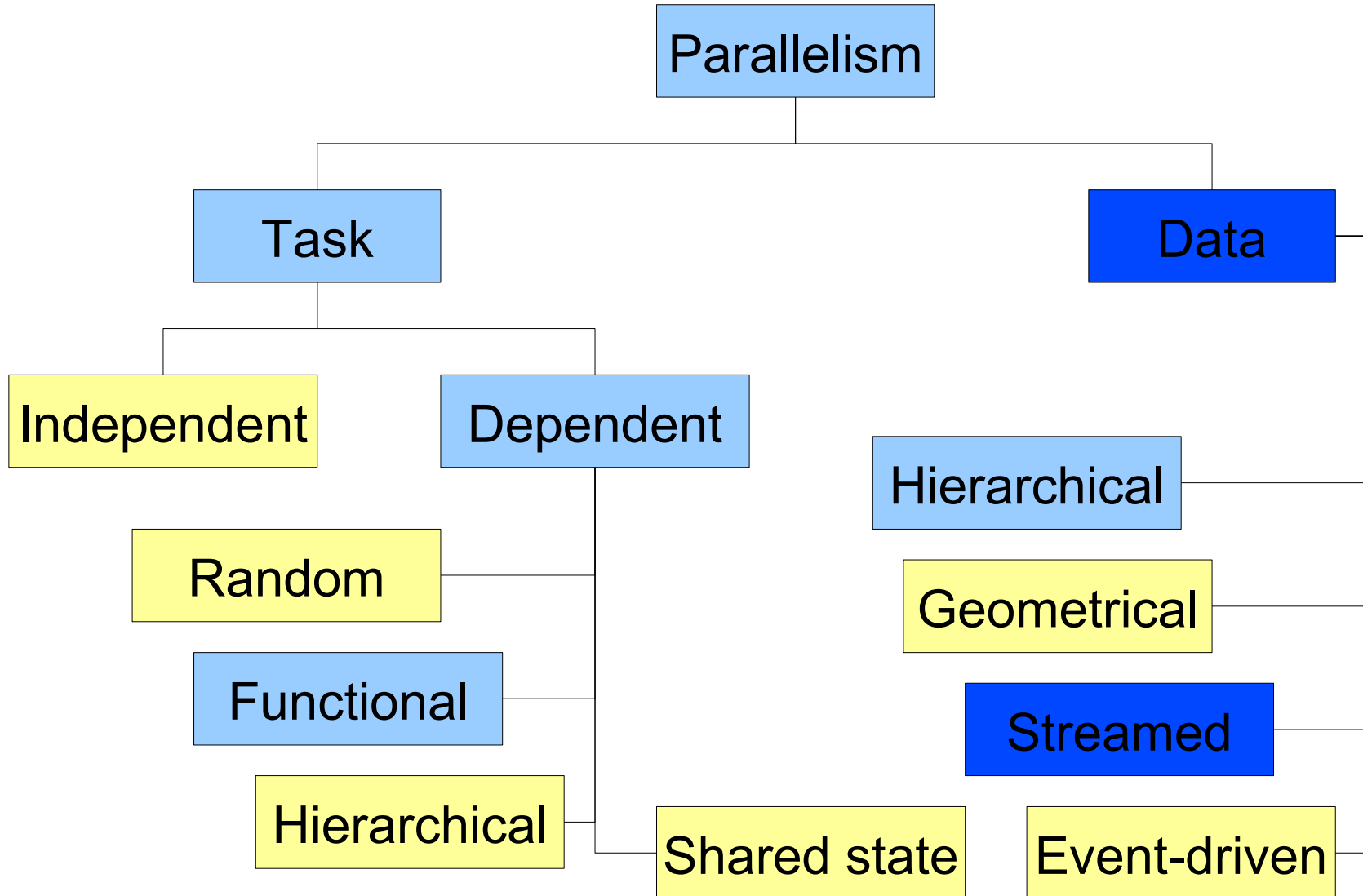
```
def pipeline = data | step1 | step2 | step3
```

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

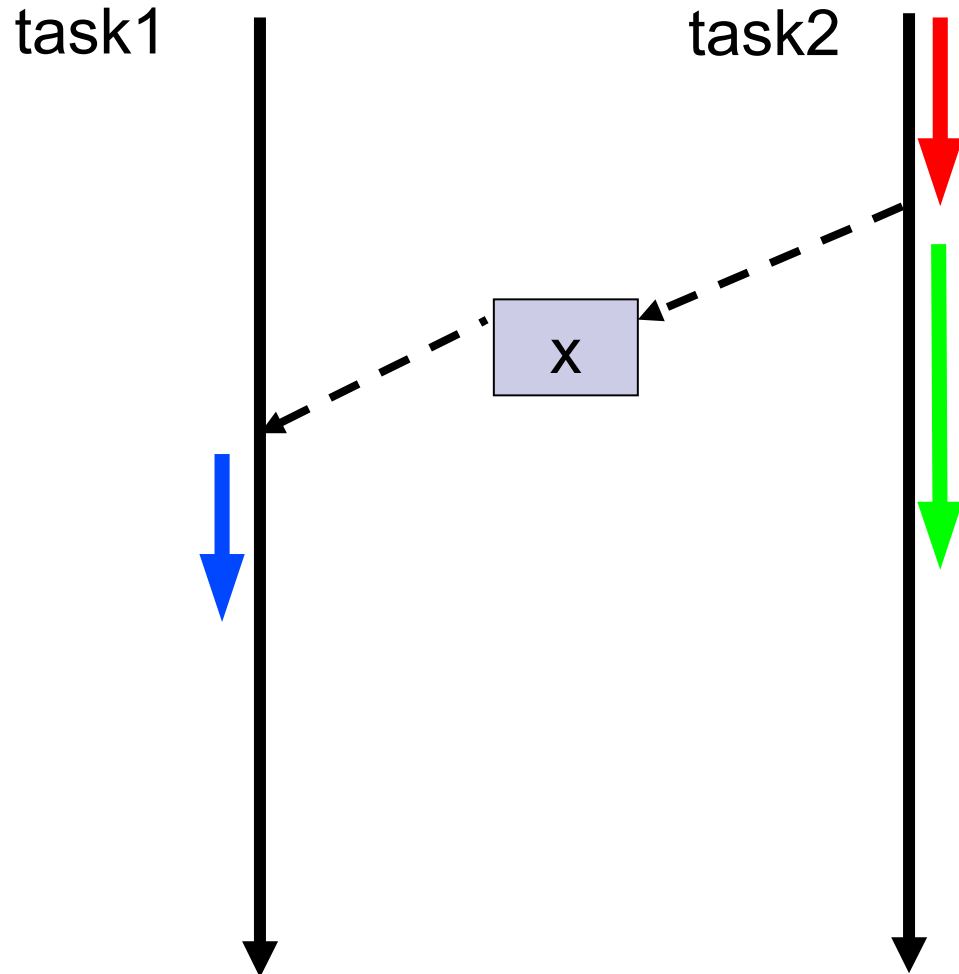
# Unroll iteration



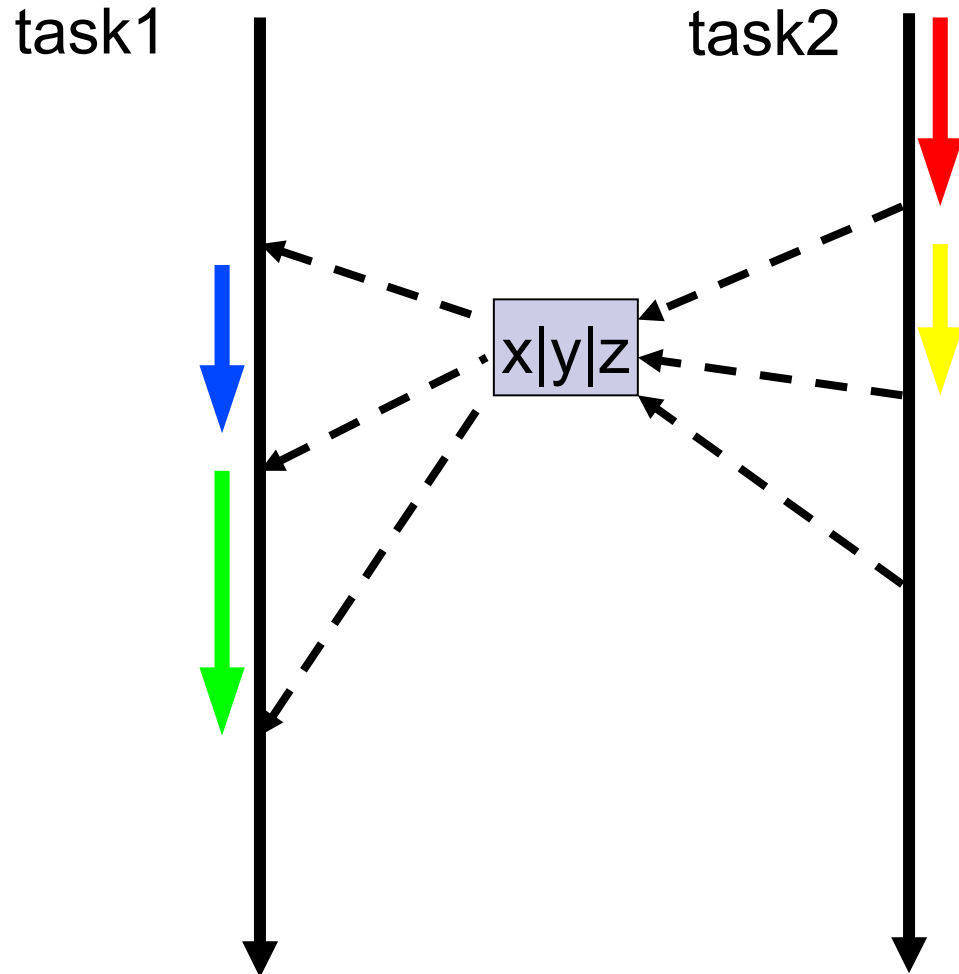
# Streamed data



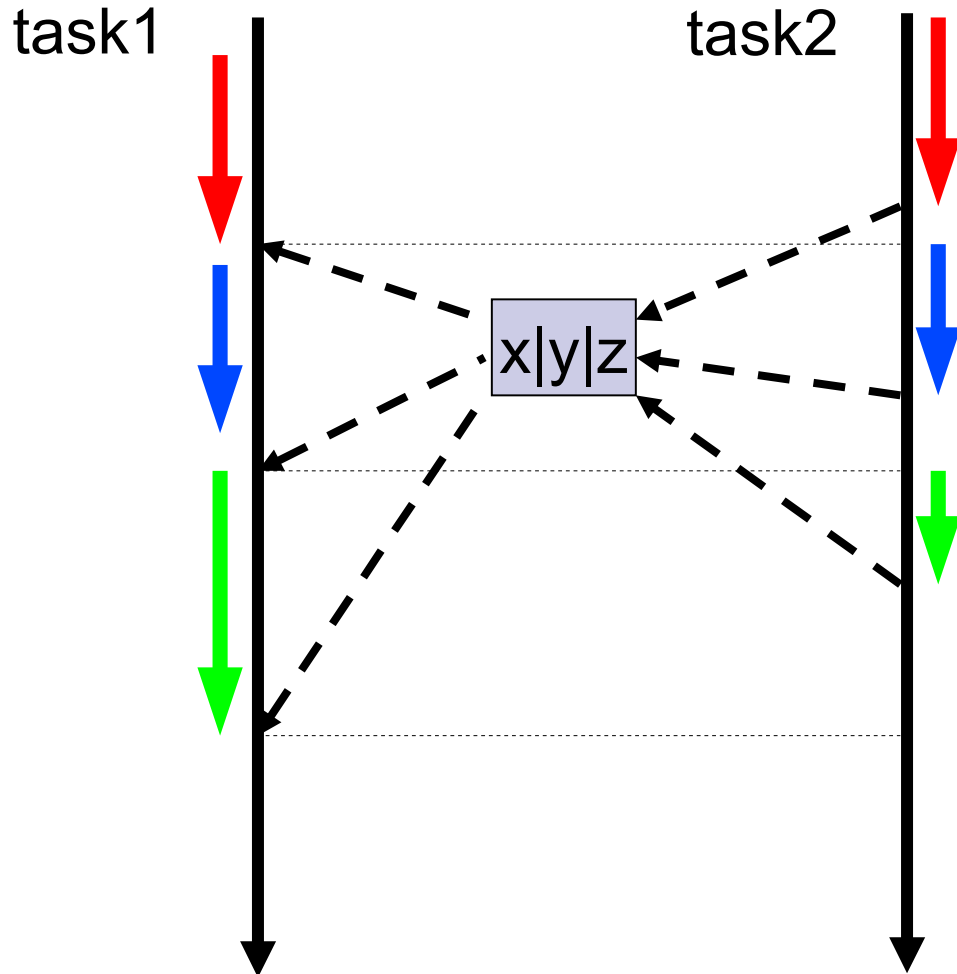
# Dataflow Variables / Promises



# Dataflow Channels



# Synchronous Channels





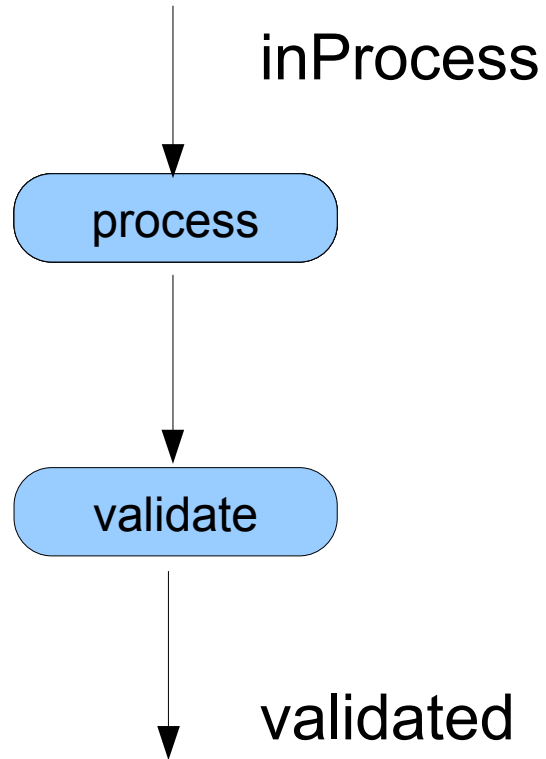
# Pipeline DSL

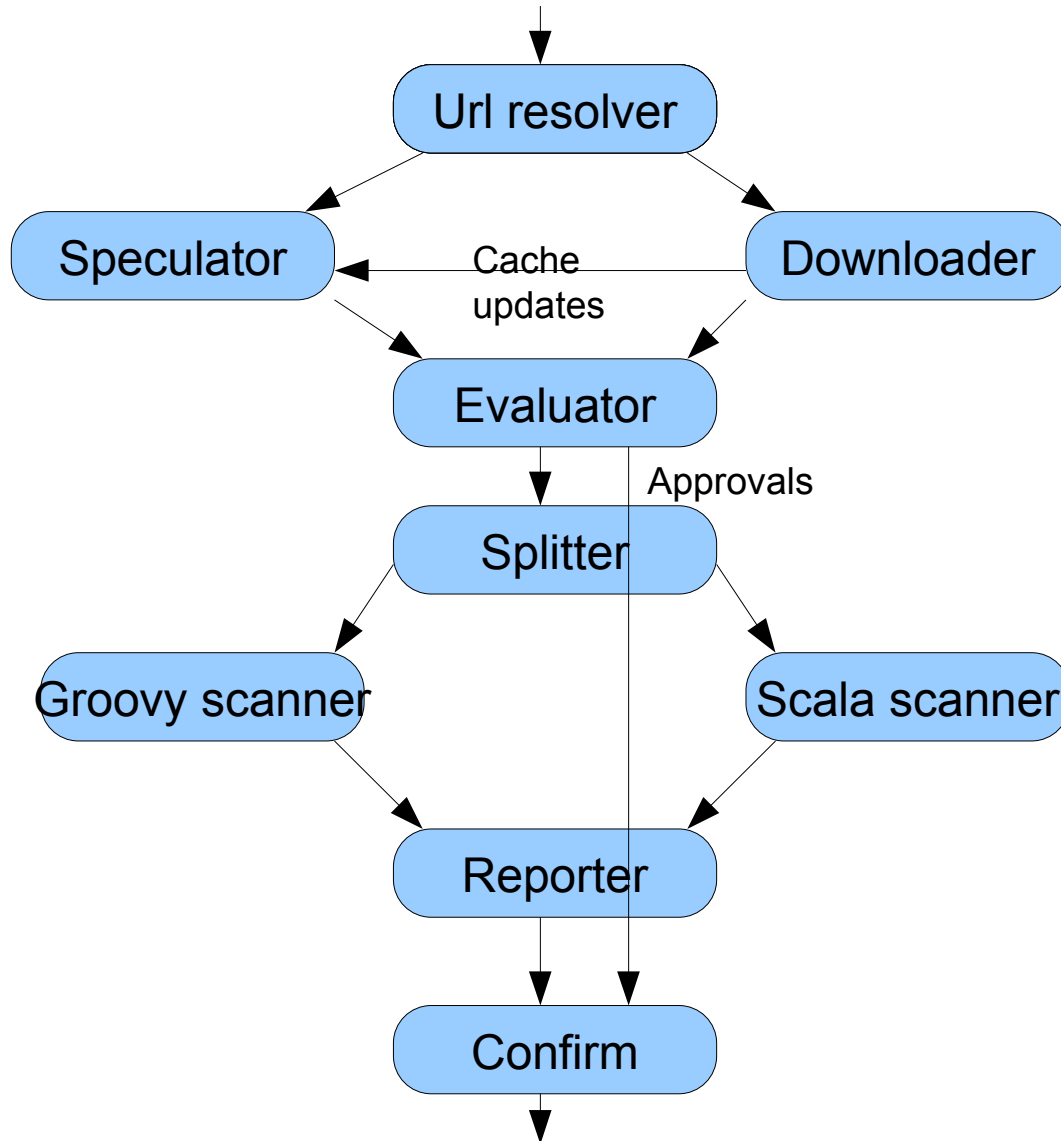
```
def toProcess = new DataflowQueue()
```

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

```
toProcess | {form -> process(form)} |  
    {processedForm -> validate(processedForm)} | validated
```

```
submissions.each {toProcess << it}
```





# Dataflow Operators

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

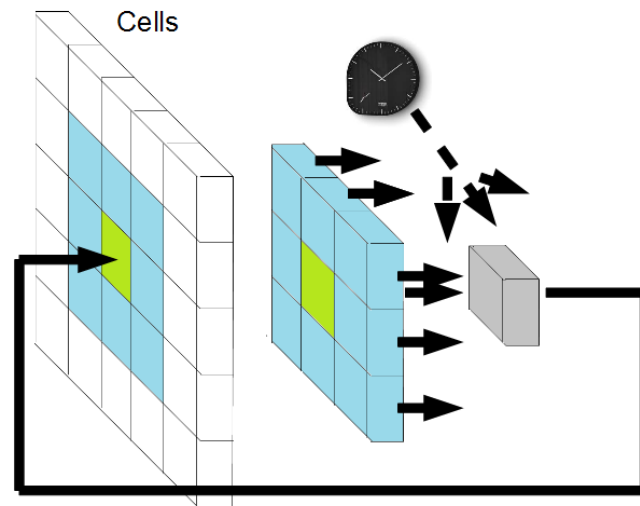
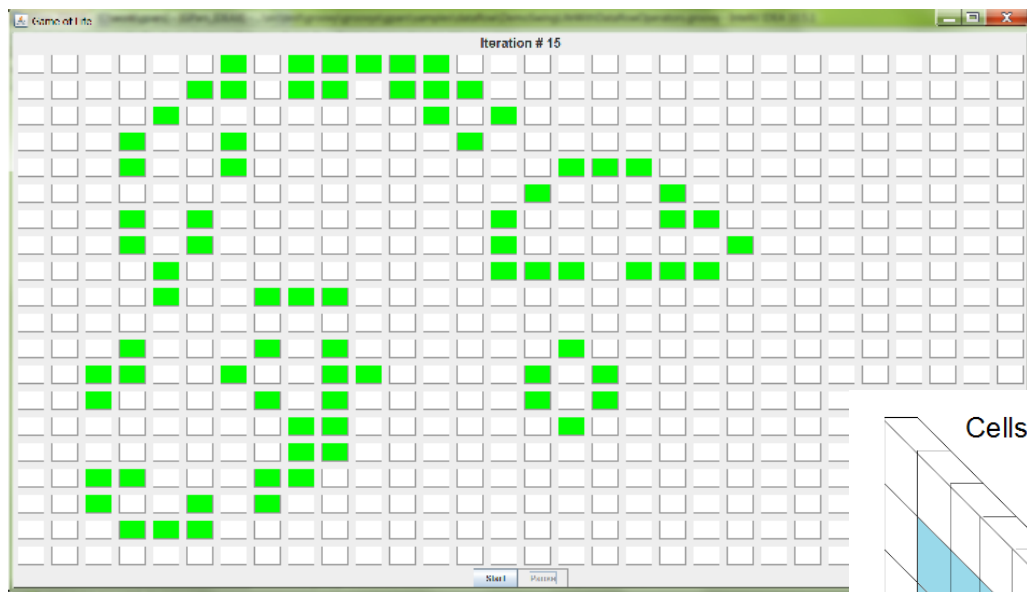
```
{header, body, footer ->
```

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

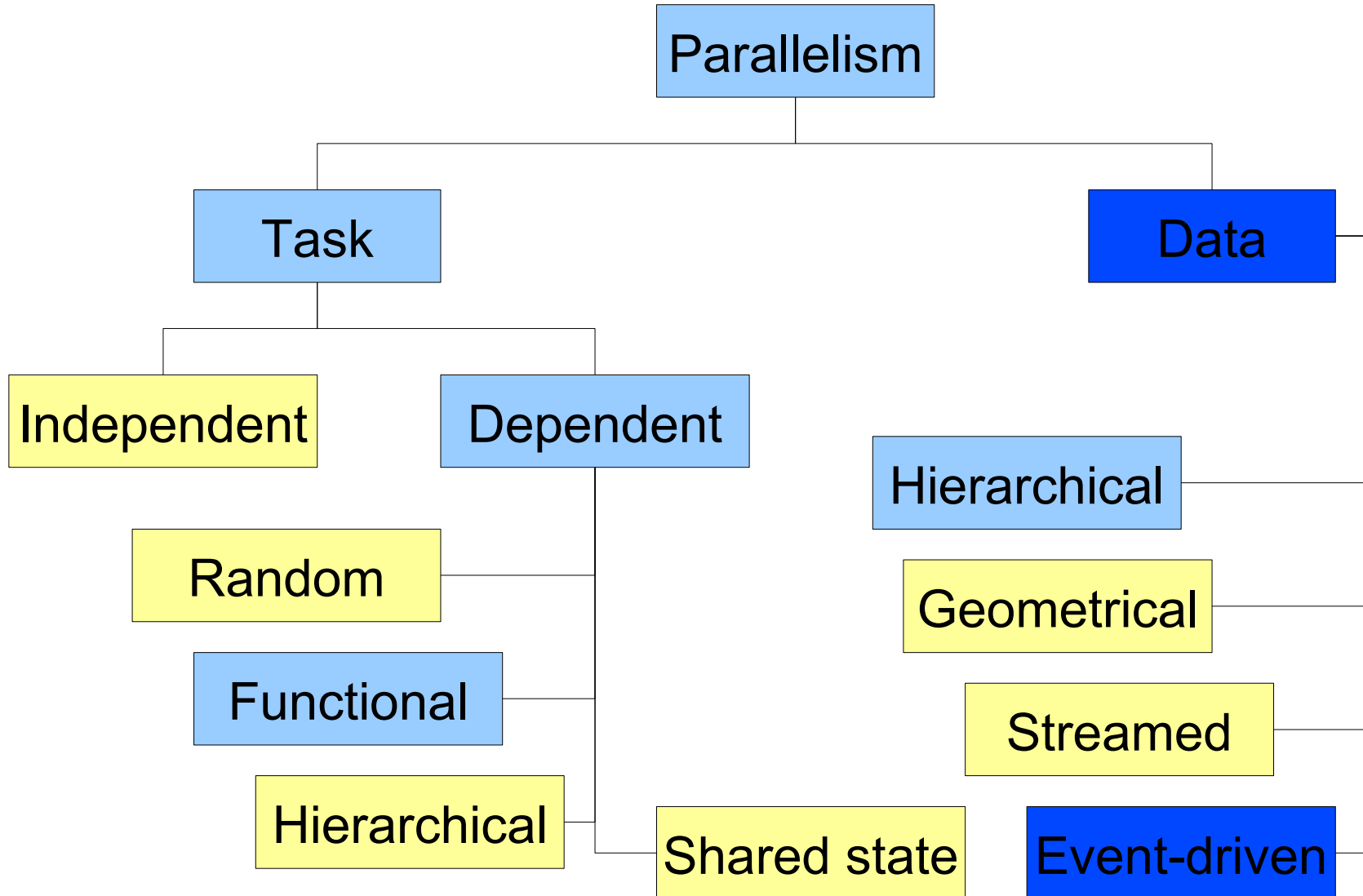
```
  bindOutput(0, article)
```

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

```
}
```

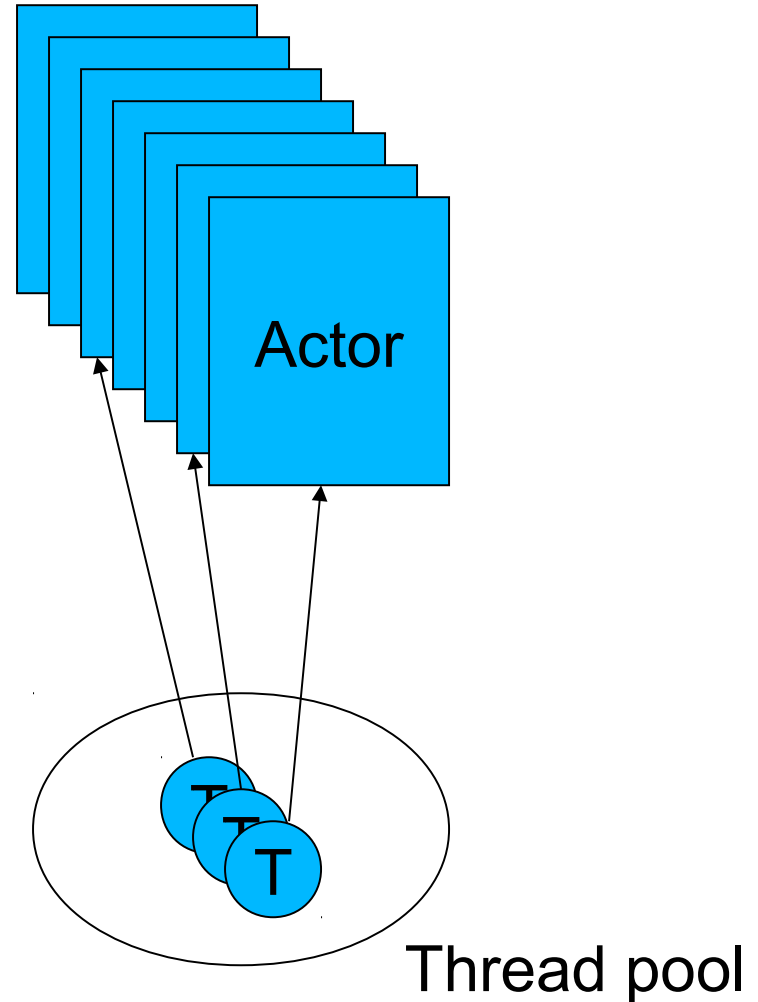


# Actors

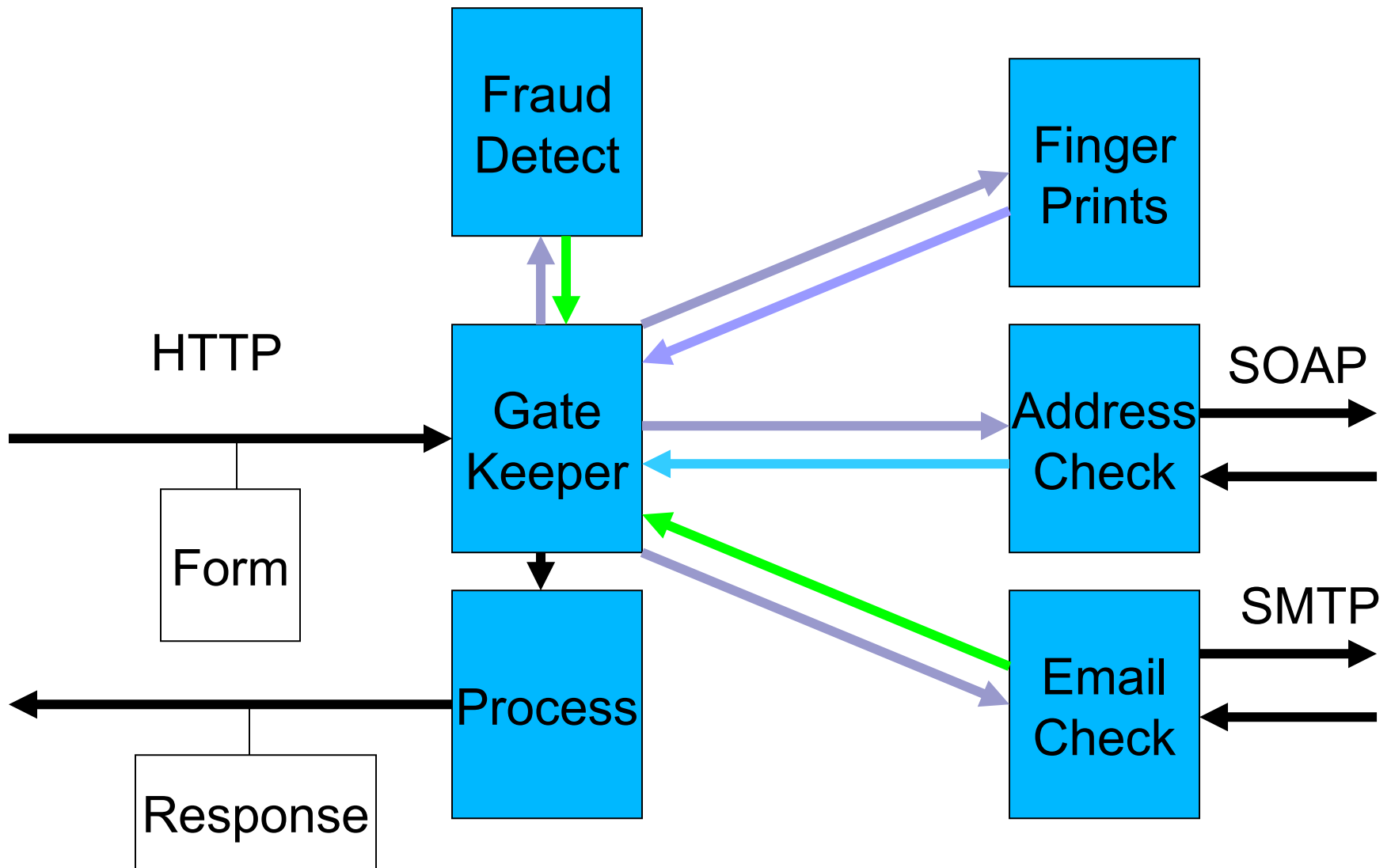


# Actors

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



# Actors use





```
graph LR; In(( )) --> Router[Router]; Router --> Enricher[Enricher]; Router --> Translator[Translator]; Router --> Endpoint[Endpoint]; Enricher --> Splitter[Splitter]; Translator --> Splitter; Endpoint --> Splitter; Splitter --> Agregator[Agregator]; Agregator --> Filter[Filter]; Filter --> Resequencer[Resequencer]; Resequencer --> Checker[Checker]; Checker --> Out[Output];
```

Enricher

Router

Translator

Endpoint

Splitter

Agregator

Filter

Resequencer

Checker

# Router

## Endpoint

# Splitter

# Agregator

## 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  
}
```

# Event driven – actors

```
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'  
    }  
}
```

# Event driven – active objects

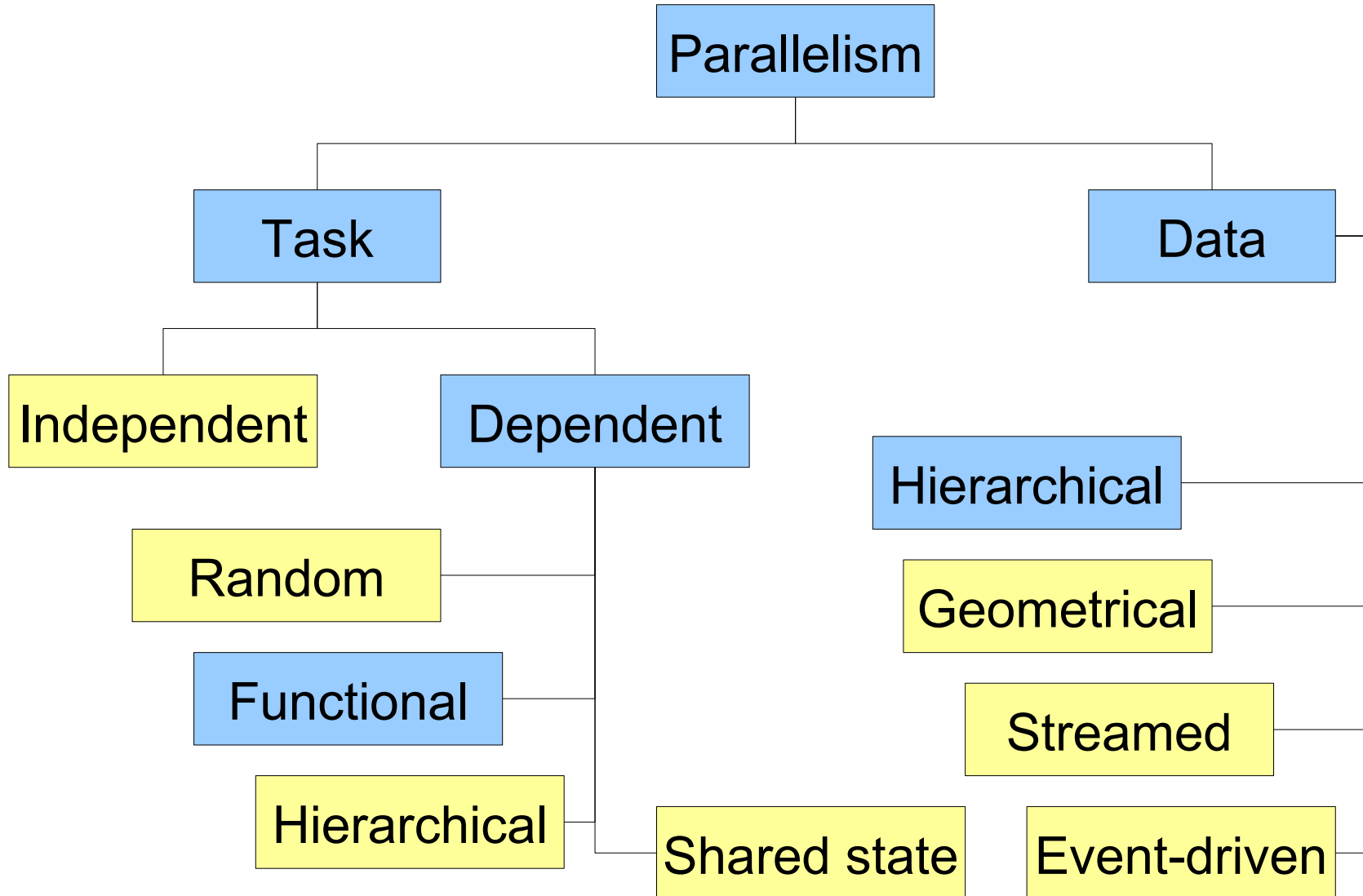
@ActiveObject

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

@ActiveMethod

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

# Conclusion



# Summary

Parallelism is not hard, multi-threading is

Jon Kerridge, Napier University

# References

<http://groovy-lang.org>

<http://grails.org>

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

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