accu
conference
2024

Concurrency Hylomorphism

Lucian Radu Teodorescu

Concurrency Hylomorphism

LUCIAN RADU TEODORESCU

GARMIN

spoiler

concurrency is HARD

spoiler

this talk on concurrency is HARD

concurrency

philosophy

stackless coroutines

stackfull coroutines

theory / practice

matter / form

C++

hylo

future hylo

hylomorphism

every physical object is compound of **matter** (hylē) and **form** (morphē)

@LucTeo@techhub.social

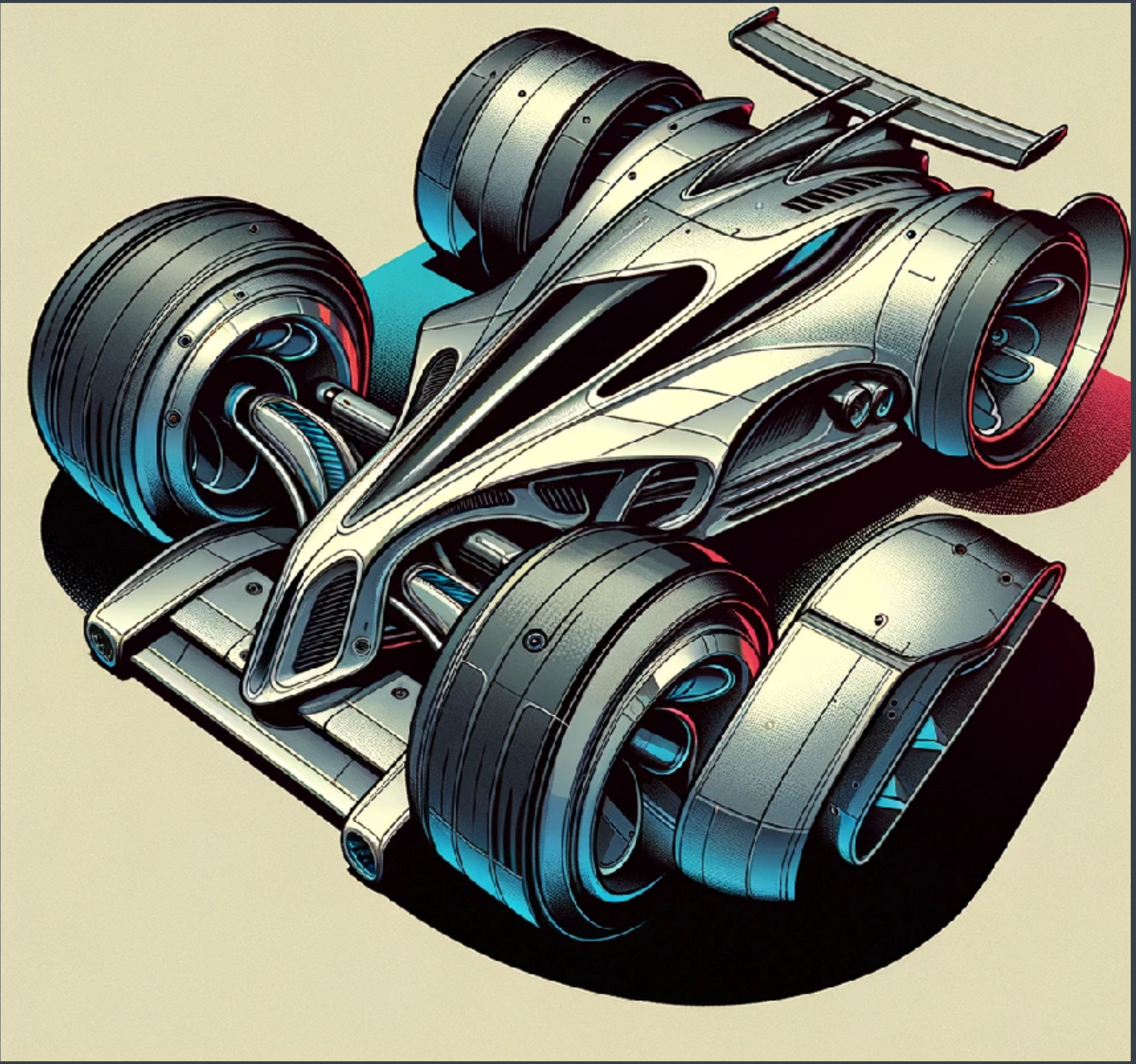
in CS

a recursive function,
corresponding to the composition of
anamorphism followed by
a **catamorphism**

aligning **hylē** and **morphē** for concurrency

aligning **potentiality** and **actuality** for concurrency

theory into **practice**
for concurrency



prototype

lucteo.ro/pres/2024-accu

1. Modeling concurrency
2. Hylo
3. Expressing concurrency
4. Structured concurrency
5. Implementation details
6. Early measurements
7. Analysis
8. Takeaways

1

Hylo

hylē

@LucTeo@techhub.social

Hylo programming language

fast by definition

safe by default

simple

www.hylo-lang.org

name **hylo**

Alexander Stepanov: STL and Its Design Principles

builds upon the best parts from C++

value semantics
pass by value, without copy
copies & moves are explicit
consuming move semantics
rules for capture access w/o consuming

value semantics

```
template <typename T>
void append2(std::vector<T>& destination, const T& value) {
  destination.push_back(value);
  destination.push_back(value);
}

std::vector<int> data;
…
append2(data, data[0]);
```

@LucTeo@techhub.social

value semantics

```
fun append2<T>(_ destination: inout Array<T>, _ value: T) {
  &destination.push_back(value)
  &destination.push_back(value)
}

var data: Array<Int>
…
append2(&data, data[0])      // ERROR
let value = data[0].copy()
append2(&data, value)        // OK
```

copies & moves
are explicit

@LucTeo@techhub.social

law of exclusivity

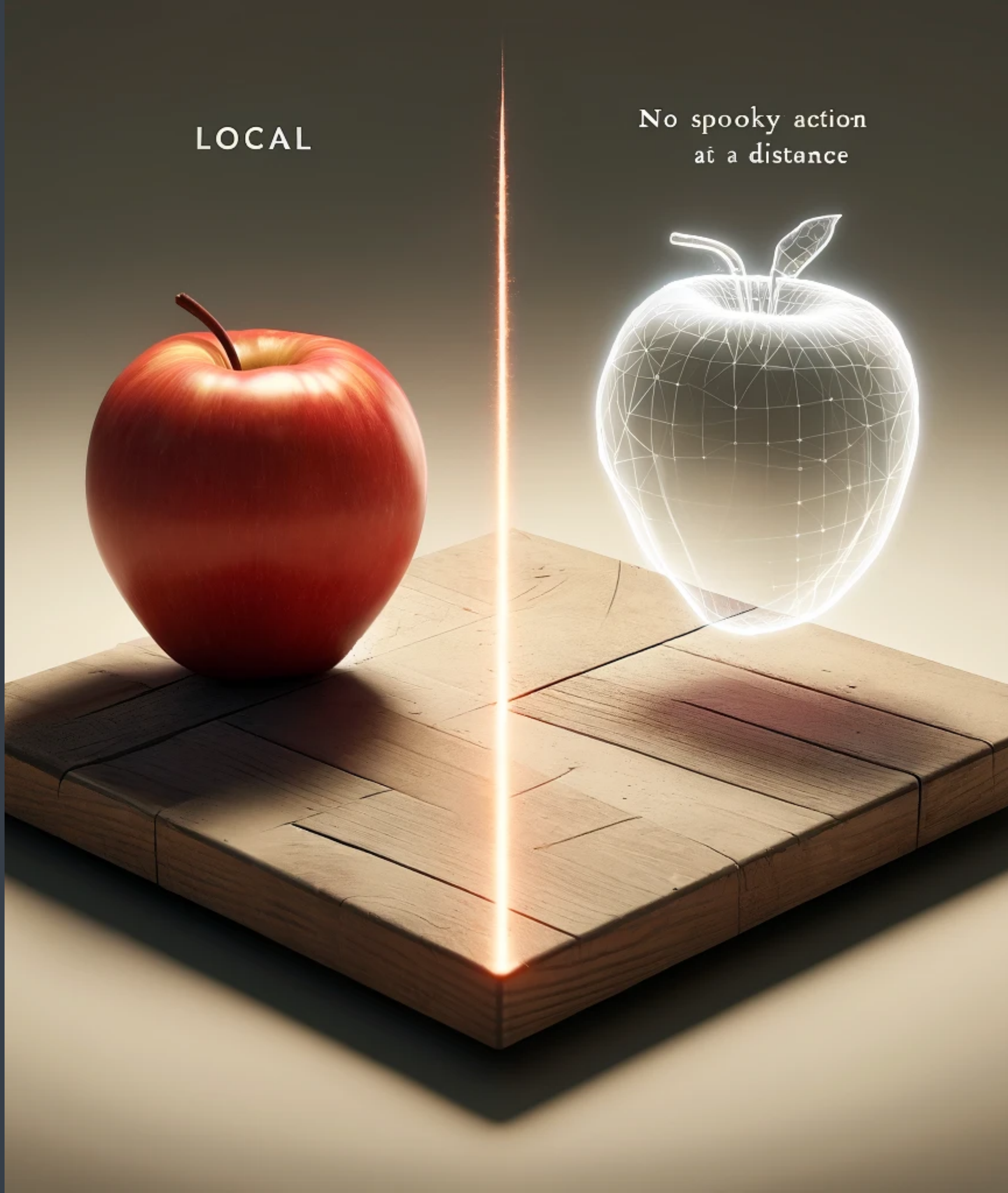
no simultaneous **read + write** access

no simultaneous **write + write** access

read + read = ok

local reasoning

no spooky action at a distance



concurrency model

targeting Hylo

Modeling concurrency

2

hylē

concurrency

partial ordering on task execution

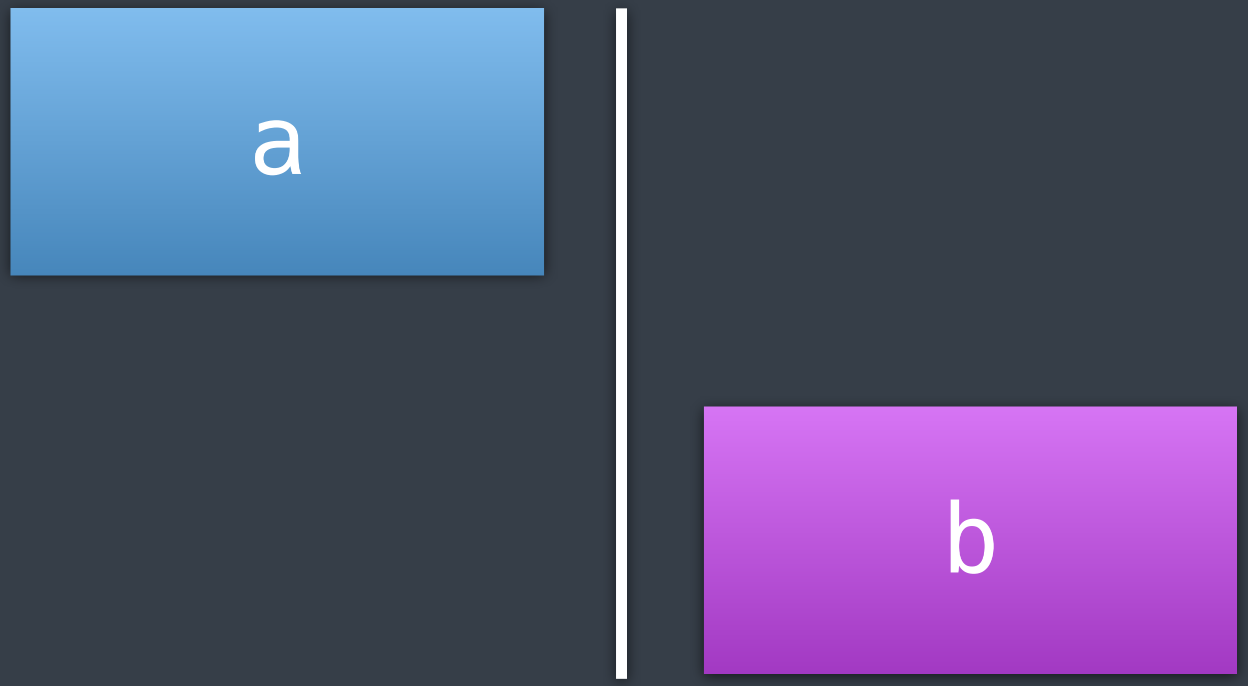
at runtime

3 execution possibilities

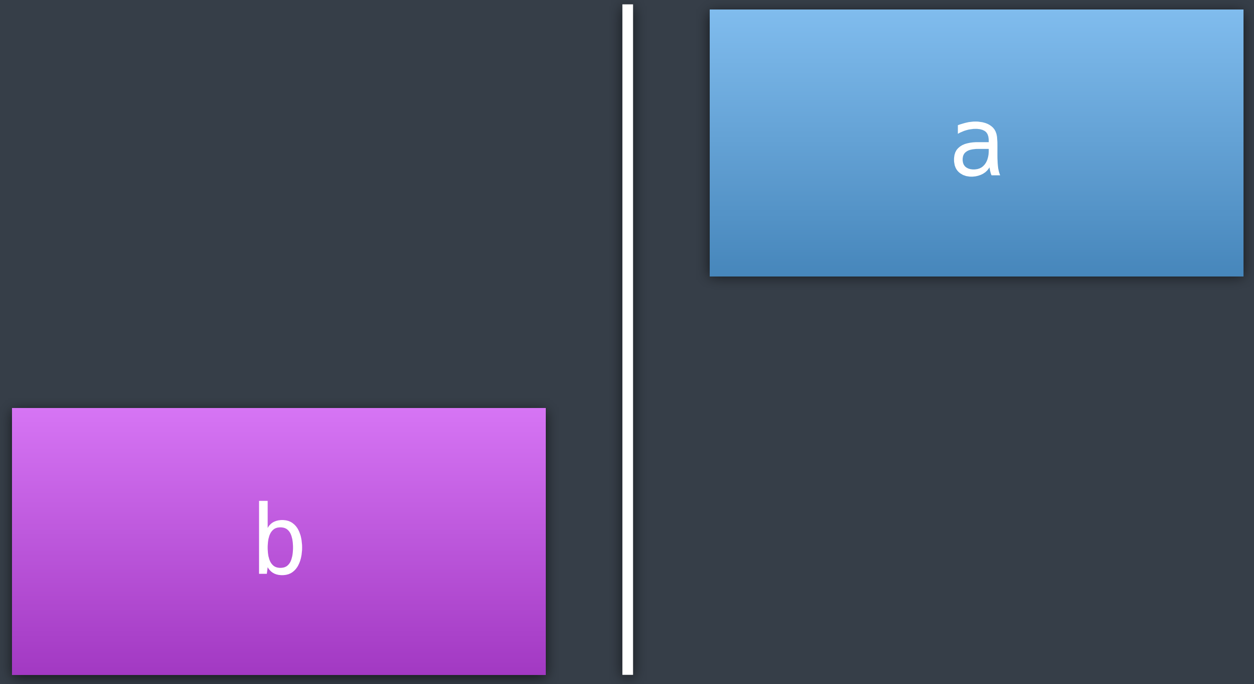
```
a < b
b < a
!(a < b) && !(b < a)
```

*) $a \neq b$

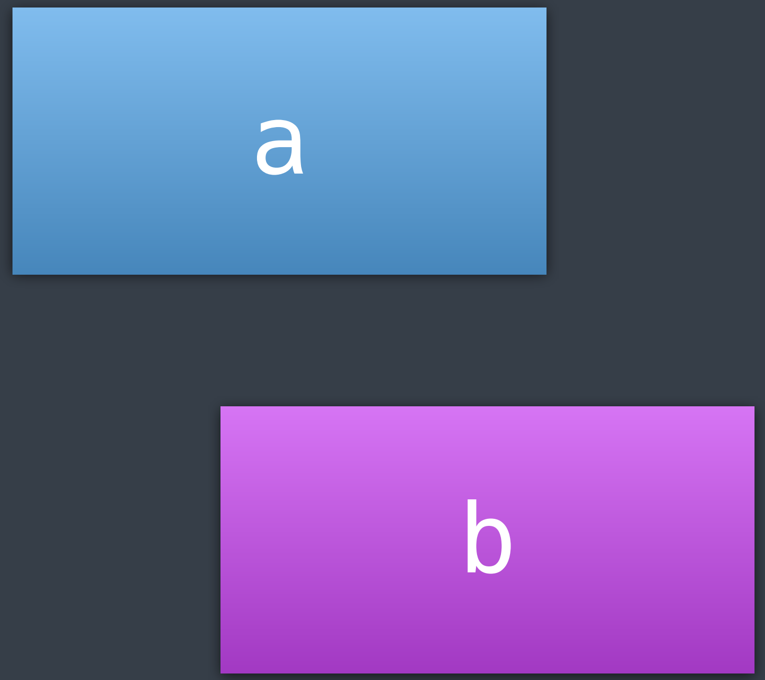
1)



2)



3)

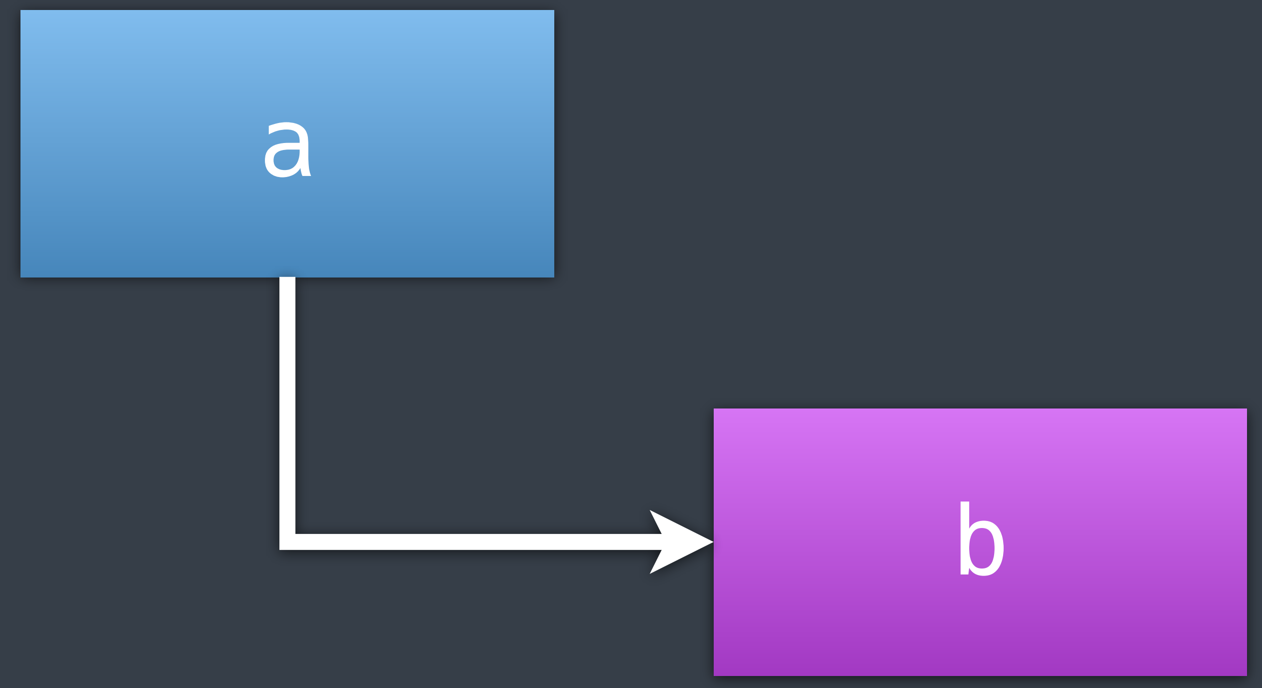


@LucTeo@techhub.social

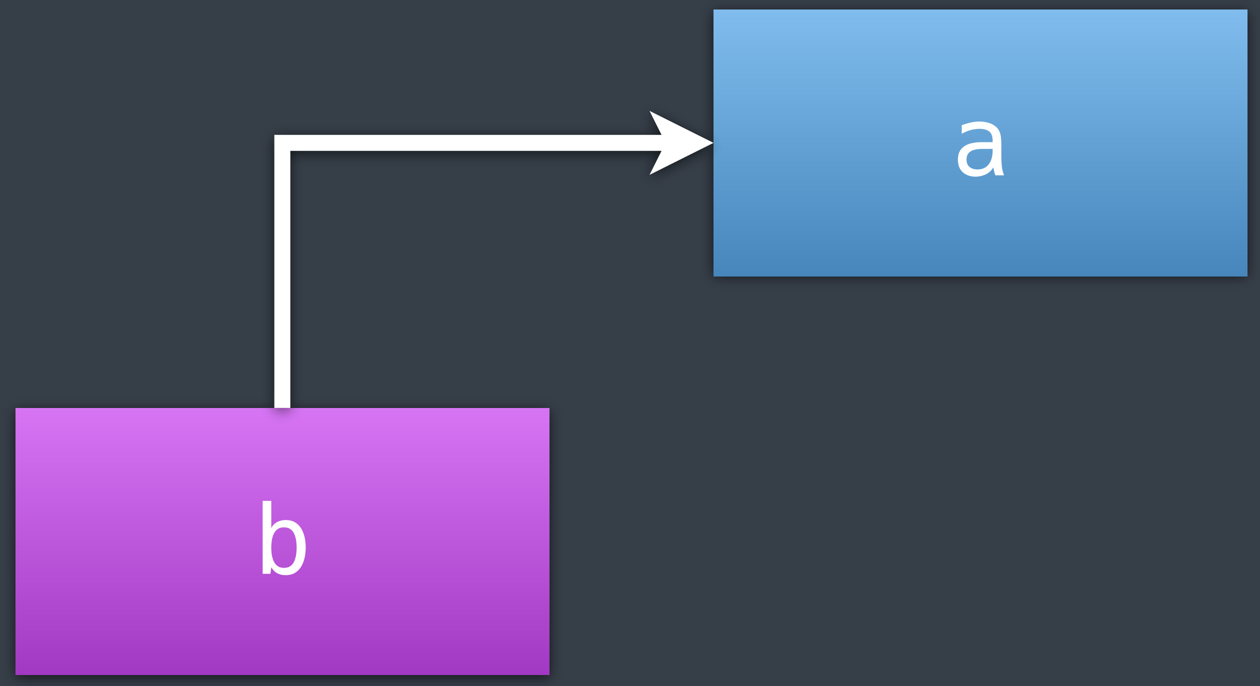
concurrency
(design time)

expressing execution constraints
ignoring actual execution

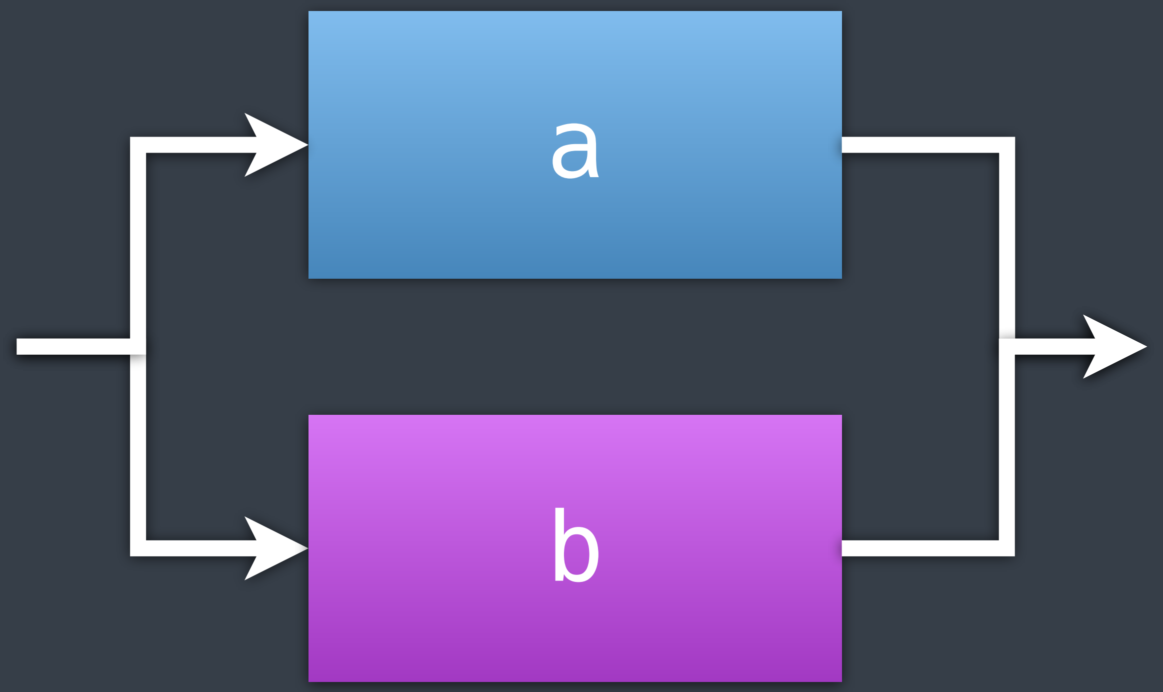
design time



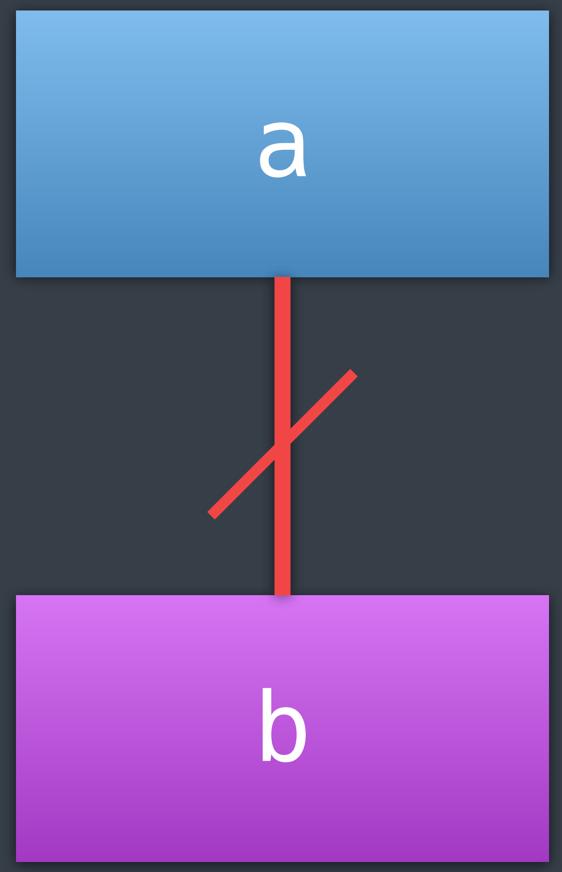
$a < b$



$b < a$



concurrent execution



mutual execution

design time

basic concurrent constraints

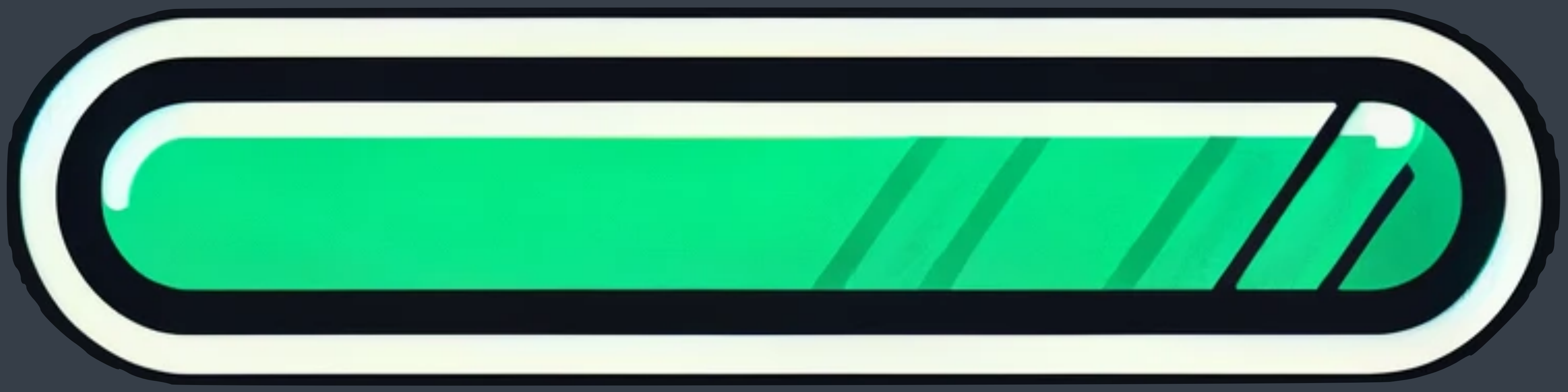
<table>
<tr><td><code>a < b</code></td><td></td></tr>
<tr><td><code>b < a</code></td><td></td></tr>
<tr><td><code>(a < b) || (b < a)</code></td><td>mutual exclusion</td></tr>
<tr><td><code>!(a < b) && !(b < a)</code></td><td>concurrent execution</td></tr>
</table>

advanced concurrent constraints

conditional concurrency

(sometimes exclusion, sometimes concurrent)

more than 2 work items



100%

there is nothing more to concurrency

3

Expressing concurrency

morphē

@LucTeo@techhub.social

future

@LucTeo@techhub.social

Hello, concurrent world!

```
void concurrent_greeting() {
  auto f = concore2full::spawn([] {
    printf("Hello, concurrent world!\n");
  });
  // do some other things...
  f.await();
}
```

@LucTeo@techhub.social

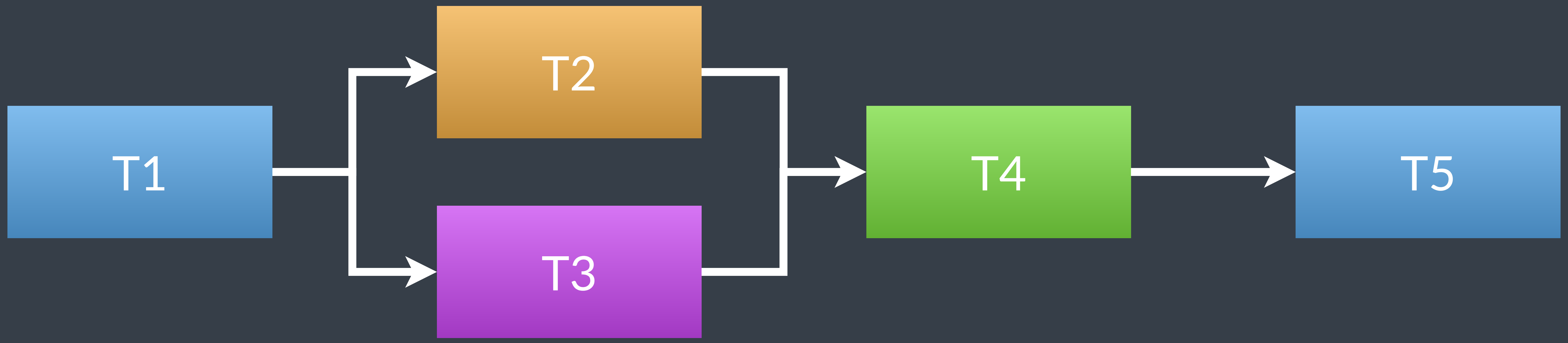
Hello, concurrent world!

```
fun concurrent_greeting() {
  var f = spawn_(fun() -> Int {
    print("Hello, concurrent world!")
    return 0
  })
  // do some other things...
  _ = f.await()
}
```

Hello, concurrent world!

```
fun concurrent_greeting() {
  var f = spawn {
    print("Hello, concurrent world!")
  }
  // do some other things...
  f.await()
}
```

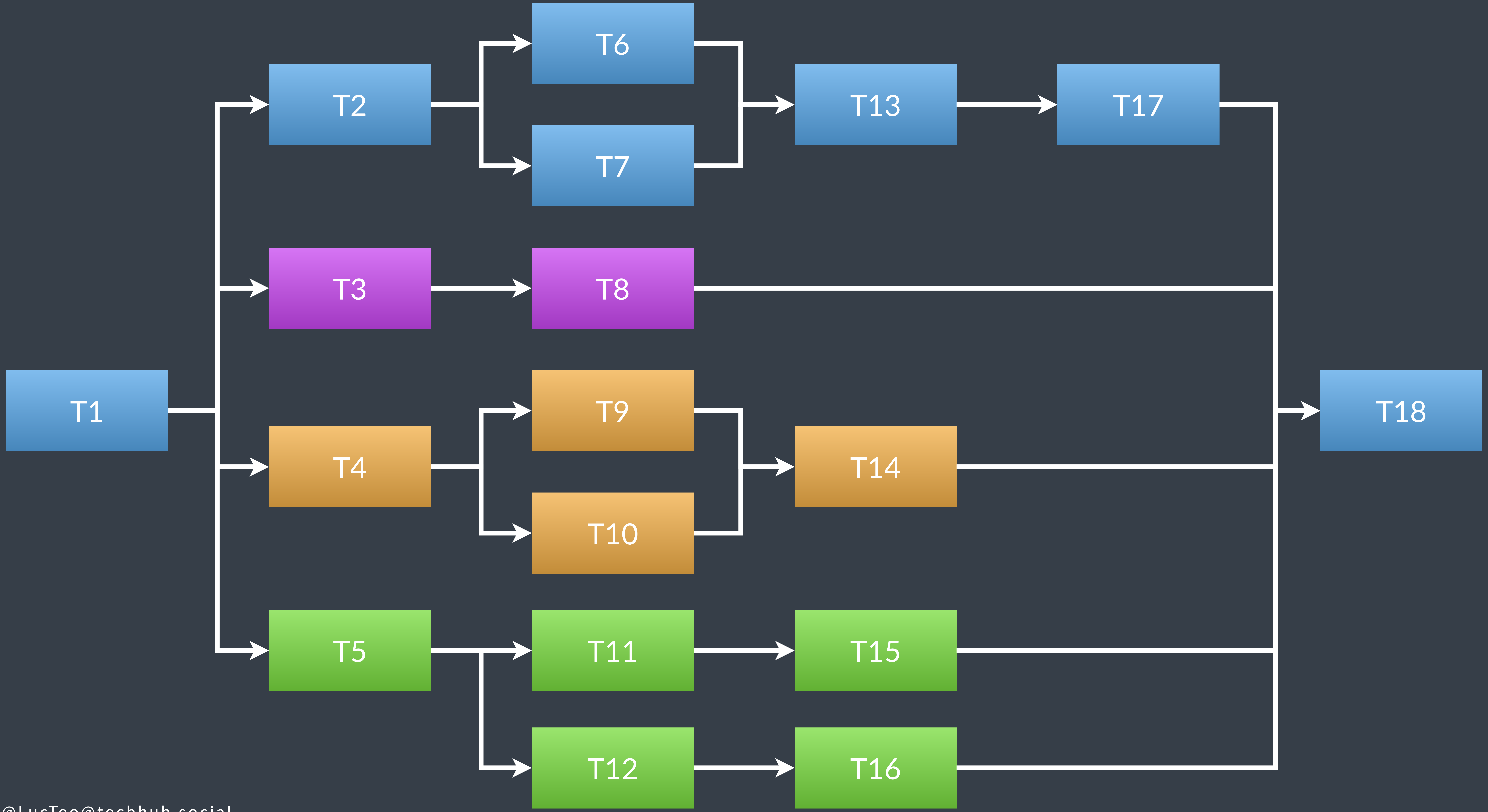
task relations



task relations

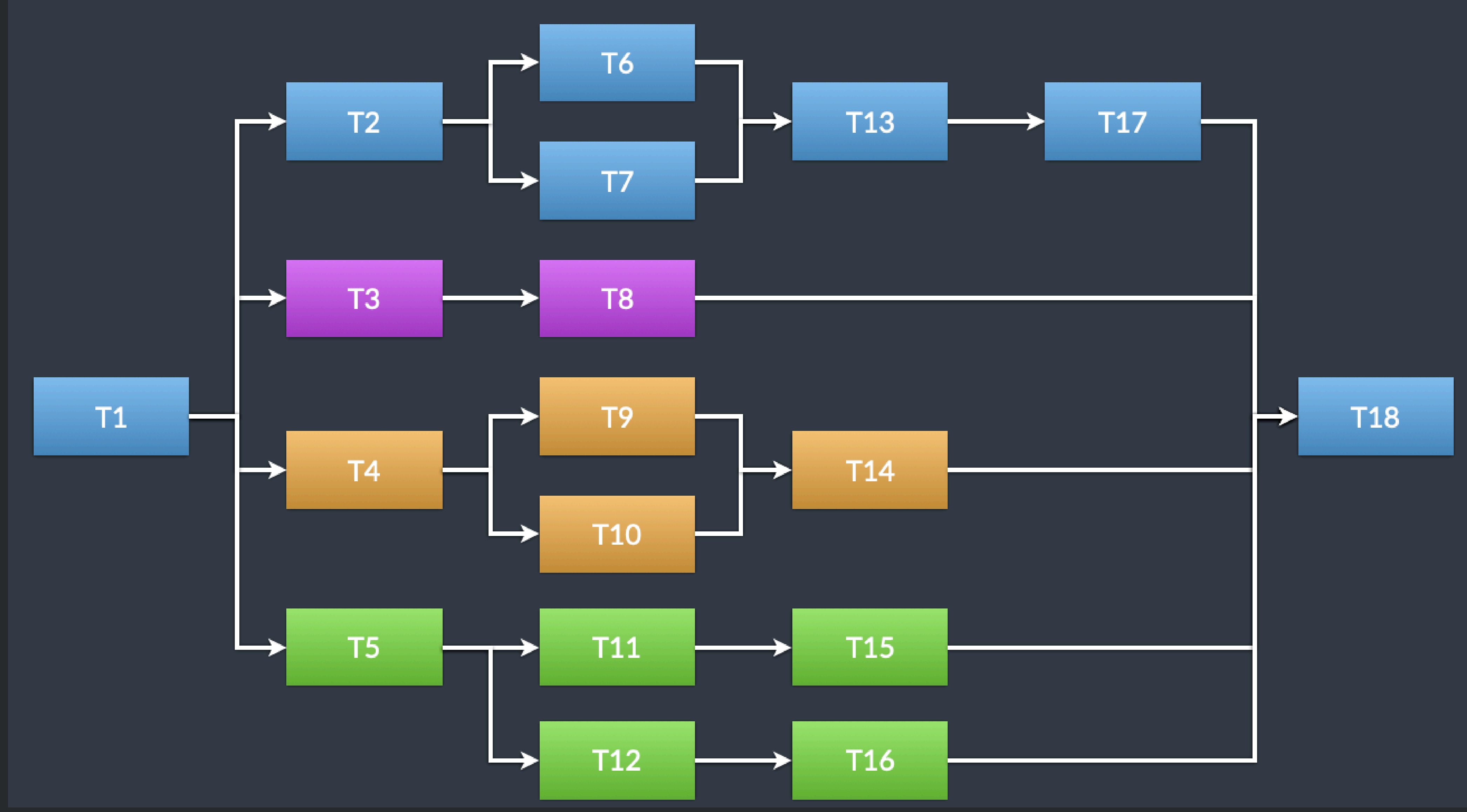
```
fun task_relations() {
  print("T1")
  var f = spawn { print("T3") }
  print("T2")
  f.await()
  print("T4")
  print("T5")
}
```

@LucTeo@techhub.social



@LucTeo@techhub.social

```
fun run_work() -> Int {
  var sum = 0
  &sum += run_task(1)

  var f2 = spawn_(fun[] () -> Int {
    var local_sum = 0
    &local_sum += run_task(2)

    var f = spawn_(fun[] () -> Int { return run_task(7) })
    &local_sum += run_task(6)
    &local_sum += f.await()

    &local_sum += run_task(13)
    &local_sum += run_task(17)
    return local_sum
  })

  var f3 = spawn_(fun[] () -> Int {
    return run_task(3) + run_task(8)
  })

  var f4 = spawn_(fun[] () -> Int {
    var local_sum = 0
    &local_sum += run_task(4)

    var f = spawn_(fun[] () -> Int { return run_task(10) })
    &local_sum += run_task(9)
    &local_sum += f.await()

    &local_sum += run_task(14)
    return local_sum
  })

  var f5 = spawn_(fun[] () -> Int {
    var local_sum = 0
    &local_sum += run_task(5)

    var f = spawn_(fun[] () -> Int { return run_task(12) + run_task(16) })
    &local_sum += run_task(11) + run_task(15)
    &local_sum += f.await()

    return local_sum
  })

  sum += f2.await() + f3.await() + f4.await() + f5.await()
  &sum += run_task(18)
  return sum
}
```



@LucTeo@techhub.social

```
fun run_work() -> Int {
  var sum = 0
  &sum += run_task(1)

  var f2 = …
  var f3 = …
  var f4 = …
  var f5 = …

  sum += f2.await() + f3.await() + f4.await() + f5.await()
  &sum += run_task(18)
  return sum
}
```

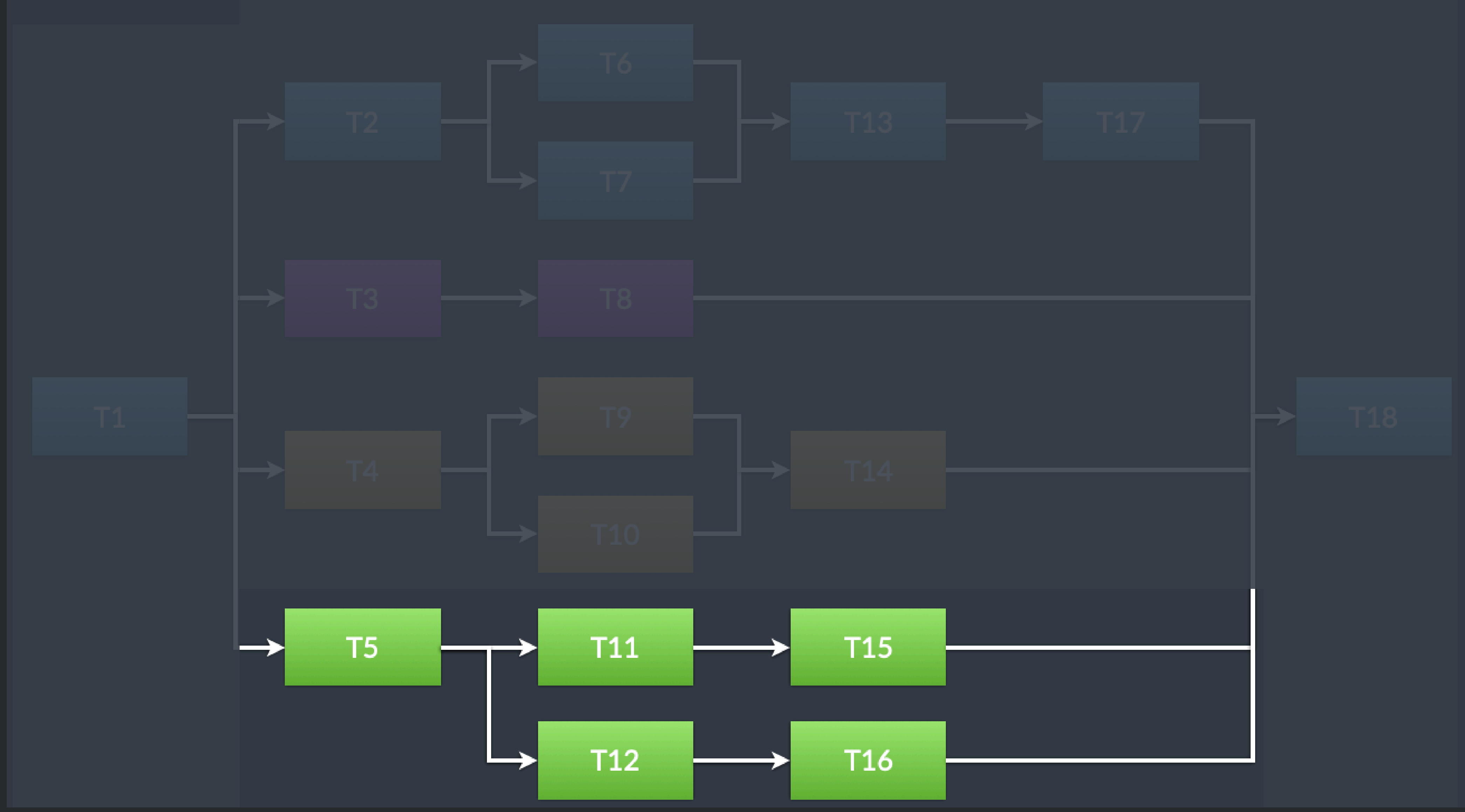
@LucTeo@techhub.social

```
var f2 = spawn_(fun[] () -> Int {
  var local_sum = 0
  &local_sum += run_task(2)

  var f = spawn_(fun[] () -> Int { return run_task(7) })
  &local_sum += run_task(6)
  &local_sum += f.await()

  &local_sum += run_task(13)
  &local_sum += run_task(17)
  return local_sum
})
```

```
var f3 = spawn_(fun[] () -> Int {
  return run_task(3) + run_task(8)
})
```

```
var f4 = spawn_(fun[] () -> Int {
  var local_sum = 0
  &local_sum += run_task(4)

  var f = spawn_(fun[] () -> Int { return run_task(10) })
  &local_sum += run_task(9)
  &local_sum += f.await()

  &local_sum += run_task(14)
  return local_sum
})
```

@LucTeo@techhub.social



```
var f5 = spawn_(fun[] () -> Int {
  var local_sum = 0
  &local_sum += run_task(5)

  var f = spawn_(fun[] () -> Int { return run_task(12) + run_task(16) })
  &local_sum += run_task(11) + run_task(15)
  &local_sum += f.await()

  return local_sum
})
```

@LucTeo@techhub.social

concurrent quick-sort

```
fun concurrent_sort<Element: Regular & Comparable>(_ a: inout ArraySlice<Element>) -> Int {
  if a.count() < size_threshold {
    // Use serial sort under a certain threshold.
    a.sort()
  } else {
    // Partition the data.
    let (m1, m2) = partition(&a)
    inout (lhs, rhs) = &a.split(at: m1)
    &rhs.drop_first(m2 - m1)

    // Spawn work to sort the right-hand side.
    let future = spawn_(
        fun[sink let q=mutable_pointer[to: &rhs].copy()]() -> Int {
      inout rhs = &(q.copy().unsafe[])

      return concurrent_sort(&rhs)
    })

    // Execute the sorting on the left side, on the current thread.
    _ = concurrent_sort(&lhs)
    _ = future.await()
  }
  return a.count()
}
```

@LucTeo@techhub.social

```
fun concurrent_sort<Element: Regular & Comparable>(_ a: inout ArraySlice<Element>) -> Int {
  if a.count() < size_threshold {
    // Use serial sort under a certain threshold.
    a.sort()
  } else {
    // Partition the data.
    let (m1, m2) = partition(&a)
    inout (lhs, rhs) = &a.split(at: m1)
    &rhs.drop_first(m2 - m1)

    // Spawn work to sort the right-hand side.
    let future = spawn {
      return concurrent_sort(&rhs)
    })

    // Execute the sorting on the left side, on the current thread.
    _ = concurrent_sort(&lhs)
    _ = future.await()
  }
  return a.count()
}
```

@LucTeo@techhub.social

concurrent inclusive scan

```
sender auto async_inclusive_scan(scheduler auto sch,
      std::span<const double> input, std::span<double> output, double init, std::size_t tile_count) {
  std::size_t const tile_size = (input.size() + tile_count - 1) / tile_count;

  std::vector<double> partials(tile_count + 1);
  partials[0] = init;

  return transfer_just(sch, std::move(partials))
       | bulk(tile_count,
           [ = ](std::size_t i, std::vector<double>& partials) {
             auto start = i * tile_size;
             auto end   = std::min(input.size(), (i + 1) * tile_size);
             partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                      begin(input) + end,
                                                      begin(output) + start);
           })
       | then(
           [](std::vector<double>&& partials) {
             std::inclusive_scan(begin(partials), end(partials),
                                 begin(partials));
             return std::move(partials);
           })
       | bulk(tile_count,
           [ = ](std::size_t i, std::vector<double>& partials) {
             auto start = i * tile_size;
             auto end   = std::min(input.size(), (i + 1) * tile_size);
             std::for_each(begin(output) + start, begin(output) + end,
               [&] (double& e) { e = partials[i] + e; }
             );
           })
       | then(
           [ = ](std::vector<double>&& partials) {
             return output;
           });
}
```

@LucTeo@techhub.social

```
sender auto async_inclusive_scan( … ) {
  …

  return transfer_just(…, std::move(partials))
       | bulk(…,
           [ = ](std::size_t i, std::vector<double>& partials) {
             …
           })
       | then(
           [](std::vector<double>&& partials) {
             …
             return std::move(partials);
           })
       | bulk(…,
           [ = ](std::size_t i, std::vector<double>& partials) {
             …
           })
       | then(
           [ = ](std::vector<double>&& partials) {
             return output;
           });
}
```

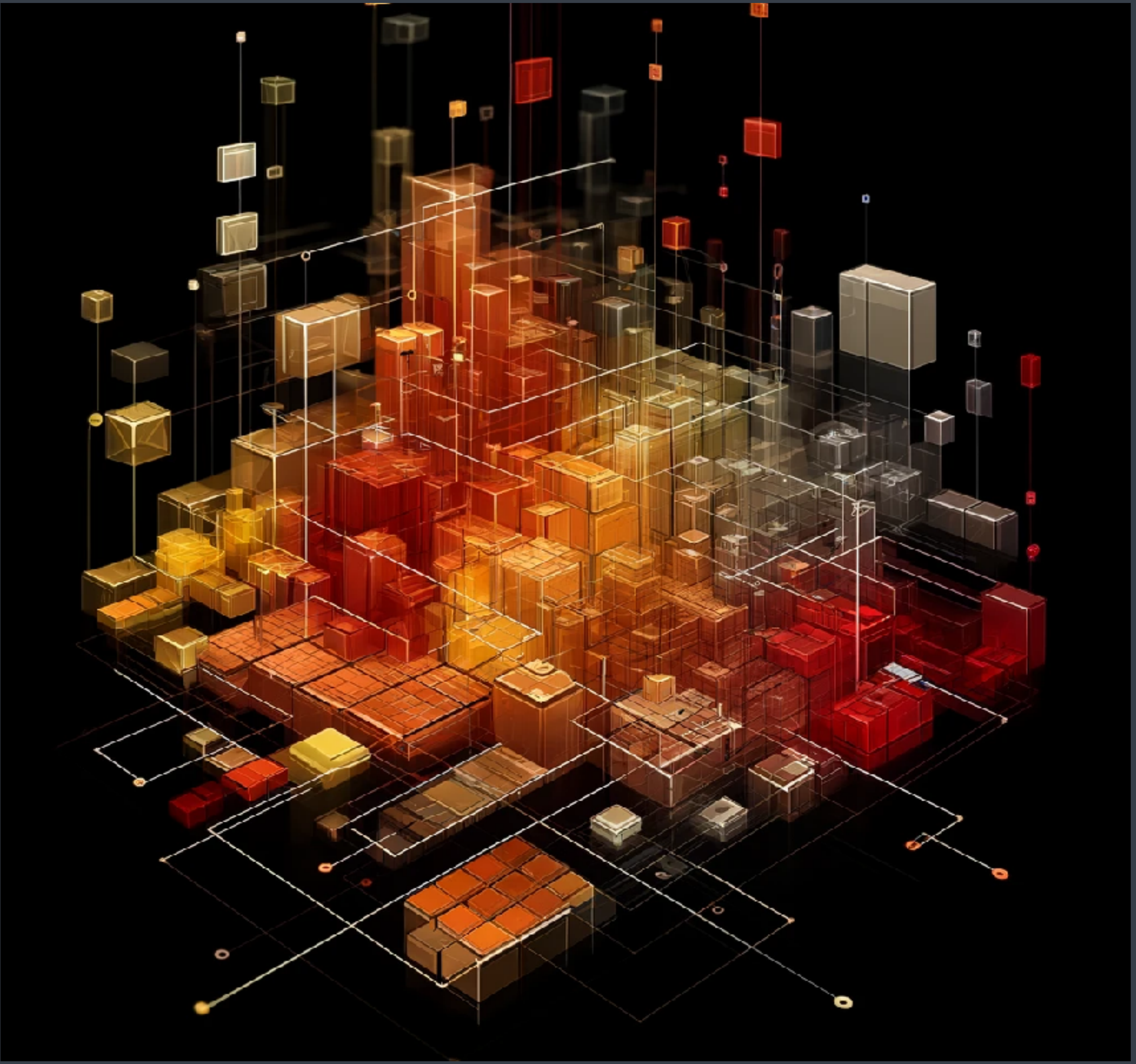
```
fun concurrent_inclusive_scan(_ input: ArraySlice<Int>, to output: inout
     ArraySlice<Int>, tile_count: Int, init_value: Int) {
  let n = input.count()
  let tile_size = (n + tile_count - 1) / tile_count

  var partials_array = Array<Int>(count: tile_count + 1, with_initial_value: 0)
  var partials = ArraySlice<Int>(full_array: &partials_array)
  &partials[0] = init_value.copy()

  spawn (count: tile_count) (index i: Int) => {
    let start = i * tile_size
    let end = min[start + tile_size, n]
    input[from: start, to: end].inclusive_scan(to: &output[from: start, to: end])
    &partials[i + 1] = output[end - 1].copy()
  }.await()

  partials.inclusive_scan()

  spawn (count: tile_count) (index i: Int) => {
    let start = i * tile_size
    let end = min[start + tile_size, n]
    &output[from: start, to: end].add(partials[i])
  }.await()
}
```

```
fun concurrent_inclusive_scan( … ) {
  …

  spawn (count: … ) (index i: Int) => {
    …
  }.await()

  …

  spawn (count: … ) (index i: Int) => {
    …
  }.await()
}
```

4

Structured concurrency

morphē



structured programming

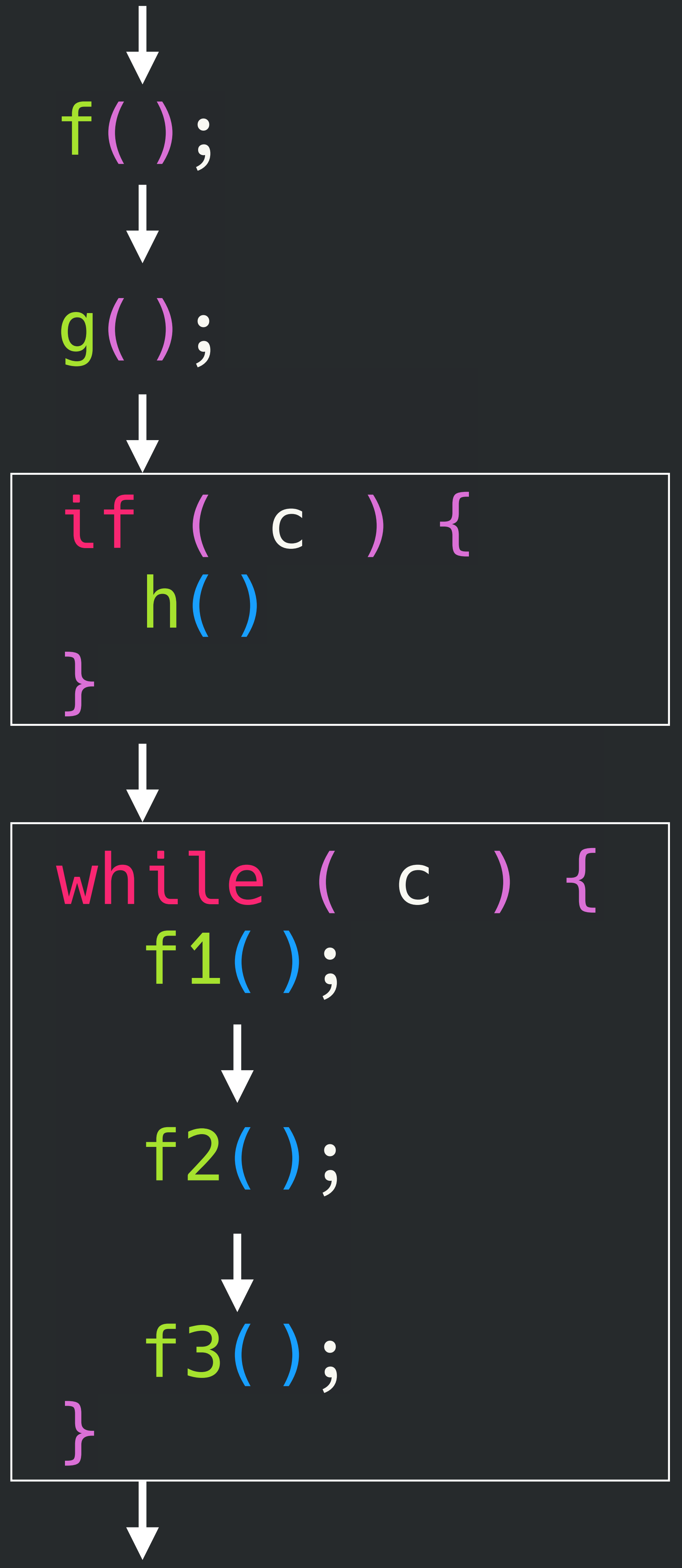
one entry, one exit
recursive decomposition

structured programming

```
f();
g();
if ( c ) {
  h()
}
while ( c ) {
  f1();
  f2();
  f3();
}
```

@LucTeo@techhub.social

structured programming



```
f();

g();

if ( c ) {
  h()
}

while ( c ) {
  f1();

  f2();

  f3();
}
```

structured programming

helps local reasoning

structured concurrency

```
void structured_concurrency() {
  f1();

  auto future = concore2full::spawn([] { f3(); });
  f2();
  future.await();

  f4();
}
```

one entry, one exit

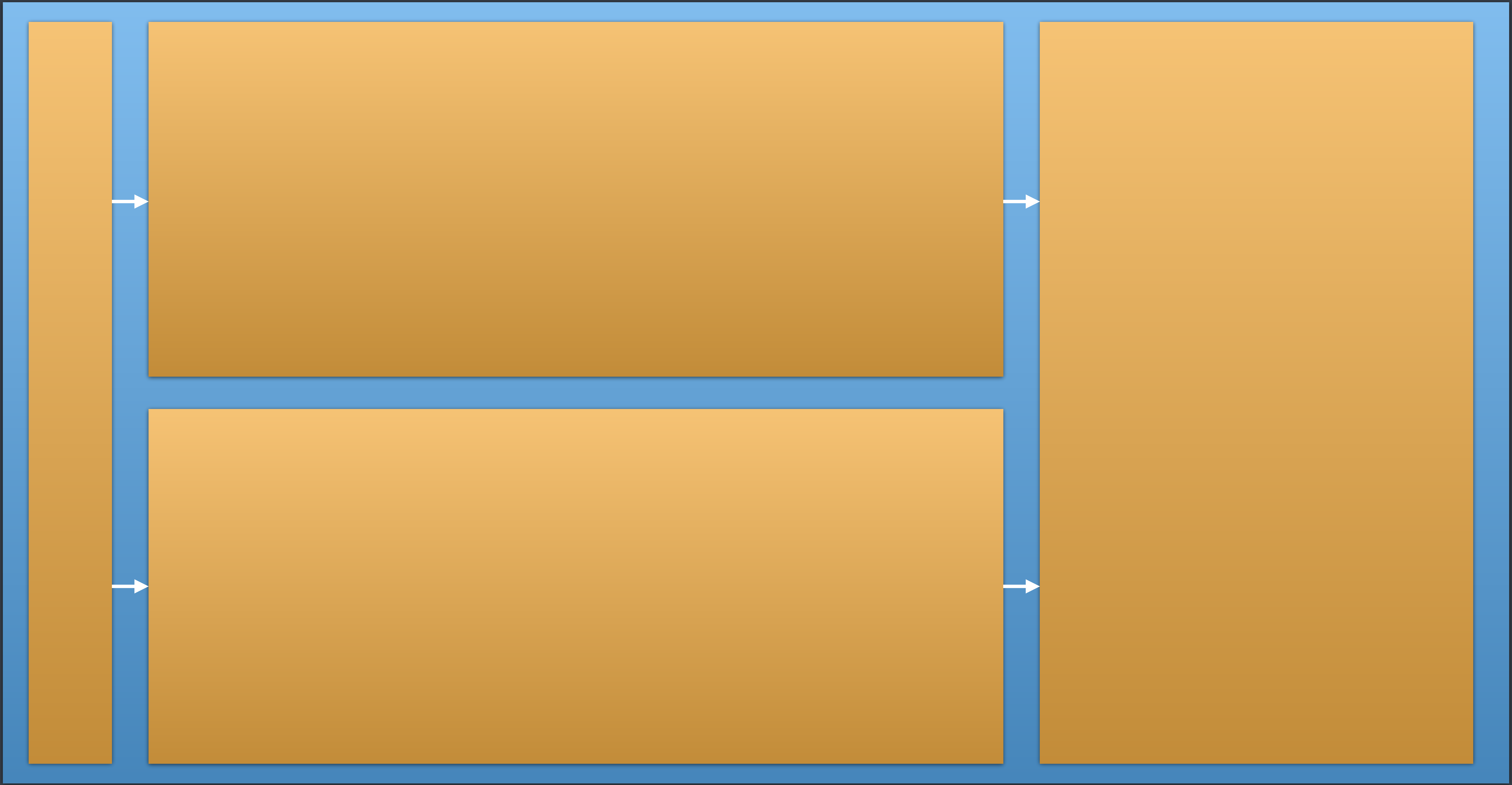
like a function call

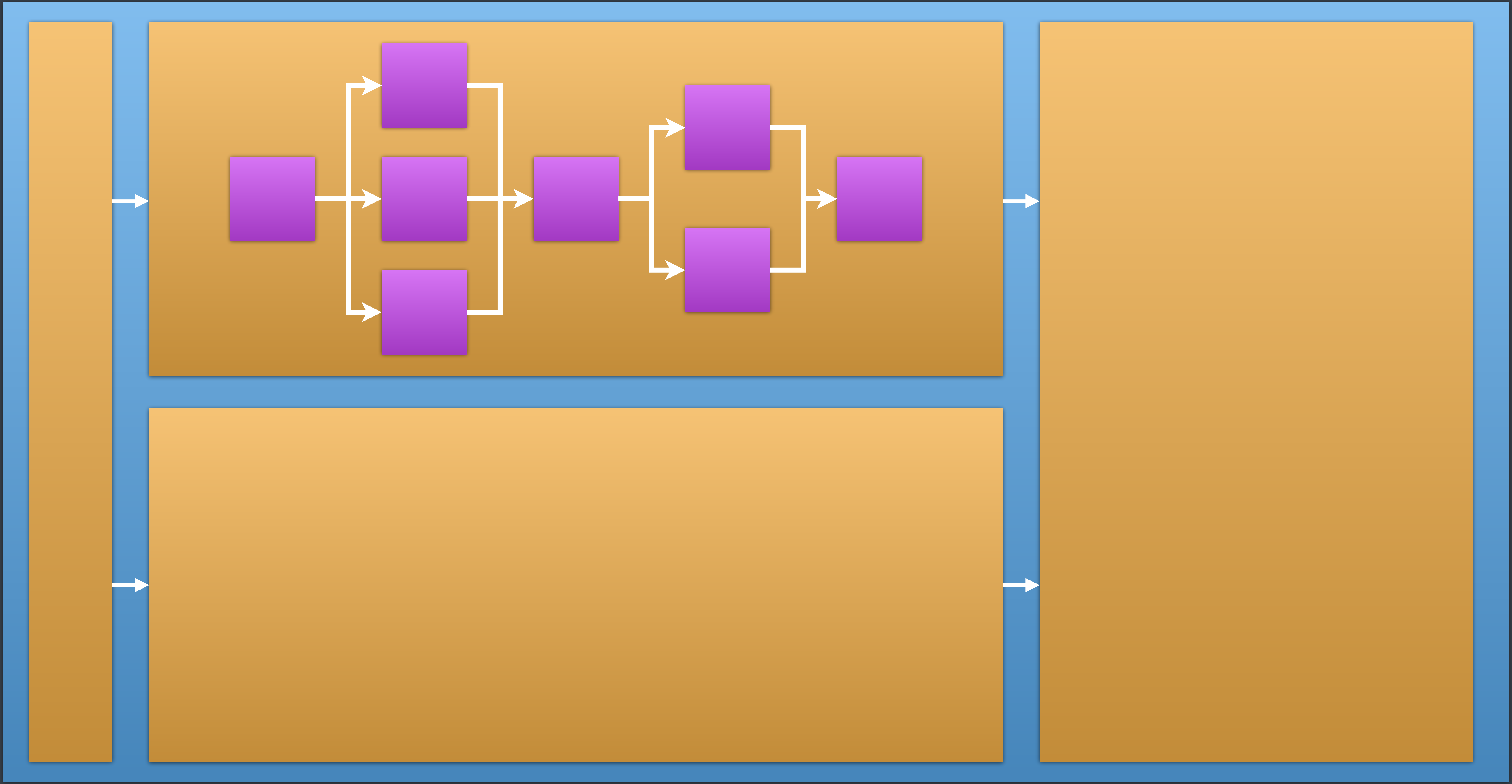
@LucTeo@techhub.social

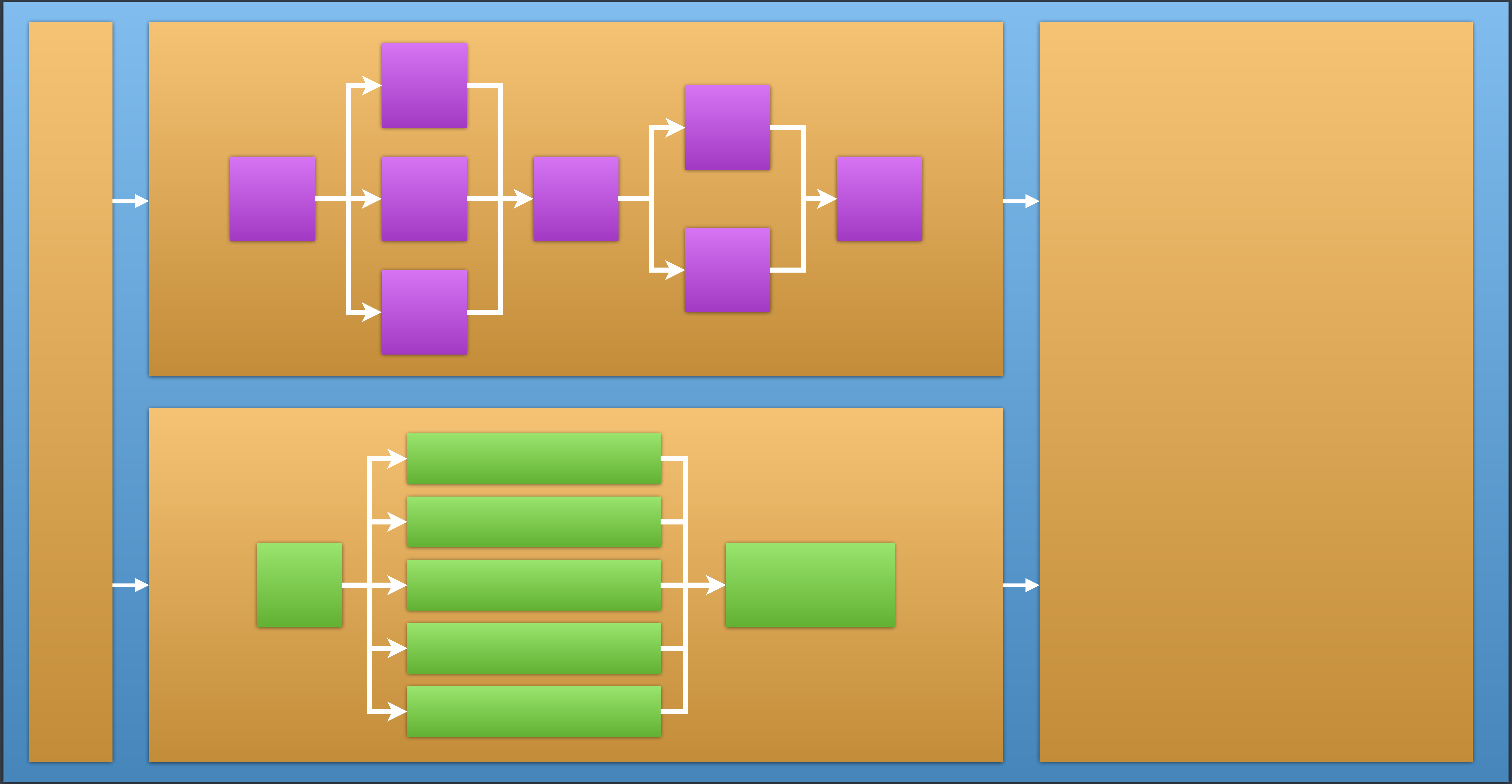
recursive decomposition

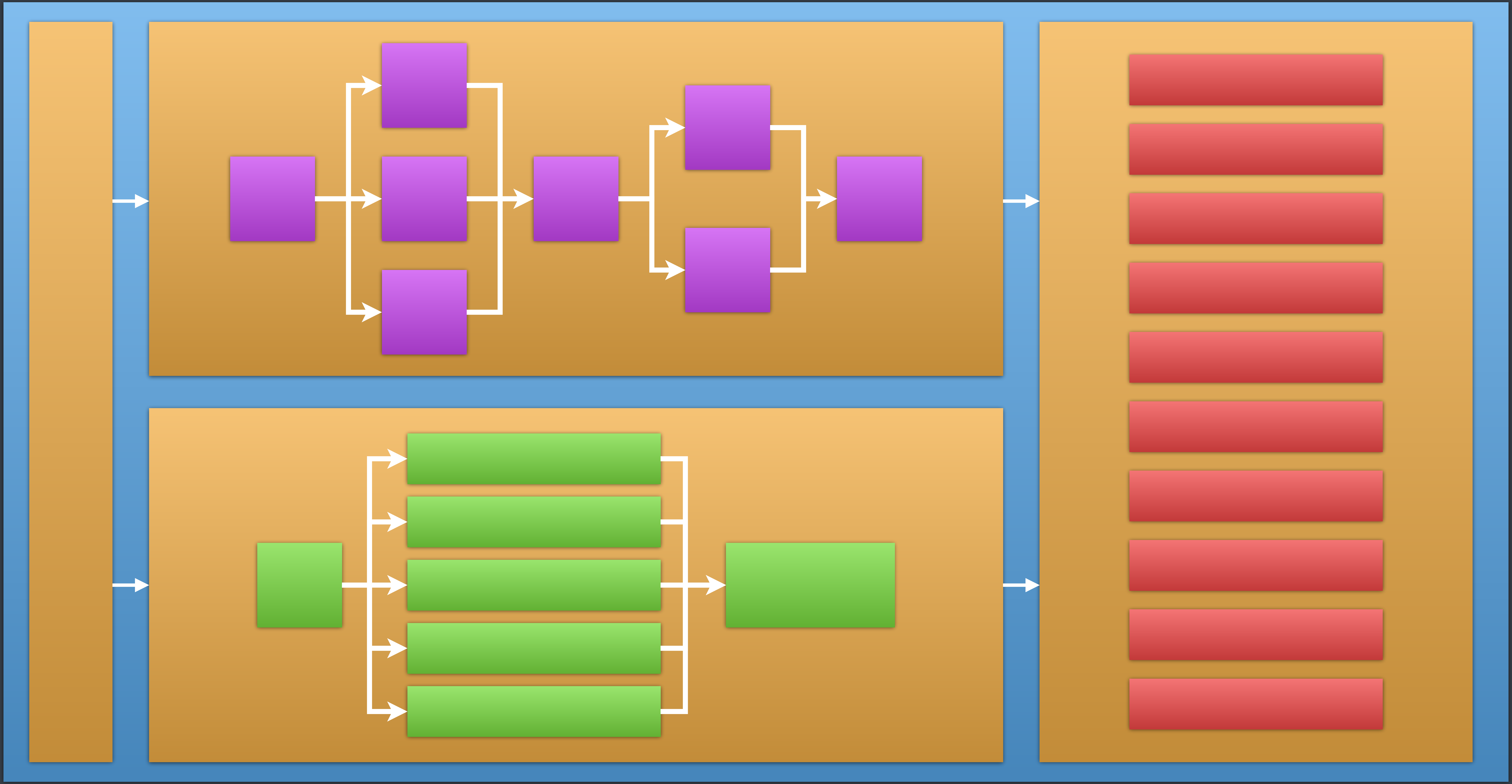
for **concurrency**

program

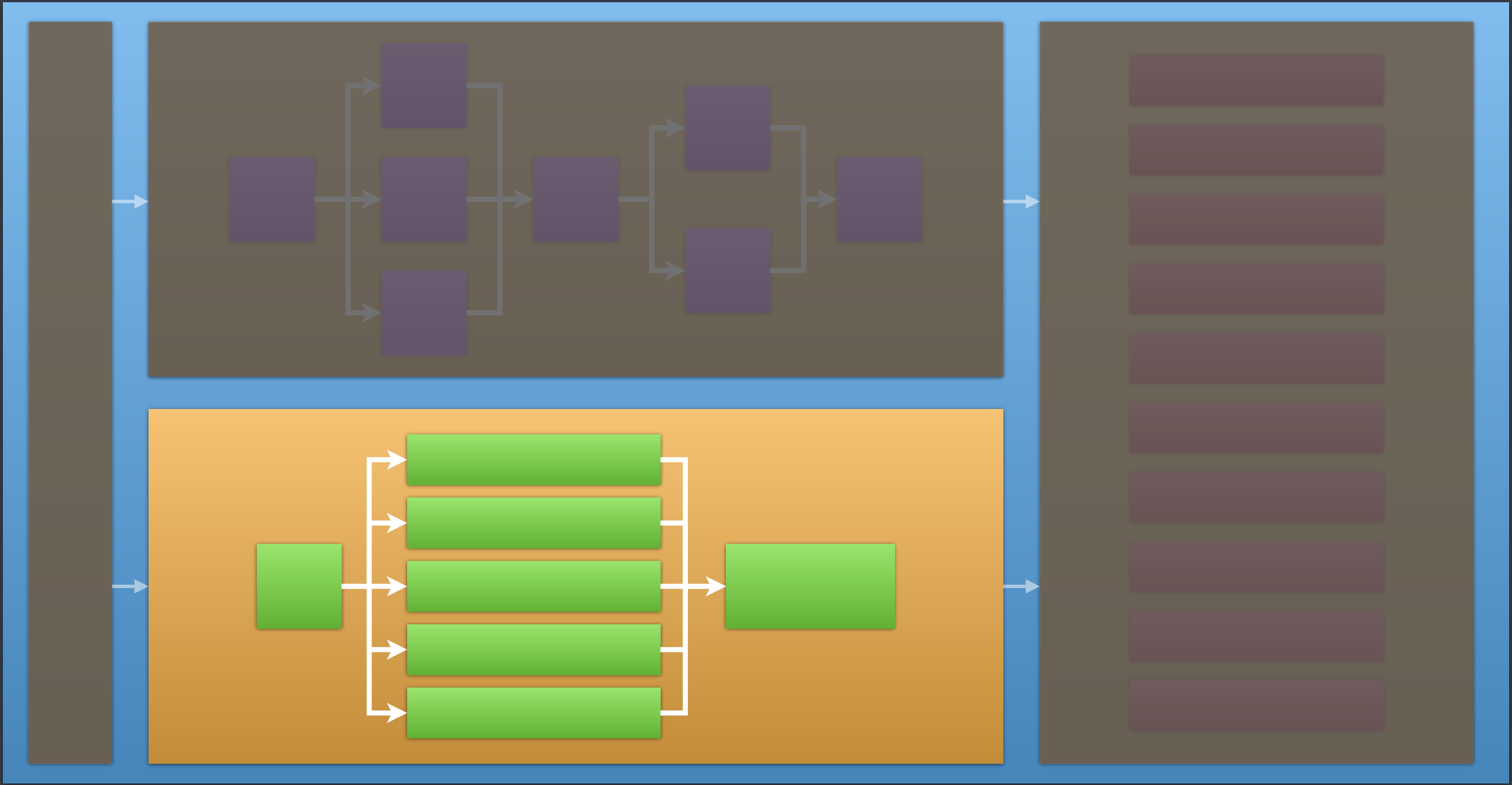








local reasoning

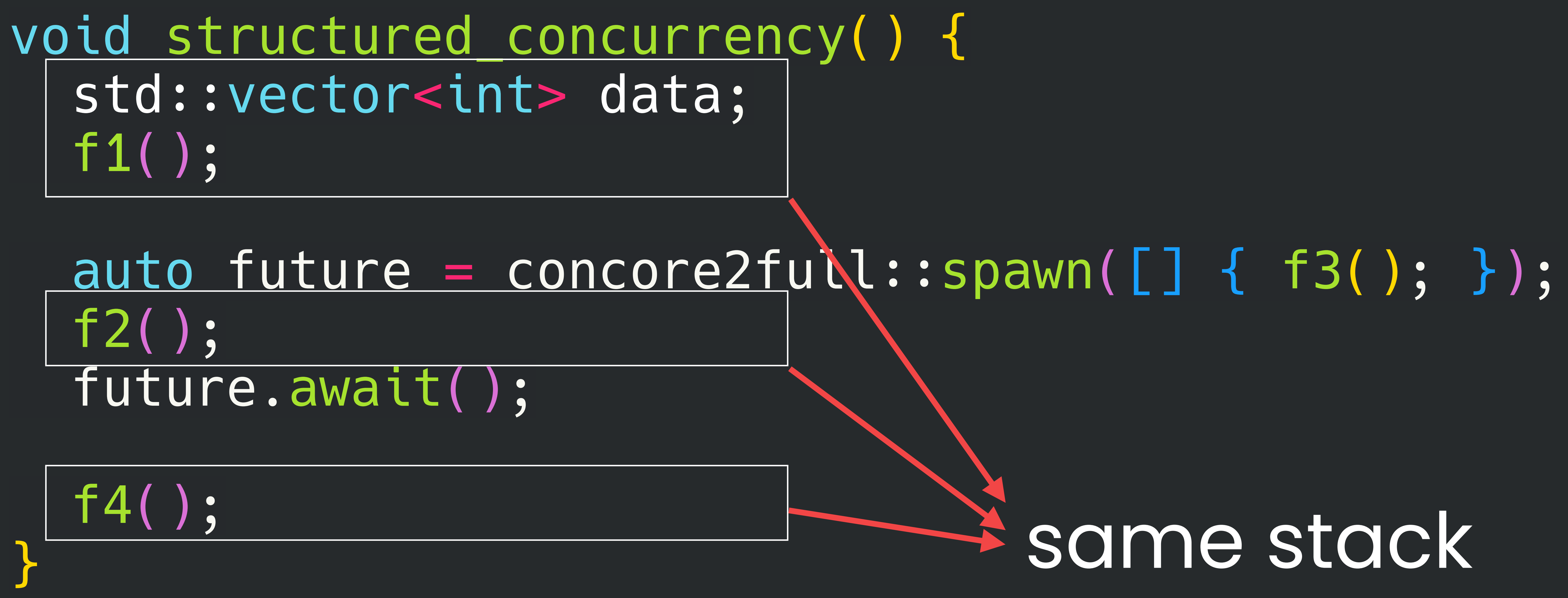


@LucTeo@techhub.social

reasonable concurrency

stack access

stack access



```
void structured_concurrency() {
  std::vector<int> data;
  f1();

  auto future = concore2full::spawn([] { f3(); });
  f2();
  future.await();

  f4();
}
```

same stack

f1 < f3 < f4

} f3 can access data

restriction

future is not movable, nor copyable

benefits

```
void structured_concurrency() {
  std::vector<int> data;
  f1();

  auto future = concore2full::spawn([] {
    f3();
  });
  f2();
  future.await();

  f4();
}
```

local reasoning

spawn frame on the stack

weakly-structured concurrency

future is movable (still not copyable)

weakly-structured concurrency

```
auto spawn_work() {
  f1();
  std::vector<int> data;

  return concore2full::escaping_spawn([] {
    f3();
  });
}

void weakly_structured_concurrency() {
  auto future = spawn_work();
  f2();
  future.await();
  f4();
}
```

f3 cannot access data

spawn frame on the heap

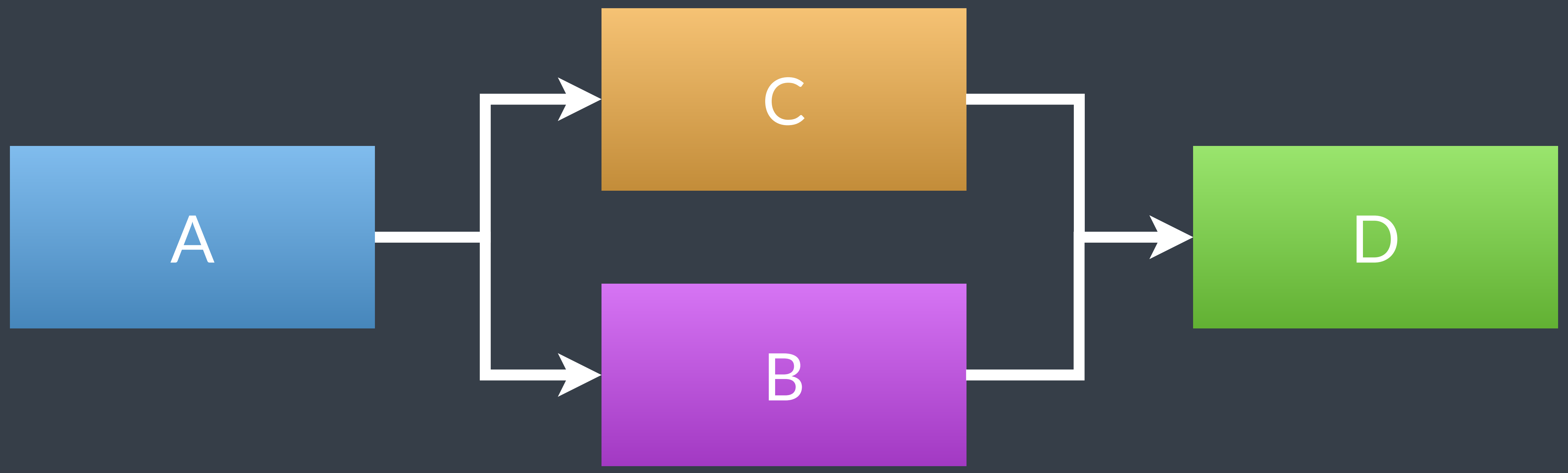
structured	weakly-structured
more structure	less structure
can access local stack	cannot access local stack
faster	allocation required
more constrained	less constrained

@LucTeo@techhub.social

Implementation details

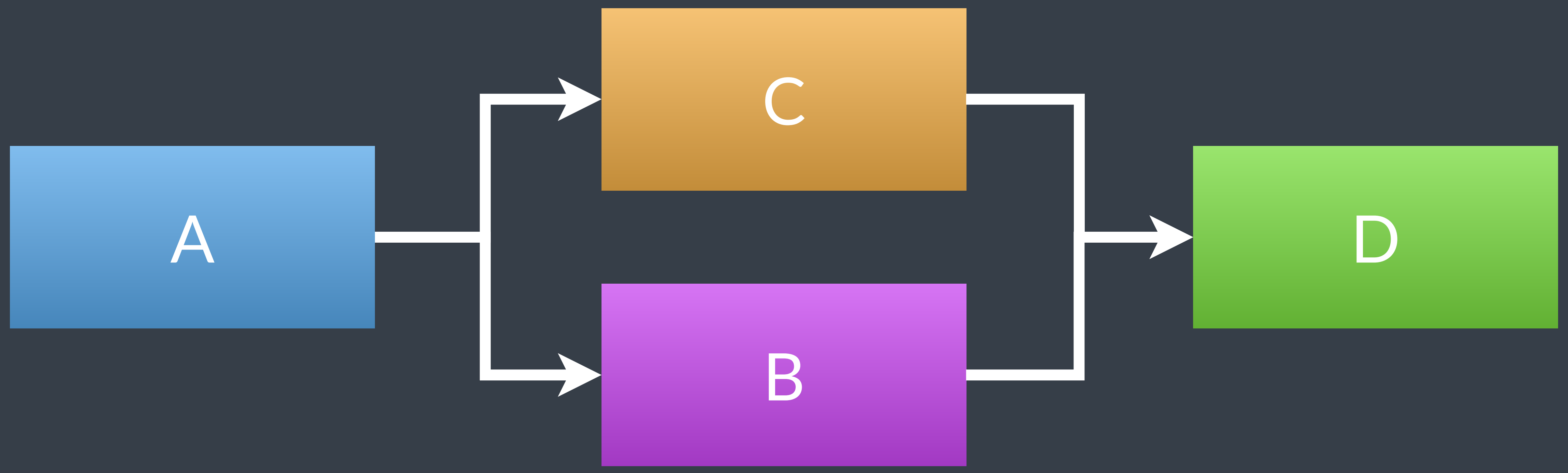
5

hylē

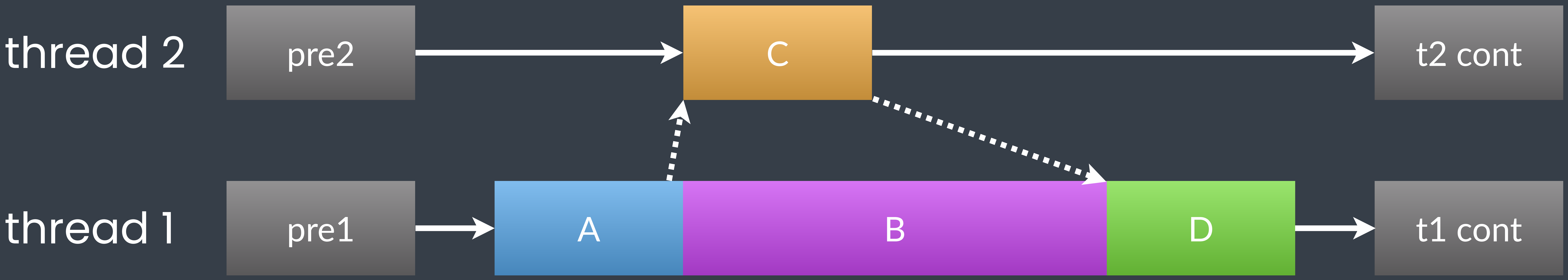


concurrency design

```
void example() {
  A();

  auto future = concore2full::spawn([] { C(); });
  B();
  future.await();

  D();
}
```



concurrency design



thread 2

pre2

C

t2 cont

thread 1

pre1

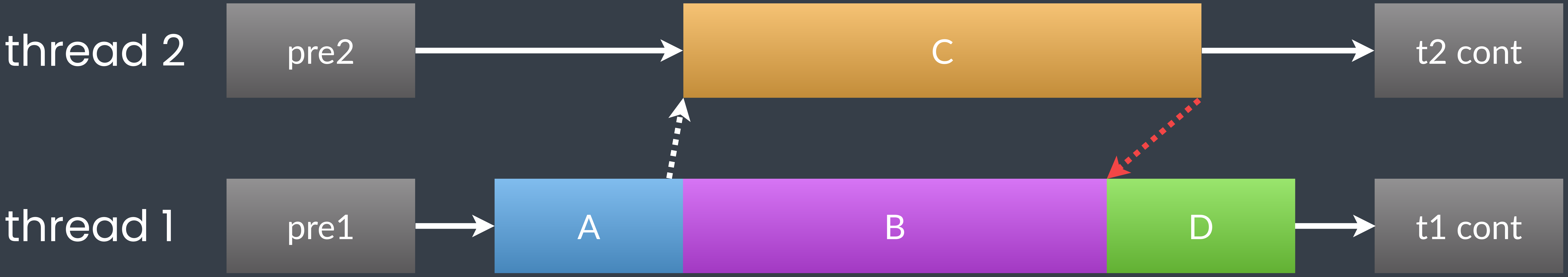
A

B

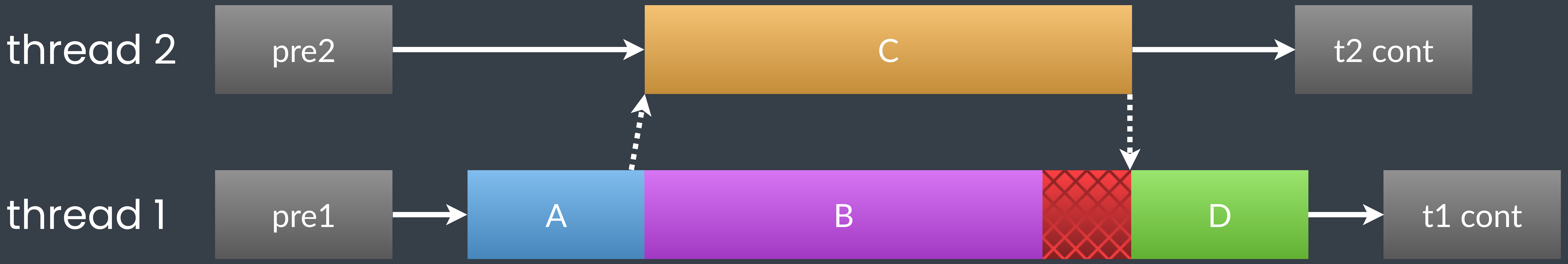
D

t1 cont

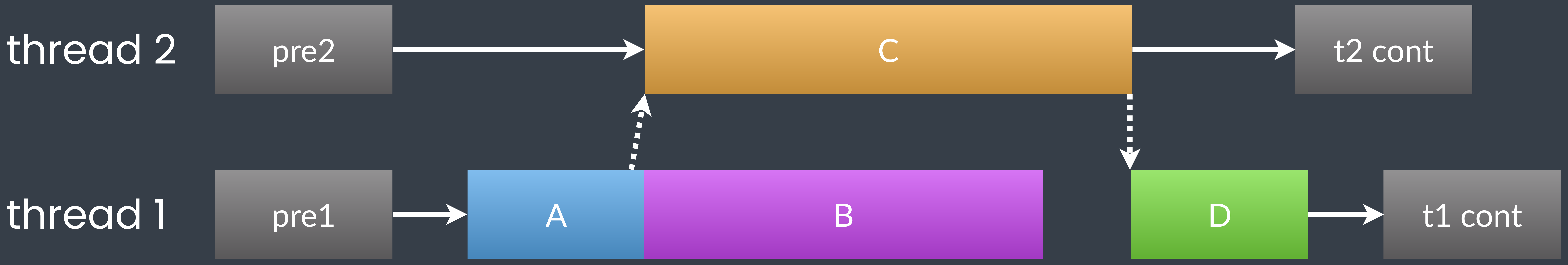
at runtime



at runtime

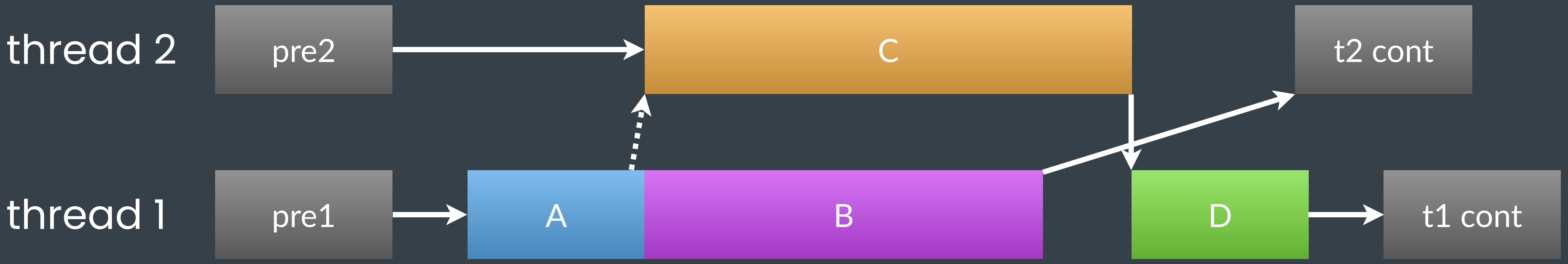


at runtime

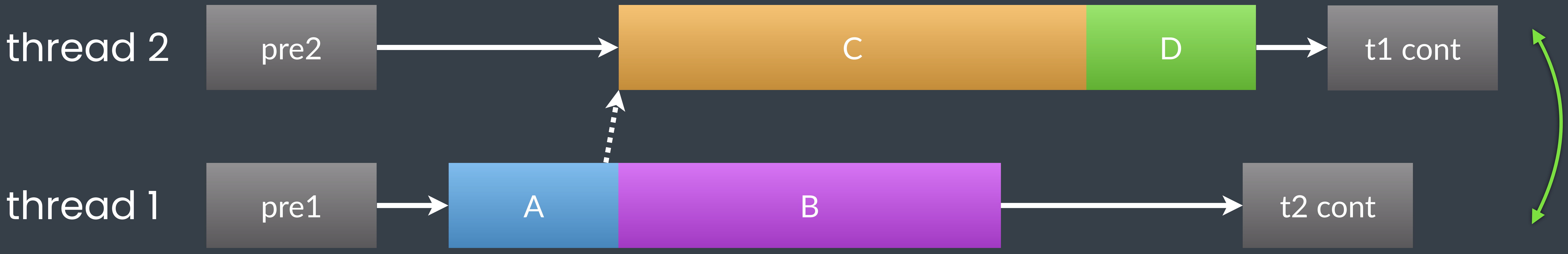


at runtime

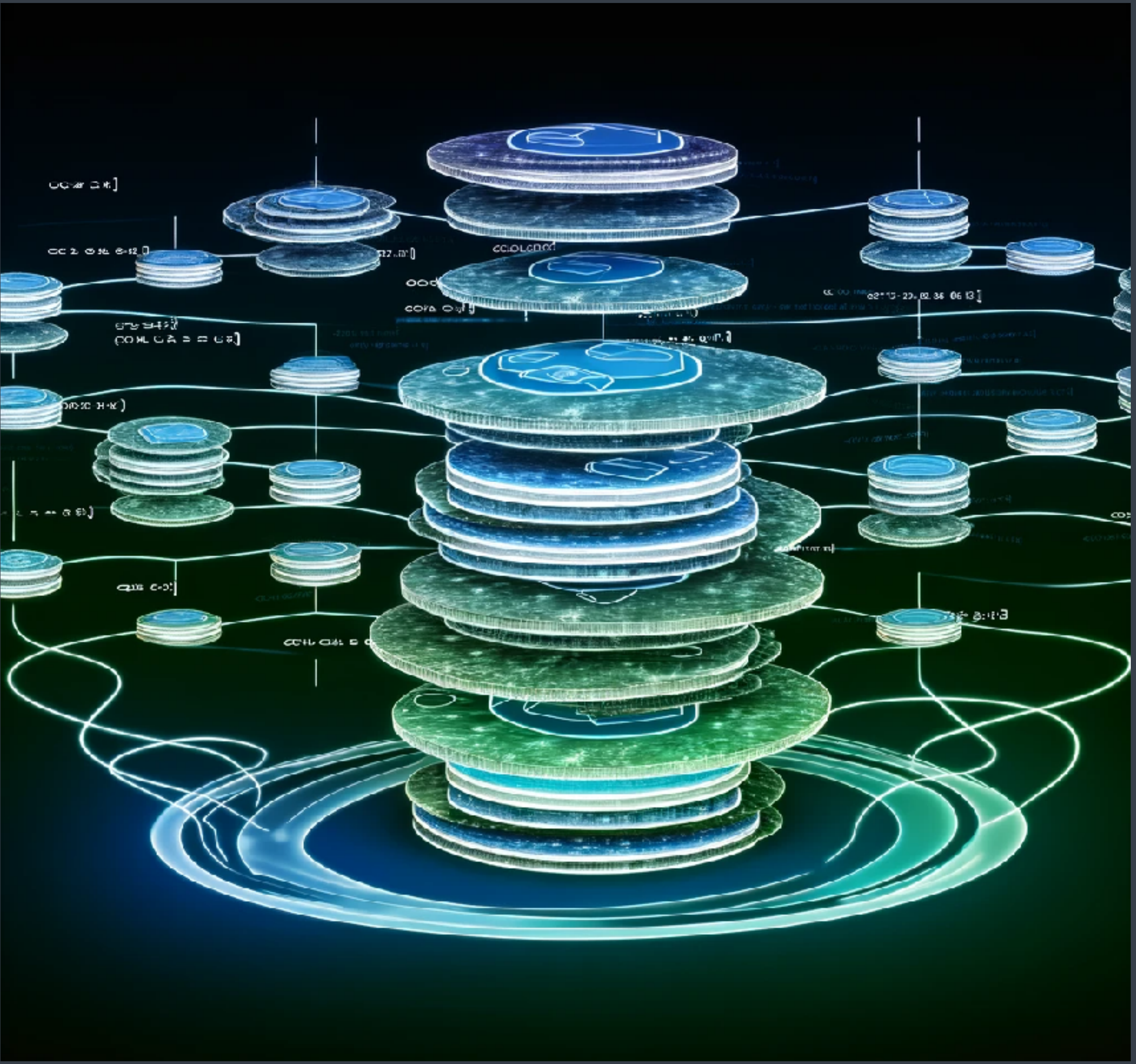
@LucTeo@techhub.social



at runtime

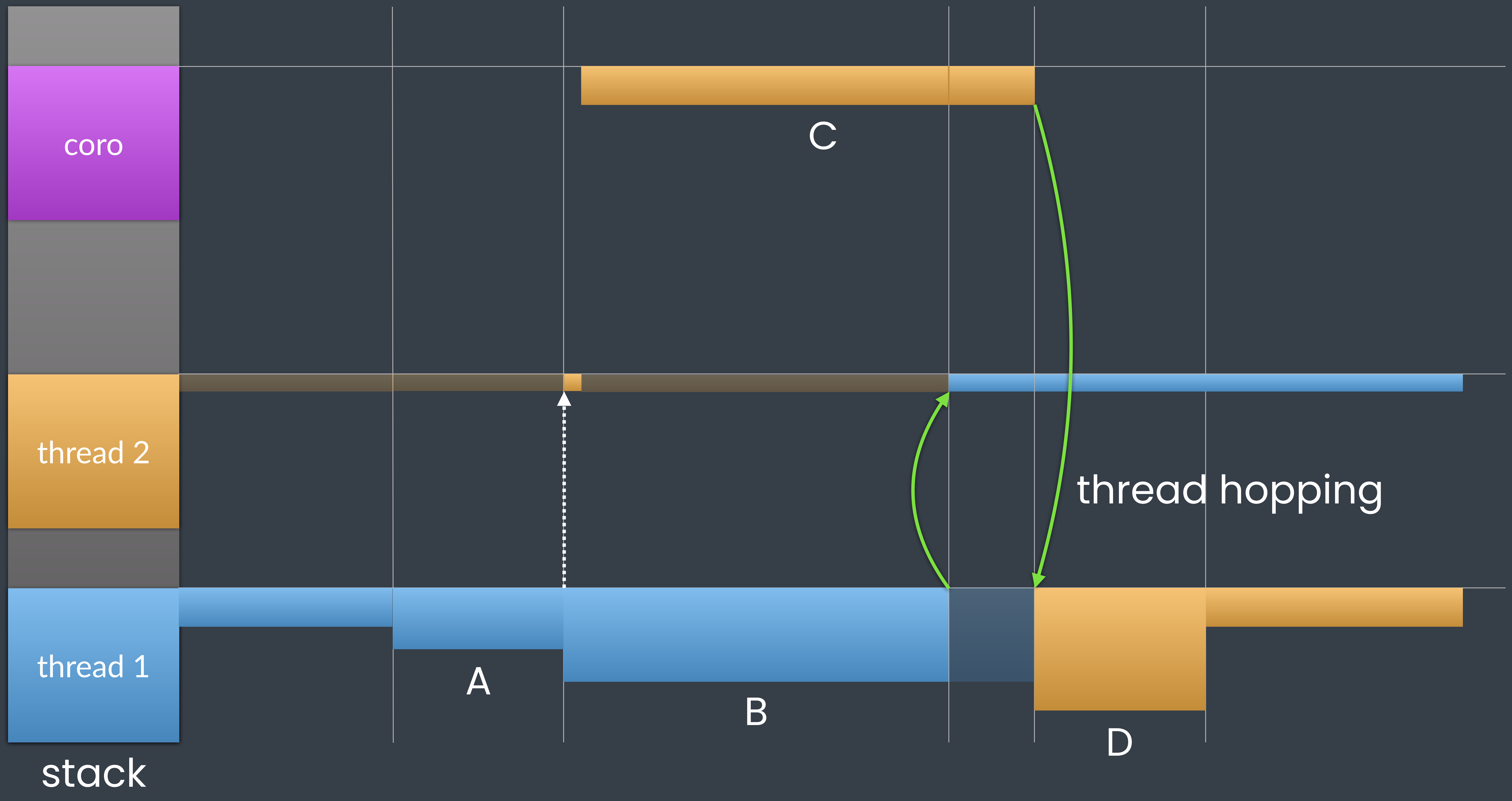


at runtime



stackfull coroutines

using `boost::context`


coro
thread 2
thread 1
stack
C
A
B
D
thread hopping

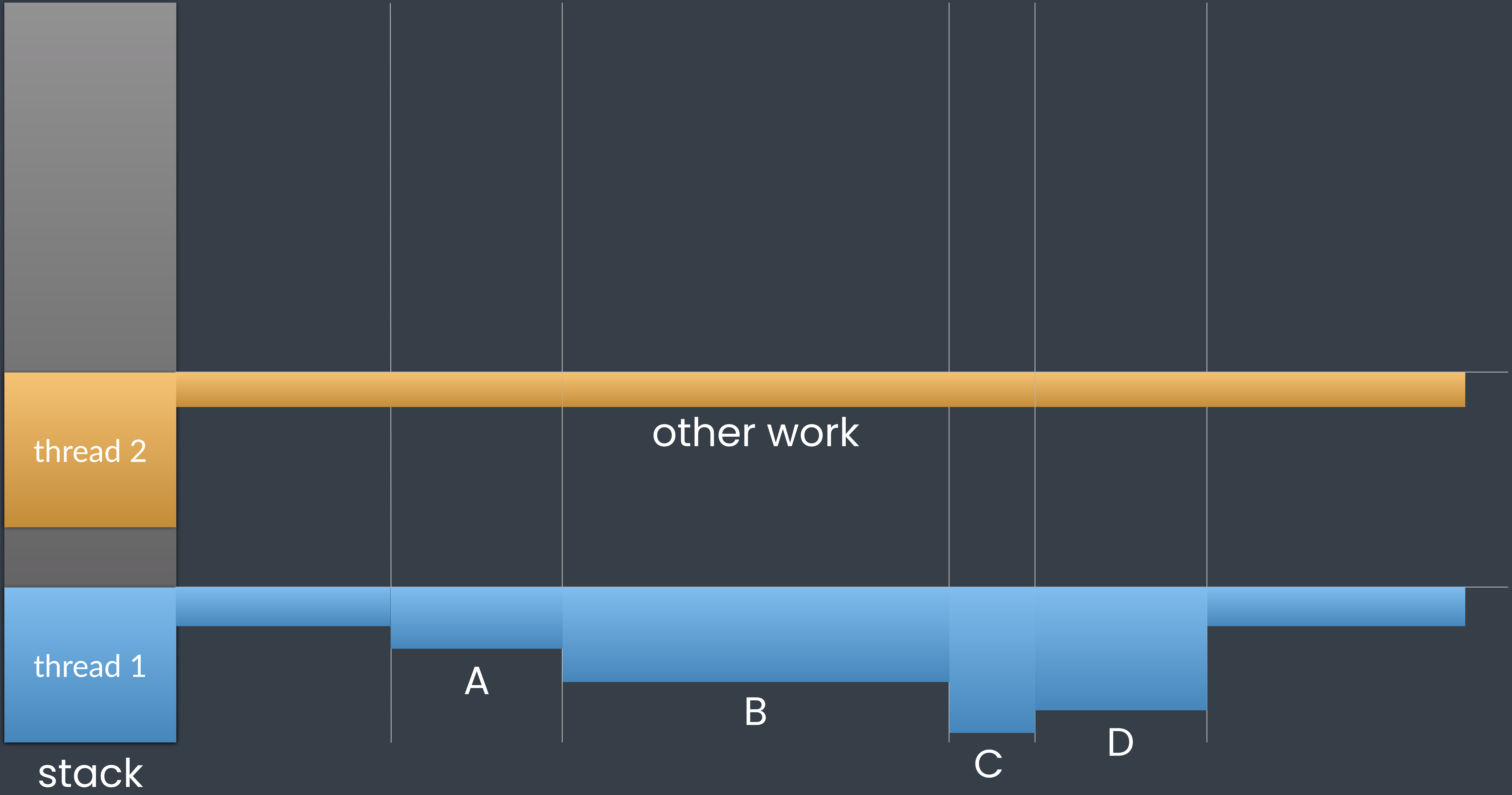
thread hopping

start thread ≠ end thread

2.

no threads when spawning

execute task inside `await`



@LucTeo@techhub.social

inplace execution

6

Early measurements

morphē

@LucTeo@techhub.social

warning!

using microbenchmarks
testing prototypes

1. skynet µbenchmark

does it scale?

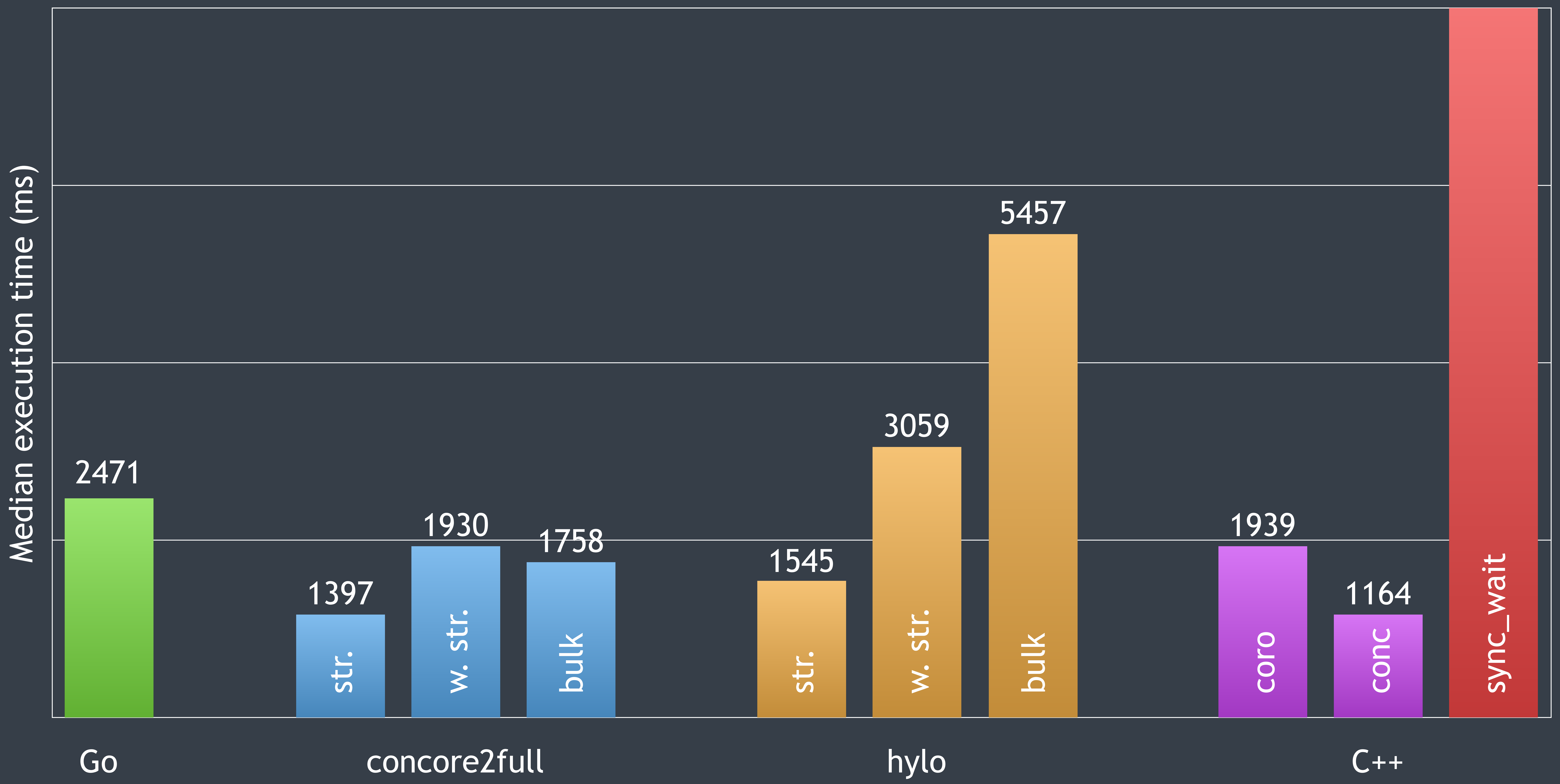
50000005000000 tasks

1. skynet µbenchmark

Creates a task, which spawns 10 new tasks, each of them spawns 10 more tasks, etc.

Ten million tasks are created on the final level.

Tasks at final level return their ordinal number; tasks at upper levels sum the values received.


Median execution time (ms)
2471
1397
1930
1758
1545
3059
5457
1939
1164
str.
w. str.
bulk
str.
w. str.
bulk
coro
conc
sync_wait
Go
concore2full
hylo
C++

interpretation

\+ can handle a large number of tasks in parallel

\+ no deadlocks

\+ good overall performance

\- slower than S/R

2. speedup

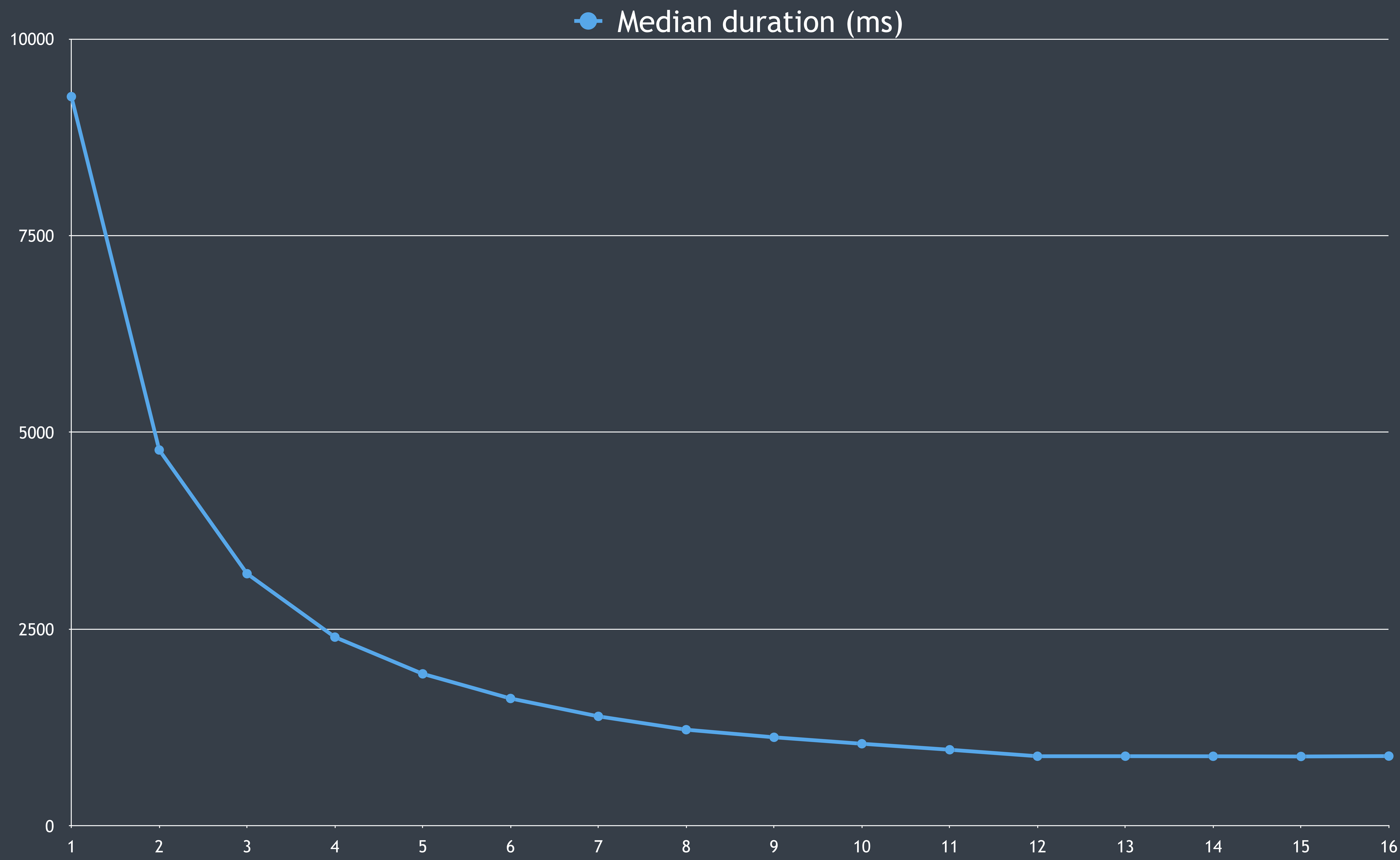
good performance?

2. speedup

compute Mandelbrot for 4096x2160, depth=1000

bulk spawn for rows

some rows are heavier than others


Median duration (ms)
10000
7500
5000
2500
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

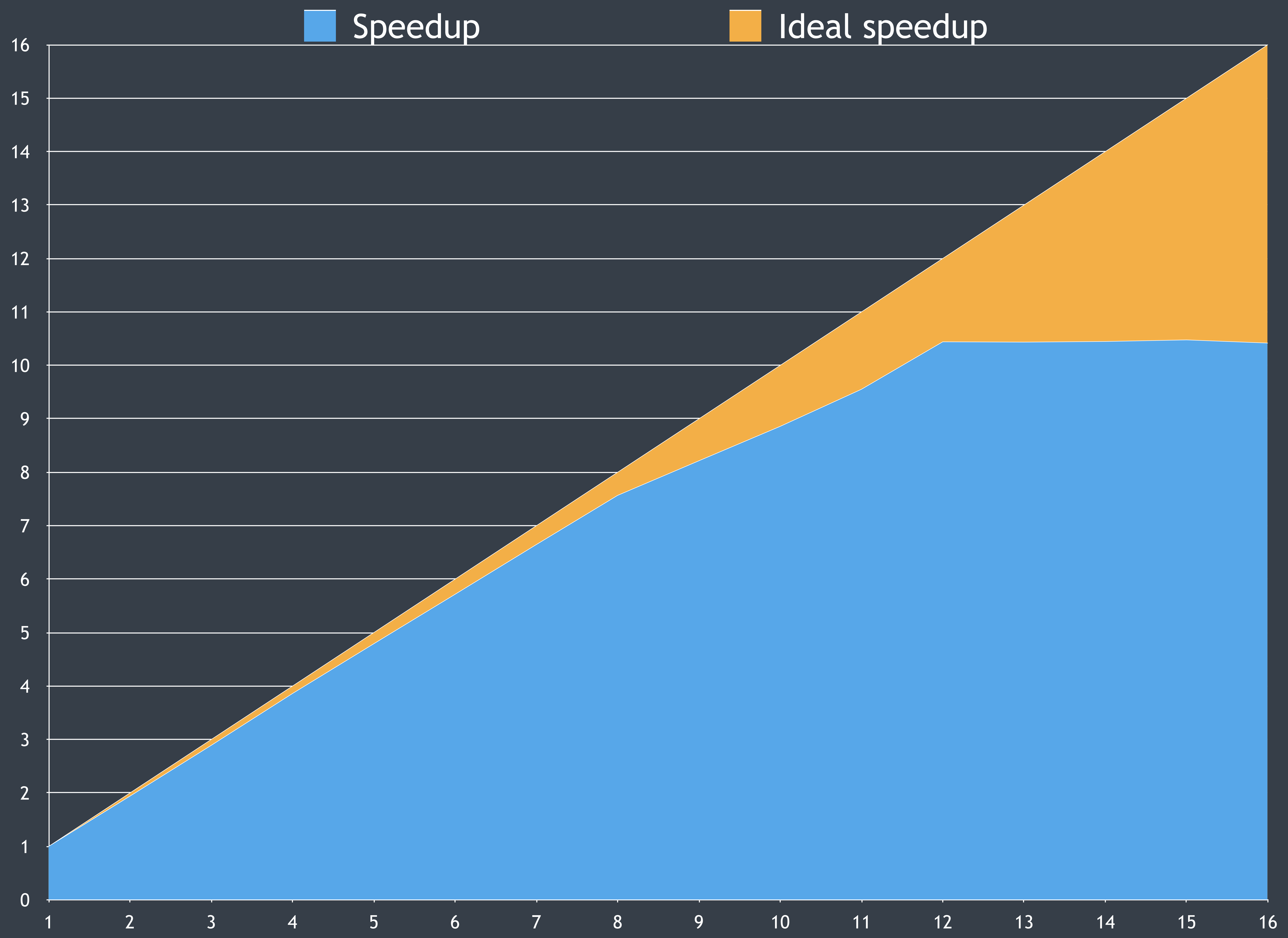


Table 1

Ideal speedup	Speedup
1	1.000
2	1.940
3	2.890
4	3.860
5	4.791
6	5.716
7	6.647
8	7.567
9	8.215
10	8.859
11	9.553
12	10.441
13	10.435
14	10.446
15	10.476
16	10.417

Analysis

7

hylē + morphē

1. modeling concurrency

express simple constraints

most of what we need

forward progress guarantee

once a task is started, it will be executed

eventually all the spawned tasks are executed

future work

conditional concurrency

1. modeling concurrency

2. safety

no race conditions

no additional synchronisation

no deadlocks

2. safety

3. performance

no blocking waits

`spawn` / `await` synchronisation

not ideal, but acceptable

relatively few `spawn` / `awaits` in the code

cost of `spawn`

memory allocation?

`callcc` — fast

some synchronisation

cost of `await`

memory allocation?

`callcc` — fast

synchronisation

(try extract task, wait for task to start)

optimisation opportunities

reuse coroutine stacks
local stacks: improve locality
optimise task handling

3. performance

@LucTeo@techhub.social

5. stack usage

stack for worker threads

very small

just jumps to a coroutine

number of coroutine stacks

~ number of worker threads
(we create a coroutine in the spawned task)

stack for await

needed to create a continuation

very small

@LucTeo@techhub.social

bottom line

low amount of stack is needed

optimisation opportunities

preallocate stacks

reuse stacks

local stacks

5. stack usage

5. interoperability

no thread-local storage

external functions calling in

may require a blocking wait

@LucTeo@techhub.social

optimisation opportunities

affinities when scheduling

5. interoperability

6. missing features

copyable futures

2 input threads, multiple output threads

cancellation

caller doesn’t need the result anymore

the work is cancelled, while caller expects results

conditional execution

spawn doesn’t immediately start work

example: implementing serialisers

other execution contexts

I/O

timers

GPUs

custom execution contexts

algorithms

8

Takeaways

hylē + morphē

holistic approach

theory

form

substance

theory

concurrency = expressing constraints

only 3 possibilities at runtime

design time: 4 basic constraints

form

easily express concurrency with `spawn` / `await`

no need for a different style

no need for additional synchronisation

structured concurrency

form

local reasoning

reasonable concurrency

substance

no race conditions
no deadlocks

substance

no blocking of threads

no oversubscription

performance scales with the number of threads

overall, fast

form

substance

theory

@LucTeo@techhub.social

reasonable concurrency

hylē + morphē

Thank You

@LucTeo@techhub.social

lucteo.ro