Zero-allocation & no type erasure futures

Vittorio Romeo

vittorioromeo.info

vittorio.romeo@outlook.com

vromeo5@bloomberg.net

@supahvee1234

ACCU 2018

14/04/2018

Bristol, UK

Bloomberg

- Building chains of asynchronous computations

  - Think of `std::future<T>` composition

- Without using:

  - Type erasure

  - Dynamic allocation

- Many templates

```cpp
auto graph = leaf{[]{ return "hello"; }}
    .then([](std::string x){ return x + " world"; })
```

- Returns `"hello world"`

- Type of `graph` :

```cpp
node::seq<
    node::leaf<
        void,
        (lambda #0)
    >,
    node::leaf<
        std::string,
        (lambda #1)
    >
>
```

```
auto graph = all{
    []{ return http_get_request("animals.com/cat/0.png"); },
    []{ return http_get_request("animals.com/dog/0.png"); }
}.then([](std::tuple<data, data> t){ /* ... */ });
```

- Type of `graph` :

```
node::seq<
    node::all<
        node::leaf<void, (lambda #0)>,
        node::leaf<void, (lambda #1)>
    >
    node::leaf<std::tuple<data, data>, (lambda #2)>
>
```

from the beginning
*std::future & std::async*

- `std :: async` provides a way of running a function *asynchronously*

- It returns an `std :: future` instance that will eventually contain its result

```cpp
auto f = std::async(std::launch::async, []
{
    std::this_thread::sleep_for(100ms);
    std::cout << "world\n";
});


std::cout << "hello ";
```
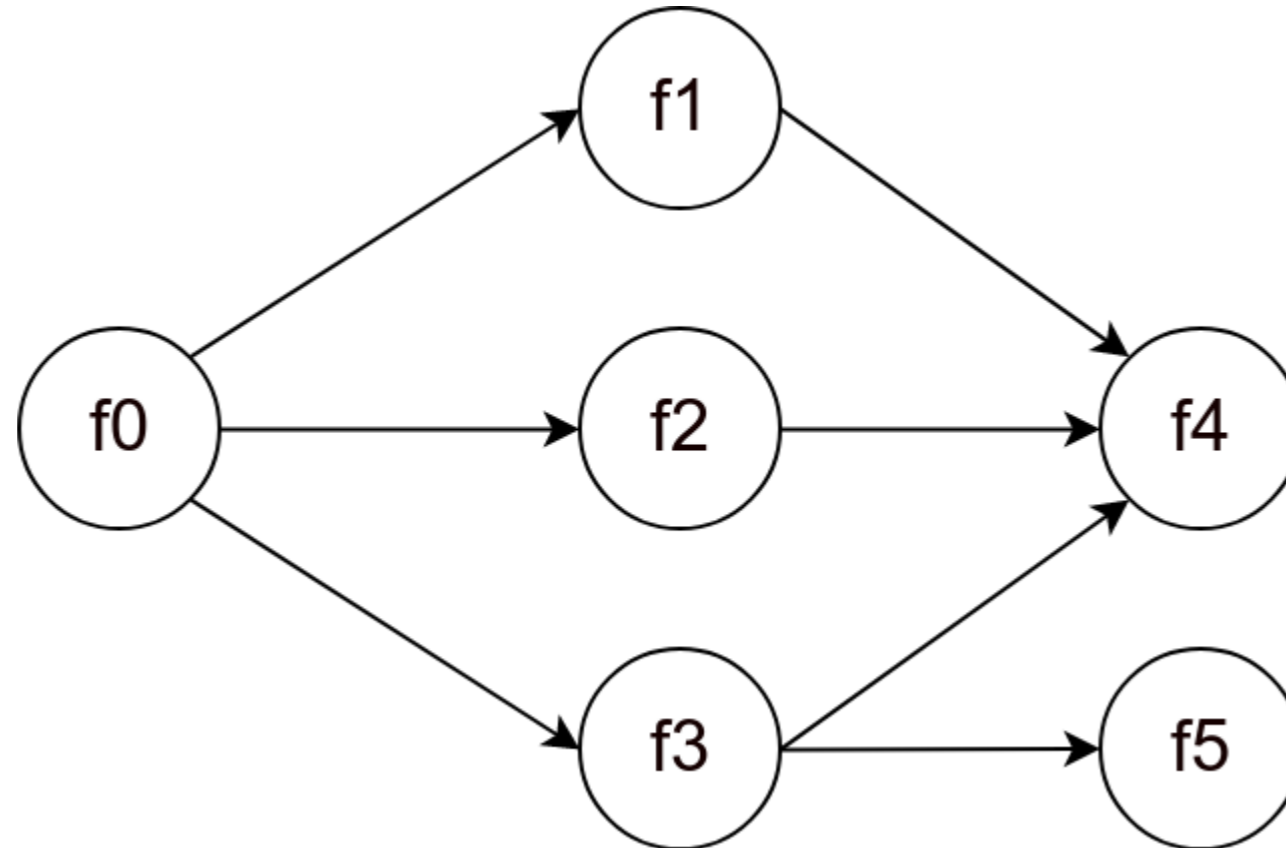
(on *wandbox.org*)

- `std :: future` and `std :: async` can easily model multiple tasks running in parallel

```cpp
auto f0 = std::async(std::launch::async, []{ /* ... */ });
auto f1 = std::async(std::launch::async, []{ /* ... */ });
auto f2 = std::async(std::launch::async, []{ /* ... */ });
```
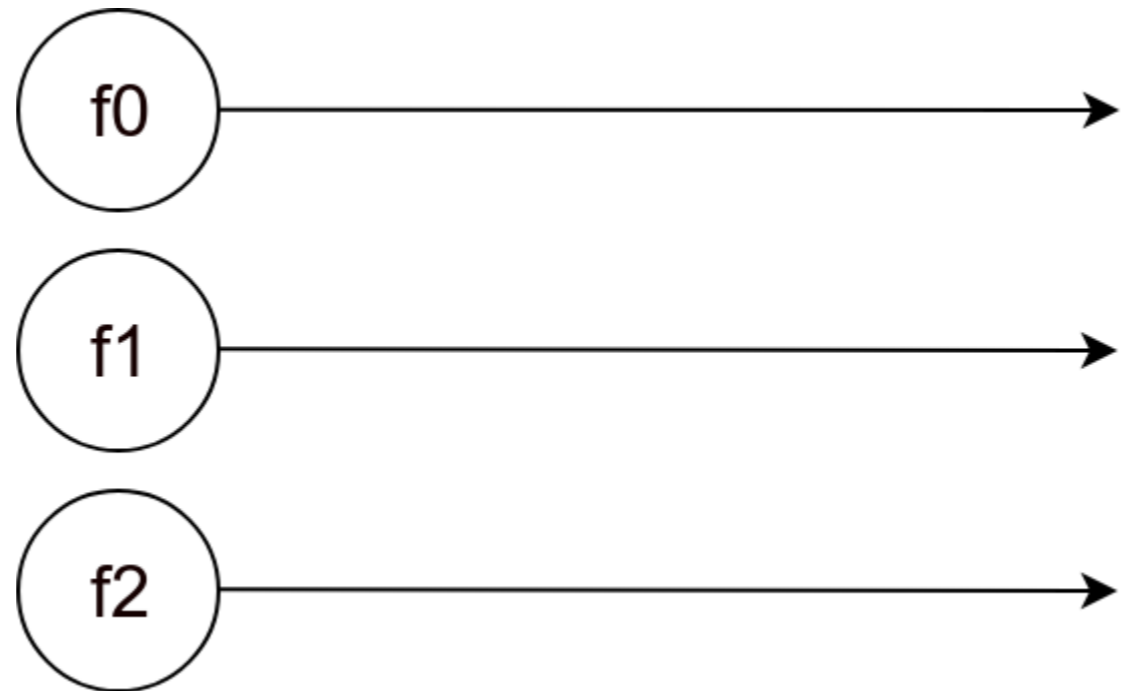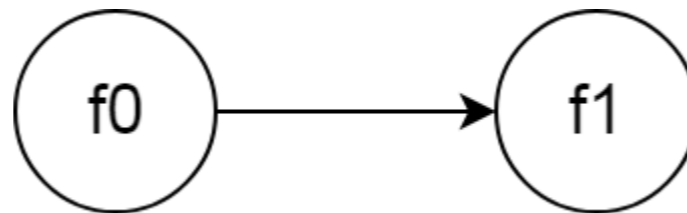
- They fall short when trying to model complicated dependency graphs

- *Sequential composition* implies nested `std::async` calls

- "*execute* `f0`, *__then__ execute* `f1`"

```cpp
auto f0 = std::async(std::launch::async, []
{
    std::cout << "hello ";
    auto f1 = std::async(std::launch::async, []
    {
        std::cout << "world\n";
    });
});
```
(on wandbox.org)

- Collecting multiple futures into a single one is not easy either

- "***when all*** `aX` *futures complete,* ***then*** *execute* `b0` "
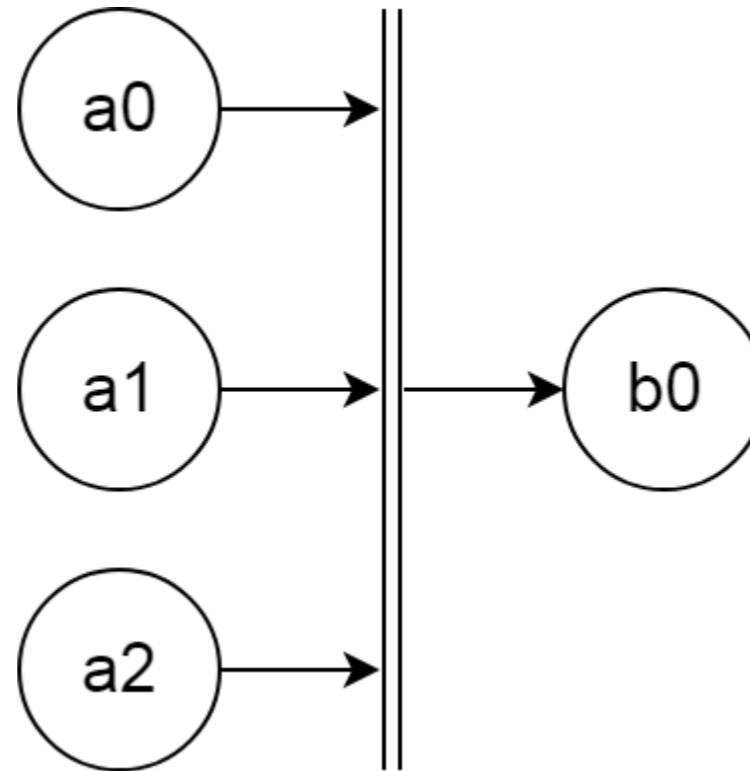
```cpp
std::future<void> f = std::async(std::launch::async, []
{
    auto a0 = std::async(std::launch::async, []{ std::cout << "a0\n"; });
    auto a1 = std::async(std::launch::async, []{ std::cout << "a1\n"; });
    auto a2 = std::async(std::launch::async, []{ std::cout << "a2\n"; });

    a0.get();
    a1.get();
    a2.get();

    auto b0 = std::async(std::launch::async, []{ std::cout << "b0\n"; });
});
```

(on *wandbox.org*)

- We want *intuitive* abstractions to express `future` composition

- Available in `boost::future`

- `std::experimental::future` attempts to standardize them

  - Part of the "Extensions For Concurrency" TS

  - Anthony Williams @ ACCU 2017
   **"Concurrency, Parallelism and Coroutines"**

```cpp
boost :: async(boost :: launch :: async, []
{
    std :: cout << "hello ";
})
.then([](auto)
{
    std :: cout << "world\n";
});
```

(on *wandbox.org*)

std::experimental::future
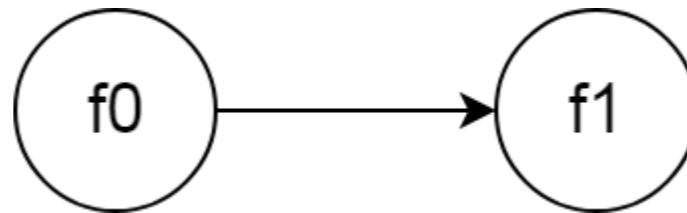
```
boost :: when_all(
    boost :: async(boost :: launch :: async, []{ std :: cout << "a0\n"; }),
    boost :: async(boost :: launch :: async, []{ std :: cout << "a1\n"; }),
    boost :: async(boost :: launch :: async, []{ std :: cout << "a2\n"; })
).then([](auto){ std :: cout << "b0\n"; });
```

(on *wandbox.org*)

from the beginning
std::experimental::future
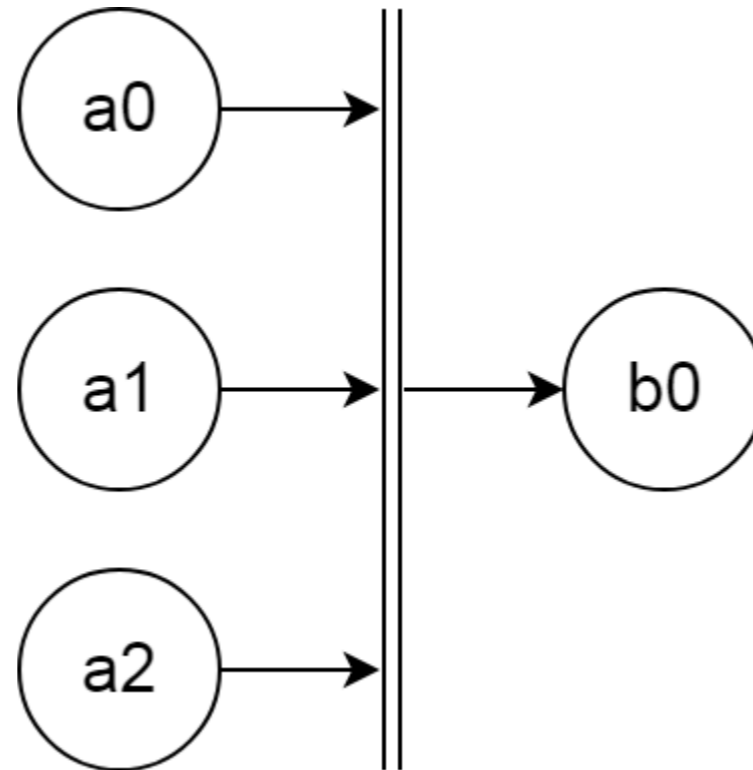
- Abstractions like `.then` , `when_all` , and `when_any` allow us to express future composition intuitively

- They always return a `future` that can be composed further

```cpp
template <class F>
auto future<T>::then(F func) → future<result_of_t<F(future<T>)>>;
```

```cpp
template <class... Futures>
auto when_all(Futures... futures) → future<std::tuple<Futures...>>;
```

```
boost :: future<int> a = /* ... */;
boost :: future<int> b = a.then([](auto){ /* ... */ });
```

- The result type of future composition is always `future<T>`

- This implies *type erasure*

- Additionally, `future` uses *dynamic allocation* to keep track of the "shared state"

- Can we avoid the overhead of *type erasure* and *dynamic allocation*?

    - Is it worth it?

an alternative design
avoiding type erasure

- *Type erasure* is necessary when the way futures are composed changes depending on **run-time control flow**

- If the **"shape"** of the future graph is **known at compile-time**, it can be encoded as part of the type system

```cpp
auto f0 = leaf{[]{ std::cout << 'a'; }};
auto f1 = f0.then([]{ std::cout << 'b'; });
```

```cpp
template <typename F>
leaf<F>::leaf(F&& f);
```

```cpp
template <typename F>
template <typename FThen>
sequential<leaf<F>, leaf<FThen>> leaf<F>::then(FThen f_then);
```

- The type of `f0` is `leaf<lambda#0>`

- The type of `f1` is `sequential< leaf<lambda#0>, leaf<lambda#1> >`

an alternative design
avoiding type erasure

```cpp
auto f0 = when_all([]{ std::cout << "a0"; },
                   []{ std::cout << "a1"; },
                   []{ std::cout << "a2"; });


auto f1 = f0.then([]{ std::cout << "b0"; });
```

```cpp
template <typename... Fs>
parallel<leaf<Fs>...> when_all(Fs... fs);
```

- The type of `f1` is:

```cpp
sequential<   parallel< leaf<lambda#0-2>... >, leaf<lambda#3>   >
```

- *Type erasure* can be avoided by encoding the structure of the graph in a **type**

- What about *dynamic allocation*?

- `future<T>` uses *dynamic allocation* in order to provide a **shared state** for the eventual result/exception

  - Additional synchronization primitives for abstraction such as `when_all` and `when_any` might also require *dynamic allocation*
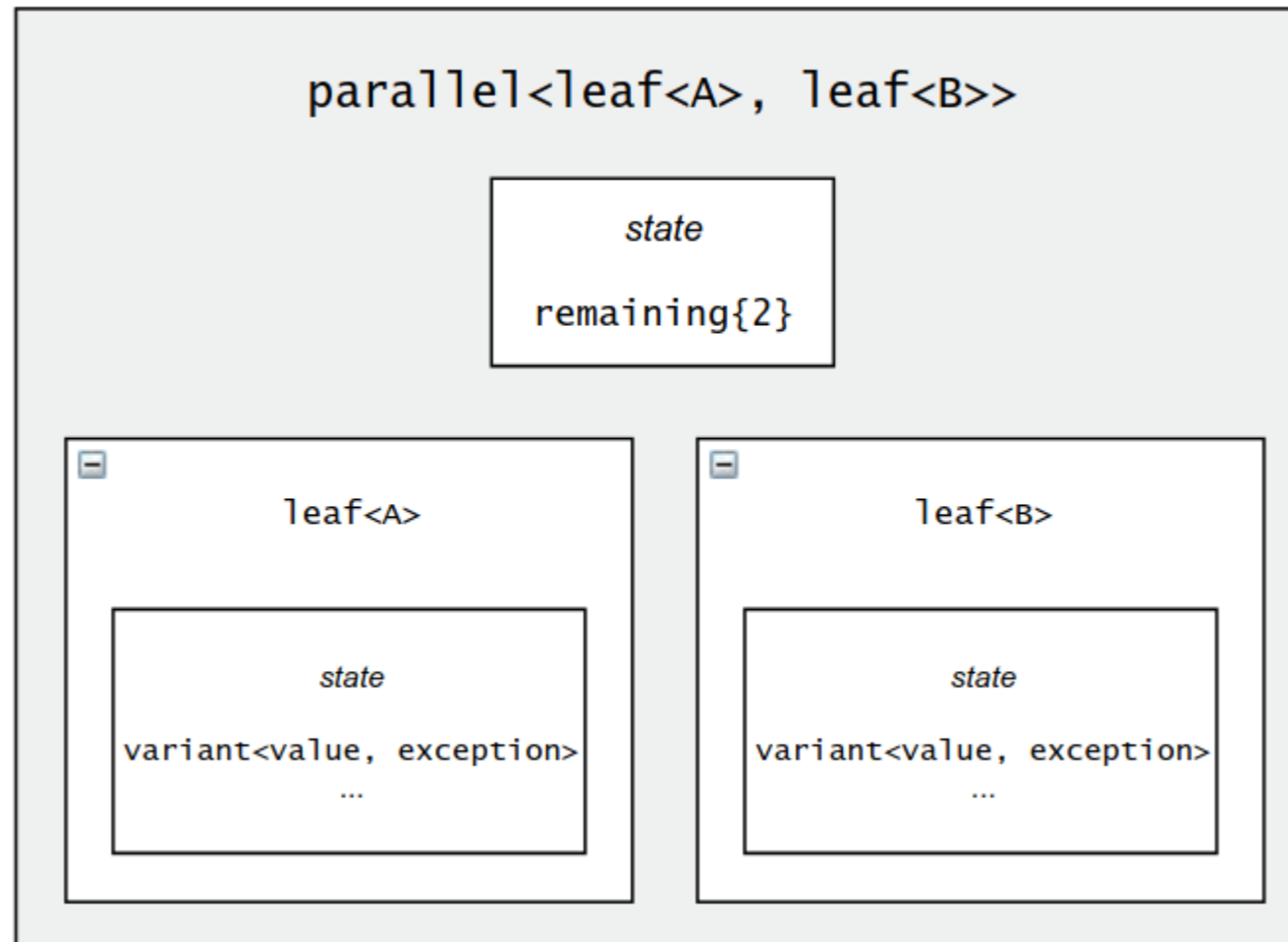
- `std :: future<T>` , `std :: promise<T>` , and the *shared state*
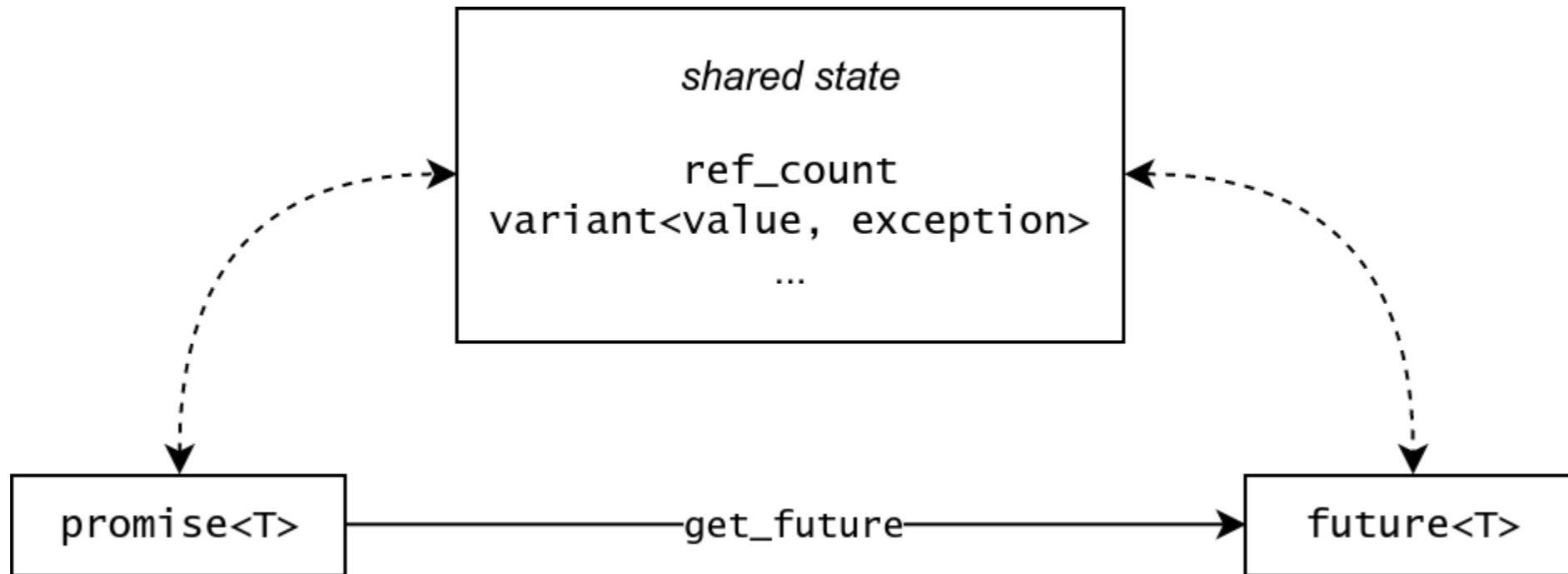
- What if we store the state *in-place* inside the final graph?

- This requires the graph object to be kept alive until execution is completed

```
graph<...>

    state

variant<value, exception>

        ...
```
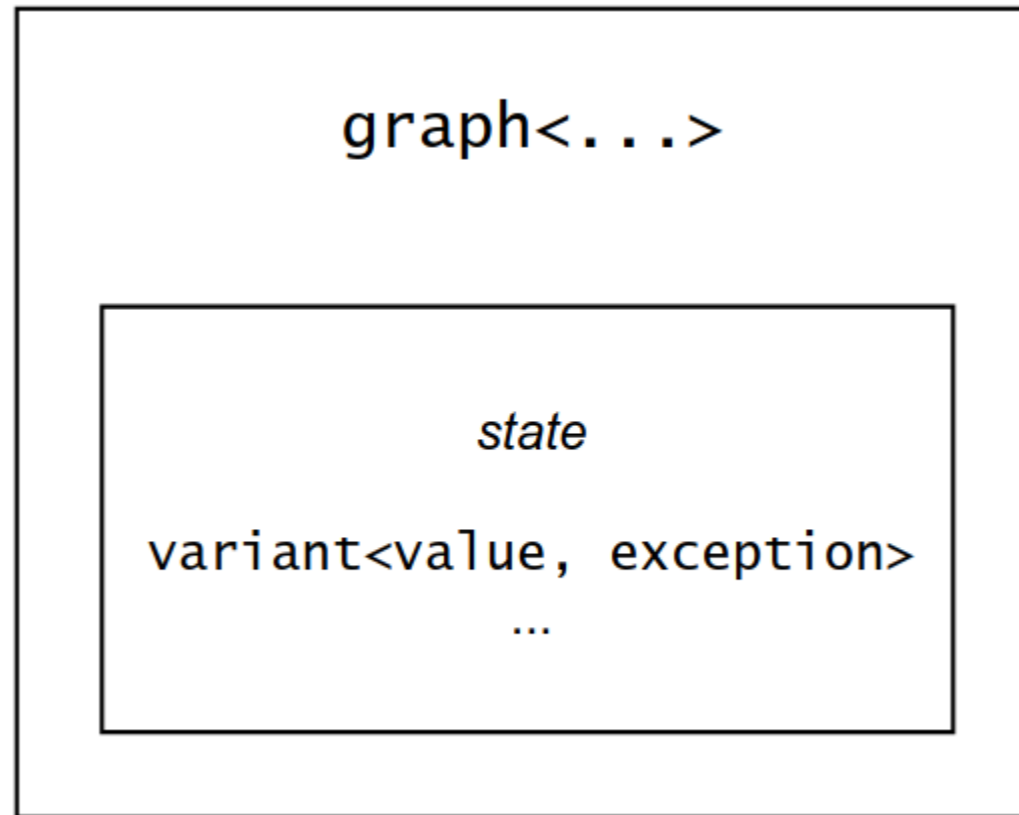
- E.g. the implementation of `when_all` could store an *atomic counter* of the remaining nodes *in-place* in the `parallel< ... >` node

```
parallel<leaf<A>, leaf<B>>
```

state

remaining{2}

leaf<A>

state

variant<value, exception>
...

leaf<B>

state

variant<value, exception>
...

an alternative design
avoiding dynamic allocation

```
when_all(  when_any(a, b), c  ).then(  d  )
```

an alternative design
avoiding dynamic allocation
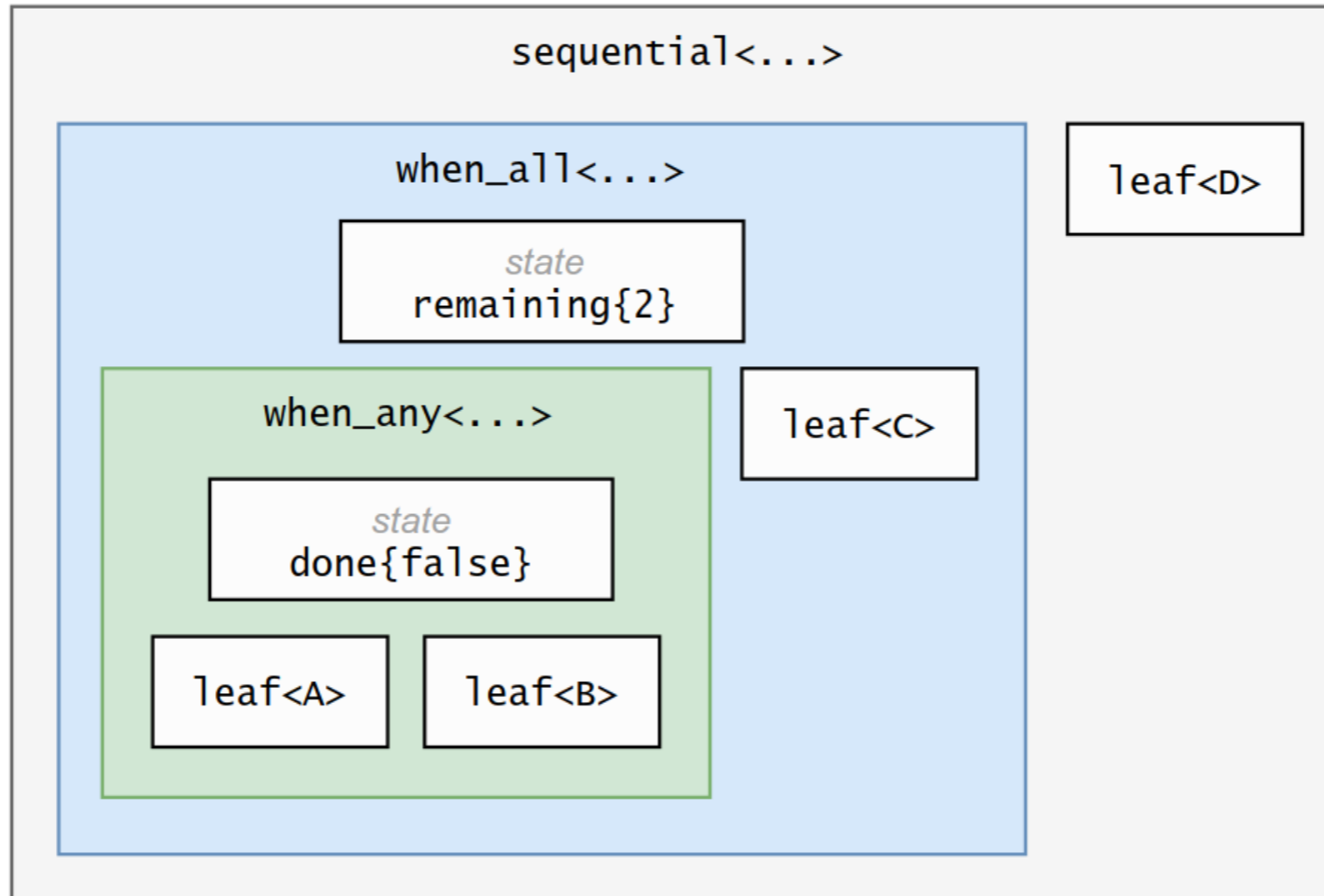
```cpp
auto f = when_all(when_any(a, b), c).then(d);
scheduler.execute(f);

// `f` must be kept alive until execution is completed
```

- If `f` is executed through an asynchronous non-blocking scheduler, the user must make sure that `f` lives long enough

  - Remember that the "shared" state exists *in-place* inside `f`

an alternative design
recap

- *Type erasure* will be avoided by encoding the entire computation graph as part of the type system

    - The "shape" of the graph must be known at compile-time

- *Dynamic allocation* will be avoided by storing the "shared state" in-place inside the graph object

    - The final graph object must outlive the execution of its nodes

We will implement:

1. `leaf` node

2. `seq` node *(sequential composition)*

3. `all` node *("when all" composition)*

4. Return value propagation

5. Blocking execution

6. Continuation-style syntax (`.then`)

7. `any` node *("when any" composition)*

- All node types will expose the following member function:

```cpp
template <typename Scheduler, typename Then>
void /* node */::execute(Scheduler& scheduler, Then&& then) &
{
    // * Execute stored computation through `scheduler`
    // * Asynchronously continue execution via `then`
}
```

- It will later be improved to support propagation of return values

- `scheduler(f)` can simply be `std::thread{f}.detach()`

- `execute` is `&` ref-qualified as it requires the node to be kept alive

- A `leaf` node simply wraps a single computation `F`

- It will expose `execute`, but won't make use of `scheduler`

  - The computation can be executed on the "current" thread

- Example usage:

```
leaf l{[]{ std::cout << "hello "; }};
l.execute(scheduler, []{ std::cout << "world\n"; });
```

  - `leaf`'s template parameters are being deduced thanks to C++17's *class template argument deduction*

  - The continuation is provided as a callback to avoid unnecessary blocking

```cpp
template <typename F>
struct leaf : F
{
    leaf(F&& f) : F{std::move(f)}
    {
    }


    template <typename Scheduler, typename Then>
    void execute(Scheduler&, Then&& then) &
    {
        (*this)();
        std::forward<Then>(then)();
    }
};
```

*(on **wandbox.org**)*

- `F` is inherited to allow EBO *(empty base optimization)*

```
template <typename F>
leaf<F>::leaf(F&& f) : F{std::move(f)}
{
}
```

- `F` is deduced from the `leaf(F&&)` constructor:

```
leaf l{some_lambda};
```

↓

```
leaf<decltype(some_lambda)> l{some_lambda};
```

- The `leaf` node on its own is not really useful

- It is the smallest *composable* piece of the graph

- Let's implement `seq` next

- The `seq` node takes two nodes `A` and `B` as input

- It executes `A`, **then** `B`

```
leaf l0{[]{ std::cout << "hello "; }};
leaf l1{[]{ std::cout << "world"; }};

seq s0{std::move(l0), std::move(l1)};
s0.execute(scheduler, []{ std::cout << "!\n"; });
```

(on wandbox.org)

```cpp
template <typename A, typename B>
struct seq : A, B
{
    seq(A&& a, B&& b) : A{std::move(a)}, B{std::move(b)}
    {
    }

    template <typename Scheduler, typename Then>
    void execute(Scheduler& scheduler, Then&& then) &
    {
        A::execute(scheduler, [this, &scheduler, then]
        {
            B::execute(scheduler, then);
        });
    }
};
```

```cpp
template <typename A, typename B>
template <typename Scheduler, typename Then>
void seq<A, B>::execute(Scheduler& scheduler, Then&& then) &
{
    A::execute(scheduler, [this, &scheduler, then]
    {
        B::execute(scheduler, then);
    });
}
```

- `A` is immediately executed

- The execution of `B` is passed as the `then` argument to `A::execute`

- This allows non-blocking asynchronous composition

```cpp
leaf l0{[]{ std::cout << "hello "; }};
leaf l1{[]{ std::cout << "world"; }};

seq s0{std::move(l0), std::move(l1)};
s0.execute(scheduler, []{ std::cout << "!\n"; });
```
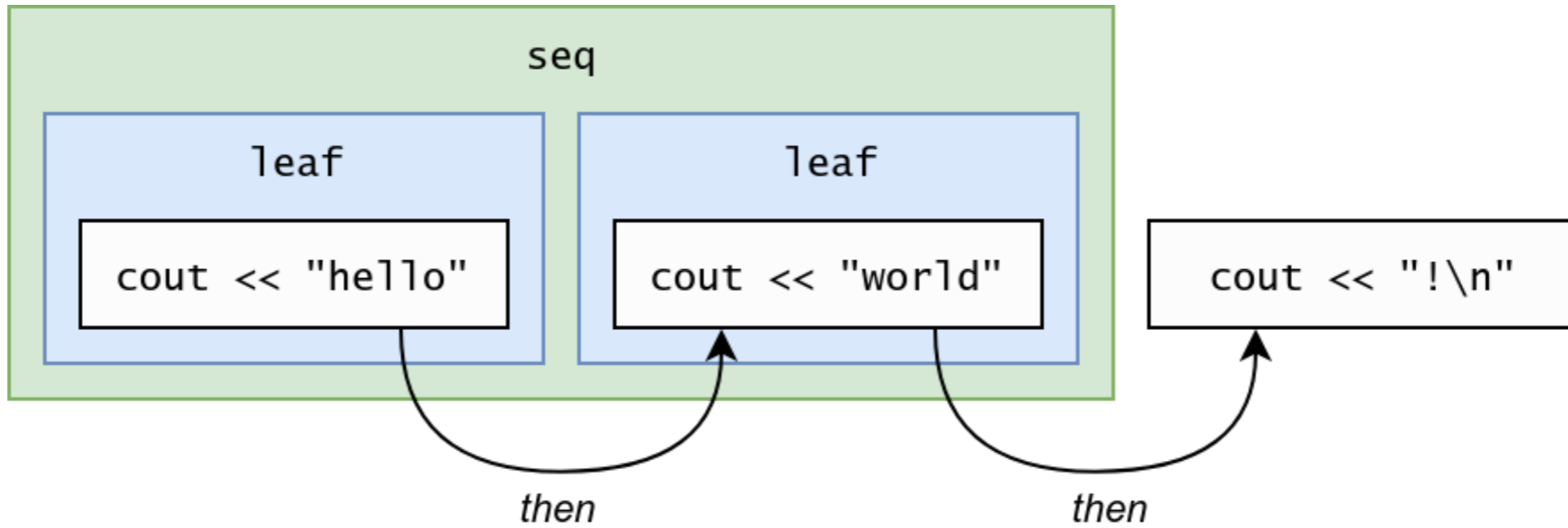
- The `seq` node allows *sequential composition* of nodes

- It executes two nodes, one after another, asynchronously

  - For `leaf` nodes, this is indistinguishable from blocking

 - For nodes like `any`, this is crucial
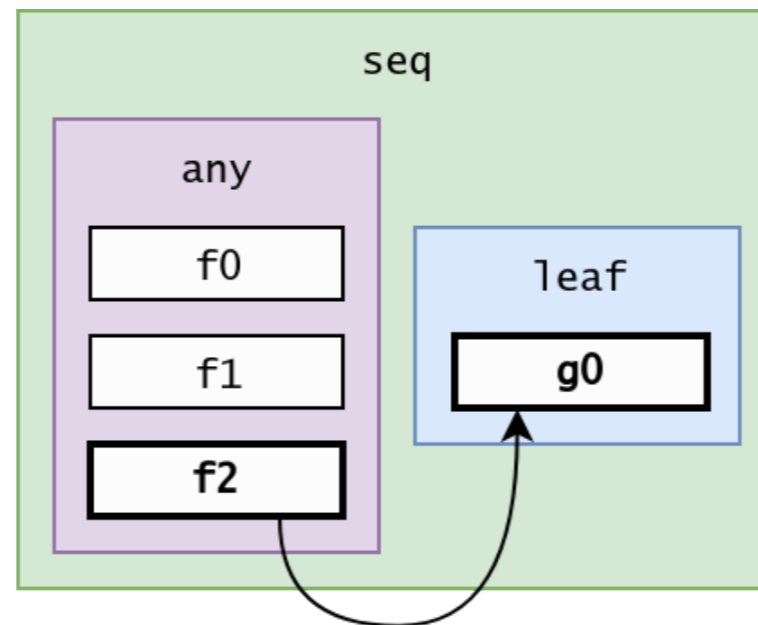
- The `all` node will be the first one to make use of `scheduler`

- It takes an arbitrary number of nodes `Fs ...` as input

- It executes `Fs ...` in parallel

- The execution of `all<Fs ... >` is completed when all `Fs ...` are completed

```cpp
all graph{leaf{[]{ std::cout << "a0\n"; }},
          leaf{[]{ std::cout << "a1\n"; }},
          leaf{[]{ std::cout << "a2\n"; }}};

graph.execute(scheduler, []{ std::cout << "b0\n"; });
std::this_thread::sleep_for(100ms);
```

(on *wandbox.org*)

```
all graph{leaf{[]{ std::cout << "a0\n"; }},
          leaf{[]{ std::cout << "a1\n"; }},
          leaf{[]{ std::cout << "a2\n"; }}};

graph.execute(scheduler, []{ std::cout << "b0\n"; });
```

(on *wandbox.org*)
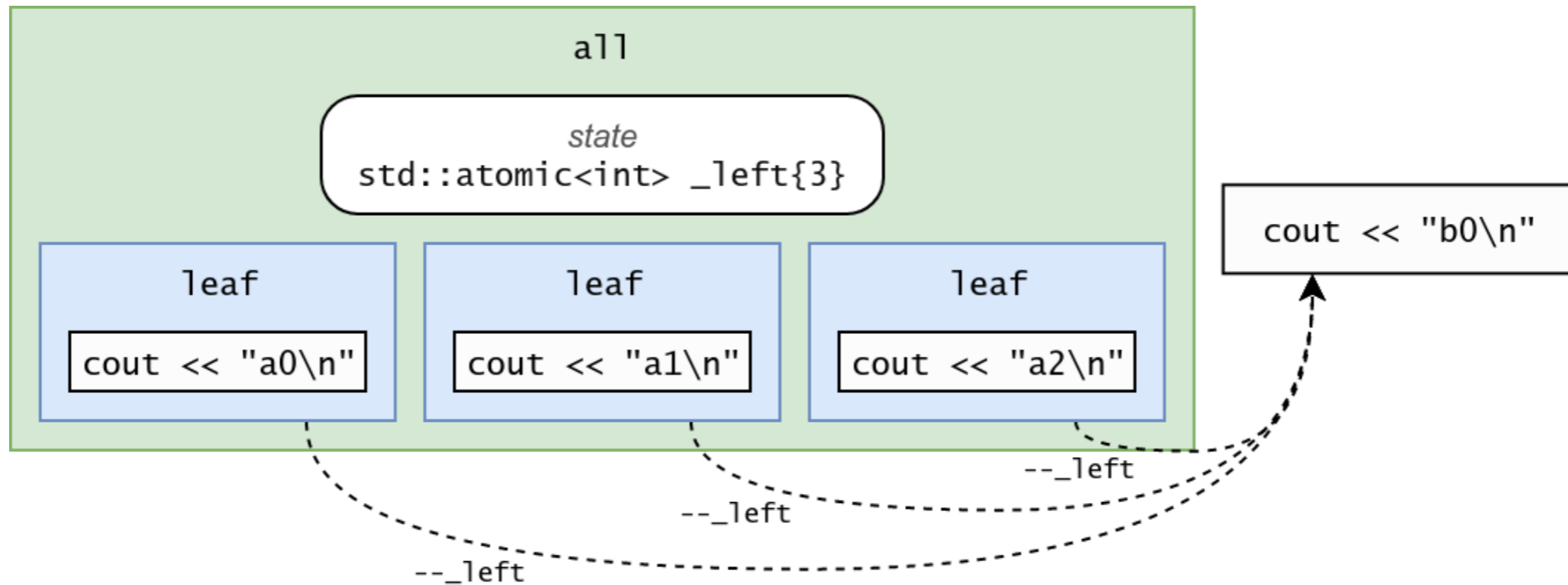
- `all<Fs ... >` contains an `atomic` counter initialized to `sizeof ... (Fs)`

  - It keeps track of how many nodes need to complete their execution

  - When it reaches `0`, the `then` continuation of the `all` node is executed

  - Every node in `Fs ...` decrements the counter upon completion

- Since the nodes can be executed on separated threads, the `all<Fs ... >` object must be kept alive until all nodes have finished

  - This is why we added a `this_thread :: sleep_for`

  - We'll see a more robust solution later

```cpp
template <typename ... Fs>
struct all : Fs ...
{
    std::atomic<int> _left;

    all(Fs&& ... fs) : Fs{std::move(fs)} ...
    {
    }


    template <typename Scheduler, typename Then>
    void execute(Scheduler& scheduler, Then&& then) &;
};
```

- Inheritance is used not only for EBO, but also because it makes it easier to work with `Fs ...` as a pack

```cpp
template <typename ... Fs>
template <typename Scheduler, typename Then>
void all<Fs ... >::execute(Scheduler& scheduler, Then&& then) &
{
    _left.store(sizeof ... (Fs));

    (scheduler([this, &scheduler, &f = static_cast<Fs&>(*this), then]
    {
        f.execute(scheduler, [this, then]
        {
            if(_left.fetch_sub(1) == 1) { then(); }
        });
    }), ... );
}
```

```
(scheduler([this, &scheduler, &f = static_cast<Fs&>(*this), then]
{
    f.execute(scheduler, [this, then]
    {
        if(_left.fetch_sub(1) == 1) { then(); }
    });
}), ... );
```

- This entire snippet is a *fold expression* over the *comma operator*

- In short, it schedules the execution of every `f` in `Fs ...`

  - The atomic counter is decremented as part of the `then` continuation of `f`

  - The last `f` will execute the `then` continuation of `all<Fs ... >`

Example expansion for two hypotetical `a` and `b` nodes:

```cpp
_left.store(2);
scheduler([this, &scheduler, &a, then]
{
    a.execute(scheduler, [this, then]
    {
        if(_left.fetch_sub(1) == 1) { then(); }
    });
}),
scheduler([this, &scheduler, &b, then]
{
    b.execute(scheduler, [this, then]
    {
        if(_left.fetch_sub(1) == 1) { then(); }
    });
});
```

- `leaf<F>` : wraps a computation into a node, allows composition

- `seq<A, B>` : executes `A` , then `B`

- `all<Fs ... >` : executes all `Fs ...` in parallel

- We can model arbitrary fork/join computation graphs

implementation
what we have so far
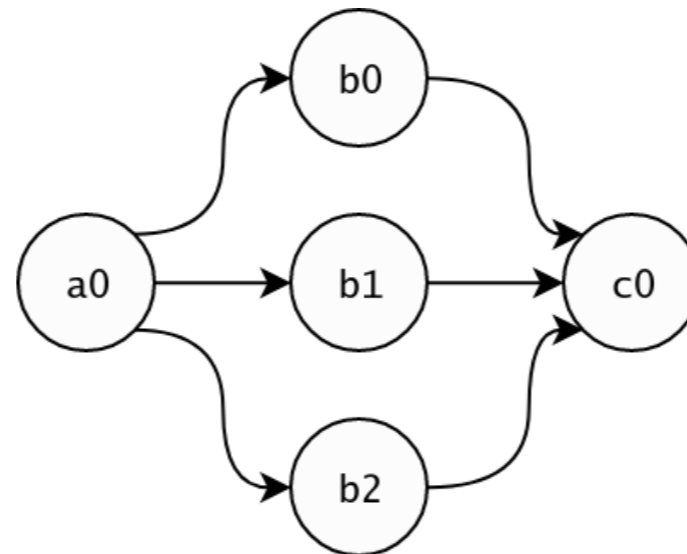
```cpp
auto graph = seq{seq{leaf{[]{ std::cout << "a0\n"; }},
                     all{leaf{[]{ std::cout << "b0\n"; }},
                         leaf{[]{ std::cout << "b1\n"; }},
                         leaf{[]{ std::cout << "b2\n"; }}}},
                 leaf{[]{ std::cout << "c0\n"; }}};
```

(on wandbox.org)

- We still can't return values from a node and pass them onwards

  - Let's deal with that next

implementation
part 4 - return value propagation

- The execution of any graph always begins from a `leaf`

- `leaf` nodes must be able to *produce* and *accept* values

- Composition nodes such as `seq` and `all` must be aware of what values their children are producing

- This information can be encoded as part of the node type

- Let's begin by computing the `in_type` and `out_type` of `leaf`

```cpp
template <typename In, typename F>
struct leaf : F
{
    using in_type = In;
    using out_type = std::result_of_t<F&(In)>;

    // ...
};
```

- A new `In` template parameter was added

- How can we make it play nicely with *class template argument deduction*?

```
template <typename F>
leaf(F&&) → leaf<first_arg_t<decltype(&std::decay_t<F>::operator())>,
                 std::decay_t<F>>;
```

```
template <typename F>
using first_arg_t = std::tuple_element_t<
    1,
    boost::callable_traits::args_t<F>
>;
```

- `boost::callable_traits::args_t<F>` returns a `std::tuple` containing all the argument types of `F`

  ○ If `F` is a `Callable`, the first type is the type of the object itself

- We can now instantiate `leaf` objects as follows:

```cpp
leaf{[](int){ }};
// Deduced as `leaf<int, /* lambda */>`

leaf{[](std::string){ }};
// Deduced as `leaf<std::string, /* lambda */>`
```
                                                                    (on *wandbox.org*)

- However, `leaf<In, F>::execute` still looks like this:

```cpp
template <typename Scheduler, typename Then>
void leaf<In, F>::execute(Scheduler&, Then&& then) &
{

    (*this)(/* ? */);
    std::forward<Then>(then)();
}
```

- The solution is simple: accept an input argument in `execute`

implementation
part 4 - return value propagation

```cpp
template <typename Scheduler, typename Input, typename Then>
void leaf<In, F>::execute(Scheduler&, Input&& input, Then&& then) &
{
    std::forward<Then>(then)(
        (*this)(std::forward<Input>(input));
    );
}
```

- We invoke `*this` with the input, and pass the result to `then`

- Usage example:

```cpp
auto graph = leaf{[](int x){ return x * 2; }};
graph.execute(scheduler, 21, [](int x){ std::cout << x << '\n'; });
```

*(on **wandbox.org**)*

```cpp
template <typename In, typename F>
struct leaf : F
{
    using in_type = In;
    using out_type = std::result_of_t<F&(In)>;

    leaf(F&& f) : F{std::move(f)}
    {
    }

    template <typename Scheduler, typename Input, typename Then>
    void execute(Scheduler&, Input&& input, Then&& then) &
    {
        std::forward<Then>(then)((*this)(std::forward<Input>(input)));
    }
};

template <typename F>
leaf(F&&) → leaf<first_arg_t<decltype(&std::decay_t<F>::operator())>,
                std::decay_t<F>>;
```

- All the other node types will require the same modifications:

  - Expose `in_type` and `out_type`

  - Accept an `input` argument in `execute`

  - Execute the child nodes by passing `input`

  - Invoke the `then` continuation by passing the result of the above operation

- Let's apply these changes to `seq`

```cpp
template <typename A, typename B>
struct seq : A, B
{
    using in_type = typename A::in_type;
    using out_type = typename B::out_type;

    // ...

};
```
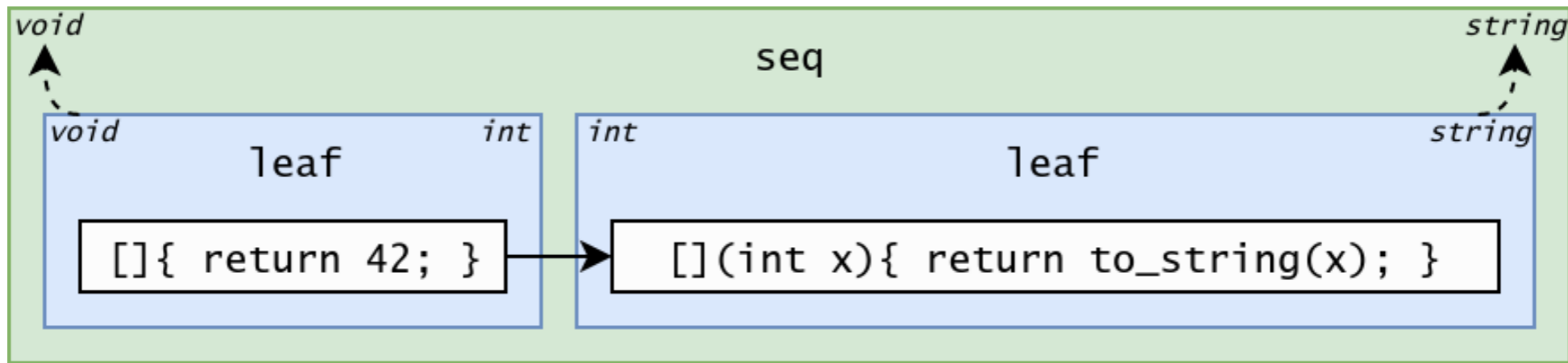
```cpp
template <typename Scheduler, typename Then>
void seq<A, B>::execute(Scheduler& scheduler, Then&& then) &
{
    A::execute(scheduler, [this, &scheduler, then]
    {
        B::execute(scheduler, then);
    });
}
```

...becomes...

```cpp
template <typename Scheduler, typename Input, typename Then>
void seq<A, B>::execute(Scheduler& scheduler, Input&& input, Then&& then) &
{
    A::execute(scheduler, FWD(input), [this, &scheduler, then](auto&& r)
    {
        B::execute(scheduler, FWD(r), then);
    });
}
```

implementation
part 4 - return value propagation

- Usage example:

```cpp
auto graph = seq{leaf{[](int x){ return x * 2; }},
                 leaf{[](int x){ return std::to_string(x); }}};

graph.execute(scheduler, 21, [](std::string x)
{
    std::cout << x << '\n';
});
```

(on wandbox.org)

```cpp
template <typename A, typename B>
struct seq : A, B
{
    using in_type = typename A::in_type;
    using out_type = typename B::out_type;

    seq(A&& a, B&& b) : A{std::move(a)}, B{std::move(b)}
    {
    }


    template <typename Scheduler, typename Input, typename Then>
    void execute(Scheduler& scheduler, Input&& input, Then&& then) &
    {
        A::execute(scheduler, FWD(input), [this, &scheduler, then](auto&& out)
        {
            B::execute(scheduler, FWD(out), then);
        });
    }
};
```
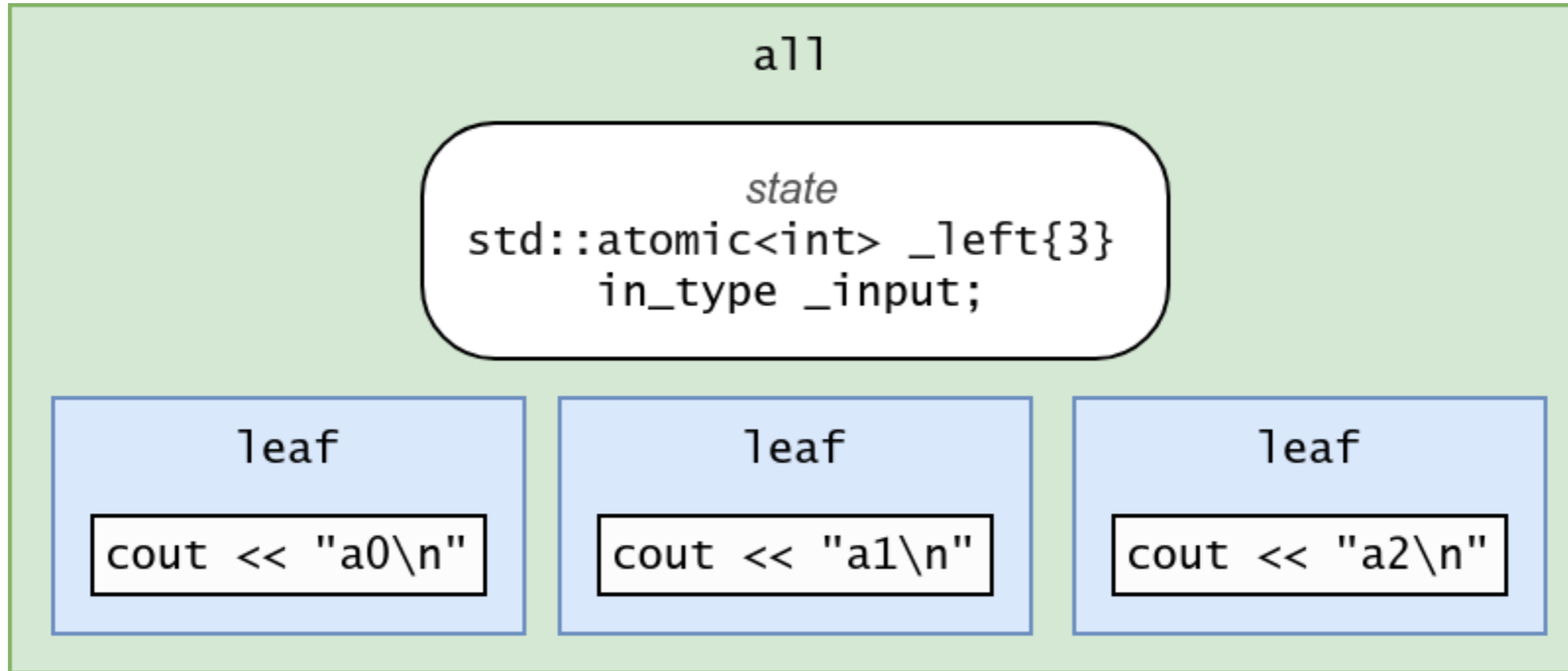
(on wandbox.org)

- `leaf` and `seq` now support asynchronous propagation of values

- `all` is slightly more complicated:

  - Multiple parallel computations need access to the `input` value

  - It cannot be passed directly from the stack, as it is not guaranteed to outlive the parallel computations

  - The value could be copied for each computation, but that might be expensive

  - We will store it inside `all` itself, so that it is guaranteed to live long enough

```cpp
template <typename Scheduler, typename Input, typename Then>
void all<Fs...>::execute(Scheduler& sched, Input&& input, Then&& then) &
{
    _left.store(sizeof...(Fs));

    (sched([this, &sched, &input,
//                          ^~~~~~
                &f = static_cast<Fs&>(*this), then]
    {
        f.execute(sched, input, [this, then]
        {//                      ^~~~~
         //         dangling reference (!)

            if(_left.fetch_sub(1) == 1) { then(); }
        });
    }), ...);
}
```
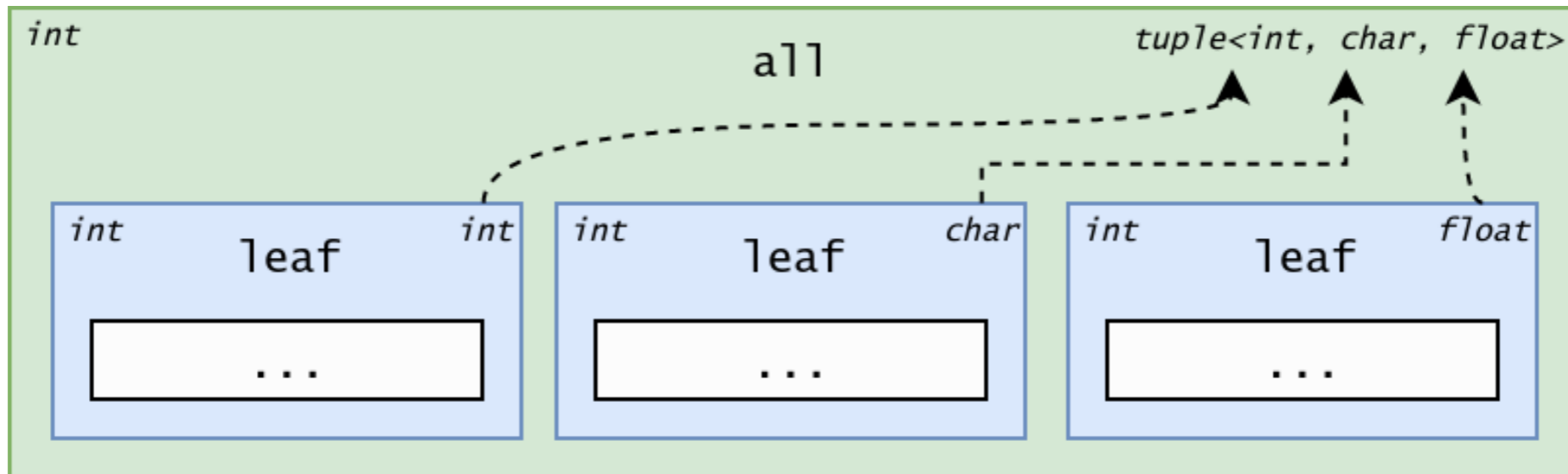
implementation
part 4 - return value propagation

- Children nodes will simply *refer* to the `_input` stored in the `all` node

implementation
part 4 - return value propagation

```cpp
template <typename... Fs>
struct all : Fs...
{
    using in_type = std::common_type_t<typename Fs::in_type...>;
    using out_type = std::tuple<typename Fs::out_type...>;

    // ...

};
```

```cpp
template <typename... Fs>
struct all : Fs...
{

    struct shared_state
    {

        in_type _input;
        std::atomic<int> _left;


        template <typename Input>
        shared_state(Input&& input) : _input{FWD(input)}
        {

            _left.store(sizeof...(Fs));
        }
    };


    // ...

};
```

```cpp
template <typename... Fs>
struct all : Fs...
{
    using in_type = std::common_type_t<typename Fs::in_type...>;
    using out_type = std::tuple<typename Fs::out_type...>;

    struct shared_state { /* ... */ };

    aligned_storage_for<shared_state> _state;
    out_type _values;

    // ...
};
```

- `_state` is constructed when calling `execute`, destroyed on completion

- `_values` will be filled during `execute` and passed down to children nodes

```cpp
template <typename Scheduler, typename Input, typename Then>
void all<Fs...>::execute(Scheduler& sched, Input&& input, Then&& then) &
{
    _state.construct(FWD(input));
    // * schedule children nodes...
    // * fill `_values` with each node's result...
    // * finally: invoke `then` and destroy `_state` on completion...
}
```

- We need to fill the `_values` tuple with the results of each node

  - Therefore we need an index to use `std::get`

- Our beautiful *fold expression* will have to be replaced with something a little bit more powerful

```
template <typename Scheduler, typename Input, typename Then>
void all<Fs...>::execute(Scheduler& sched, Input&& input, Then&& then) &
{
    _state.construct(FWD(input));

    enumerate_types<Fs...>([&](auto i, auto t)
    {
        // ...
    });
}
```

- `enumerate_types<Fs...>` will invoke the passed lambda with:

  - `i`: `std::integral_constant<int, I>{}` storing the current index

  - `t`: `type_wrapper<T>{}` storing the current type

```cpp
template <typename Scheduler, typename Input, typename Then>
void all<Fs...>::execute(Scheduler& sched, Input&& input, Then&& then) &
{
    _state.construct(FWD(input));

    enumerate_types<Fs...>([&](auto i, auto t)
    {
        sched([this, &sched,
               &f = static_cast<unwrap<decltype(t)>&>(*this), then]
        {
            // ...
        });
    });
}
```

- `f` evaluates to `*this`, casted to the type in `Fs...` of the current iteration

```cpp
template <typename Scheduler, typename Input, typename Then>
void all<Fs...>::execute(Scheduler& sched, Input&& input, Then&& then) &
{
    _state.construct(FWD(input));

    enumerate_types<Fs...>([&](auto i, auto t)
    {
        sched([this, &sched,
               &f = static_cast<unwrap<decltype(t)>&>(*this), then]
        {
            f.execute(sched, _state→_input, [this, then](auto&& r)
            {
                // ...
            });
        });
    });
}
```

- `f` is executed with `_state→_input`, which lives as long as needed

```cpp
template <typename Scheduler, typename Input, typename Then>
void all<Fs ...>::execute(Scheduler& sched, Input&& input, Then&& then) &
{
    _state.construct(FWD(input));
    enumerate_types<Fs ...>([&](auto i, auto t)
    {
        sched([this, &sched,
               &f = static_cast<unwrap<decltype(t)>&>(*this), then]
        {
            f.execute(sched, _state→_input, [this, then](auto&& r)
            {
                std::get<decltype(i){}>(_values) = FWD(r);
                if(_state→_left.fetch_sub(1) == 1)
                {
                    _state.destroy();
                    then(std::move(_values));
                }
            });
        });
    });
}
```

- Usage example:

```cpp
auto graph = seq{all{leaf{[](int x){ return x; }},
                     leaf{[](int x){ return x + 1; }},
                     leaf{[](int x){ return x + 2; }}},
                 leaf{[](std::tuple<int, int, int> y)
                 {
                     return get<0>(y) + get<1>(y) + get<2>(y);
                 }}};

graph.execute(scheduler, 0, [](int x){ std::cout << x << '\n'; });
```

(on *wandbox.org*)

`all<Fs ... >` return value propagation recap:

- The `input` and the *atomic counter* are stored **in-place** in `shared_state`

  - It is constructed when calling `execute`, destroyed on completion

- The output values are stored in a `std::tuple<typename Fs::out_type ...>`

  - This lives **in-place** inside the node

  - It is filled by enumerating `Fs ...` at compile-time

  - The last computation to finish invokes `then` with the tuple

- While enumerating `Fs ...`, we can apply an optimization:

  - If `i == 0`, do not schedule the current computation

- I've carefully avoided using `void` in all the examples

- It is not a *"regular type"* - it requires extra care

- A preliminary step is defining an empty `nothing` type and using it place of `void`

```
struct nothing { };
```

- A real solution uses *metaprogramming* to automatically convert `void` to `nothing` transparently to the user

  - github.com/SuperV1234/orizzonte/nothing.hpp

  - This will not be covered in the talk

- So far we have used `this_thread::sleep_for` in order to prevent `graph` from being destroyed too early

```
all graph{leaf{[]{ std::cout << "a0\n"; }},
          leaf{[]{ std::cout << "a1\n"; }},
          leaf{[]{ std::cout << "a2\n"; }}};

graph.execute(scheduler, []{ std::cout << "b0\n"; });
std::this_thread::sleep_for(100ms);
```

- Can we do better?

- We can use a **latch** (e.g. `std :: experimental :: latch` or `boost :: latch`)

- It basically is a **counter** + **condition variable** + **mutex**

    - The current thread is blocked until the counter reaches zero

```cpp
std :: experimental :: latch l{3};

std :: thread{[&l]{ l.count_down(); }}.detach();
std :: thread{[&l]{ l.count_down(); }}.detach();
std :: thread{[&l]{ l.count_down(); }}.detach();

l.wait(); // blocks until `counter = 0`
```

- Let's create a `sync_execute` abstraction that uses a latch

```cpp
template <typename Scheduler, typename Graph, typename Then>
void sync_execute(Scheduler& scheduler, Graph&& graph, Then&& then)
{
    std::experimental::latch l{1};

    graph.execute(scheduler, nothing{}, [&](auto&& res)
    {
        then(FWD(res));
        l.count_down();
    });

    l.wait();
}
```

- Given a `graph`, it is executed under a latch `l`

- The continuation attached at the end of the graph will unblock the latch

- Usage example:

```cpp
auto graph = seq{all{leaf{[](int x){ return x; }},
                     leaf{[](int x){ return x + 1; }},
                     leaf{[](int x){ return x + 2; }}},
                 leaf{[](std::tuple<int, int, int> y)
                 {
                     return get<0>(y) + get<1>(y) + get<2>(y);
                 }}};

sync_execute(scheduler, graph, [](int x){ std::cout << x << '\n'; });
```

(on *wandbox.org*)

- Removes the need for `sleep_for` , providing a deterministic lifetime for `graph`

- Which one is better?

```
auto graph = seq{seq{leaf{a}, leaf{b}}, leaf{c}};
```

```
auto graph = leaf{a}.then(b).then(c);
```

- Let's add a `.then` member function to every node type

```cpp
template <typename In, typename F>
struct leaf : F
{
    template <typename X>
    auto then(X&& x);

    // ...
};
```

- It will take an arbitrary object `x` :

  - If `x` is a node, it will return a `seq{std::move(*this), x}`

  - If `x` is not a node, it will return `seq{std::move(*this), leaf{x}}`

```cpp
template <typename X>
auto leaf<In, F>::then(X&& x)
{

    if constexpr(detail::is_executable<X>{})
    {

        return seq{std::move(*this), FWD(x)};
    }
    else
    {

        return seq{std::move(*this), leaf{FWD(x)}};
    }

}
```

- `is_executable` uses the *detection idiom* to detect whether or not `x` exposes `.execute`

- Usage example:

```cpp
auto graph = leaf{[]{ return 10; }}
    .then([](int x){ return x * 2; })
    .then(all{leaf{[](int x){ return x + 5; }},
              leaf{[](int x){ return x - 5; }}})
    .then([](std::tuple<int, int> y)
    {
        return std::get<0>(y) + std::get<1>(y);
    });
```

(on *wandbox.org*)

- The `any` node is the most complicated one

- It takes `Fs ...` as input, and as soon as any of them it's completed, the `then` continuation is invoked

- The remaining computations still run in the background

```
any graph{a.then(b), c.then(d)};
```

- How do we know when we can safely destroy `graph` ?

  - `b` could finish before `c` or `d`

  - Our previous `latch` solution will not work

```
any graph{leaf{[]{ std::cout << "a0\n"; }},
          leaf{[]{ std::cout << "a1\n"; }},
          leaf{[]{ std::cout << "a2\n"; }}};

graph.execute(scheduler, []{ std::cout << "b0\n"; });
```

(on *wandbox.org*)
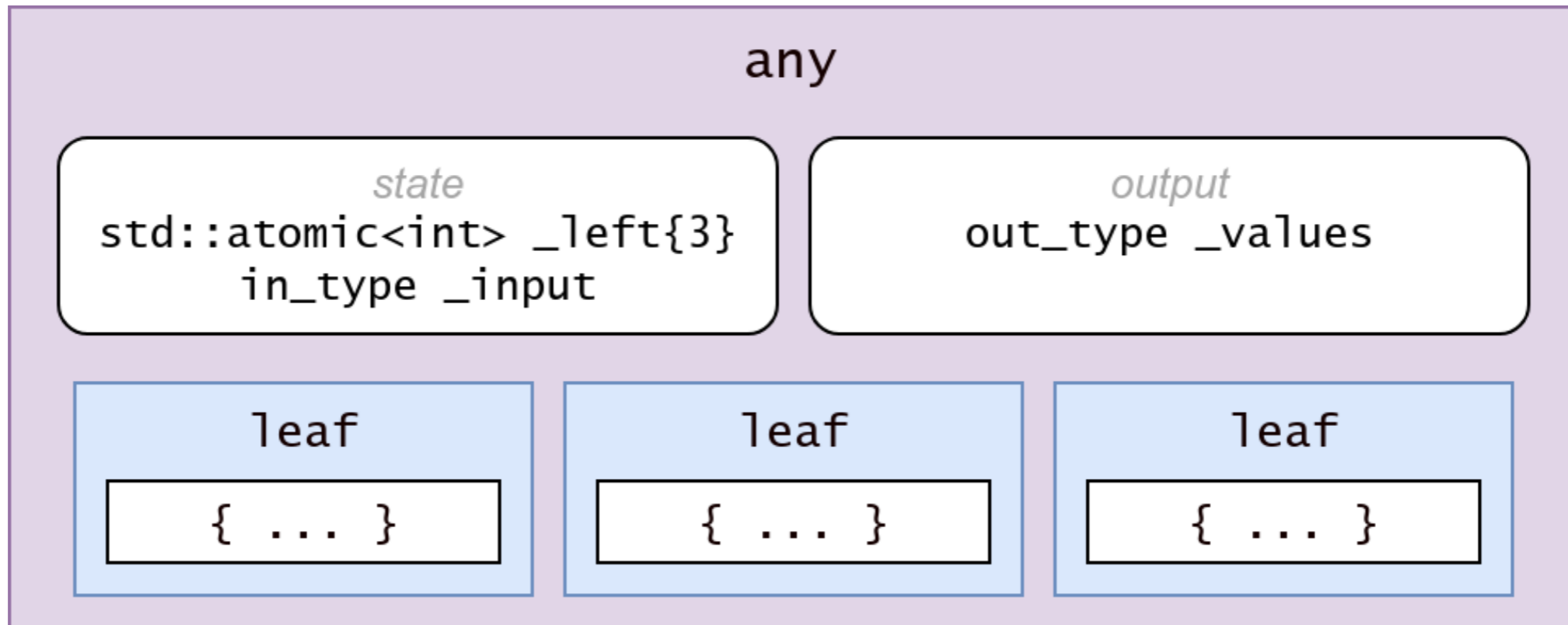
```
template <typename... Fs>
struct any : Fs...
{
    using in_type = std::common_type_t<typename Fs::in_type...>;
    using out_type = std::variant<typename Fs::out_type...>;

    any(Fs&&... fs) : Fs{std::move(fs)}... { }
    // ...
```
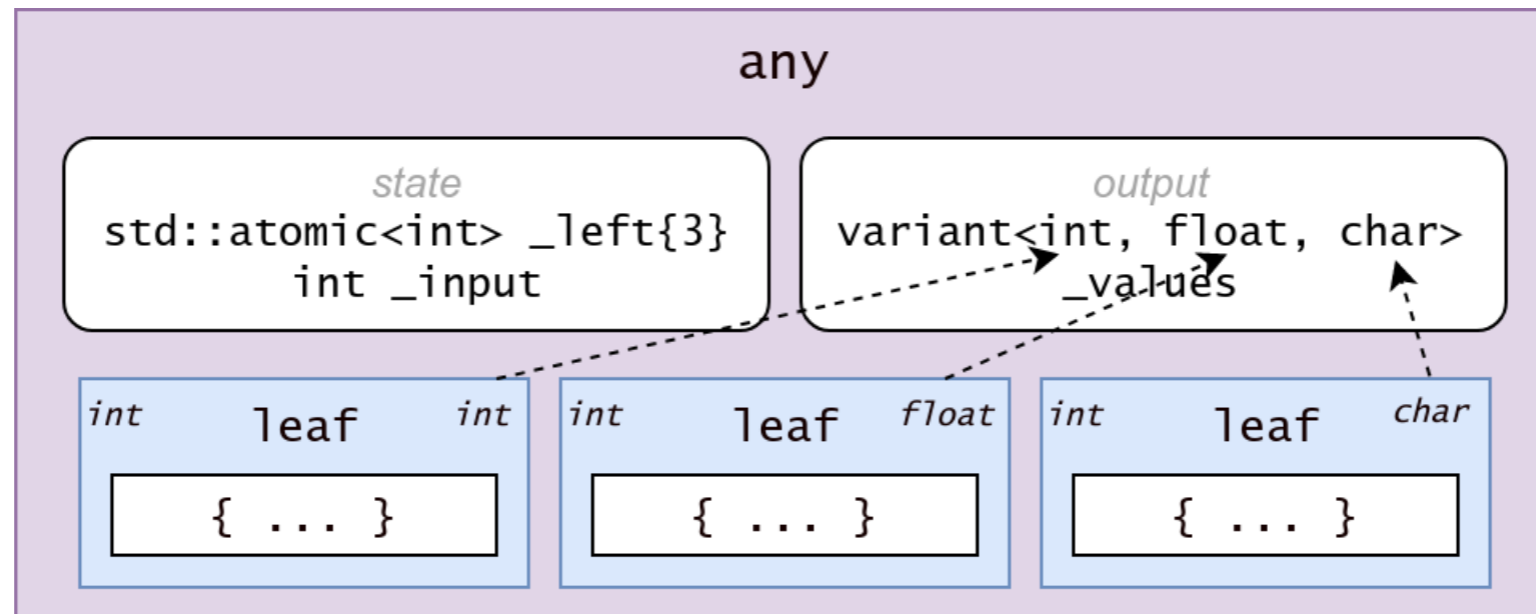
```cpp
template <typename... Fs>
struct any : Fs...
{
    struct shared_state
    {
        in_type _input;
        std::atomic<int> _left;

        template <typename Input>
        shared_state(Input&& input) : _input{FWD(input)}
        {
            _left.store(sizeof...(Fs));
        }
    };

    aligned_storage_for<shared_state> _state;
    out_type _values;
    // ...
```

```
template <typename Scheduler, typename Input, typename Then, typename Cleanup>
void execute(Scheduler& sched, Input&& input, Then&& then, Cleanup&& cleanup) &
{
    _state.construct(FWD(input));
    (sched([this, &sched, &f = static_cast<Fs&>(*this), then, cleanup]
    {
        // ...
    }), ... );
}
```

- Every node gets a new argument in `execute` : `cleanup`

- Similarly to `then` , it is executed when a node is ready to be cleaned up

  - For `leaf` , `seq` , and `all` : cleanup is the same as completion

  - For `any` : cleanup and completion may happen at different times

```
(sched([this, &sched, &f = static_cast<Fs&>(*this), then, cleanup]
{
    f.execute(sched, _state→_input, [this, then, cleanup](auto&& r)
    {
        // ...
    }, cleanup);
}), ... );
```

- `cleanup` is copied alongside `then`

- `cleanup` is propagated to children nodes' `execute`

```
f.execute(sched, _state→_input, [this, then, cleanup](auto&& r)
{
    const auto l = _state→_left.fetch_sub(1);
    if(l == sizeof...(Fs))
    {

        _values = FWD(r);
        then(std::move(_values));

    }


    if(l == 1) { _state.destroy(); cleanup(); }
}, cleanup);
```

- `l == sizeof...(Fs)` is the "completion condition"

- `l == 1` is the "cleanup condition"

```cpp
template <typename Scheduler, typename Input, typename Then, typename Cleanup>
void execute(Scheduler& sched, Input&& input, Then&& then, Cleanup&& cleanup) &
{
    _state.construct(FWD(input));
    (sched([this, &sched, &f = static_cast<Fs&>(*this), then, cleanup]
    {
        f.execute(sched, _state→_input, [this, then, cleanup](auto&& r)
        {
            const auto l = _state→_left.fetch_sub(1);
            if(l == sizeof...(Fs))
            {
                _values = FWD(r);
                then(std::move(_values));
            }

            if(l == 1) { _state.destroy(); cleanup(); }
        }, cleanup);
    }), ... );
}
```

```cpp
template <typename Scheduler, typename Graph, typename Then>
void sync_execute(Scheduler& scheduler, Graph&& graph, Then&& then)
{
    latch l{std::decay_t<Graph>::cleanup_count() + 1};

    graph.execute(scheduler, nothing{},
        [&](auto&&... res) { then(FWD(res)...); l.count_down(); },
        [&] { l.count_down(); });
}
```

- `cleanup_count` returns the count of `any` nodes in the graph

```
auto f0 = all{any{b0, b1}, any{c0, c1}};
sync_execute(scheduler, f0, []{ /* then */ });
```
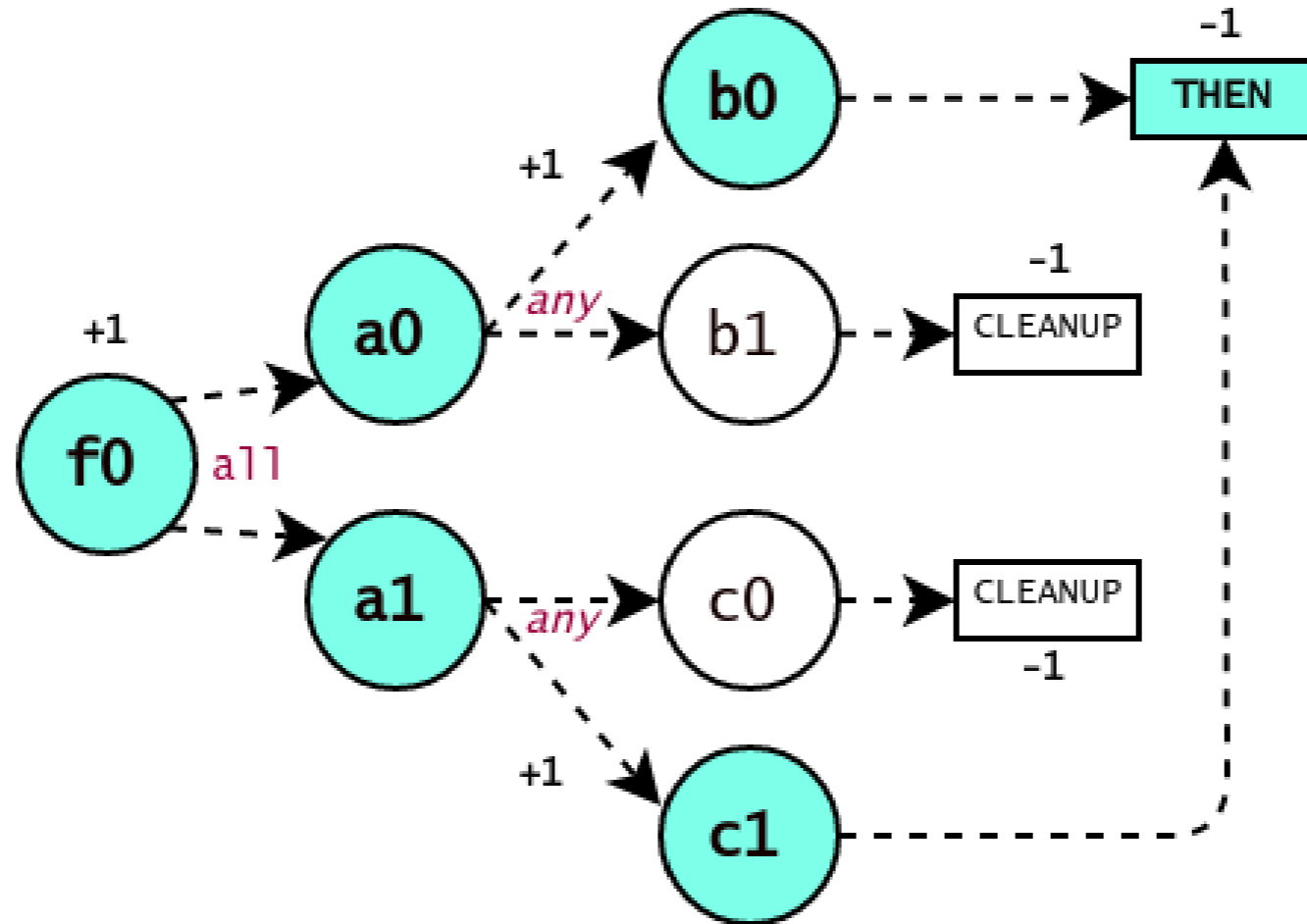
`all<Fs ... >` node recap:

- Invokes the `then` continuation as soon as one of `Fs ...` is completed

  - The remaining computations are run in the background

  - The node must be kept alive until **all** computations are done

- Output values are stored in a `std :: variant<typename Fs :: out_type ... >`

- Introduces an additional `cleanup` step for each node type

  - Only `any` actively invokes it when all `Fs ...` are done

  - The latch is initializes to `1 + count_of_any_nodes`

  - This allows to block until the graph can be destroyed

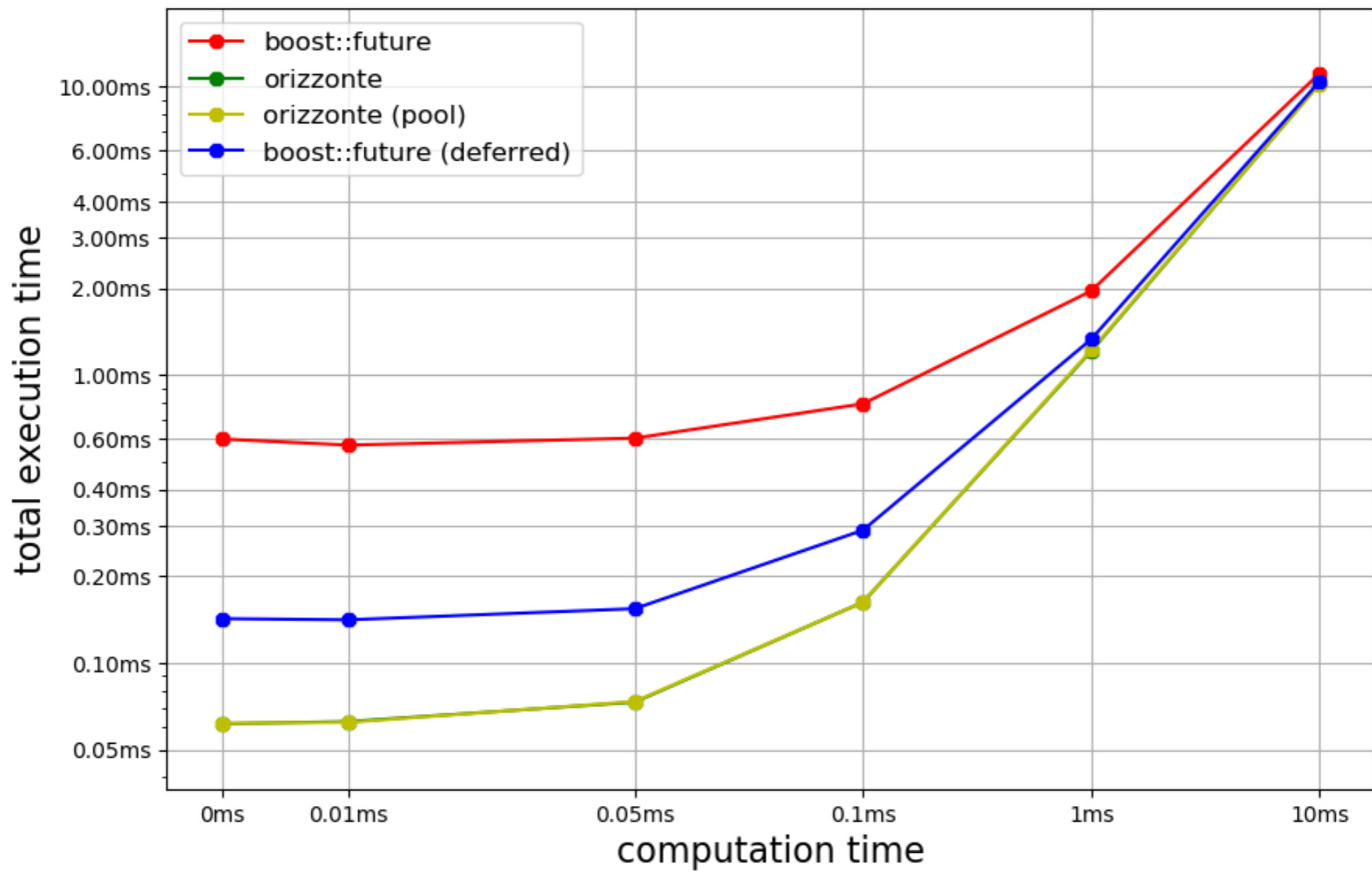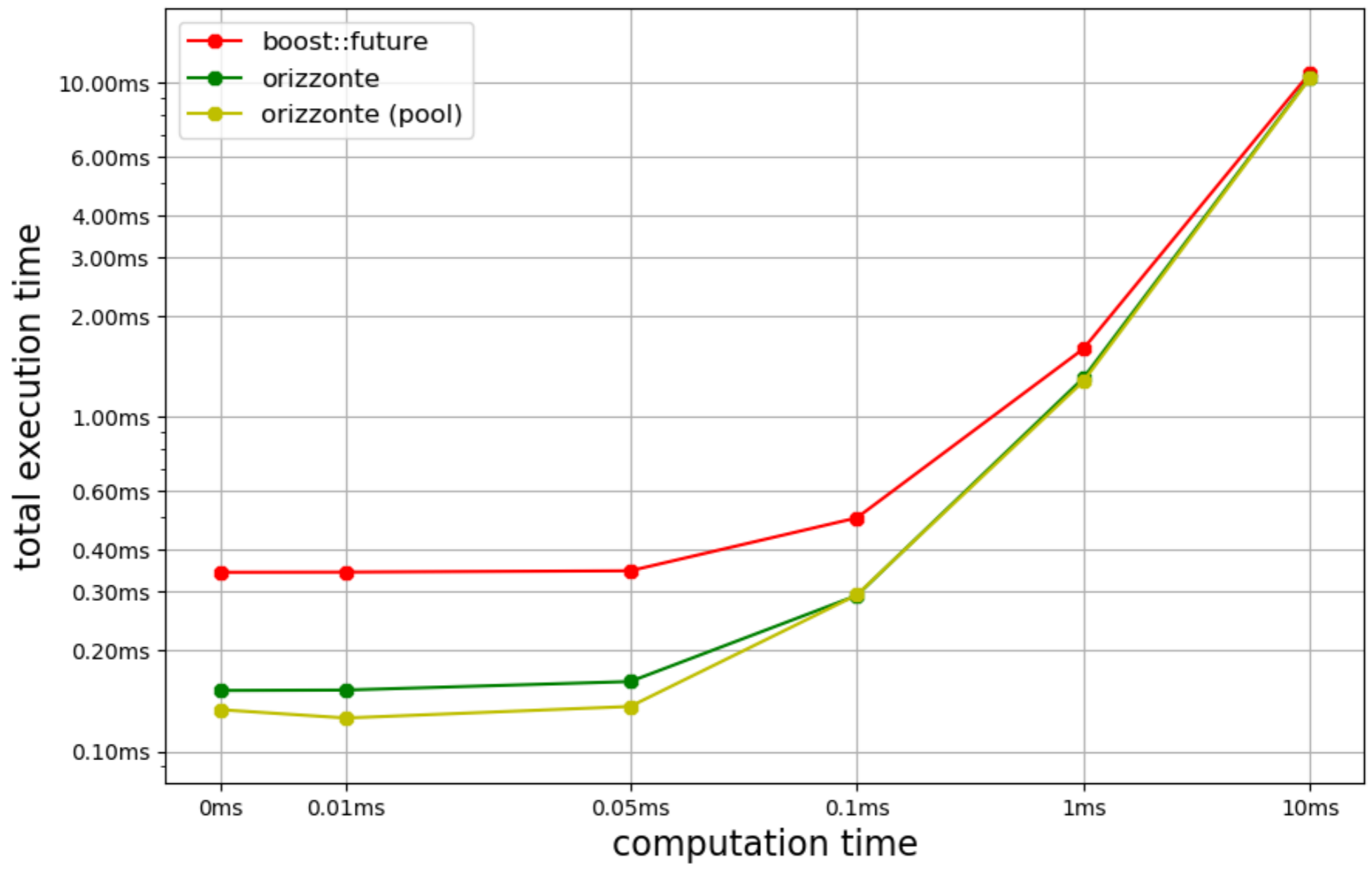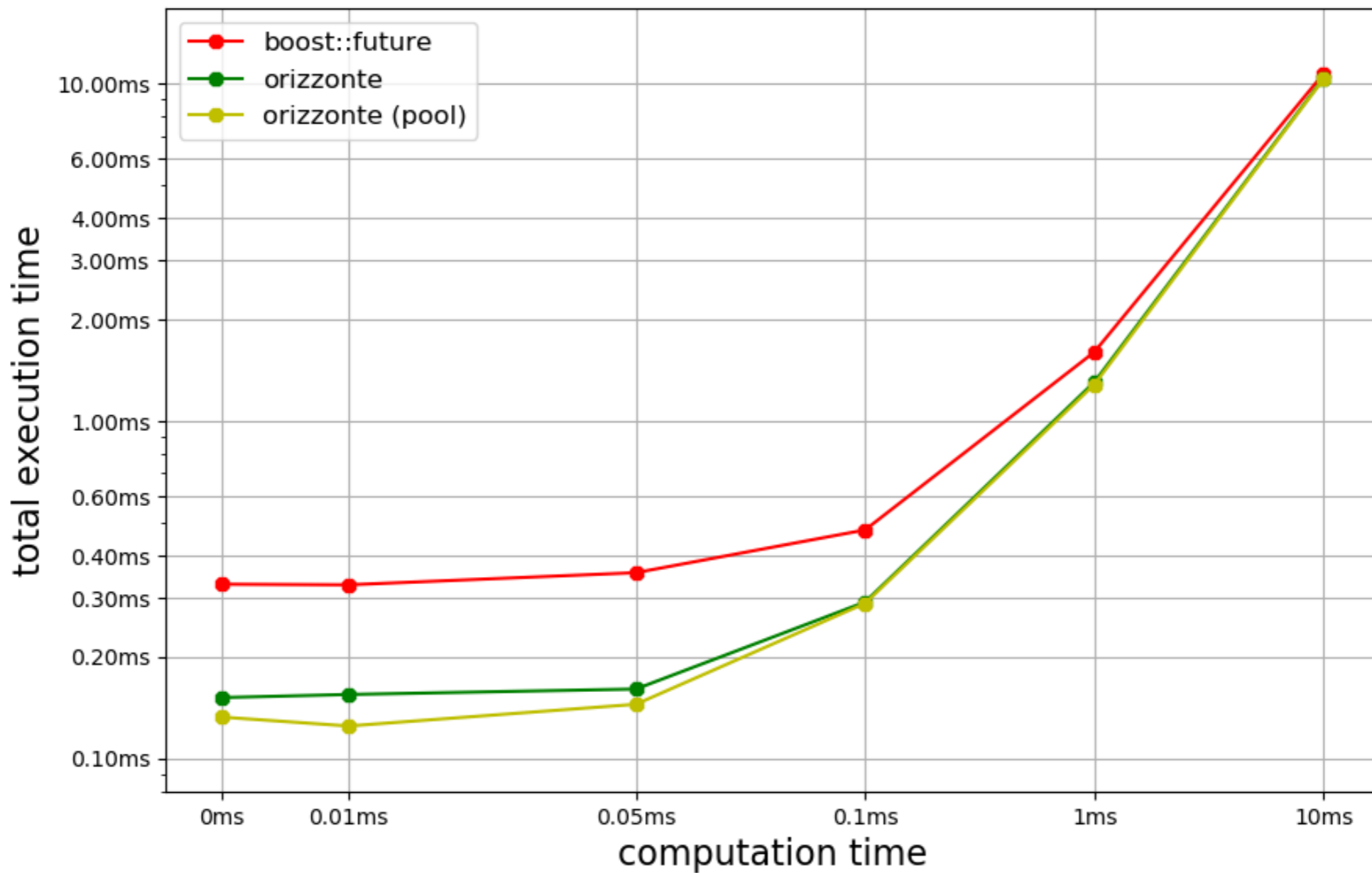# Run-time benchmarks vs `boost::future`

Single node
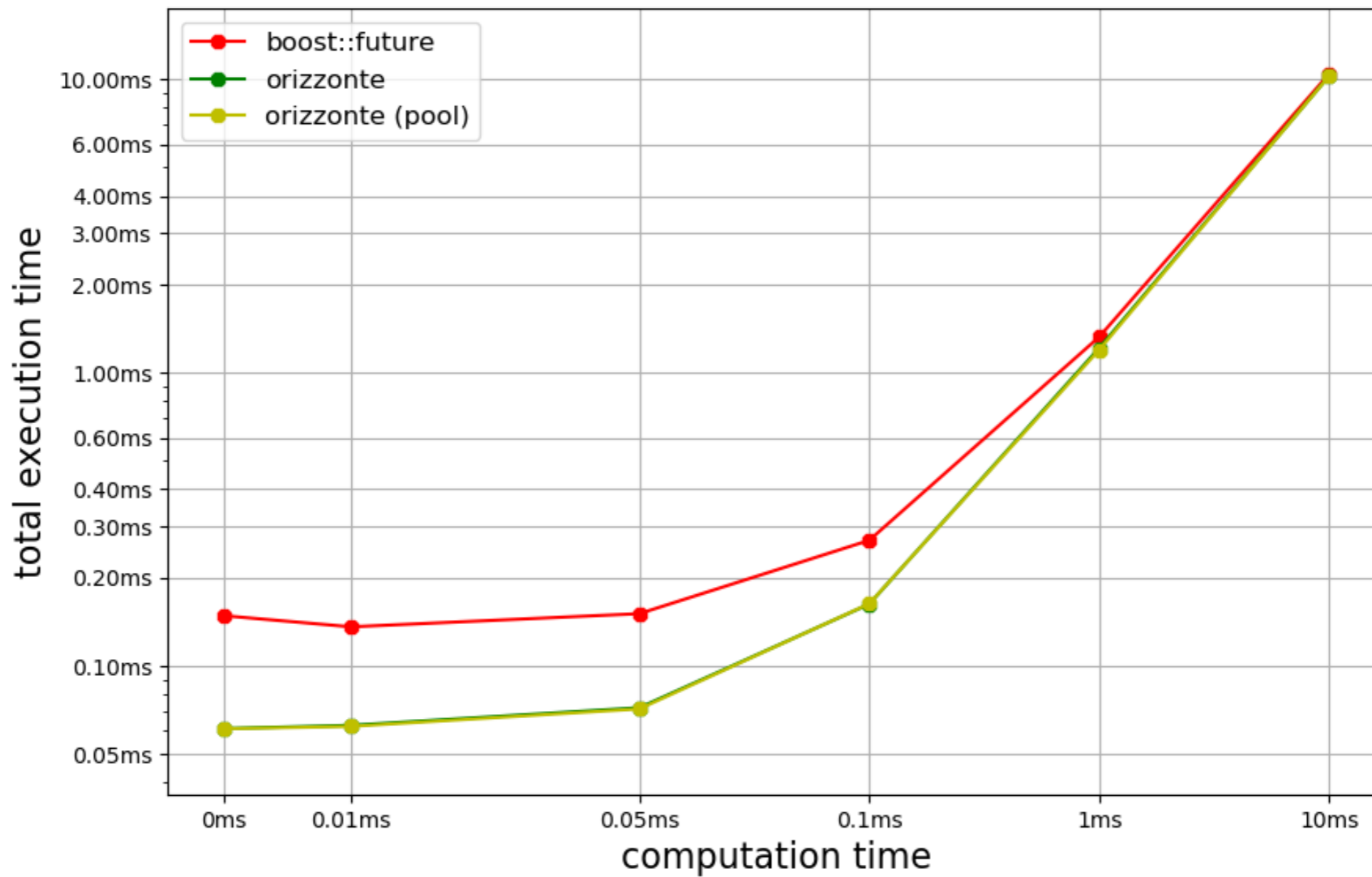
`a.then(b).then(c)`
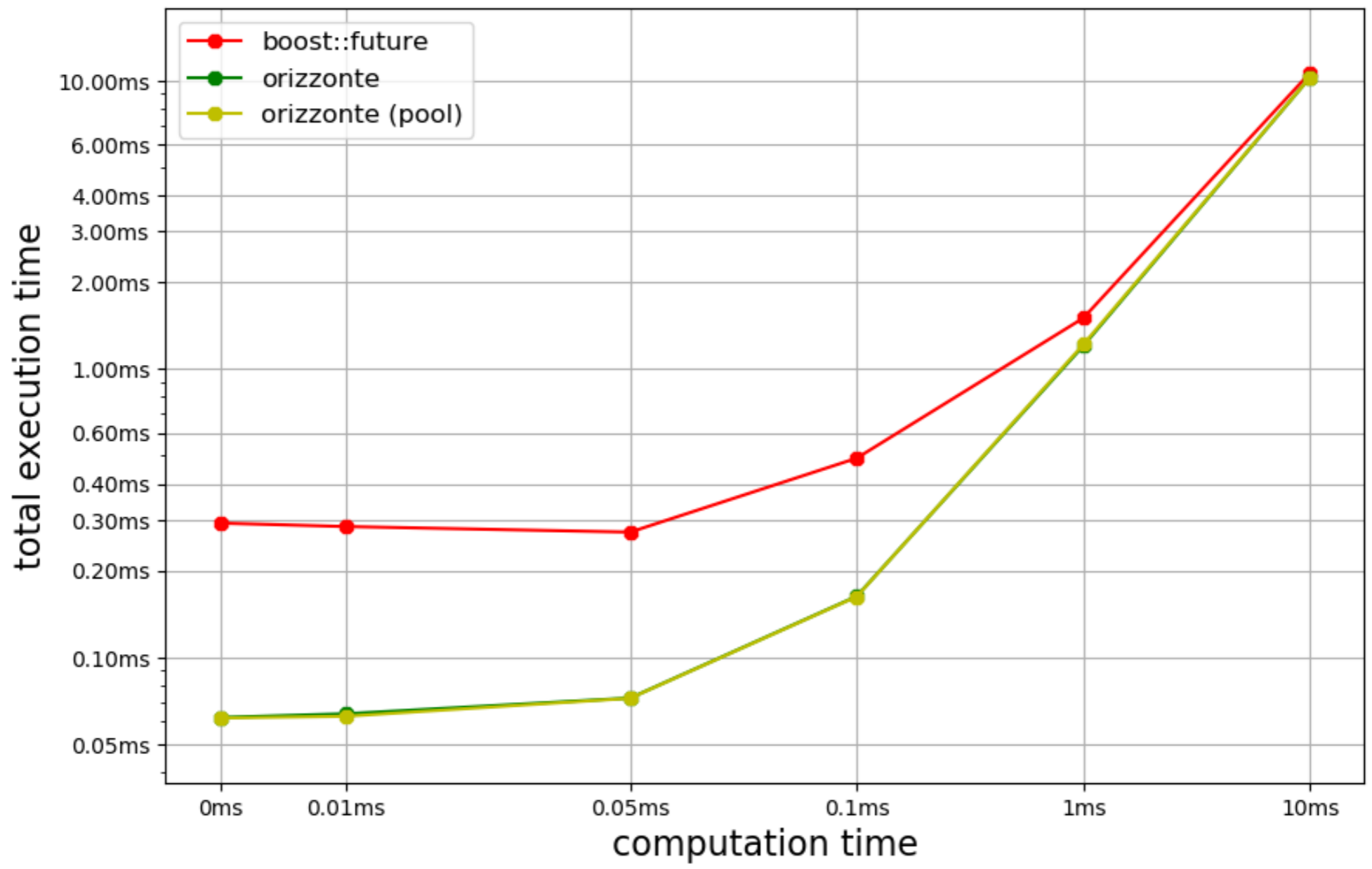
`a.then(b).then(c).then(d).then(e).then(f).then(g).then(h)`
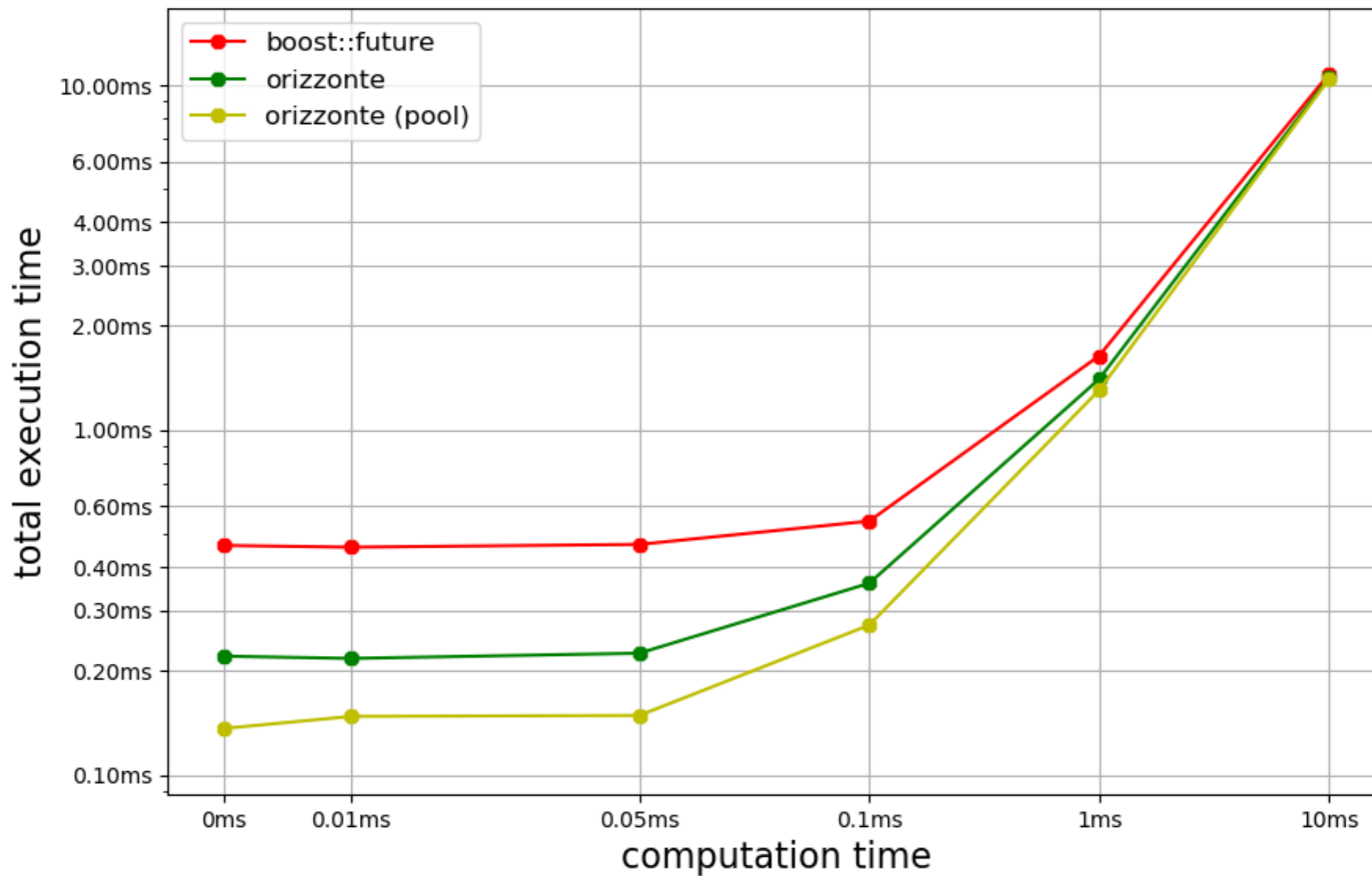
```
all{a0, a1, a2}.then(b0)
```

```
any{a0, a1, a2}.then(b0)
```

`any{a0.then(all{c0, c1, c2}), a1, a2}.then(b0)`

- WIP library available at: https://github.com/SuperV1234/orizzonte

- "*Expression templates*" can be applied to pretty much anything

- Sanitizers are invaluable (especially `ThreadSanitizer`)

- C++17 features **greatly** simplify the implementation

- Future directions/ideas:

    - Automatic cancellation in `any` nodes

    - Composable type-erasing wrapper

    - Exception handling (automatic failure path)

Thanks!

https://vittorioromeo.info

vittorio.romeo@outlook.com

vromeo5@bloomberg.net

@supahvee1234

https://github.com/SuperV1234/orizzonte