Zero-allocation continuations

Bloomberg

Vittorio Romeo
https://vittorioromeo.info
vittorio.romeo@outlook.com

C++::London
11/09/2017

About me

- Developer at **Bloomberg L.P.**
- Modern C++ enthusiast
  - Conference talks **(YouTube playlist)** | **(SkillsMatter)**
  - Video tutorials & articles **(vittorioromeo.info)**
  - Open-source projects **(GitHub/SuperV1234)**

```cpp
// pseudocode
auto f = when_all([]{ return get("cat.com/0.png"); },
                  []{ return get("dog.com/0.png"); })
         .then([](auto p0, auto p1)
         {
             send_email("mail@grandma.com",
                        combine(p0, p1));
         });
```

Utilities such as `when_all` and `.then` allow us to build DAGs of asynchronous computations in an intuitive manner.

- `std::future` will soon let us create asychronous pipelines:
  - **N4538**: "Extensions for concurrency" (TS)
  - Anthony Williams @ ACCU 2016
    "Concurrency, Parallelism and Coroutines"
- `boost::future` has been allowing us to do that for a while

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

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

Notice that a `future` is always returned: this allows composability later on, but means that **type erasure** is happening.

Here's a possible implementation of `then`:

```
template <class T, class F>
auto then(future<T>& parent, std::launch policy, F&& f)
    -> future<result_of_t<F(future<T>&)>>
{
    return std::async(std::launch::async, [
        parent = std::move(parent),
        f = std::forward<F>(f)
    ]
    {
        parent.wait();
        return std::forward<F>(f)(parent);
    });
}
```

`std::async` and type erasure imply allocations and significant overhead.

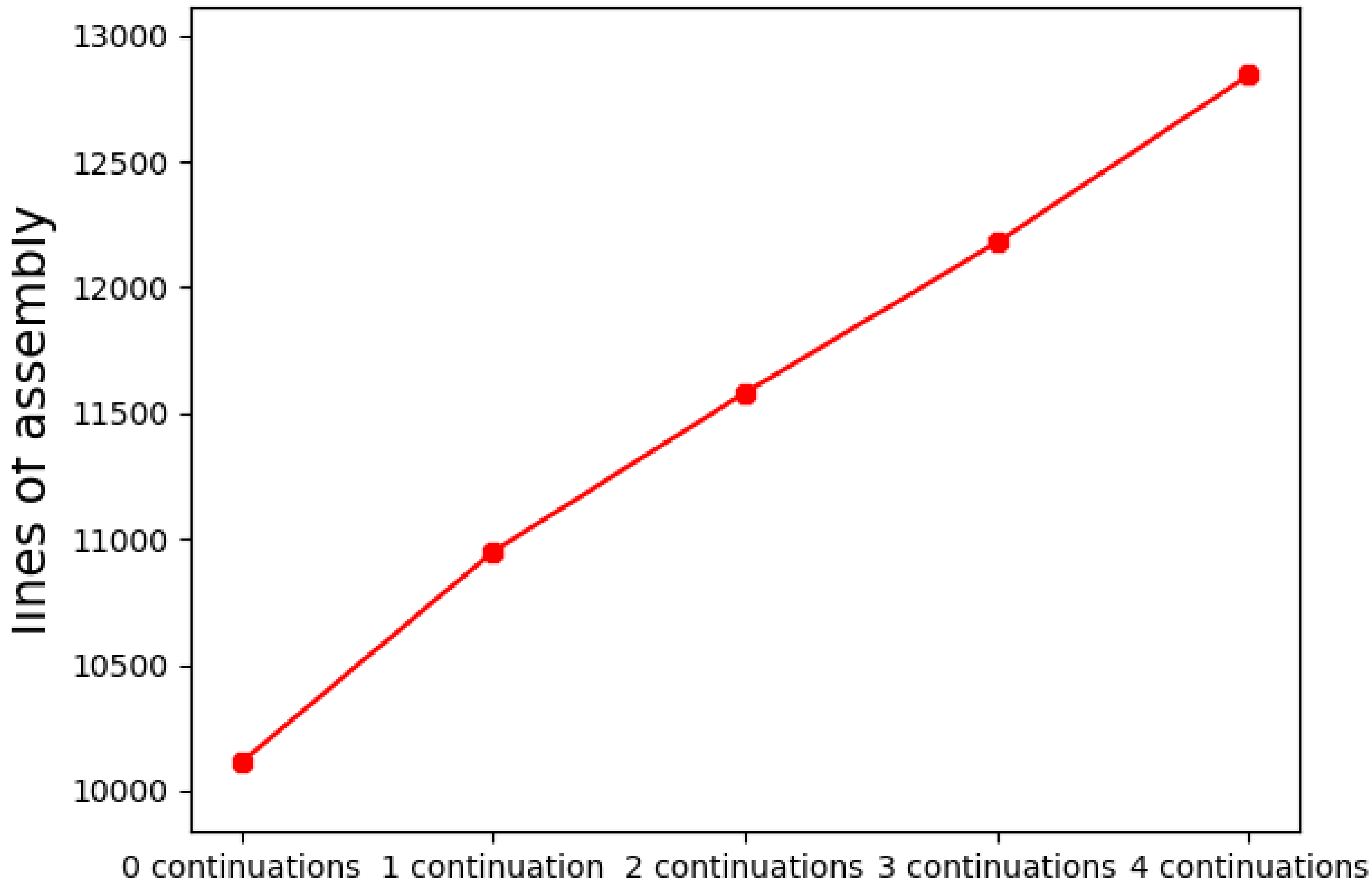
Let's create a small *(and naive)* experiment...

```cpp
// one continuation
int main()
{
    return boost::async([] { return 123; })
        .then([](auto x) { return x.get() + 1; })
        .get();
}
```

```cpp
// two continuations
int main()
{
    return boost::async([] { return 123; })
        .then([](auto x) { return x.get() + 1; })
        .then([](auto x) { return x.get() + 1; })
        .get();
}
```

```cpp
// three continuations
int main()
{
    return boost::async([] { return 123; })
        .then([](auto x) { return x.get() + 1; })
        .then([](auto x) { return x.get() + 1; })
        .then([](auto x) { return x.get() + 1; })
        .get();
}
```

```cpp
// four continuations
int main()
{
    return boost::async([] { return 123; })
        .then([](auto x) { return x.get() + 1; })
        .then([](auto x) { return x.get() + 1; })
        .then([](auto x) { return x.get() + 1; })
        .then([](auto x) { return x.get() + 1; })
        .get();
}
```


lines of assembly
13000
12500
12000
11500
11000
10500
10000
0 continuations
1 continuation
2 continuations
3 continuations
4 continuations

> This is a terrible benchmark that doesn't prove anything!

I *almost completely* agree.

My point is that `std::future` is "too general" in some scenarios, creating unnecessary overhead.

Type-erasure and overhead cannot really be avoided when asynchronous chains are composed at run-time... but they are **unnecessary** in situations like our original example scenario:

```
auto f = when_all([]{ return get("cat.com/0.png"); },
                  []{ return get("dog.com/0.png"); })
         .then([](auto p0, auto p1)
         {
             send_email("mail@grandma.com",
                        combine(p0, p1));
         });
```

The "shape" of the computation chain is known at compile-time - can we do better than `std::future`?

An alternative design

Here's the plan:

- The "shape" of the entire computation chain must be known at compile-time
- No type erasure and zero allocations
- User-friendly `.then` and `when_all` syntax
- The computation chain must be kept alive until completion

> No type erasure and zero allocations

This is possible if we "encode" every step of the computation chain as part of the **type**.

Since the "shape" of the graph will be known at compile-time, this is a reasonable approach.

Let's begin with a simple task - a completely linear computation:

```
int main()
{
    auto f = initiate([]{ return 1; })
                 .then([](int x){ return x + 1; })
                 .then([](int x){ return x + 1; });

    std::move(f).wait_and_get(/* some scheduler */);
}
```

In our final implementation:

- Using a synchronous scheduler will result in 2 lines of assembly code
- Using an asynchronous scheduler will result in 891 lines of assembly code *(that do not depend on the number of continuations)*

That's possible because `decltype(f)` will be a huge type containing all the types of the continuations - **the compiler is able to aggressively inline**.

Key idea: move `*this` in a new "parent" object.

```
template <typename Parent, typename F>
struct computation : Parent, F
{
    computation(Parent&&, F&&);

    template <typename FThen>
    auto then(FThen f_then)
    {
//                vv - why?
        return ::computation{std::move(*this),
                             std::move(f_then)};
    }

    // ...
};
```

```cpp
auto f = initiate([]{ return 1; })                    // (A)
           .then([](int x){ return x + 1; })          // (B)
           .then([](int x){ return x + 1; });         // (C)

std::move(f).wait_and_get(/* some scheduler */);
```

We want the code above to be executed in this order:

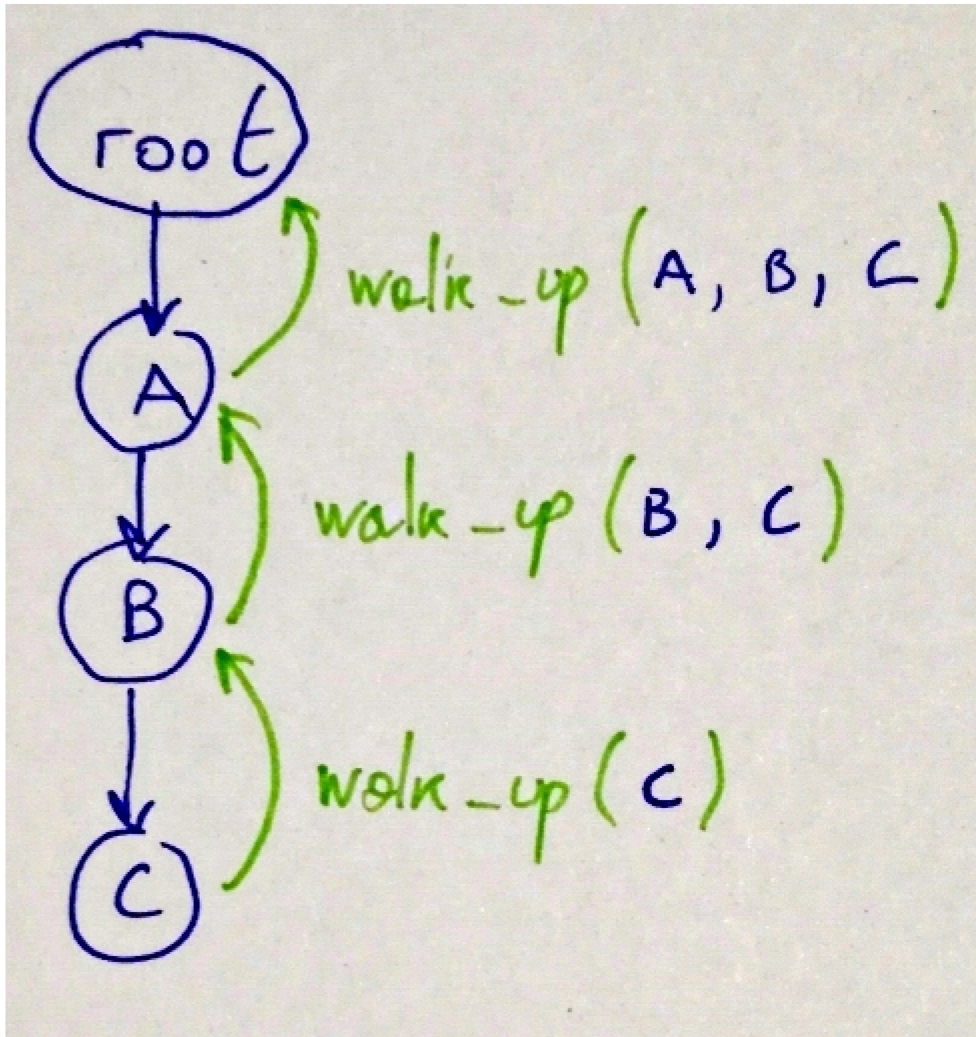
$A \to B \to C$

However, after building the chain, we're left with `f`, which is the entire computation. We need a way of moving up the tree and find the "root" of the computation, so that we can start executing it.

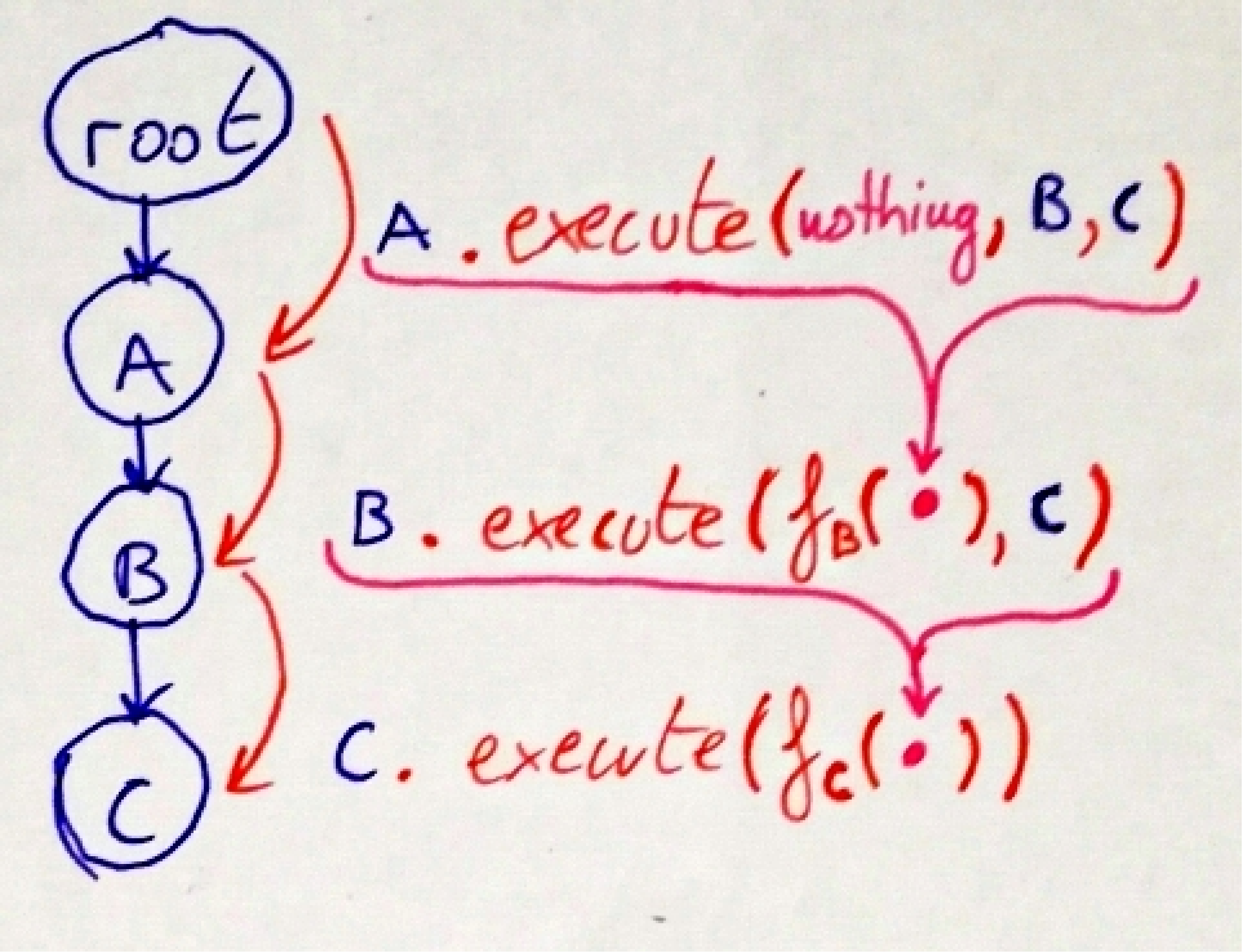
Every node in our graph will provide two operations:

- `x.walk_up( ... )`, which allows us to retrieve the `x`'s parent node and propagate `x` up the call stack
- `x.execute( ... )`, which allows us to schedule `x`'s computation and move towards the leaves of the DAG

```cpp
auto f = initiate([]{ return 1; })                    // (A)
          .then([](int x){ return x + 1; })   // (B)
          .then([](int x){ return x + 1; });  // (C)
```



When we get to the top we can start executing our chain.



Let's begin with the implementation of `initiate`, which takes one `FunctionObject` and being building an asynchronous computation graph:

```
template <typename F>
auto initiate(F&& f)
{
    return root{}.then(FWD(f));
}
```

```
#define FWD(x) ::std::forward<decltype(x)>(x)
```

```
struct root
{
    template <typename Scheduler,
              typename Child, typename... Children>
    void walk_up(Scheduler&& s,
                 Child& c, Children&... cs) &
    {
        s([&]
        {
            c.execute(s, cs...);
        });
    }
};
```

Calling `.walk_up` on the root begins the "descent" towards the leaves of the DAG.

```cpp
template <typename Parent, // Parent in the graph
          typename F>      // Computation function obj
struct node : Parent, F
{
    node(Parent&& p, F&& f);

    template <typename FThen>
    auto then(FThen&& f_then) &&;

    template <typename Scheduler, typename... Children>
    void walk_up(Scheduler&& s, Children&... cs) &;

    template <typename Scheduler, typename Child,
                                  typename... Children>
    void execute(Scheduler&& s, Child&, Children&...) &

    template <typename Scheduler>
    decltype(auto) wait_and_get(Scheduler&& s) &&;
};
```

```cpp
template <typename ParentFwd, typename FFwd>
node< /* ... */ >::node(ParentFwd&& p, FFwd&& f)
    : Parent{FWD(p)}, F{FWD(f)}
{
}
```

Perfectly-forward the *parent node* and the *computation* in the current node instance.

```cpp
template <typename FThen>
auto node< /* ... */ >::then(FThen&& f_then) &&
{
    return ::node{std::move(*this), FWD(f_then)};
}
```

Create and return a new `node`, passing the current node instance as the *parent node* and `f_then` as the *computation*.

```cpp
template <typename Scheduler, typename... Children>
void node</* ... */>::walk_up(Scheduler&& s,
                              Children&... cs) &
{
    auto& as_parent = static_cast<Parent&>(*this);
    as_parent.walk_up(s, *this, cs...);
}
```

Move one node up the graph. Cast `*this` as its `Parent`, and call `.walk_up` on it.

The children are propagated up the call stack using `cs...`. Upon reaching the `root`, we'll have access to all children.

```
template <typename Scheduler,
          typename Child,
          typename... Children>
void node< /* ... */ >::execute(Scheduler&& s,
                              Child& c,
                              Children&... cs) &
{
    static_cast<F&>(*this)();
    c.execute(s, cs...);
}
```

Invokes the current computation. Calls `execute` on its children recursively. The `c, cs...` arguments were obtained from `.walk_up`.

```cpp
template <typename Scheduler>
void node< /* ... */ >::wait_and_get(Scheduler&& s) &&
{
    bool_latch l;

    auto f = std::move(*this).then([&]
    {
        l.count_down(); // Unblock
    });

    f.walk_up(s); // Begin going up
    l.wait();     // Block on the latch
}
```

Attach a continuation to `*this` that will unblock the current thread. Begin going up the graph and block the current thread.

```cpp
auto f = initiate([]{ return 1; })                    // (A)
         .then([](int x){ return x + 1; })   // (B)
         .then([](int x){ return x + 1; });  // (C)
```

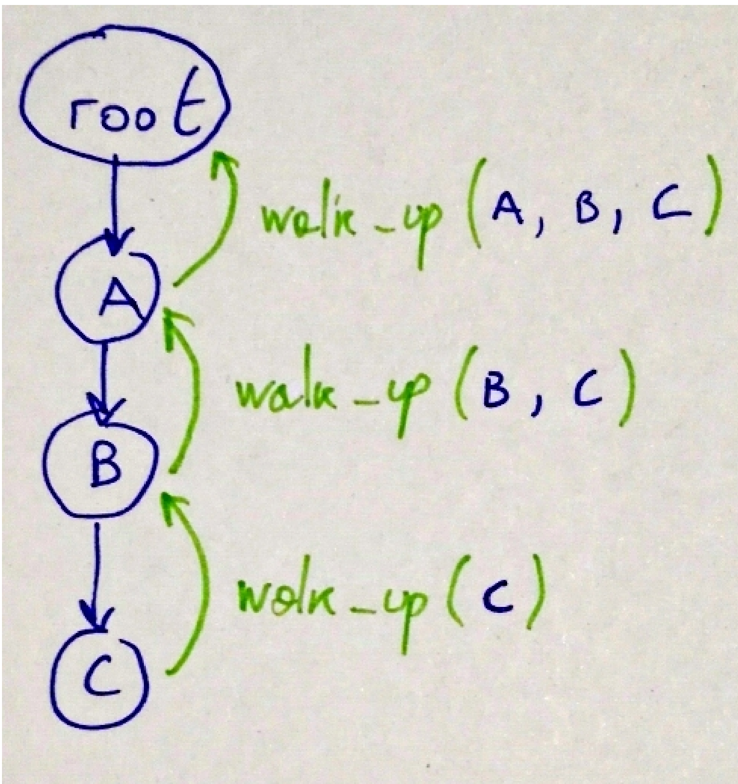
...generates something along the lines of...

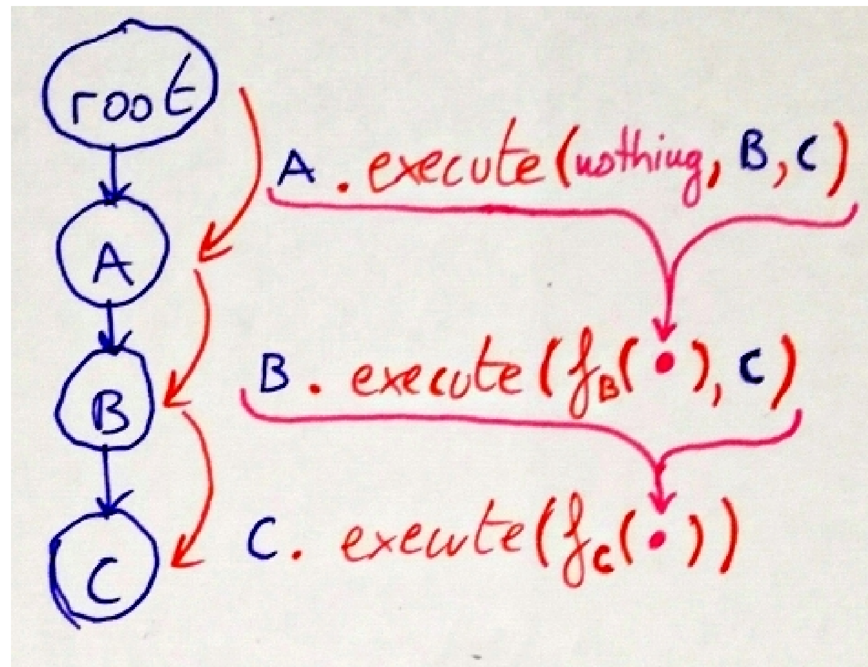
```
C
├─────B // parent of C
│     ├─────A // parent of B
│     │     │
│     │     ├────root // parent of A
│     │     └────F0   // computation of A
│     │
│     └─────F1 // computation of B
│
└─────F2 // computation of C
```

```
C<
    B<
        A<root, F0>,
        F1
    >,
    F2
>
```

Everything is embedded into the type.

```
C<B<A<root, F0>, F1>, F2>
```



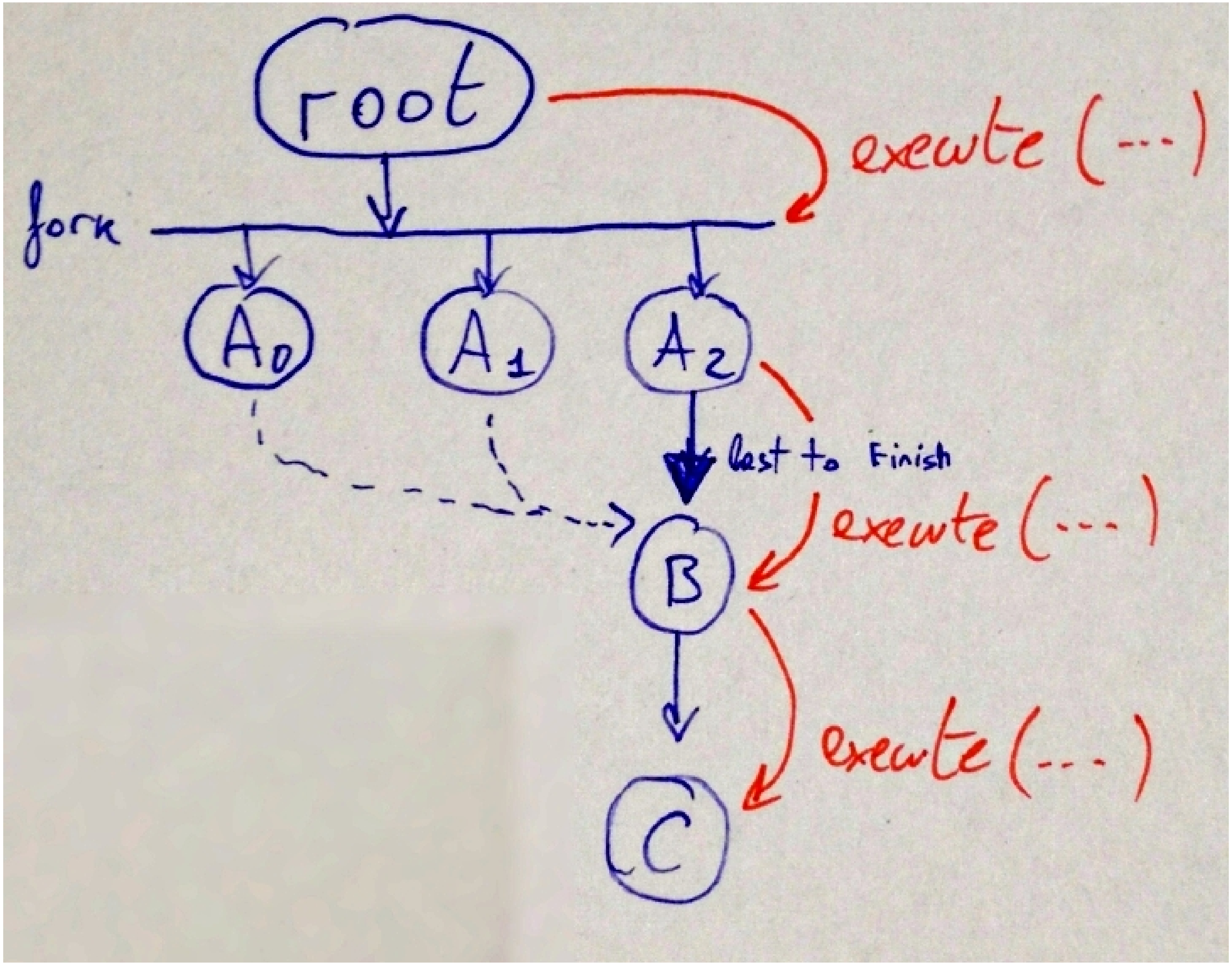


Key takeaways:

- Given a DAG of asynchronous operations known at compile-time, it is possible to create a **zero-allocation** & **zero-type-erasure** future-like class
- The idea is to **encode the entire graph in the type system**
- That can be done through `node` classes that **store their parent** and **allow movement up/down the graph**
- The graph can be created by providing `x.then`, which **moves `x` into the newly-created `node`**
- This technique can be generalized to `when_all` or `when_any`

`when_all`

The technique generalizes to arbitrary fork/join graphs:



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

Semantics:

- Schedule `b` and `c` on separate threads, run `a` on the current thread
- When `a`, `b`, and `c` are done, run `d` on the last living thread

The plan:

- Create a new `when_all` node type, which stores a parent and N computations
- The node will also store an `atomic<int>` initialized to N
- When executed, the N computation will be run in parallel
- Each computation will decrement the `atomic<int>` upon completion
- The last computation to finish will also execute the next `node`

```cpp
template <typename Parent, // Parent in the graph
          typename... Fs>  // Computation function objs
struct when_all : Parent, Fs...
{
    std::atomic<int> _left{sizeof...(Fs)};

    when_all(Parent&& p, Fs&&... fs);

    template <typename FThen>
    auto then(FThen&& f_then) &&;

    template <typename Scheduler, typename... Children>
    void walk_up(Scheduler&& s, Children&... cs) &;

    template <typename Scheduler, typename Child,
                                  typename... Children>
    void execute(Scheduler&& s, Child&, Children&...) &

    // ...
};
```

All of these...

- `when_all::then`
- `when_all::walk_up`
- `when_all::wait_and_get`

...are identical to `node`.

The *cool* stuff happens in `when_all::execute`.

```cpp
template <typename Scheduler,
          typename Child, typename... Children>
void when_all< /* ... */ >::execute(
    Scheduler&& s, Child& c, Children&... cs) &
{
    compile_time_for(auto f : fs...)
    {
        auto g = [&]{ f();
                      if(--left == 0) c.execute(...); };

        if constexpr(f /* is last in */ fs...)
        {
            g();
        }
        else
        {
            s([g = std::move(g)]{ g(); });
        }
    }
}
```

```
template <typename Scheduler,
          typename Child, typename... Children>
void when_all< /* ... */ >::execute(
    Scheduler&& s, Child& c, Children&... cs) &
{
    ([&](auto f)
     {
         auto g = [&]
         {
             f();
             if(--_left == 0) // last to complete
             {
                 c.execute(s, std::move(_out), cs...);
             }
         };

         if constexpr(is_last<Fs...>(f)) { g(); }
         else { s([g = std::move(g)]{ g(); }) };

     }(static_cast<Fs>(*this)), ...);
}
```

Key takeaways:

- Parallelism is implemented with **a new node type** that uses an **atomic counter** to keep track of the number of active computations
- Computations can be cleverly scheduled to **minimize thread waste**
- C++17 language features are **lifesavers**

```cpp
auto f = initiate([]{ cout << "c++ ";    return 1; },
                  []{ cout << "london "; return 2; })
        .then([](auto t)
        {
            auto [a, b] = t;
            assert(a + b == 3);
            return a + b;
        })
        .then([](auto x)
        {
            assert(x == 3);
            cout << "rocks!\n";
        });

std::move(f).wait_and_get(world_s_best_thread_pool{});
```

https://wandbox.org/permlink/vw3CvuV1CX74V1xY

> But sorry Vittorio - what about...
>
> - return values
> - lifetime management for `when_all` arguments
> - "fun" stories about subtle bugs
> - metaprogramming utilities to simulate "regular `void`"
> - `when_any`
>
> ...?

It's hard. Find me later!

Articles on *vittorioromeo.info*:

- zero allocation continuations - part 1
- zero allocation continuations - part 2
- zero allocation continuations - part 3

These slides:

- https://github.com/SuperV1234/cpplondon

Last version of the code:

- github.com/SuperV1234/vittorioromeo.info/.../p2_parallel.cpp

Thanks!

- https://vittorioromeo.info
- vittorio.romeo@outlook.com
- vromeo5@bloomberg.net
- @supahvee1234