Parallel Objects for Multicores

A Glimpse at the Parallel Language Encore

Dave Clarke & Tobias Wrigstad
Uppsala University

Overview

Tutorial Overview

Background and Motivation

Language Design Inversion

Encore Language Design (5 Inversions)

(Exercise Session)

Motivation

Background

In the early 2000's hardware hit a wall

- *"Too much power used too inefficiently"*
- *CPU temperature approaching sun's surface*
- *Adding 2x transistors yields 2% speedup*

Solution: multi- and manycore machines

- *Use 2x transistors to build 2x cores*
- *200% speedup — in theory*
- *Essentially pushes the problem over to software*
- *"'No one' knows how to program these machines"*

Object-Oriented Parallel Programming

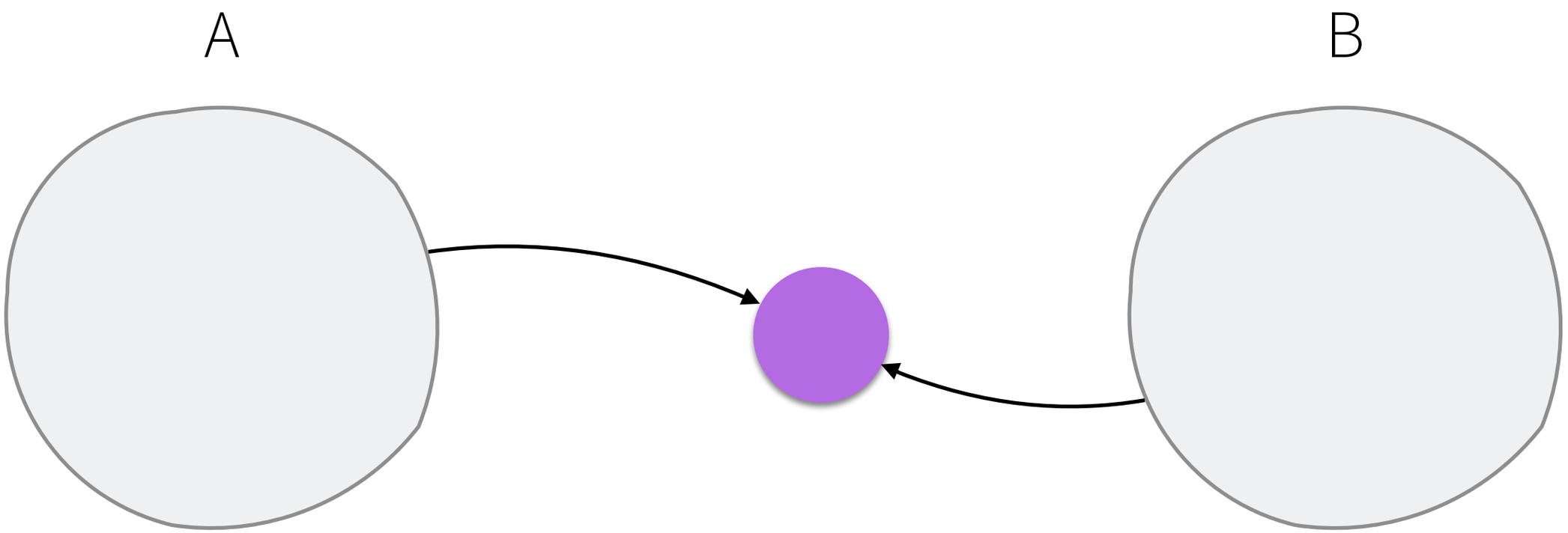
Combining object-orientation and parallelism is hard

- ***Aliasing*** *make reasoning about efficient parallelism difficult*
- *Abstract dynamic structures stress memory bottlenecks*
- *Compositionality of concurrency control…*

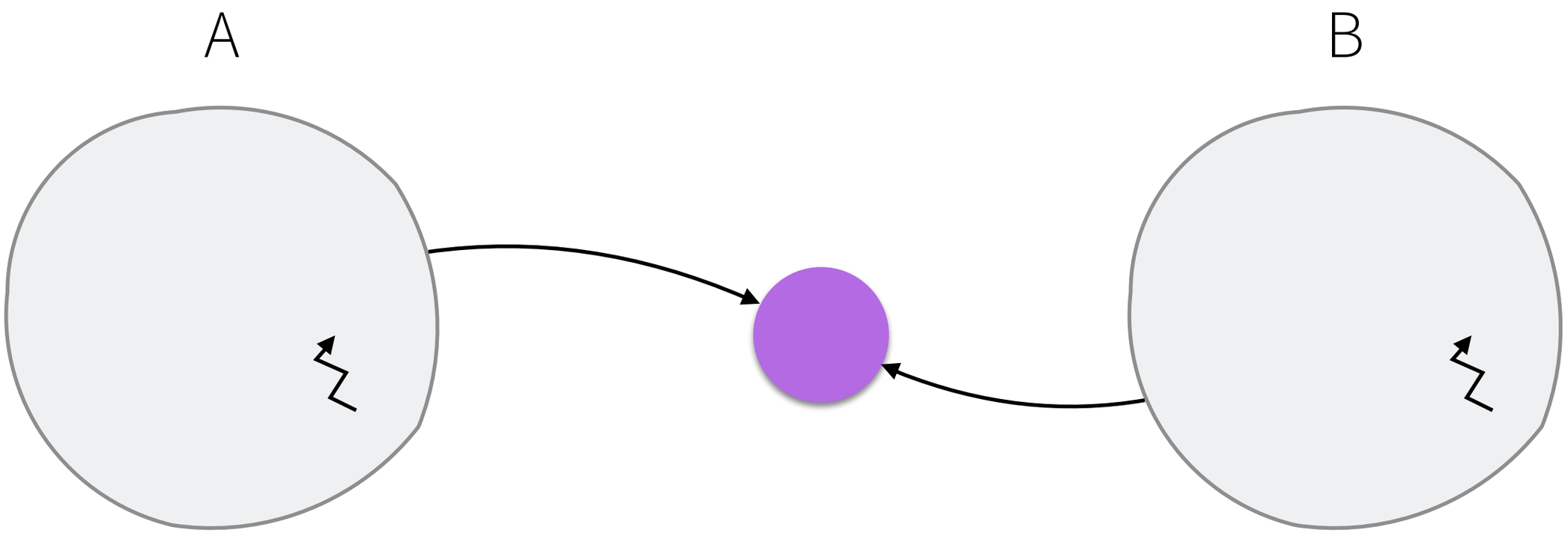
One root cause: classical languages evolved in a predominantly sequential setting

- *Support for concurrency & parallelism as an afterthought*
- *Thread libraries are easily integrated, but hard to use*
- *Essentially pushes the problem over to application programmers*
- *"'No one' knows how to program with **lots of threads**"*

Aliasing Problem: Shared Mutable State

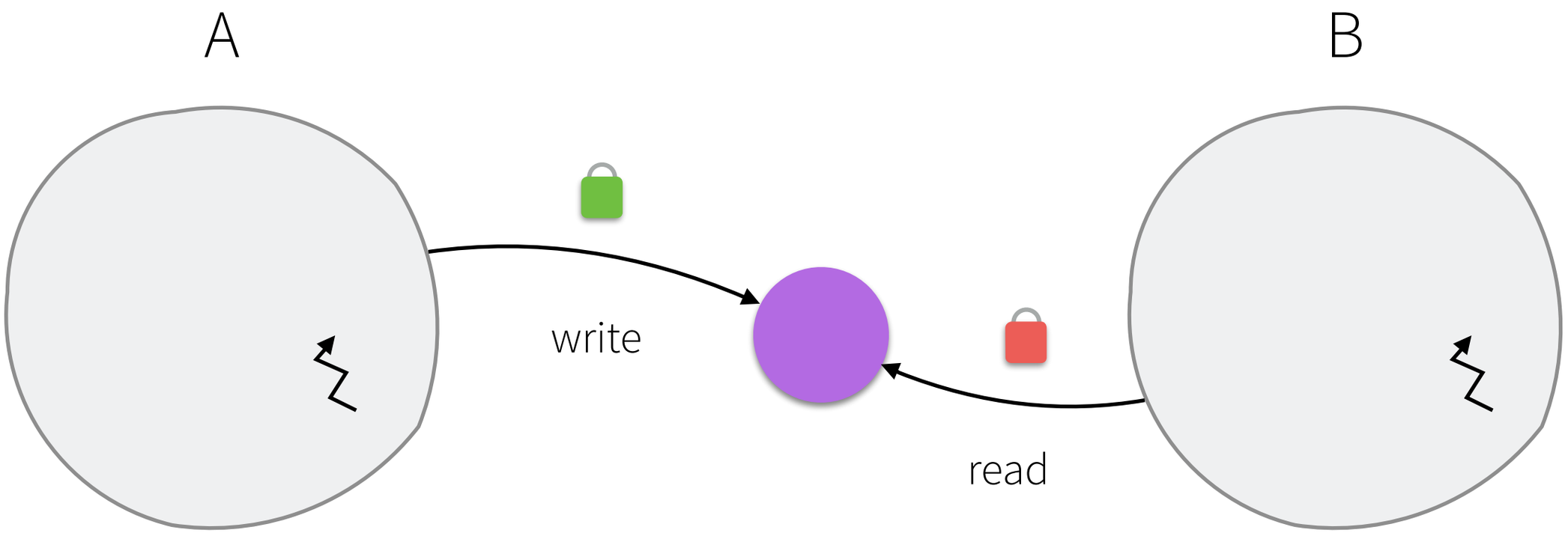


Aliasing Problem: Shared Mutable State



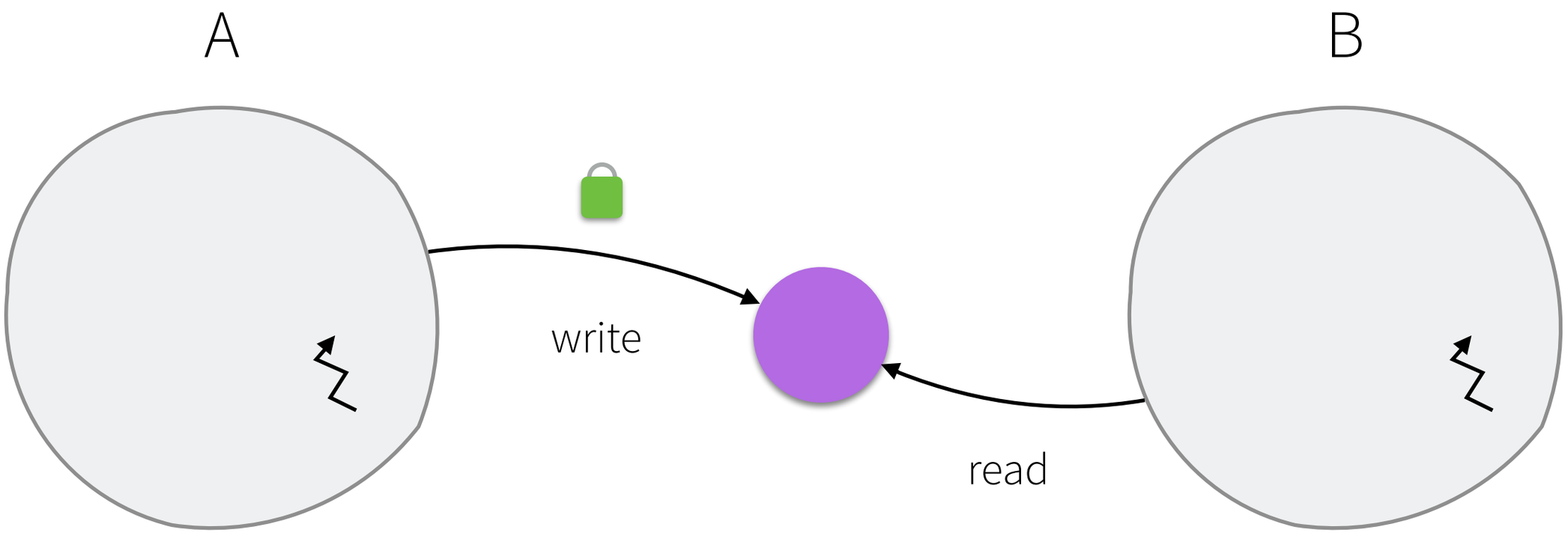
even worse with concurrency/parallelism

Locks

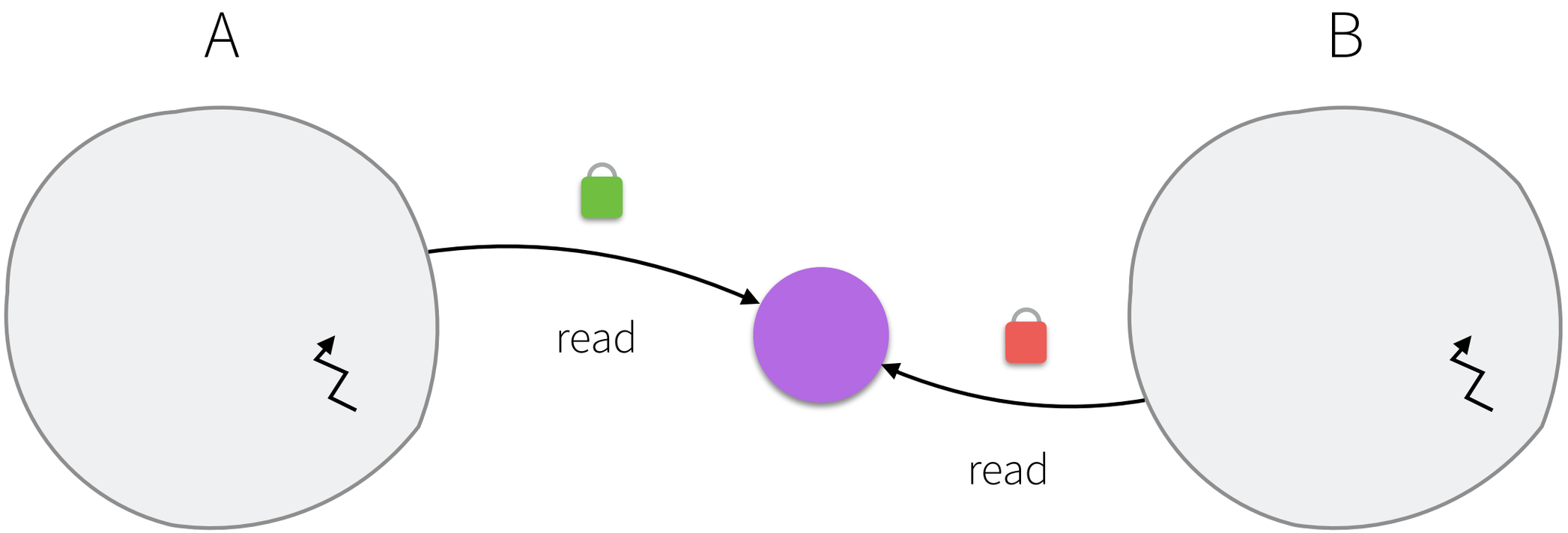


Must acquire a lock before accessing a certain resource

Locking Too Little

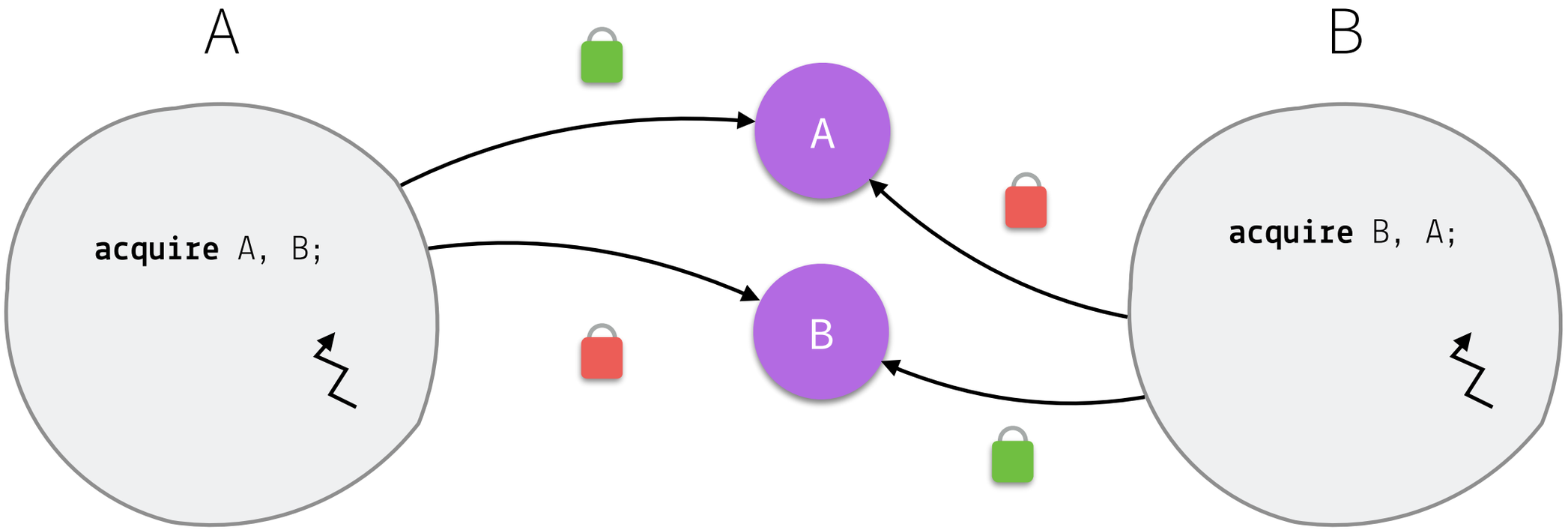


Locking Too Much



force interleaved access even for commuting operations

Compositionality of Locks



deadlock

Locks are Good, Locks are Bad

Threads and locks are easy to add to a programming language with minimal changes

- Place **burden on programmer** instead of programming **language designer**
- Code that requires synchronisation is indistinguishable from code that does not

Locks perform quite well quite often

- Uncontended locks are cheap
- Highly contended locks are expensive

- Coarse-grained locking is **simpler** but reduces parallelism
- Fine-grained locking allows parallelism, but is **harder** (e.g. deadlocks)

UPSCALE

Rethink object-oriented programming languages

- *Remove sequential bias in classical languages*
- *Keep a sufficiently object-oriented programming model*
- *Save industry investments in OOSD*

CWI
Centrum Wiskunde & Informatica

UiO : University of Oslo

End goal: make massively parallel programming in OO-languages possible & affordable

Imperial College London

Language Design Inversion

Inversion

Most modern languages are designed **first** for sequential programming, with parallel programming constructs tacked on — Erlang is one exception.

Mutability, possibly data dependencies, shared state, poor locality etc all limit possible parallelism and scalability.

Inversion = adopt defaults that favours parallelism and scalability.

Inversion in design of Encore

Concurrent-by-default

(Data)Parallel-by-default

Data-race-free-by-default

Isolated-by-default

Asynchronous-by-default

Linear-by-default

Immutable-by-default

Local-by-default

Multi-object-by-default

…

… by-default

Defaults can be overridden
— additional code overhead.

Some defaults are conflicting
— need to be addressed.

Concurrent-by-Default

Concurrency-by-Default

Java objects were designed for sequential access.

Threads trample over objects.

Locks/monitors added to protect objects.

Erlang has concurrency by default (actors), but it is not object-oriented.

Actors/Active Objects

Active Object Characteristics

Mailbox

Single thread of control

Isolation

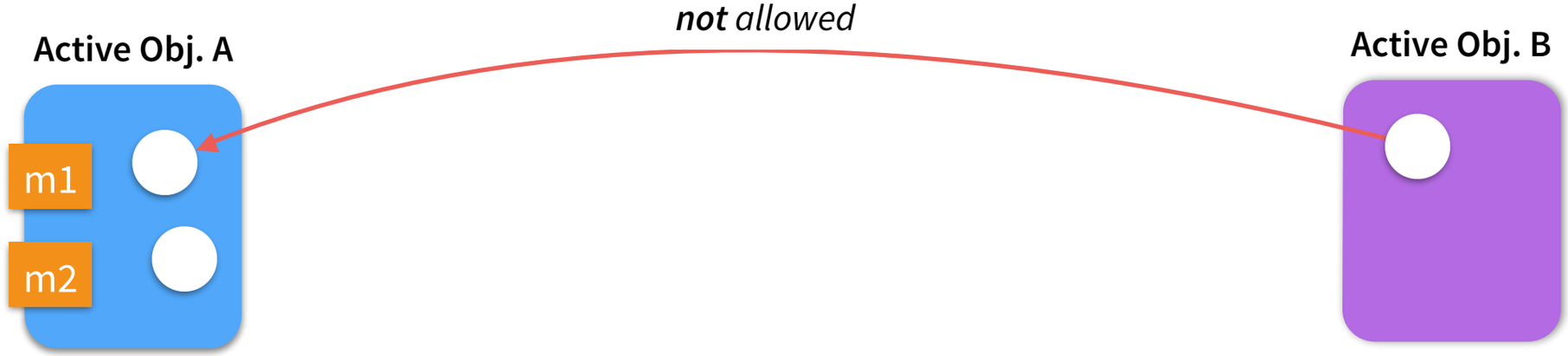
Asynchronous communication

- Saturation of asynchronous operations on different object enables efficient use of parallel machines

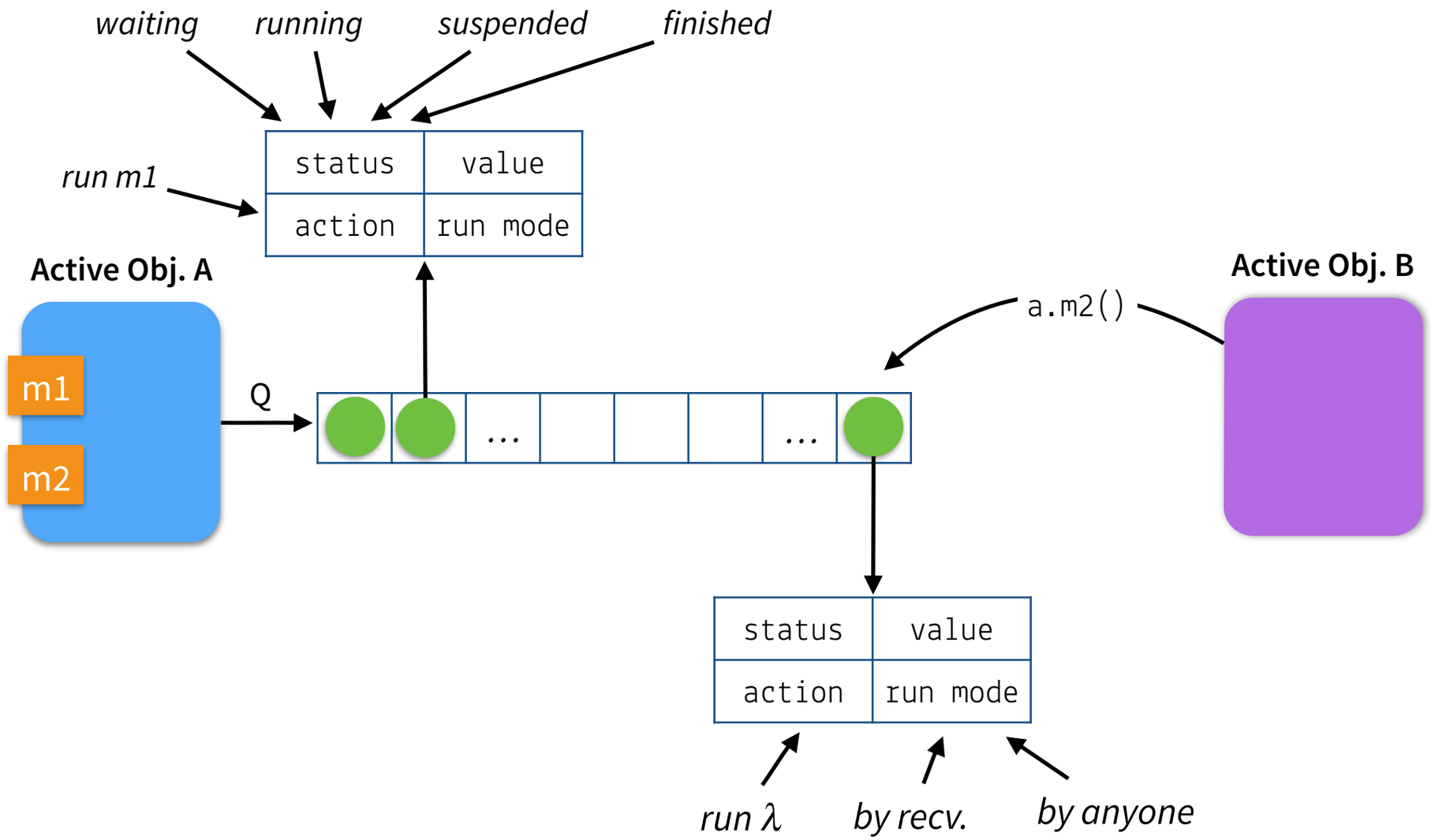
Method suites defined in classes + usually OO

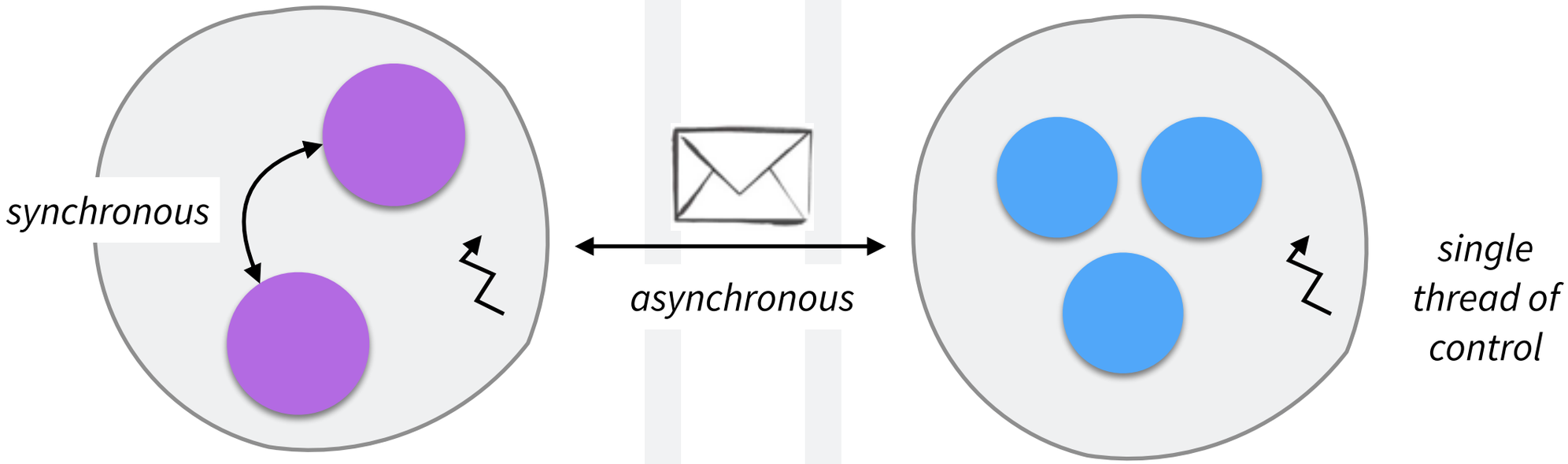
Return values handled using futures

Actor Characteristics

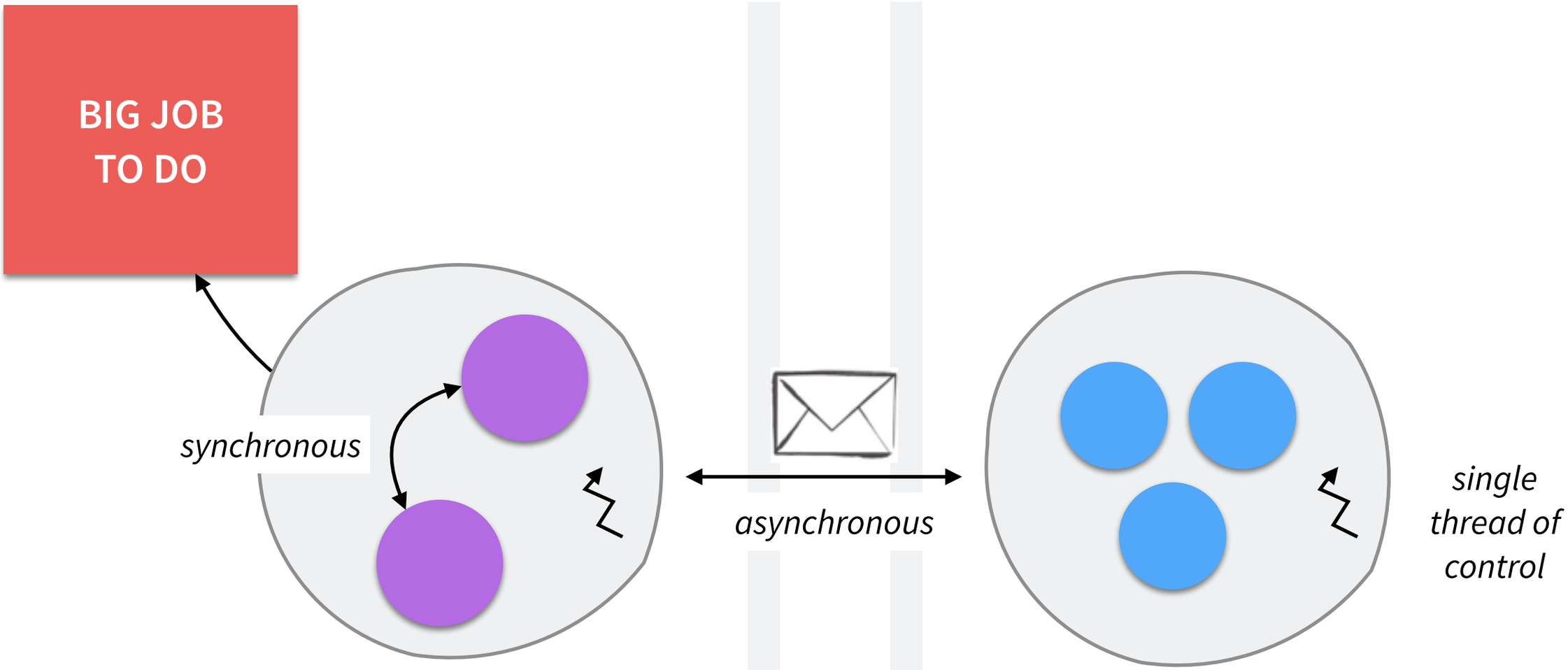


Actor Characteristics

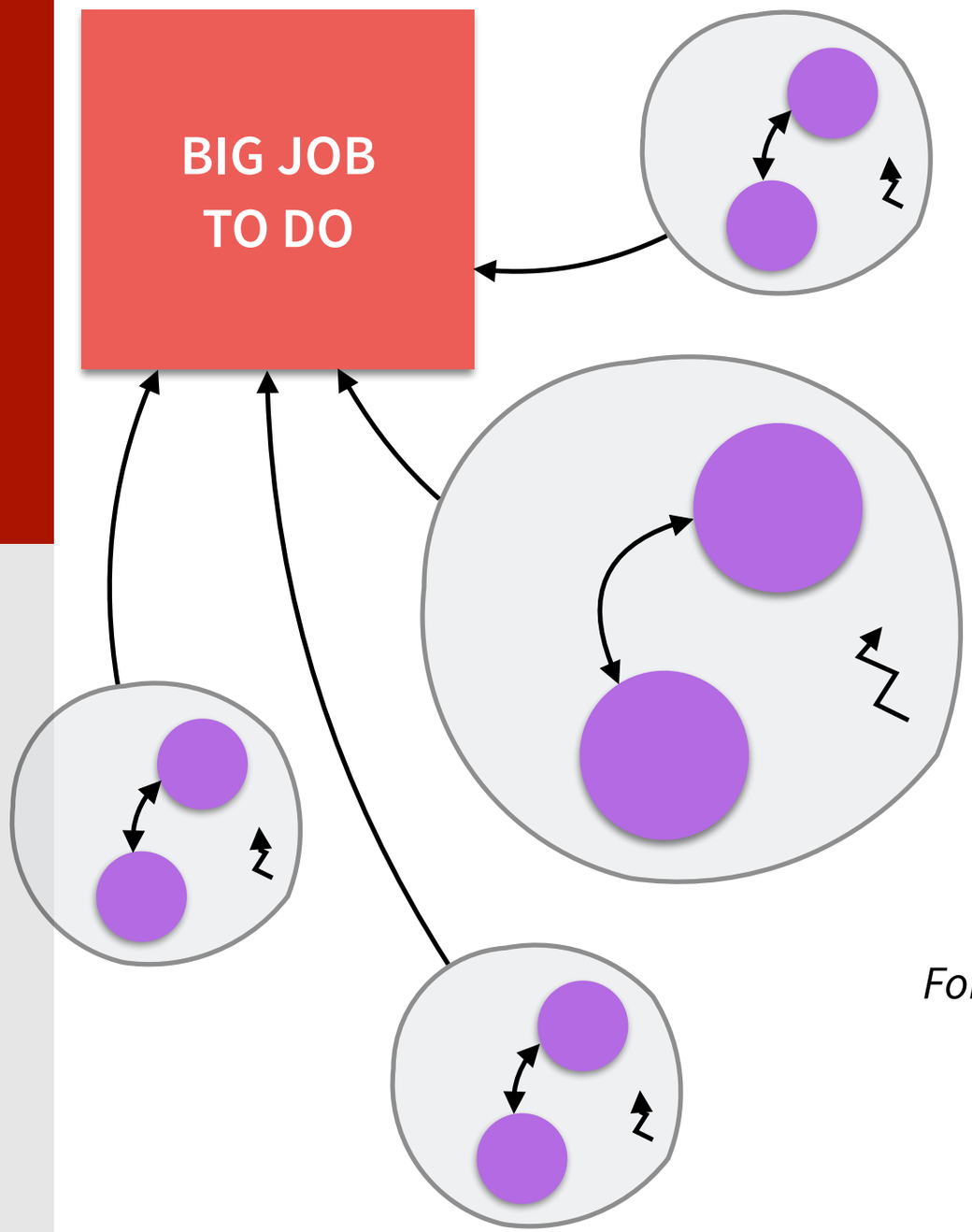




Active Object-based Parallelism



Active Object-based Parallelism



Fork multiple actors

Thread Ring Example (A Litte Bit Boring)

```
class Main
  def main(): void
    let
      index     = 1
      first     = new Worker(index)
      next      = null : Worker
      nhops     = 50 000 000
      ring_size = 503
      current   = first
    in {
      while (index < ring_size) {
        index = index + 1;
        next = new Worker(index);
        current ! setNext(next);
        current = next;
      };
      current ! setNext(first);
      first ! run(nhops);
    }
```

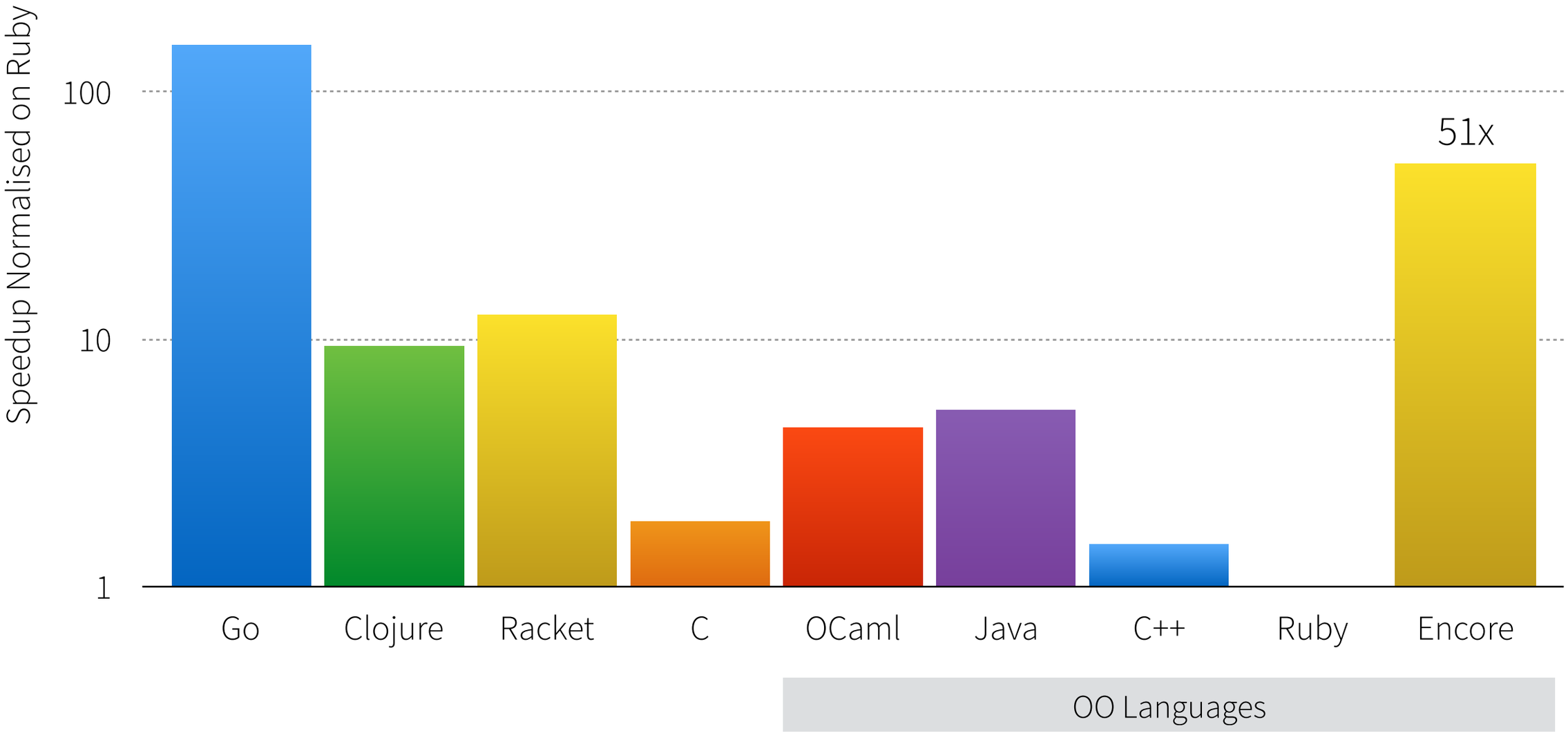
```
class Worker
  id : int
  next : Worker

  def init(id : int): void
    this.id = id

  def setNext(next: Worker): void
    this.next = next

  def run(n : int): void
    if (n > 0)
    then this.next!run(n-1)
    else print(this.id)
```

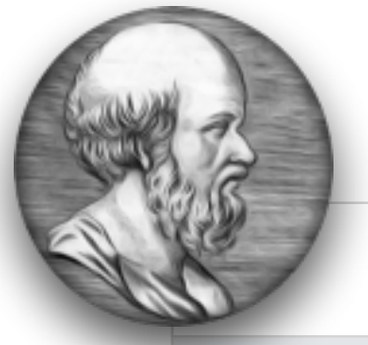
Threadrings Benchmark [pl shootout bench]

PonyRT inside!

Tested on a 4 core laptop

Note: higher is better





Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Sieve of Eratosthenes

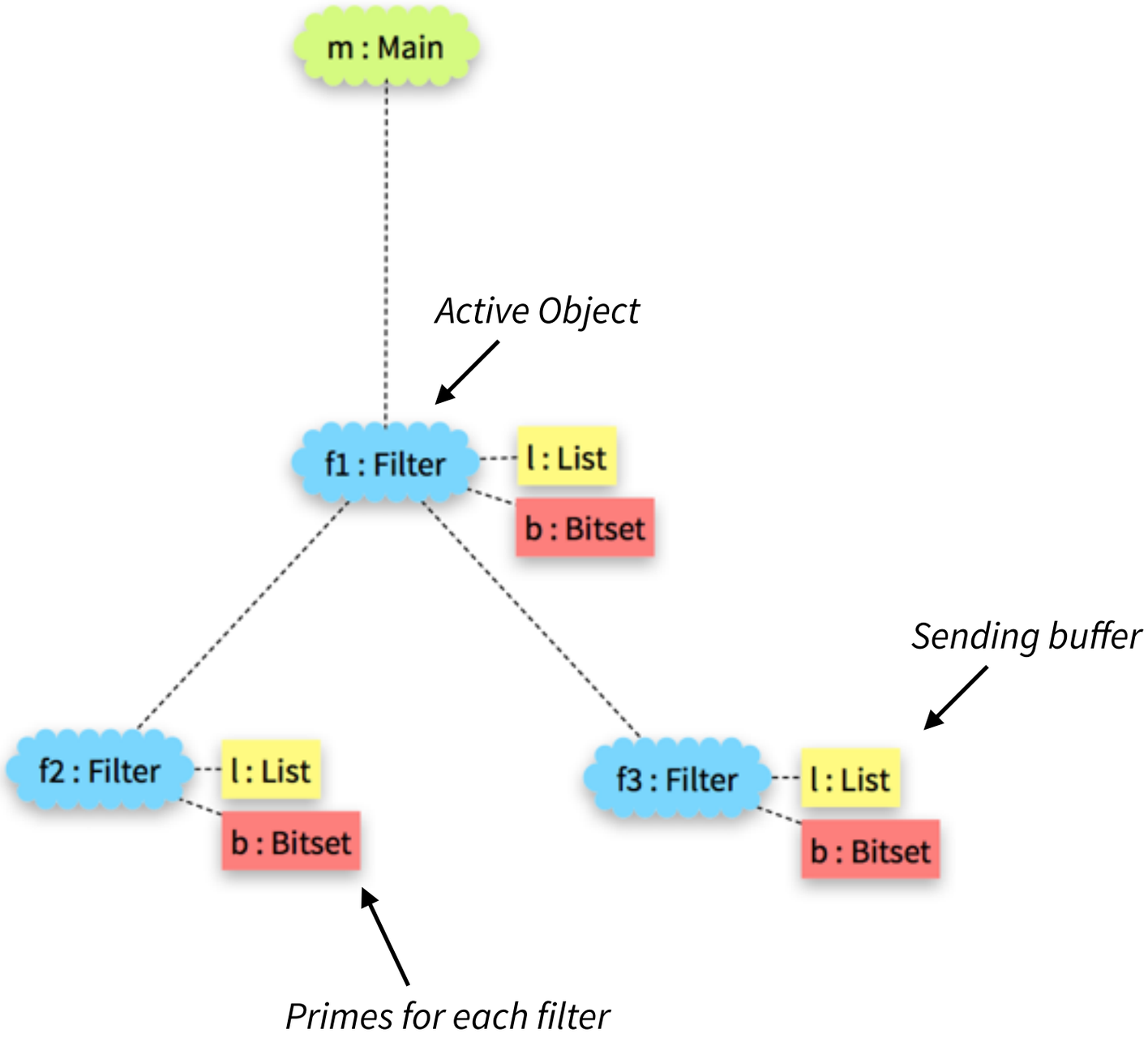
	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Parallel Sieve of Eratosthenes

Source		2	3	4	5	6	7	8	9	10	W1
	11	12	13	14	15	16	17	18	19	20	
	21	22	23	24	25	26	27	28	29	30	W2
	31	32	33	34	35	36	37	38	39	40	
	41	42	43	44	45	46	47	48	49	50	W3
	51	52	53	54	55	56	57	58	59	60	
	61	62	63	64	65	66	67	68	69	70	W4
	71	72	73	74	75	76	77	78	79	80	
	81	82	83	84	85	86	87	88	89	90	W5
	91	92	93	94	95	96	97	98	99	100	

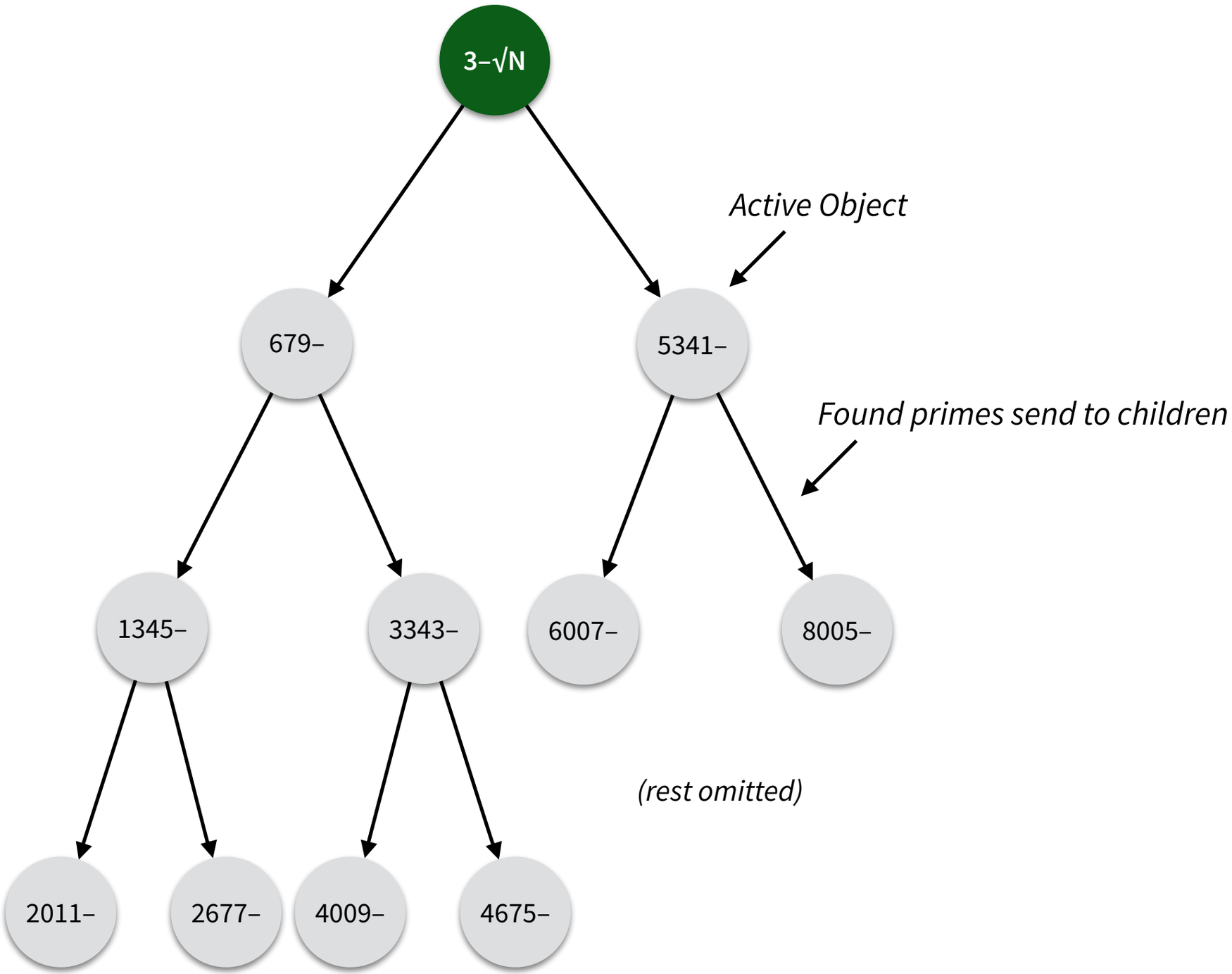
Prime Sieve Benchmark

~ 200 LOC Encore + 130 LOC from libraries

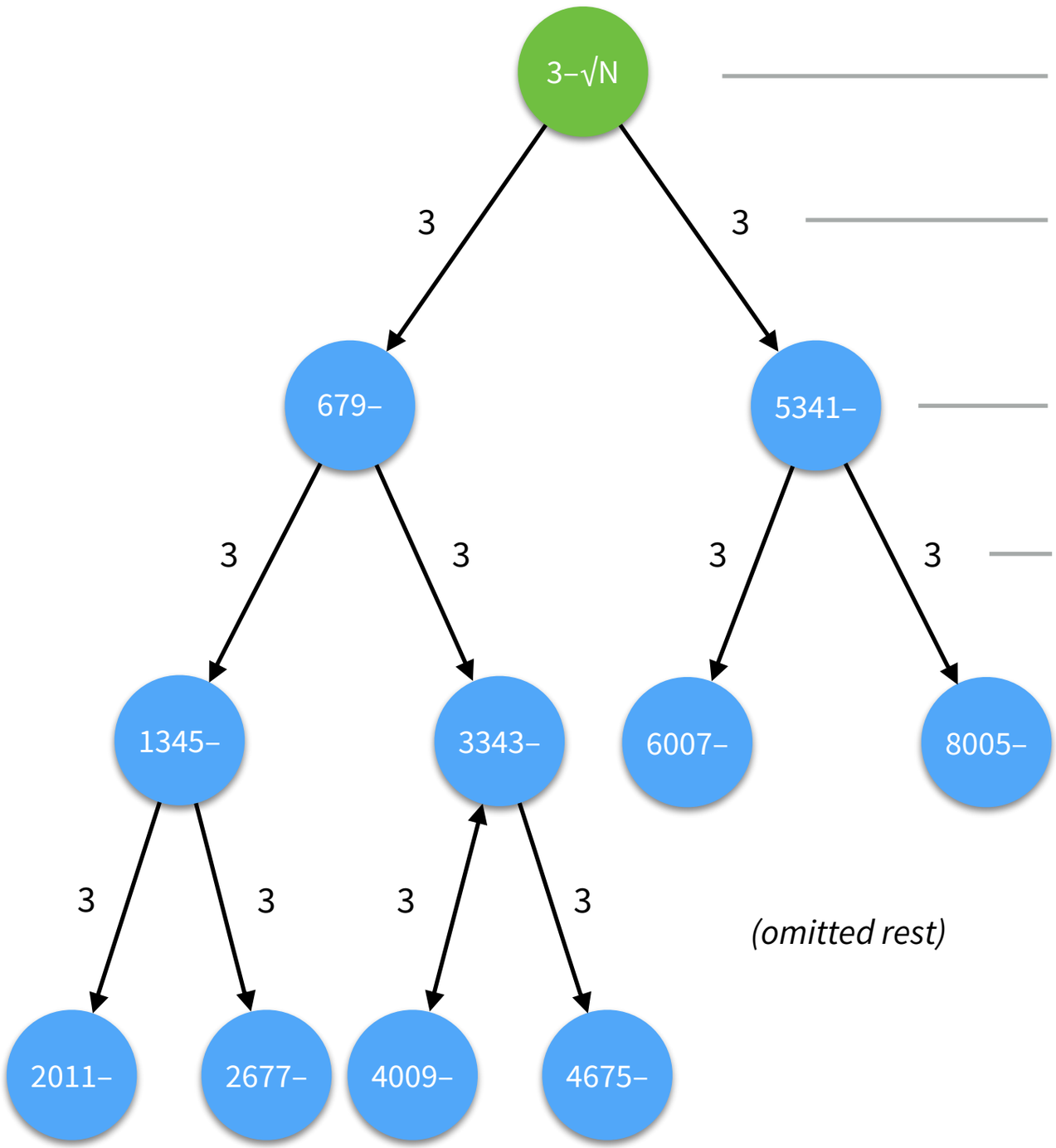


Parallel Prime Sieve in a Nutshell

~ 200 LOC Encore + 130 LOC from libraries



Parallel Prime Sieve in a Nutshell



Scans vector of numbers linearly to find primes

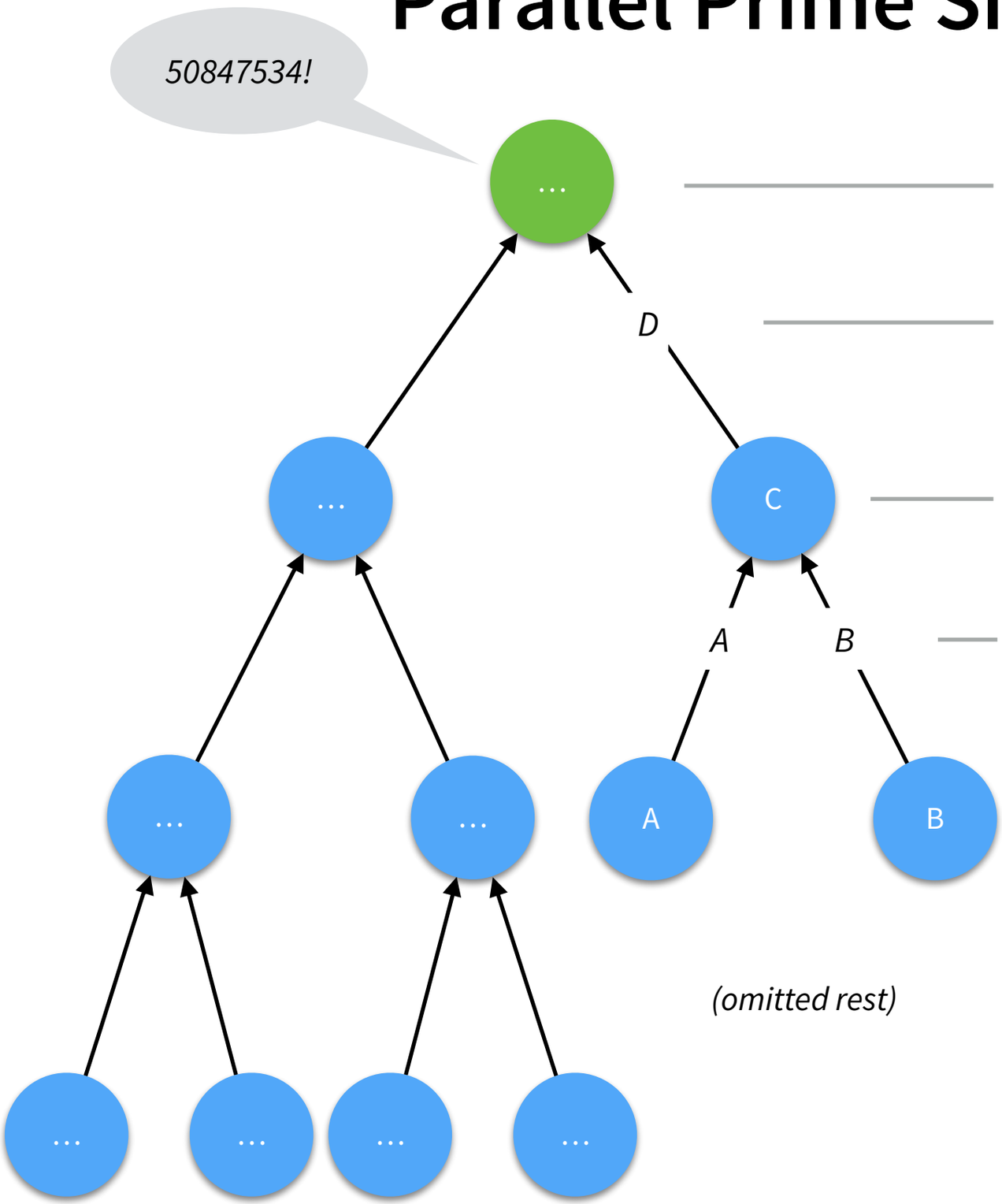
Forwards each prime P to its immediate children

Cancels all multiples of P in their range

Forwards each prime P to its immediate children

(omitted rest)

Parallel Prime Sieve in a Nutshell



Aggregate result with children, display

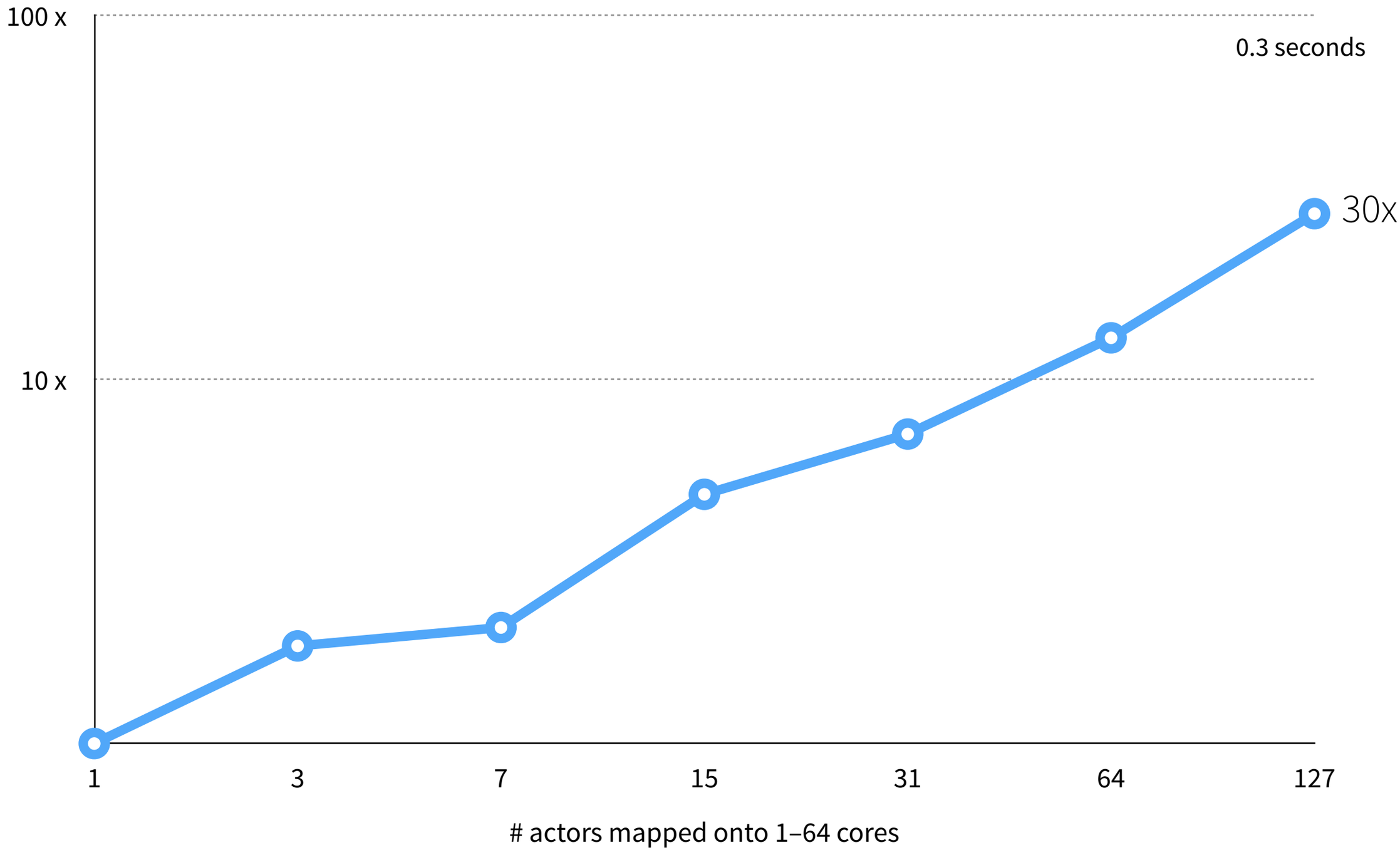
$D = A + B + C$

Aggregate result with children, send to parent

e.g., "A primes found"

When done, send result to parent

Strong Scalability (Normalised on 1, calculating 1.6B primes)

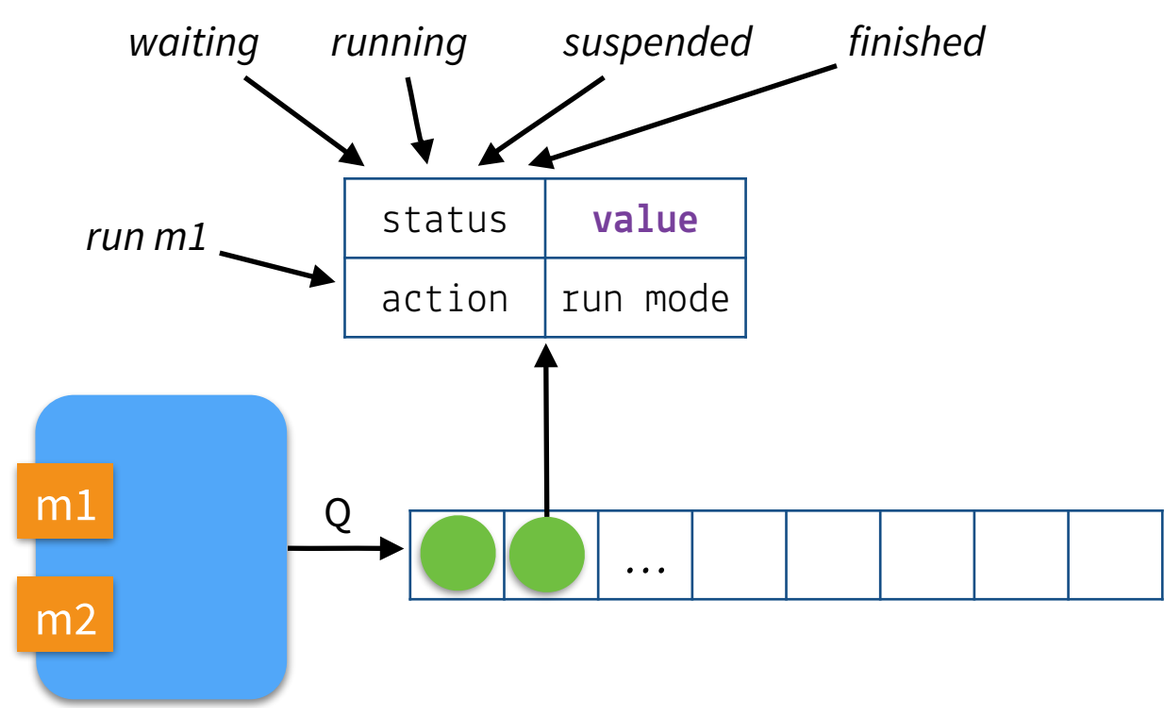


Back to the Futures

A future is a placeholder for a value

Asynchronous methods return futures …

… when the method is complete, its result is assigned to the future — the future is *fulfilled*.

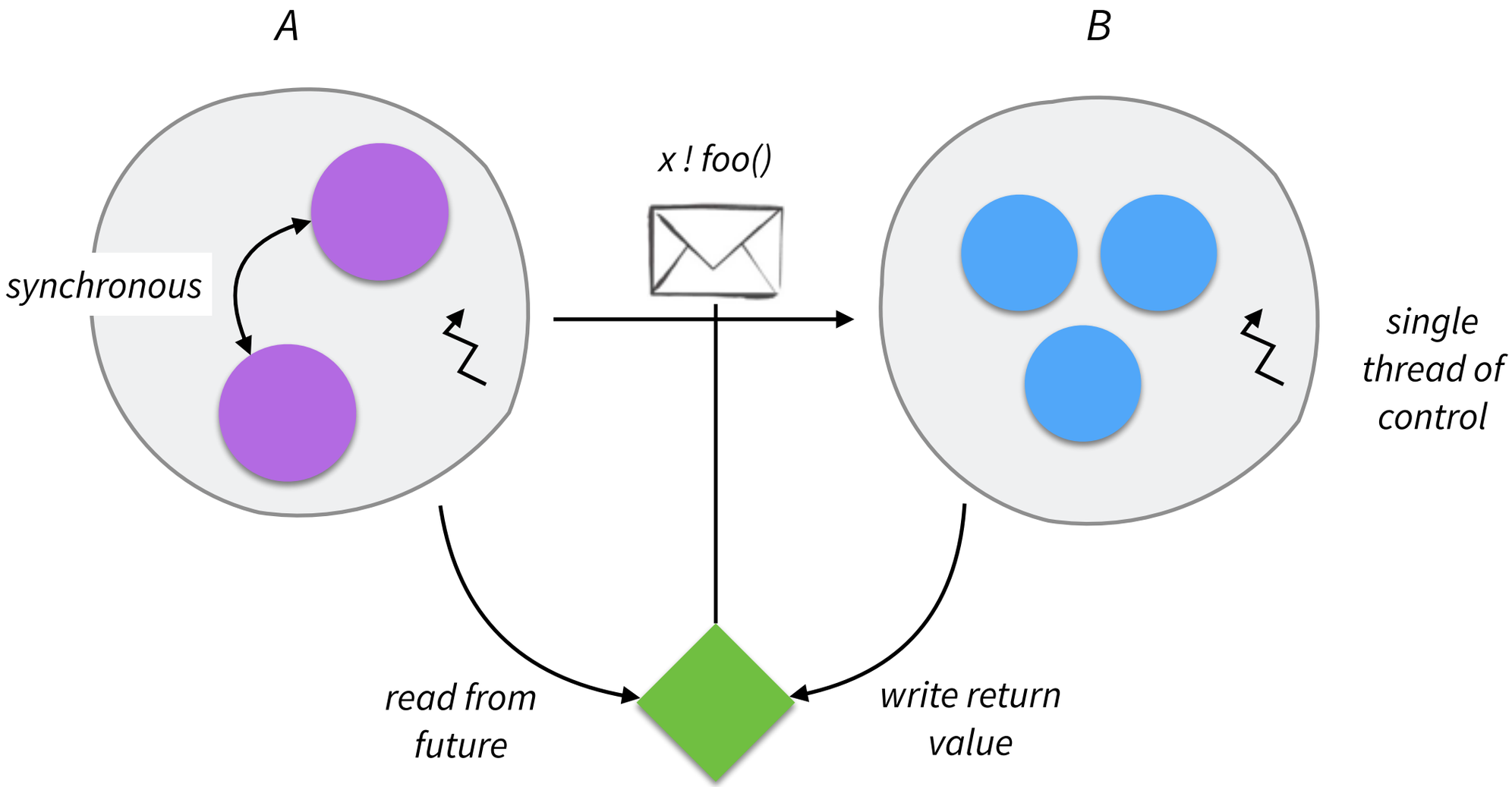


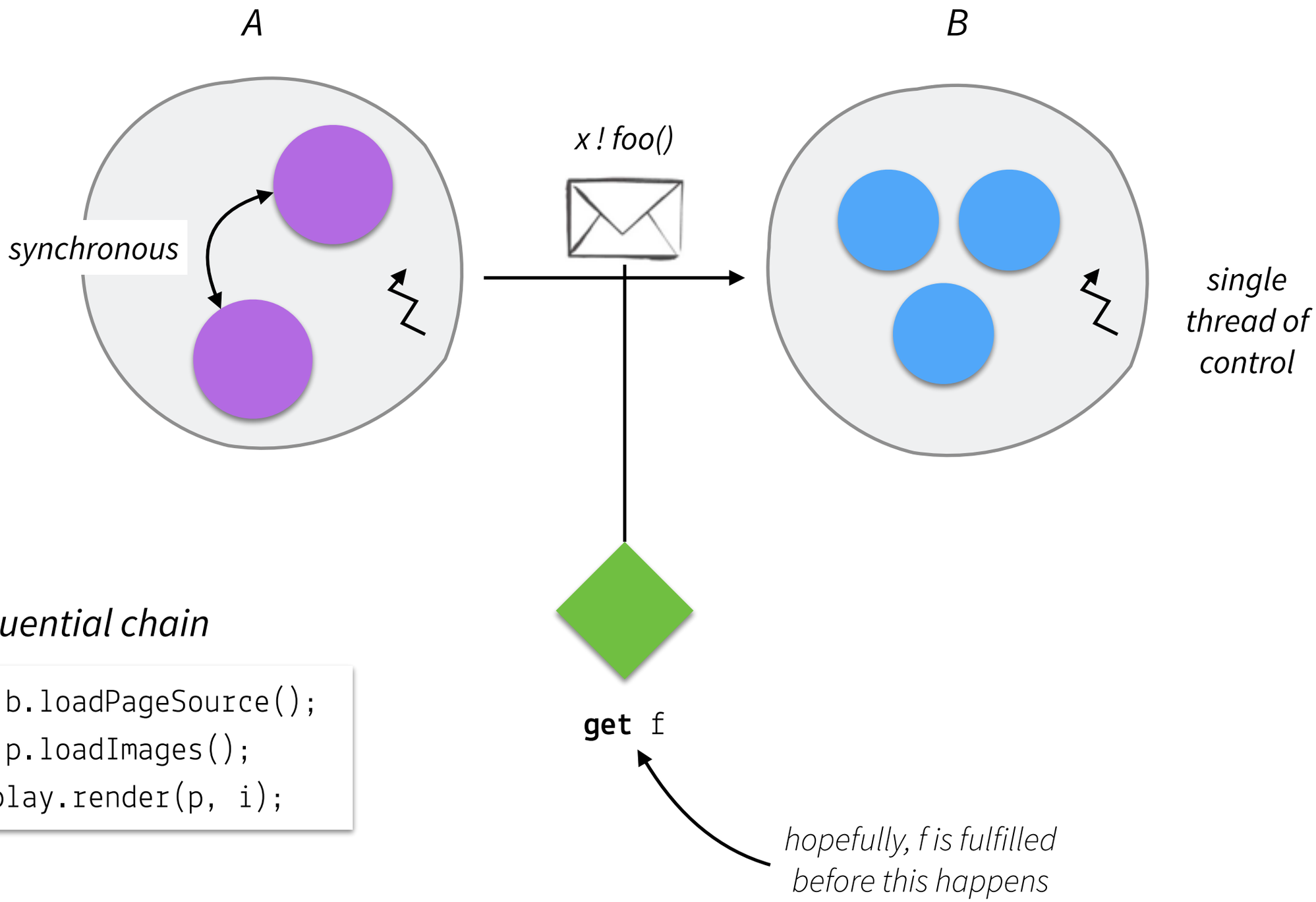
Accessing a future: get

```
get :: Fut t -> t
```

returns the value associated with a future, if available, otherwise blocks current active object until it is

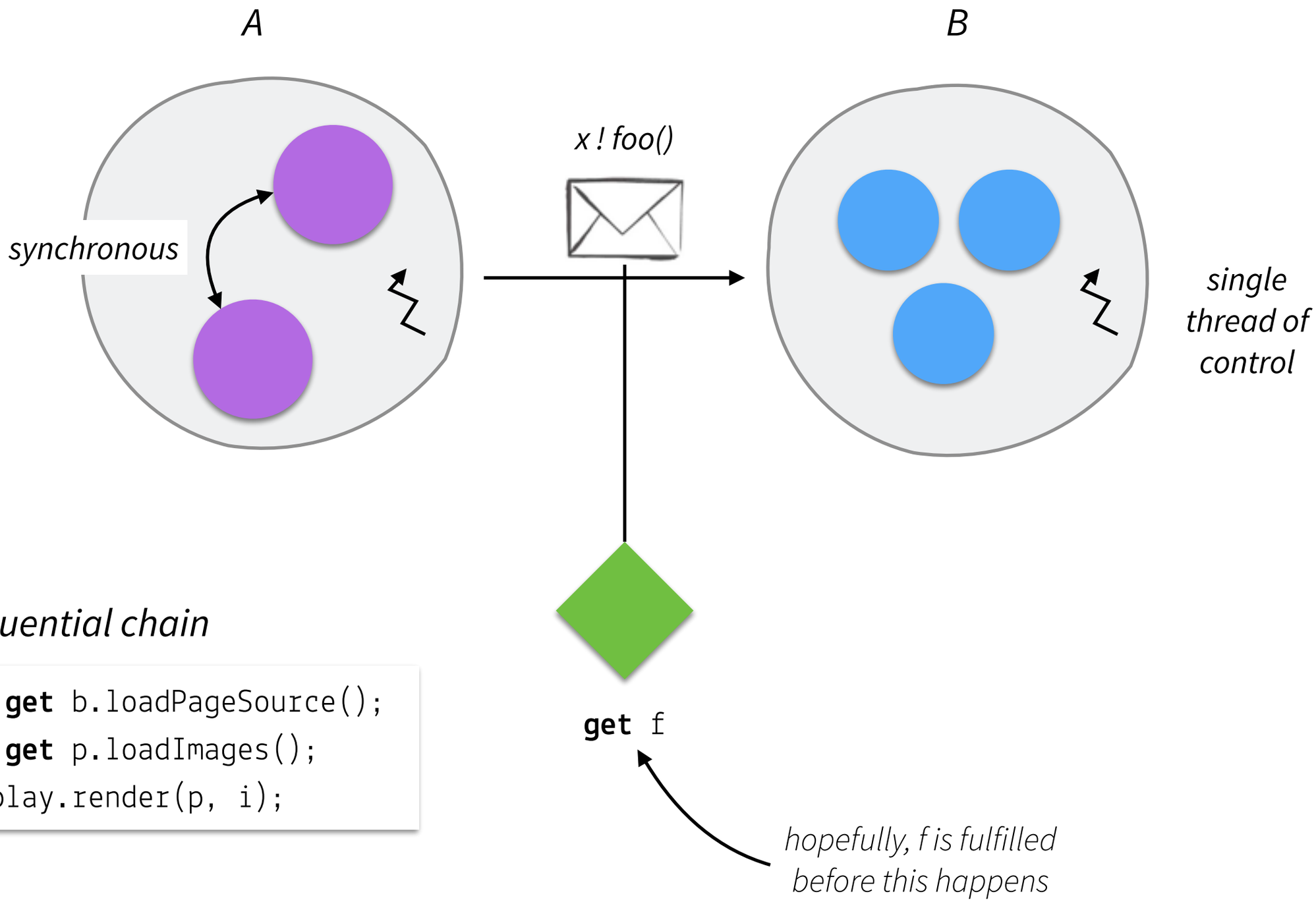
get immediately after a call ~ synchronous call


A
B
x ! foo()
synchronous
single thread of control
read from future
write return value



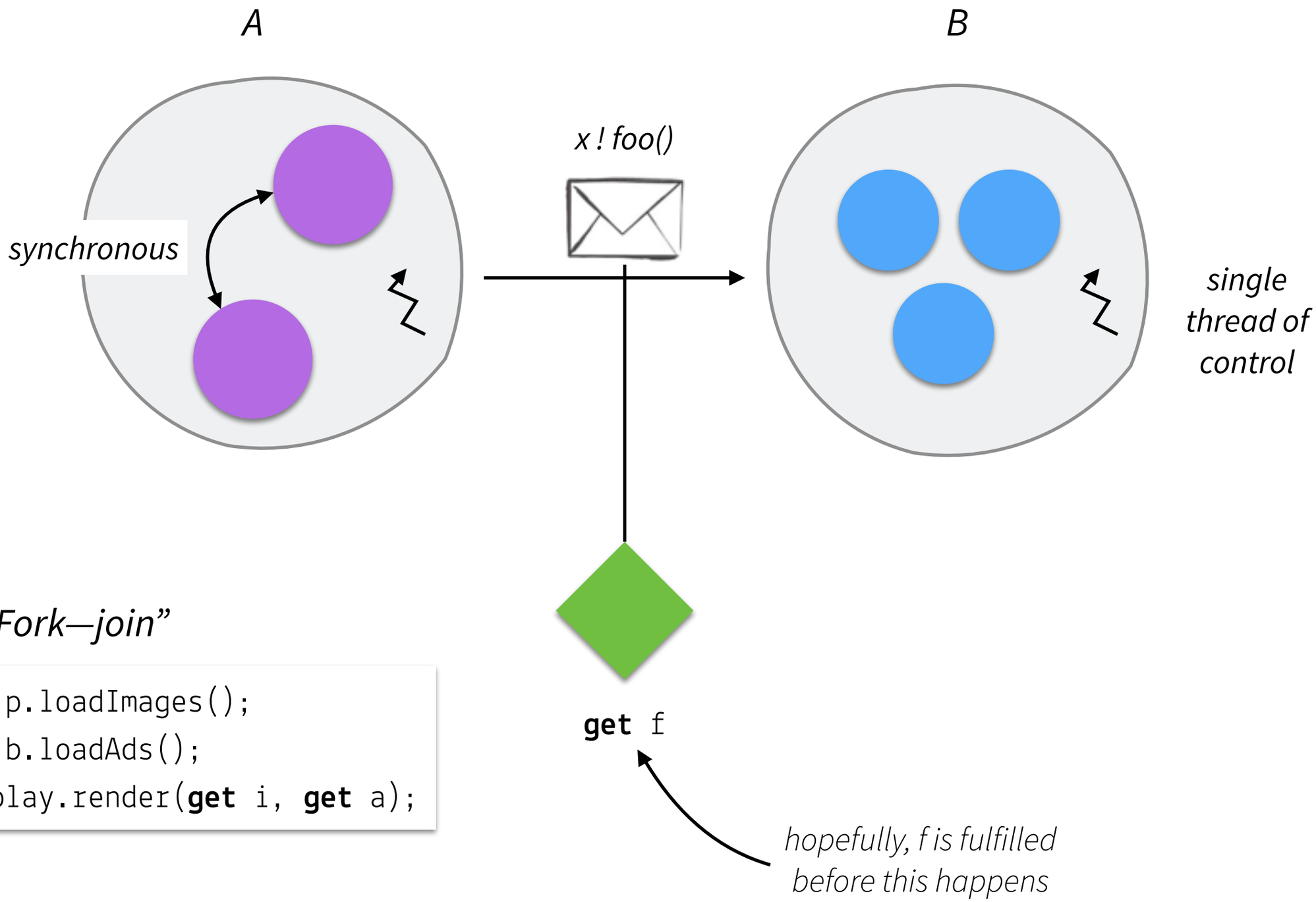
Sequential chain

```
p = b.loadPageSource();
i = p.loadImages();
display.render(p, i);
```



Sequential chain

```
p = get b.loadPageSource();
i = get p.loadImages();
display.render(p, i);
```



"Fork—join"

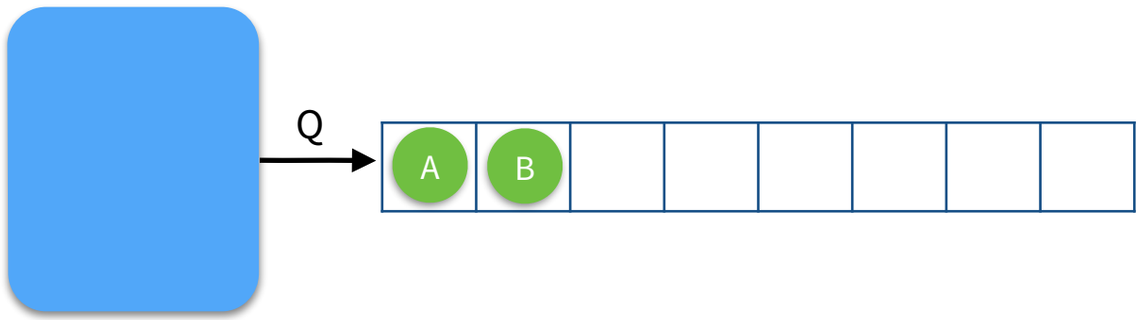
```
i = p.loadImages();
a = b.loadAds();
display.render(get i, get a);
```

Operations on Futures

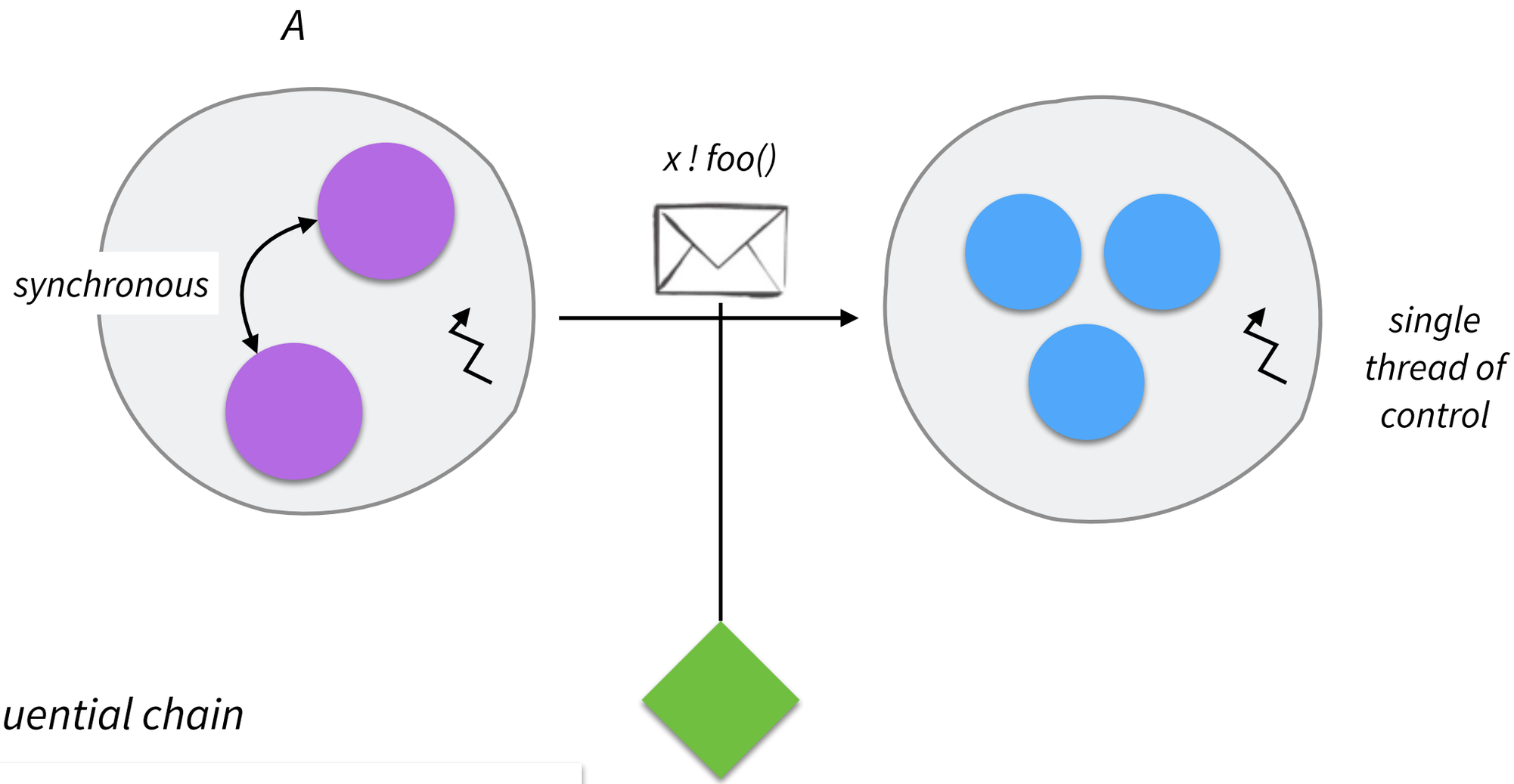
`await` `:: Fut t -> t`
– like **`get`**, but relinquishes control of the active object until a value in future is available, then returns that value

`poll` `:: Fut t -> Bool`
– checks whether the future has been fulfilled

+ chaining (next slide)



chain :: `Fut t -> (t -> t') -> Fut t'`
– apply a function *asynchronously* to the result of future, returning a future for the result

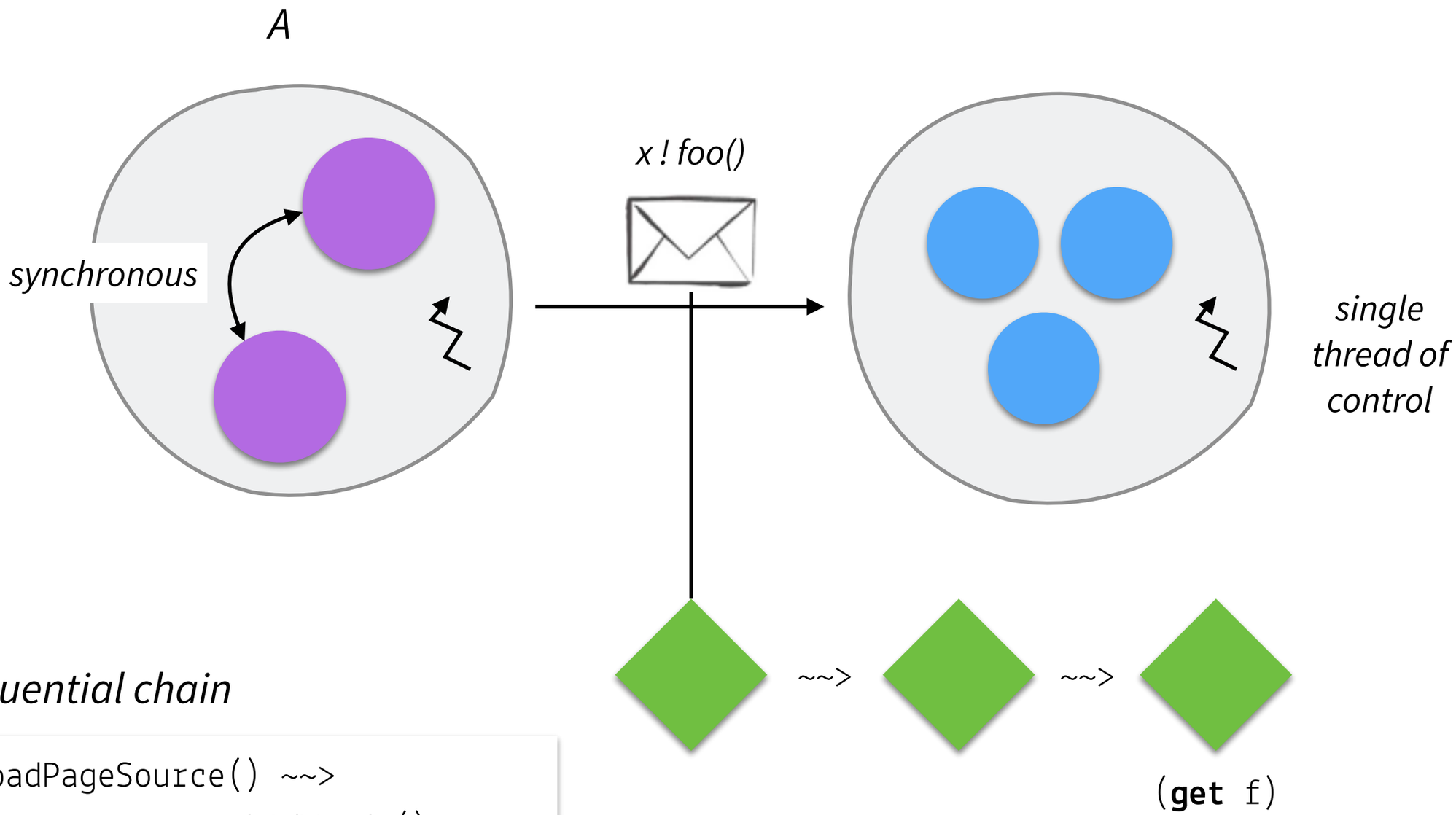


Sequential chain

```
b.loadPageSource() ~~>
   λ p -> p.searchAdWords() ~~>
       λ w -> getAds(w);
```

creates a "workflow" that is disconnected from A — avoids blocking A

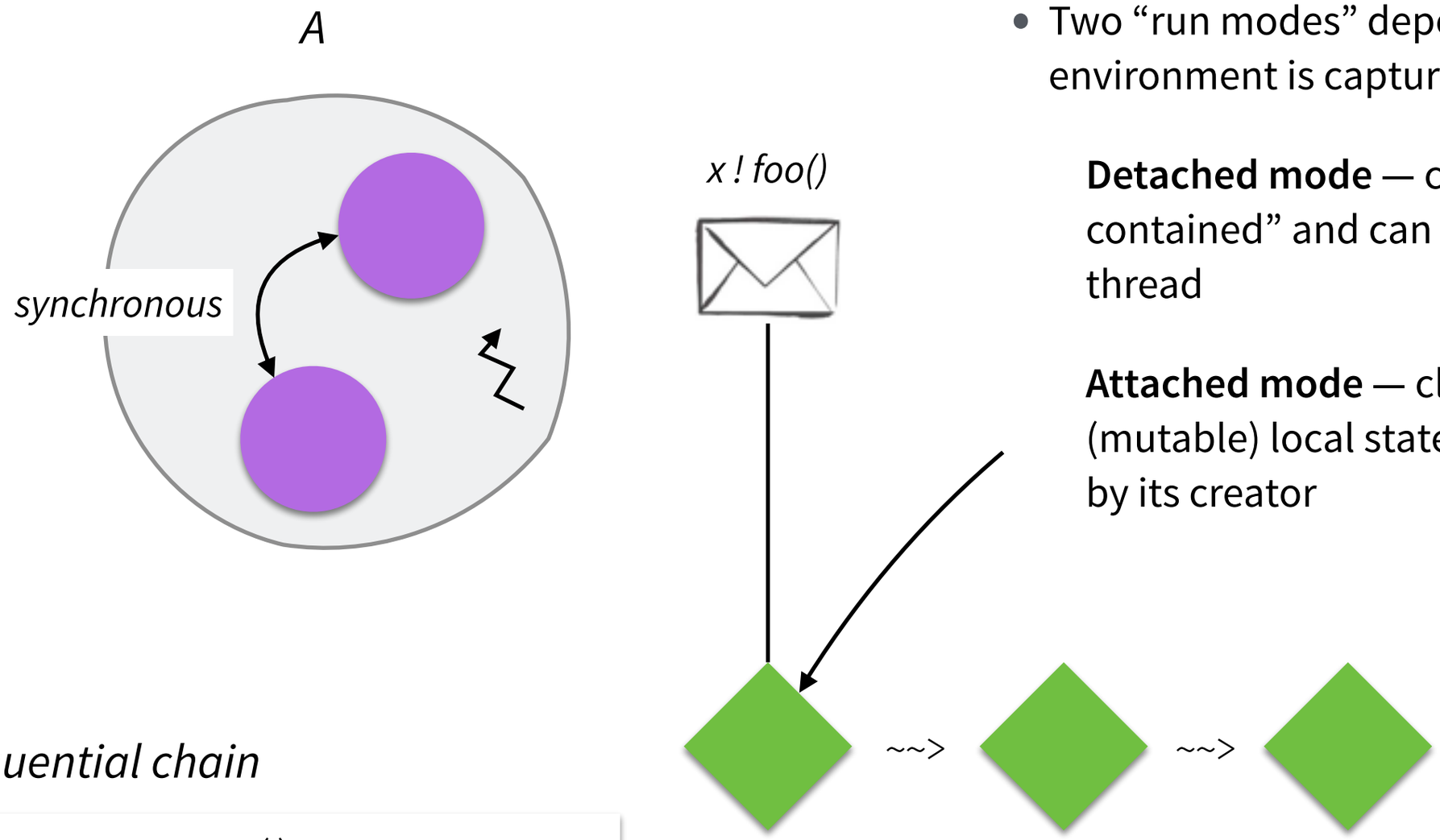
chain :: `Fut t -> (t -> t') -> Fut t'`
– apply a function *asynchronously* to the result of future, returning a future for the result



Sequential chain

```
b.loadPageSource() ~~>
   λ p -> p.searchAdWords() ~~>
       λ w -> getAds(w);
```

creates a "workflow" that is disconnected from A — avoids blocking A



- Two “run modes” depending on how environment is captured

  Detached mode — closure is “self-contained” and can be run by any thread

  Attached mode — closure captures (mutable) local state and must be run by its creator

Sequential chain

```
b.loadPageSource() ~~>
   λ p -> p.searchAdWords() ~~>
       λ w -> getAds(w);
```

creates a ”workflow” that is disconnected from A — avoids blocking A

Cooperative Multi-Tasking

- **`await`** (`Fut t -> t`) — like **`get`** but it relinquishes control of the active object to process another message (if there is one), if the future has not been fulfilled
- **`suspend`** relinquishes control of active object to process another message
- Both require active object to reestablish its class invariants *before* relinquishing control

Essentially the aliasing problem, but without the concurrency

Comparison

- **`get`** and **`await`** are costly as they require copying and storing the current calling context (stack), when the future has not been fulfilled
- **`chain`**ing is cheaper, but eventually a get is needed if you need the value

Data-race-free-by-Default and Isolation-by-Default

Passive Objects

Not all objects need their own (logical) thread of control

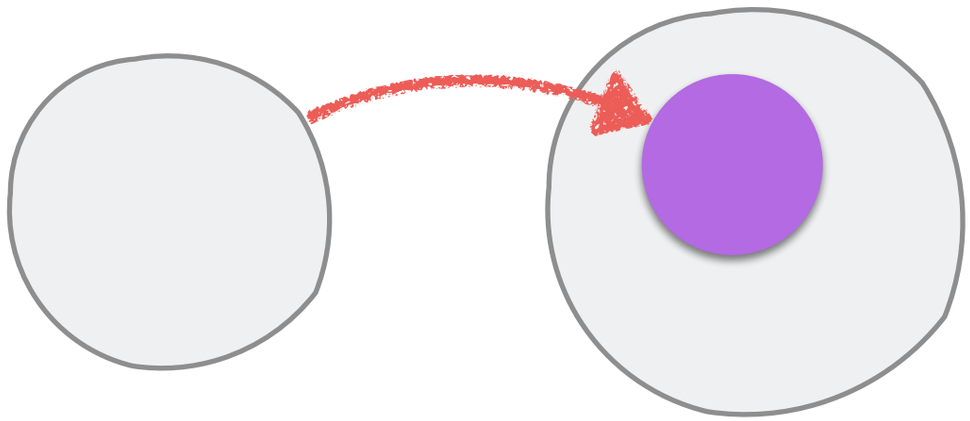
Synchronous communication, ”borrows” the thread of control of the caller

Sharing passive objects across active objects is unsafe, so must be isolated

Passive objects act as regular objects …

… without synchronisation overhead.

…possible to reason about how their state changes during an operation



Gradual Sharing?

1. Isolation (so trivially race-free)
2. Sharing, but sharing in race-free manner
3. Sharing with races

Explain DRF here

- Who controls race-freedom?

  Guaranteed by system (enforced at declaration-site)

  Guaranteed by programmer (enforced at use-site | not at all)

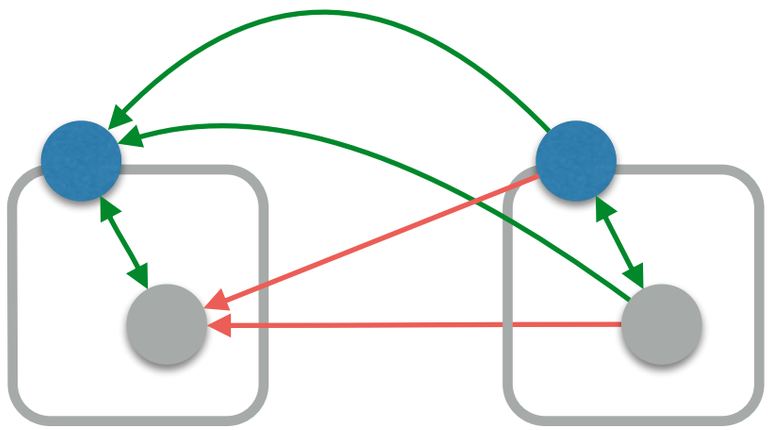
Basic Isolation

Fields can only be accessed by their active object.

But what about objects in fields?

Isolation by enforcing copying values across active objects

…by using powerful type system to enable transfer, cooperation, read-sharing, etc.



Benefits & Costs of Isolation

Benefits

Per Active Object GC — without synchronisation!

Single Thread of Control abstraction inside each active object

Costs

Cloning is expensive

No sharing of mutable state

Data-race Freedom

Data-race freedom is achieved because there is only one thread of control per active object

Fields and passive objects are only accessed by one thread, under the control of the active object's concurrency control

Thus no data races

Of course, DRF does not imply determinism

Order of messages in queues are non-deterministic

(Data)Parallel-by-Default

(Data)Parallel-by-default

Most languages are sequential by default, adding constructs for parallelism on top.

Encore explores parallel-by-default by integrating **parallel computation as a first-class entity**.

Parallel computations are manipulated by **parallel combinators**.

Work in progress

Futures are a handle on
one parallel computation.

Generalise to support
many parallel computations.

Parallel Types and Combinators

Parallel combinators express parallelism within an active object (and beyond)

Typed, higher-order, and functional — inspired by Haskell, Orc, LINQ, and others

Recall — **Fut t** = a handle to just one parallel computation

Par t = handle to parallel computation producing multiple **t**-typed values

Analogy: **Par t** ≈ [**Fut t**]

Except that **Par t** is an abstract type (don't want to rely on orderings, etc.)

Parallel Combinators: Interaction with Active Objects I

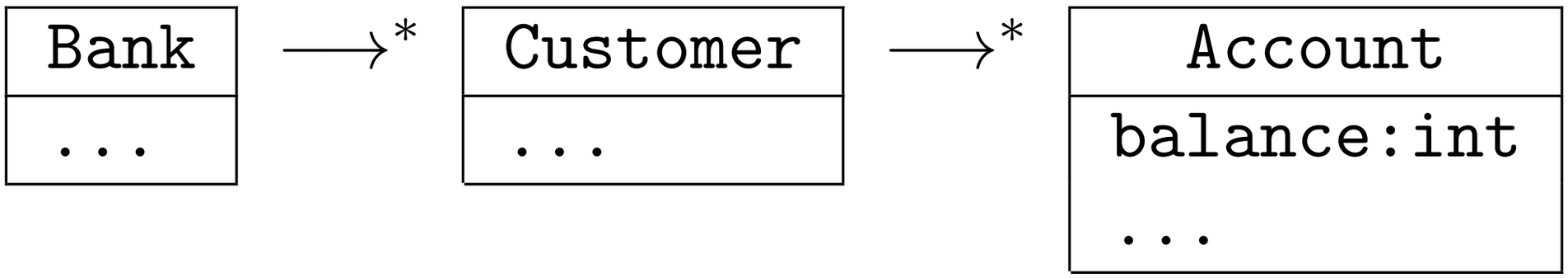
By analogy, `[o1.m1(), o2.m2(), o3.m3()] :: [Fut a]` is a parallel value

In Encore, `par(o1.m1(), o2.m2(), o3.m3()) :: Par a`

`each :: [a] -> Par a` — convert list into parallel value

Parallel Combinators: Interaction with Active Objects II

”Big variables” — multi-association between classes suggests parallelism




```
b.getCustomers() :: Par Customer
```

Parallel Combinators: Example

”Sum up the total value of all accounts in the bank with more than 9900 Euro”

```
class Main
  customers:Person*

  def main(): void
    let
      sum = this.customers . get_accounts . get_balance . (filter > 9900) . sum
    in
      print("Total: {}\n", sum)
```

Parallel Combinators: Example

”Sum up the total value of all accounts in the bank with more than 9900 Euro”

```
class Main
  customers:Person*

def main(): void
  this.customers
  ~~> bindp get_accounts                -- flatten accounts
  ~~> pmap get_balance                  -- get balance per account
  ~~> filter ( \ x:int -> x > 9900 )    -- filter accounts
  ~~> sum                               -- reduce operation
  ~~> ( \sum:int print("Total: {}\n", sum) )
```

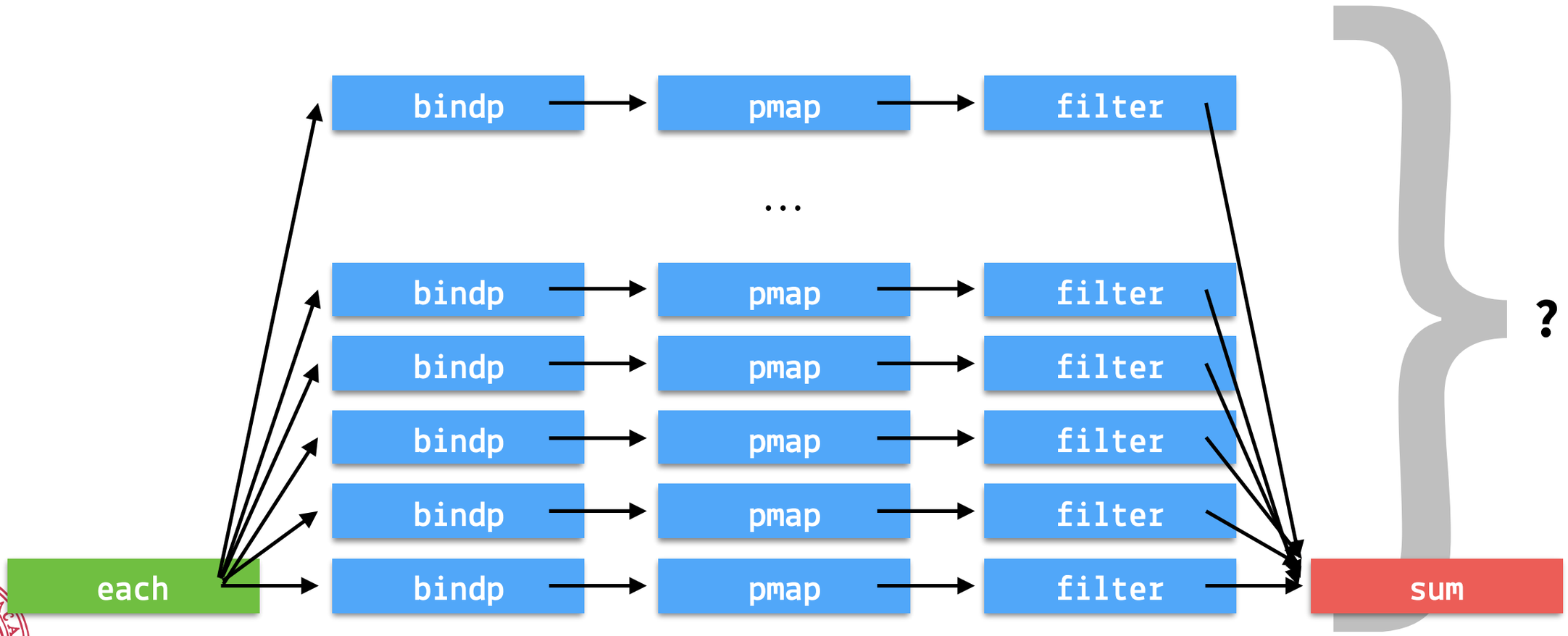
Parallel Combinators: Example

”Sum up the total value of all accounts in the bank with more than 9900 Euro”

```
class Main
  def main(): void
   let
      customers = get_customers()              -- get customers id
      par = each(customers)                    -- List t -> Par t
in
 {
   par = bindp(par, get_accounts);             -- flatten accounts
   par = pmap(par, get_balance);              -- get balance per account
   par = filter(par, \(x: int) -> { x > 9900 }); -- filter accounts
   print("Total: {}\n", sum(par));              -- reduce operation
 }
```

Parallel Combinators: Example

”Sum up the total value of all accounts in the bank with more than 9900 Euro”



Parallel Combinators (More Examples)

```
bindp :: Par a -> (a -> Par b) -> Par b
```

generalises monadic bind = map, then flatten

```
otherwise :: Par a -> (() -> Par a) -> Par a
```

if first parallel value is empty, return the value of the second argument

```
filter :: Par a -> (a -> Bool) -> Par a
```

keeps values matching predicate.

```
select :: Par a -> Fut (Maybe a)
```

returns the first finished result, if there is one.

```
selectAndKill :: Par a -> Maybe a
```

returns the first finished result, if there is one and **kills** all remaining

Parallel Combinators: From Parallel Types to Regular Values

Synchronisation

`sync :: Par t -> [t]` — synchronises a parallel value, giving list of results

Reduction

`sum :: Par Int -> Int` — performs parallel sum of result of parallel integer-valued computation

Many such functions exist.

Parallel Combinators: Challenges

- Integration with OO fragment

  Capabilities handle race conditions — ”if you have a reference, you can use it fully”

- Optimisation

  Parallel semantics by default opens door to many optimisations and scheduling strategies

- Program Methodology

  Case studies shall reveal design patterns for using parallel combinators and active objects in unison

Unique-by-default

SFM Summer School

Bertinoro, June, 2015

Alias Freedom is a Strong and Useful Property

- Strong updates

  Change type of object (e.g., typestate, verification)

- Optimisations

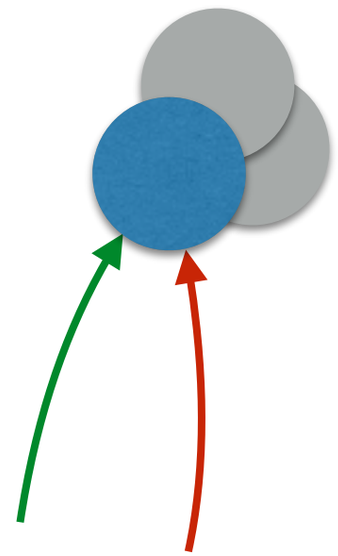
  Explode the object into registers, no need to synch with main memory

- Reasoning

  Sequential reasoning, pre/postconditions, no need for taking locks

- Ownership transfer

  E.g. enable object transfer through pointer swizzle



- Mainstream OOPLs make sharing default

  Benefit: keeps things simple for the programmer (cf. Rust)

  Price: hard to establish (and maintain) actual uniqueness

- Analysis of object-oriented code shows that:

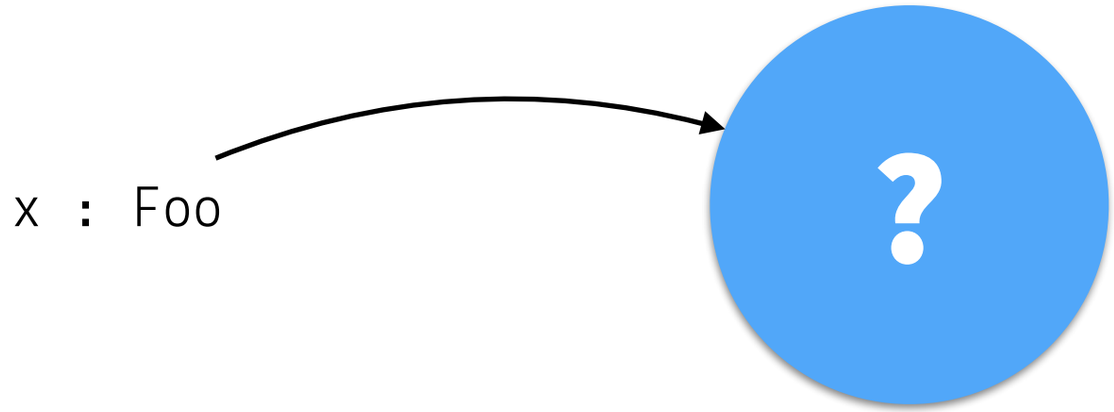
  Most variables are never null

  Most objects are not shared across threads

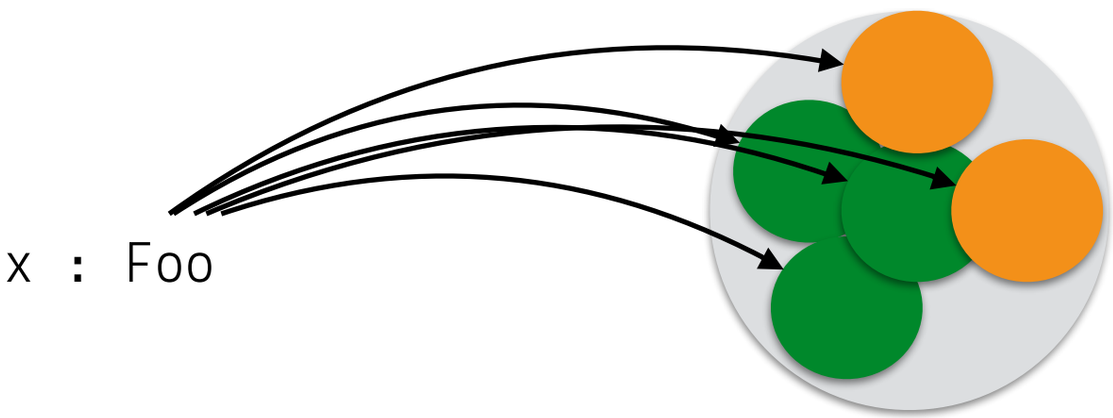
  Most objects are not aliased on the heap

  However — most mainstream programming languages do not capture that

Normal OOP



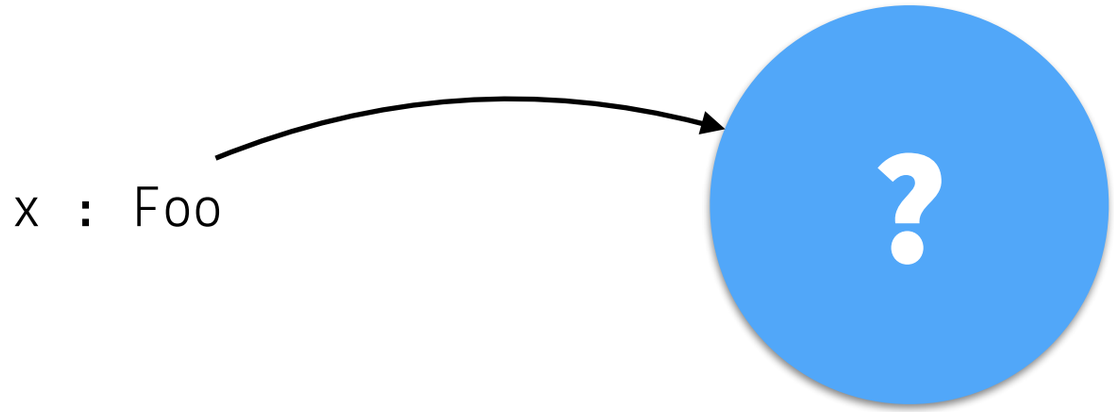
Encore



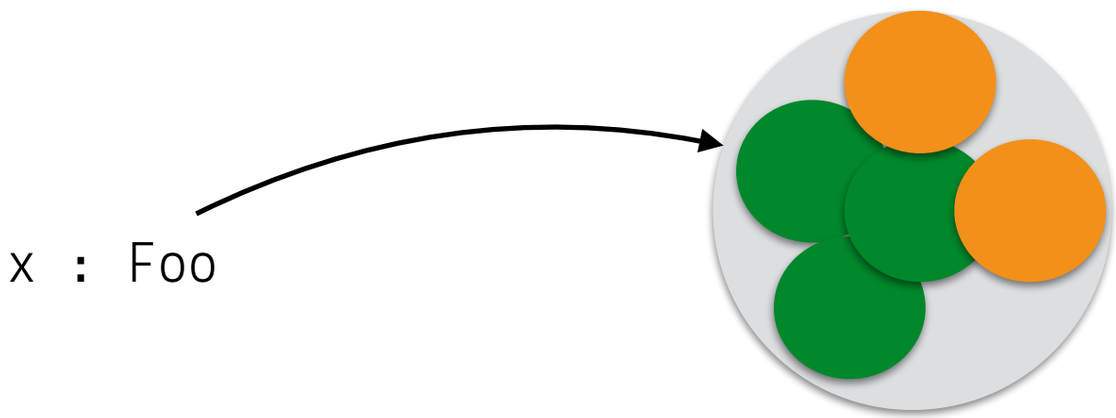
Exclusive

Safe

Normal OOP



Encore

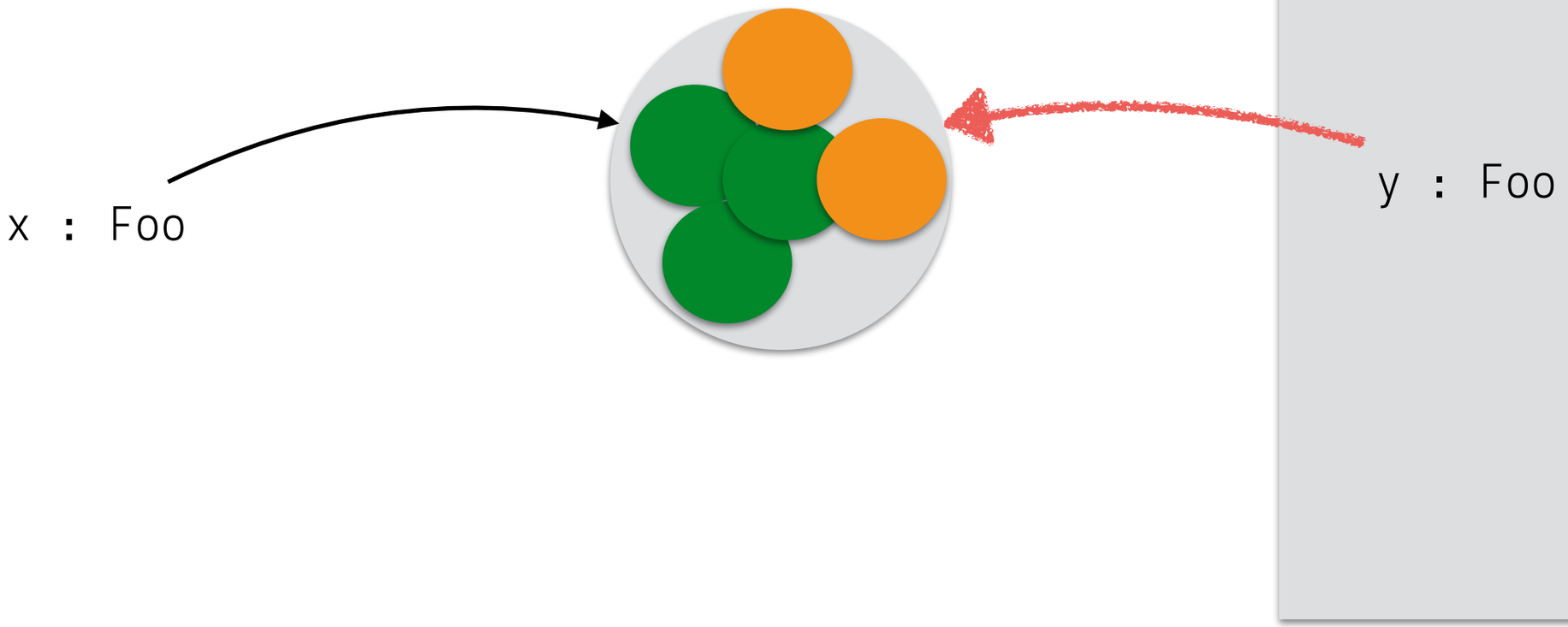


Normal OOP

Separate Thread

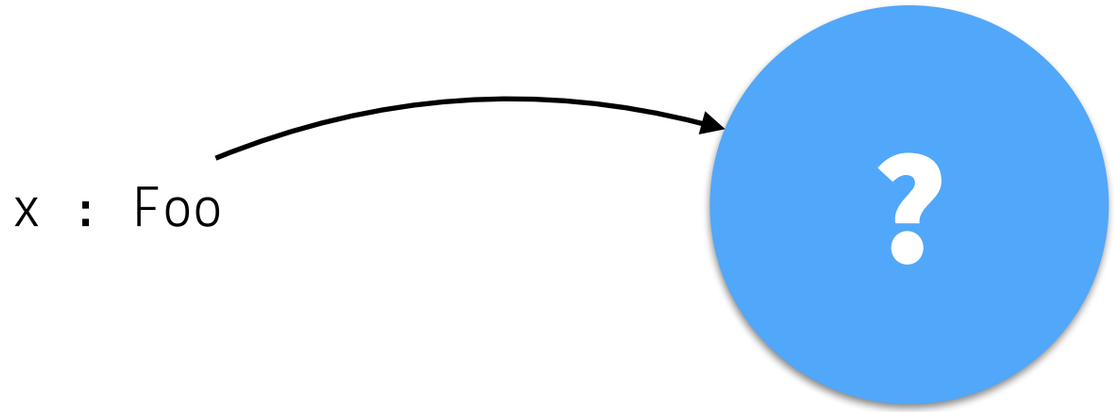
Encore

Separate Thread or Active Obj.



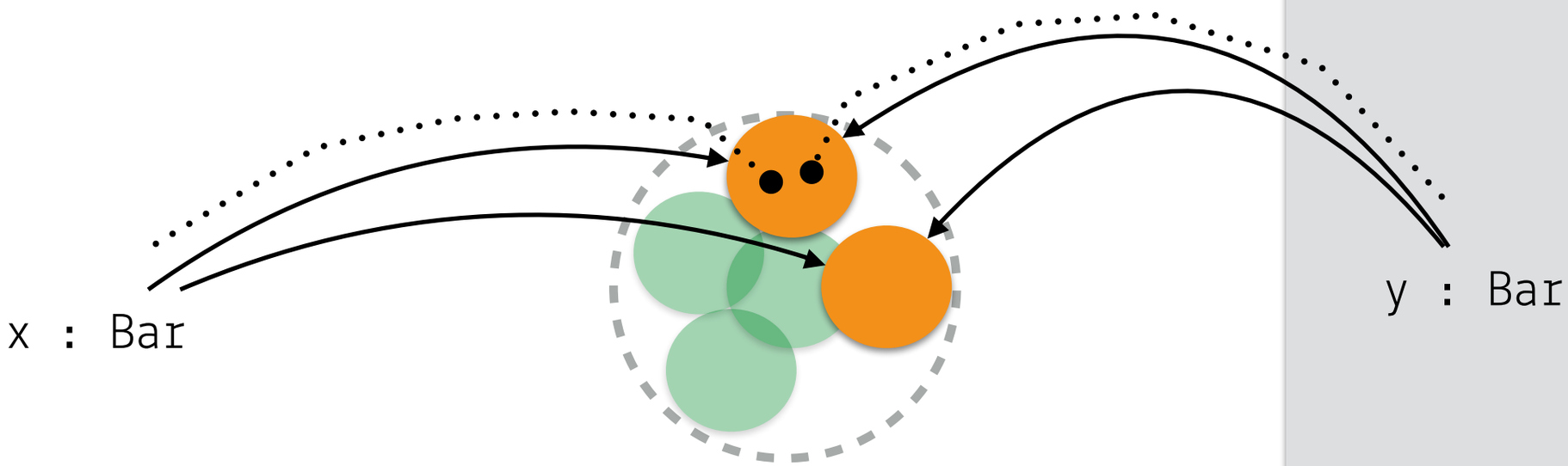
Normal OOP

Separate Thread

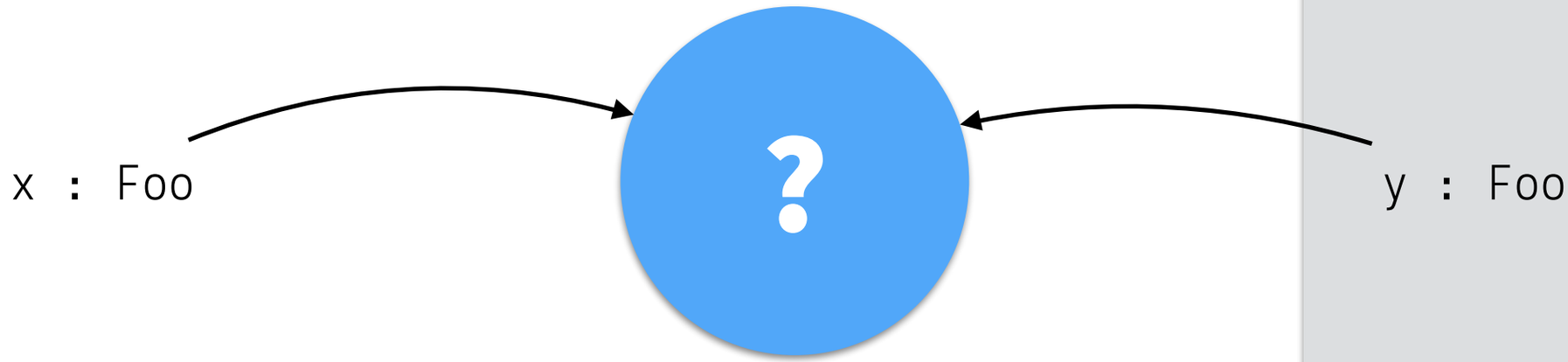


Encore

Separate Thread or Active Obj.

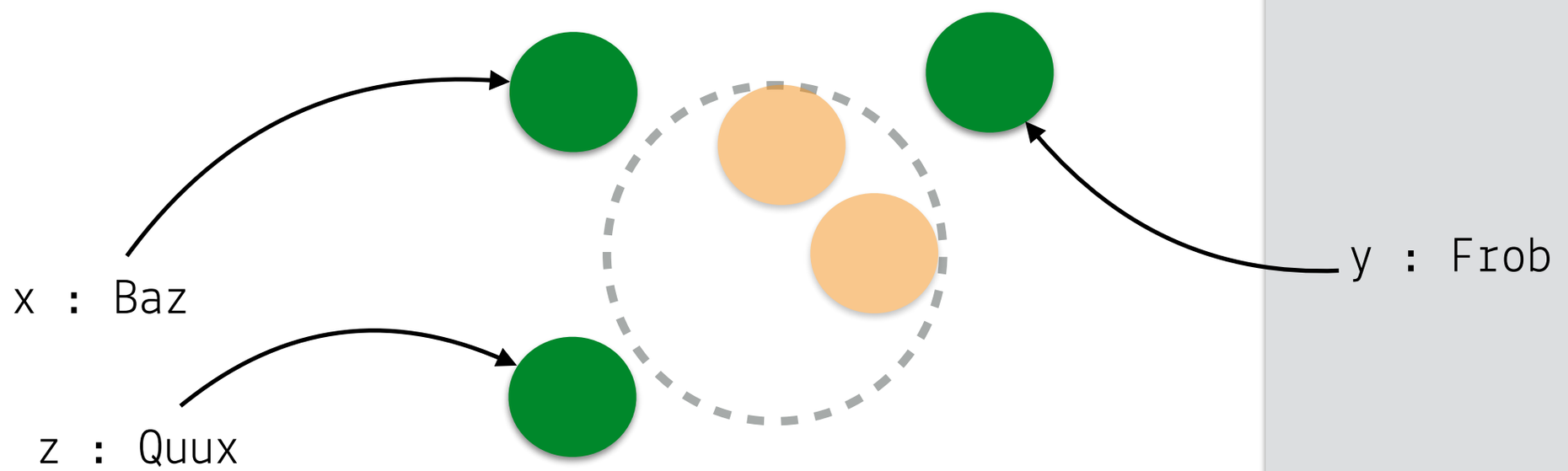


Normal OOP



Separate Thread

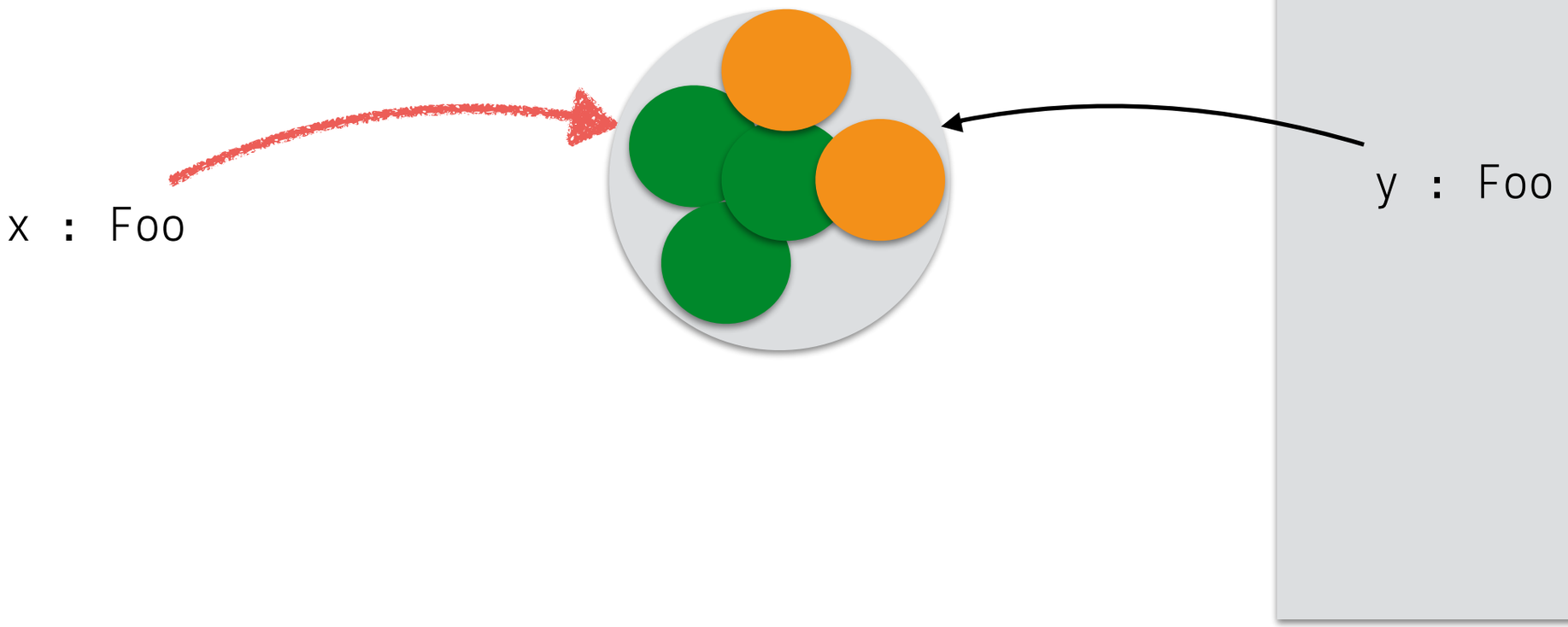
Encore



Separate Thread or Active Obj.

Normal OOP

Encore



Weak pair → `class Pair = Cell ⊗ Cell { … }`

Strong pair → `class Pair = Cell ⊕ Cell { … }`

Two-faced Stream

```
linear trait Put {
 def yield(Object o) : void …
}
```

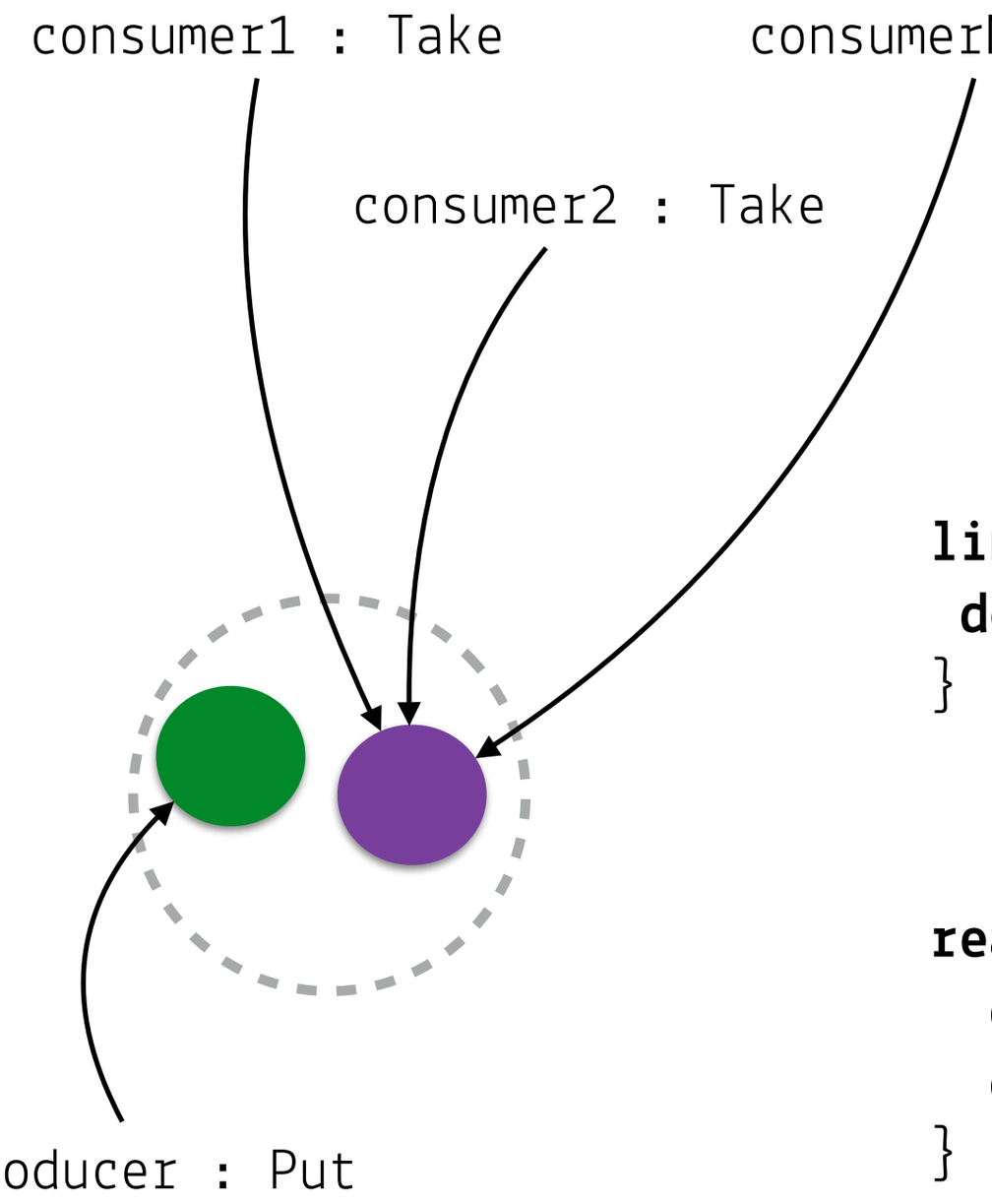
Linear

```
readonly trait Take {
  def read() : Object …
  def next() : Take …
}
```

ReadOnly

```
class TwoFacedStream = Put ⊗ Take { … }
```

(SPMCQ)

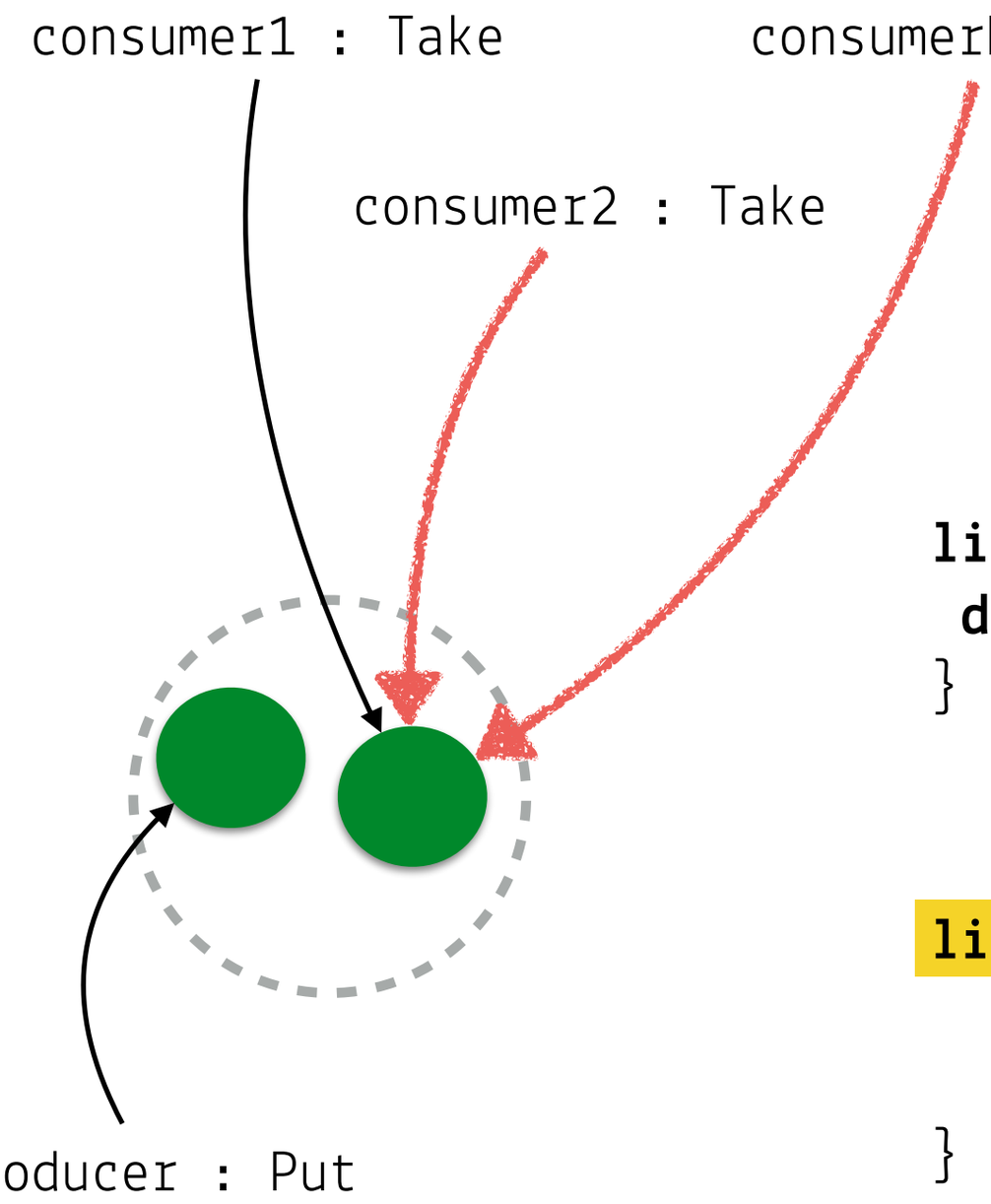



```
linear trait Put {
 def yield(Object o) : void …
}
```

```
readonly trait Take {
  def read() : Object …
  def next() : Take …
}
```

```
class TwoFacedStream = Put ⊗ Take { … }
```

(SPSCQ)

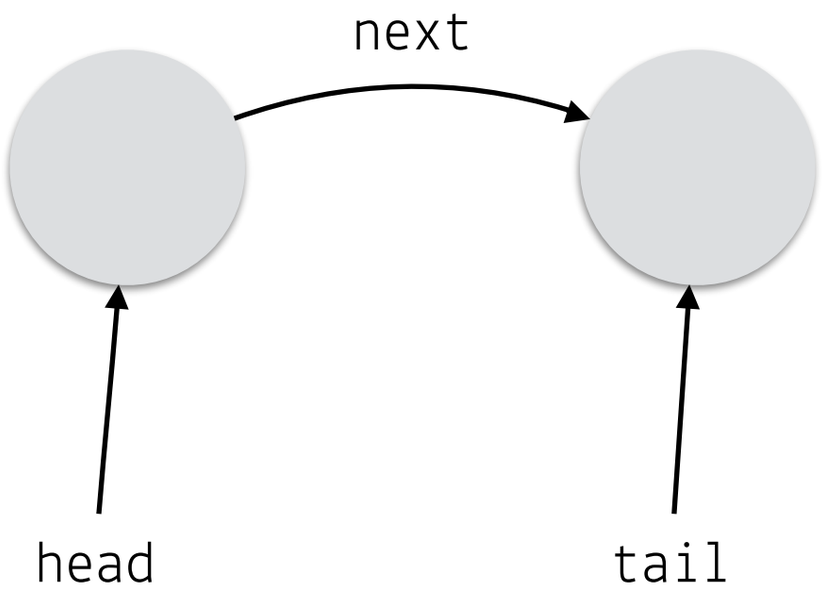



```
linear trait Put {
 def yield(Object o) : void …
}
```

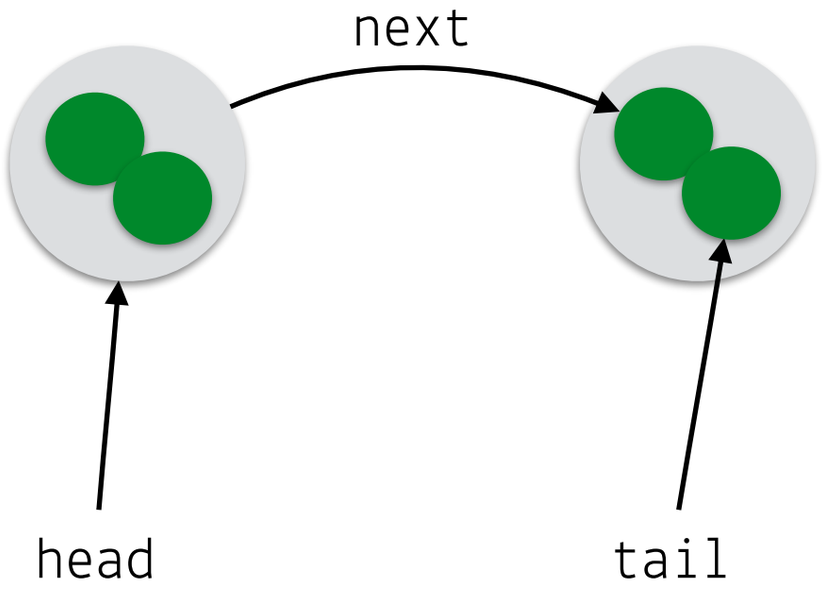
```
linear trait Take {
  def read() : Object …
  def next() : Take …
}
```

```
class TwoFacedStream = Put ⊗ Take { … }
```

Not All Aliasing is Evil

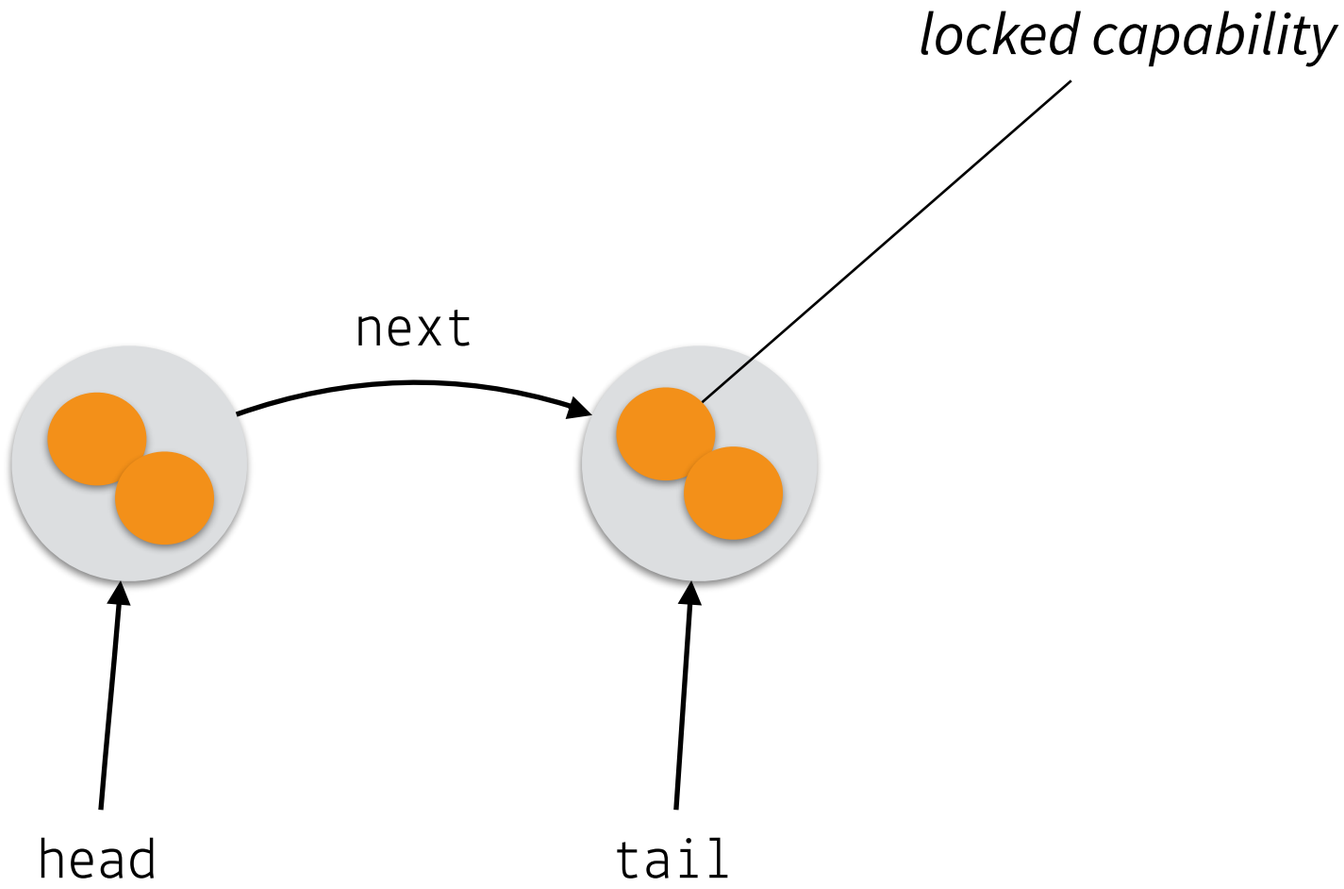


Not All Aliasing is Evil



Possibility 1: `next` and `tail` reference different parts of the object

Not All Aliasing is Evil

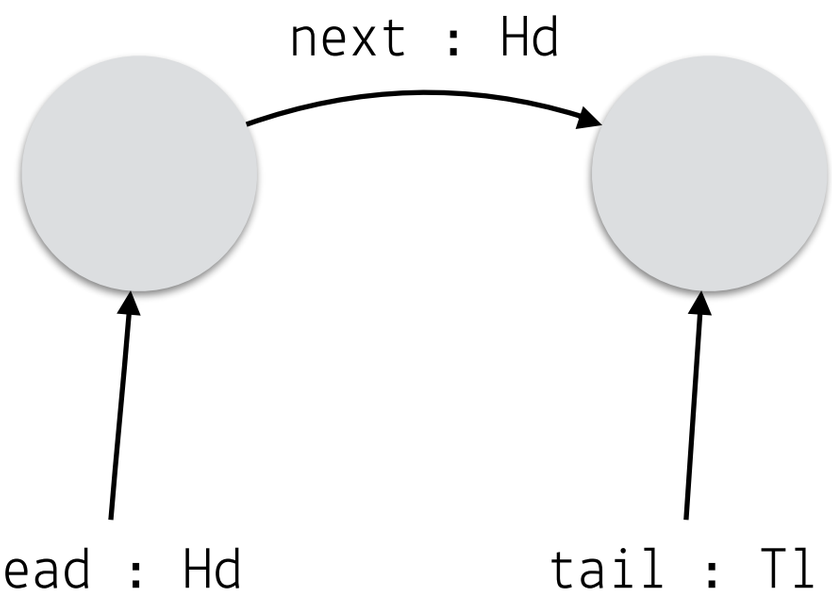


Possibility 2: list is constructed from parts that may be freely aliased

Not All Aliasing is Evil

`Link = Hd ∨ Tl`

Programmer may only dereference `Hd` *or* `Tl`, never both



Possibility 3: introduce aliasing in a tractable way

```
if head != tail
then tail ∨ tail.next = new Link(…)
else …
```

Unique-as-Default

- Slightly more tricky programming

  Intentional sharing incurs syntactic cost, becomes clearly visible

  Need to work harder in some cases to maintain uniqueness

- Sometimes, type system is not strong enough to track uniqueness

  Thread-locality gives many similar guarantees modulo transfer

  Use capabilities that protect against data races

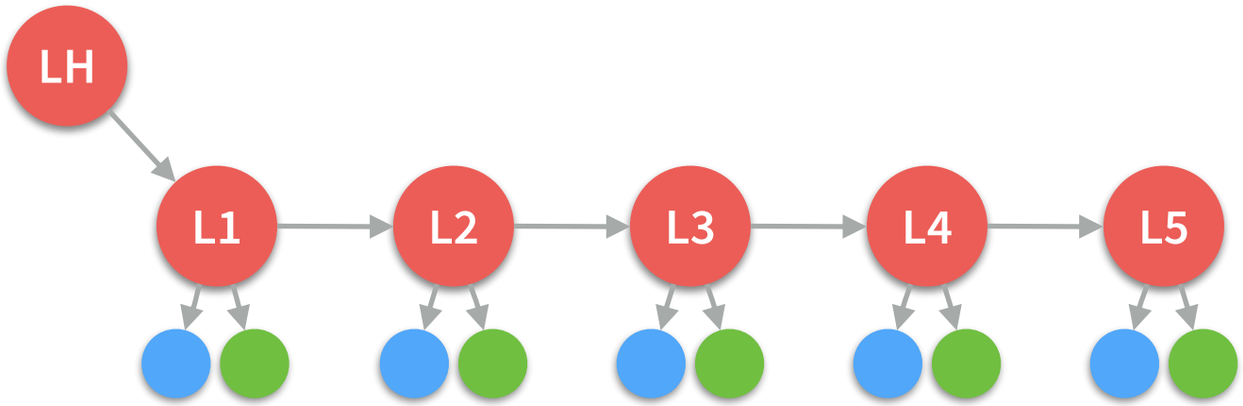
  Will be revisited in the talk on ownership types soon

Locality-by-default

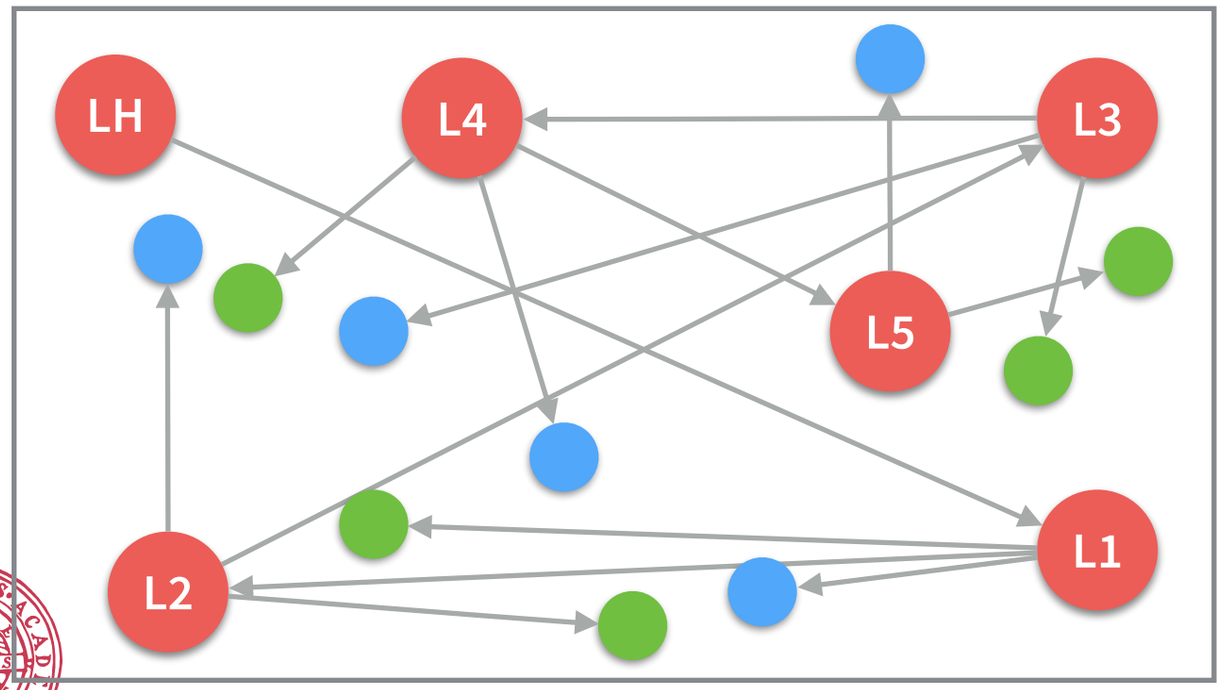
SFM Summer School

Bertinoro, June, 2015

Encore Memory Management

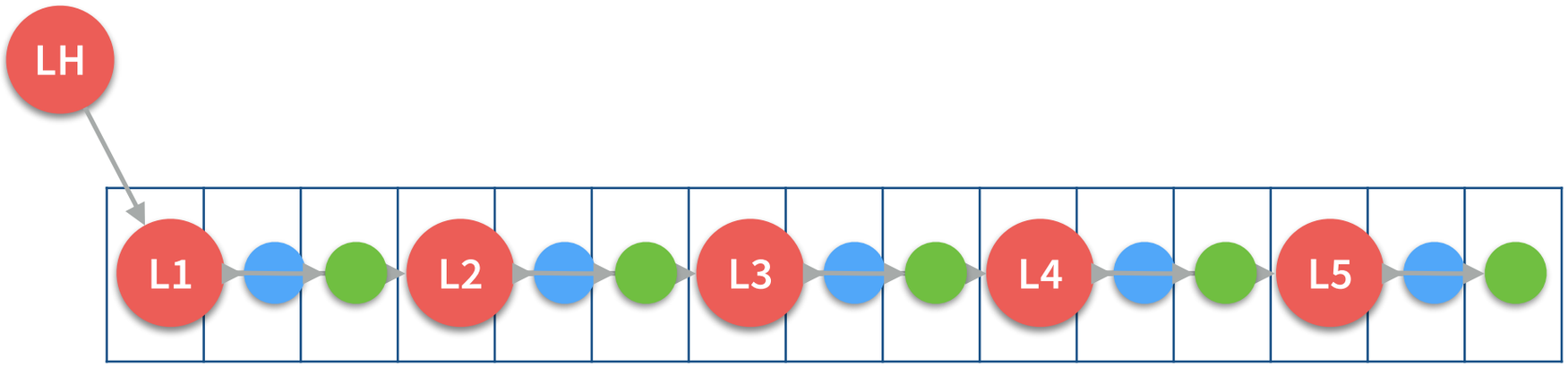


Programmer's mind



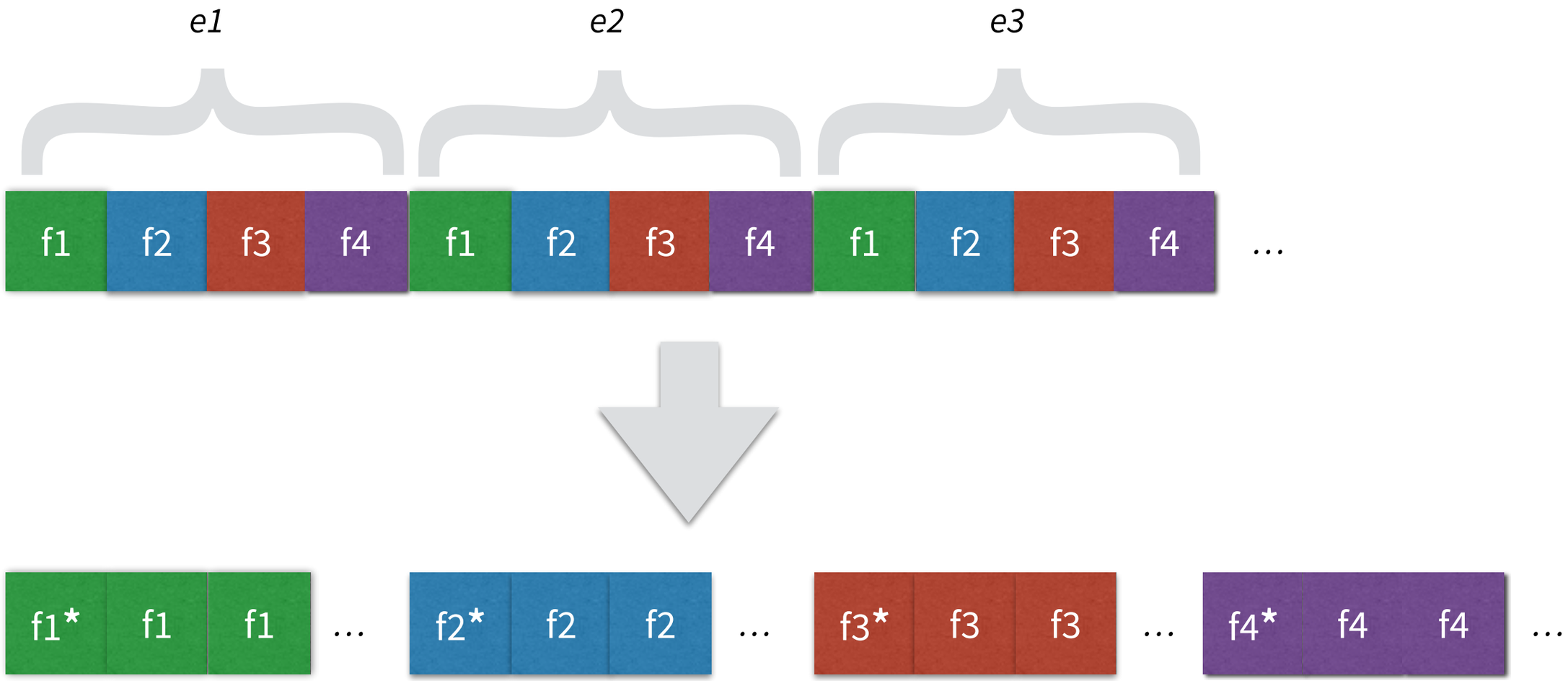
Reality

Encore Memory Management



Projecting the list onto an array

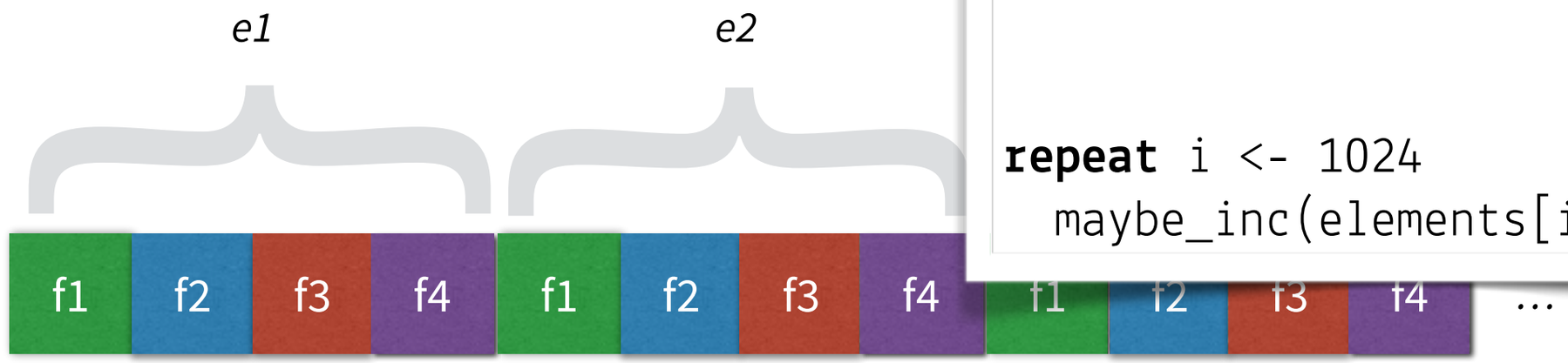
Problem: Bad Memory Efficiency

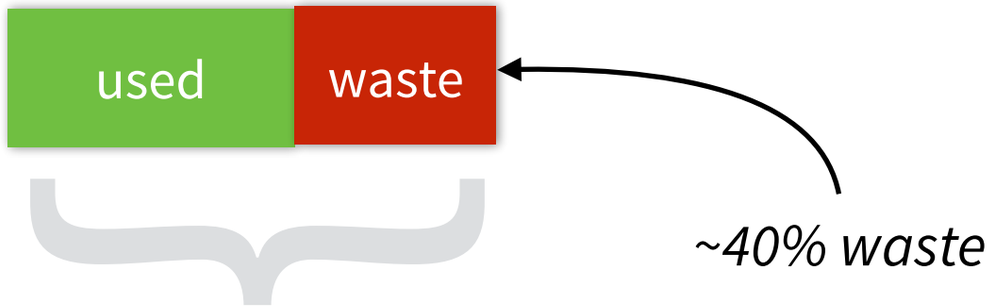


* = *aligned with cache line start*

cache line size

```
def maybe_inc(e:element) : void
  if (e.f1) e.f2++


repeat i <- 1024
  maybe_inc(elements[i])
```





cache line size

1024 accesses

Assume e not in cache, cost of e.f1 ≈ 100 cycles

Access e.f2 will be a hit, cost ≈ 1 cycle

= 102400 units = **41370 units of waste**

Each turn in the loop will stall!

(modulo misalignment and prefetching)

```
def maybe_inc(e:element) : void
  if (e.f1) e.f2++


repeat i <- 1024
  maybe_inc(elements[i])
```

used (100%)

used (100%)

never loaded!

never loaded!

first e.f1 access

first e.f2 access

1024 accesses

First access to e.f1 a miss ≈ 100 cycles

2 subsequent items hits ≈ 2 cycles

As soon as we have more than

~0% waste

At most 1/3 elements will stall

40% fewer memory accesses — **faster program!**

cache line size

Encore Memory Management

- *Locality–by–default*

  Allocate objects building up large structures from the same memory pool

  Locality requires different placement strategy for different data structures (e.g., hierarchical for trees, linear for linked lists)

- *Structure splitting*

  Especially good for performing many similar operations on part of a big structure (e.g., column-wise accesses, vectorisation)

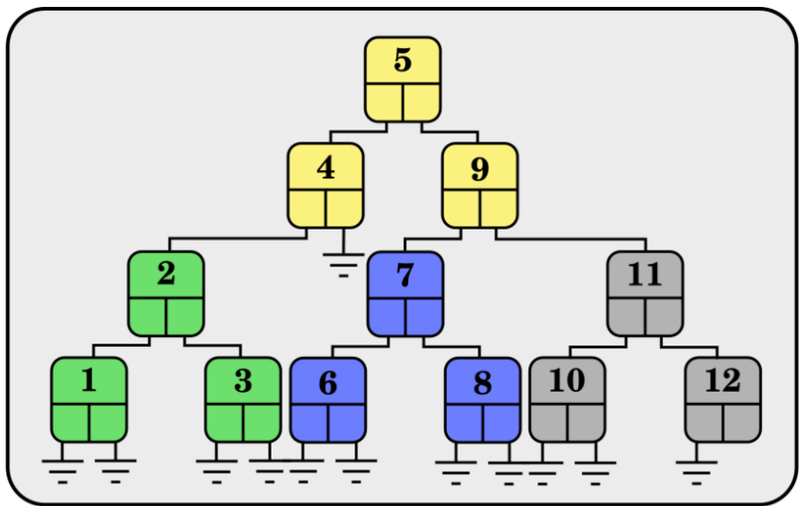
  ”Small updates” may cause more writes to disjoint locations = more invalidation, i.e., not a silver bullet

  ”Maximal splitting” seems to work well in the general case, but grouping certain substructures may be an optimisation

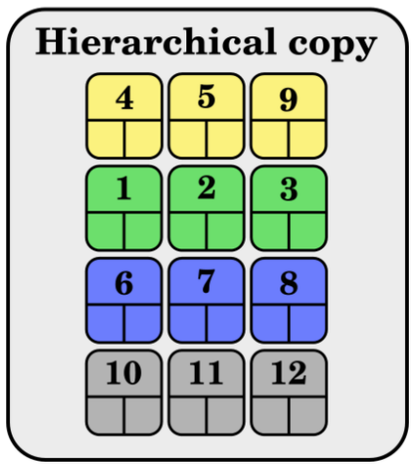
Ordering Data in Pools

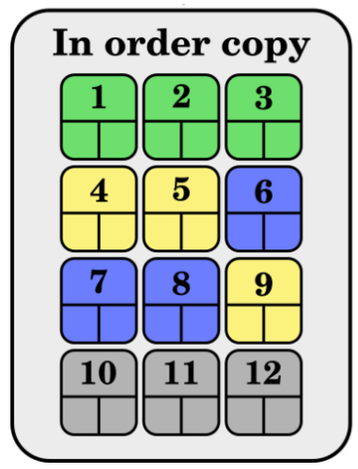
- Linked/array-like structures are simple to organise in memory
- No ”best” organisation strategy — dependent on data structure definition and use

  For example, consider a binary tree

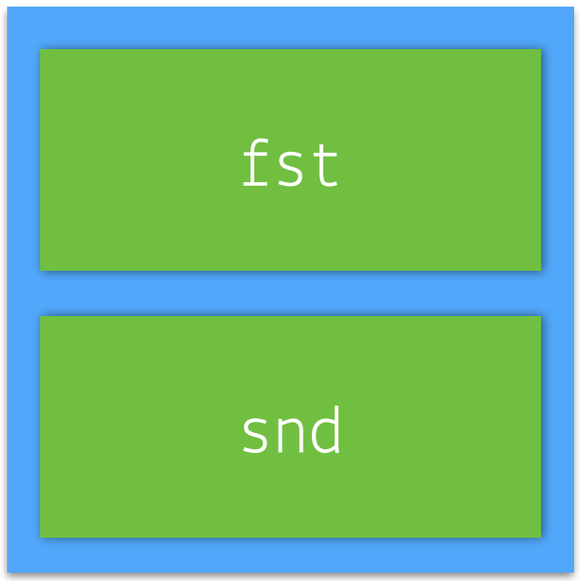


Which one is best?



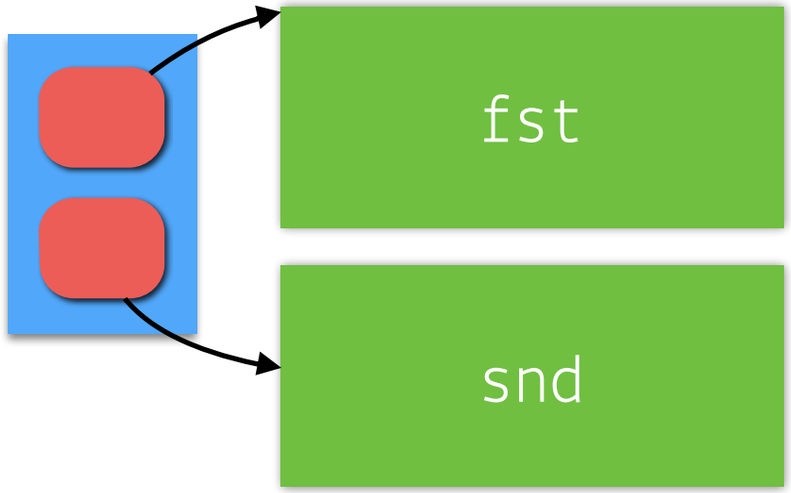


Data Representation (of a simple pair)

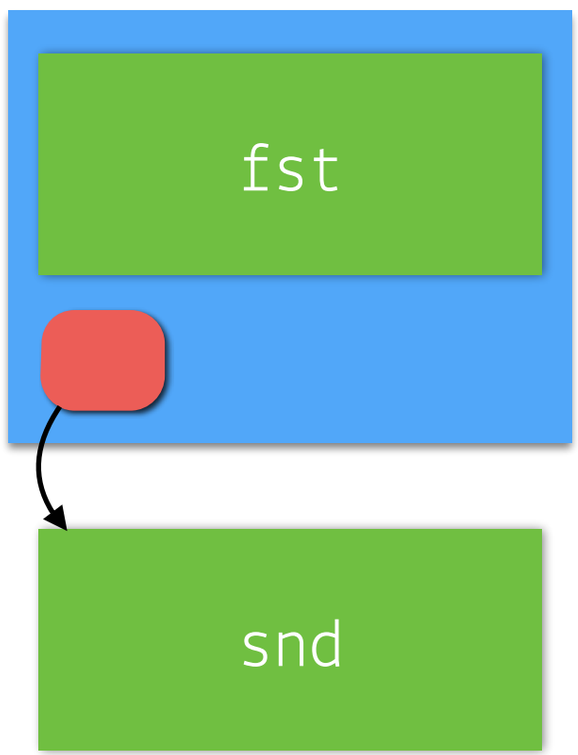


Embed both

Embed none



Embed one



Externalise: *Make it possible to change between these possibilities at use-site, without touching the ”business logic” of the pair*

Linked Pools

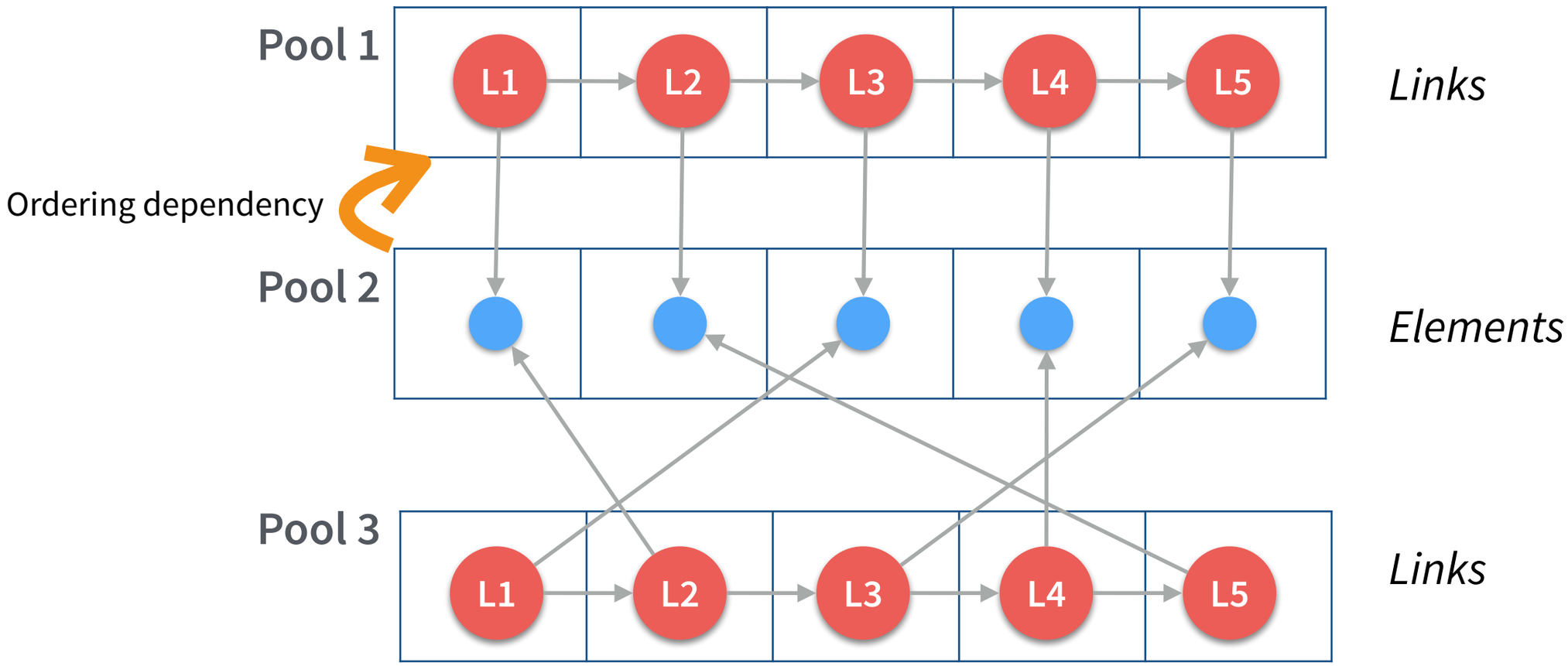
- There is a deception in the linked list example: commonly, the list does not **embed** its elements, rather they are stored in the list by pointer only

  If element objects are spread across more than one pool, little is accomplished

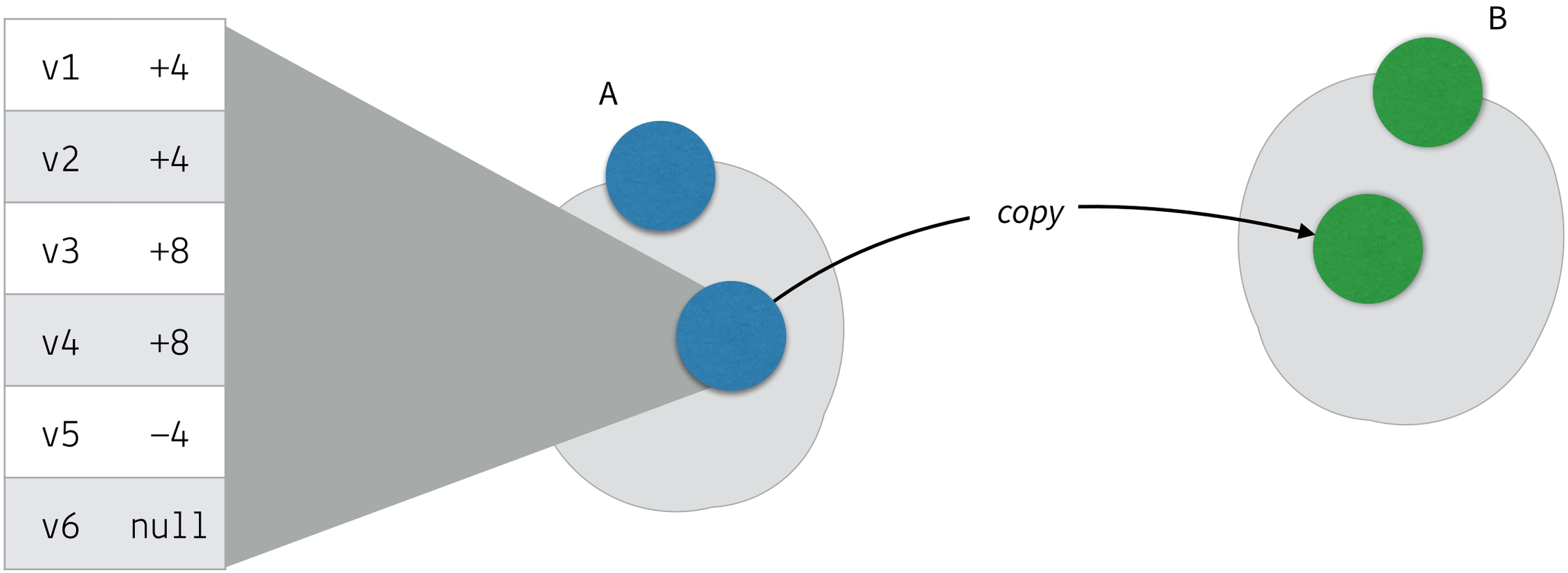
  If element objects are mixed with link objects, less locality

  Optimal case: element objects in a single pool (modulo splitting) and order in element pool is linked to the order in the link pool

Linked Pool Example



The Case for Object-Relative Addressing

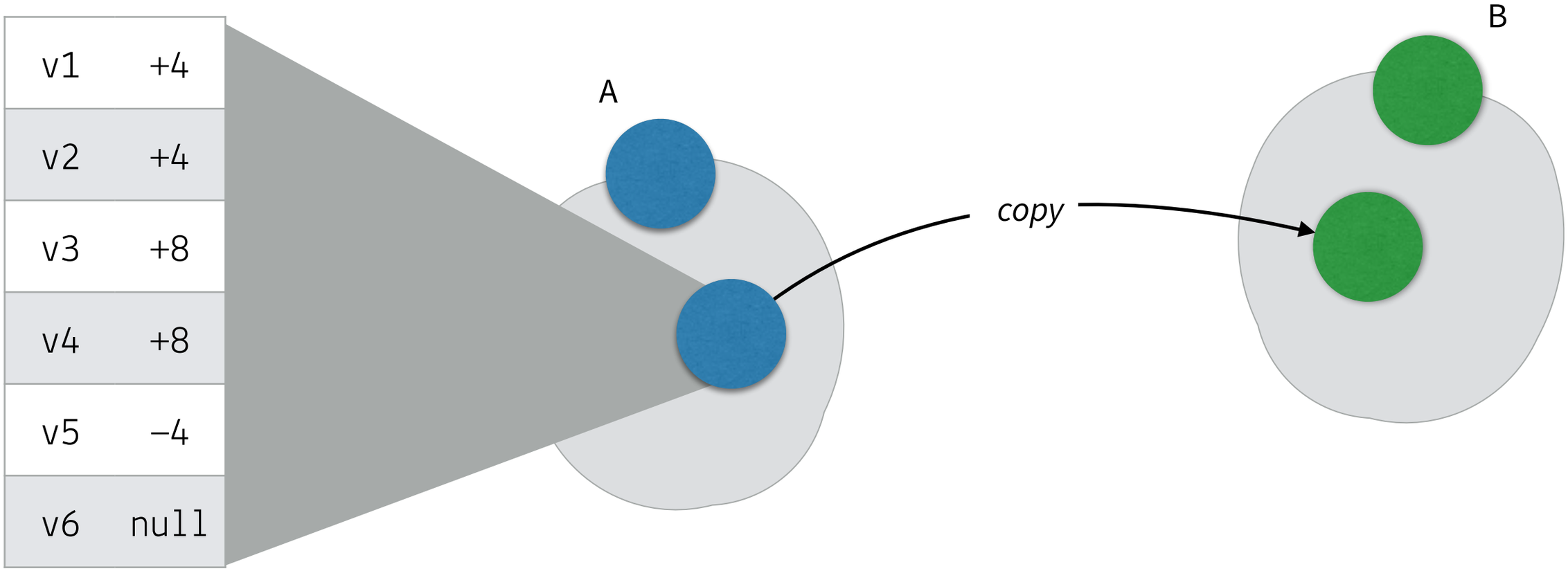


Copy:

All object-relative addresses on A's heap are valid when copied to B's heap.

Hence, copying N links can be reduced to a "memcpy" of start–end addresses.

The Case for Object-Relative Addressing



Example win:

Can fit 32 pointers in a single cache line as opposed to 8 — can store many small subtrees in a single cache line in the tree hierarchy example

Exercises

Design Exercises

"Implement" the system described in the handout using ideas from Encore.

Which objects should be active? Which passive?

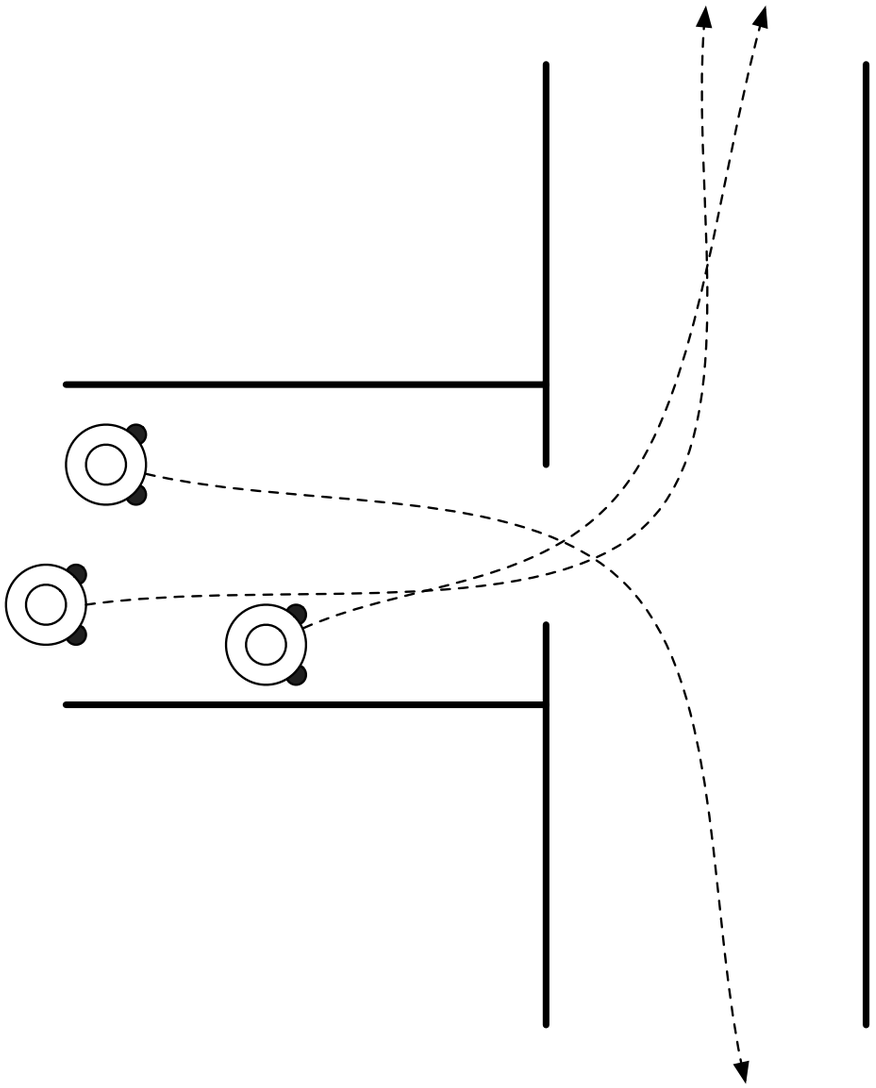
How is data distributed among the active objects?

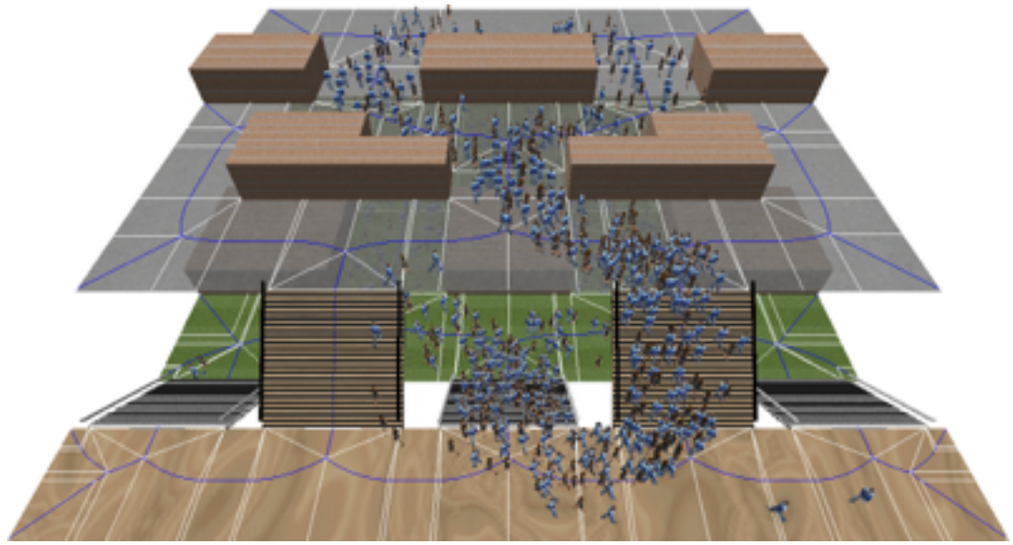
What is the amount of data passed between active objects?

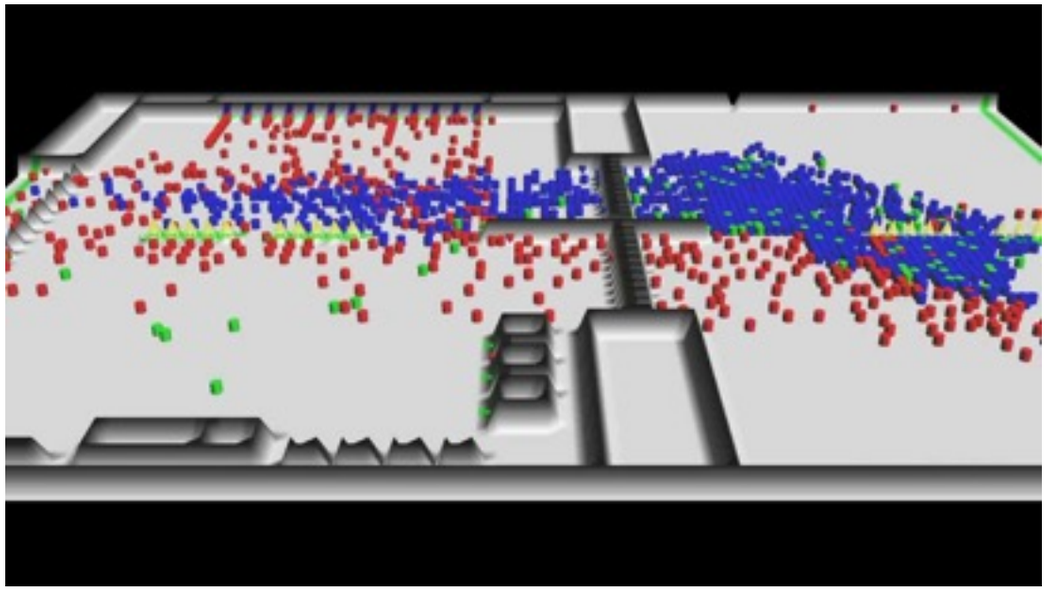
What are the dependencies?

What is the degree of parallelism? Locality?

Crowd Simulation







Parallel Object-Oriented Programming

- The Encore programming language

  Make the design defaults give good properties ”for free” (focus on parallelism)

- Starting with active objects and futures as the vanilla model

- ”Secret sauces”

  Parallel combinators, fancy capability-based types, modular layout specification, …

- A lot of what I have shown you is in some incomplete state of implementation

- We are looking for collaborators at any level

- We are also looking for users that can tell us what their pains are

Thanks for listening!