accu conference 2025

Easy Senders/Receivers

Lucian Radu Teodorescu

Easy Senders/Receivers

LUCIAN RADU TEODORESCU
GARMIN

teachability problem

poll

using C++?

poll

using C++?
recommend C++ as easy?

easy senders/receivers

lower limit: as easy as using C++

poll

using `std::function`?

poll

using `std::function`?

using iterators?

easy senders/receivers

on **average**: using `std::function` + iterators

easy senders/receivers

upper limit: heavy metaprogramming

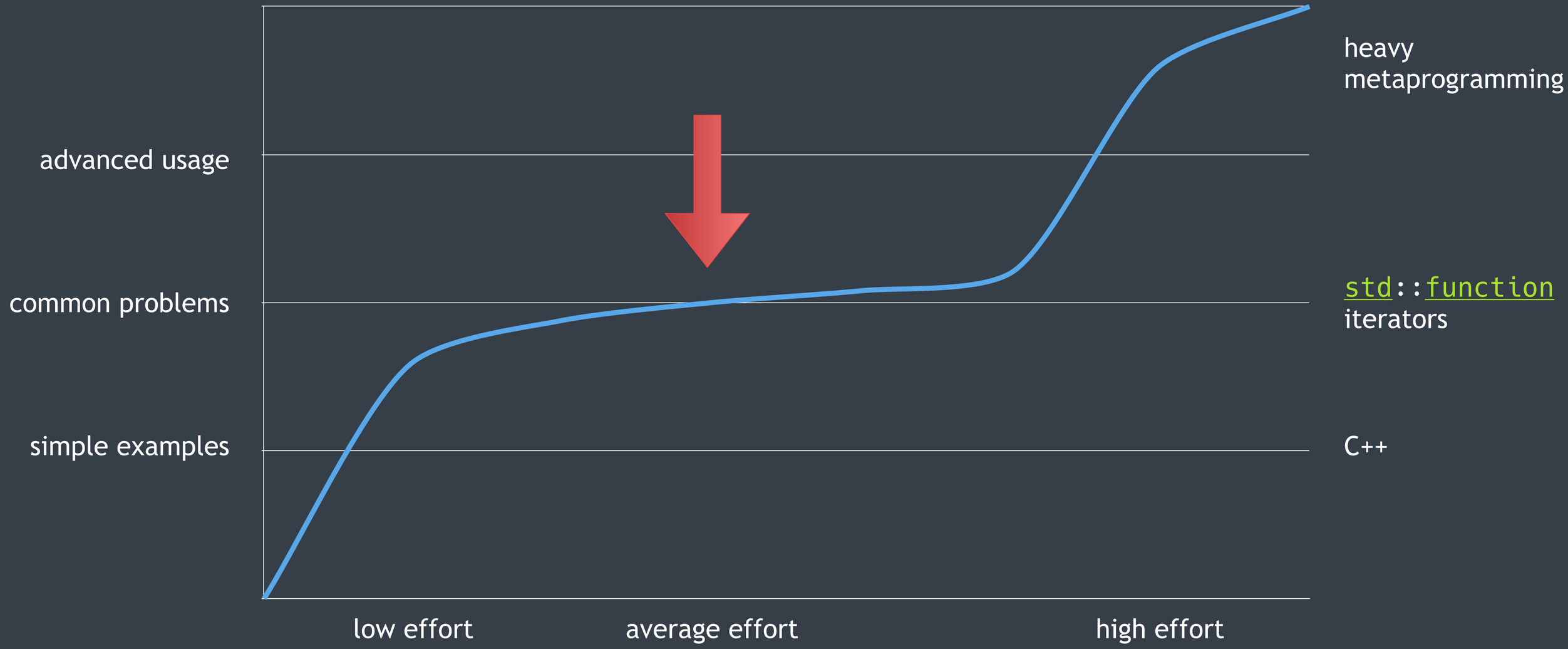
`std::function`

encapsulate work
defer execution
compose work w/o executing it

iterators

level of indirection
some use of templates
conventions
some complexity

learning senders/receivers



learning **anti-pattern**

become an expert first
then write simple examples

senders

like `std::function`, but for **async**

complexity: like iterators

1. Sync & async
2. Senders
3. Representing concurrency
4. Standard algorithms
5. Examples
6. Bonus
7. Conclusions

lucteo.ro/pres/2025-accu/

Sync & async

1

synchronous execution

function calls
happens on the same thread
consumes HW resources before the call is complete
call completes when the work is done

asynchronous execution

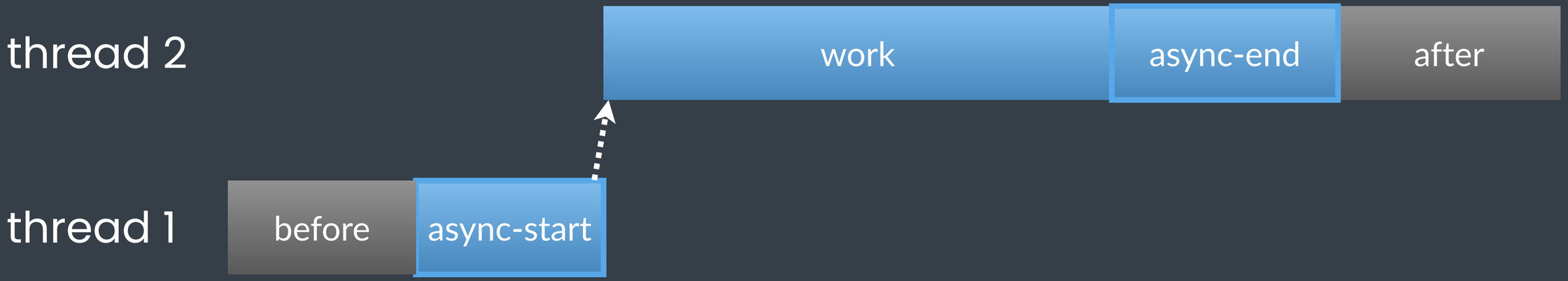
more than a function call

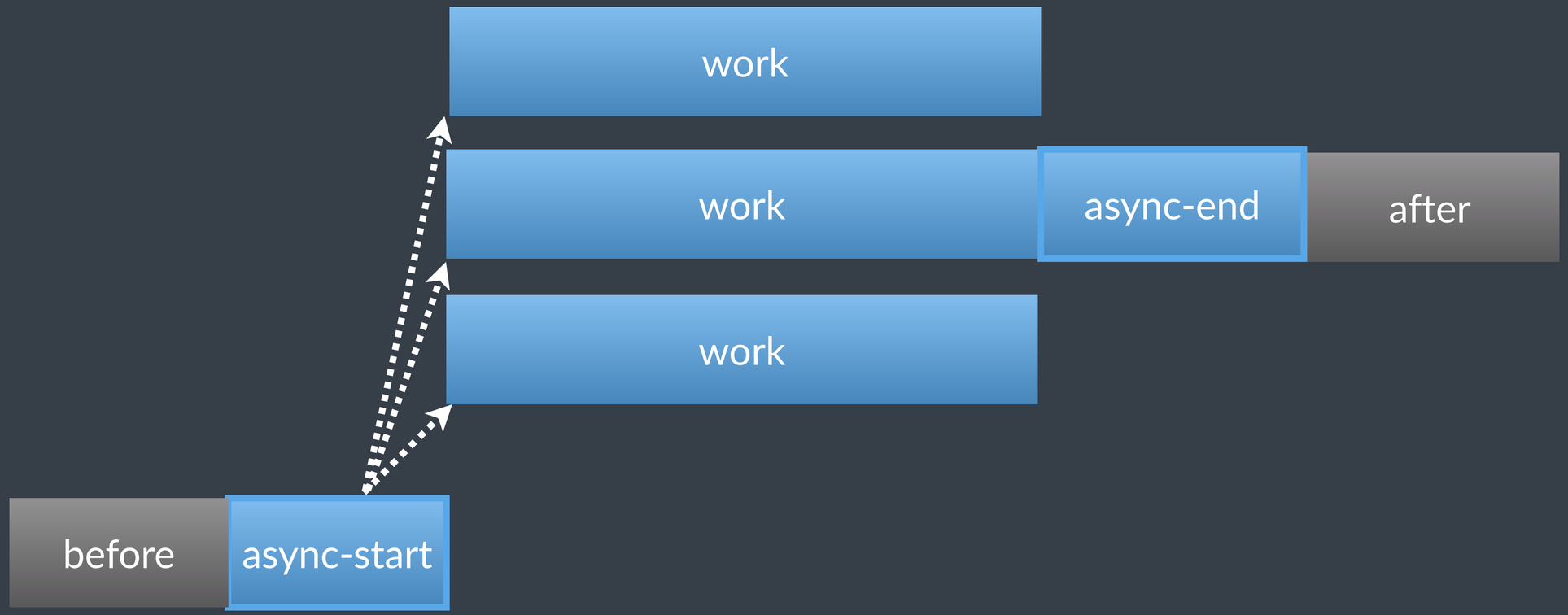
start thread != end threads

call completion != work completion

HW resources not bound to a thread


before
work
after


thread 2
work
async-end
after
thread 1
before
async-start


work
work
async-end
after
work
before
async-start

	sync	async
completion time	Immediate	deferred
completion place	same thread	different thread

synchronous execution

total ordering of work items

concurrent execution

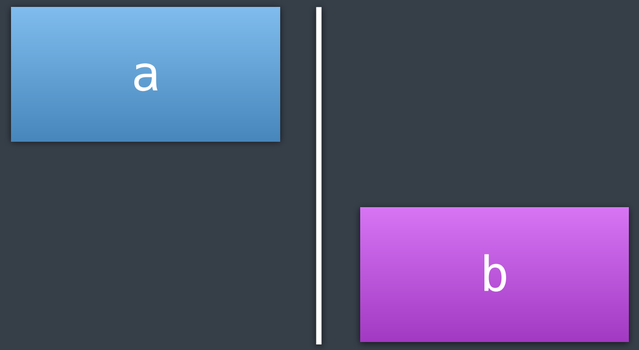
partial ordering of work items

at runtime

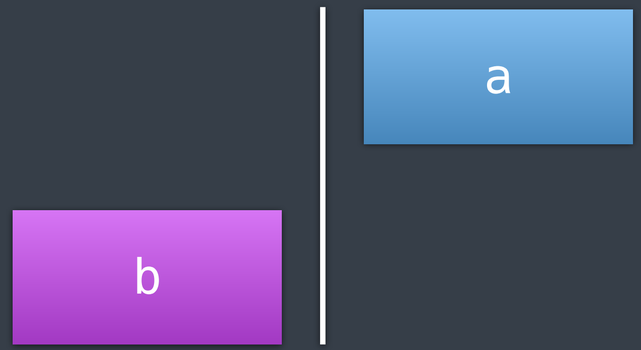
3 execution possibilities

$$a < b$$
$$b < a$$
$$\neg(a < b) \land \neg(b < a)$$

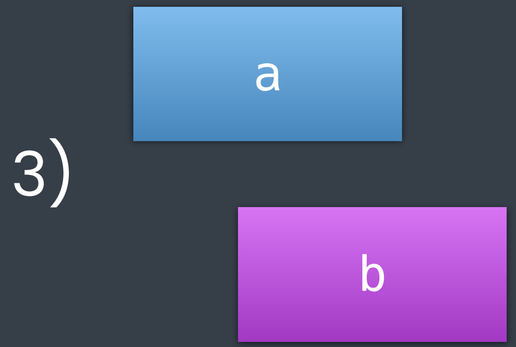
1)



2)



3)



concurrency
(design time)

expressing **constraints on execution**
ignoring actual execution

design time

basic concurrent constraints

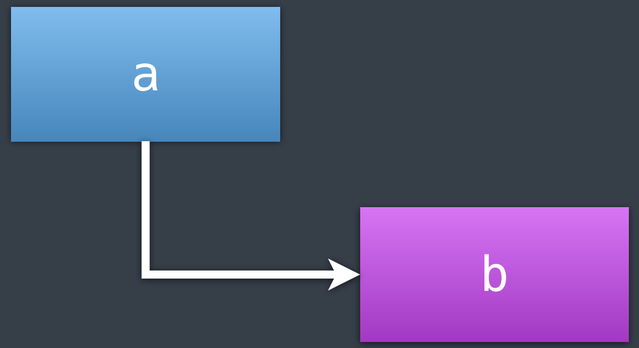
$$a < b$$

$$b < a$$

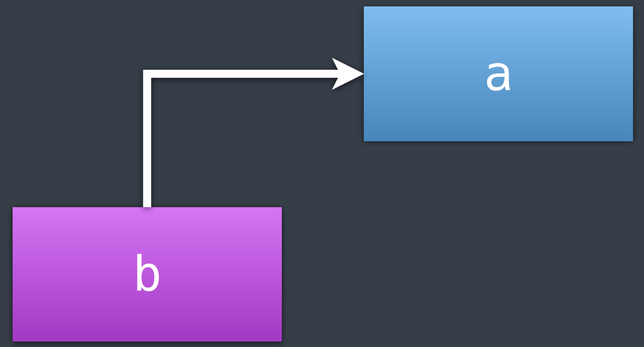
$$(a < b) \lor (b < a)$$ mutual exclusion

$$\neg(a < b) \land \neg(b < a)$$ concurrent execution

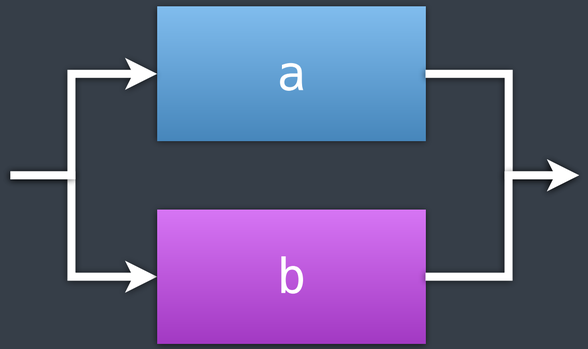
design time



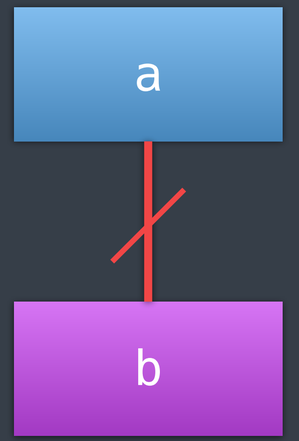
a < b



b < a



concurrent execution



mutual execution


focus on constraints

structured programming

focus on **control flow**

stop thinking about **gotos**

structured **concurrency**

focus on **relations between work items**

stop thinking about **threads**

structured concurrency

focus on **control flow**

stop thinking about **threads**

thread allocation

low level detail
concern for the framework

concurrency == asynchrony

	sync	async
completion time	Immediate	deferred
completion place	same thread	different thread
work ordering	total	partial
focus	control flow	control flow

Senders

2

senders

describe **asynchronous** work

compose well

are execute lazily

safe concurrency

Hello, world!

```
using namespace std::execution;
using std::this_thread::sync_wait;

sender auto computation
  = just("Hello, world!")
  | then([](std::string_view s) {
      std::print(s);
    });
sync_wait(std::move(computation));
```

```
std::function<void()> computation = []() {
  std::string_view s = "Hello, world!";

  std::print(s);
};
computation();
```

Hello, world!

```
sender auto computation
  = just("Hello, world!")
  | then([](std::string_view s) {
      std::print(s);
    });
sync_wait(           computation );
```

`just(…)`, `then(…)` return senders

```
just(X) | then(f) ==
  then(just(X), f)
```

just

```
just(X)
```

```
std::function<T()>{[]() {
  return X;
}}
```

then

```
snd | then(f)
```

```
std::function<T()>{[arg = …]() {
  return f(arg);
}}
```

sync_wait

```
sender auto snd = …
sync_wait(snd);
```

```
std::function<void()> f = …
f();
```

completion signals

how the work completes

completion signals

```
sender auto snd = …
```

success: `set_value(…)`
error: `set_error(…)`
stopped: `set_stopped()`

```
std::function<int()> f = …
```

success: an `int` value
error: an exception
stopped: terminate?

set_value()

multiple values in one signal

```
just(13, “prime”)
```

```
set_value(13, “prime”)
```

multiple types of success completion

```
when_any(just(13),
  just(“prime”))
```

```
set_value(13)
set_value(“prime”)
```

`set_error()`

not just exceptions

```
set_error(std::exception_ptr{…})
set_error(std::error_code{…})
set_error(“error”)
set_error(-1)
```

```
set_stopped()
```

(signals cancellation)
only one form

completion signatures

list of possible completion signals

completion signatures

```
completion_signatures<
  set_value_t(int),
  set_value_t(double, bool),
  set_error_t(std::exception_ptr),
  set_error_t(std::error_code),
  set_stopped_t()
>
```

senders

describe asynchronous work
compose well
are execute lazily
safe concurrency

Representing concurrency

3

concurrent work

```
scheduler auto sched = get_parallel_scheduler()
sender auto computation
  = schedule(sched)
  | then([] {
    std::print("Hello, from a different thread");
  });
sync_wait(computation);
```

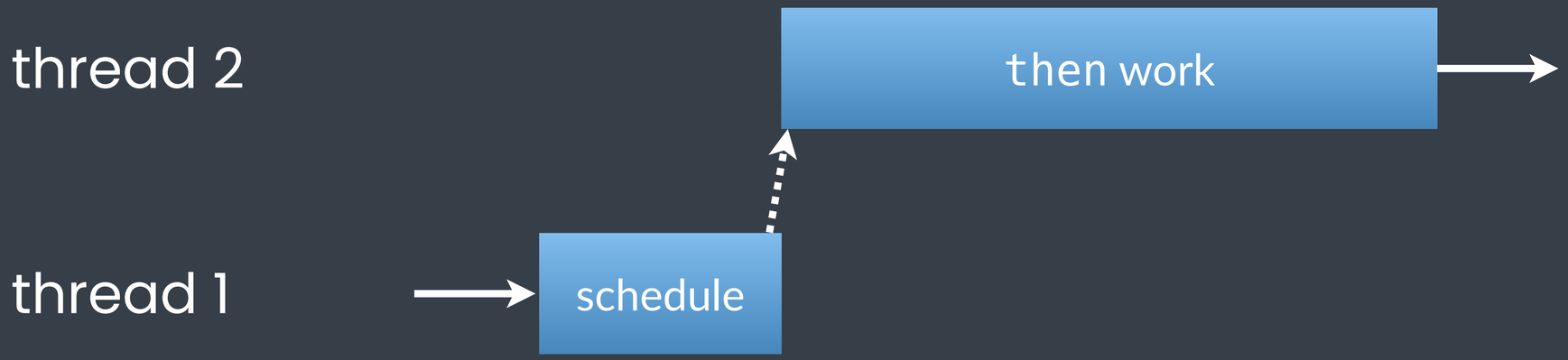
scheduler

a handler to a thread pool

where we can execute work

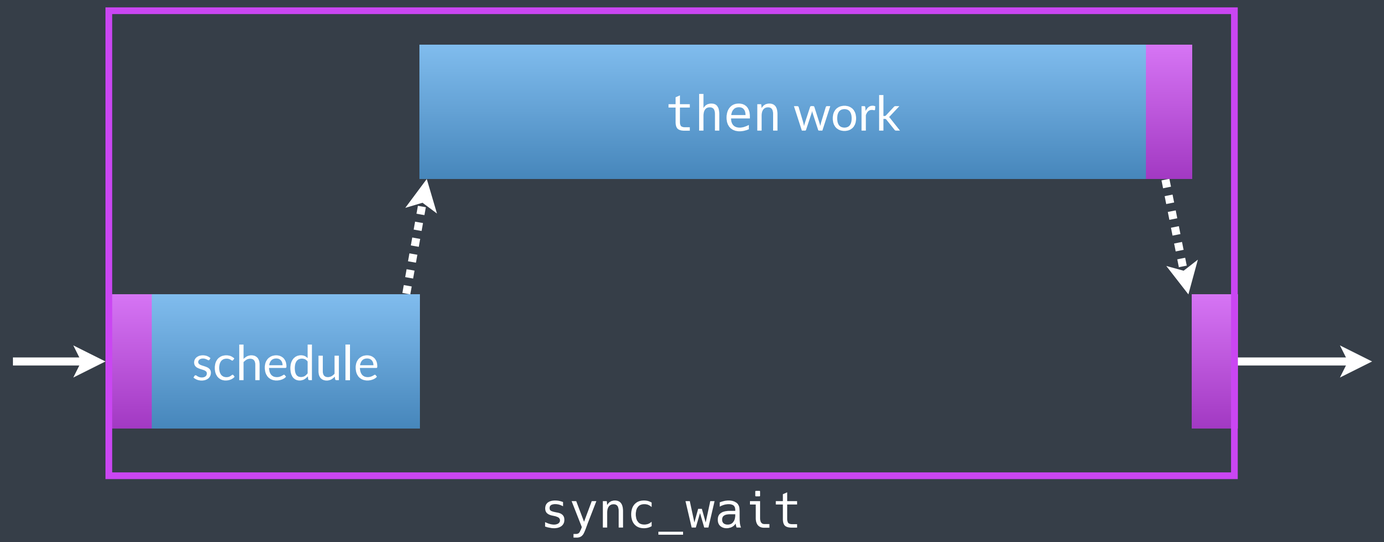
```
schedule(sched)
```

sends a success completion signal
from a thread belonging to `sched`


thread 2
then work
thread 1
schedule

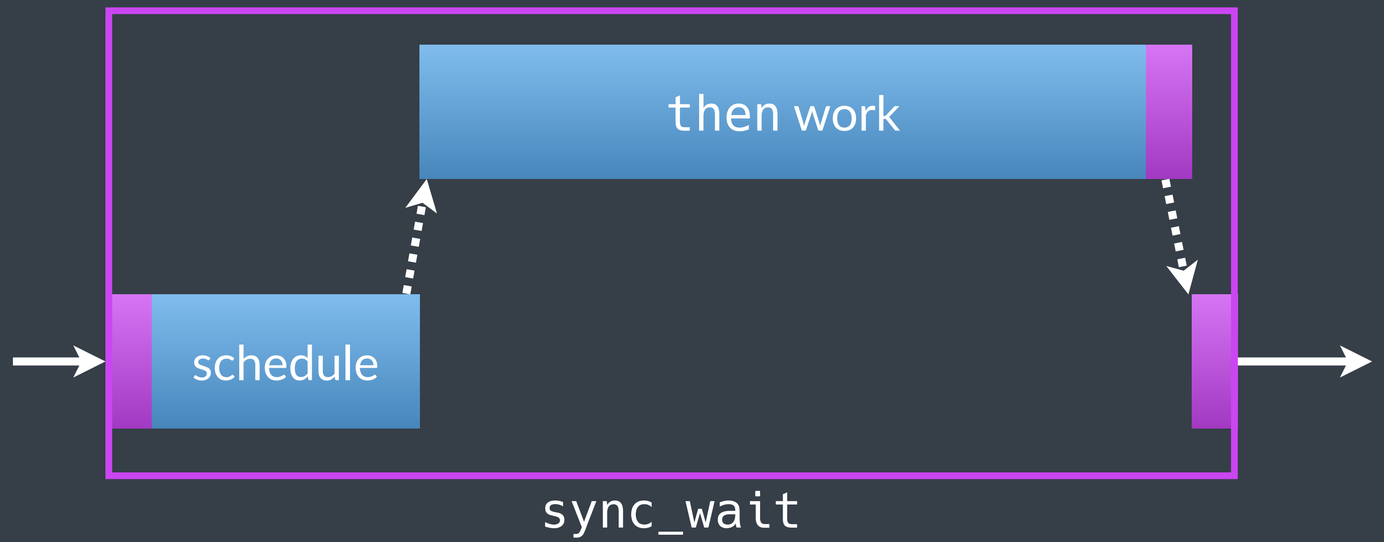
thread 2

thread 1



thread 2

thread 1



```
sync_wait(snd)
```

start executing the work described by `snd`
synchronously **wait** for the completion
returns the result

starts_on and continues_on

```
sender auto snd =
just() | then(f) | continues_on(sched) | then(g);

starts_on(sched2, snd)
```

switching between schedulers

```
sender auto read_data_snd                       // the work to read the data
  = just(connection, buffer)                    //   for a connection and a buffer
  | then(read_data);                            //   … call the function to read the data

sender auto all_work_snd                        // all the work
  = starts_on(io_sched, read_data_snd)          //   start the reading on I/O scheduler
  | continues_on(work_sched)                    //   continue on the work scheduler
  | then(process_data)                          //   process the received data
  | continues_on(io_sched)                      //   go back to the I/O scheduler
  | then(write_result);                         //   write the result back

sync_wait(all_work_snd);                        // execute all work
```

bulk execution

```
double some_value;
std::vector<double> x;
std::vector<double> y;

sender auto process_elements
  = just(some_value)
  | bulk(x.size(), [&](size_t i, double a) {
    y[i] = a * x[i] + y[i]
  });
```

senders

describe asynchronous work

compose well

are execute lazily

safe concurrency

Standard algorithms

4

algorithms in the base proposal

sender **factories**
sender **adaptors**
sender **consumers**

sender **factories**

```
just()
just_error()
just_stopped()
read_env()
schedule()
```

just*

```
just(vs…)
just_error(e)
just_stopped()
```

```
vs…
throw e                  // imperfect
std::terminate() // ???
```

`read_env(tag)`

<advanced>

queries the receiver for a property

```
read_env(get_scheduler)
read_env(get_stop_token)
```

```
schedule(sched)
```

completes on a thread from `sched`

sender **adaptors**

```
then(), upon_error(), upon_stopped(),
let_value(), let_error(), let_stopped(),
starts_on(), continues_on(), schedule_from(), on(),
bulk(),
split(), when_all(),
into_variant(),
stopped_as_optional(), stopped_as_error()
```

then()

```
prev | then(f)
```

```
f(prev())
```

upon_error()

```
prev | upon_error(f)
```

```
try {
  return prev();
} catch (...) {
  return f(std::current_exception());
}
```

upon_stopped()

```
prev | upon_stopped(f)
```

```
try {
  return prev();
} catch (was-stopped) {
  return f();
}
```

`let_value()`

monadic bind

value lifetime ≥ async operation

let_value()

```
prev | let_value(f)
```

```
auto val = prev();
{
  std::function<…> work=f(val);
  work();
}
```

then() and let_value()

```
sender_of<T> | then(function<U(T)>) → sender_of<U>
```

```
sender_of<T> | let_value(function<sender_of<U>(T)>) → sender_of<U>
```

parallel to optional

```
sender auto to_int(std::string_view sv);

sender auto initial_work = ...;
sender auto work = initial_work
  | let_value(to_int)
  | then([](int n) { return n + 1; })
  | then([](int n) { return
                     std::to_string(n); })
  | upon_error([](auto e) { return "NaN"s; });
```

```
std::optional<int> to_int(std::string_view sv);

std::optional<std::string> initial = ...;
auto result = initial
  .and_then(to_int)
  .transform([](int n) { return n + 1; })
  .transform([](int n) { return
                         std::to_string(n); })
  .value_or(“NaN"s);
```

`let_error()` and `let_stopped()`

similar to `upon_error()` and `upon_stopped()`

starts_on(sched, work)

starts `work` on the scheduler `sched`

starts_on(sched, work)

equivalent to

```
schedule(sched) | let_value([] { return work; })
```

```
prev | continues_on(sched)
```

`prev` then continue executing on `sched`

```
on(sched, snd)
```

```
starts_on(sched, snd) | continues_on(back)
```

```
schedule_from(sched, snd)
```

<advanced>

used for customizations

```
when_all(s1, s2, …)
```

completes when all given senders complete
returns the values from all senders

```
sender auto s1 = schedule(sched) | then(f) | then([]{ std::print("thread 1"); });
sender auto s2 = schedule(sched) | then(g) | then([]{ std::print("thread 2"); });
sync_wait(when_all(s1, s2));
```

```
prev | split()
```

allows `prev` to be used **more than once**,
while **executing it once**

split()

```
sender auto common = schedule(sch) | then([]{ std::print("once"); }) | split();
sender auto s1 = common | then([]{ std::print("thread 1"); });
sender auto s2 = common | then([]{ std::print("thread 2"); });
sync_wait(when_all(s1, s2));
```

```
prev | bulk(N, f)
```

calls `f(i, args…)` ∀ `i` ∈ `[0, N)`

(`args…` are the completion value of `prev`)

prev | into_variant()

multiple value completions → `std::variant`

prev | stopped_as_optional()

stopped completion → std::optional

```
prev | stopped_as_error(err)
```

stopped completion → error completion

sender **consumers**

```
sync_wait()
sync_wait_with_variant()
```

```
sync_wait(snd)
```

blocks the current thread until `snd` completes

→ `std::optional<std::tuple<values-sent-by(snd)>>`

`sync_wait_with_variant(snd)`

allows multiple value completions

senders

describe asynchronous work

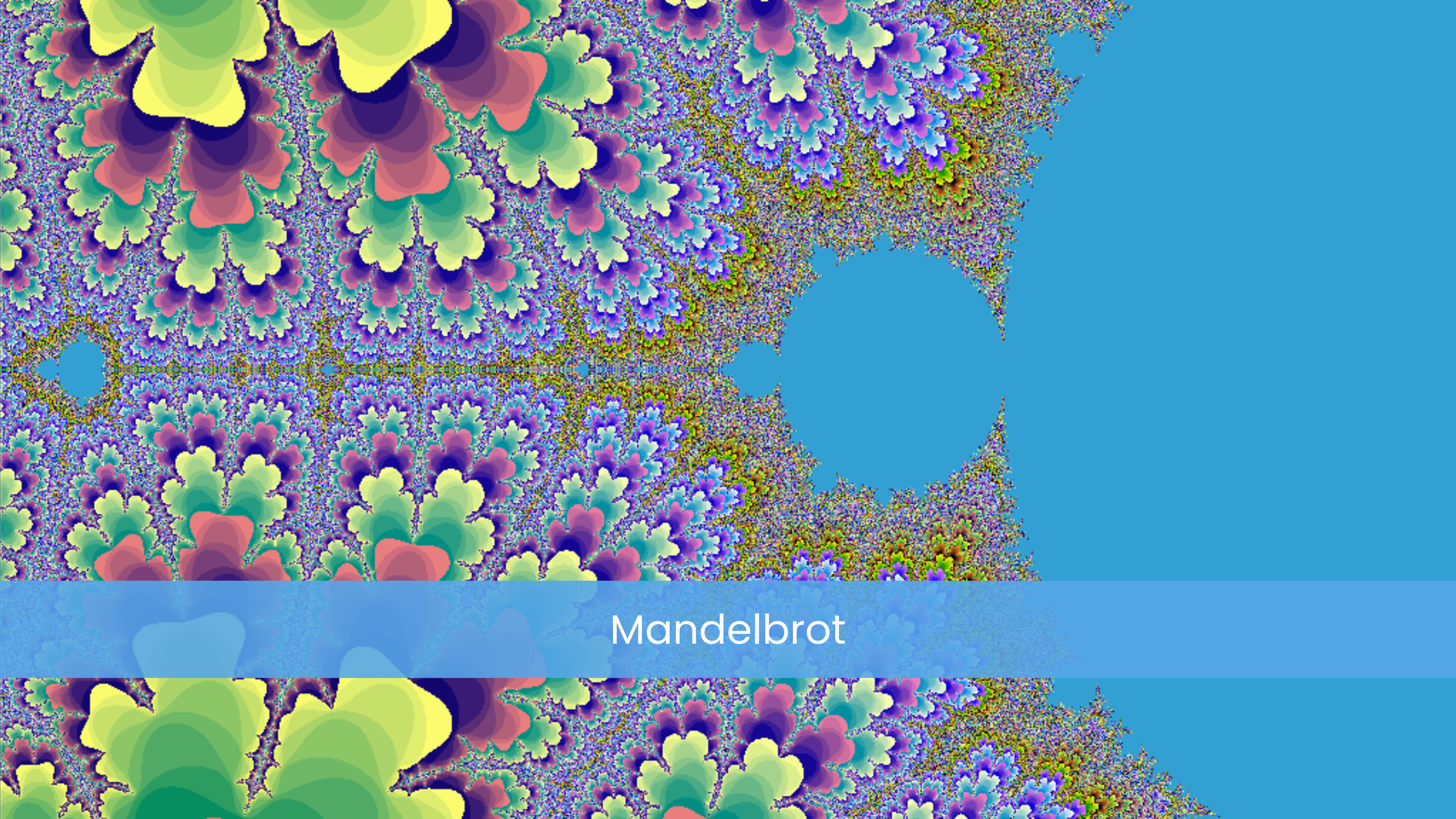
compose well

are execute lazily

safe concurrency

Examples

5


Mandelbrot

serial Mandelbrot

```
int mandelbrot_core(std::complex<double> c, int depth) {
  int count = 0;
  std::complex<double> z = 0;
  for (int i = 0; i < depth; i++) {
    if (abs(z) >= 2.0) break;
    z = z * z + c;
    count++;
  }
  return count;
}

template <typename F>
void serial_mandelbrot(int* vals, int max_x, int max_y, int depth, F&& transform) {
  for (int y = 0; y < max_y; y++) {
    for (int x = 0; x < max_x; x++) {
      vals[y * max_x + x] = mandelbrot_core(transform(x, y), depth);
    }
  }
}
```

concurrent Mandelbrot

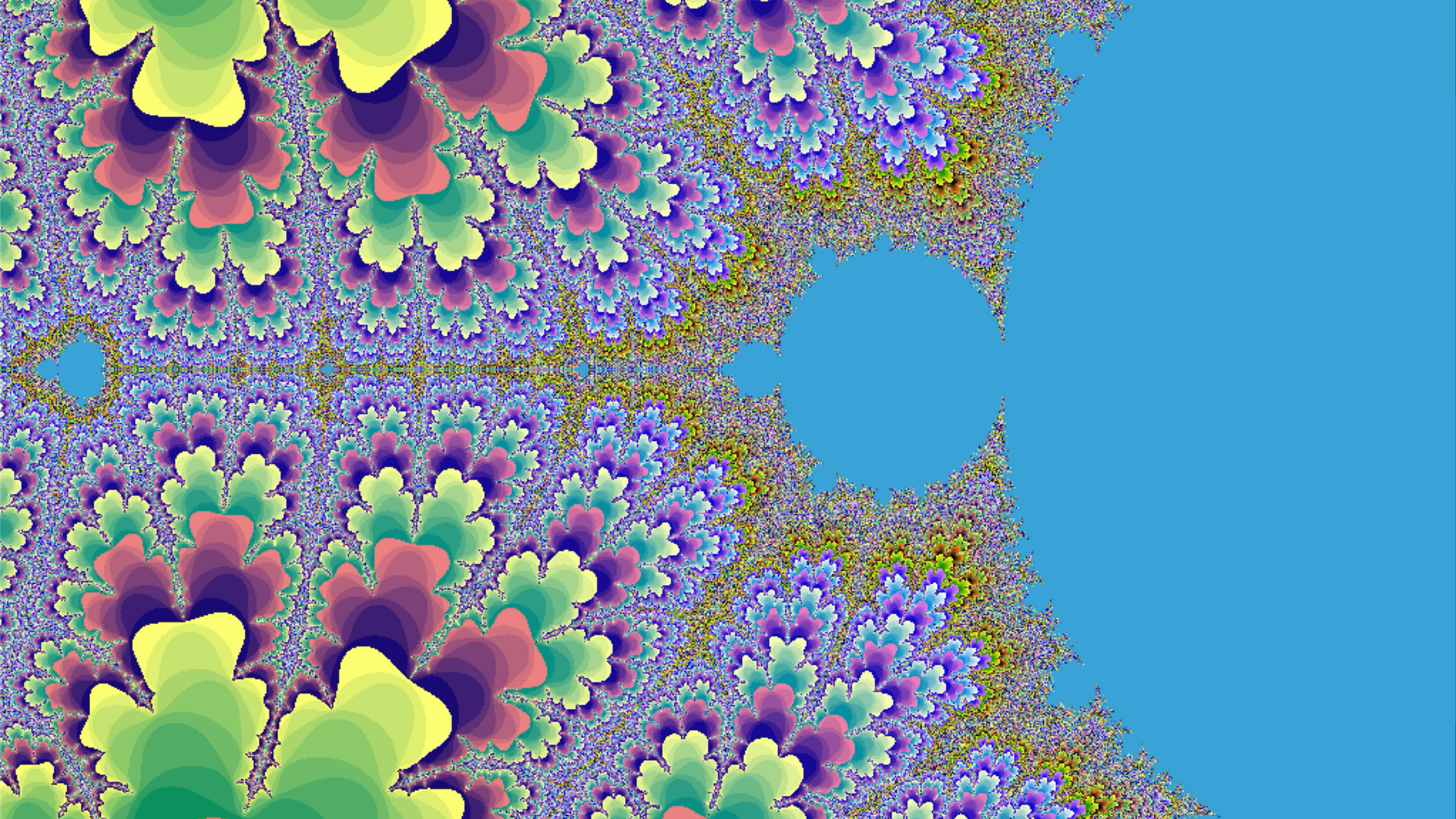
```
template <typename F>
void mandelbrot_concurrent(int* vals, int max_x, int max_y, int depth, F&& transform) {
  auto sched = get_parallel_scheduler();

  auto snd = schedule(sched)
           | bulk(max_y, [=](int y) {
               for (int x = 0; x < max_x; x++) {
                 vals[y * max_x + x] = mandelbrot_core(transform(x, y), depth);
               }
             });

  sync_wait(snd);
}
```

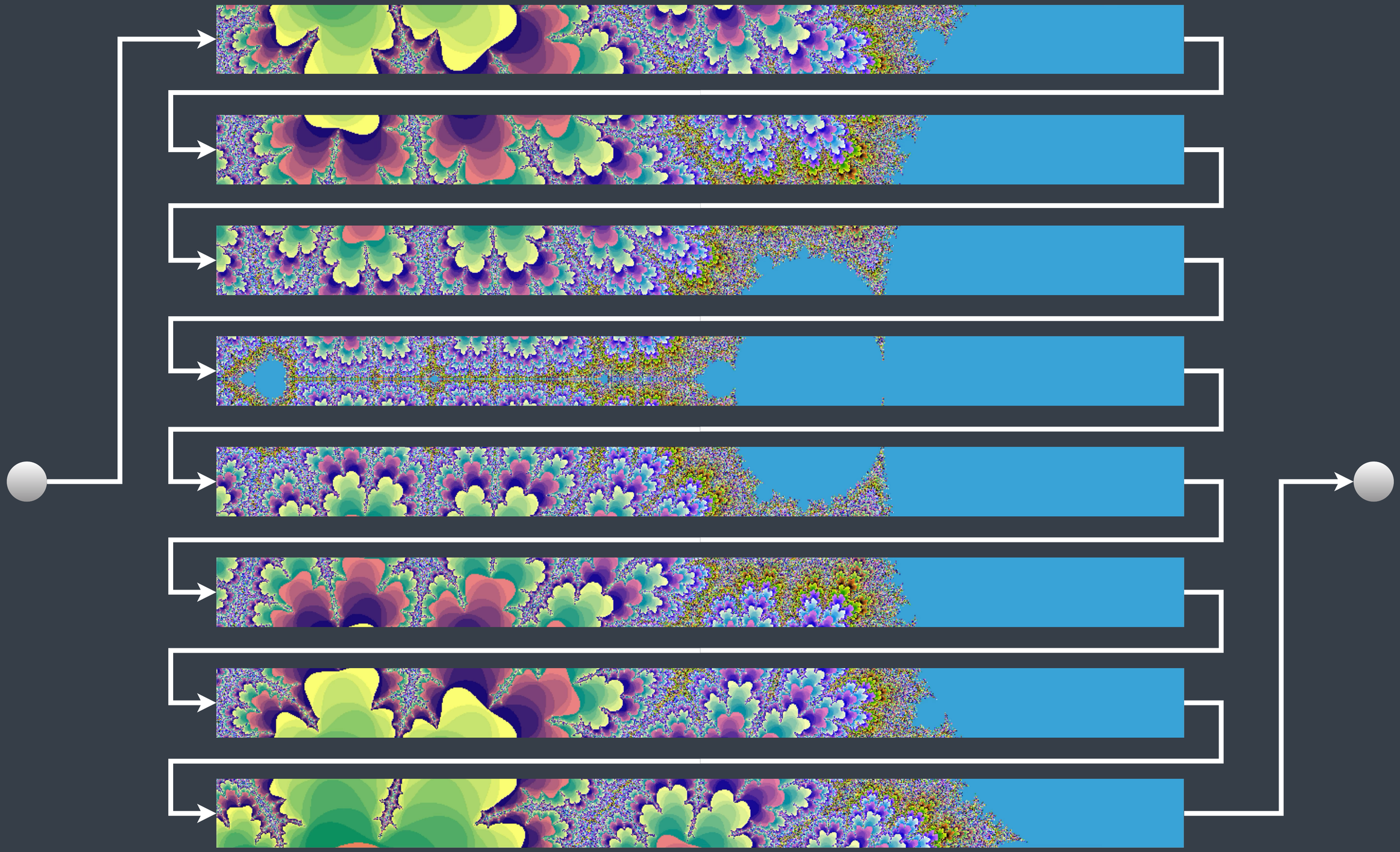
serial Mandelbrot

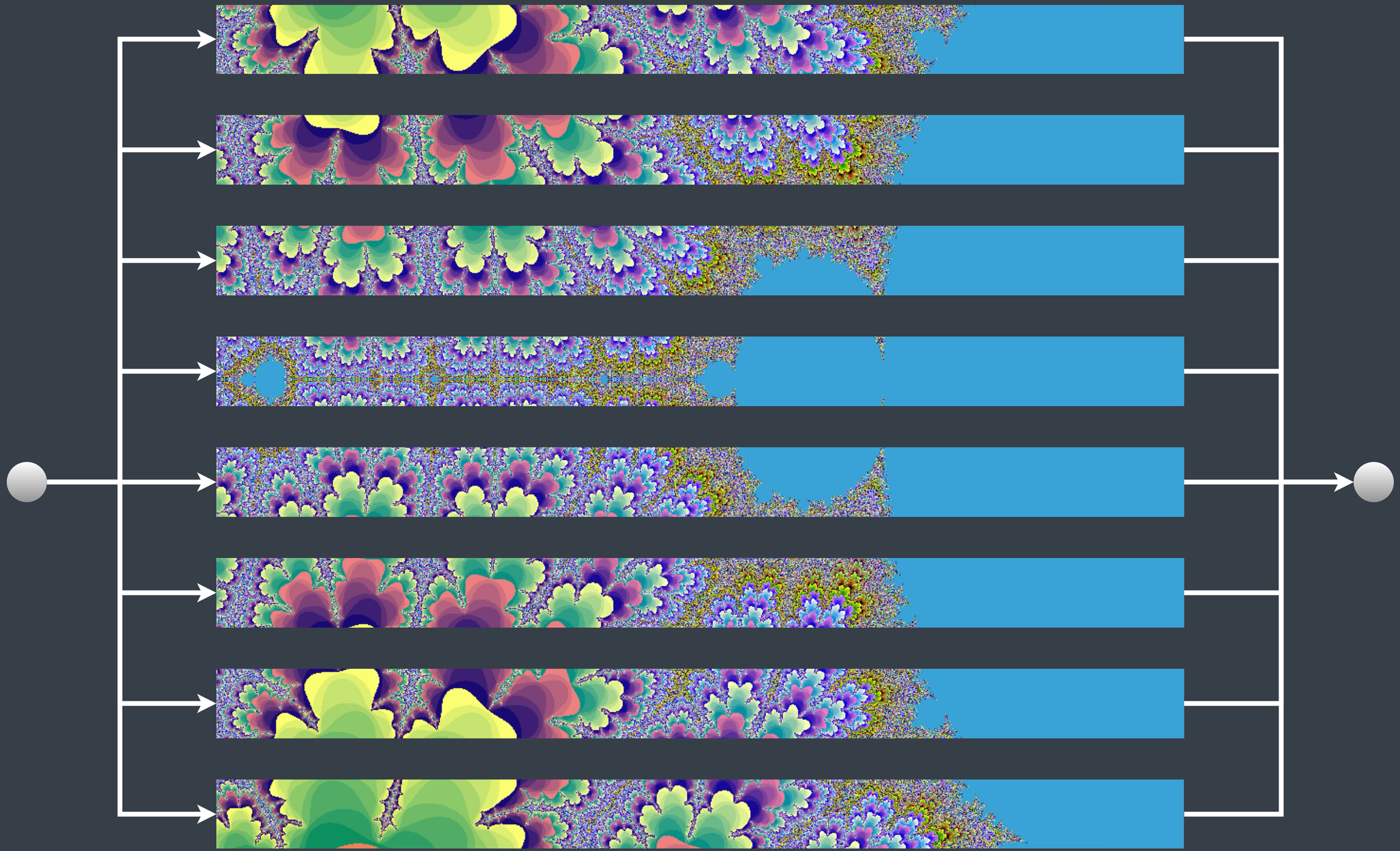
```
template <typename F>
void serial_mandelbrot(int* vals, int max_x, int max_y, int depth, F&& transform) {
  for (int y = 0; y < max_y; y++) {
    for (int x = 0; x < max_x; x++) {
      vals[y * max_x + x] = mandelbrot_core(transform(x, y), depth);
    }
  }
}
```

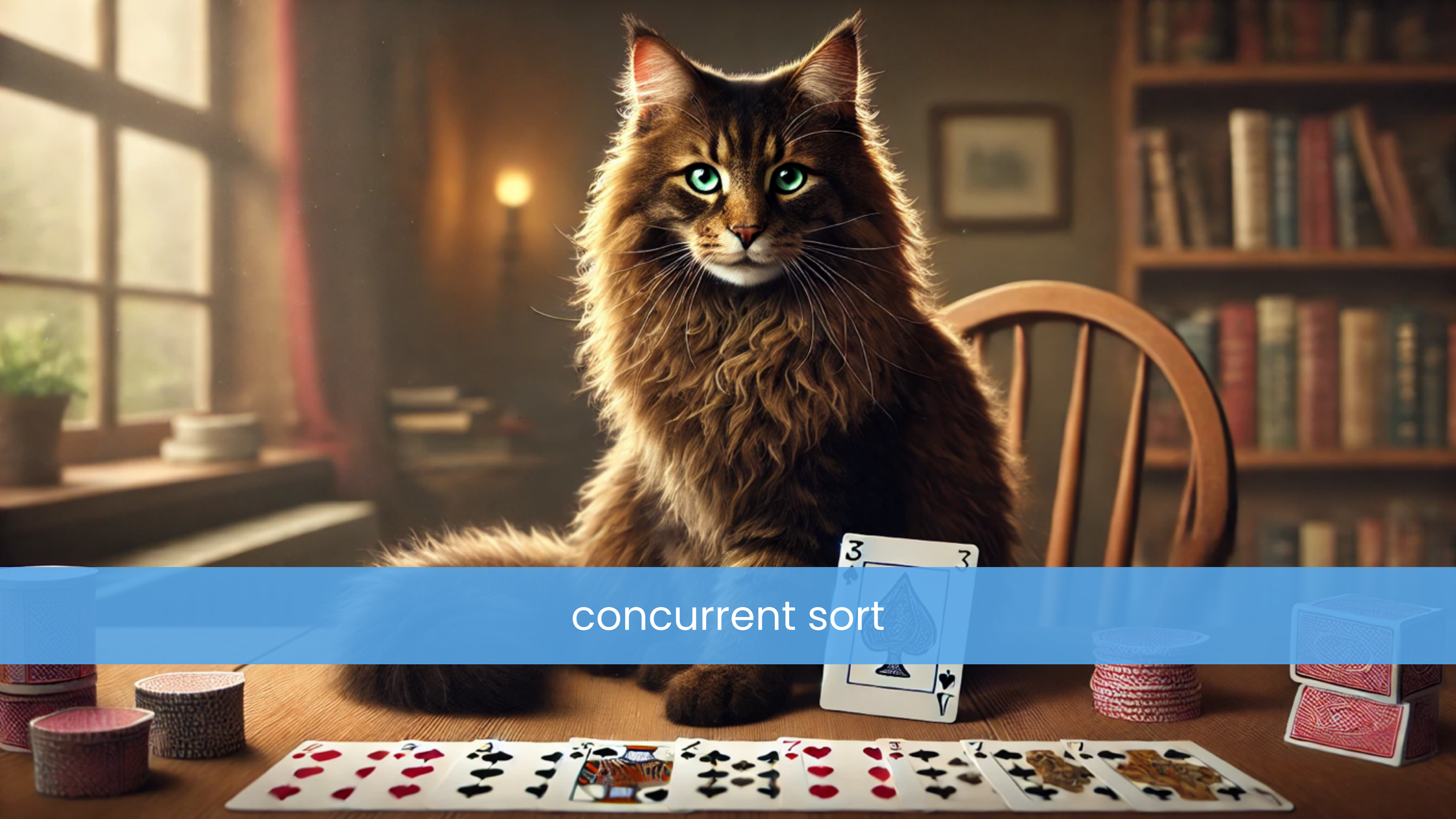
concurrent Mandelbrot

```
template <typename F>
void mandelbrot_concurrent(int* vals, int max_x, int max_y, int depth, F&& transform) {
  auto sched =get_parallel_scheduler();

  auto snd = schedule(sched)
           | bulk(max_y, [=](int y) {
               for (int x = 0; x < max_x; x++) {
                 vals[y * max_x + x] = mandelbrot_core(transform(x, y), depth);
               }
             });

  sync_wait(snd);
}
```





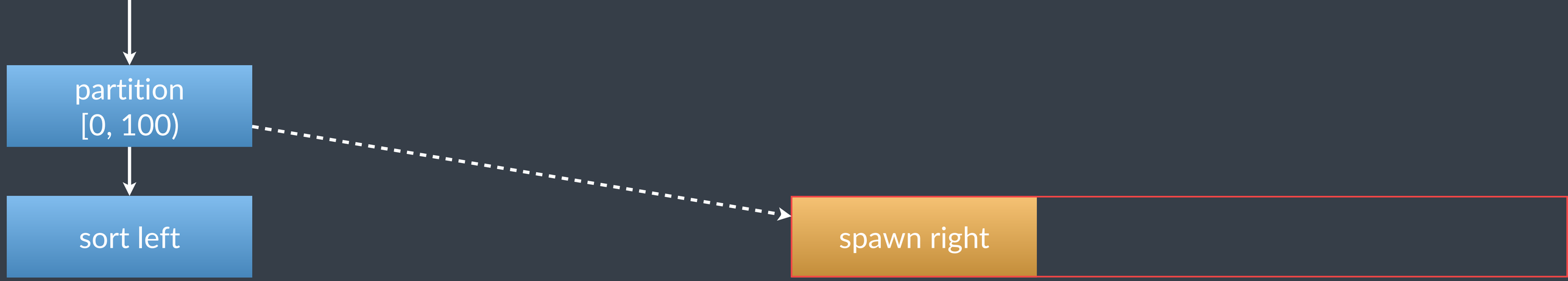


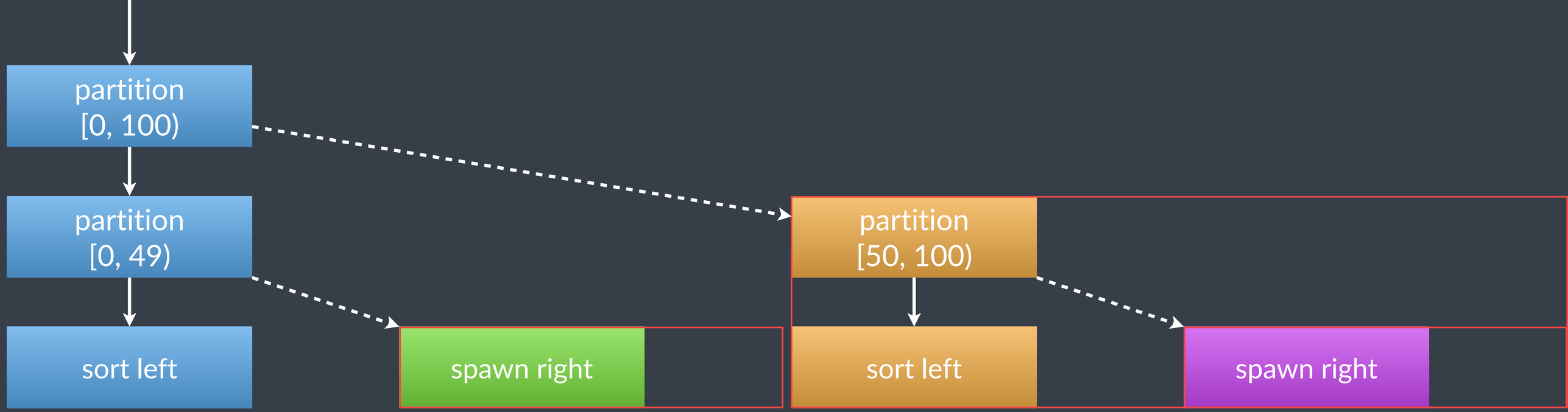


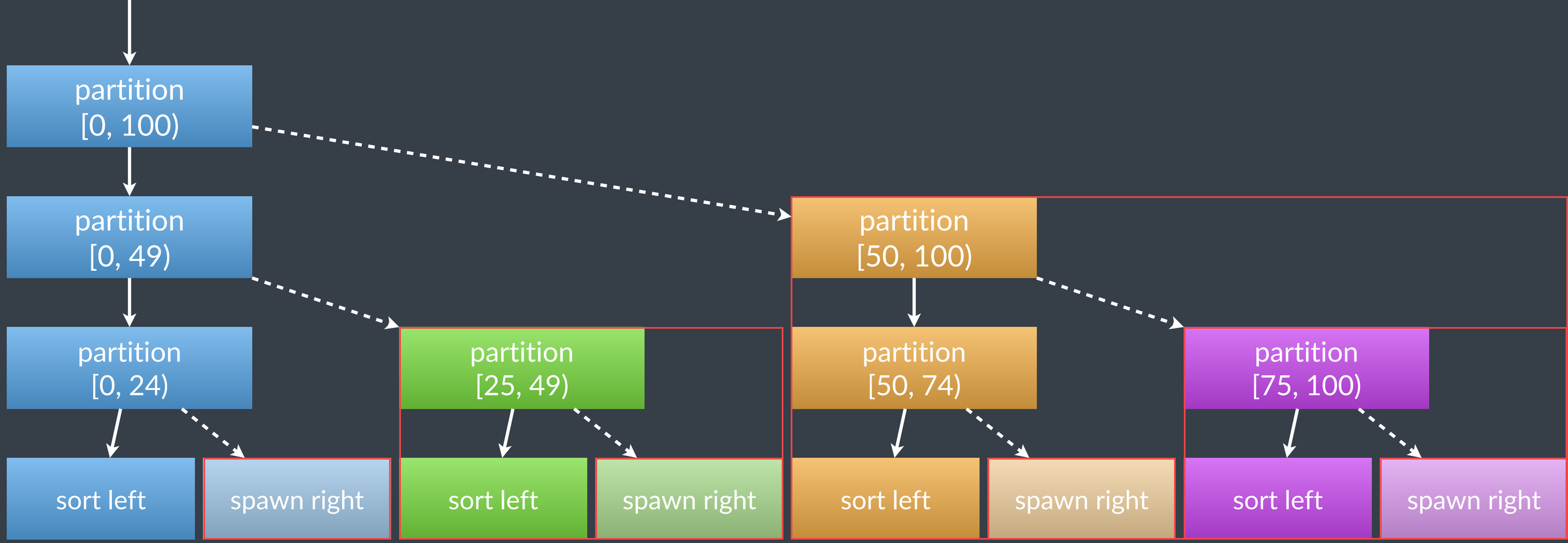
concurrent sort

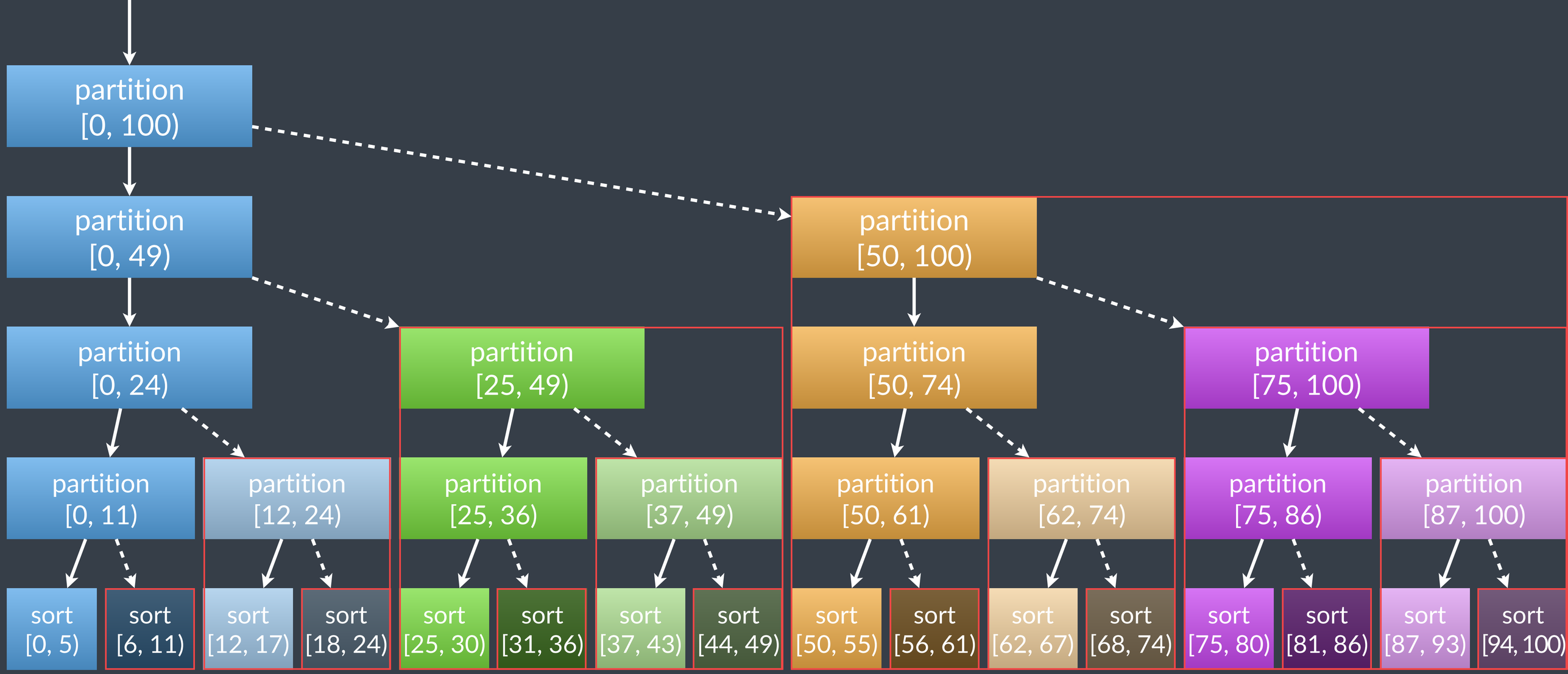
```cpp
template <std::random_access_iterator It>
void concurrent_sort(It first, It last) {
  counting_scope scope;
  concurrent_sort_impl(first, last, scope);
  sync_wait(scope.on_empty());
}

template <std::random_access_iterator It>
void concurrent_sort_impl(It first, It last, async_scope& scope) {
  auto size = std::distance(first, last);
  if (size_t(size) < size_threshold) {
    std::sort(first, last);
  } else {
    auto p = sort_partition(first, last);
    auto mid1 = p.first;
    auto mid2 = p.second;

    sender auto snd = schedule(get_parallel_scheduler())
         | then([=, &scope] { concurrent_sort_impl(mid2, last, scope); });

    scope.spawn(snd);
    concurrent_sort_impl(first, mid1, scope);
  }
}
```

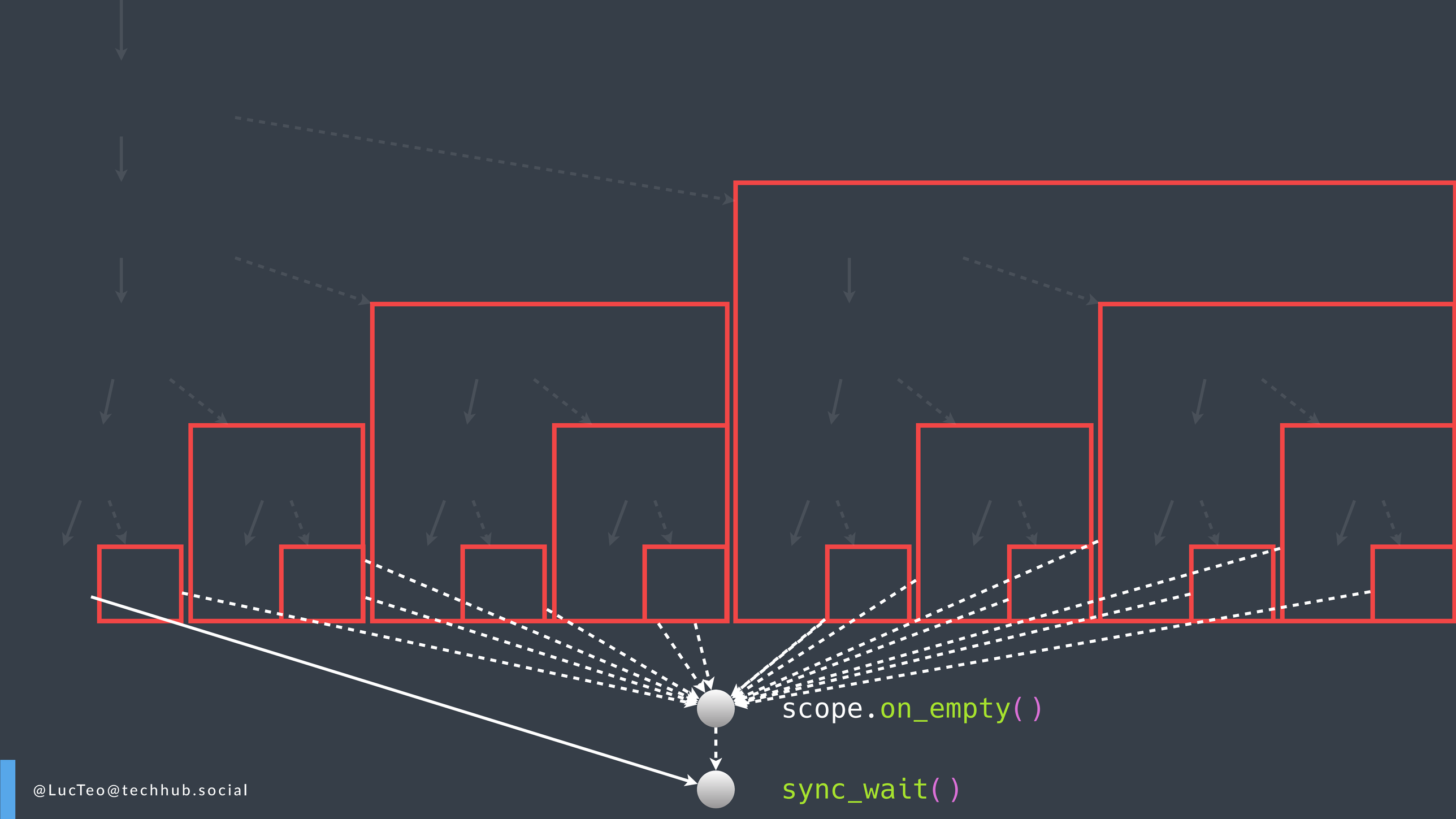
```cpp
template <std::random_access_iterator It>
void concurrent_sort(It first, It last) {
  counting_scope scope;
  concurrent_sort_impl(first, last, scope);
  sync_wait(scope.on_empty());
}

template <std::random_access_iterator It>
void concurrent_sort_impl(It first, It last, async_scope& scope) {
  auto size = std::distance(first, last);
  if (size_t(size) < size_threshold) {
    std::sort(first, last);
  } else {
    auto p = sort_partition(first, last);
    auto mid1 = p.first;
    auto mid2 = p.second;

    sender auto snd = schedule(get_parallel_scheduler())
        | then([=, &scope] { concurrent_sort_impl(mid2, last, scope); });

    scope.spawn(snd);
    concurrent_sort_impl(first, mid1, scope);
  }
}
```

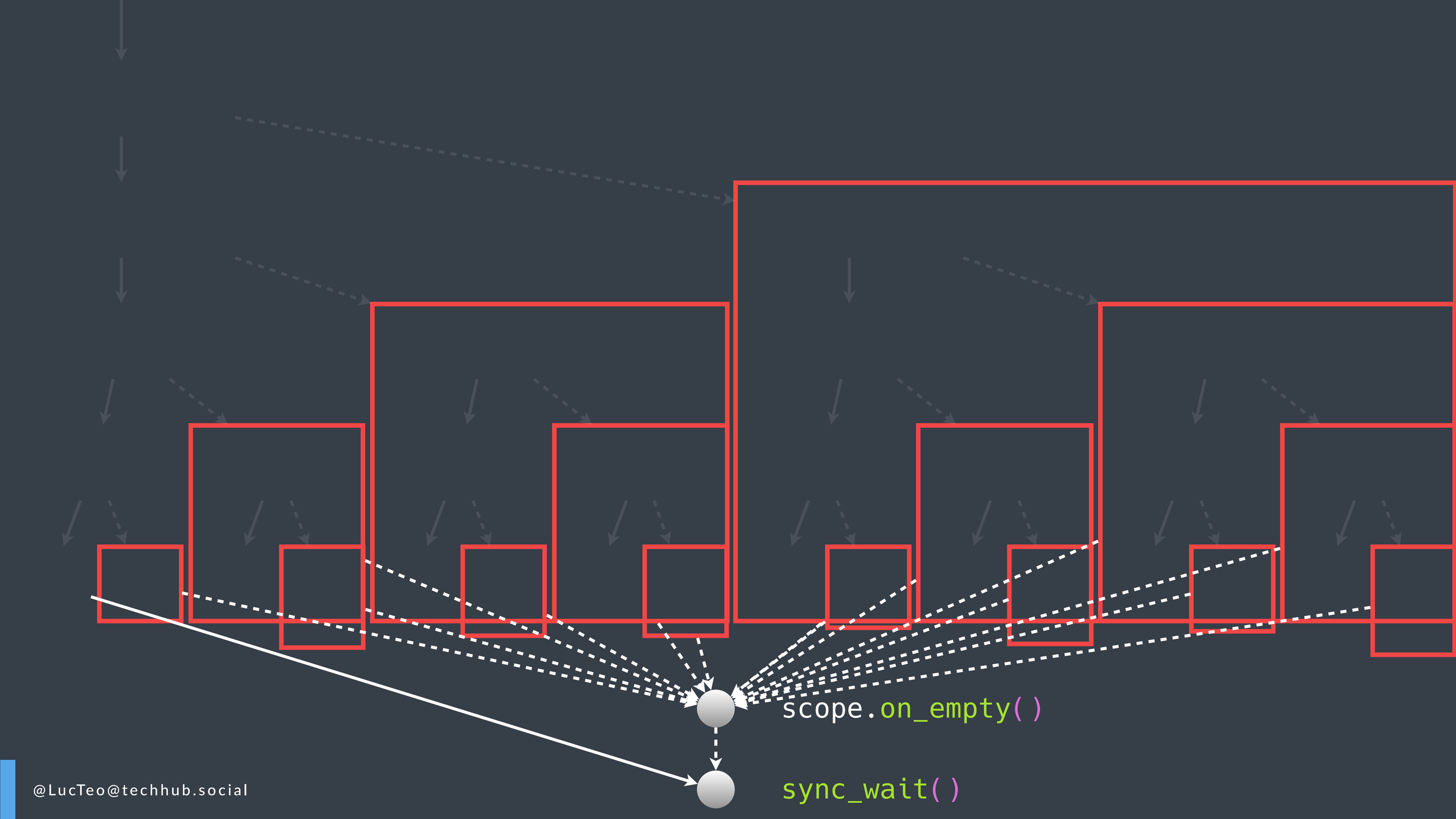

partition
[0, 100)
sort left
spawn right


partition
[0, 100)
partition
[0, 49)
sort left
spawn right
partition
[50, 100)
sort left
spawn right


partition
[0, 100)
partition
[0, 49)
partition
[0, 24)
sort left
spawn right
partition
[25, 49)
sort left
spawn right
partition
[50, 100)
partition
[50, 74)
sort left
spawn right
partition
[75, 100)
sort left
spawn right


partition
[0, 100)
partition
[0, 49)
partition
[0, 24)
partition
[0, 11)
partition
[12, 24)
partition
[25, 49)
partition
[25, 36)
partition
[37, 49)
partition
[50, 100)
partition
[50, 74)
partition
[50, 61)
partition
[62, 74)
partition
[75, 100)
partition
[75, 86)
partition
[87, 100)
sort
[0, 5)
sort
[6, 11)
sort
[12, 17)
sort
[18, 24)
sort
[25, 30)
sort
[31, 36)
sort
[37, 43)
sort
[44, 49)
sort
[50, 55)
sort
[56, 61)
sort
[62, 67)
sort
[68, 74)
sort
[75, 80)
sort
[81, 86)
sort
[87, 93)
sort
[94, 100)


scope.on_empty()
sync_wait()


scope.on_empty()
sync_wait()

```cpp
template <std::random_access_iterator It>
void concurrent_sort(It first, It last) {
  counting_scope scope;
  concurrent_sort_impl(first, last, scope);
  sync_wait(scope.on_empty());
}

template <std::random_access_iterator It>
void concurrent_sort_impl(It first, It last, async_scope& scope) {
  auto size = std::distance(first, last);
  if (size_t(size) < size_threshold) {
    std::sort(first, last);
  } else {
    auto p = sort_partition(first, last);
    auto mid1 = p.first;
    auto mid2 = p.second;

    sender auto snd = schedule(get_parallel_scheduler())
         | then([=, &scope] { concurrent_sort_impl(mid2, last, scope); });

    scope.spawn(snd);
    concurrent_sort_impl(first, mid1, scope);
  }
}
```

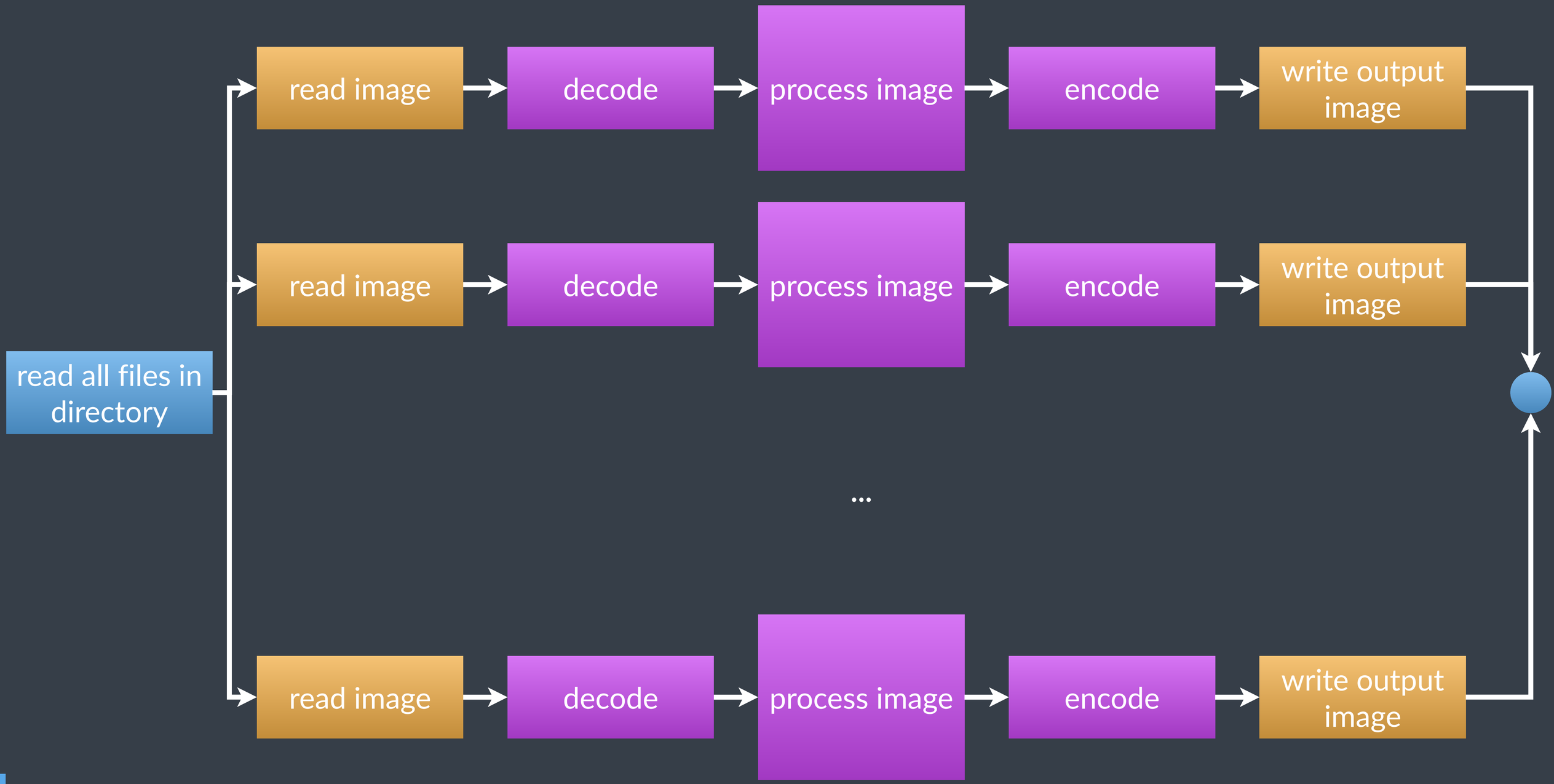
image processing

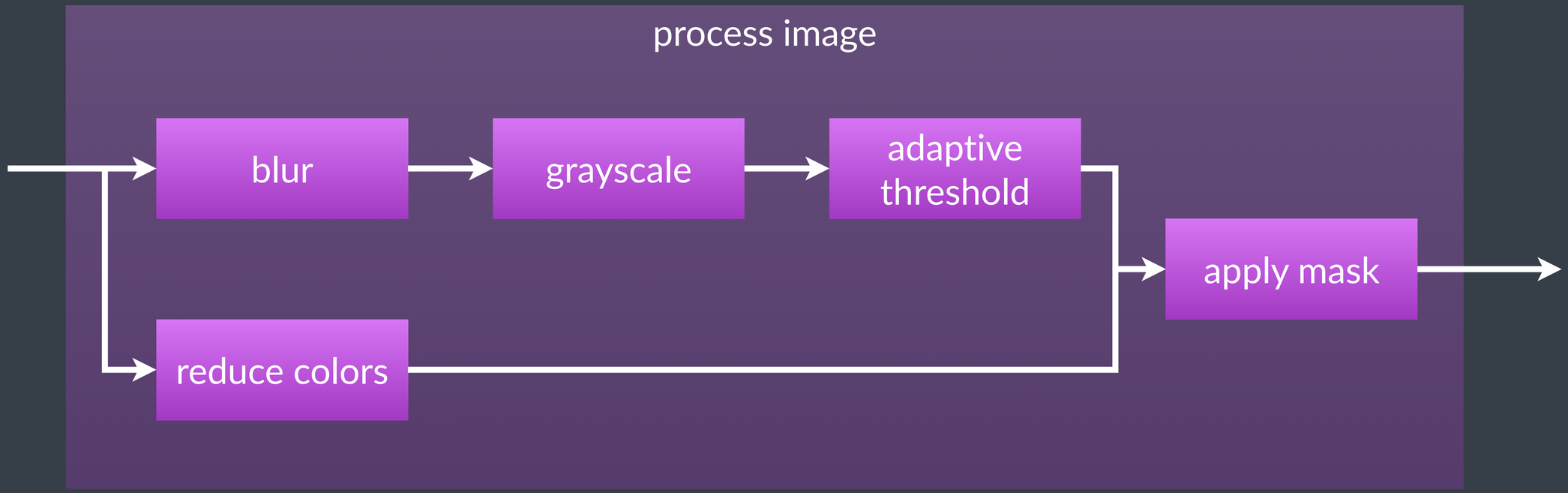
application

reads all the images in an input folder
apply a filter
write them to an output folder

concurrency

I/O happens on a dedicated thread
different images can be processed concurrently
one image, more than 1 thread


read all files in directory
read image
decode
process image
encode
write output image
read image
decode
process image
encode
write output image
...
read image
decode
process image
encode
write output image


process image
blur
grayscale
adaptive threshold
apply mask
reduce colors

```cpp
int main(int argc, char** argv) {

  sender auto everything = process_files("data", "out", blur_size, num_colors, block_size, diff);

  auto [processed] = sync_wait(everything).value();

  printf("Processed images: %d\n", processed);
  return 0;
}
```

```
task<int> process_files(const char* in_folder_name, const char* out_folder_name,
                        int blur_size, int num_colors, int block_size, int diff) {
  counting_scope scope;
  static_thread_pool io_pool(1);
  auto io_sched = io_pool.get_scheduler();
  auto cpu_sched = get_parallel_scheduler();

  int processed = 0;
  for (const auto& entry : fs::directory_iterator(in_folder_name)) {
    auto extension = entry.path().extension();
    if (!entry.is_regular_file() || (extension != ".jpg") && (extension != ".jpeg"))
      continue;
    auto in_filename = entry.path().string();
    auto out_filename = (fs::path(out_folder_name) / entry.path().filename()).string();
    printf("Processing %s\n", in_filename.c_str());

    sender auto work = …

    scope.spawn(work);
  }
  co_await scope.on_empty();
  co_return processed;
}
```

senders & coroutines

coroutines can **behave as senders**

coroutines can `co_await` senders

*) with small changes to coroutine type

`std::task`

in the process of standardization
coroutine designed for asynchronous work
integrates well with senders
~ a type-erased sender

```
task<int> process_files(const char* in_folder_name, const char* out_folder_name,
                        int blur_size, int num_colors, int block_size, int diff) {
  counting_scope scope;
  static_thread_pool io_pool(1);
  auto io_sched = io_pool.get_scheduler();
  auto cpu_sched = get_parallel_scheduler();

  int processed = 0;
  for (const auto& entry : fs::directory_iterator(in_folder_name)) {
    auto extension = entry.path().extension();
    if (!entry.is_regular_file() || (extension != ".jpg") && (extension != ".jpeg"))
      continue;
    auto in_filename = entry.path().string();
    auto out_filename = (fs::path(out_folder_name) / entry.path().filename()).string();
    printf("Processing %s\n", in_filename.c_str());

    sender auto work = …

    scope.spawn(work);
  }
  co_await scope.on_empty();
  co_return processed;
}
```

use `std::task` when

needing a type-erased sender
logic is complex

```
sender auto file_content = // read file on I/O scheduler
    co_await (schedule(io_sched) | then([=] { return read_file(entry); }));

sender auto work =
    transfer_just(cpu_sched, cv::_InputArray::rawIn(file_content)) // -> CPU sched
    | then([=](cv::InputArray file_content) -> cv::Mat {           // decode the image
        return cv::imdecode(file_content, cv::IMREAD_COLOR);
      })
    | let_value([=](const cv::Mat& img) {                          // apply the filter
        return tr_cartoonify(img, blur_size, num_colors, block_size, diff);
      })
    | then([=](const cv::Mat& img) {                               // encode back the image
        std::vector<unsigned char> out_image_content;
        if (!cv::imencode(extension, img, out_image_content)) {
          throw std::runtime_error("cannot encode image");
        }
        return out_image_content;
      })
    | continues_on(io_sched)                                       // -> I/O sched
    | then([=](const std::vector<unsigned char>& bytes) {          // write file
        write_file(out_filename.c_str(), bytes);
      })
    | then([=] { printf("Written %s\n", out_filename.c_str()); })
    | then([&] { processed++; });
```

```
auto tr_cartoonify(const cv::Mat& src, int blur_size, int num_colors, int block_size, int diff)
{
  auto sched = get_parallel_scheduler();
  sender auto snd =
      when_all(
          just(src)
              | continues_on(sched)
              | then([=](const cv::Mat& src) {
                  auto blurred = tr_blur(src, blur_size);
                  auto gray = tr_to_grayscale(blurred);
                  auto edges = tr_adaptthresh(gray, block_size, diff);
                  return edges;
                }),
          just(src)
             | continues_on(sched)
             | then([=](const cv::Mat& src) {
                  return tr_reducecolors(src, num_colors);
                })
          )
      | then([](const cv::Mat& edges, const cv::Mat& reduced_colors) {
          return tr_apply_mask(reduced_colors, edges);
        });
  return snd;
}
```

Beyond senders

6

receivers

```
struct my_receiver {
  // This is a receiver type.
  using receiver_concept = receiver_t;

  void set_value(int value) noexcept {
    printf("Received value: %d\n", value);
  }
  void set_error(std::exception_ptr e) noexcept {
    printf("Received error\n");
  }
  void set_stopped() noexcept {
    printf("Received stopped signal\n");
  }
};

sender auto snd = just(13);
auto op = connect(snd, my_receiver{});
start(op);
```

receivers

receives completion signals
part of sender adaptors & consumers
implementation detail

operation states

the actual **async operation**

writing a simple sender

```
auto just_int(int x) { return just_int_sender{x}; }
```

writing a simple sender

```cpp
struct just_int_sender {
  // The data of the sender.
  int value_to_send_;

  // This is a sender type.
  using sender_concept = sender_t;

  // This sender always complete with an `int` value.
  using completion_signatures = completion_signatures<set_value_t(int)>;

  // No environment to provide.
  empty_env get_env() const noexcept { return {}; }

  // Connect to the given receiver, and produce an operation state.
  template <receiver Receiver>
  auto connect(Receiver receiver) noexcept {
    return detail::just_int_op{value_to_send_, receiver};
  }
};
```

writing a simple sender

```cpp
namespace detail {

template <receiver Receiver>
struct just_int_op {
  int value_to_send_;
  Receiver receiver_;

  // This is an operation-state type.
  using operation_state_concept = operation_state_t;

  // The actual work of the operation state.
  void start() noexcept {
    // No real work, just send the value to the receiver.
    set_value(std::move(receiver_), value_to_send_);
  }
};

} // namespace detail
```

a then sender

```
template <ex::sender Previous, std::invocable<int> Fun>
then_sender<Previous, Fun> then(Previous prev, Fun f) {
  return {prev, f};
}
```

a then sender

```
template <sender Previous, std::invocable<int> Fun>
struct then_sender {
  Previous previous_;
  Fun f_;

  // This is a sender type.
  using sender_concept = sender_t;

  // This sender always complete with an `int` value.
  using completion_signatures = completion_signatures<
      set_value_t(int),
      set_error_t(std::exception_ptr),
      set_stopped_t()>;

  // No environment to provide.
  empty_env get_env() const noexcept { return {}; }

  // Connect to the given receiver, and produce an operation state.
  template <receiver Receiver>
  auto connect(Receiver receiver) noexcept {
    return connect(previous_, detail::then_receiver{f_, receiver});
  }
};
```

a then sender

```
namespace detail {

template <receiver Receiver, typename Fun>
struct then_receiver {
  Fun f_;
  Receiver receiver_;

  // This is a receiver type.
  using receiver_concept = receiver_t;

  // Called when the previous sender completes with a value.
  void set_value(int value) noexcept {
    try {
      ex::set_value(std::move(receiver_), f_(value));
    } catch (...) {
      ex::set_error(std::move(receiver_), std::current_exception());
    }
  }
  void set_error(std::exception_ptr e) noexcept { ex::set_error(std::move(receiver_), e); }
  void set_stopped() noexcept { ex::set_stopped(std::move(receiver_)); }
};

}
```

environments

store **properties** for receivers, senders

`read_env(tag)`

<advanced>

queries the receiver for a property

```
read_env(get_scheduler)
read_env(get_stop_token)
```

properties

allocator
stop token
scheduler
delegation scheduler
completion scheduler
forward progress guarantee

receiver environments

allows information flow from **right to left**

cancellation

left to right: `set_stopped(recv)`

right to left: `get_stopped_token(recv)`

left to right cancellation

our predecessor has been canceled
we received `set_stopped()`

=> we call `set_stopped()` too

right to left cancellation

caller (receiver) tells us that it doesn't need our results
its stopped token is signaled
we check that stop token

=> we call don't do work and call `set_stopped()`

all algorithms can be **customised**

e.g., **more efficient** implementations for certain schedulers

advanced concerns

writing generic sender algorithms
metaprogramming required for completion signatures
provide good diagnosis

Conclusions

7

covered

sync vs async
elements of thinking concurrently
senders & completion signals
standard algorithms
examples
S/R add-ons: parallel scheduler, `counting_scope`, `task`

	sync	async
completion time	Immediate	deferred
completion place	same thread	different thread
work ordering	total	partial
focus	control flow	control flow

structured concurrency

focus on **control flow**

stop thinking about **threads**

senders

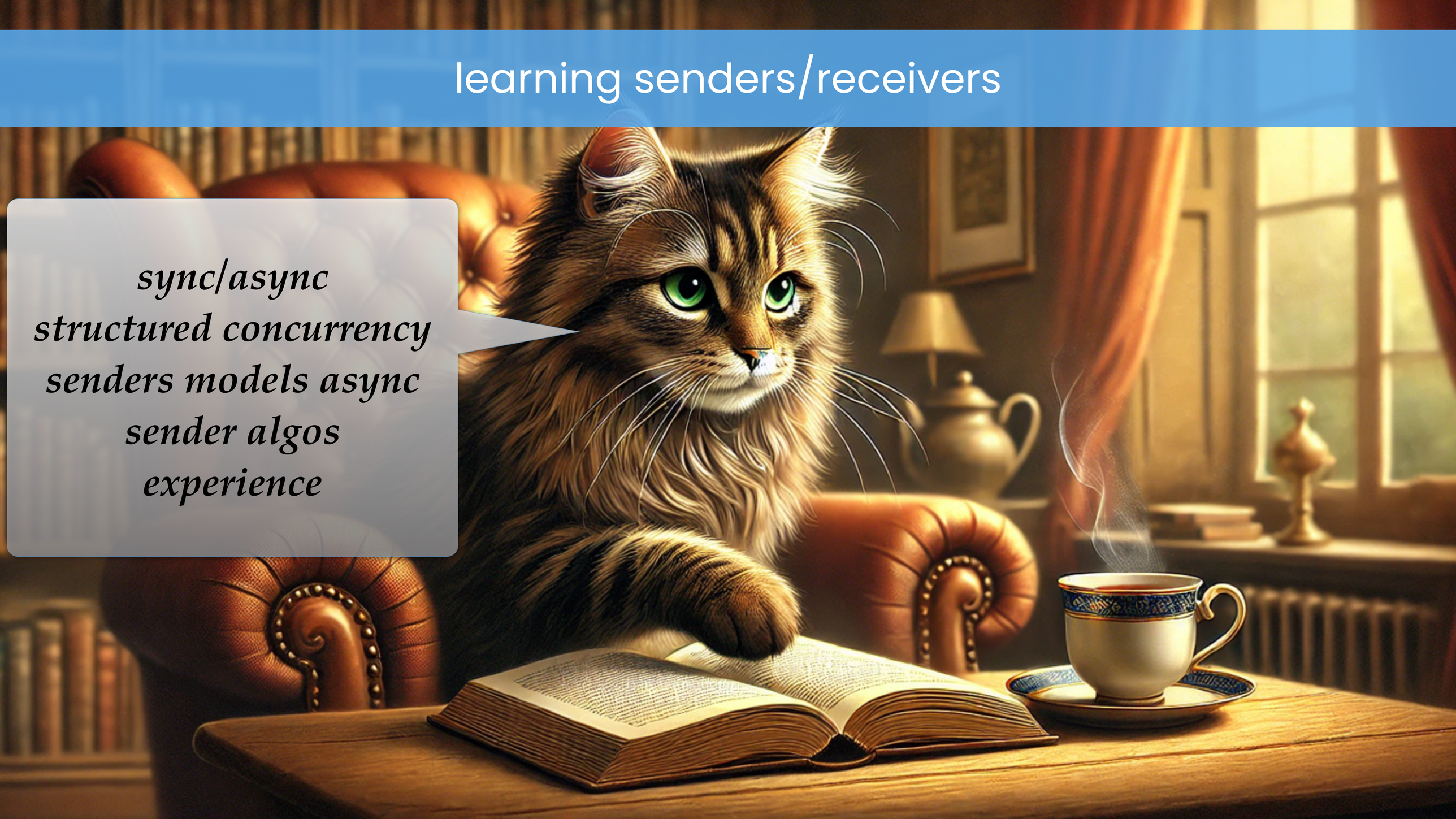
describe asynchronous work
compose well
are execute lazily
safe concurrency

algorithms in the base proposal

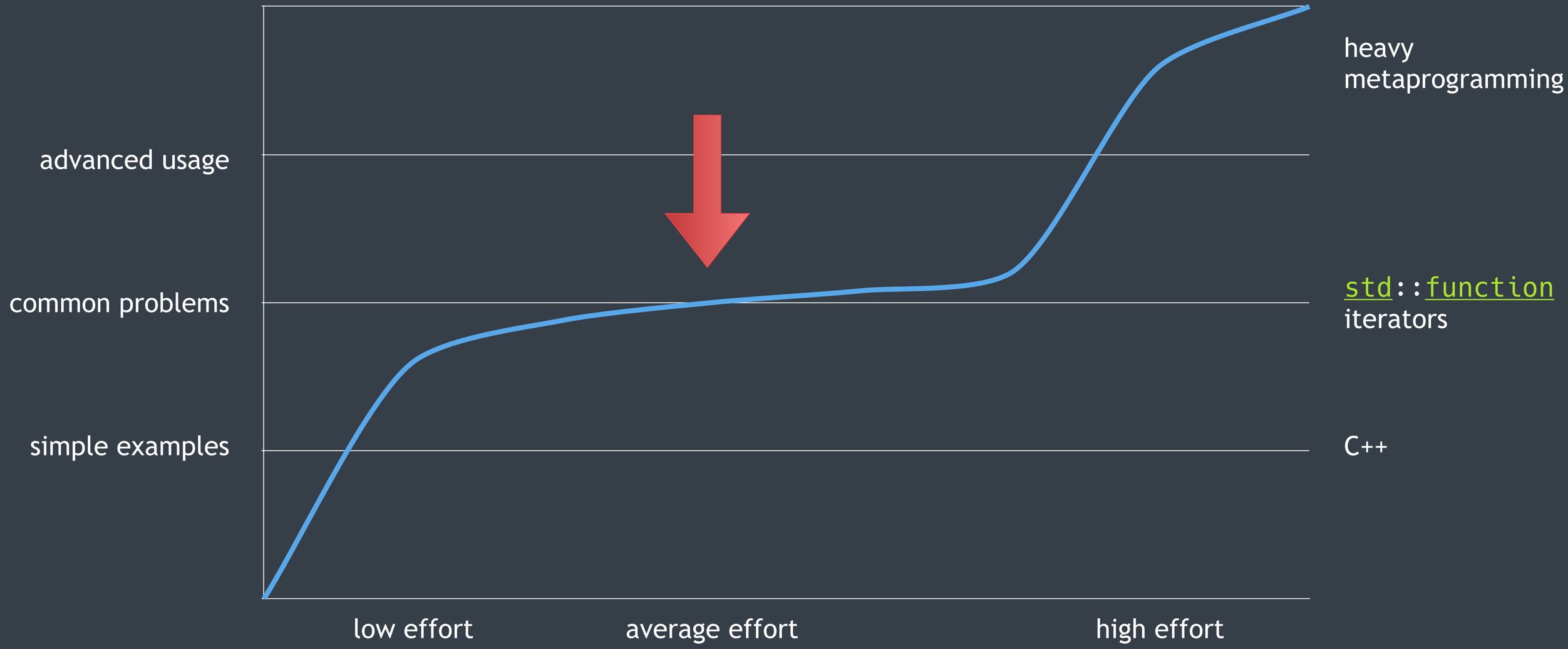
sender **factories**
sender **adaptors**
sender **consumers**


using senders/algos
adding concurrency
get_parallel_scheduler()
counting_scope
task

learning senders/receivers



learning senders/receivers




senders/receivers are easy

Thank You

@LucTeo@techhub.social
lucteo.ro