Threads Considered Harmful

Lucian Radu Teodorescu

Threads Considered Harmful

lucteo.ro/pres/2021-accu/

LUCIAN RADU TEODORESCU

Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of($A1$, $A2$, $\cdots$, An)), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1$, $B2 \rightarrow E2$, $\cdots$, $Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

Dijkstra, 1968

My second remark is that our intellectual powers are geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed

Edgar Dijkstra

@LucT3o

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of($A1, A2, \cdots, An$)), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1, B2 \rightarrow E2, \cdots, Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible

Edgar Dijkstra

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** *B* **then** *A*), alternative clauses (**if** *B* **then** *A1* **else** *A2*), choice clauses as introduced by C. A. R. Hoare (case[i] of (*A1*, *A2*, ⋯ , *An*)), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1$, $B2 \rightarrow E2$, ⋯ , $Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, *n* equals the number of persons in the room minus one!

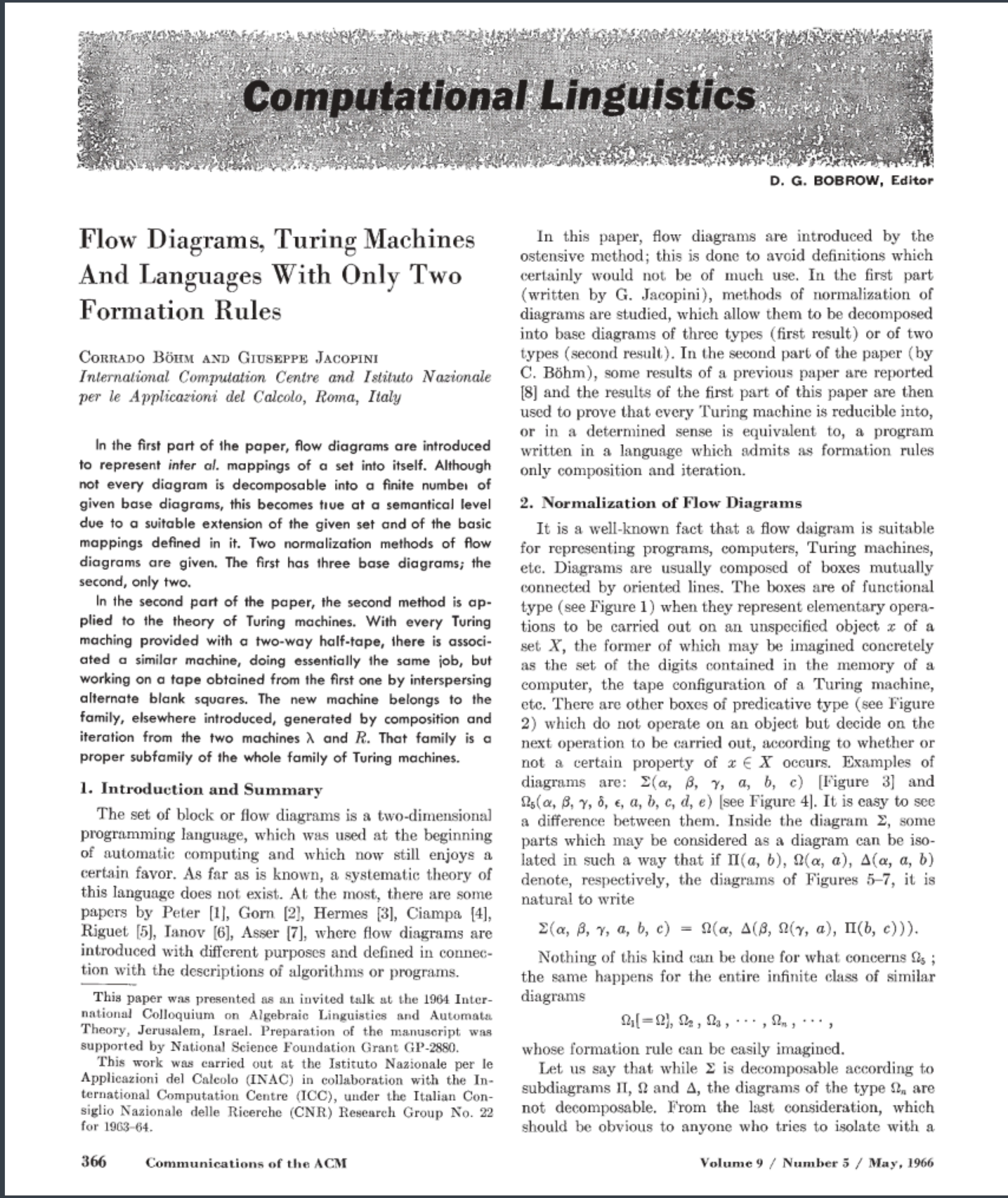
The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

Böhm–Jacopini theorem
1966

all programs can be represented with *sequence, selection* and *repetition*

flow diagrams



D. G. BOBROW, Editor

Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules

Corrado Böhm and Giuseppe Jacopini
International Computation Centre and Istituto Nazionale per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent *inter al.* mappings of a set into itself. Although not every diagram is decomposable into a finite numbe of given base diagrams, this becomes tiue at a semantical level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two.

In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing maching provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interspersing alternate blank squares. The new machine belongs to the family, elsewhere introduced, generated by composition and iteration from the two machines λ and R. That family is a proper subfamily of the whole family of Turing machines.

1. Introduction and Summary

The set of block or flow diagrams is a two-dimensional programming language, which was used at the beginning of automatic computing and which now still enjoys a certain favor. As far as is known, a systematic theory of this language does not exist. At the most, there are some papers by Peter [1], Gorn [2], Hermes [3], Ciampa [4], Riguet [5], Ianov [6], Asser [7], where flow diagrams are introduced with different purposes and defined in connection with the descriptions of algorithms or programs.

This paper was presented as an invited talk at the 1964 International Colloquium on Algebraic Linguistics and Automata Theory, Jerusalem, Israel. Preparation of the manuscript was supported by National Science Foundation Grant GP-2880.

This work was carried out at the Istituto Nazionale per le Applicazioni del Calcolo (INAC) in collaboration with the International Computation Centre (ICC), under the Italian Consiglio Nazionale delle Ricerche (CNR) Research Group No. 22 for 1963–64.

In this paper, flow diagrams are introduced by the ostensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported [8] and the results of the first part of this paper are then used to prove that every Turing machine is reducible into, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration.

2. Normalization of Flow Diagrams

It is a well-known fact that a flow daigram is suitable for representing programs, computers, Turing machines, etc. Diagrams are usually composed of boxes mutually connected by oriented lines. The boxes are of functional type (see Figure 1) when they represent elementary operations to be carried out on an unspecified object x of a set X, the former of which may be imagined concretely as the set of the digits contained in the memory of a computer, the tape configuration of a Turing machine, etc. There are other boxes of predicative type (see Figure 2) which do not operate on an object but decide on the next operation to be carried out, according to whether or not a certain property of $x \in X$ occurs. Examples of diagrams are: $\Sigma(\alpha, \beta, \gamma, a, b, c)$ [Figure 3] and $\Omega_5(\alpha, \beta, \gamma, \delta, \epsilon, a, b, c, d, e)$ [see Figure 4]. It is easy to see a difference between them. Inside the diagram Σ, some parts which may be considered as a diagram can be isolated in such a way that if $\Pi(a, b)$, $\Omega(\alpha, a)$, $\Delta(\alpha, a, b)$ denote, respectively, the diagrams of Figures 5–7, it is natural to write

$$\Sigma(\alpha, \beta, \gamma, a, b, c) = \Omega(\alpha, \Delta(\beta, \Omega(\gamma, a), \Pi(b, c))).$$

Nothing of this kind can be done for what concerns Ω_5; the same happens for the entire infinite class of similar diagrams

$$\Omega_1[=\Omega], \Omega_2, \Omega_3, \cdots, \Omega_n, \cdots,$$

whose formation rule can be easily imagined.

Let us say that while Σ is decomposable according to subdiagrams Π, Ω and Δ, the diagrams of the type Ω_n are not decomposable. From the last consideration, which should be obvious to anyone who tries to isolate with a

structured programming

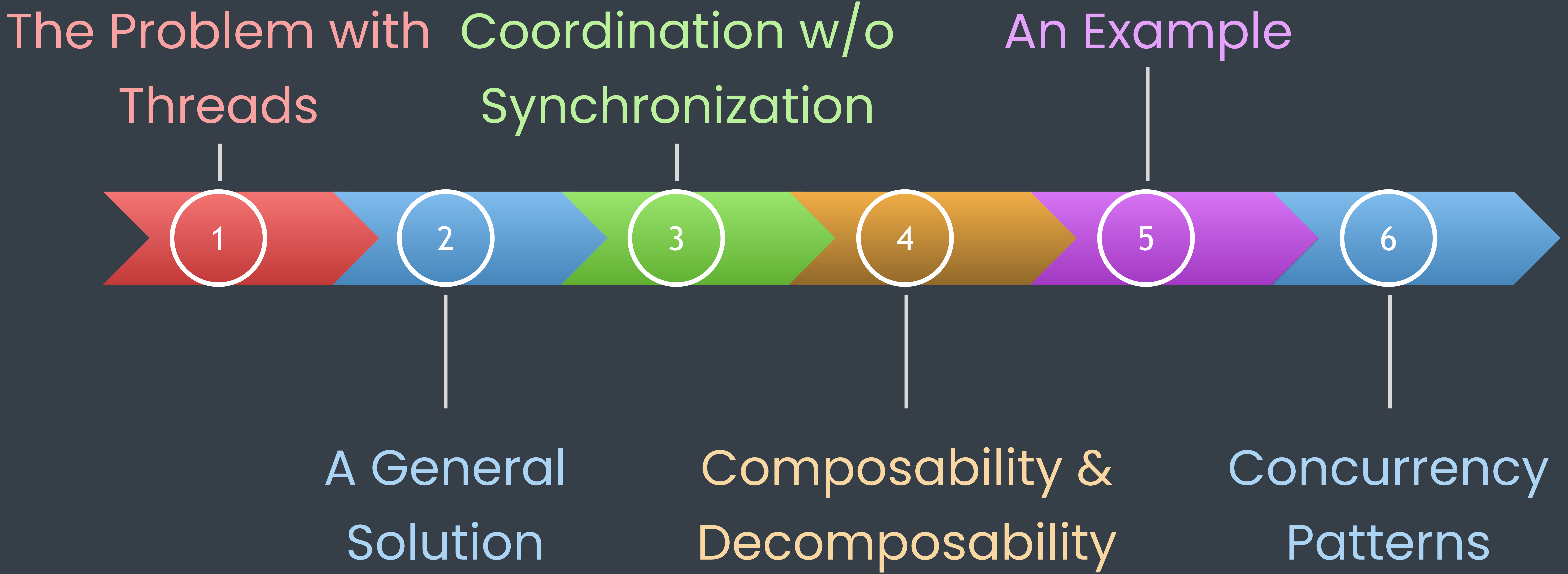
this talk

threads are like gotos

reasoning with threads is hard

finding a general alternative to threads

Agenda


The Problem with Threads
1
A General Solution
2
Coordination w/o Synchronization
3
Composability & Decomposability
4
An Example
5
Concurrency Patterns
6

The Problem with Threads

1

threads

raw threads + syncrhonization (locks)

problems with threads

performance
understandability
thread safety
composability

you are likely to get it wrong!

performance
understandability
thread safety
composability

1. cost of locking

synchronization

==

locks

==

bottlenecks

I've often joked that
instead of picking up Djikstra's
cute acronym
we should have called the basic
synchronization object
"**the bottleneck**"

David Butenhof

GAMES | BROWSE THESAURUS | WORD OF THE DAY | WORDS AT PLAY

Merriam-Webster SINCE 1828

bottleneck

Dictionary | Thesaurus

bottleneck noun

Definition of *bottleneck* (Entry 2 of 3)

1 a : a narrow route

b : a point of traffic congestion

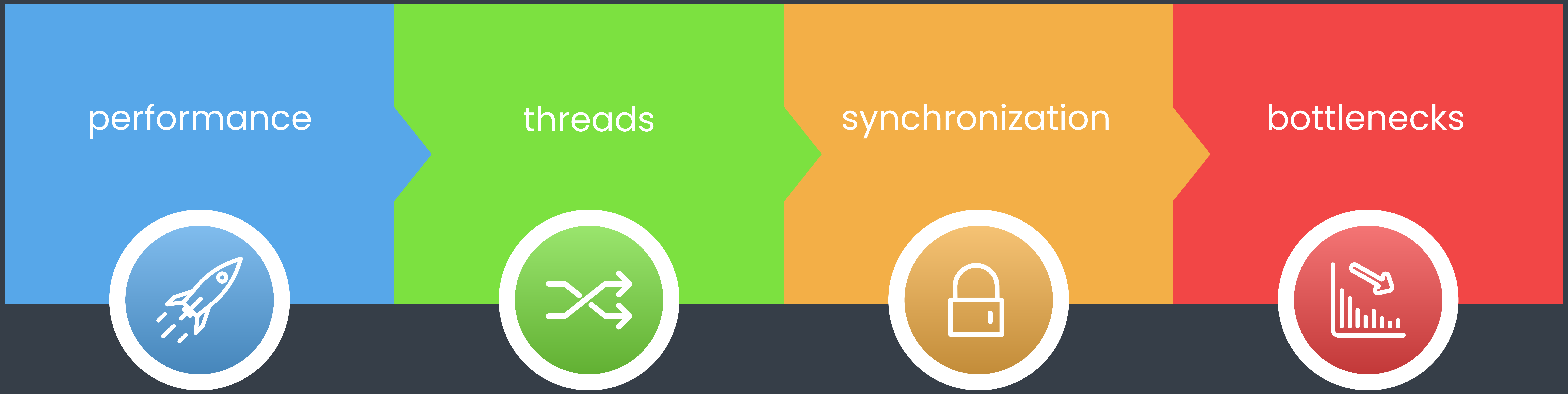
2 a : someone or something that retards or halts free movement and progress

b : IMPASSE

c : a dramatic reduction in the size of a population (as of a species) that results in a decrease in genetic variation

3 : a style of guitar playing in which glissando effects are produced by sliding an object (such as a knife blade or the neck of a bottle) along the strings

— called also *bottleneck guitar*


performance
threads
synchronization
bottlenecks

locks do not scale

chain of locks

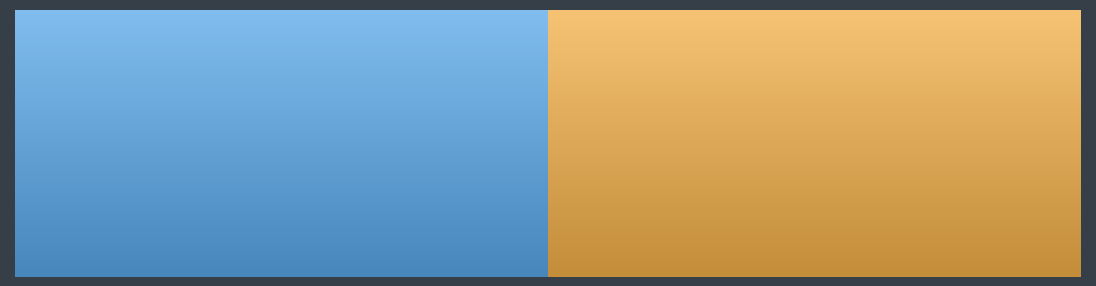
prolonged pauses

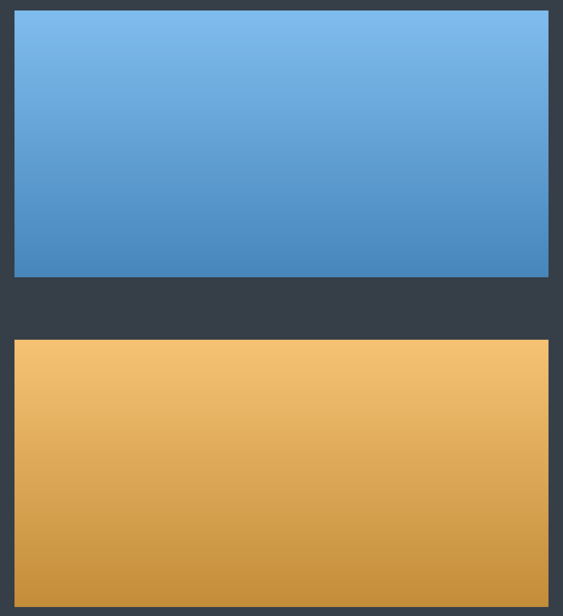
2. adding a lot of threads

multiple threads on the same core

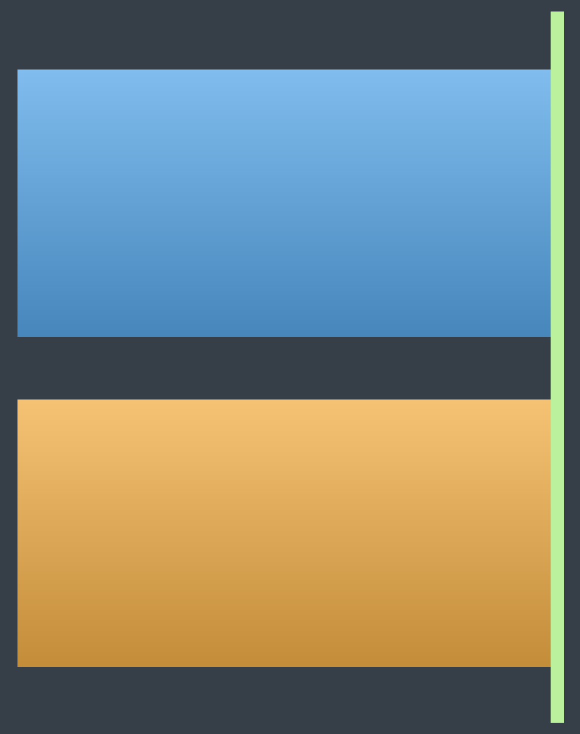
2 x 1 second

1 core

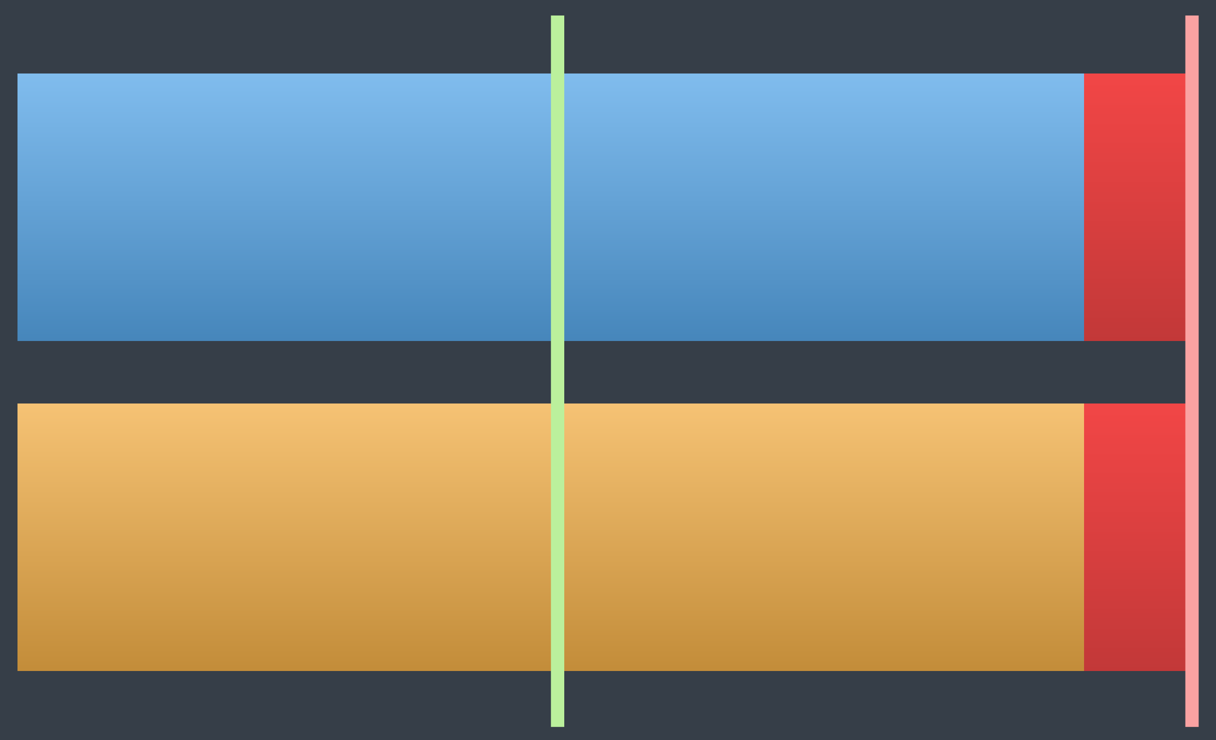




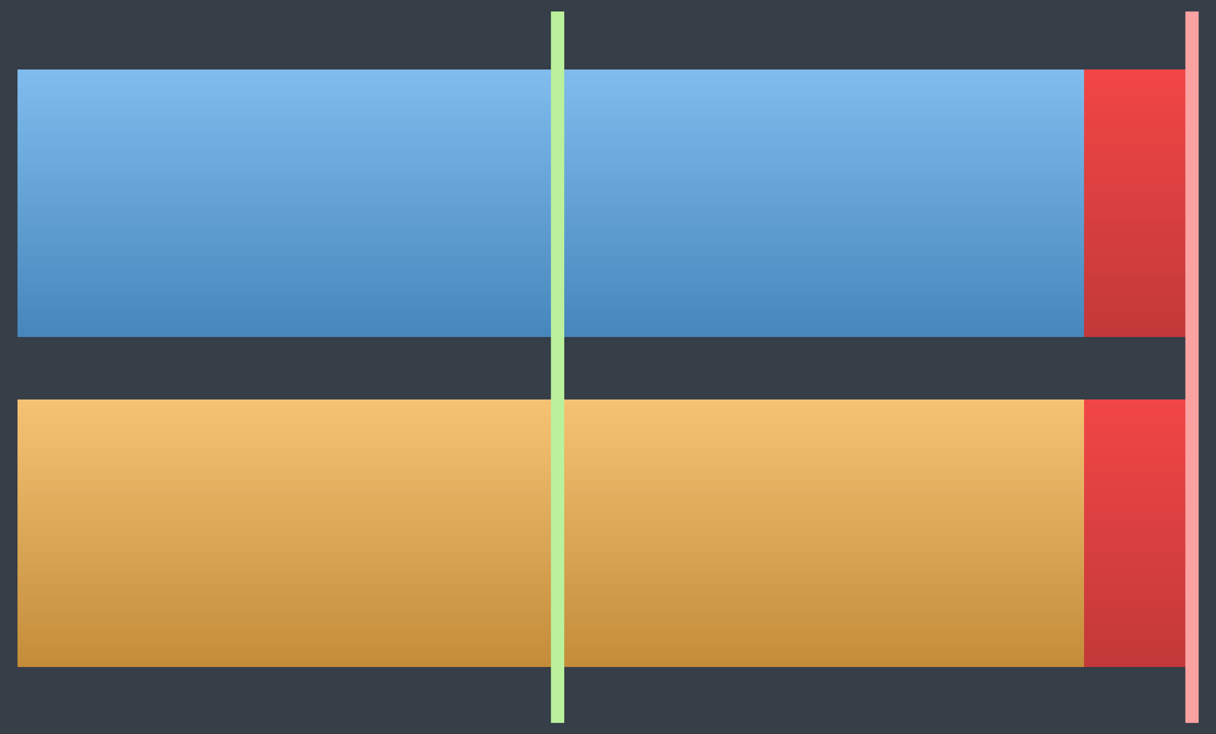
2 threads, 1 core



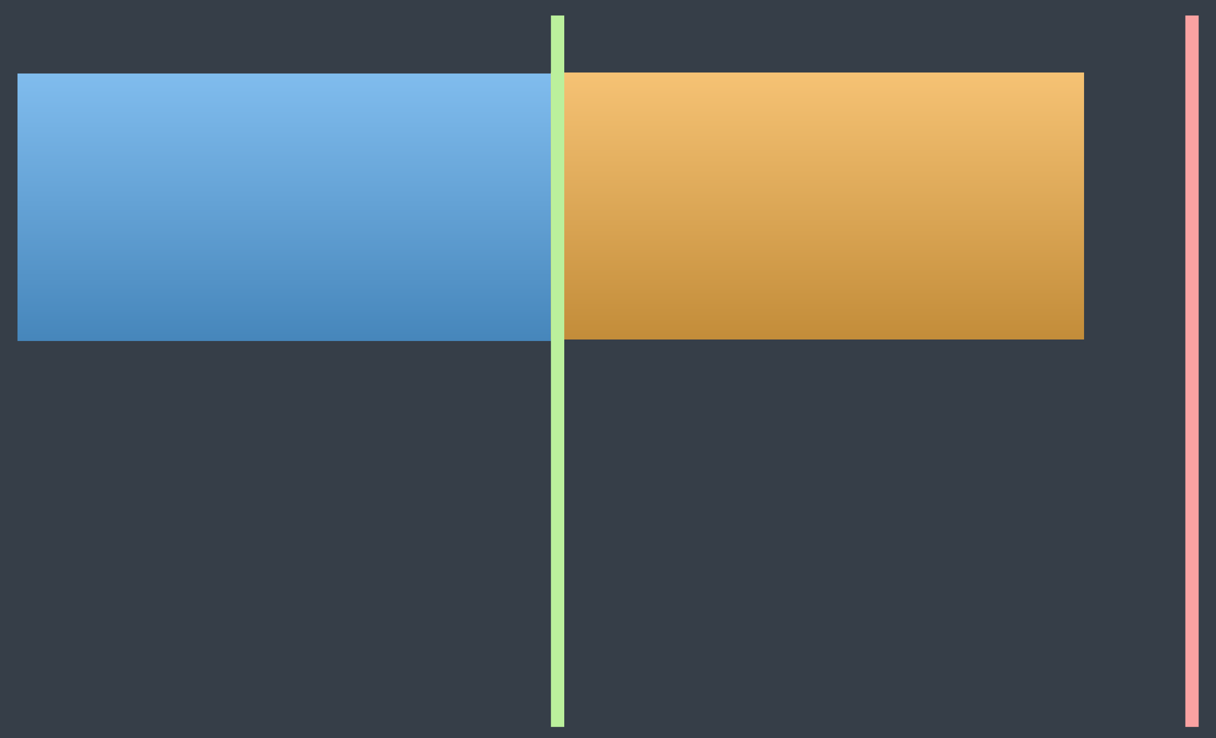
2 threads, 1 core



good ?



alternative



example 1

Happy
families
are
all
alike;
every
unhappy
family
is
unhappy
in
its
own
way.

he
vigorously
embraced
the
pillow
on
the
other
side
and
buried
his
face
in
it

Happy
families
are
all
alike;
every
unhappy
family
is
unhappy
in
its
own
way.

he
vigorously
embraced
the
pillow
on
the
other
side
and
buried
his
face
in
it

ready?

Happy
families
are
all
alike;
every
unhappy
family
is
unhappy
in
its
own
way.

he
vigorously
embraced
the
pillow
on
the
other
side
and
buried
his
face
in
it

results

individual texts	~3+3 s
interleaved texts	~10s

example 2

task = 1 step

serial execution of steps = walking

concurrent execution = walking with tied legs

threads = # cores

~ for CPU intensive tasks ~

3. composability

one cannot simply compose two programs

deadlocks, livelocks

performance problems

essential in sw. eng.

threads & locks

low level primitives
like **goto**

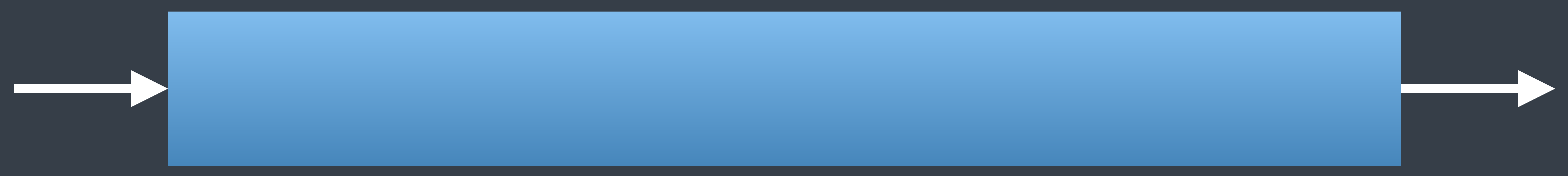
A General Solution

2

concurrency vs **parallelism**

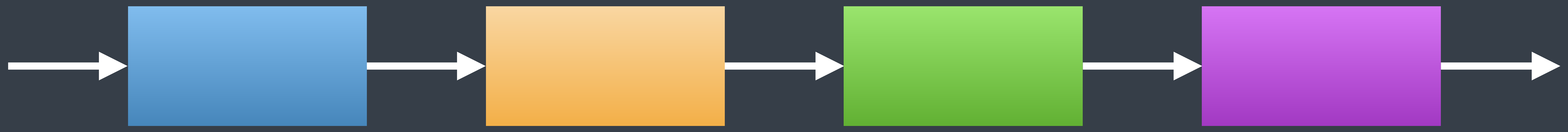
goal: design for expressing the concurrency

a single-threaded application

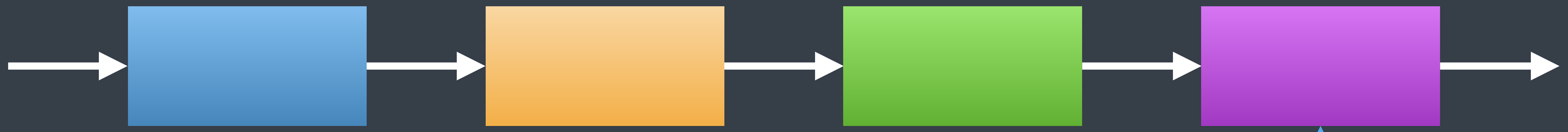


a single-threaded application

a single-threaded application



a single-threaded application



Task

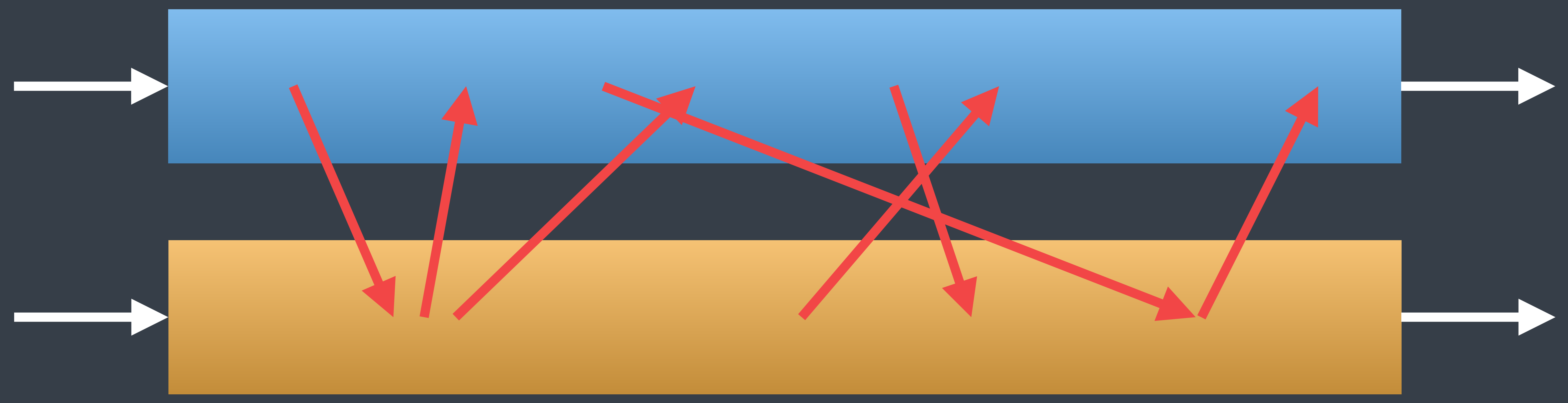
independent unit of work

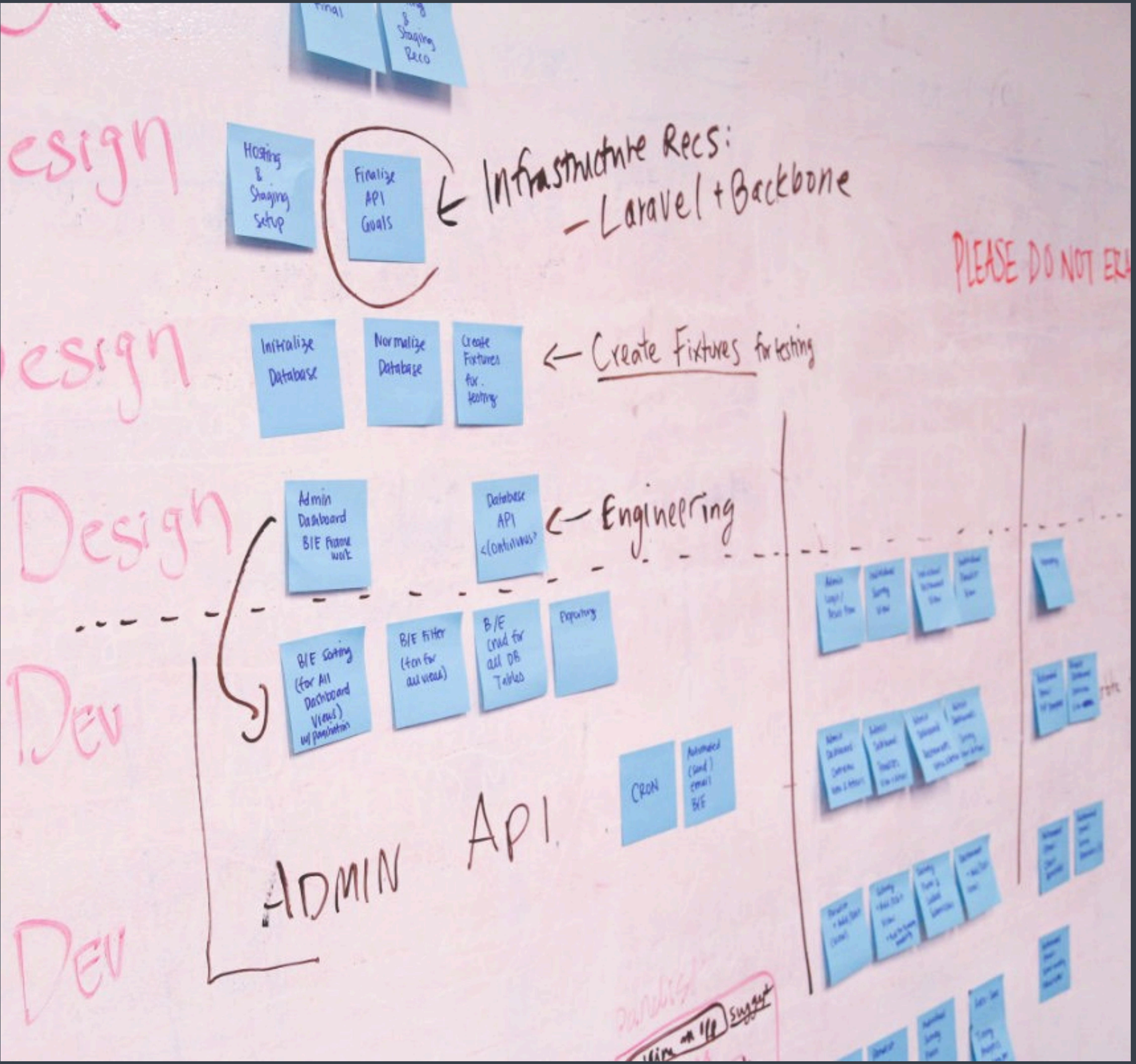
task

```
using Task = std::function< void() >;
```

```cpp
std::vector<Task> tasks;

int r1, r2, r3;
tasks.emplace_back([&]() { r1 = f1(); }
tasks.emplace_back([&]() { r2 = f2(r1); }
tasks.emplace_back([&]() { r3 = f3(r2); }
tasks.emplace_back([&]() { f4(r3); }

for (const auto& t: tasks)
    t();
```

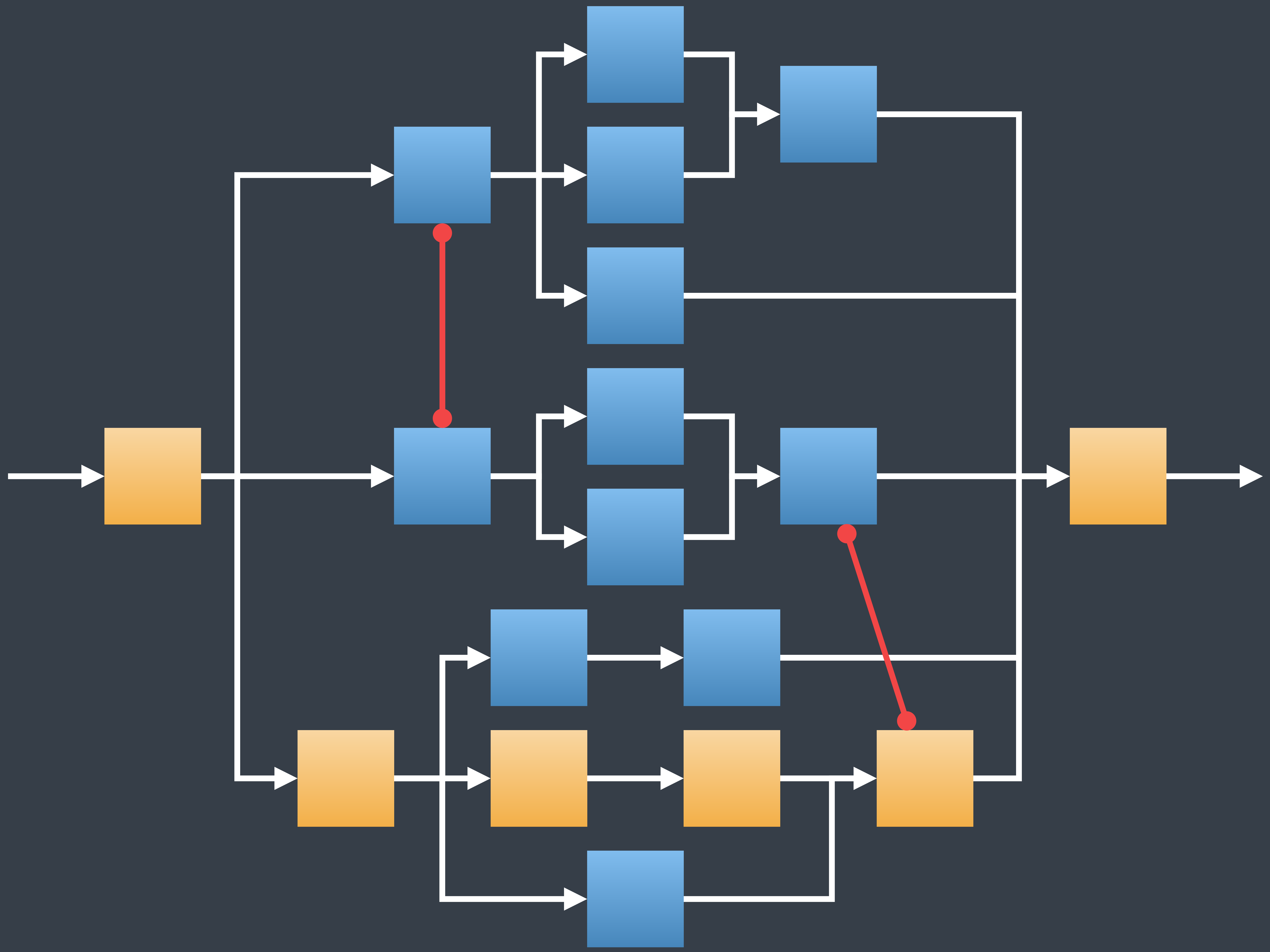
a multi-threaded application



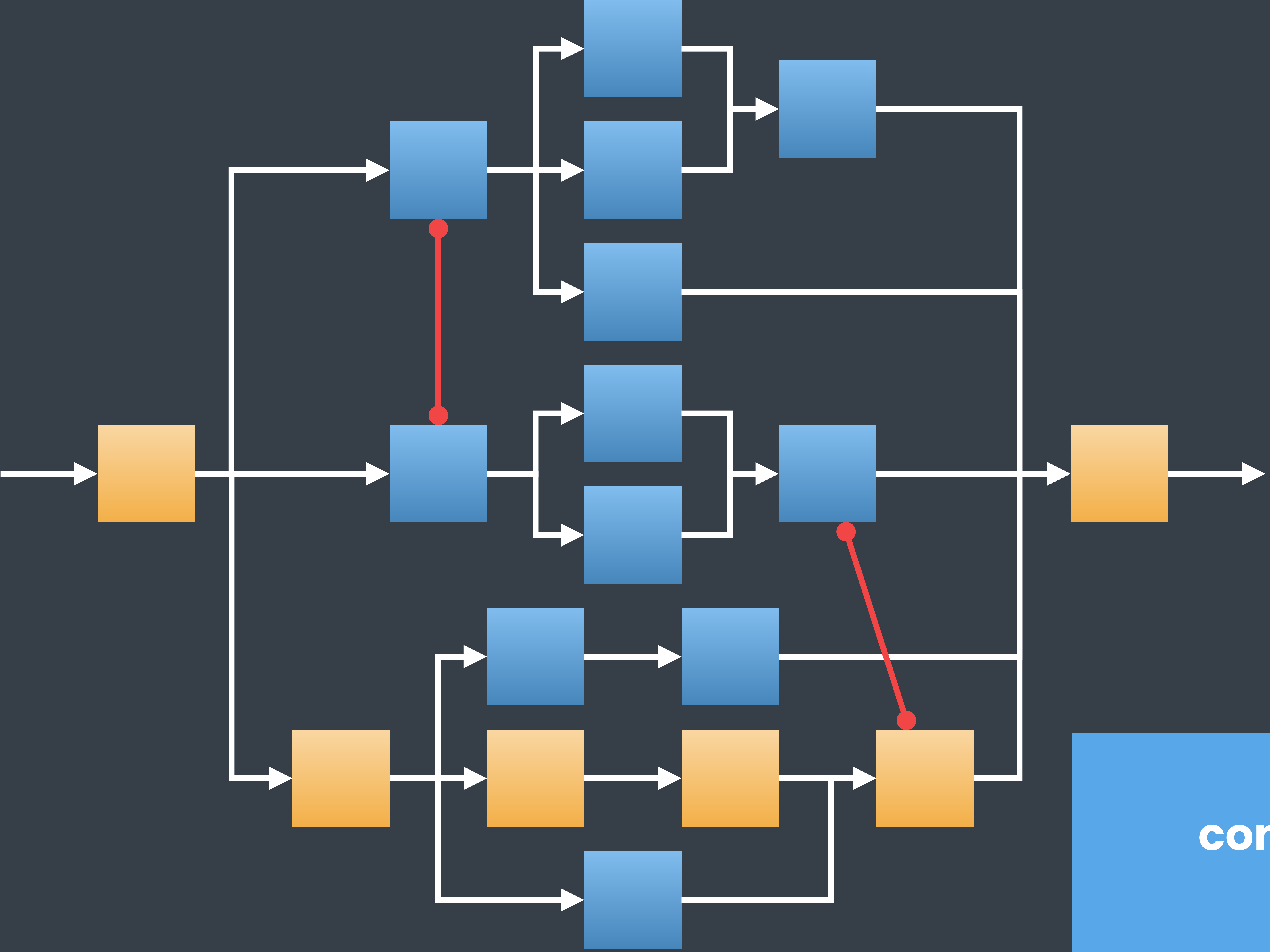


tasks

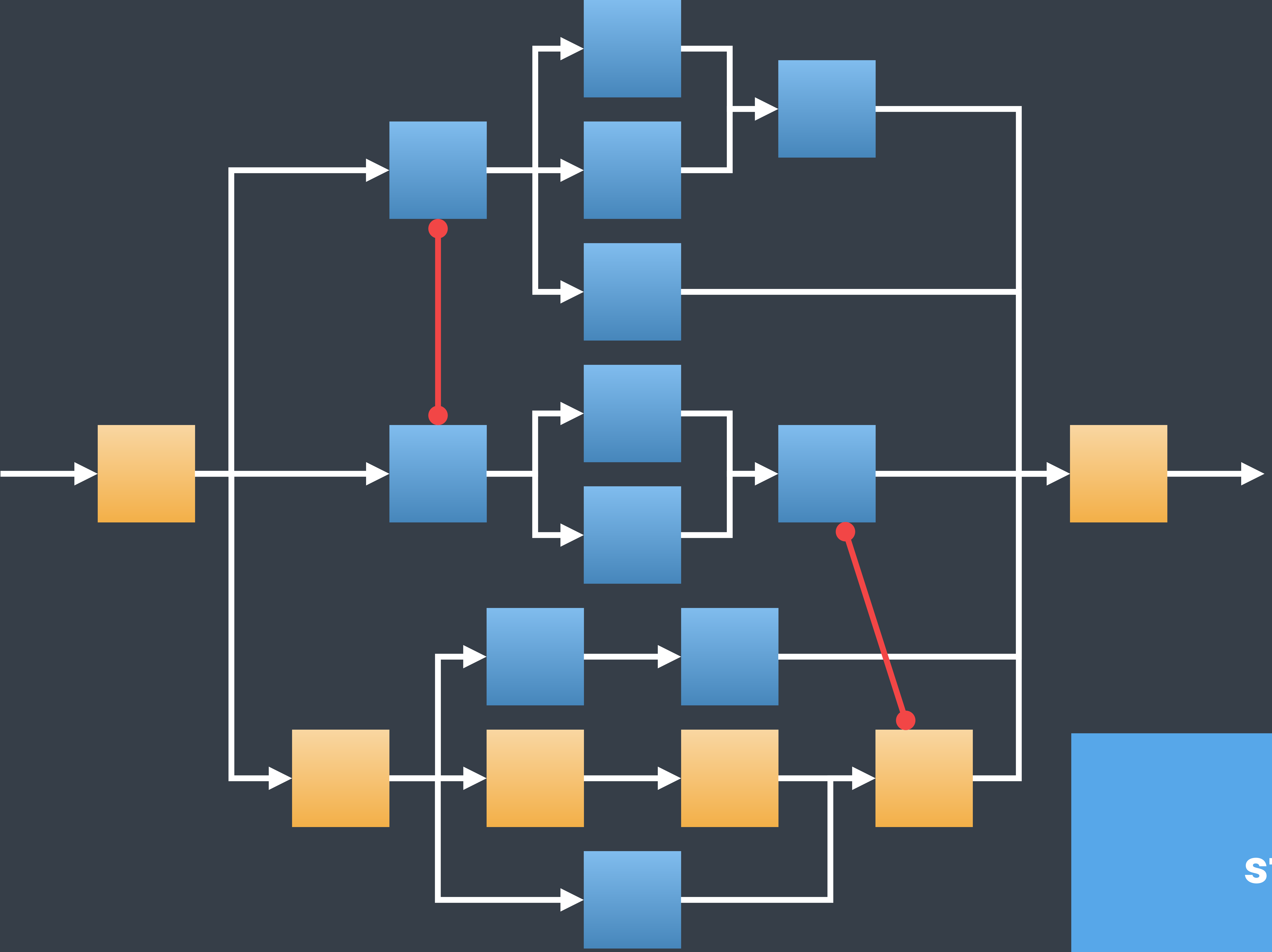
independent units of work



@LucT3o



constraints instead of locks



static vs dynamic

general algorithm

a task is **enqueued** when it can safely be executed

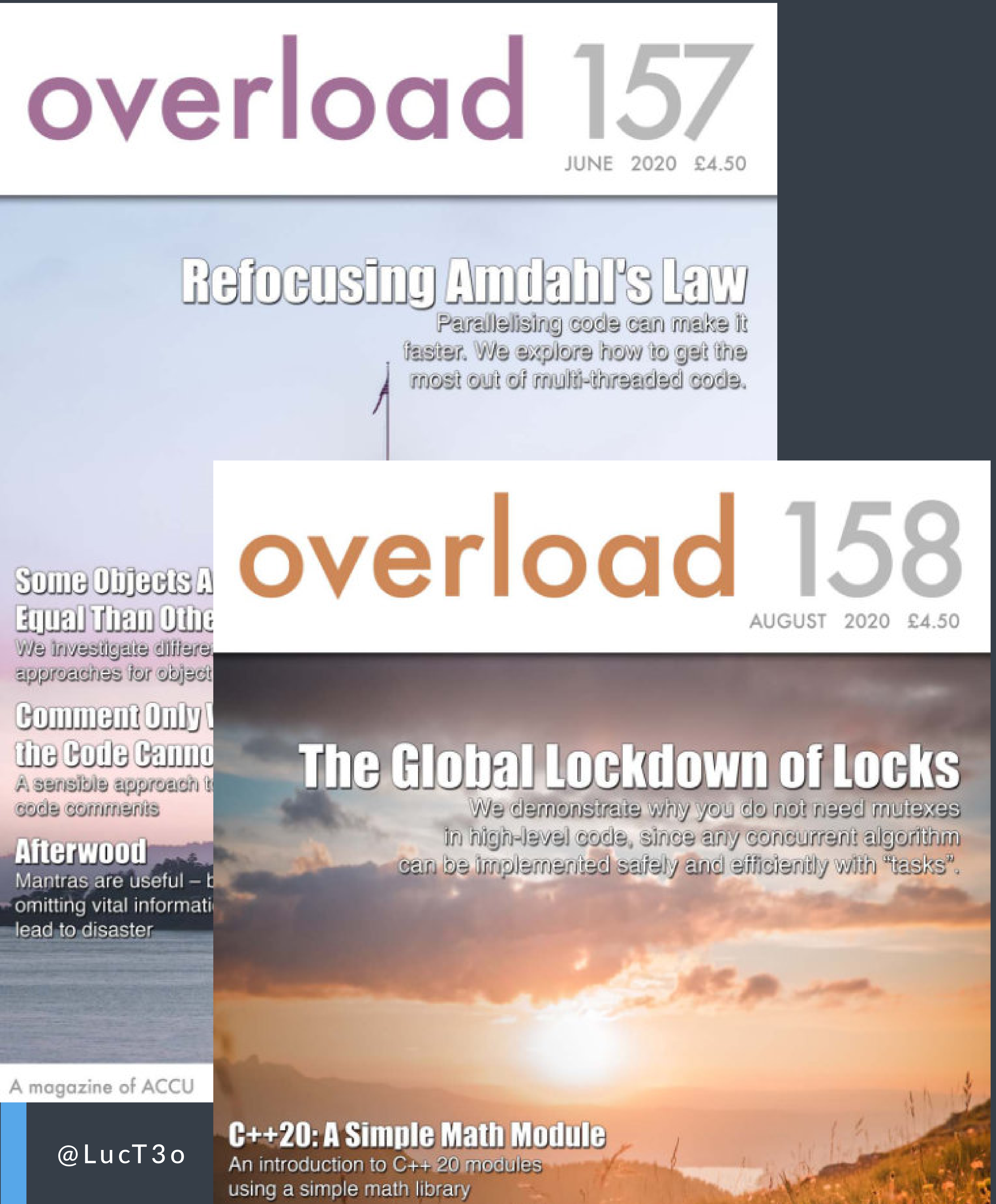
… after predecessors/constraints are done

after a task is done it checks if there are successors to enqueue

two key moments

- before the task is started (caller)
- after a task is finished (callee)

theoretical results



- all concurrent algorithms
- safety ensured
- no need for locks
- high efficiency for greedy algorithm
- high speedups

high performance

overhead of tasks management can be small

tasks are independent by design

$$S_p \geq \frac{N}{K + \frac{N - K}{P}}$$

high performance

overhead of tasks management can be small

tasks are independent by design

$$S_p \geq \frac{N}{K + \frac{N-K}{P}}$$

$$S_{1000} = 500.25$$

$$S_{10} = 9.91$$

$$N = 1000$$

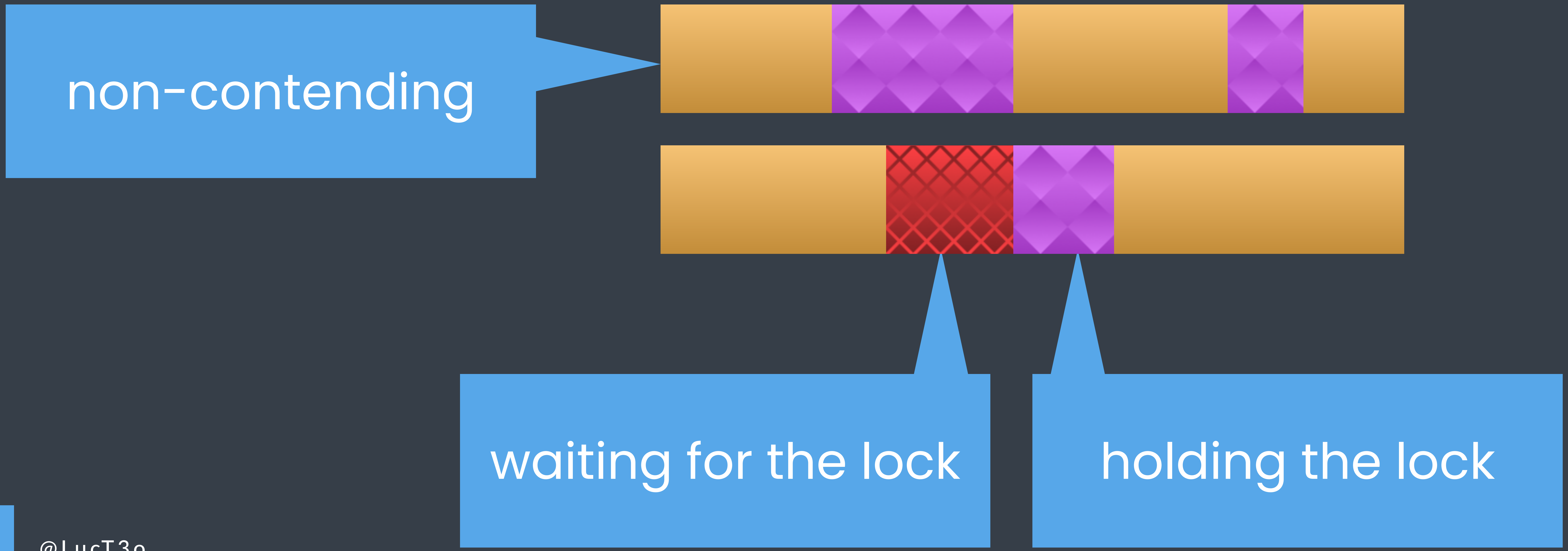
$$K = 1$$

3

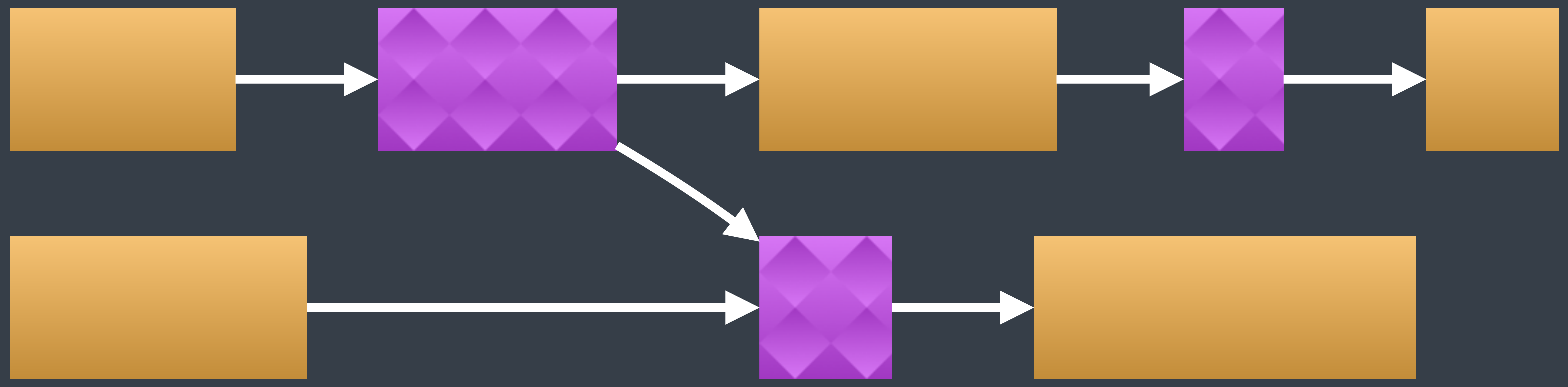
Coordination without Synchronization

1. mutexes

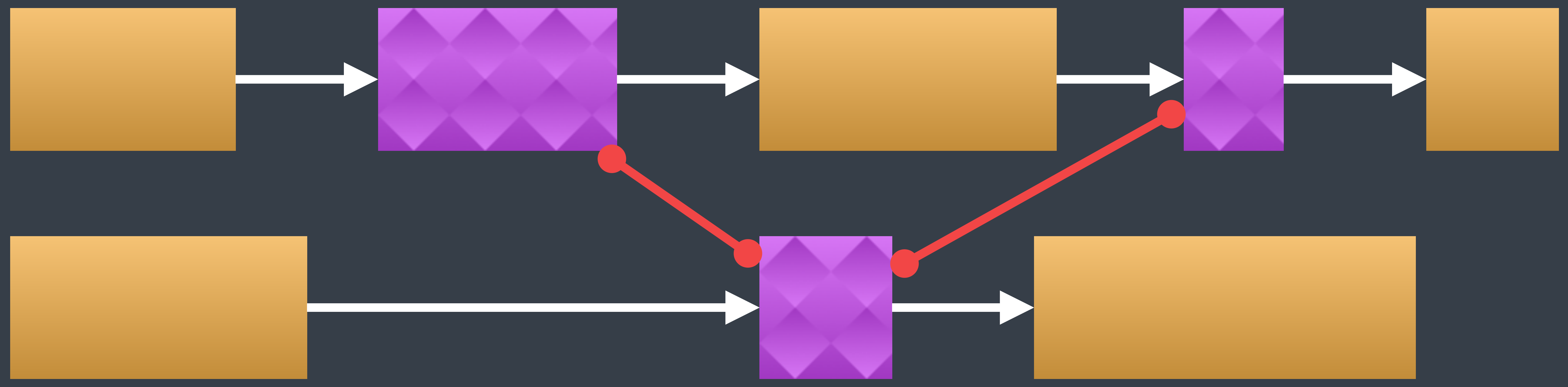
two threads with a mutex



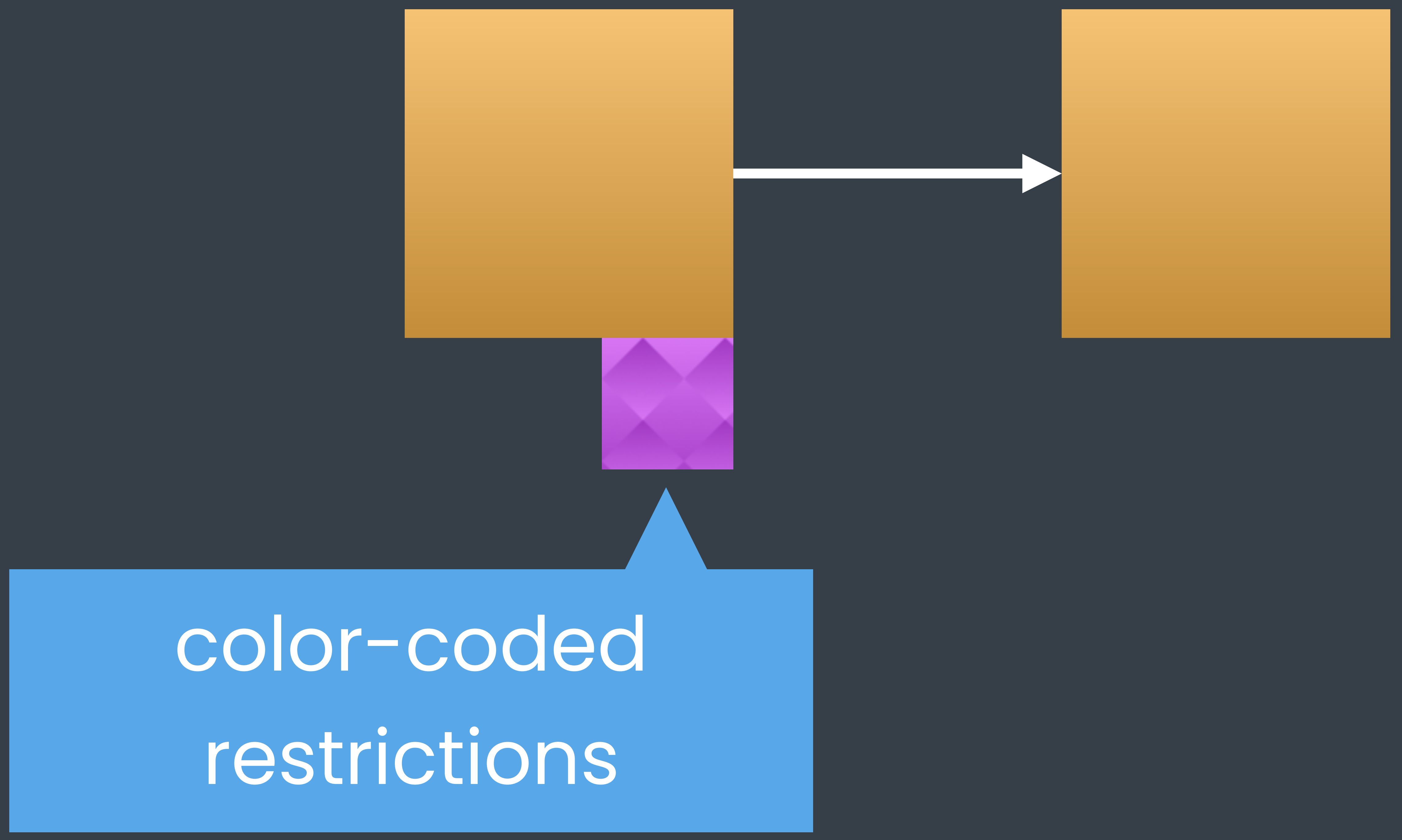
a possible solution



a possible solution (2)



a better notation

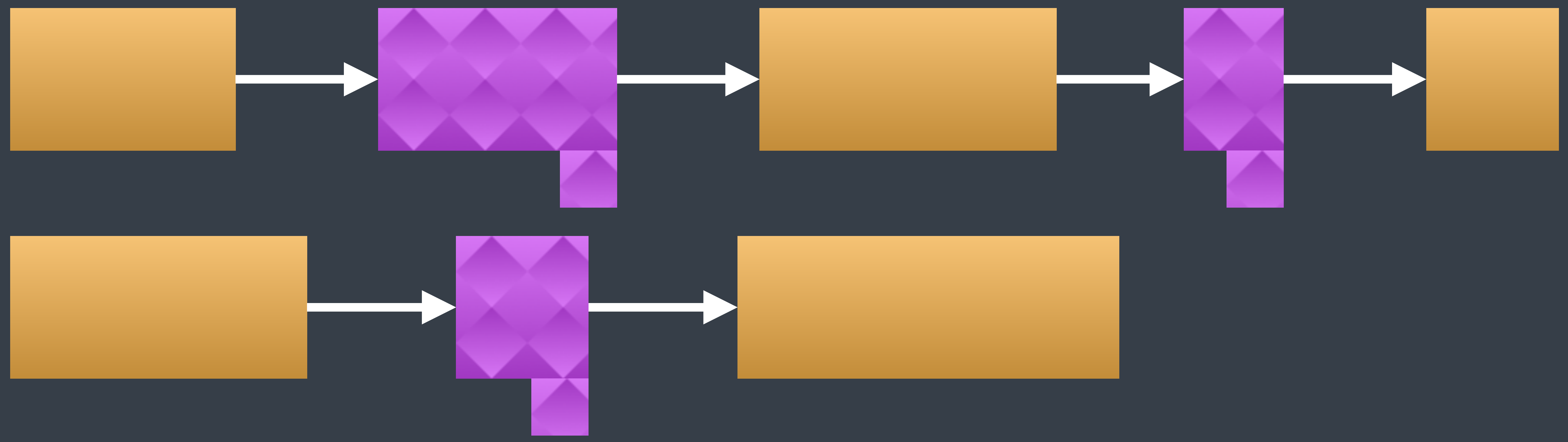


a task with restrictions

cannot run in parallel

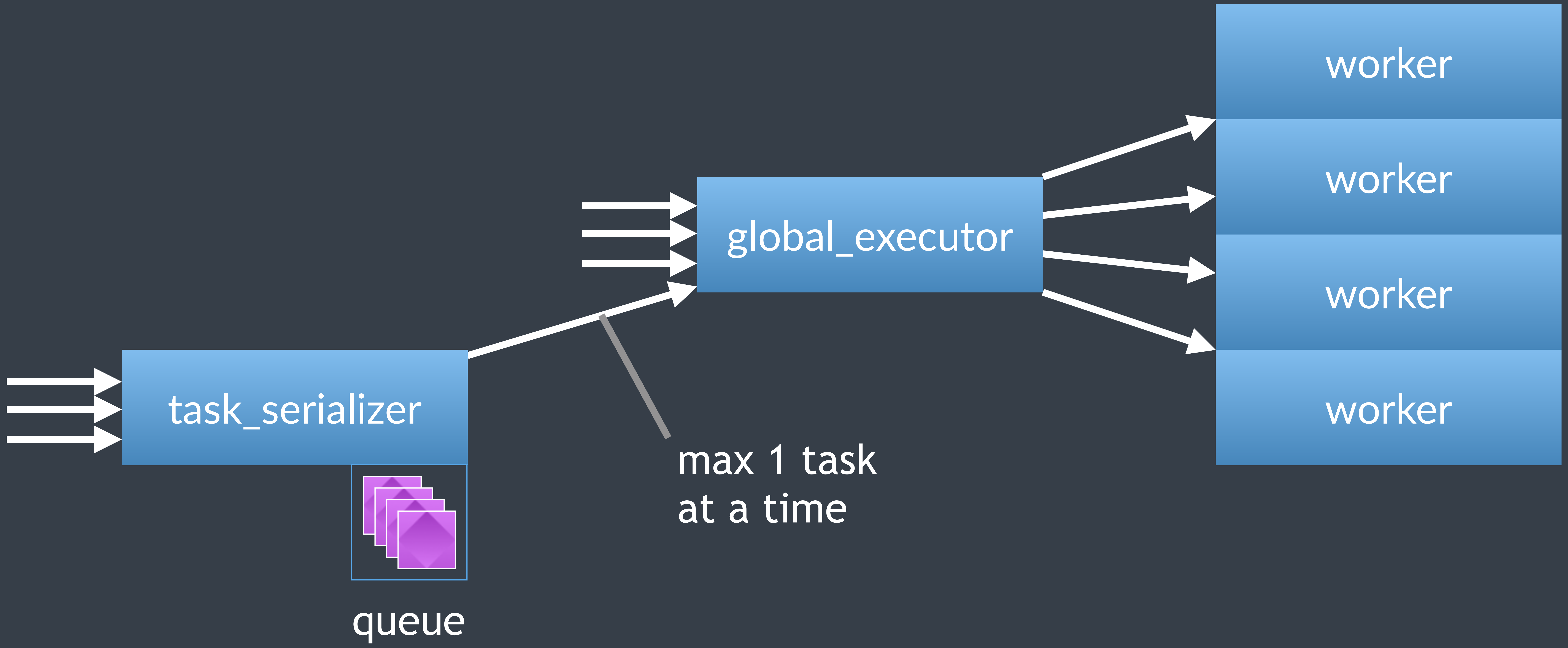
with a task colored like the restriction

dynamic representation



example

```
concore::serializer my_ser;
backup_engine my_backup_engine; // global resource

void trigger_backup(app_data data) {
    my_ser.execute([=]{ my_backup_engine.save(data); });
}
```


task_serializer
queue
max 1 task
at a time
global_executor
worker
worker
worker
worker

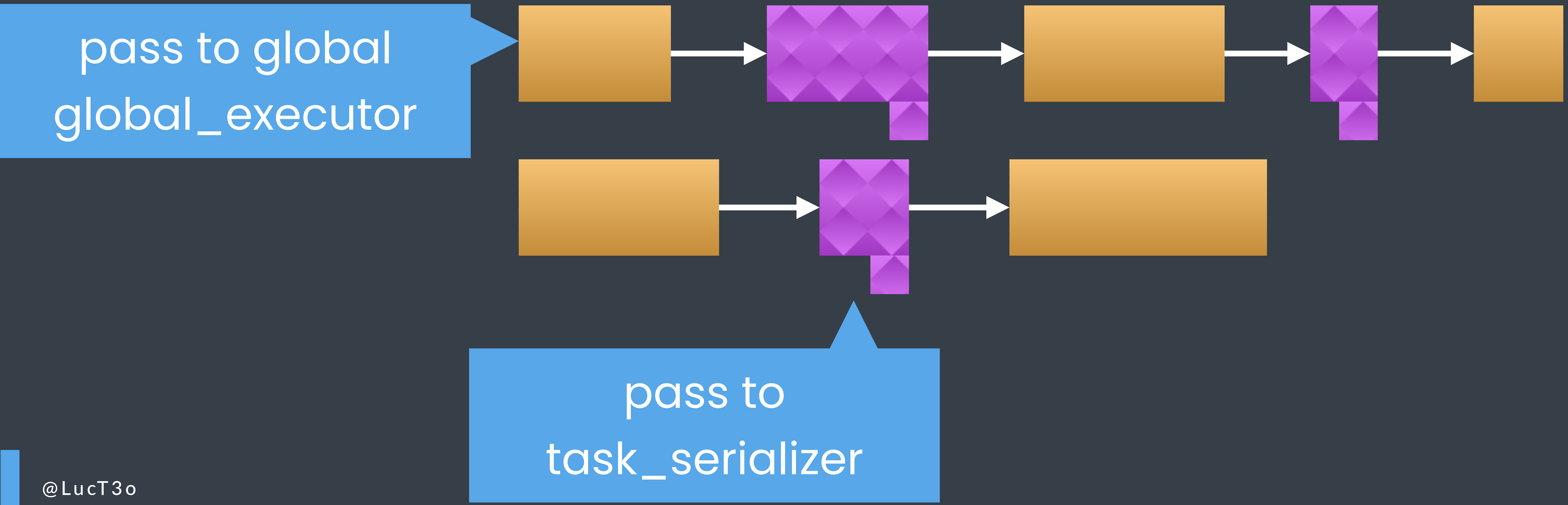
implementation details

keep a queue of tasks to be enqueued

keep track if we are executing a task

when finishing executing a task, enqueue the next task

dynamic representation



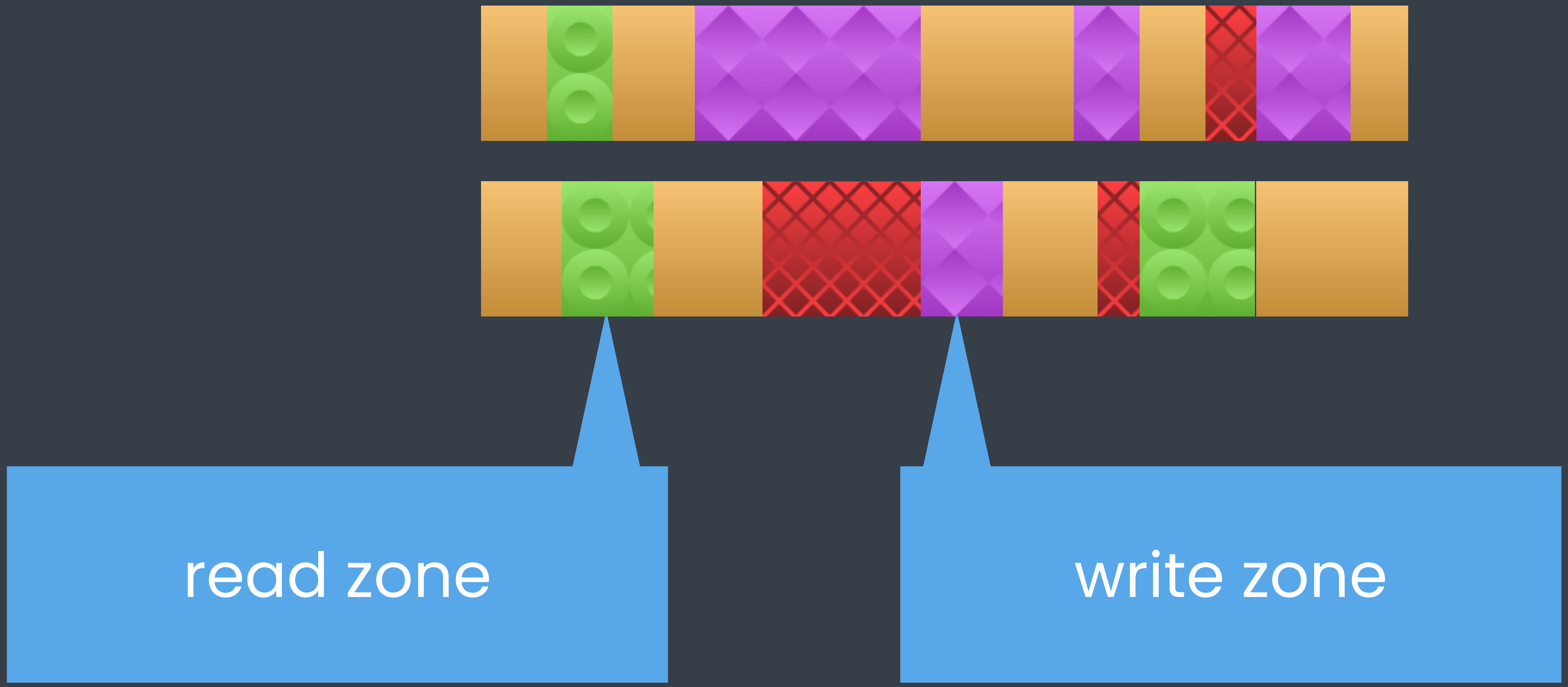
task executors

global_executor — a task is executed as soon as a worker is free

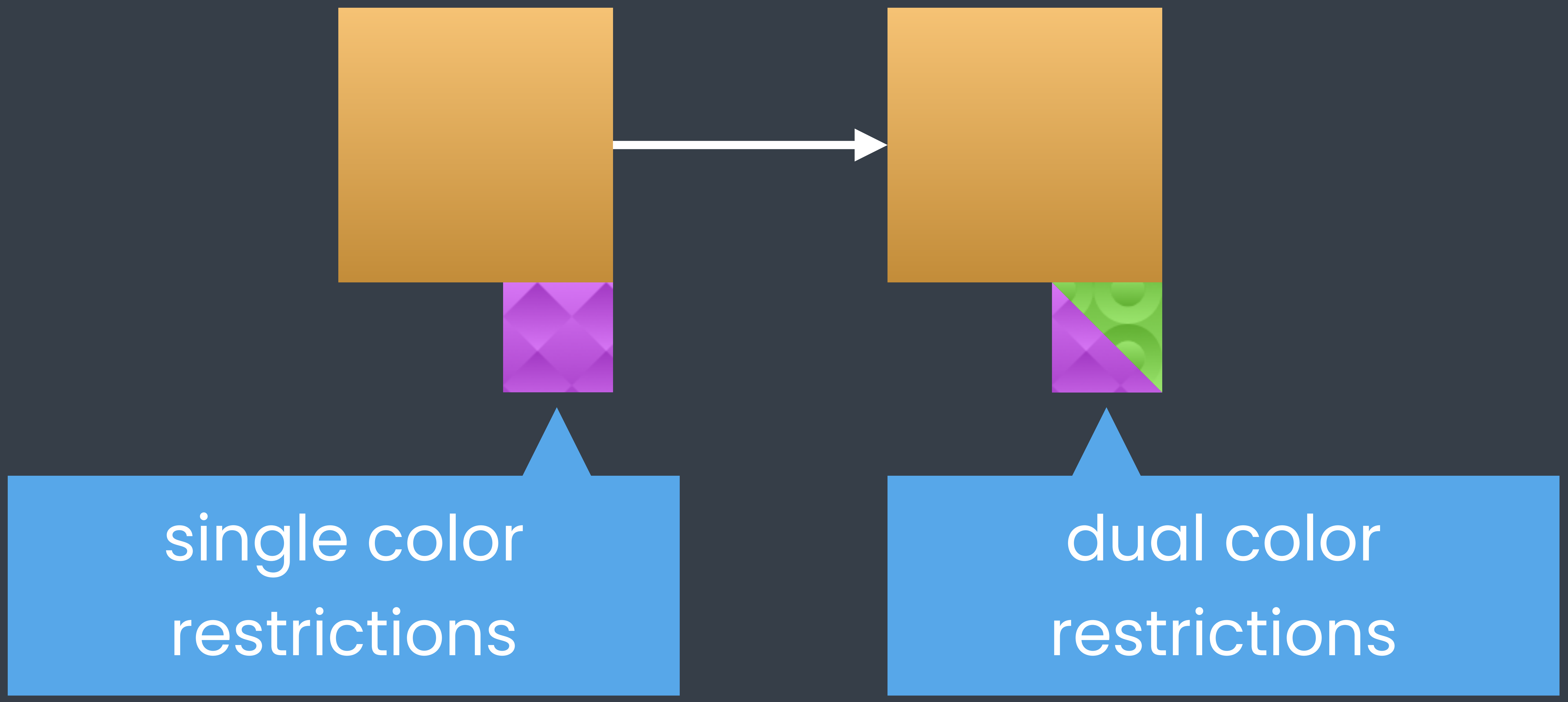
task_serializer — execute at most one task at a given time

2. read-write mutexes

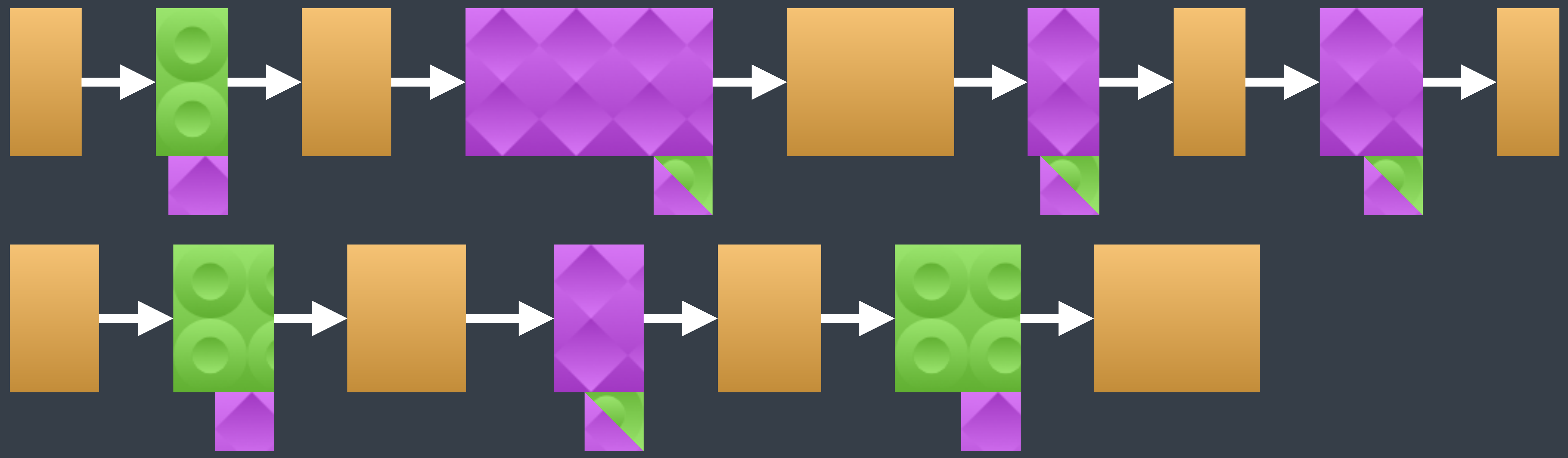
read-write mutex



same notation

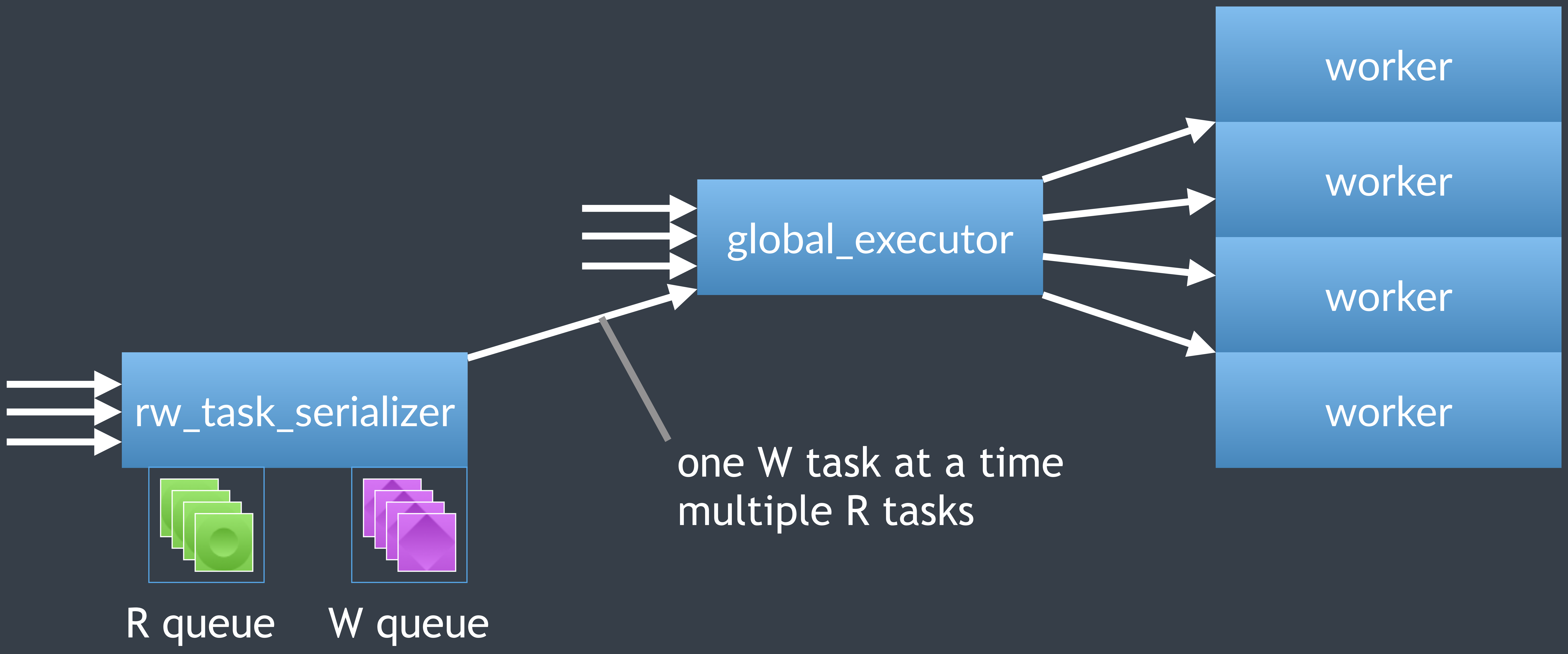


problem representation



example

```
concore::rw_serializer my_ser;
backup_engine my_backup_engine; // global resource

void get_latest_backup_info(std::function<void (backup_info)> f) {
    my_ser.reader().execute([f] {
        // query backup data
        auto data = std::move(my_backup_engine.get_info());
        // call the callback
        f(std::move(data));
    });
}
```


rw_task_serializer
R queue
W queue
global_executor
one W task at a time
multiple R tasks
worker
worker
worker
worker

task executors

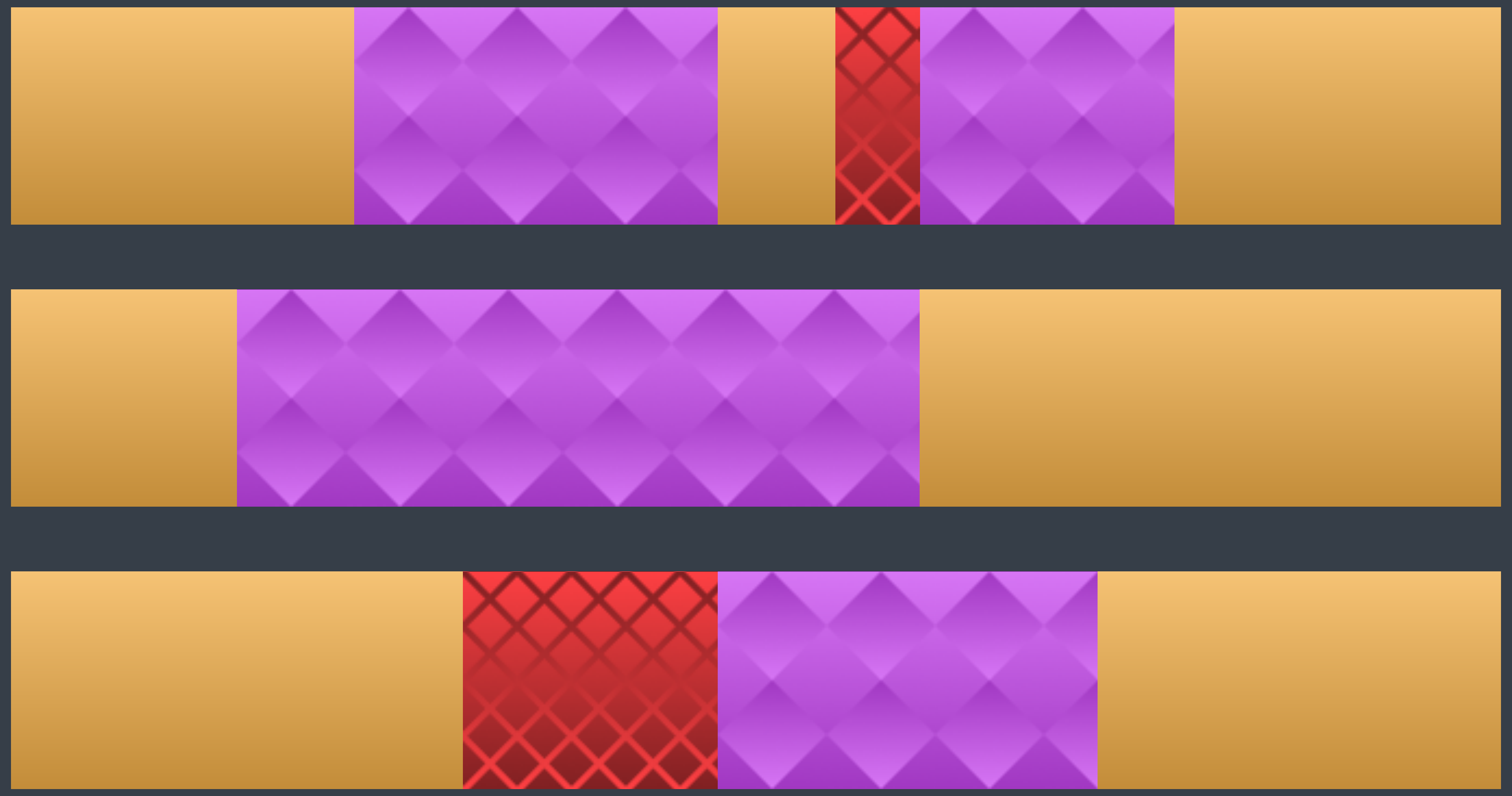
global_executor — a task is executed as soon as a worker is free

task_serializer — execute at most one task at a given time

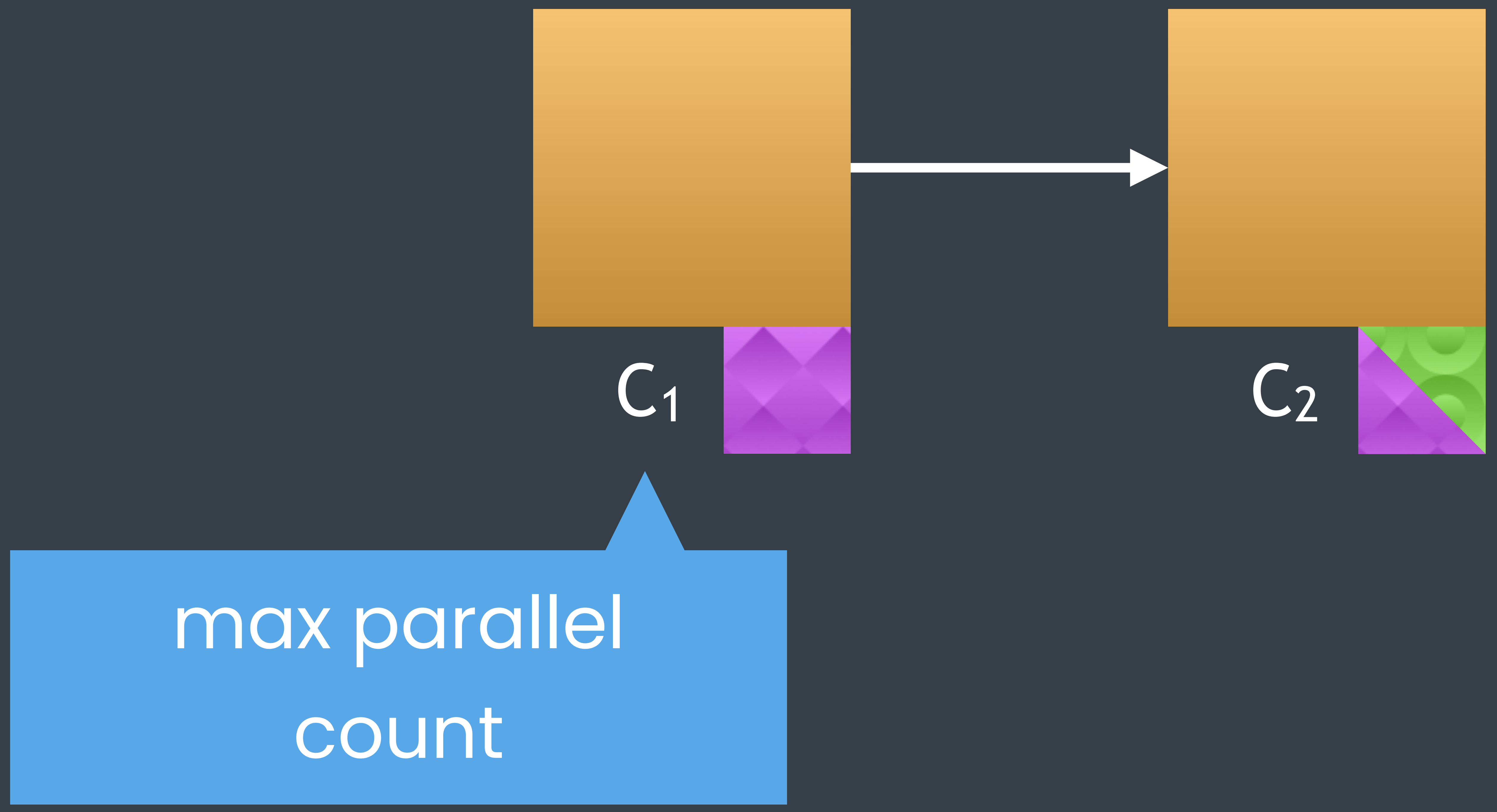
rw_task_serializer — restrictions between R and W tasks

3. semaphores

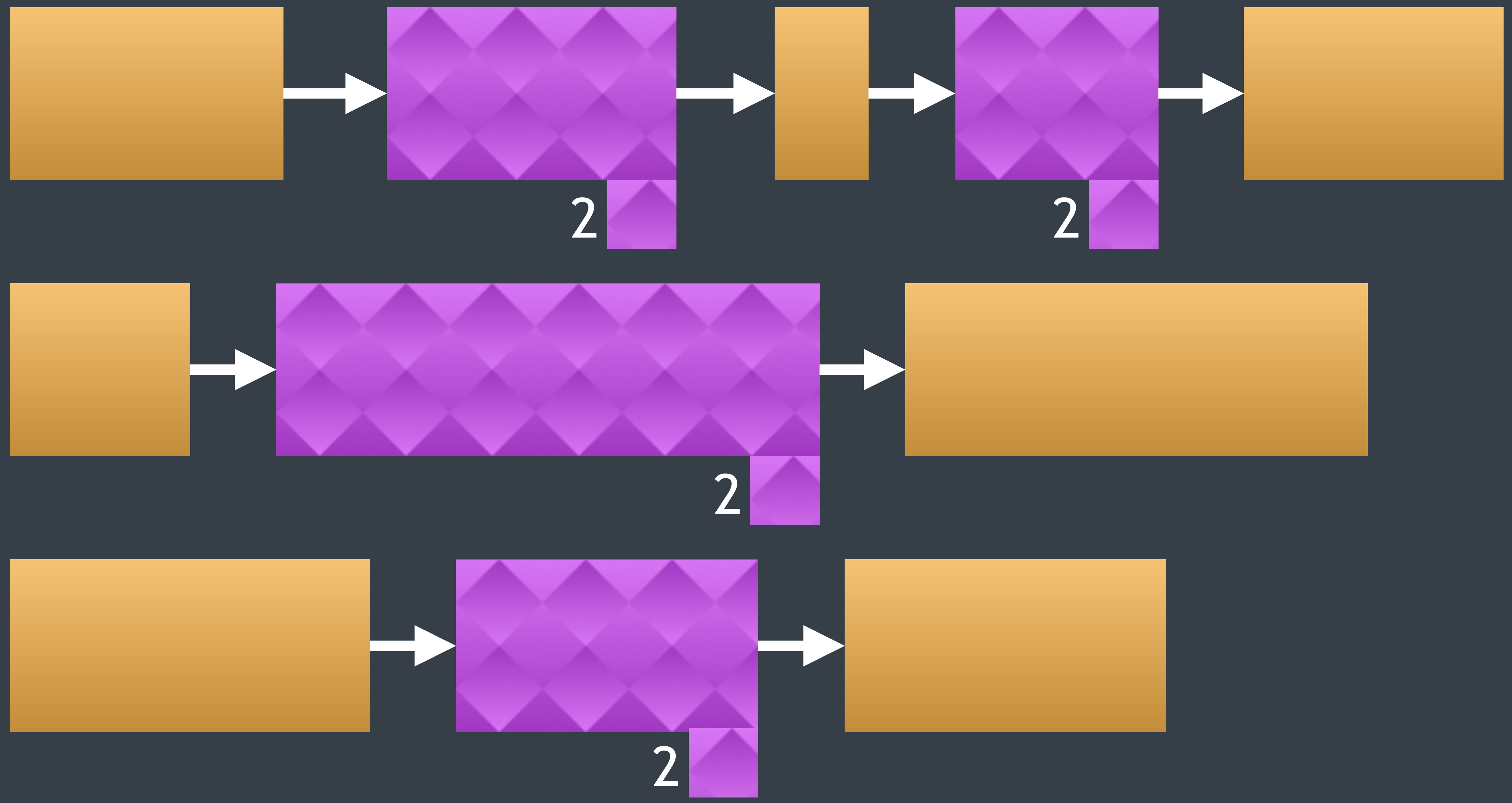
semaphore, count 2



improving notation

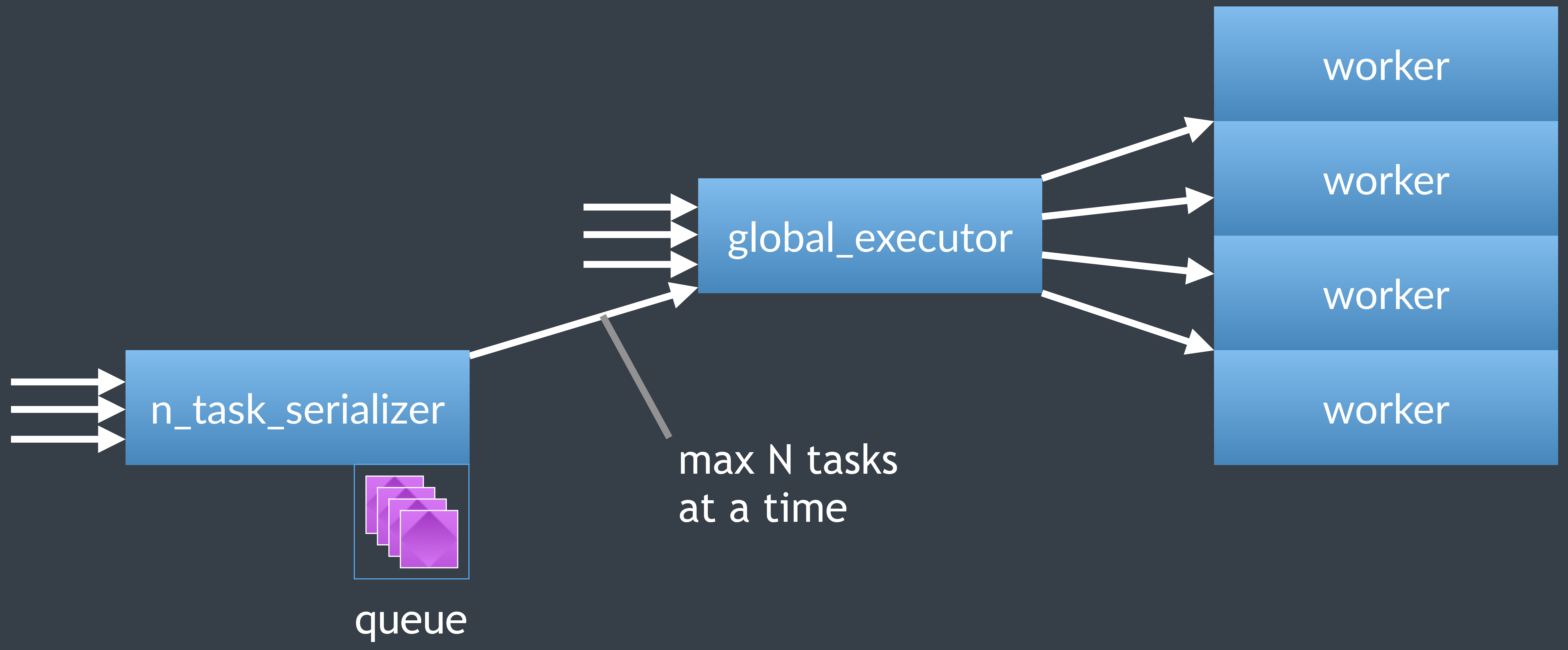


problem representation



example

```
concore::n_serializer my_ser{10};
concore::concurrent_queue<backup_engine> my_backup_engines; // 10

void trigger_backup(app_data data) {
    my_ser.execute([=]{
        // acquire a free backup engine
        backup_engine engine;
        bool res = my_backup_engines.try_pop(engine);
        assert(res);
        // do the backup
        engine.save(data); // assume no exceptions
        // release the backup engine to the system
        my_backup_engines.push(std::move(engine));
    });
}
```


worker
worker
worker
worker
global_executor
n_task_serializer
max N tasks
at a time
queue

task executors

global_executor — a task is executed as soon as a worker is free

task_serializer — execute at most one task at a given time

rw_task_serializer — restrictions between R and W tasks

n_task_serializer — execute at most N tasks at a given time

~ results so far ~

systematic way

raising the abstraction level

NO LOCKS

we have a systematic way of avoiding locks
no blocking needed

maximizing throughput

make sure you have enough tasks in the system

just to be **clear**

Locks	Serializers
can block threads	do not block
reduce throughput	throughput is ok if enough tasks

4

Composability and Decomposability

1. task systems are composable

inner constraints are kept
may require some extra constraints

no different than regular software

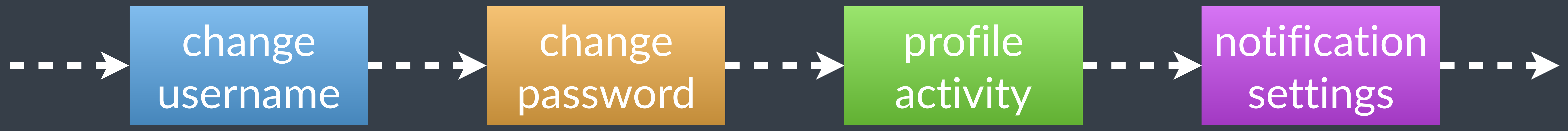
2. decomposing tasks

e.g., decomposing serializer tasks

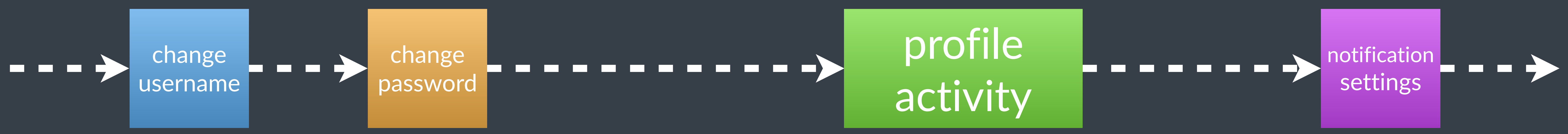
serialize all “user account” actions

some tasks can be decomposed

user account serializer



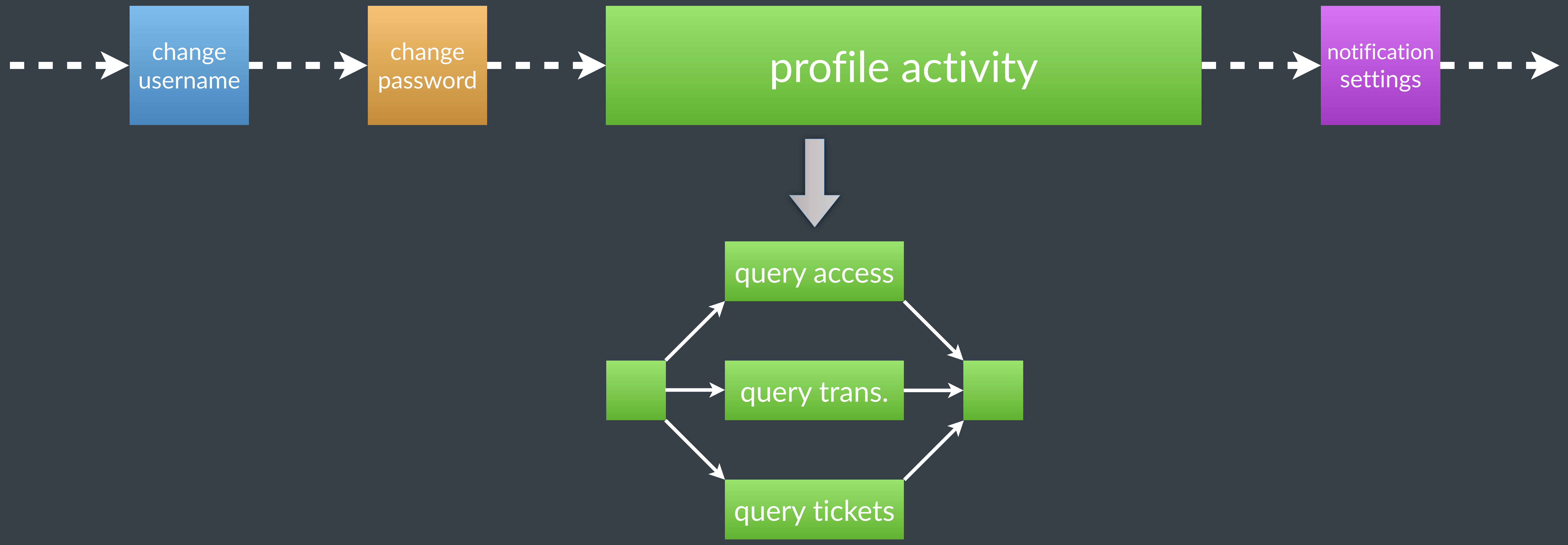
user account serializer



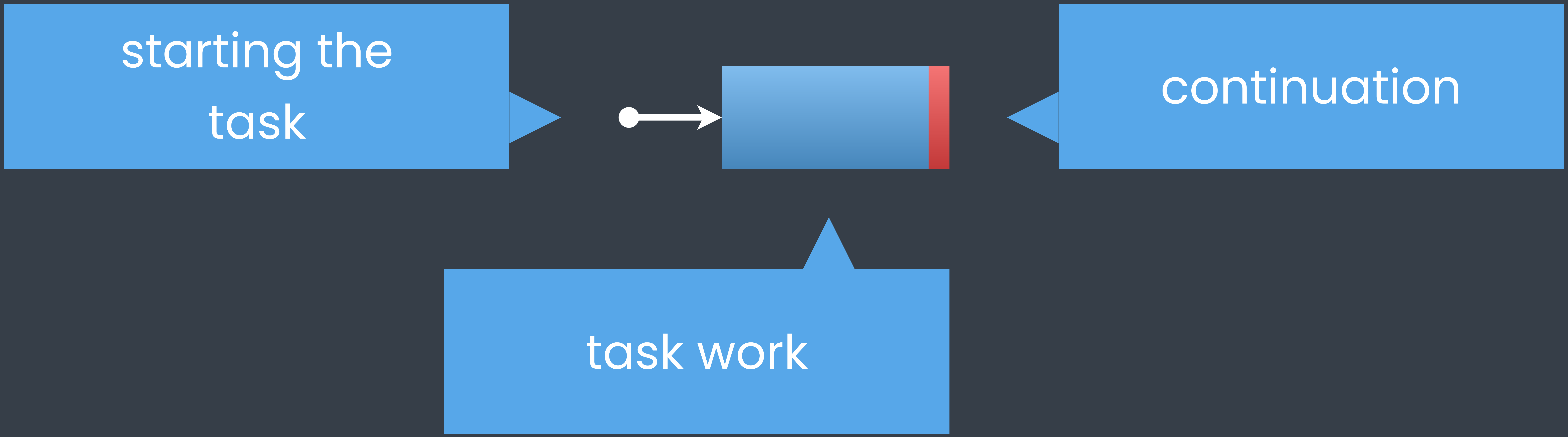
user account serializer

user account serializer

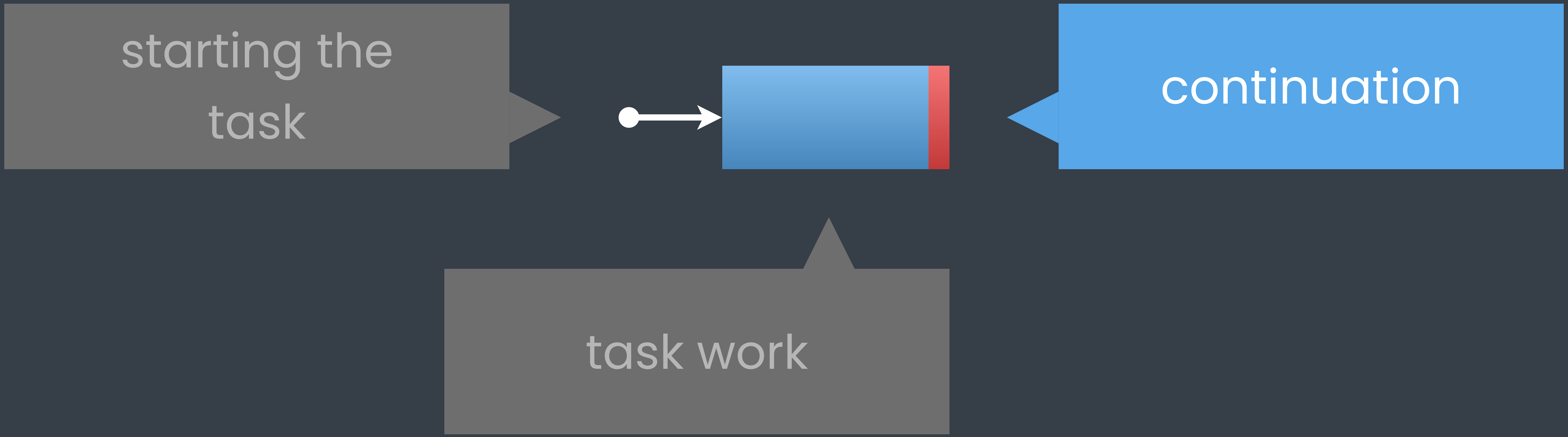
user account serializer



task continuations



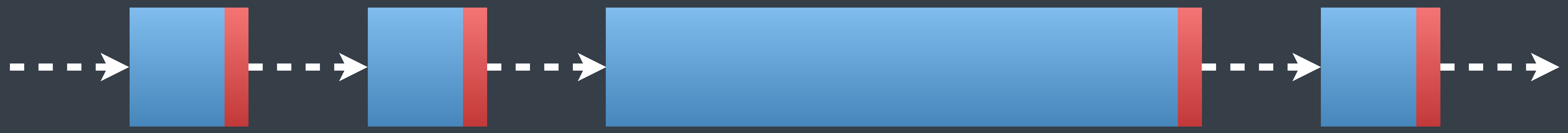
task continuations



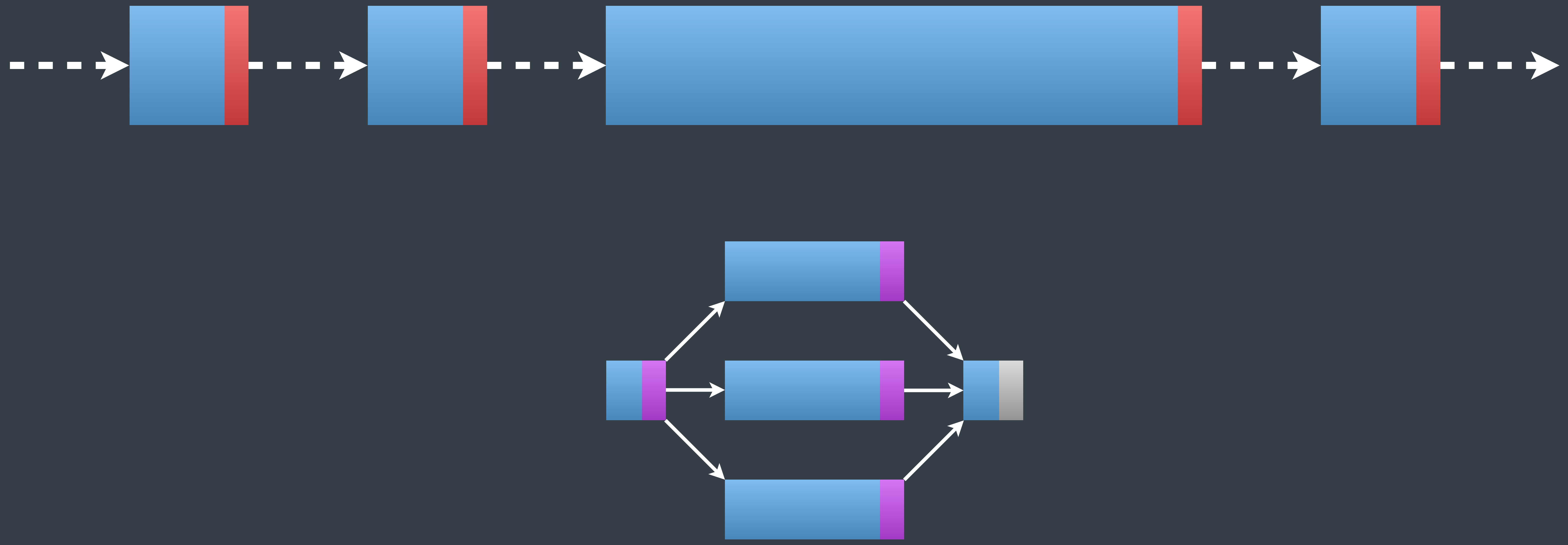
new task

```
using Task = std::pair<
    std::function< void() >, // work
    std::function< void() >  // continuation
>;
```

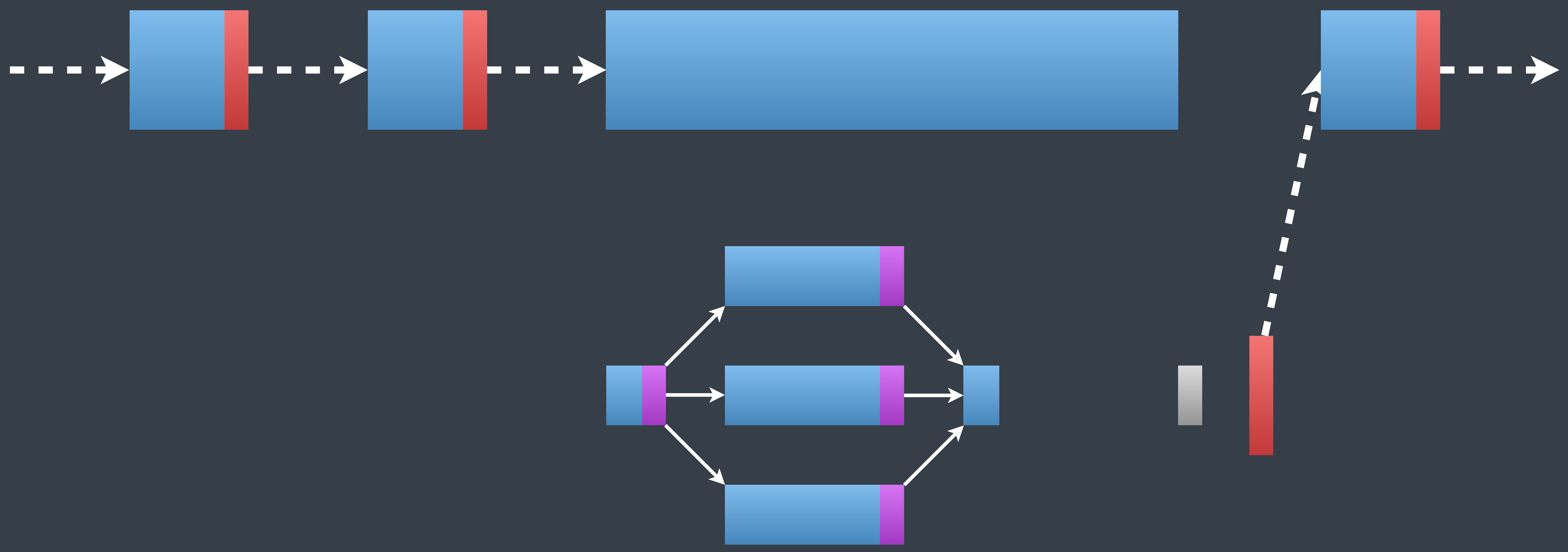
serializer



serializer

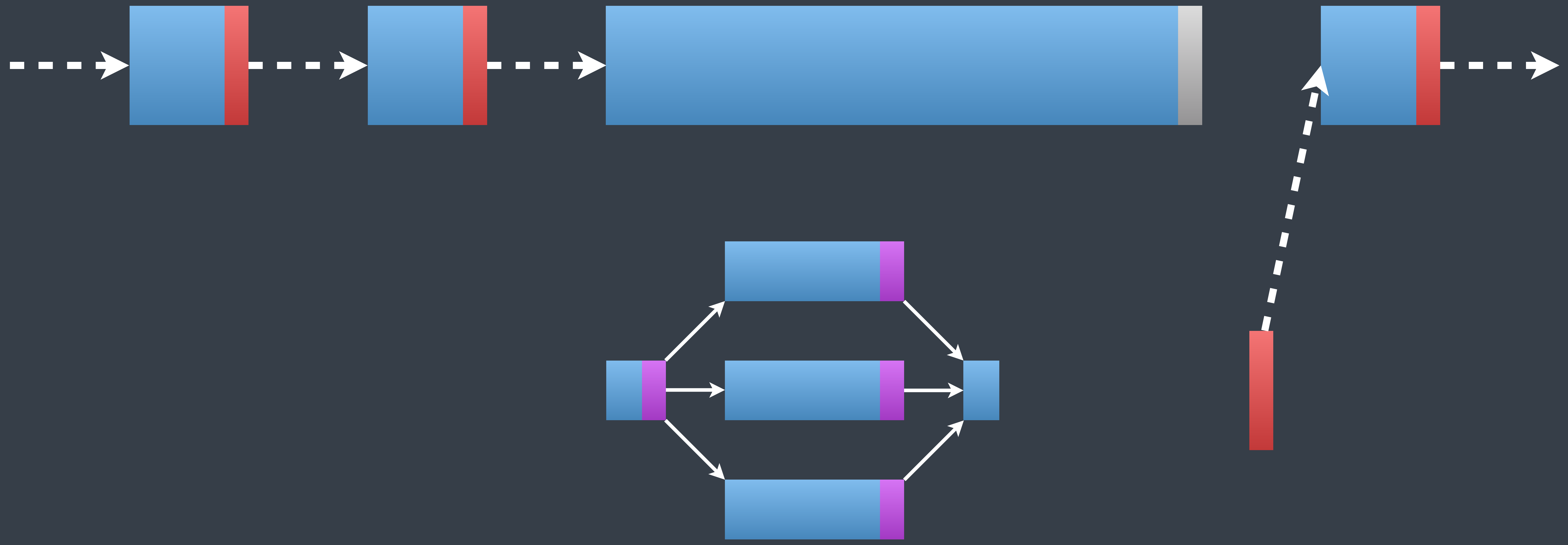


serializer



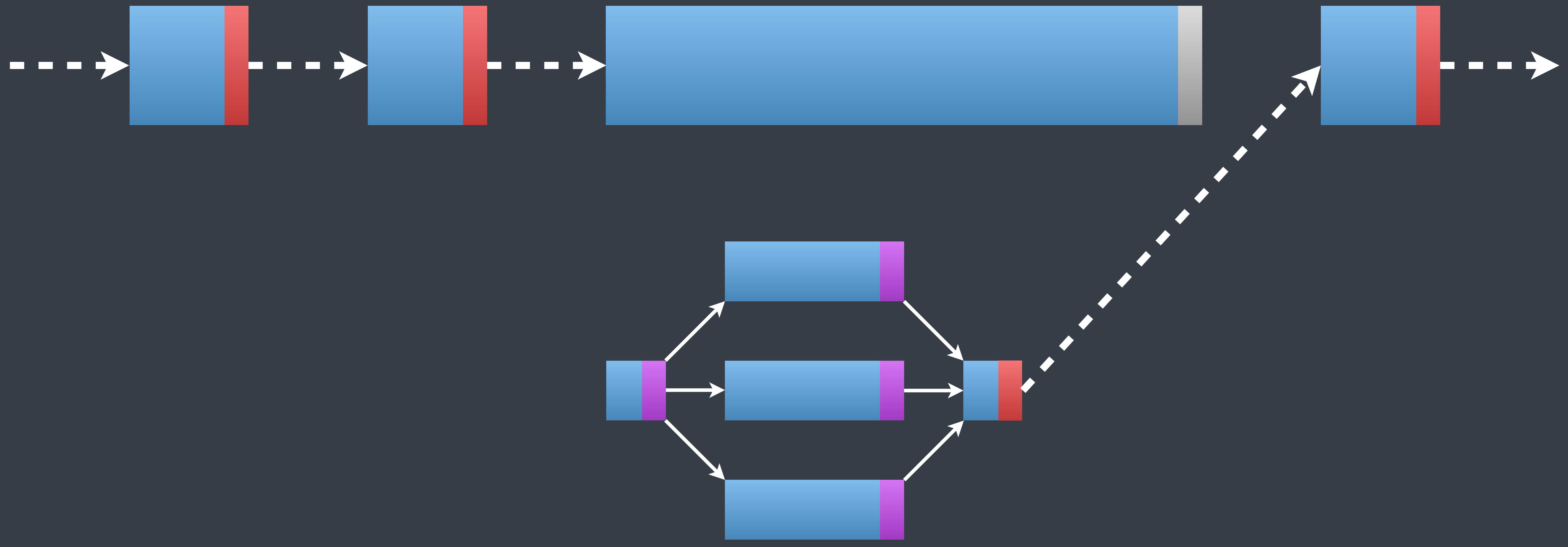
1. exchange continuations

serializer



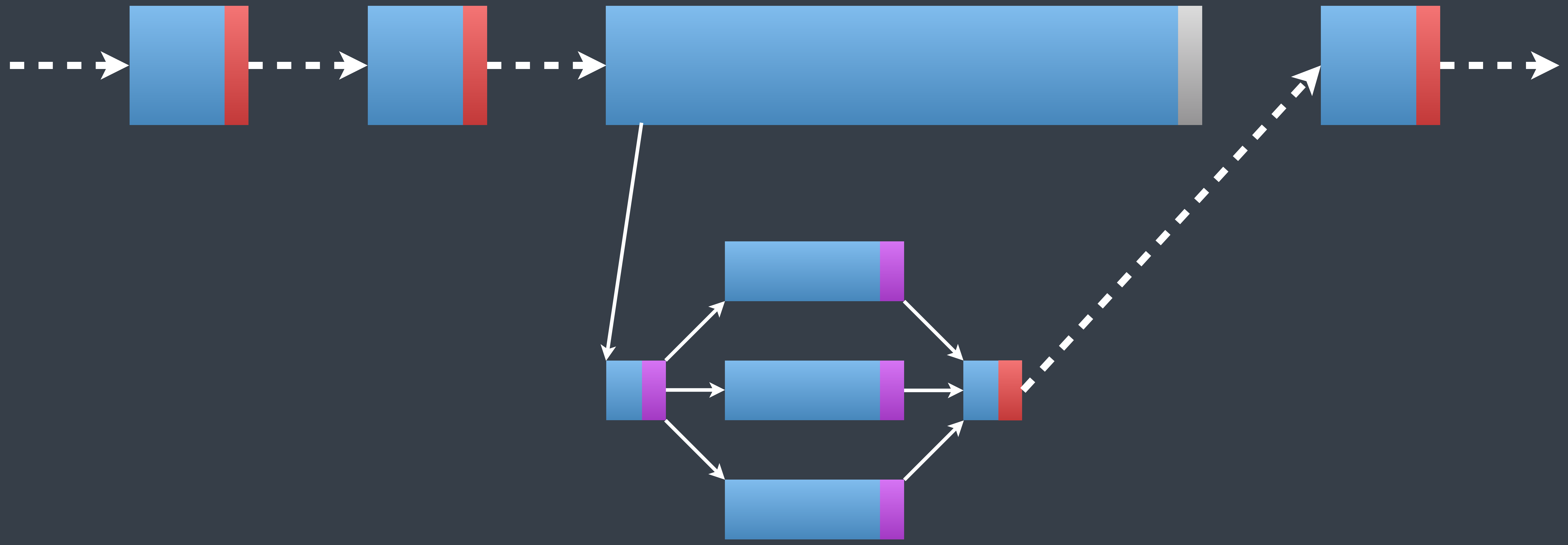
1. exchange continuations

serializer



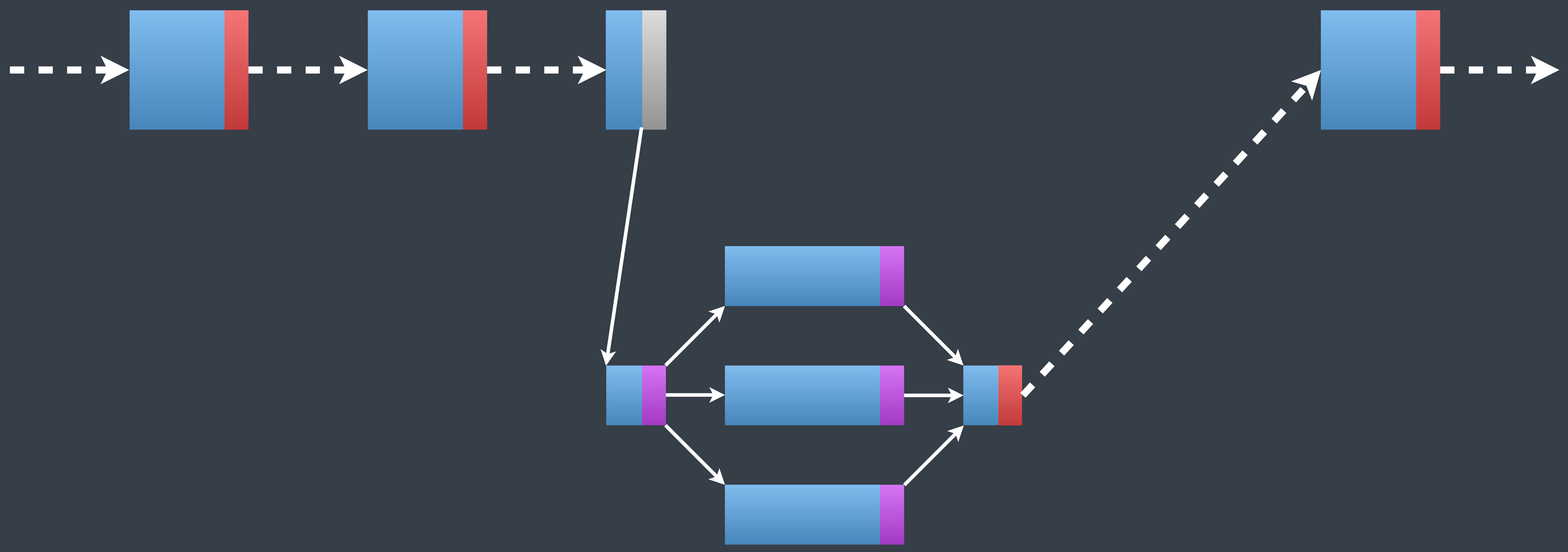
1. exchange continuations

serializer



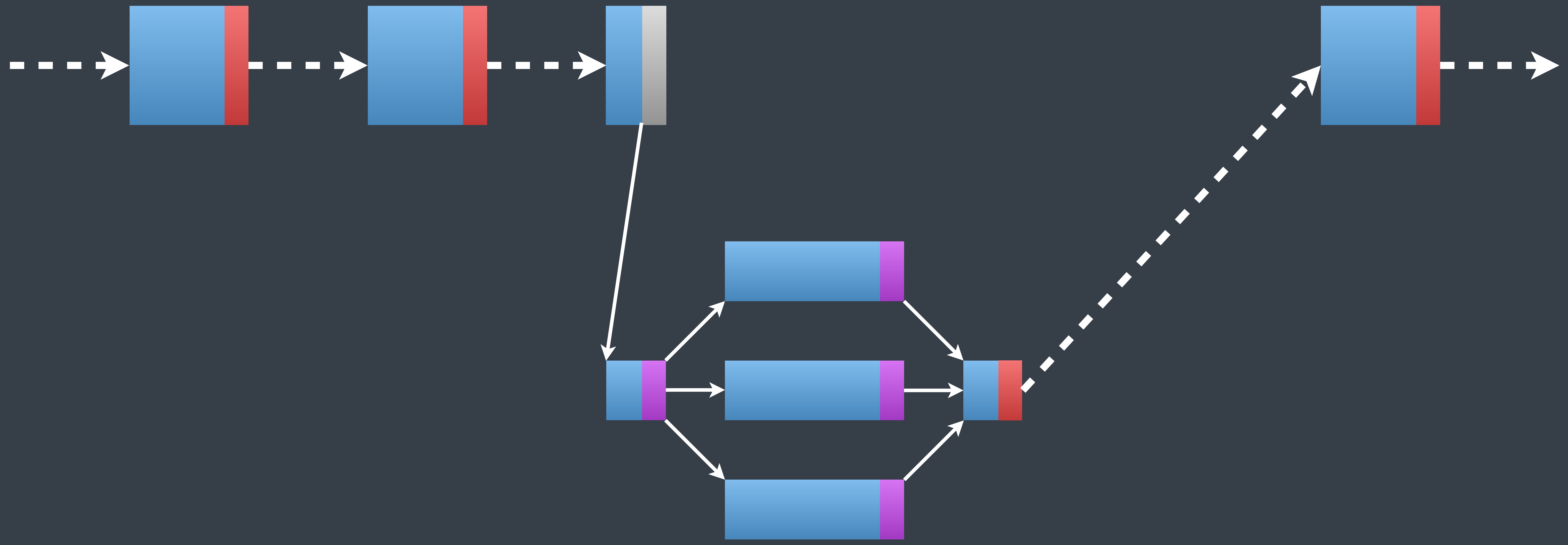
2. call new logic

serializer



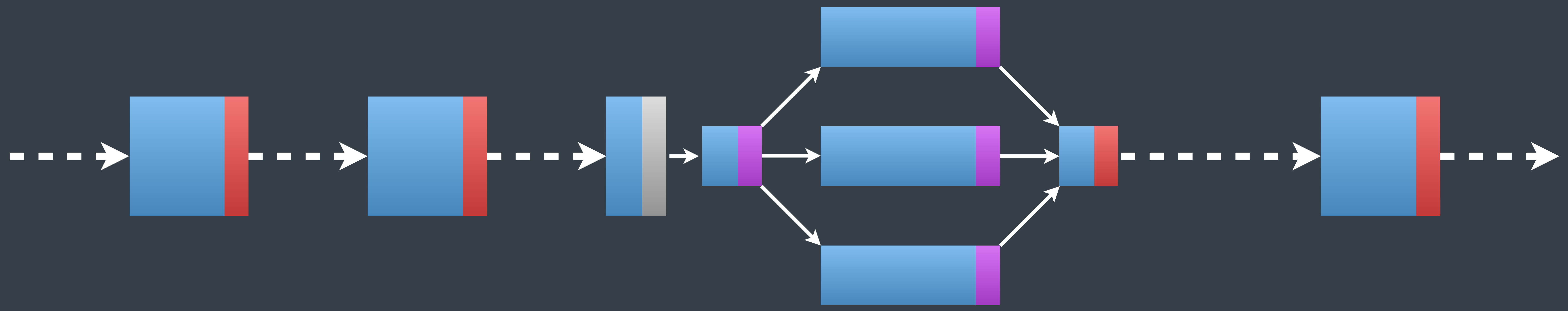
3. remove old logic

serializer



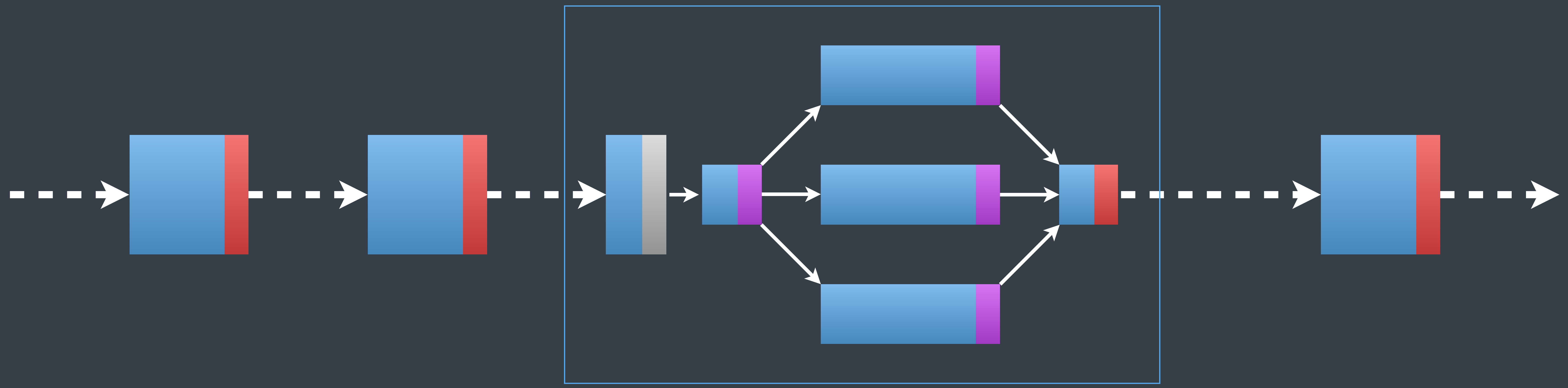
3. remove old logic

serializer



done

serializer



done

```cpp
void profile_activity(ProfileActivityResPtr res) {
    // 1. exchange continuations
    auto cont = concore::exchange_cur_continuation();

    // 2. call new logic (spawn tasks)
    concore::task lastTask{[res]{ aggregate_results(res); }, {}, cont};
    concore::finish_task ft(std::move(lastTask), 3);
    auto event = doneTask.event();
    concore::spawn([event, res] {
        query_access(res);
        event.notify_done();
    });
    concore::spawn([event, res] {
        query_transactions(res);
        event.notify_done();
    });
    concore::spawn([event, res] {
        query_tickets(res);
        event.notify_done();
    });
}

my_serializer.execute([r] { profile_activity(r); });
```


top-down approach

An Example

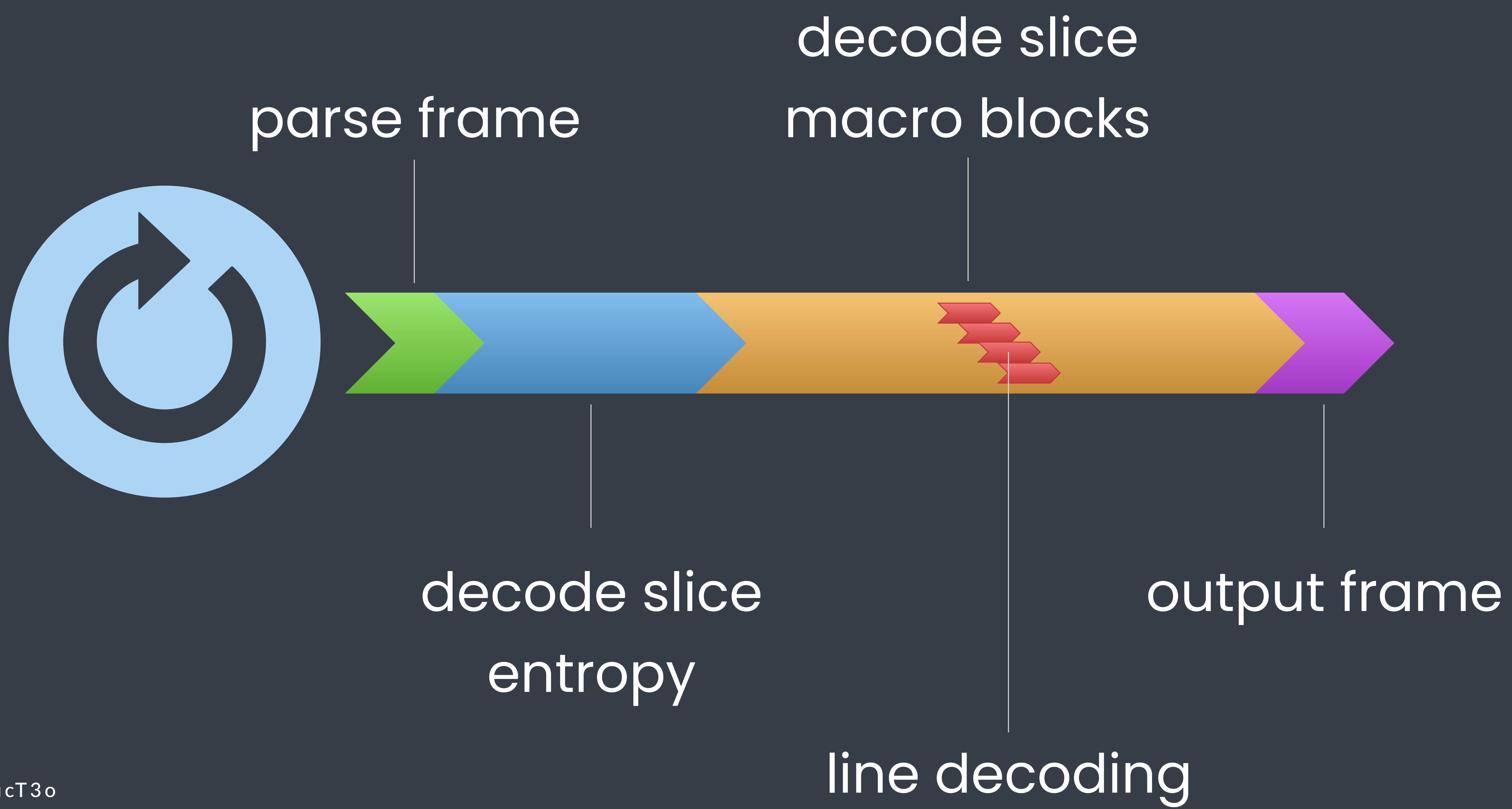
5

h264dec

video decompression software

part of StarBench parallel benchmark suite

basic flow


decode slice
macro blocks
parse frame
decode slice
entropy
output frame
line decoding

concurrency constraints

- stages need to processed in order for a frame
- parsing needs to be in order
- decode macro-blocks needs to be in order
  - a line depends on the previous line
- frame output needs to be in order

1. Starbench pthreads solution

included in the benchmark

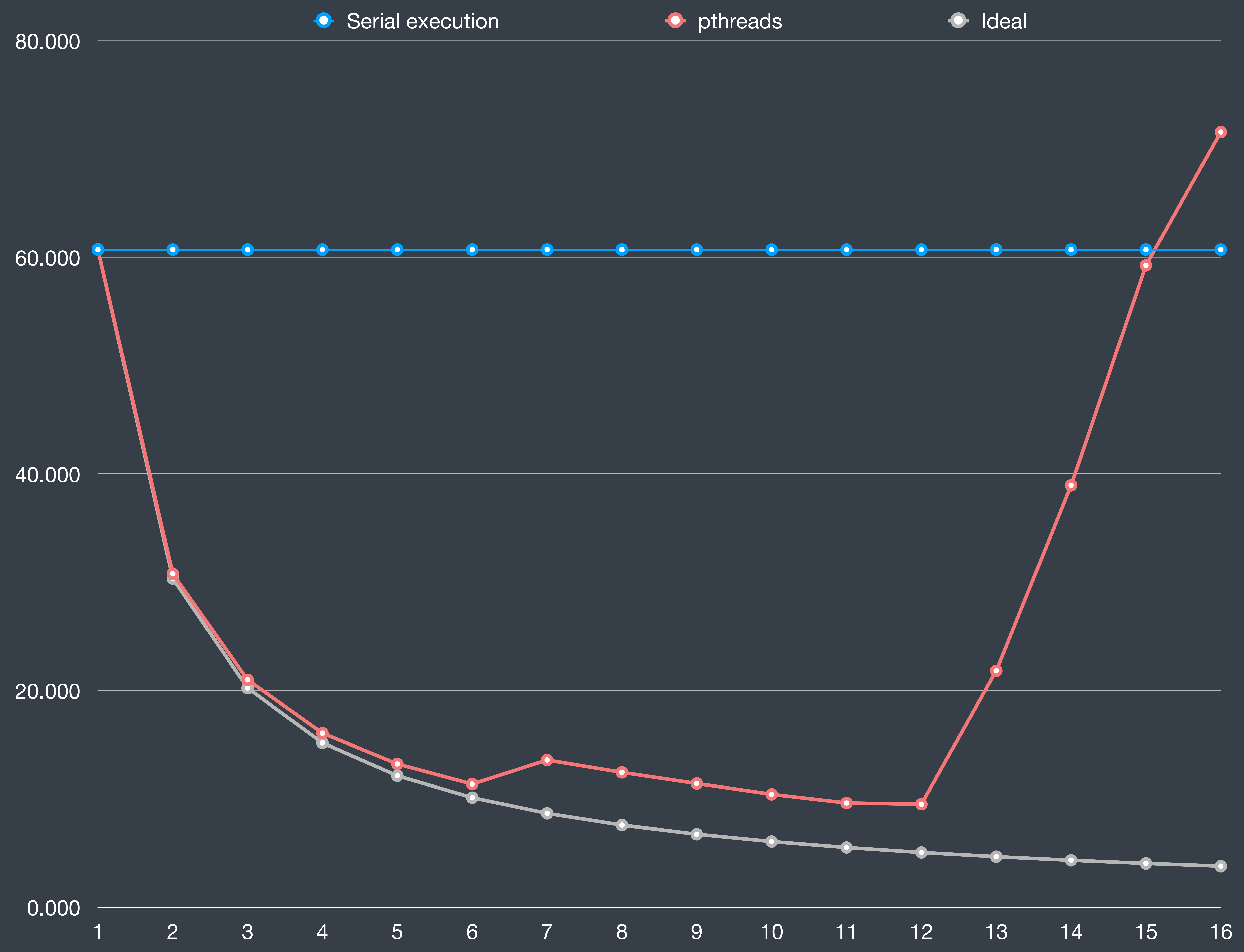
threads

1 parser thread
N threads for entropy / macro-blocks
1 reorder thread (!)
1 output thread

inter-frame dependency

busy wait for the previous macro-block

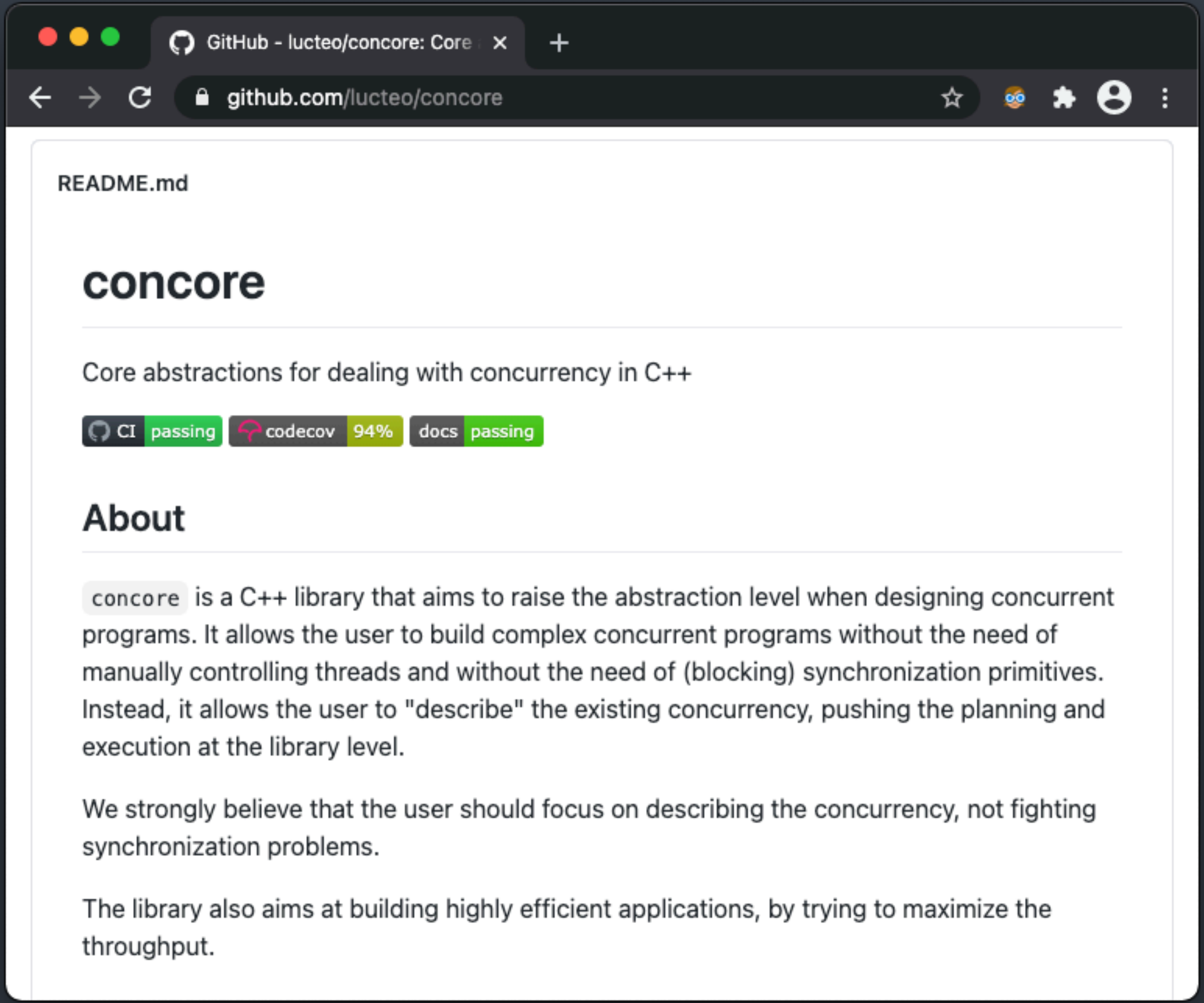
```
    for(i=0; i< mb_width; i++){
        if (frames || line>0){
            while (rle->mb_cnt >= rle->prev_line->mb_cnt -1);
        }
        h264_decode_mb_internal( d, d->mrs, s, &m[i]);
        rle->mb_cnt++;
    }
```



Serial execution
pthreads
Ideal
80.000
60.000
40.000
20.000
0.000
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

@LucT3o

2. concore solution



README.md

concore

Core abstractions for dealing with concurrency in C++

About

`concore` is a C++ library that aims to raise the abstraction level when designing concurrent programs. It allows the user to build complex concurrent programs without the need of manually controlling threads and without the need of (blocking) synchronization primitives. Instead, it allows the user to "describe" the existing concurrency, pushing the planning and execution at the library level.

We strongly believe that the user should focus on describing the concurrency, not fighting synchronization problems.

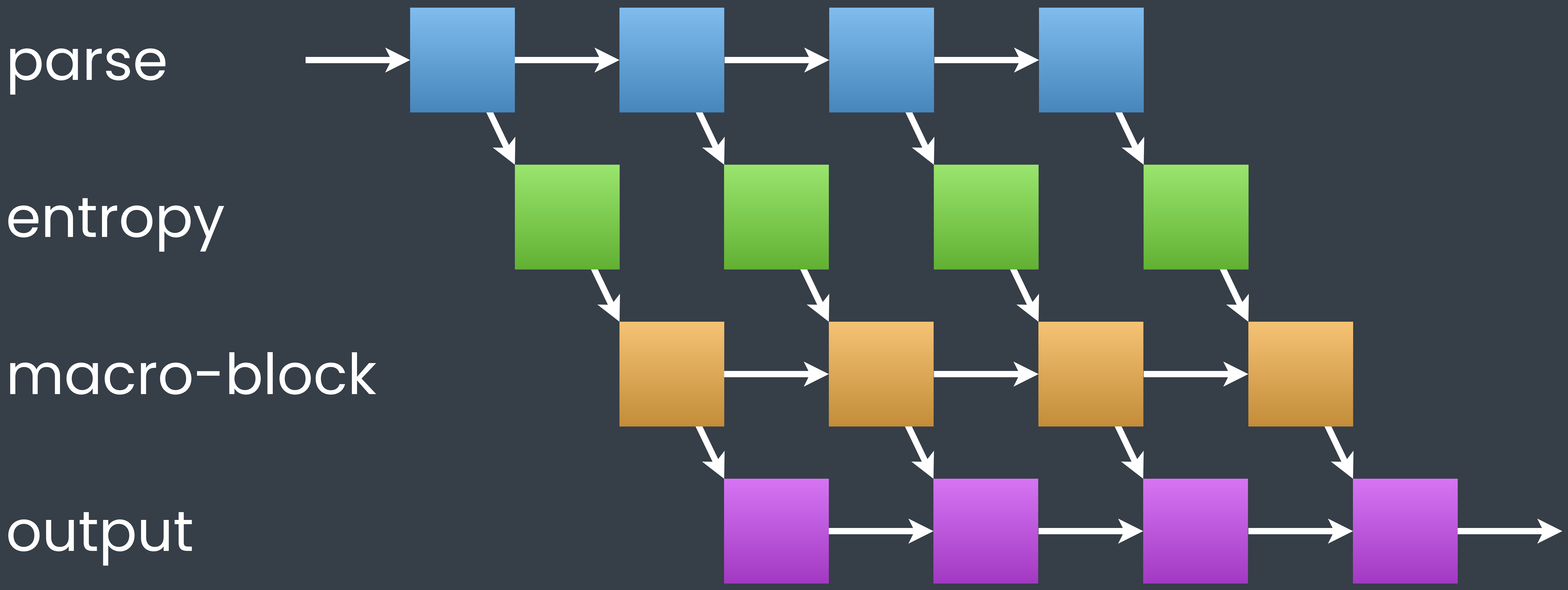
The library also aims at building highly efficient applications, by trying to maximize the throughput.

general approach

pipeline for general flow

tasks for macro-block lines

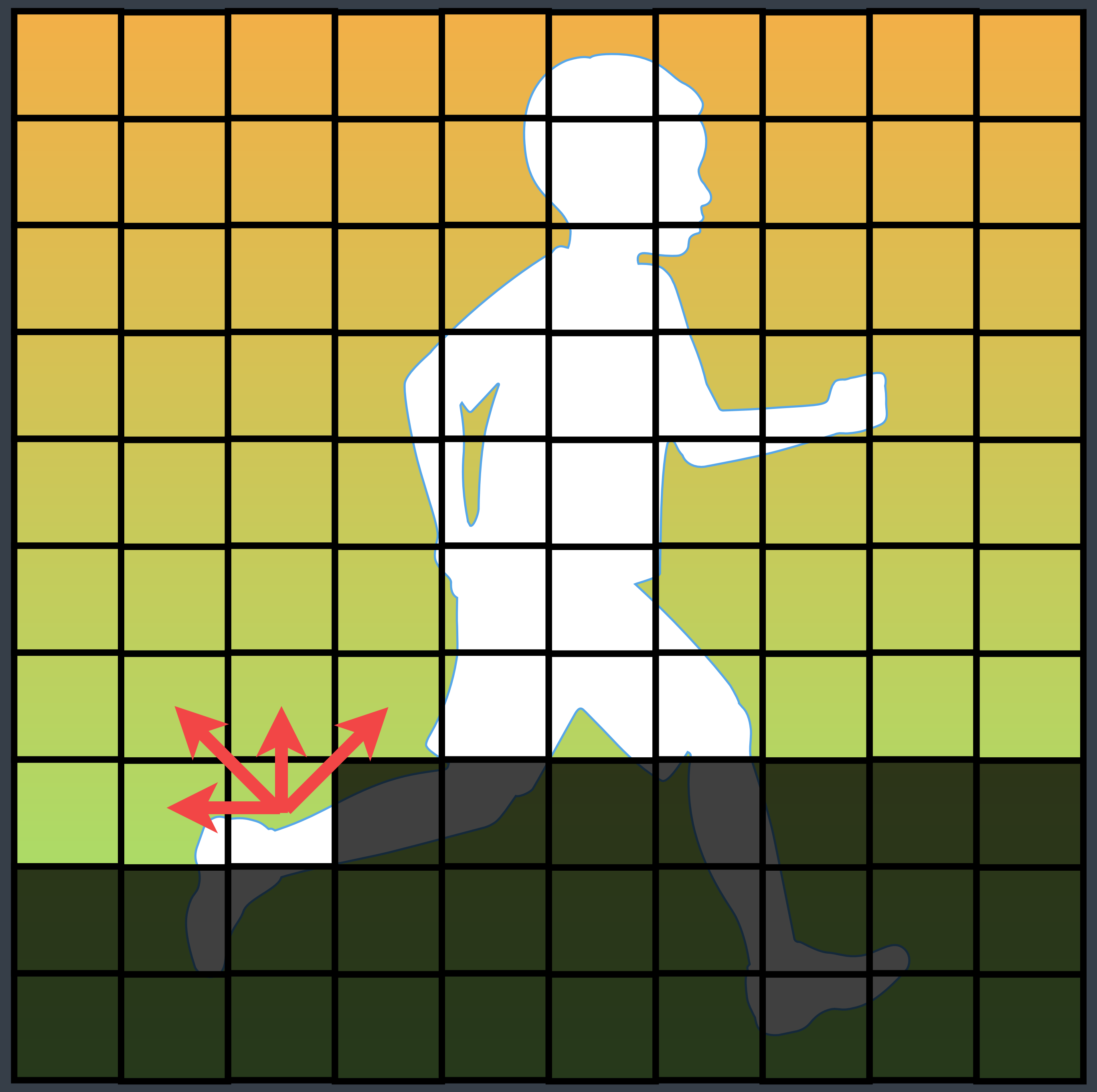
pipeline



pipeline

```
concore::task_group group = concore::task_group::create();

concore::pipeline<DecFrame> process{h->threads, group};

auto in_order = concore::stage_ordering::in_order;
auto conc = concore::stage_ordering::concurrent;

process.add_stage(in_order,
    [&process](DecFrame& frm) { stage_parse(frm, process); });
process.add_stage(conc, &stage_decode_slice_entropy);
process.add_stage(in_order, &stage_decode_slice_mb);
process.add_stage(in_order, &stage_gen_output);

// Push the first frame through the pipeline
process.push(DecFrame{0, &ctx});

// Wait until we process all the pipeline
concore::wait(group);
```

macro-block dependencies



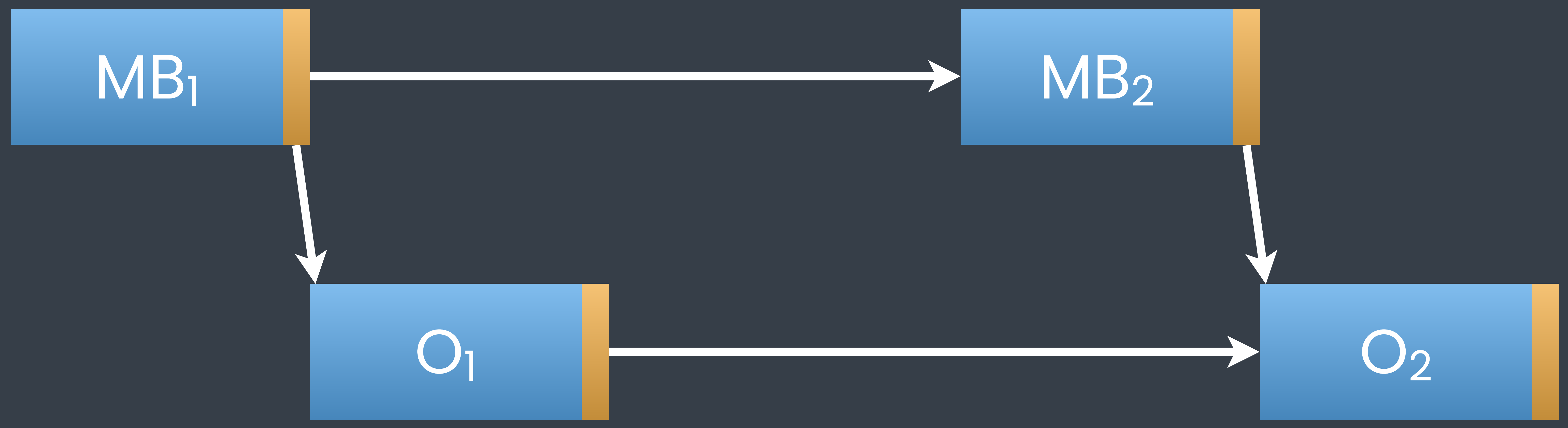
processing macro-blocks

```
struct process_matrix {
    using cell_fun_t = std::function<void(int x, int y)>;

    void start(int w, int h, cell_fun_t cf, concore::task&& donet) {
        width = w;
        height = h;
        cell_fun = cf;
        done_task = std::move(donet);
        ref_counts.resize(h * w);
        for ( int y=0; y<h; y++ ) {
            for ( int x=0; x<w; x++ )
                ref_counts[y*w + x].store( x==0 || y==0 || x==w-1 ? 1 : 2 );
        }

        // Start with the first cell
        concore::spawn(create_cell_task(0, 0));
    }
```

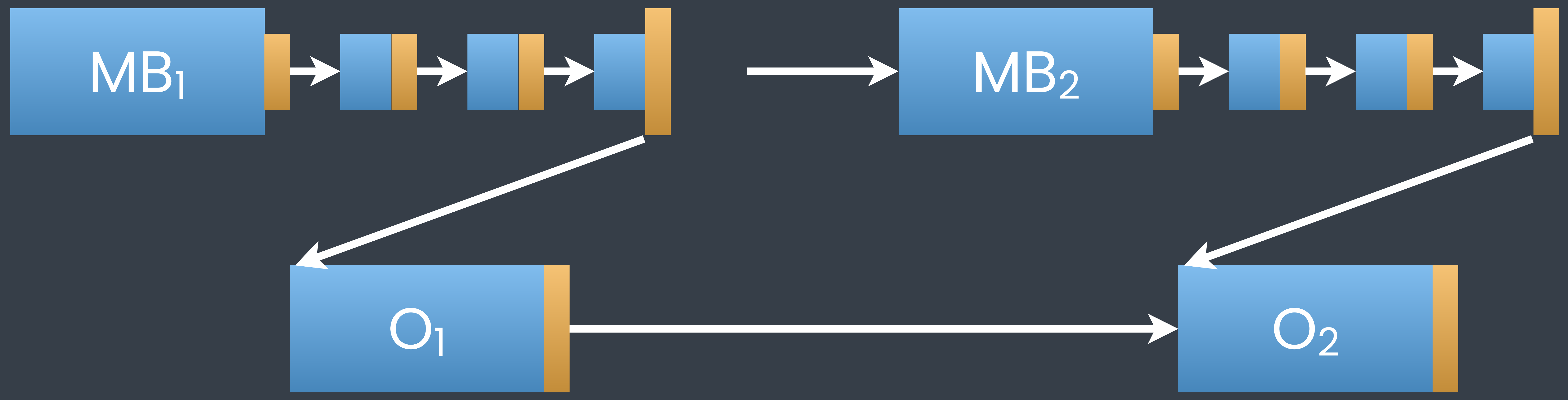
processing macro-blocks

```
…
concore::task create_cell_task(int x, int y) {
    auto f = [this, x, y] { cell_fun(x, y); };
    auto cont = [this, x, y] (std::exception_ptr) {
        if (y < height - 1 && x > 0)        // Spawn bottom task
            unblock_cell(x - 1, y + 1);
        if (x < width - 1)                  // Spawn right task
            unblock_cell(x + 1, y, false);
        if (y == height-1 && x == width-1 ) // Finish?
            concore::spawn(std::move(done_task), false);
    };
    return concore::task{f, {}, cont};
}
void unblock_cell(int x, int y, bool wake_workers = true) {
    int idx = y * width + x;
    if (ref_counts[idx]-- == 1)
        concore::spawn(create_cell_task(x, y), wake_workers);
}
};
```

decomposition

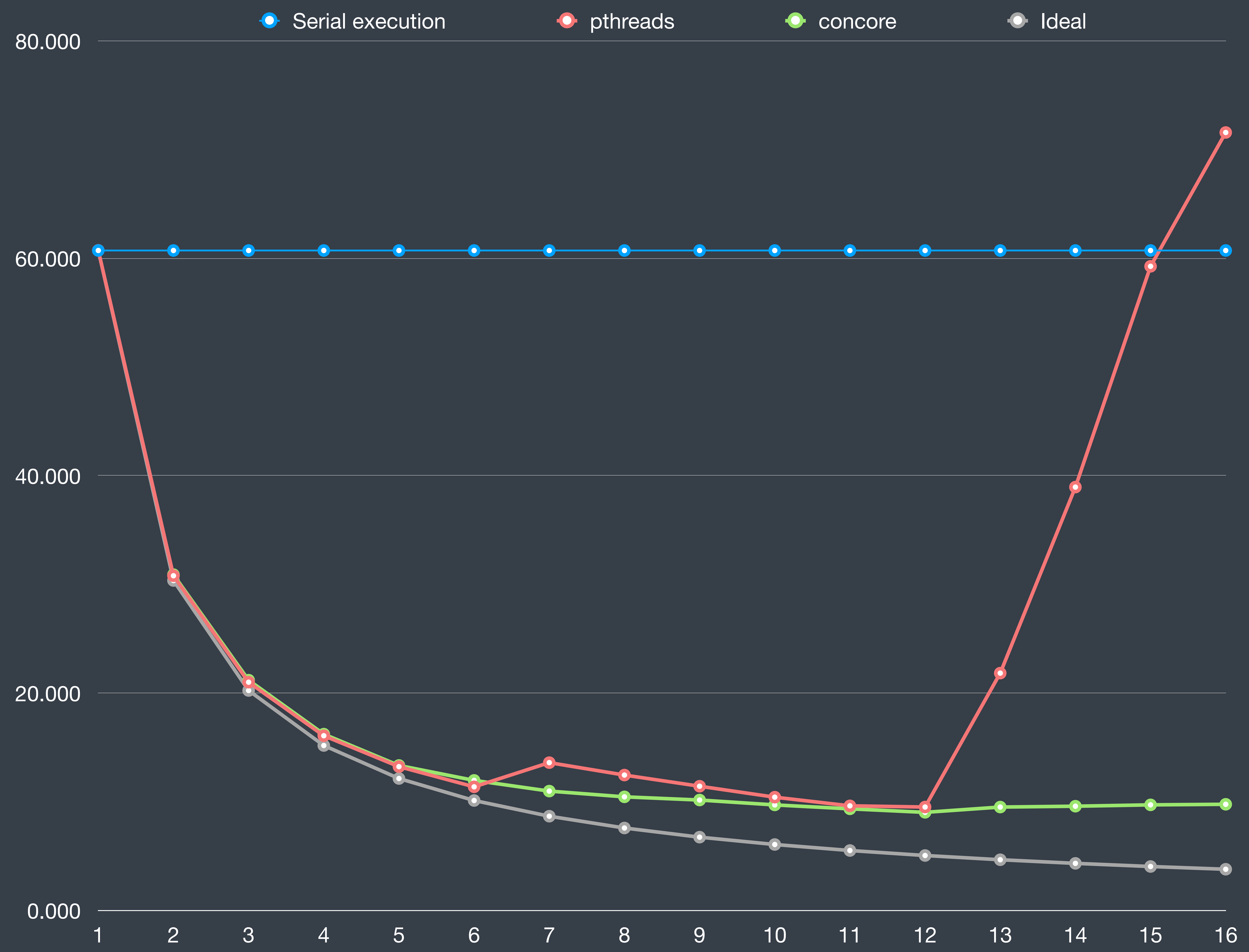


decomposition



decomposition

```
void stage_decode_slice_mb(DecFrame& frm) {
    ...
    // This will be broken into multiple tasks. Exchange continuation.
    auto cont = concore::exchange_cur_continuation();
    auto grp = concore::task_group::current_task_group();
    concore::task end_task{[] {}, grp, std::move(cont)};

    auto chunk_fun = [&frm] (int x, int y) {
        decode_slice_mb_chunk(frm.global_ctx, *frm.frame_data, x, y);
    };
    int width = h->mb_width/mb_line_chunk_size;
    fd.mb_processing.start(width, h->mb_height, chunk_fun, std::move(end_task));
}
```

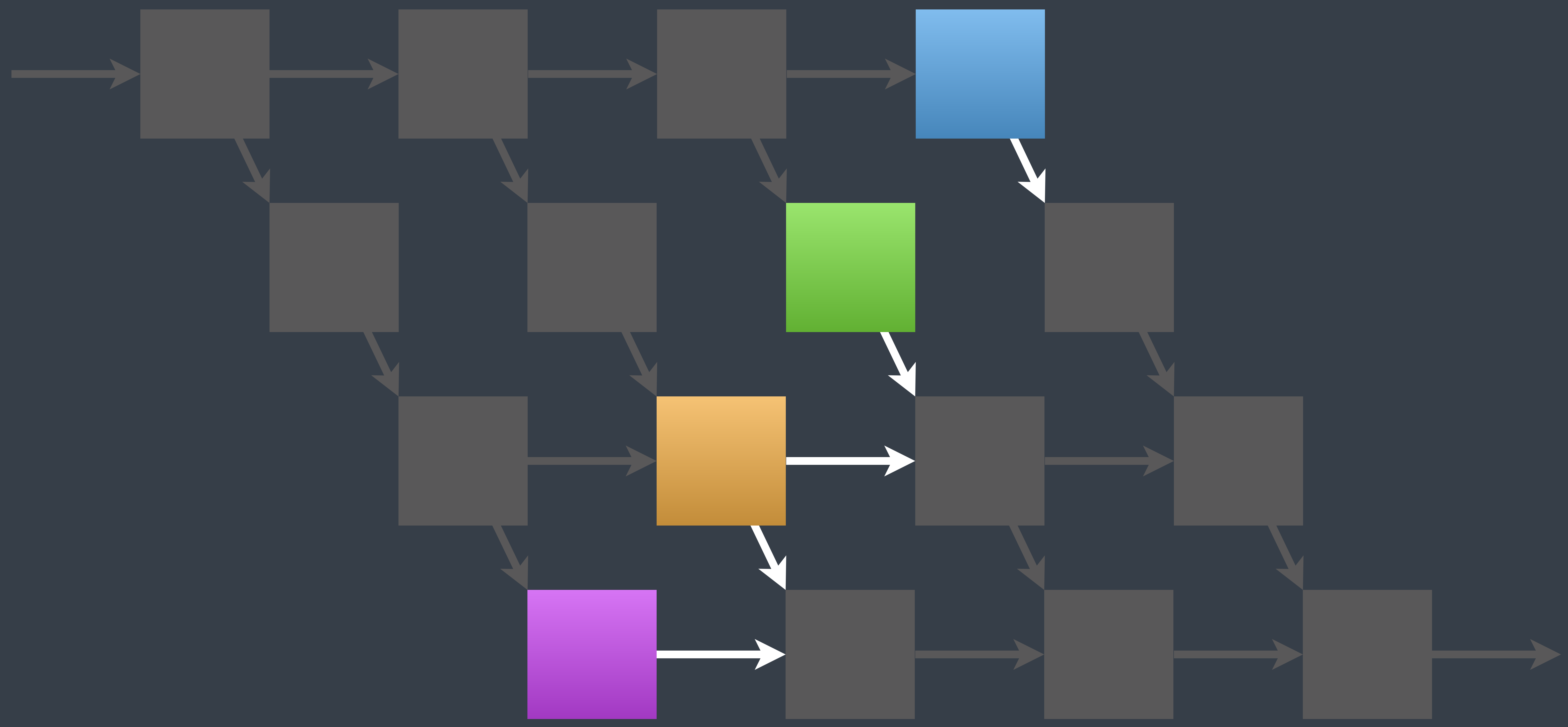

Serial execution
pthreads
concore
Ideal
80.000
60.000
40.000
20.000
0.000
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
@LucT3o

more performance

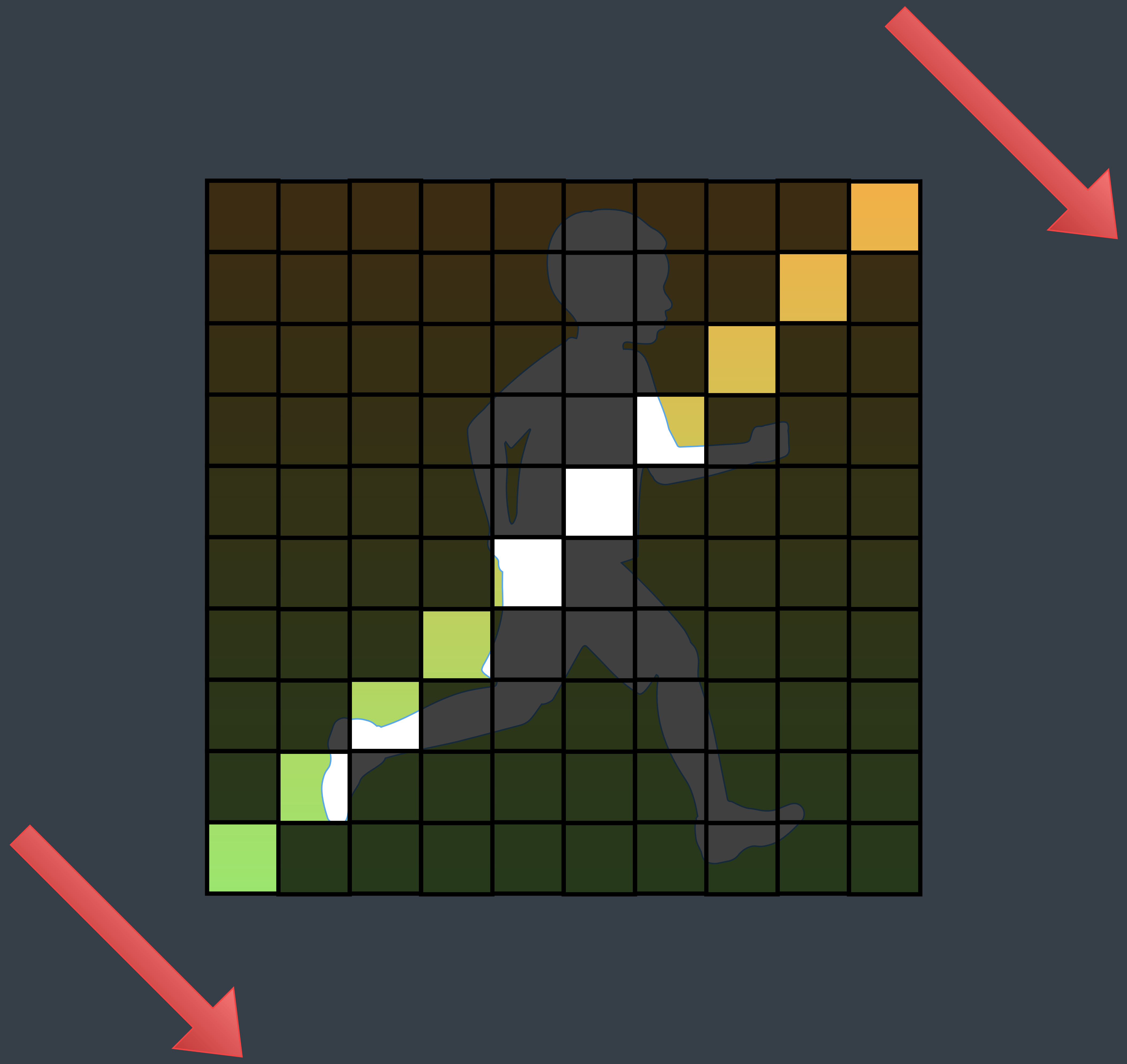
not a lot of (naive) concurrency

small tasks → more overhead

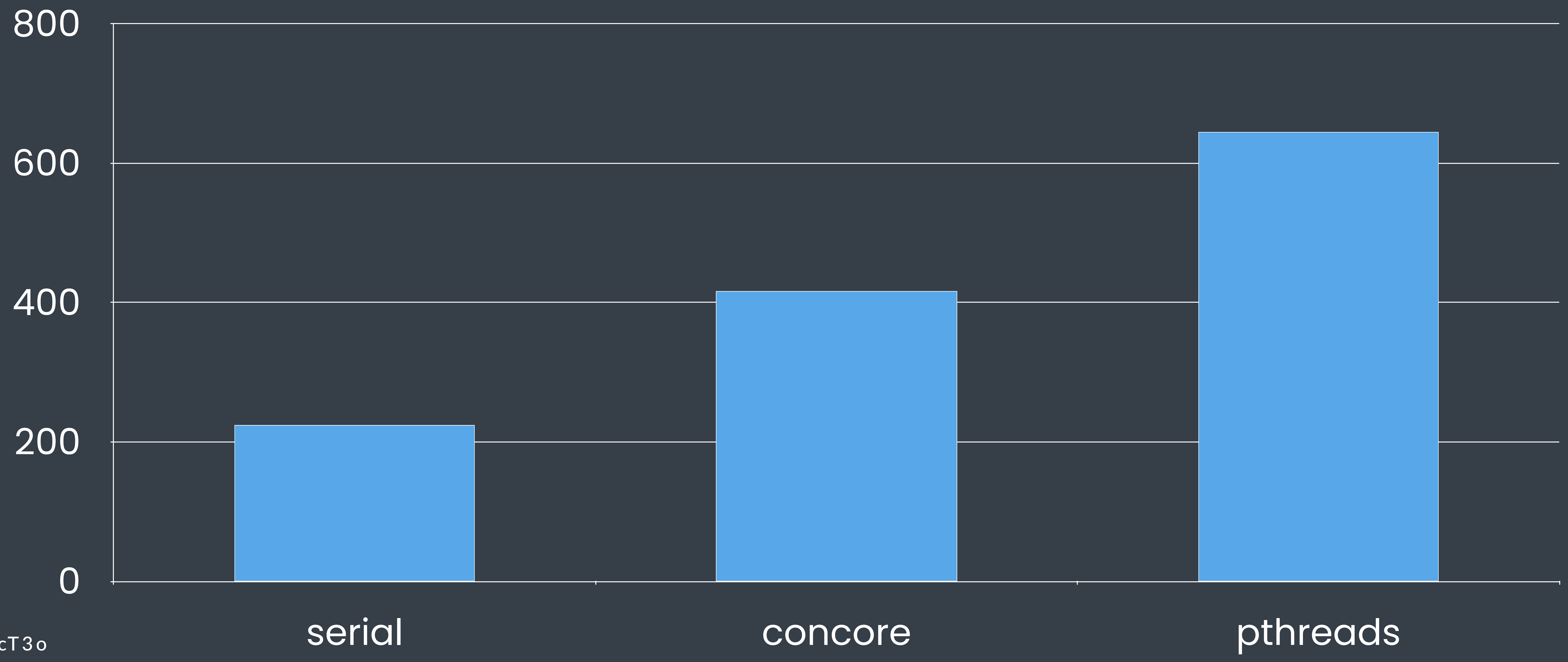
limited concurrency (1)



limited concurrency (2)



lines of code



results

easy to write

high-level concurrency abstractions

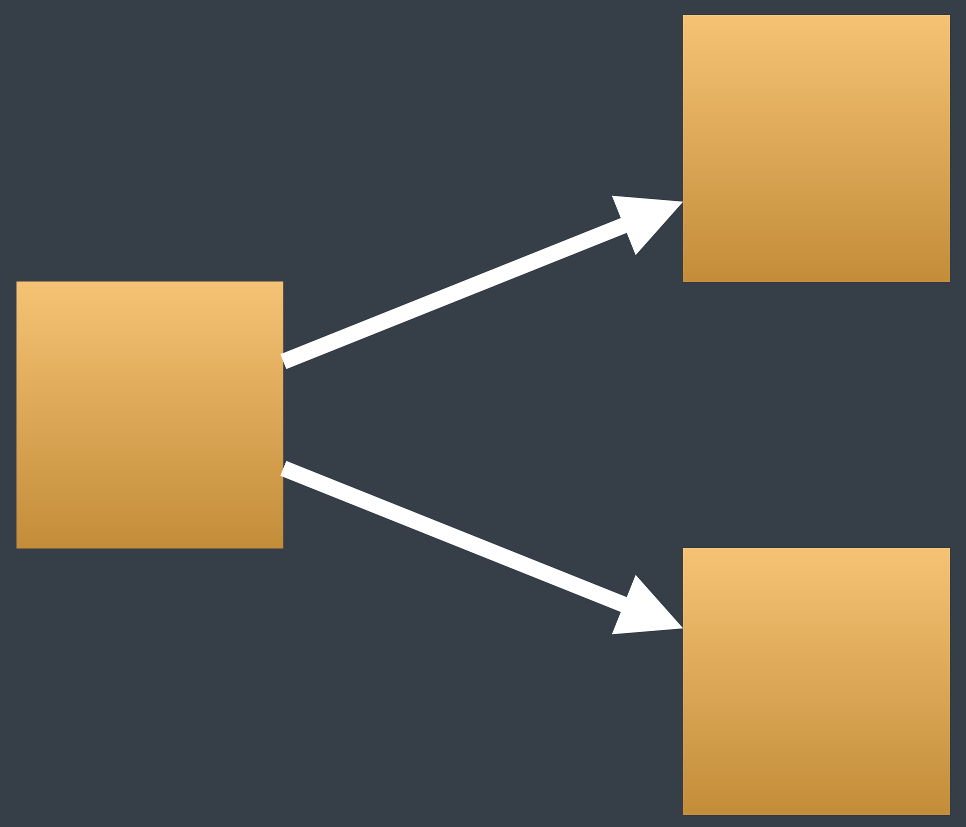
efficient

https://github.com/lucteo/h264dec-concore

Concurrency Patterns

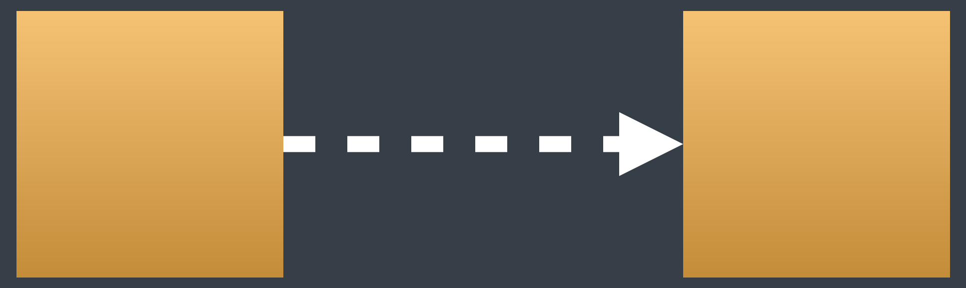
6

1. create concurrent work



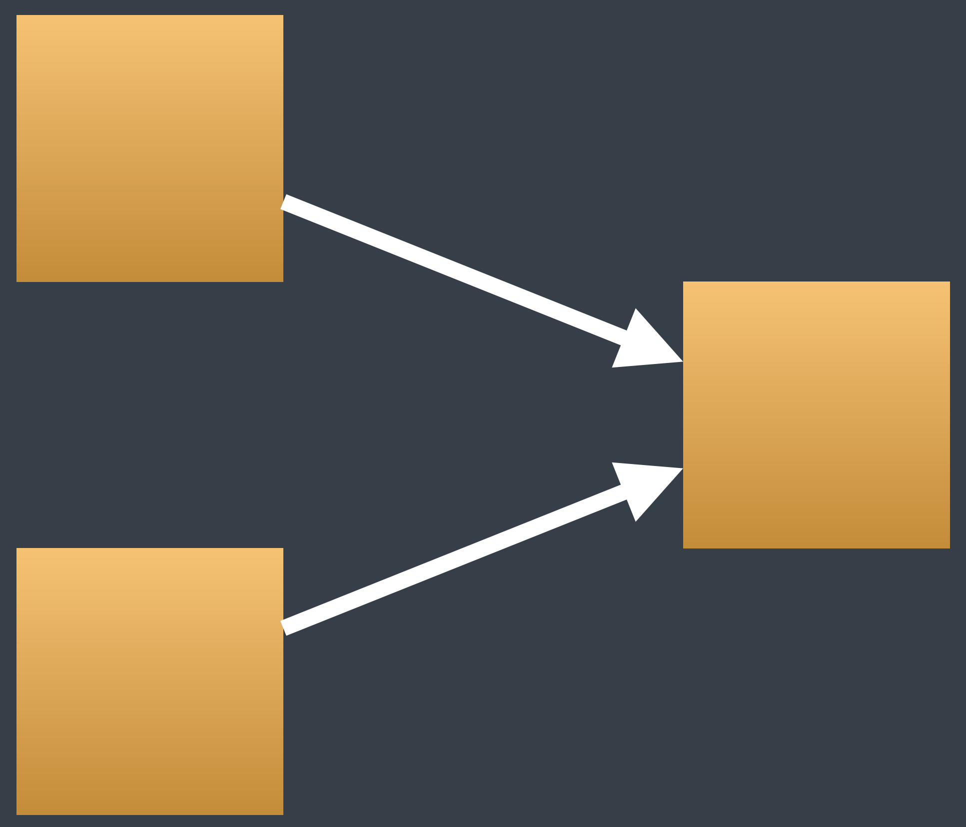
```cpp
void start() {
    initComponents();
    concore::spawn([]{ loadAssets(); });
    concore::spawn([]{ initilizeComputationEngine (); });
}
```

2. delayed continuation



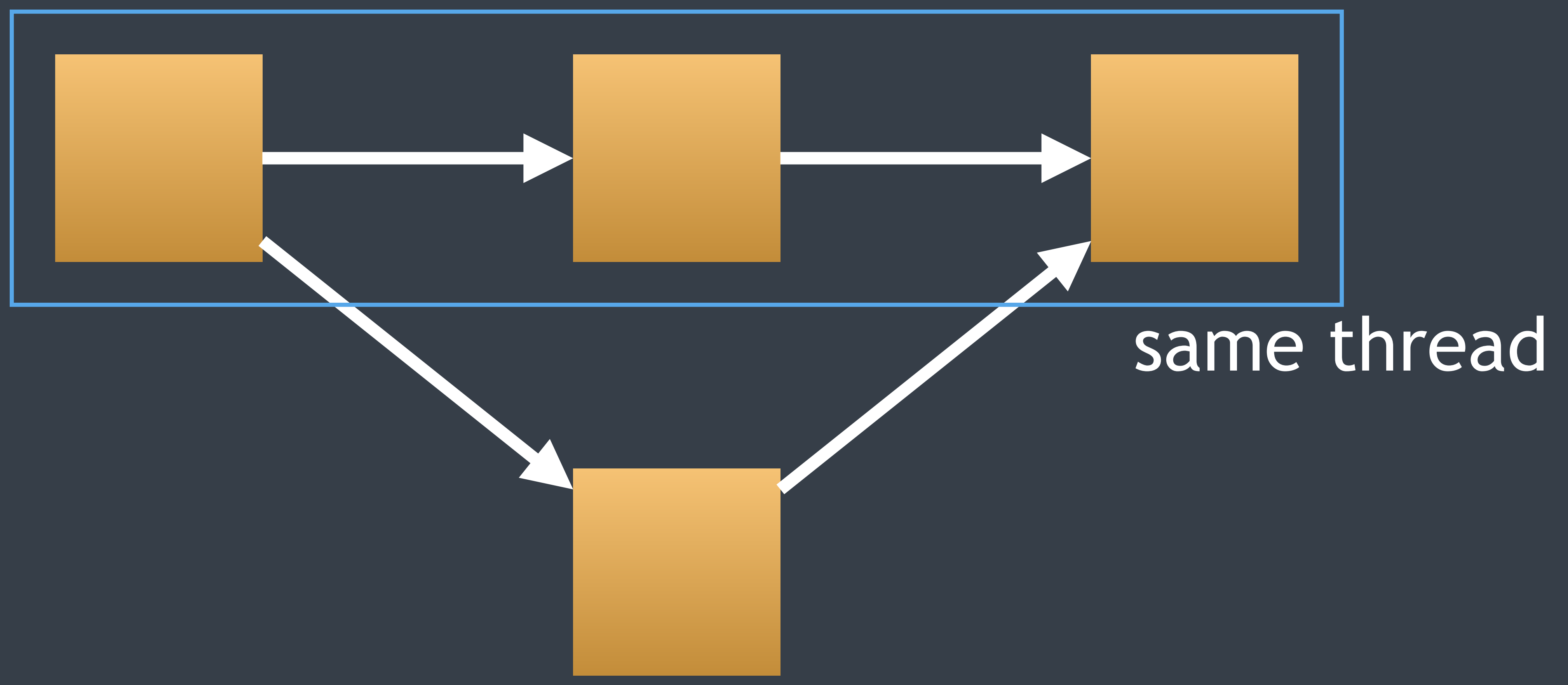
```cpp
void handleResponse(HttpResponseData respData, HandlerType handler) {
    // the work for this task: process the response
    HttpResponse resp = respData.toResponse();
    // create a continuation to handle the response
    concore::task cont{[resp = std::move(resp), handler] {
        handler(resp);
    }};
    concore::spawn(std::move(cont));
}
void httpAsyncCall(const char* url, HandlerType handler) {
    // does HTTP logic, and eventually async calls handleRespnse()
}
void useHttpCode() {
    // the work to be executed as a continuation
    HandlerType handler = [](HttpResponse resp) {
        printResponse(resp);
    };
    // call the Http code asynchronously, passing the continuation work
    httpAsyncCall("www.google.com", handler);
    // whenever the response comes back, the above handler is called
}
```

3. join



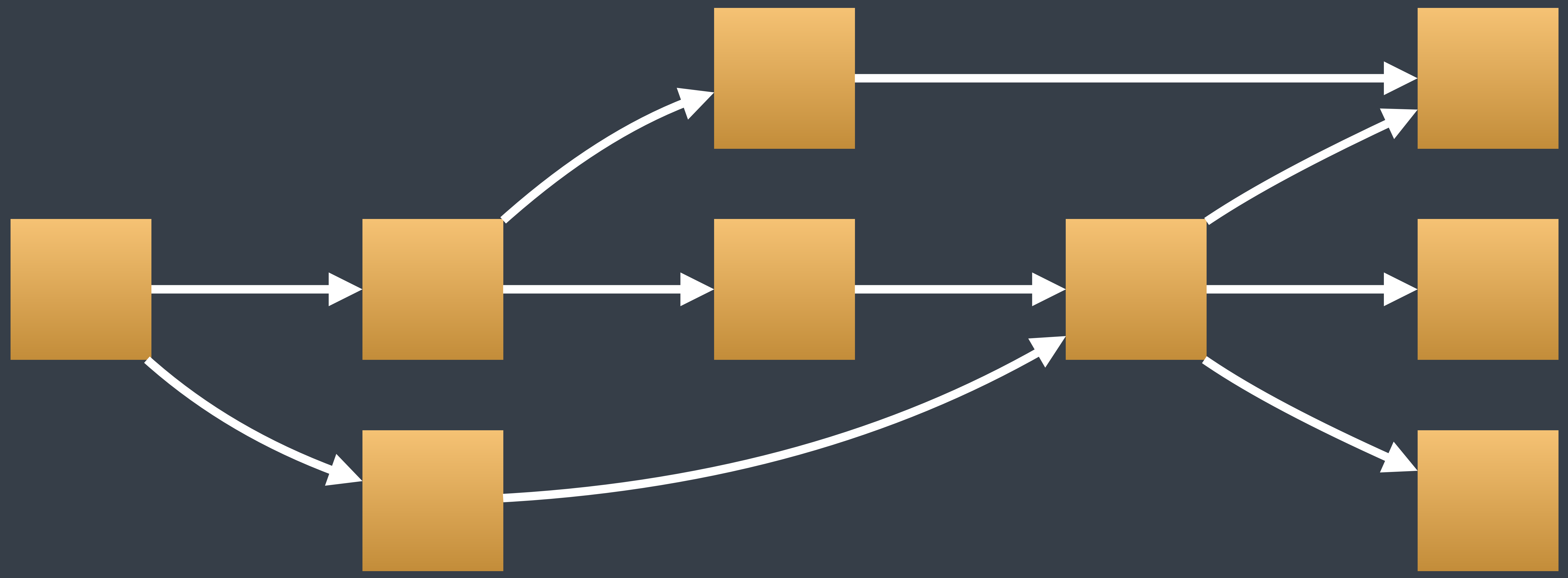
```cpp
concore::finish_task doneTask([]{
    listenForRequests();
}, 2); // waits on 2 tasks

// Spawn 2 tasks
auto event = doneTask.event();
concore::spawn([event] {
    loadAssets();
    event.notify_done();
});
concore::spawn([event] {
    initilizeComputationEngine();
    event.notify_done();
});
// When they complete, the done task is triggered
```

4. fork-join



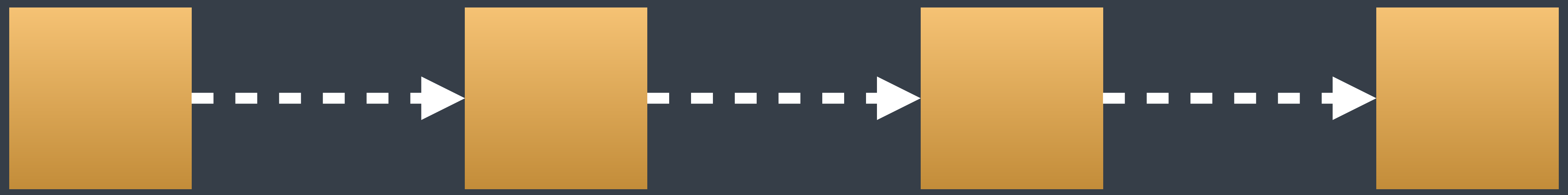
```cpp
template <typename F>
void conc_apply(int start, int end, int granularity, F f) {
    if (end - start <= granularity)
        for (int i = start; i < end; i++)
            f(i);
    else {
        int mid = start + (end - start) / 2;
        auto grp = concore::task_group::create();
        concore::spawn([=] { conc_apply(start, mid, granularity, f); }, grp);
        concore::spawn([=] { conc_apply(mid, end, granularity, f); }, grp);
        concore::wait(grp);
    }
}
```

5. task graphs

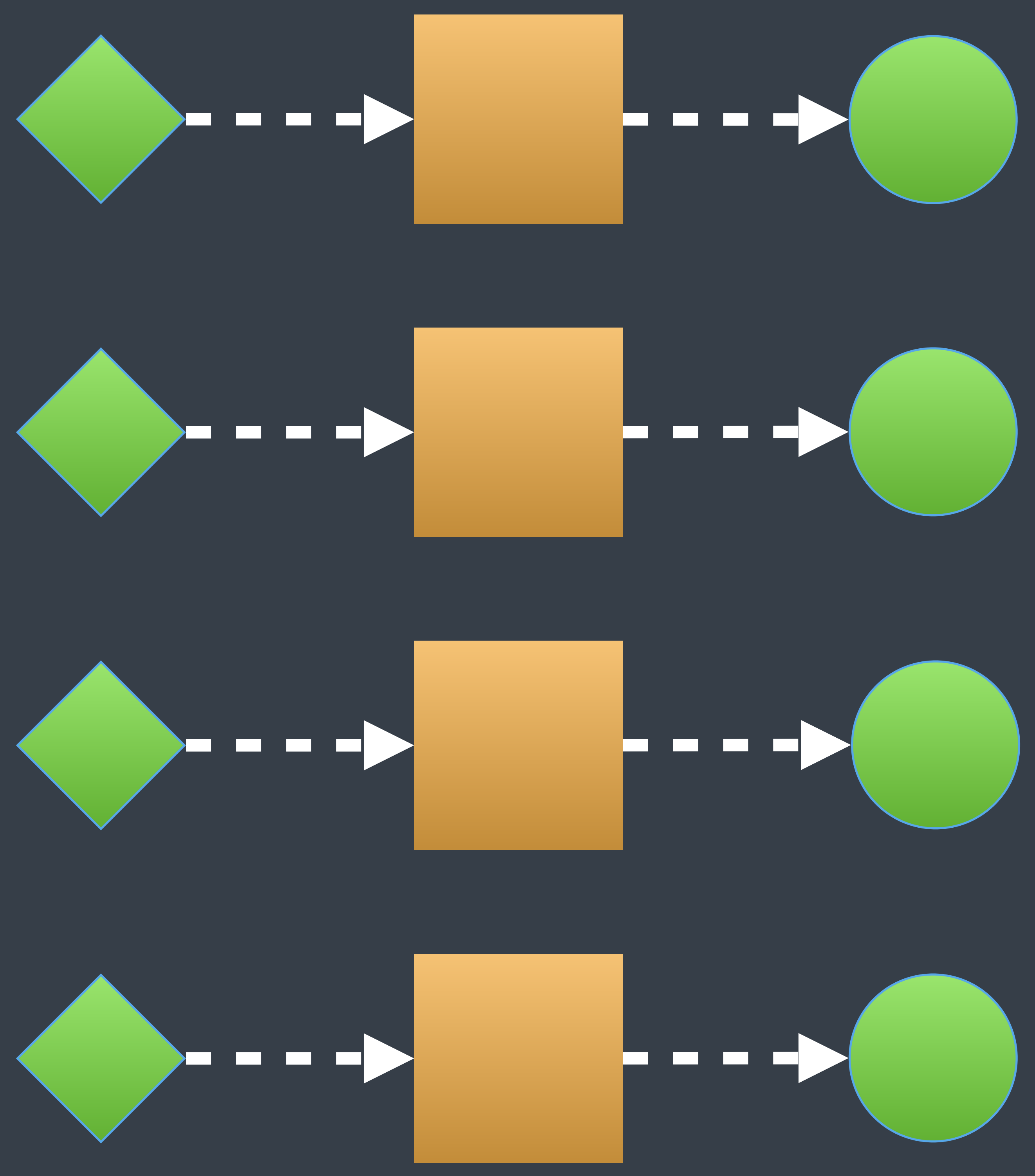


```cpp
std::shared_ptr<RequestData> data = CreateRequestData();
// create the tasks
concore::chained_task t1{[data] { ReadRequest(data); }};
concore::chained_task t2{[data] { Parse(data); }};
concore::chained_task t3{[data] { Authenticate(data); }};
concore::chained_task t4{[data] { StoreBeginAction(data); }};
concore::chained_task t5{[data] { AllocResources(data); }};
concore::chained_task t6{[data] { ComputeResult(data); }};
concore::chained_task t7{[data] { StoreEndAction(data); }};
concore::chained_task t8{[data] { UpdateStats(data); }};
concore::chained_task t9{[data] { SendResponse(data); }};
// set up dependencies
concore::add_dependencies(t1, {t2, t3});
concore::add_dependencies(t2, {t4, t5});
concore::add_dependency(t4, t7);
concore::add_dependencies({t3, t5}, t6);
concore::add_dependencies(t6, {t7, t8, t9});
// start the graph
concore::spawn(t1);
```

6. serializers



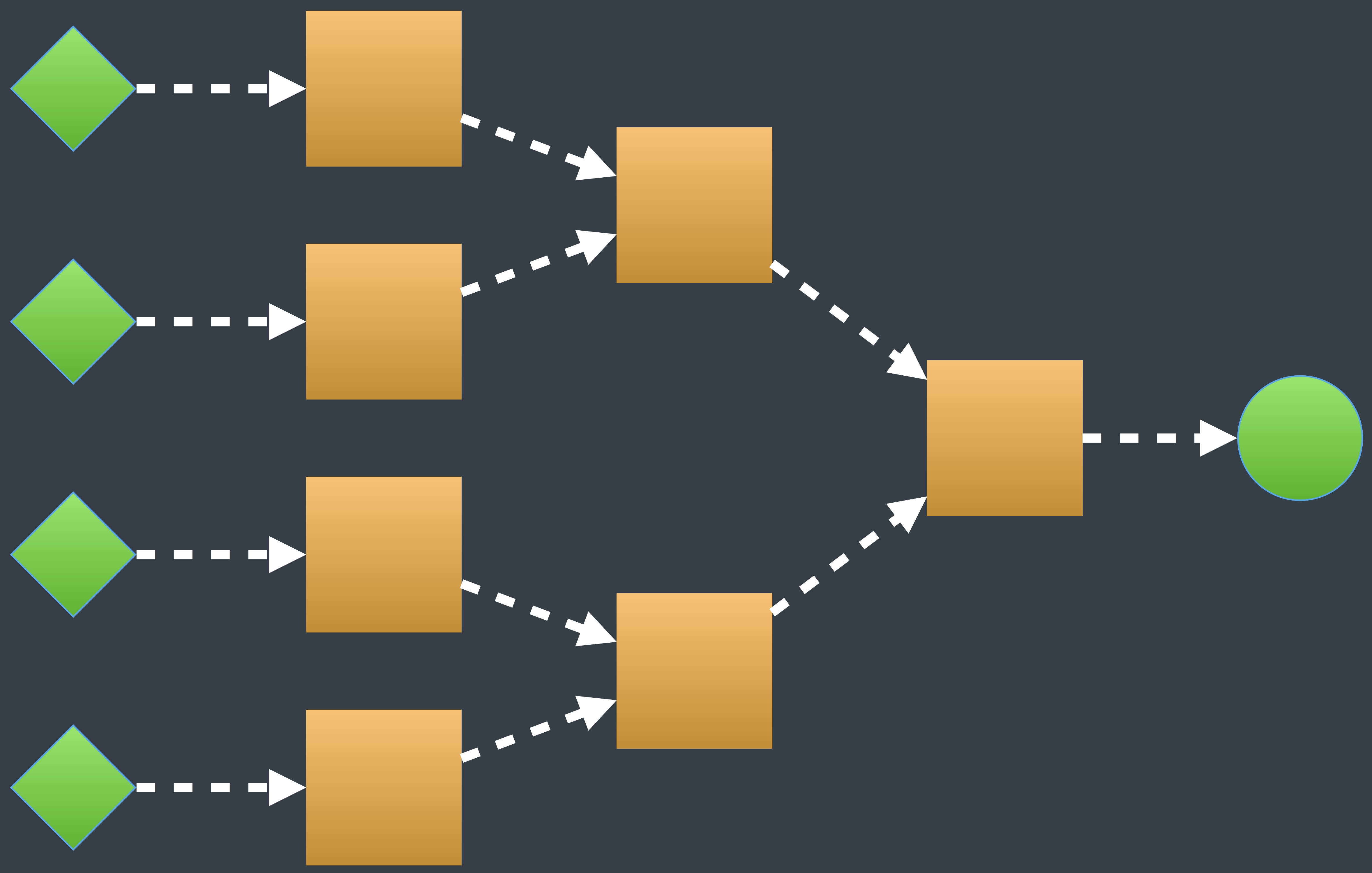
7. concurrent for



```cpp
std::vector<int> ids = getAssetIds();
int n = ids.size();
std::vector<Asset> assets(n);

concore::conc_for(0, n, [&](int i) { assets[i] = prepareAsset(ids[i]); });
```

```cpp
template< class ExecutionPolicy, class ForwardIt, class UnaryFunction2 >
void for_each( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last,
               UnaryFunction2 f );
```

8. concurrent reduce

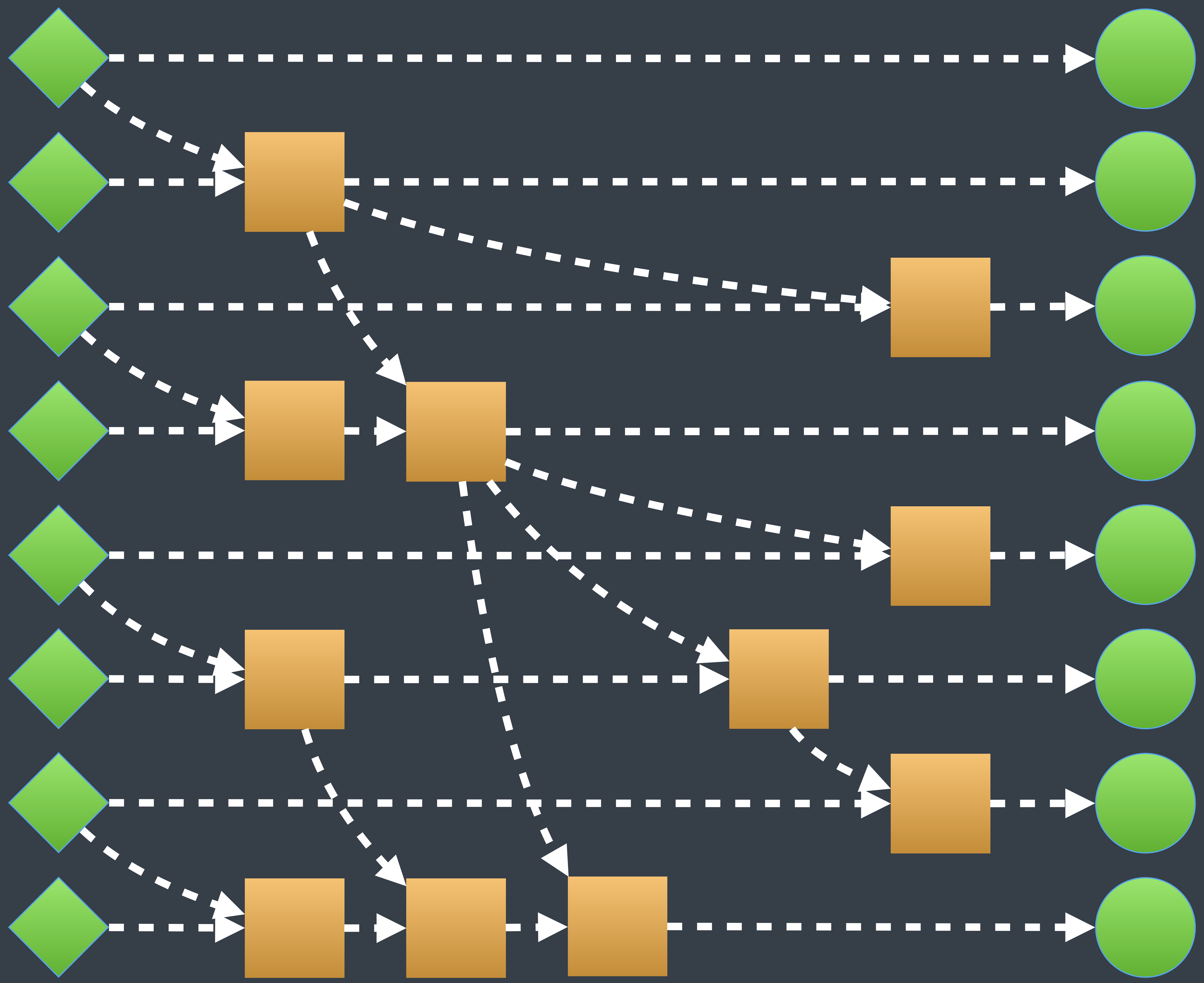


```cpp
std::vector<Resource> res = getResources();

auto oper = [&](int prevMem, const Resource& res) -> int {
    return prevMem + getMemoryConsumption(res);
};
auto reduce = [](int lhs, int rhs) -> int { return lhs + rhs; };

int totalMem = concore::conc_reduce(res.begin(), res.end(), 0, oper, reduce);
```

@LucT3o

```cpp
template<class ExecutionPolicy,
         class ForwardIt, class T, class BinaryOp, class UnaryOp>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt first, ForwardIt last,
                   T init, BinaryOp binary_op, UnaryOp unary_op);
```

9. concurrent scan



```cpp
std::vector<FeatureVector> in = getInputData();
std::vector<FeatureVector> out(in.size());

auto op = [](FeatureVector lhs, FeatureVector rhs) -> FeatureVector {
    return combineFeatures(lhs, rhs);
};

concore::conc_scan(in.begin(), in.end(), out.begin(), FeatureVector(), op);
```

```cpp
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2,
          class BinaryOperation, class T >
ForwardIt2 inclusive_scan( ExecutionPolicy&& policy,
                           ForwardIt1 first, ForwardIt1 last,
                           ForwardIt2 d_first,
                           BinaryOperation binary_op, T init );
```

high-level concurrency abstractions

no more low-level primitives

7

Conclusions

My second remark is that our intellectual powers are geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed

Edgar Dijkstra

@LucT3o

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** *B* **then** *A*), alternative clauses (**if** *B* **then** *A*1 **else** *A*2), choice clauses as introduced by C. A. R. Hoare (case[i] of (*A*1, *A*2, ⋯ , *An*)), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1$, $B2 \rightarrow E2$, ⋯ , $Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, *n* equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (according to the concurrent design) and the process (execution over multiple threads) as trivial as possible

Edgar Dijkstra
(paraphrased)

@LucT3o

(?) Edgar Dijkstra: Threads Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of$(A1, A2, \cdots, An)$), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1$, $B2 \rightarrow E2$, $\cdots$, $Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

Threads Considered Harmful

… but we have an alternative

… a global method

...with no locks

threading primitives

pushed down to the framework level

systematic way

raising the abstraction level
composable/decomposable

no excuse for
raw threads and locks

http://nolocks.org

Thank You

@LucT3o

lucteo.ro

nolocks.org

LUCIAN RADU TEODORESCU