accu
2023

CONCURRENCY APPROACHES

PAST, PRESENT, AND FUTURE

LUCIAN RADU TEODORESCU

Concurrency Approaches:

past, present, and future

LUCIAN RADU TEODORESCU

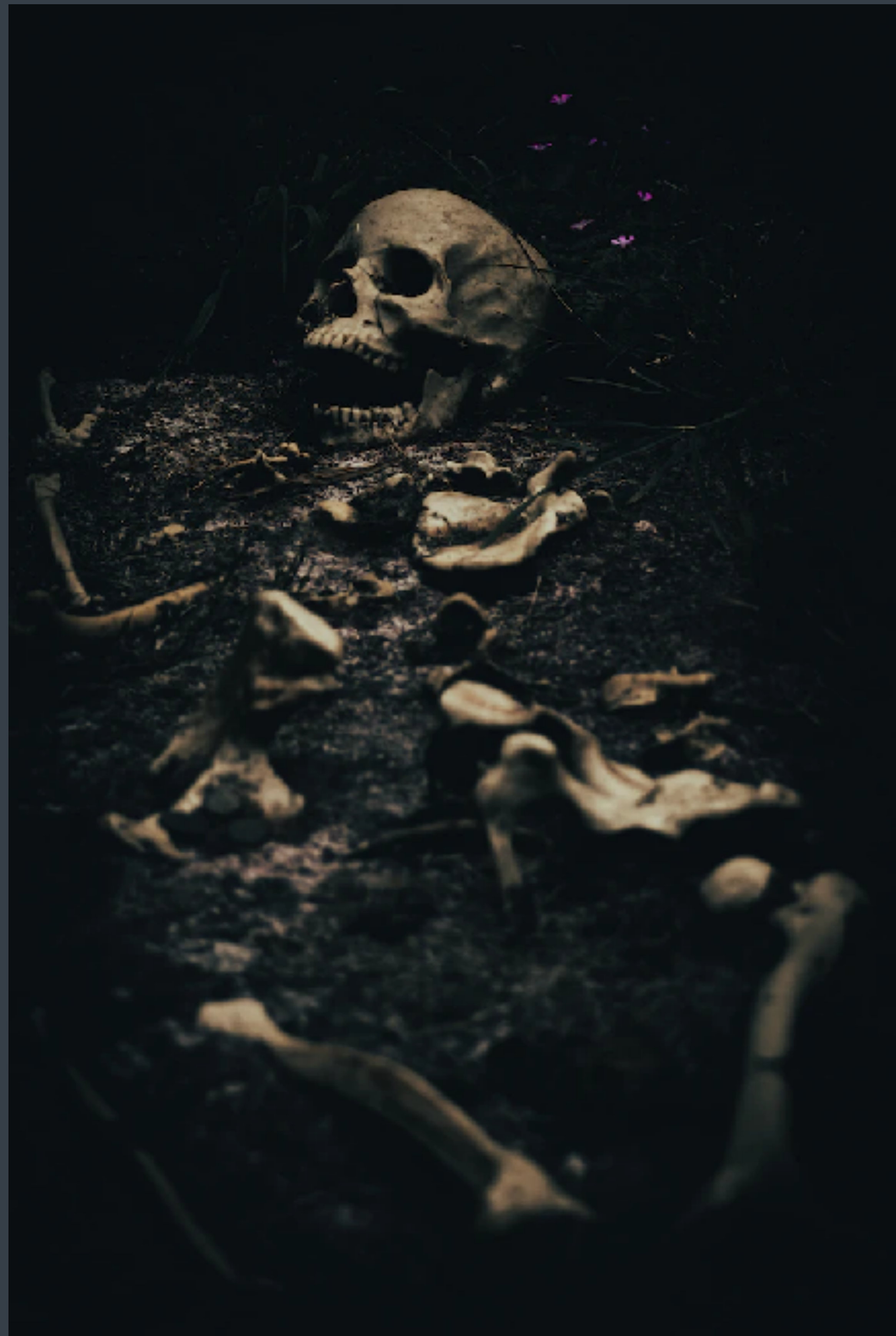
GARMIN

spoiler

concurrency is HARD

harder than **public speaking**

public speaking feared more than **death**



spoiler

concurrency is very HARD

overload 174

APRIL 2023 £4.50

In Search of a Better Concurrency Model

Lucian Radu Teodorescu presents current plans for concurrency in the VAL programming language

Drawing a Line Under Aligned Memory

Paul Floyd reminds us about various aligned memory functions

C++20 Concepts: Testing Constrained Functions

Andreas Fertig gives a worked example of testing constraints on functions or classes and other template constructs

Meta Verse

Teedy Deigh turns on, jacks in, and checks out the immersive experience

A magazine of ACCU

ISSN: 1354-3172

materials

lucteo.ro/pres/2023-accu

Agenda

Concurrency models

Concurrency goals

Past and present models

A possible future

1

Concurrency models

concurrency

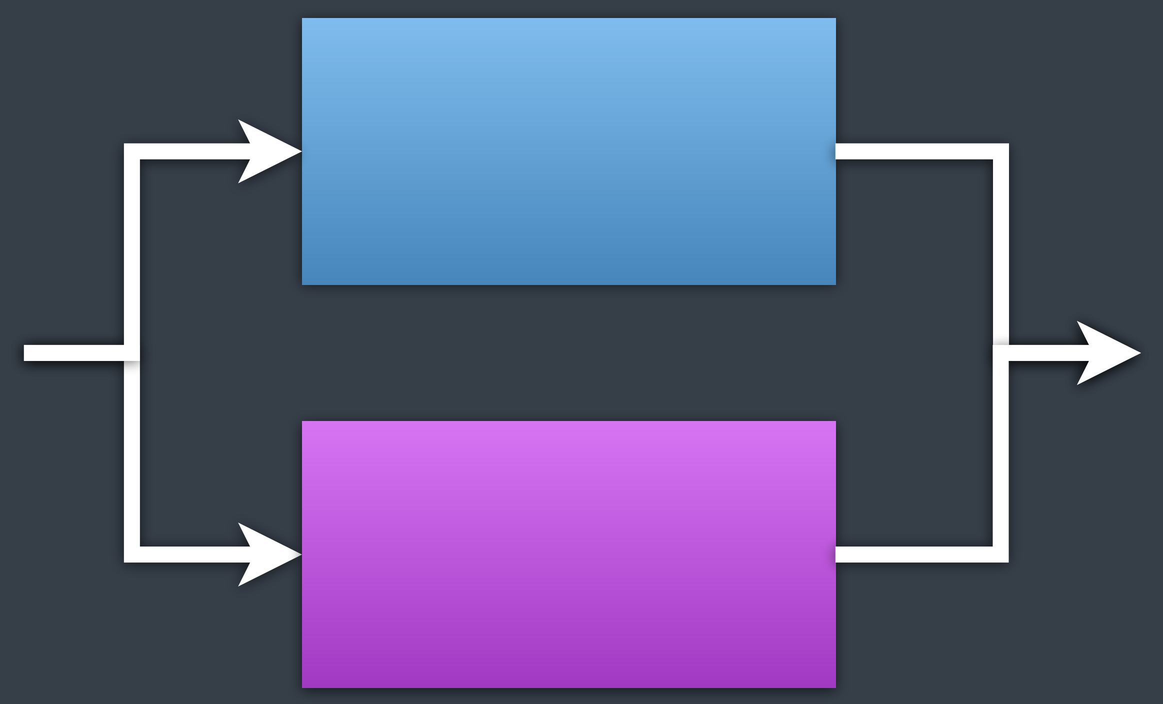
set of rules

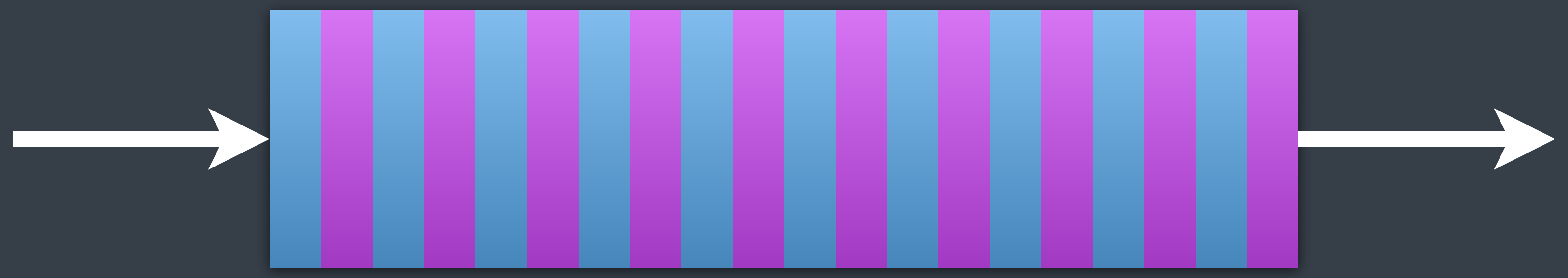
multiple activities in overlapping time periods

partial order over activities

parallelism

simultaneous execution of activities





concurrency	parallelism
design time	run time
independent of hardware	dependent on hardware

hardware threads

enable parallelism

software threads

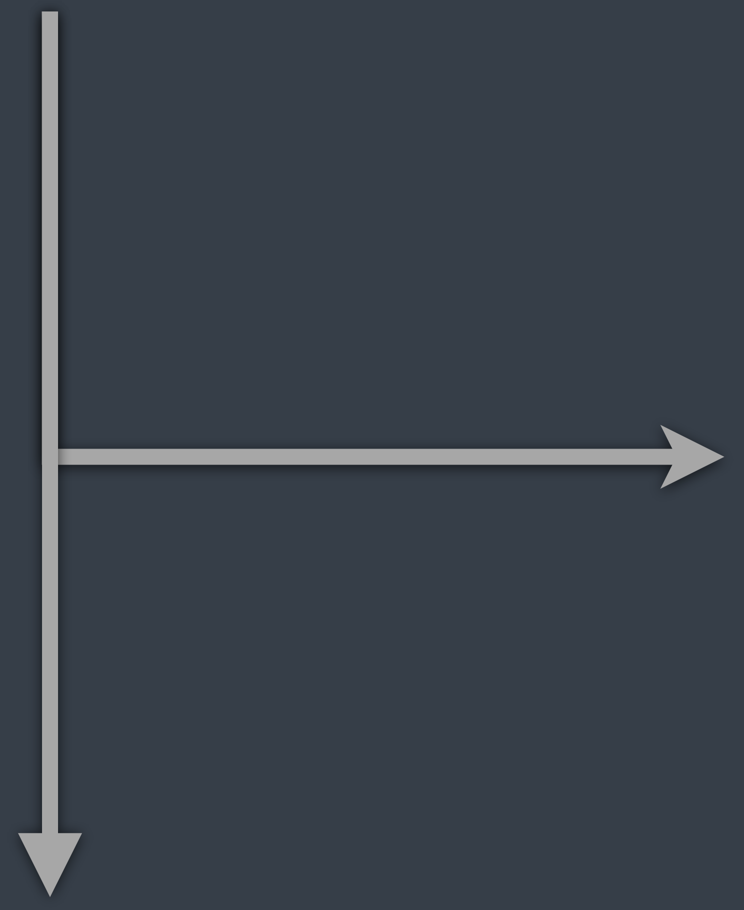
enable expressing concurrency

focus on **concurrency**

@LucTeo@techhub.social

focus on **concurrency**

rules overlapping activities



- can run concurrently
- cannot run concurrently
- before/after
- etc.

how?

example

```
pthread_create()
pthread_join()

pthread_mutex_init()
pthread_mutex_destroy()
pthread_mutex_lock()
pthread_mutex_unlock()
```

@LucTeo@techhub.social

example

can **directly** express:

run activity concurrently
join concurrent activities

don’t run concurrently

example

can **indirectly** express:

activity before other activity

example

guarantees:

none

example

possible **problems**:

deadlocks,
malign race conditions,
blocking threads more than needed,
oversubscription,
high latency,
etc.

Concurrency goals

2

@LucTeo@techhub.social

multiple levels

safety → blocker

fast → important

ease of use → nice to have / important

@LucTeo@techhub.social

SAFETY

S1: race conditions

The concurrency model
shall not allow undefined behaviour
caused by **race conditions**.

example

shared state

S2: deadlocks

The concurrency model
shall not allow deadlocks.

example

acquiring two resources

starvation?

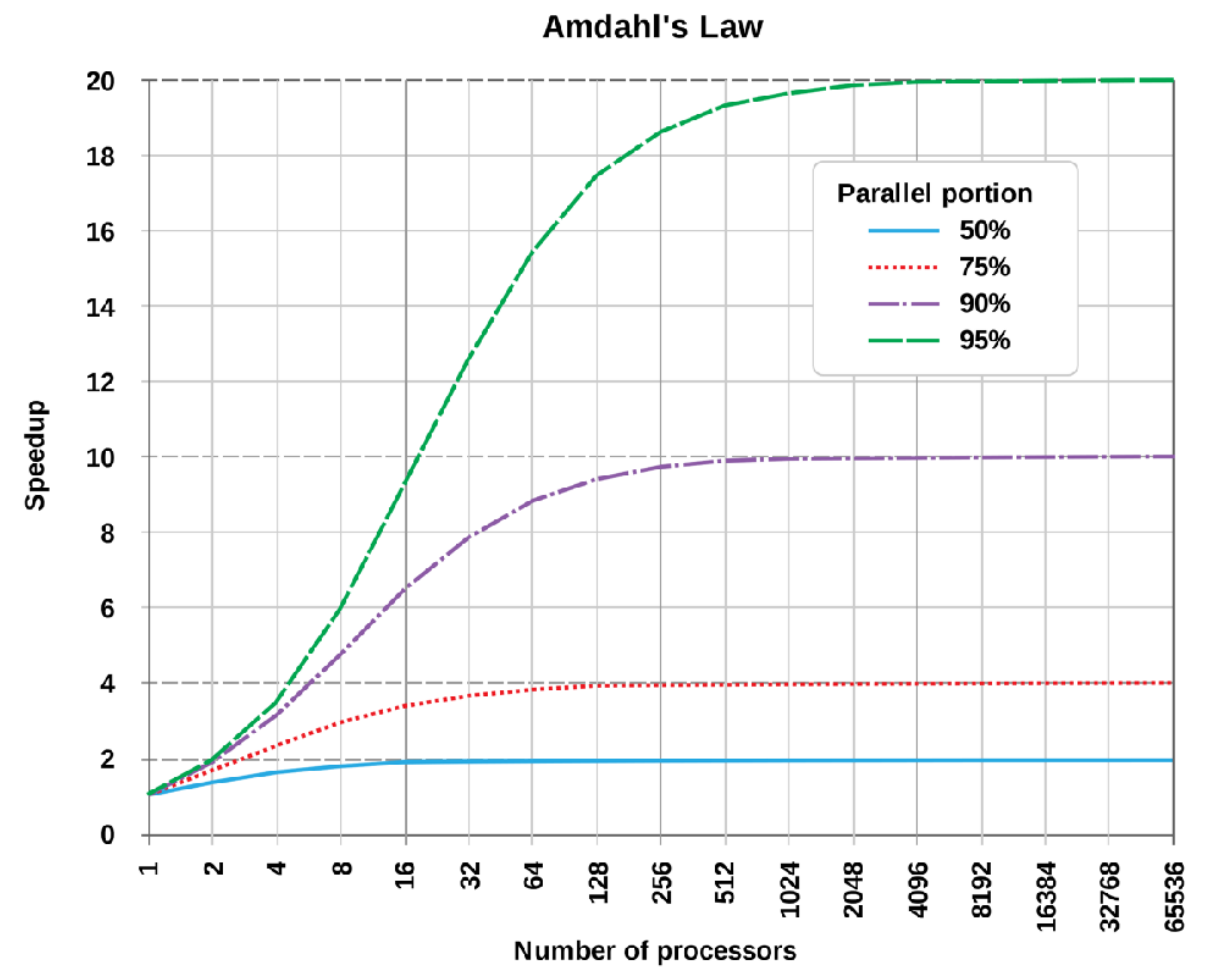
cannot guarantee lack of starvation


Какъ прямо ѣхати
живу не бывати
нѣтъ пути
ни проѣзжему
ни прохожему
ни пролетному

FAST

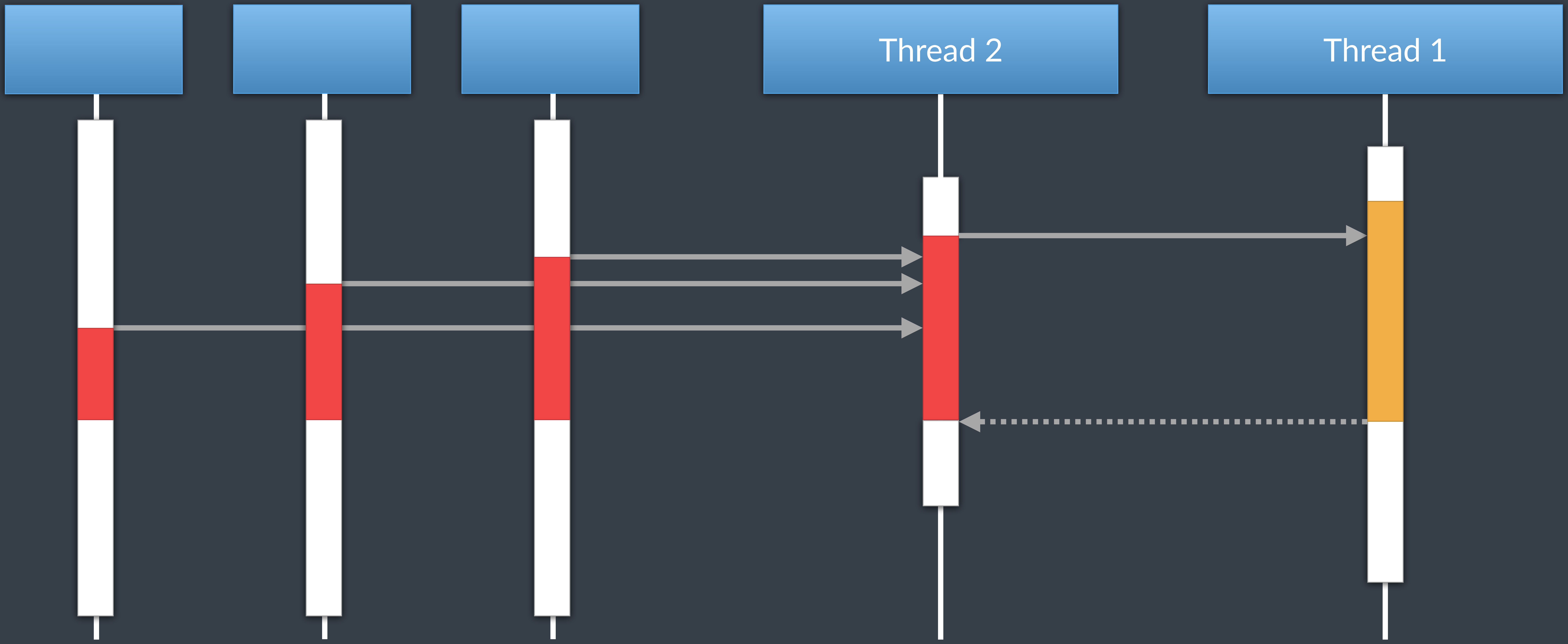
F1: scalability

For applications that express enough concurrent behaviour, the concurrency model shall guarantee that the **performance** of the application **scales with** the number of **hardware threads**.


Amdahl's Law
Speedup
0
2
4
6
8
10
12
14
16
18
20
Number of processors
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Parallel portion
50%
75%
90%
95%

F2: blocking

The concurrency model
shall not require **blocking threads**
(keeping the threads idle for longer periods of time).


Thread 2
Thread 1

can dramatically

reduce performance

mutex == bottleneck

F3: oversubscription

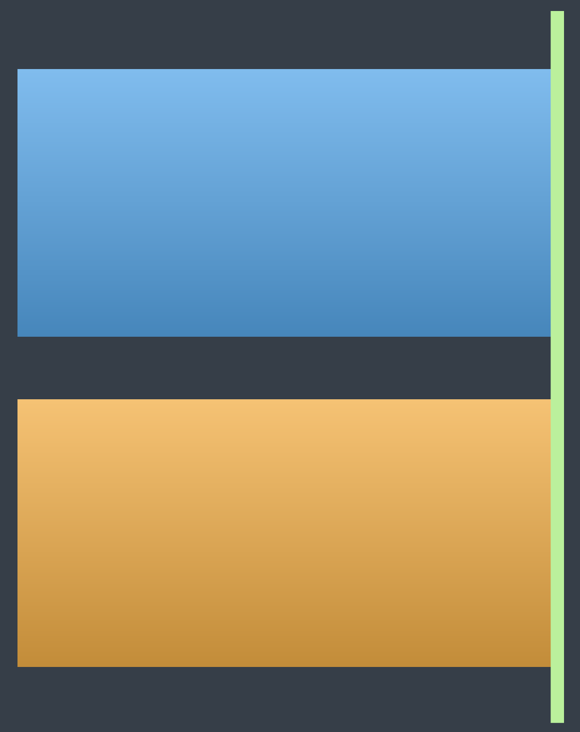
The concurrency model shall allow
limiting the oversubscription on hardware threads.

multiple threads on the same core

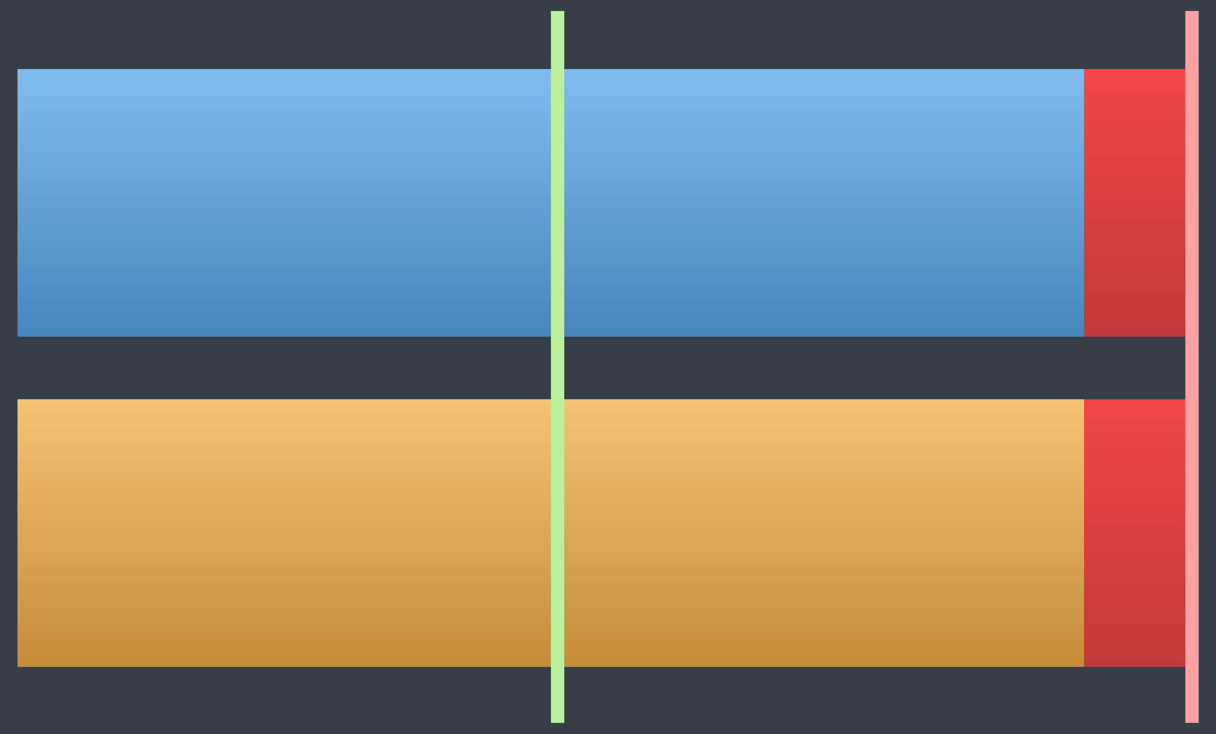
2 x 1 second

1 core

2 threads, 1 core

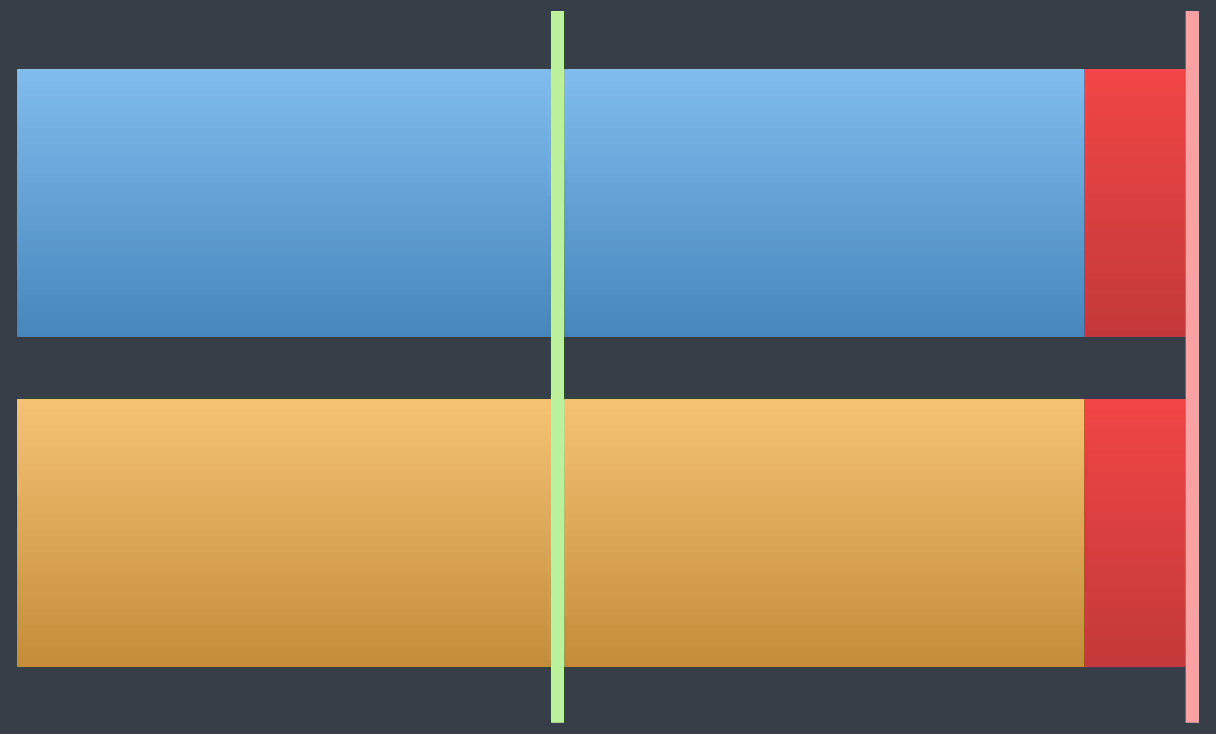


2 threads, 1 core

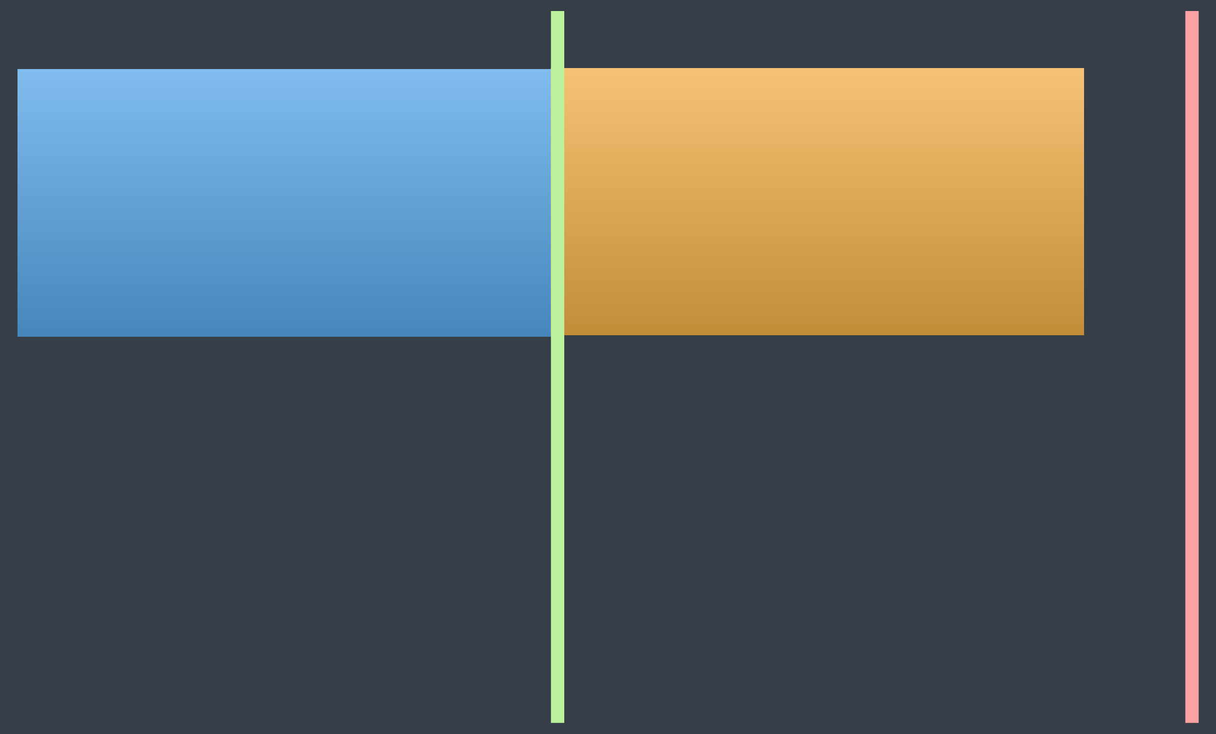


@LucTeo@techhub.social

good ?



alternative



live demo

for

a

variety

of

reasons,

this

is

a

book

eagerly

awaited

to

say

that

their

patience

has

been

rewarded

would

be

an

understatement

for a variety of reasons, this is a book eagerly awaited

to say that their patience has been rewarded would be an understatement

ready?

for
a
variety
of
reasons,
this
is
a
book
eagerly
awaited

to
say
that
their
patience
has
been
rewarded
would
be
an
understatement

important !

no blocking
no oversubscription

F4: synchronization

The concurrency model
shall not require any synchronisation code
during the execution of the tasks
(except in the concurrency-control code).

not just blocking

all synchronisation costs

F5: memory allocation

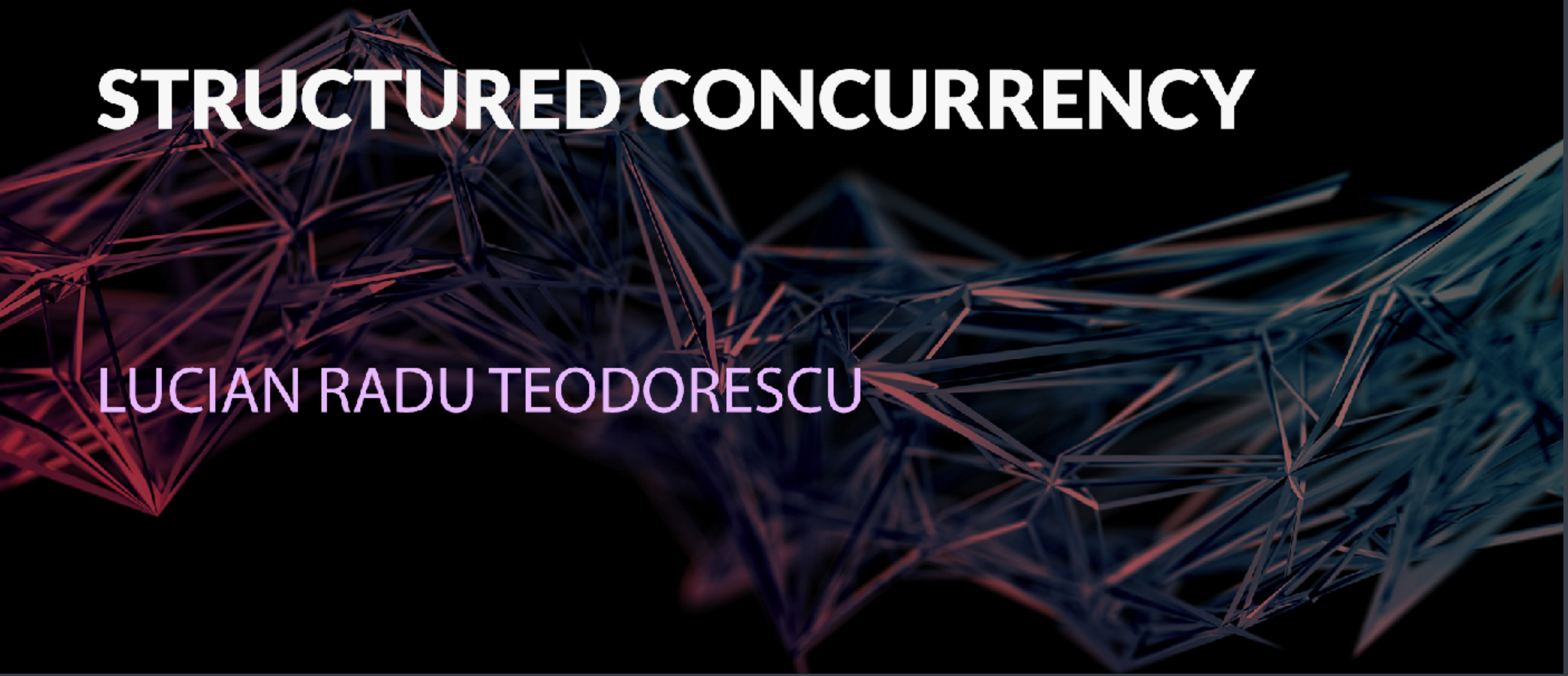
The concurrency model
shall not require **dynamic memory allocation**
(unless type-erasure is requested by the user).

EASE OF USE

E1: structured

The concurrency model shall match the description of **structured concurrency**.

accu
2022



STRUCTURED CONCURRENCY

LUCIAN RADU TEODORESCU

@LucTeo@techhub.social

structured concurrency

1. abstractions as building blocks
2. recursive decomposition
3. local reasoning
4. single entry, single exit point
5. soundness and completeness

negative example

threads and locks

E2: same syntax/semantics

Concurrent code shall be expressed using the
same syntax and semantics
as non-concurrent code.

negative example

```
auto work1() -> int;
auto work2() -> double;
auto work3() -> std::string;
auto combine_res(int i, double d, const std::string& s) -> int;

auto compute_in_parallel() -> int {
    static_thread_pool pool{8};
    ex::scheduler auto sched = pool.get_scheduler();

    ex::sender auto work =
        ex::when_all(
            ex::schedule(sched) | ex::then(work1),
            ex::schedule(sched) | ex::then(work2),
            ex::schedule(sched) | ex::then(work3)
        );
    auto [i, d, s] = std::this_thread::sync_wait(std::move(work)).value();
    return combine_res(i, d, s);
}
```

E3: function colouring

Function colouring shall not be required for expressing concurrent code.

negative example

```
public int RegularFunction() { ... }

public async Task<int> AsyncFunction() { ... }
```

negative example

```
int regular_function(int input) { ... }
```

```
task<int> coroutine_is_colored(int input) { ... }
```

main problem

it’s pervasive

E4: extra code

Except for concurrency-control code,
the user shall **not** be required to
add any extra code in concurrent code,
versus non-concurrent code.

synchronisation

only present in concurrent-control code

E5: minimal rules

The concurrency model should have a **minimum set of rules** for the user to follow to stay within the model.

traditional textbook

creating threads
problems with threading
creating locks
more problems with threading
avoiding these problems
tips for efficiency
…

Past and present models

3

@LucTeo@techhub.social

concurrency models evaluation

	S1	S2	F1	F2	F3	F4	F5	E1	E2	E3	E4	E5
threads and locks	⛔	⛔	⛔	⛔	⛔	⛔	⚠️	⛔	⛔	✅	⛔	⛔
tasks	⚠️	✅	✅	✅	✅	✅	⛔	⛔	⛔	⛔	✅	⛔
C# asynchronous model	⚠️	✅	✅	⛔	⛔	✅	⛔	✅	⚠️	⛔	✅	⛔
C++ coroutines	⚠️	✅	✅	⚠️	✅	✅	⛔	✅	⚠️	⛔	✅	⛔
senders/receivers	⚠️	✅	✅	✅	✅	✅	⚠️	✅	⛔	⛔	✅	⛔
Rust’s fearless concurrency	✅	⚠️	⚠️	⛔	⛔	⛔	⛔	⛔	⛔	✅	⛔	⛔

locks and threads

	S1	S2	F1	F2	F3	F4	F5	E1	E2	E3	E4	E5
threads and locks	⛔	⛔	⛔	⛔	⛔	⛔	⚠️	⛔	⛔	✅	⛔	⛔
tasks	⚠️	✅	✅	✅	✅	✅	⛔	⛔	⛔	⛔	✅	⛔
C# asynchronous model	⚠️	✅	✅	⛔	⛔	✅	⛔	✅	⚠️	⛔	✅	⛔
C++ coroutines	⚠️	✅	✅	⚠️	✅	✅	⛔	✅	⚠️	⛔	✅	⛔
senders/receivers	⚠️	✅	✅	✅								
Rust’s fearless concurrency	✅	⚠️	⚠️	⛔								

E3 = function colouring

tasks

	S1	S2	F1	F2	F3	F4	F5	E1	E2	E3	E4	E5
threads and locks	⛔	⛔	⛔	⛔	⛔	⛔	⚠	⛔	⛔	✅	⛔	⛔
tasks	⚠	✅	✅	✅	✅	✅	⛔	⛔	⛔	⛔	✅	⛔
C# asynchronous model	⚠	✅	✅	⛔	⛔	✅	⛔	✅	⚠	⛔	✅	⛔
C++ coroutines	⚠	✅	✅	⚠	✅	✅	⛔	✅	⚠	⛔	✅	⛔
senders/receivers	⚠	✅	✅	✅								
Rust’s fearless concurrency	✅	⚠	⚠	⛔								

F5 = memory allocation
E4 = extra code

@LucTeo@techhub.social

C# asynchronous model

	S1	S2	F1	F2	F3	F4	F5	E1	E2	E3	E4	E5
threads and locks	⛔	⛔	⛔	⛔	⛔	⛔	⚠️	⛔	⛔	✅	⛔	⛔
tasks	⚠️	✅	✅	✅	✅	✅	⛔	⛔	⛔	⛔	✅	⛔
C# asynchronous model	⚠️	✅	✅	⛔	⛔	✅	⛔	✅	⚠️	⛔	✅	⛔
C++ coroutines	⚠️	✅	✅	⚠️	✅	✅	⛔	✅	⚠️	⛔	✅	⛔
senders/receivers	⚠️	✅	✅	✅								
Rust’s fearless concurrency	✅	⚠️	⚠️	⛔								

F2 = blocking
F3 = oversubscription
F5 = memory allocation
E3 = function colouring

C++ coroutines

F5 = memory allocation
E3 = function colouring

	S1	S2	F1	F2	F3	F4	F5	E1	E2	E3	E4	E5
	⛔	⛔	⛔	⛔	⛔	⛔	⚠	⛔	⛔	✅	⛔	⛔
tasks	⚠	✅	✅	✅	✅	✅	⛔	⛔	⛔	⛔	✅	⛔
C# asynchronous model	⚠	✅	✅	⛔	⛔	✅	⛔	✅	⚠	⛔	✅	⛔
C++ coroutines	⚠	✅	✅	⚠	✅	✅	⛔	✅	⚠	⛔	✅	⛔
senders/receivers	⚠	✅	✅	✅	✅	✅	⚠	✅	⛔	⛔	✅	⛔
Rust’s fearless concurrency	✅	⚠	⚠	⛔	⛔	⛔	⛔	⛔	⛔	✅	⛔	⛔

senders/receivers

E2 = same syntax/semantics
E3 = function colouring

	S1	S2	F1	F2	F3	F4	F5	E1	E2	E3	E4	E5
	⛔	⛔	⛔	⛔	⛔	⛔	⚠️	⛔	⛔	✅	⛔	⛔
tasks	⚠️	✅	✅	✅	✅	✅	⛔	⛔	⛔	⛔	✅	⛔
C# asynchronous model	⚠️	✅	✅	⛔	⛔	✅	⛔	✅	⚠️	⛔	✅	⛔
C++ coroutines	⚠️	✅	✅	⚠️	✅	✅	⛔	✅	⚠️	⛔	✅	⛔
senders/receivers	⚠️	✅	✅	✅	✅	✅	⚠️	✅	⛔	⛔	✅	⛔
Rust’s fearless concurrency	✅	⚠️	⚠️	⛔	⛔	⛔	⛔	⛔	⛔	✅	⛔	⛔

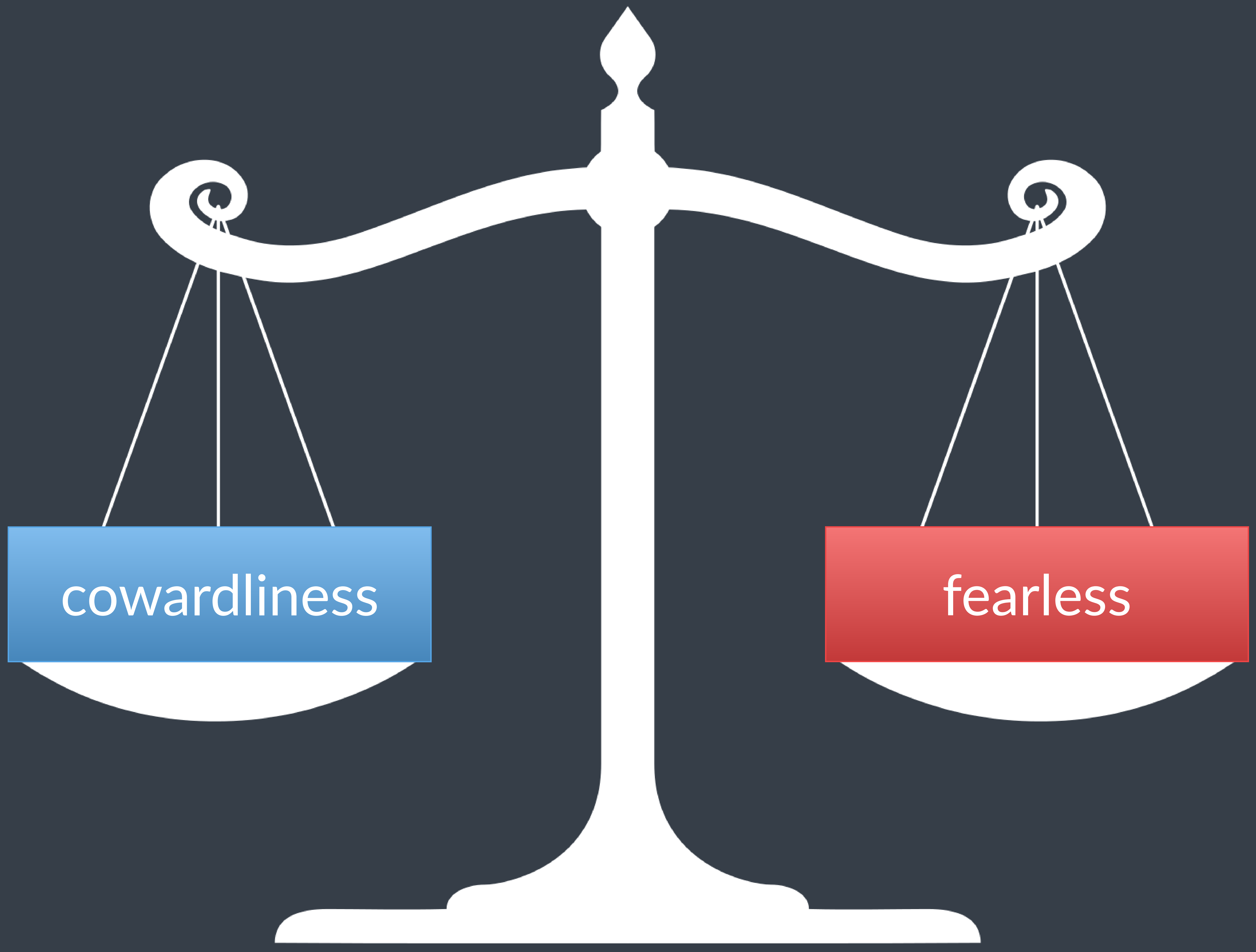
@LucTeo@techhub.social

Rust's fearless concurrency

S2 = deadlocks
F2 = blocking
F4 = syncrhonisation
F5 = memory allocation
E1 = structured
E2 = same syntax/semantics
E4 = extra code

	S1	S2	F1	F2	F3	F4	F5	E1	E2	E3	E4	E5
	⛔	⛔	⛔	⛔	⛔	⛔	⚠️	⛔	⛔	✅	⛔	⛔
	⚠️	✅	✅	✅	✅	✅	⛔	⛔	⛔	⛔	✅	⛔
	⚠️	✅	✅	⛔	⛔	✅	⛔	✅	⚠️	⛔	✅	⛔
	⚠️	✅	✅	⚠️	✅	✅	⛔	✅	⚠️	⛔	✅	⛔
	⚠️	✅	✅	✅	✅	✅	⚠️	✅	⛔	⛔	✅	⛔
Rust's fearless concurrency	✅	⚠️	⚠️	⛔	⛔	⛔	⛔	⛔	⛔	✅	⛔	⛔

@LucTeo@techhub.social



cowardliness

fearless

courage

A possible future

4

@LucTeo@techhub.social

an experiment
within in an experiment

Val programming language

fast by definition
safe by default
simple

www.val-lang.dev/

concurrency experiment

good concurrency model?

goals

safe concurrency

fast as senders/receivers (or close)

no function colouring

proposed Val concurrency model

	S1	S2	F1	F2	F3	F4	F5	E1	E2	E3	E4	E5
threads and locks	⛔	⛔	⛔	⛔	⛔	⛔	⚠️	⛔	⛔	✅	⛔	⛔
tasks	⚠️	✅	✅	✅	✅	✅	⛔	⛔	⛔	⛔	✅	⛔
C# asynchronous model	⚠️	✅	✅	⛔	⛔	✅	⛔	✅	⚠️	⛔	✅	⛔
from C++ coroutines	⚠️	✅	✅	⚠️	✅	✅	⛔	✅	✅	✅	✅	✅
from senders/receivers	✅	✅	✅	✅	✅	✅	⚠️	✅	⛔	⛔	✅	⛔
Rust's fearless concurrency	✅	⚠️	⚠️	⛔	✅	⛔	⛔	⛔	⛔	✅	⛔	⛔

starting example

```
fun long_task(input: Int) -> Int {
  var result = input
  for let i in 0 ..< 42 {
    sleep(1)
    &result += 1
  }
  return result
}

fun greeting_task() -> Int {
  print("Hello world! Have an int.")
  return 13
}

fun example() -> Int {
  var handle = long_task(input: 0)
  let x = greeting_task()
  let y = handle
  return x+y
}
```

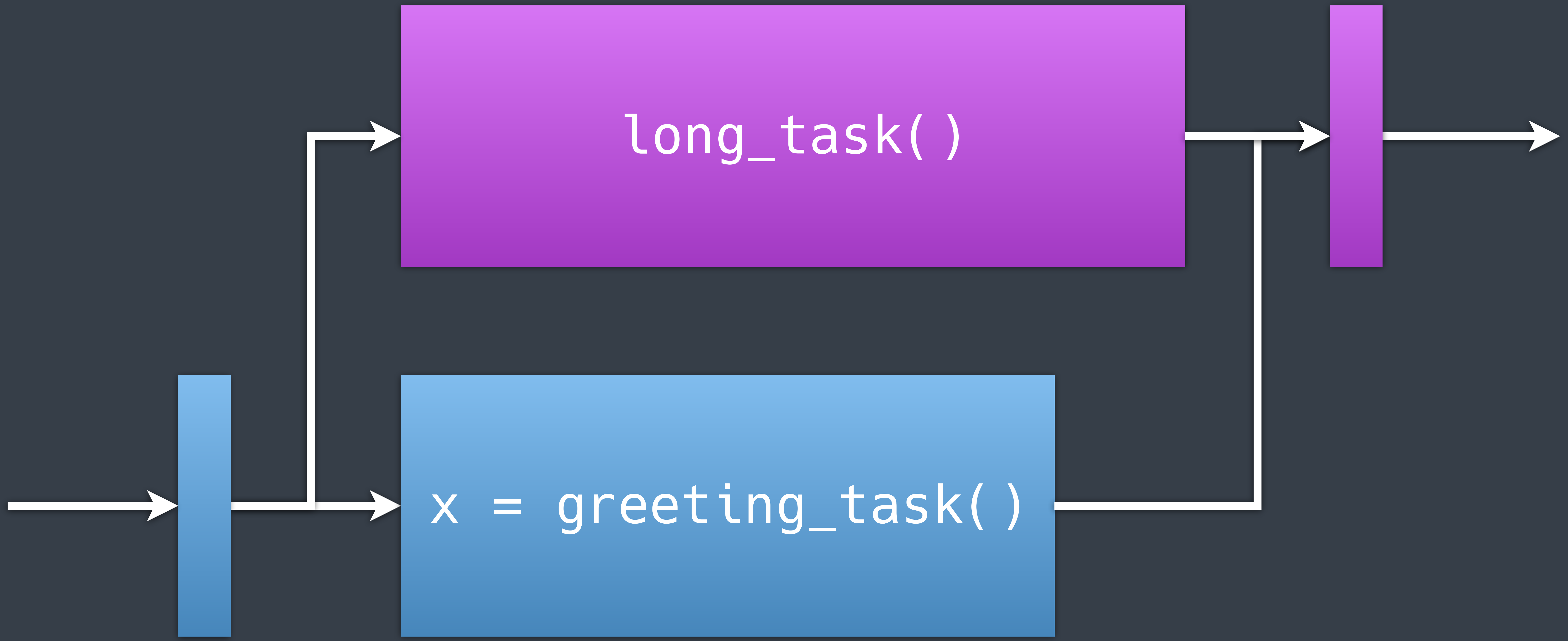
@LucTeo@techhub.social

starting example

```
fun long_task(input: Int) -> Int {
  var result = input
  for let i in 0 ..< 42 {
    sleep(1)
    &result += 1
  }
  return result
}

fun greeting_task() -> Int {
  print("Hello world! Have an int.")
  return 13
}

fun concurrency_example() -> Int {
  var handle = spawn long_task(input: 0) // create concurrent work
  let x = greeting_task()
  let y = handle.await() // joining; switching threads
  return x+y
}
```

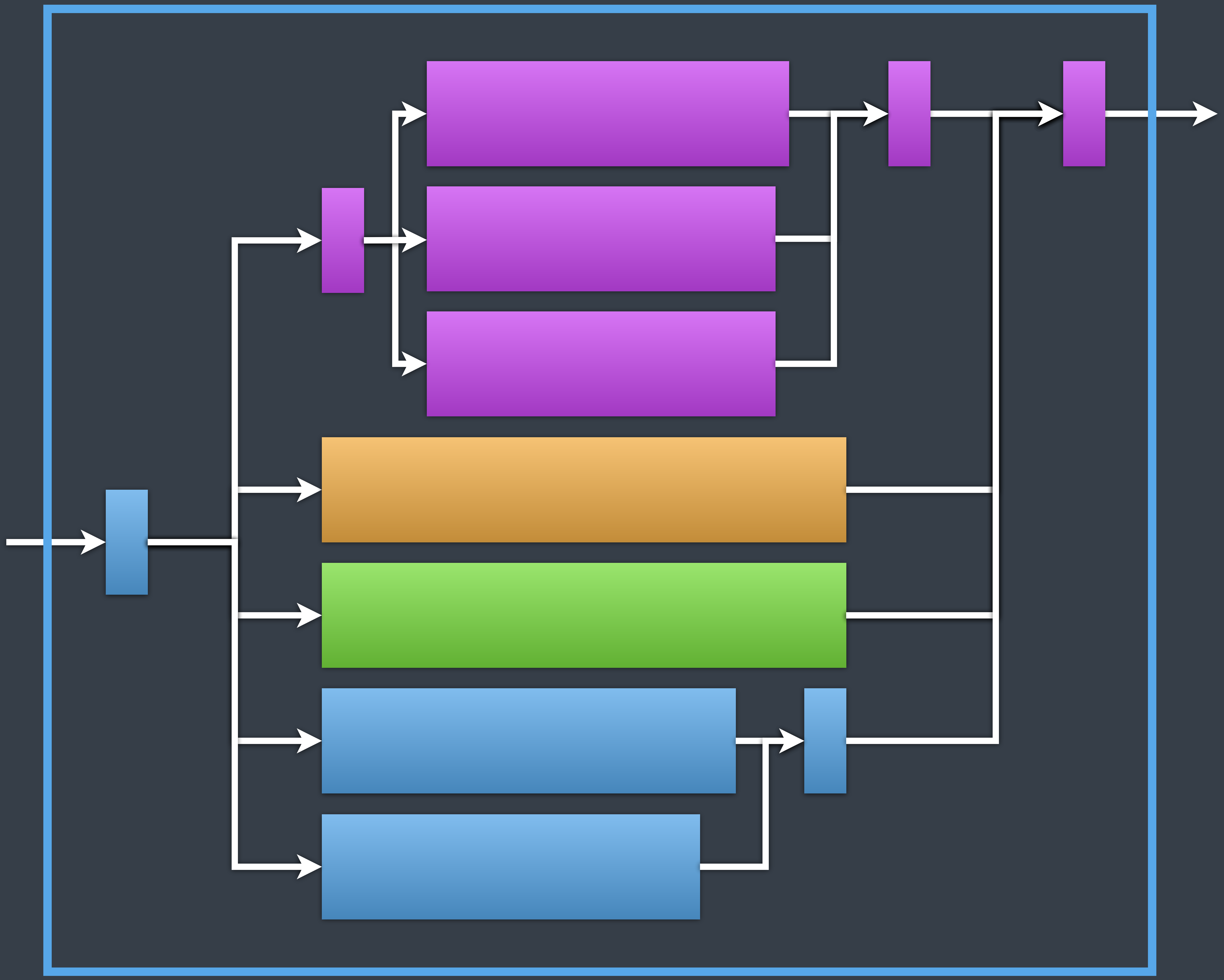
in C++, with coroutines

```cpp
int long_task(int input) {
  int result = input;
  for (int i=0; i<42; i++) {
    sleep(1);
    result += 1;
  }
  return result;
}
task<int> long_task_wrapper(int input) {
  co_await global_task_pool.enter_thread();
  co_return long_task(input);
}

int greeting_task() {
  std::cout << "Hello world! Have an int.";
  return 13;
}

task<int> concurrency_example() {
  auto handle = long_task_wrapper(0);
  auto x = greeting_task()
  auto y = co_await handle;
  co_return x+y;
}
```


long_task( )
x = greeting_task( )

computation

one entry, one exit point
can switch threads in the middle



computations, examples

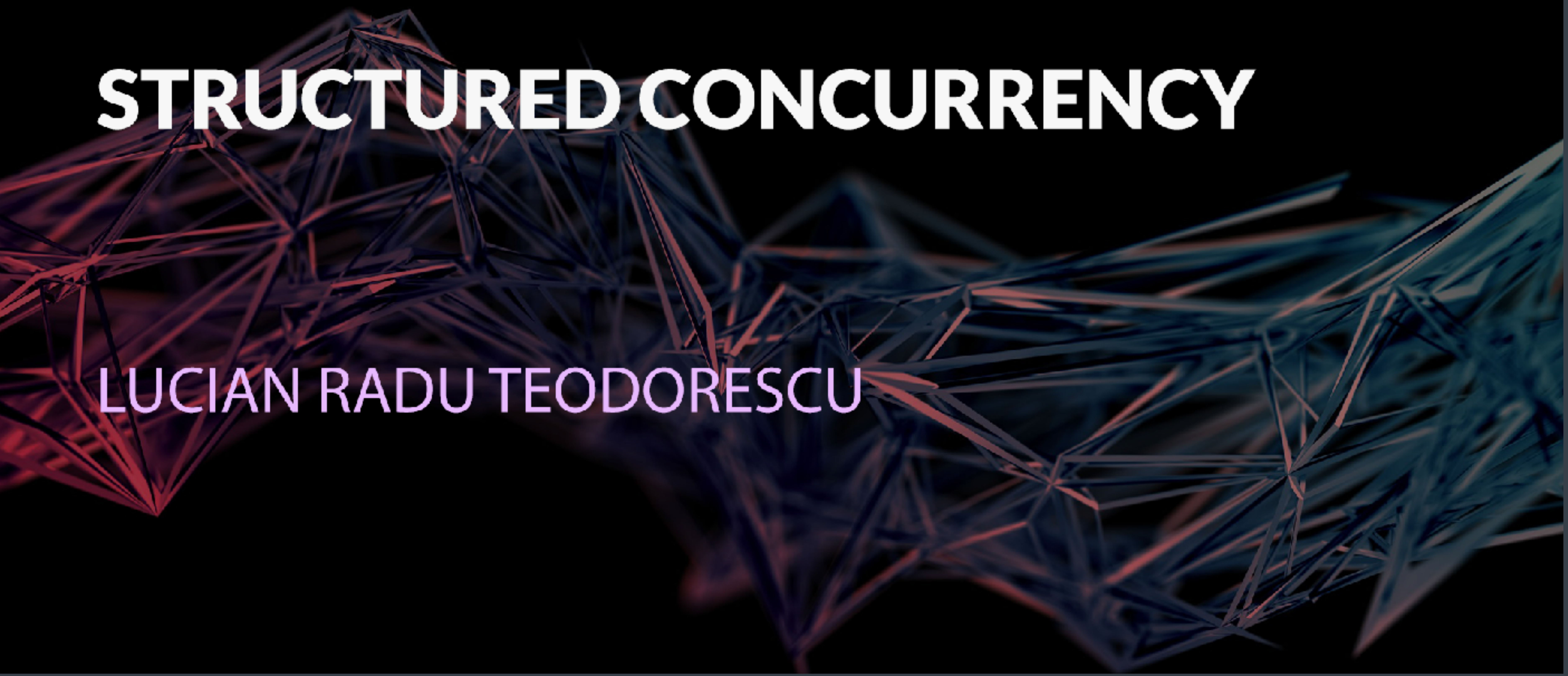
senders

C++ coroutines

C# coroutines

Val coroutines

accu 2022



STRUCTURED CONCURRENCY

LUCIAN RADU TEODORESCU

@LucTeo@techhub.social

EASE OF USE

E1: structured concurrency

use **computations** as the main abstraction

structured concurrency

1. abstractions as building blocks
2. recursive decomposition
3. local reasoning
4. single entry, single exit point
5. soundness and completeness

ACCU 2022

STRUCTURED CONCURRENCY

LUCIAN RADU TEODORESCU

@LucTeo@techhub.social

main abstraction

function = computation

coroutines

concurrency is embedded within **functions**

local reasoning

guaranteed by Val

local reasoning

```
fun long_task(input: Int) -> Int { ... }

fun greeting_task() -> Int { ... }

fun concurrency_example() -> Int {
  var handle = spawn long_task(input: 0)
  let x = greeting_task()
  let y = handle.await()
  return x+y
}
```

E2: same syntax/semantics

just use functions

in C++, with coroutines

```
int long_task(int input) {
  int result = input;
  for (int i=0; i<42; i++) {
    sleep(1);
    result += 1;
  }
  return result;
}
task<int> long_task_wrapper(int input) {
  co_await global_task_pool.enter_thread();
  co_return long_task(input);
}

int greeting_task() {
  std::cout << "Hello world! Have an int.";
  return 13;
}

task<int> concurrency_example() {
  auto handle = long_task_wrapper(0);
  auto x = greeting_task()
  auto y = co_await handle;
  co_return x+y;
}
```

in C++, with coroutines

```
int long_task(int input) {
  int result = input;
  for (int i=0; i<42; i++) {
    sleep(1);
    result += 1;
  }
  return result;
}
task<int> long_task_wrapper(int input) {
  co_await global_task_pool.enter_thread();
  co_return long_task(input);
}

int greeting_task() {
  std::cout << "Hello world! Have an int.";
  return 13;
}

task<int> concurrency_example() {
  auto handle = long_task_wrapper(0);
  auto x = greeting_task()
  auto y = co_await handle;
  co_return x+y;
}
```

in C++, with coroutines

```
int long_task(int input) {
  int result = input;
  for (int i=0; i<42; i++) {
    sleep(1);
    result += 1;
  }
  return result;
}
task<int> long_task_wrapper(int input) {
  co_await global_task_pool.enter_thread();
  co_return long_task(input);
}

int greeting_task() {
  std::cout << "Hello world! Have an int.";
  return 13;
}

task<int> concurrency_example() {
  auto handle = long_task_wrapper(0);
  auto x = greeting_task()
  auto y = co_await handle;
  co_return x+y;
}
```

@LucTeo@techhub.social

same syntax & semantics

```
fun long_task(input: Int) -> Int {
  var result = input
  for let i in 0 ..< 42 {
    sleep(1)
    &result += 1
  }
  return result
}

fun greeting_task() -> Int {
  print("Hello world! Have an int.")
  return 13
}

fun concurrency_example() -> Int {
  var handle = spawn long_task(input: 0)
  let x = greeting_task()
  let y = handle.await() // switching threads?
  return x+y
}
```

E3: function colouring

coroutines are pervasive

caller of a coroutine is usually a coroutine

C++ colouring example

```
int f1(int input) { ... }

int f2(int input) { return f1(input); }

int f3(int input) { return f2(input); }

int f4(int input) { return f3(input); }

int main() {
  f4(19);
}
```

C++ colouring example

```
task<int> f1(int input) { ... }

int f2(int input) { return co_await f1(input); }

int f3(int input) { return f2(input); }

int f4(int input) { return f3(input); }

int main() {
  f4(19);
}
```

@LucTeo@techhub.social

C++ colouring example

```cpp
task<int> f1(int input) { ... }

task<int> f2(int input) { co_return co_await f1(input); }

int f3(int input) { return co_await f2(input); }

int f4(int input) { return f3(input); }

int main() {
  f4(19);
}
```

C++ colouring example

```cpp
task<int> f1(int input) { ... }

task<int> f2(int input) { co_return co_await f1(input); }

task<int> f3(int input) { co_return co_await f2(input); }

int f4(int input) { return co_await f3(input); }

int main() {
  f4(19);
}
```

@LucTeo@techhub.social

C++ colouring example

```cpp
task<int> f1(int input) { ... }

task<int> f2(int input) { co_return co_await f1(input); }

task<int> f3(int input) { co_return co_await f2(input); }

task<int> f4(int input) { co_return co_await f3(input); }

int main() {
  f4(19);
}
```

C++ colouring example

```
task<int> f1(int input) { ... }

task<int> f2(int input) { co_return co_await f1(input); }

task<int> f3(int input) { co_return co_await f2(input); }

task<int> f4(int input) { co_return co_await f3(input); }

int main() {
  sync_wait( f4(19) );
}
```

concurrent intensive app

most of the functions are coroutines

no colouring in Val

```
fun f1(input: Int) -> Int { ... }

fun f2(input: Int) -> Int { f1(input) }

fun f3(input: Int) -> Int { f2(input) }

fun f4(input: Int) -> Int { f3(input) }

fun main() -> Int { f4(19) }
```

E4: extra code

no need for synchronisation code

concurrency-control code

start of activity
end of activity

```
spawn f
```

```
schedule(global_scheduler) | then(f)
```

or

```
schedule(global_scheduler) | let_value(f)
```

`h.await()`

`when_all(cur_work, h_work)`

plus generalisations

different schedulers
spawning/joining multiple activities

E5: minimal rules

The concurrency model should have a **minimum set of rules** for the user to follow to stay within the model.

minimal rules

how work is spanned
how work is joined

minimal rules

no synchronisation
no sync/async distinction


Какъ прямо ѣхати
живу не бывати
нѣтъ пути
ни проѣзжему
ни прохожему
ни пролетно

FAST

from coroutines

F1: scalability

F2: blocking

F3: oversubscription

F4: synchronization

memory allocation

C++ coroutines need dynamic memory for stack frames

every call

coroutines all the way down

doesn’t work

coroutines

stackless

stores one frame

suspend in called function

C++ coroutines

stackfull

stores the entire stack

suspend anywhere

Boost coroutine2

costs

stackless

calling

stackfull

spawning

exceeding stack space

to be confirmed!

success of the model

cost of using stackfull coroutines

SAFETY

S1: race conditions

by construction of Val
law of exclusivity

S2: deadlocks

no synchronisation primitives
activities don’t have cylces

Takeaways

5

@LucTeo@techhub.social

a **framework for evaluating** concurrency models

safety / fast / ease of use

proposed concurrency model

big advancements in usability

safety by default

fast?

more research

improving stackfull coroutines

also, on Val

ease of use

safety by default

fast?

on C++

new perspectives

Thank You

@LucTeo@techhub.social

lucteo.ro