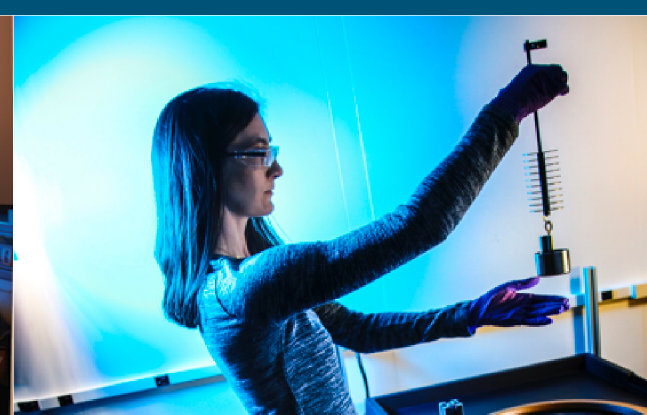


# facebook

# A Universal Abstraction for Async



*PRESENTED BY*

Eric Niebler and David S. Hollman



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Goals for Async Interfaces

C++ needs an async abstraction:

- That is composable
- That has low abstraction overhead
- That works with coroutines, fibers, and threads, etc.
- That is extensible to multiple execution environments (both concurrent and parallel)

# Disclaimer

This talk doesn't represent the official views of WG21.  
It is merely sketching the ideas behind some recent proposals.



# Disclaimer 2

This talk makes use of C-style casts.

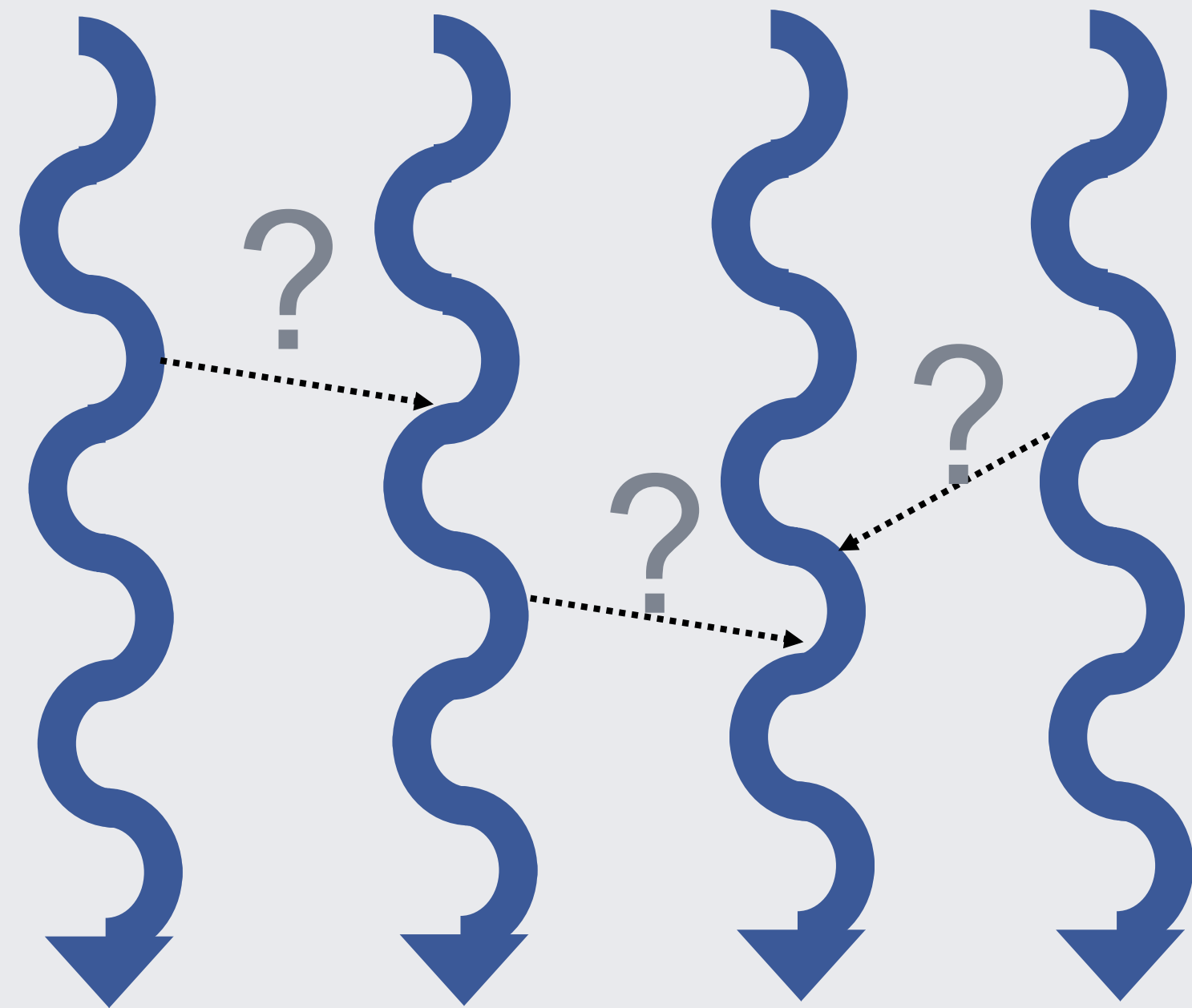
Viewer discretion is advised.

# I. Background and Introduction

# Understanding Parallelism vs. Concurrency

Spoiler alert: They're not the same thing.

# Concurrency vs. Parallelism

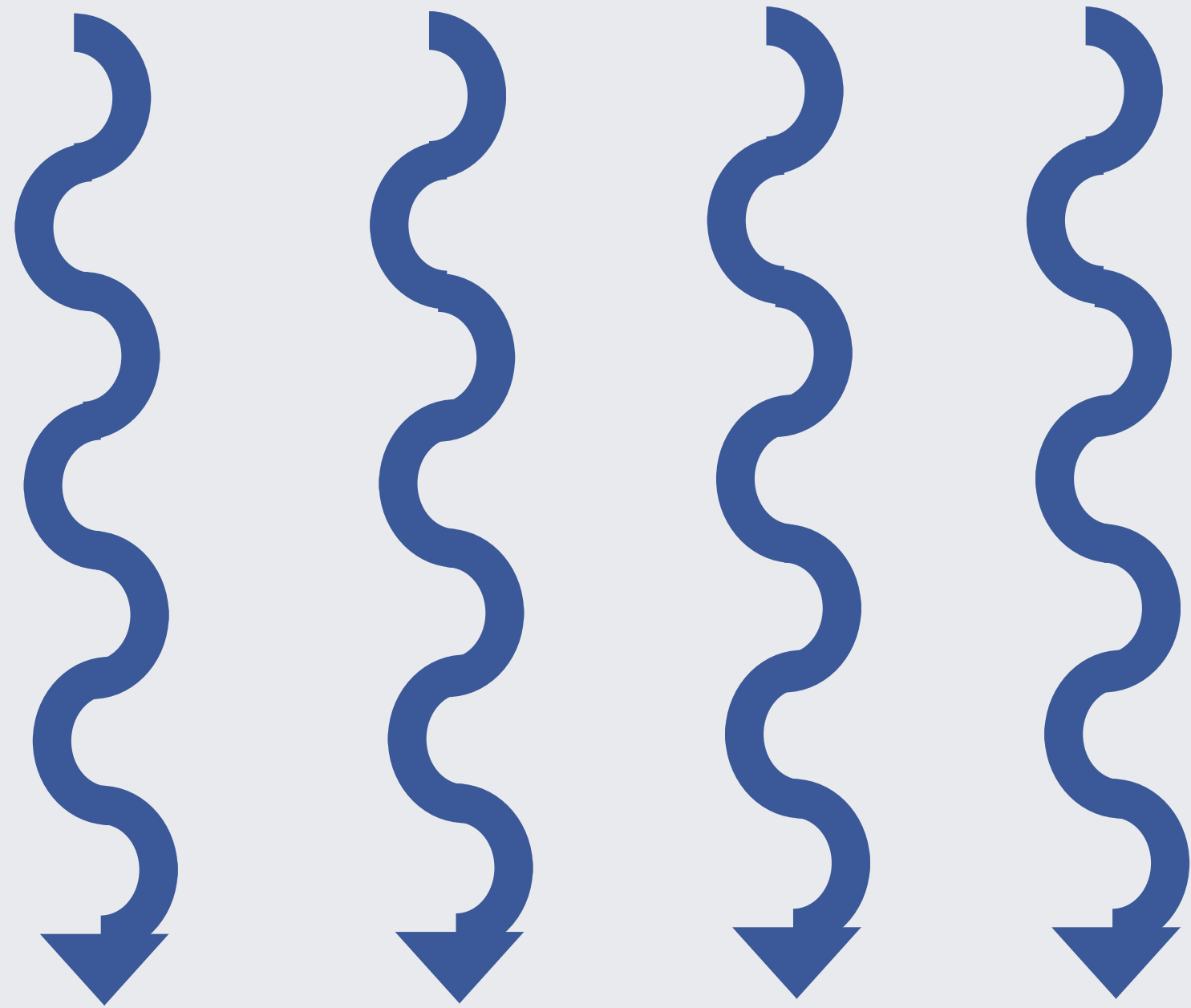


Concurrency:

Multiple logical threads of execution with unknown inter-task dependencies.



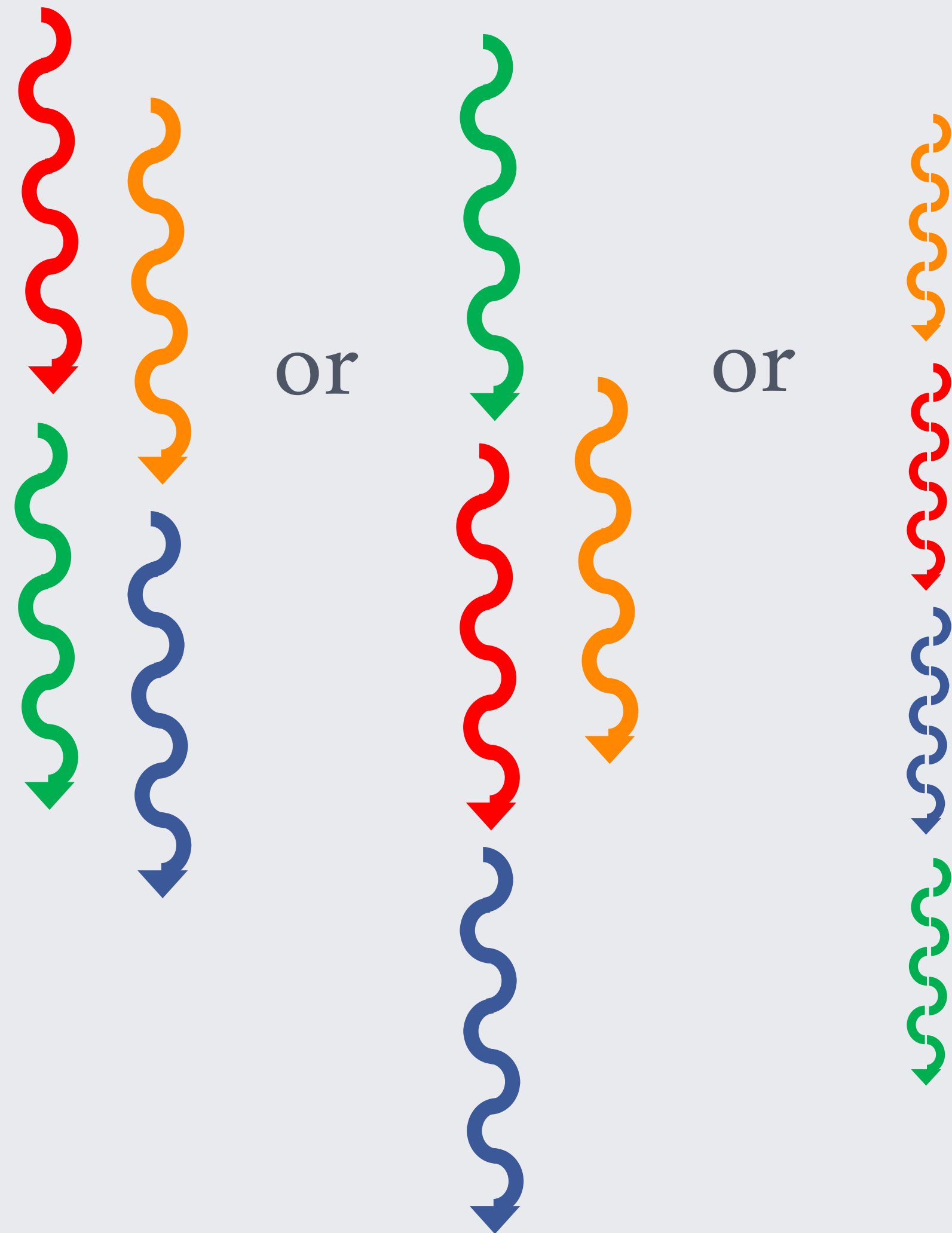
# Concurrency vs. Parallelism



## Parallelism:

Multiple logical threads of execution with no inter-task dependencies.

# Concurrency vs. Parallelism



Parallelism:

Scheduler has the freedom to use the fastest execution order.

# Concurrency, by Example

```
// variable accessible to both workers:
```

```
std::atomic<int> x = { 0 };
```

```
// Worker A:
```

```
while(x.load() == 0) { /* yield */ }
```

```
cout << "Hello" << endl;
```

```
// Worker B:
```

```
x.store(1);
```

- This program is not *guaranteed* to ever print Hello unless Worker A and Worker B are executed on agents with a **concurrent** forward progress guarantee.
- Generally speaking, concurrency imposes extra requirements on the scheduler.

# Parallelism, by Example

```
// variable accessible to both workers:
```

```
int x = 0;
```

```
// Worker A:
```

```
// (not an atomic operation)
```

```
x += 1;
```

```
// Worker B:
```

```
// (not an atomic operation)
```

```
x += 1;
```

- This program can result in  $x == 1$  or  $x == 2$  if Worker A and Worker B are executed in **parallel**.
- Parallelism is a ***contract*** that grants extra freedom to the scheduler (and imposes extra requirements on the *user*).

# Parallelism and Concurrency are Opposites

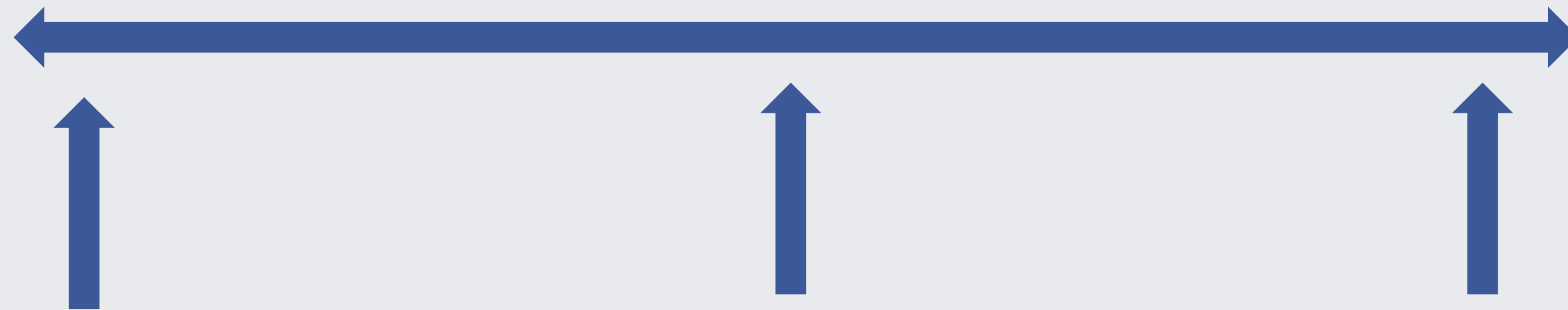
Less freedom for the scheduler (usually because of missing information)

More information provided to the scheduler (thus more freedom)

Concurrency

Serial

Parallelism



# Concurrency is a Stronger Scheduling Guarantee than Serial

```
std::atomic<int> x = { 0 };
```

```
// Worker A:
```

```
while(x.load() == 0) { /* yield */ }  
cout << "Hello" << endl;
```

```
// Worker B:
```

```
x.store(1);
```

```
// Serial program:
```

```
std::atomic<int> x = { 0 };  
while(x.load() == 0) { /* yield */ }  
cout << "Hello" << endl;  
x.store(1);
```

When you use *serial* execution for the program with *concurrent* requirements, it (obviously) never prints Hello.

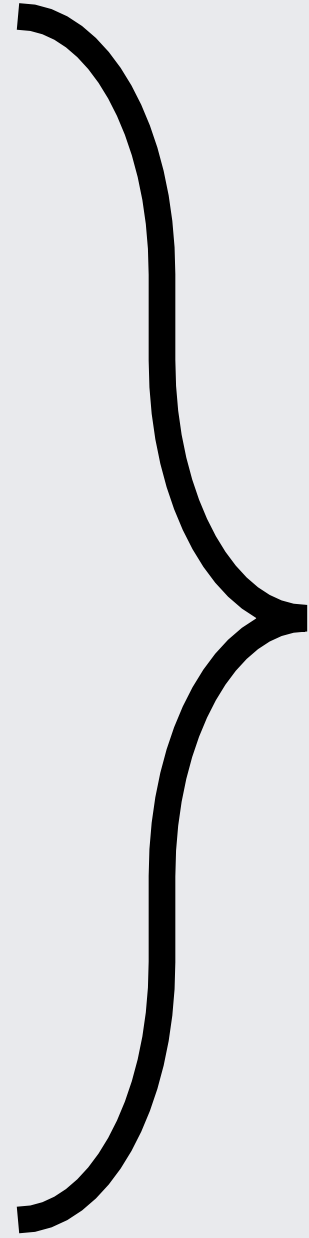


# Parallelism is a Weaker Scheduling Requirement than Serial

```
int x = 0;
```

```
// Worker A:  
// (not an atomic operation)  
x += 1;
```

```
// Worker B:  
// (not an atomic operation)  
x += 1;
```



```
// Serial program:  
int x = 0;  
x += 1;  
x += 1;
```

When you use *serial* execution for the program with *parallel* requirements, it (obviously) results in `x == 2`.

# Parallelism is "More Universal"

- When you use *concurrent* features to express *parallelism*, you end up with unreasonable overheads.
- The programming model is not *restrictive* enough for the compiler or runtime system to avoid these overheads.

# Why are the parallel algorithms fast?

- Because they let the user communicate to the scheduler critical information about (the lack of) cross-task dependencies.
- In other words, it communicates the full structure of the algorithm's *task graph* to the scheduler.

## II. Senders and Receivers

Why are standard futures slow?

# Why are Futures slow?

```
future<int> async_algo() {  
    promise<int> p;  
    auto f = p.get_future();  
    thread t { [p = move(p)]() mutable {  
        int answer = // compute!  
        p.set_value(answer);  
    }};  
    t.detach();  
    return f;  
}
```

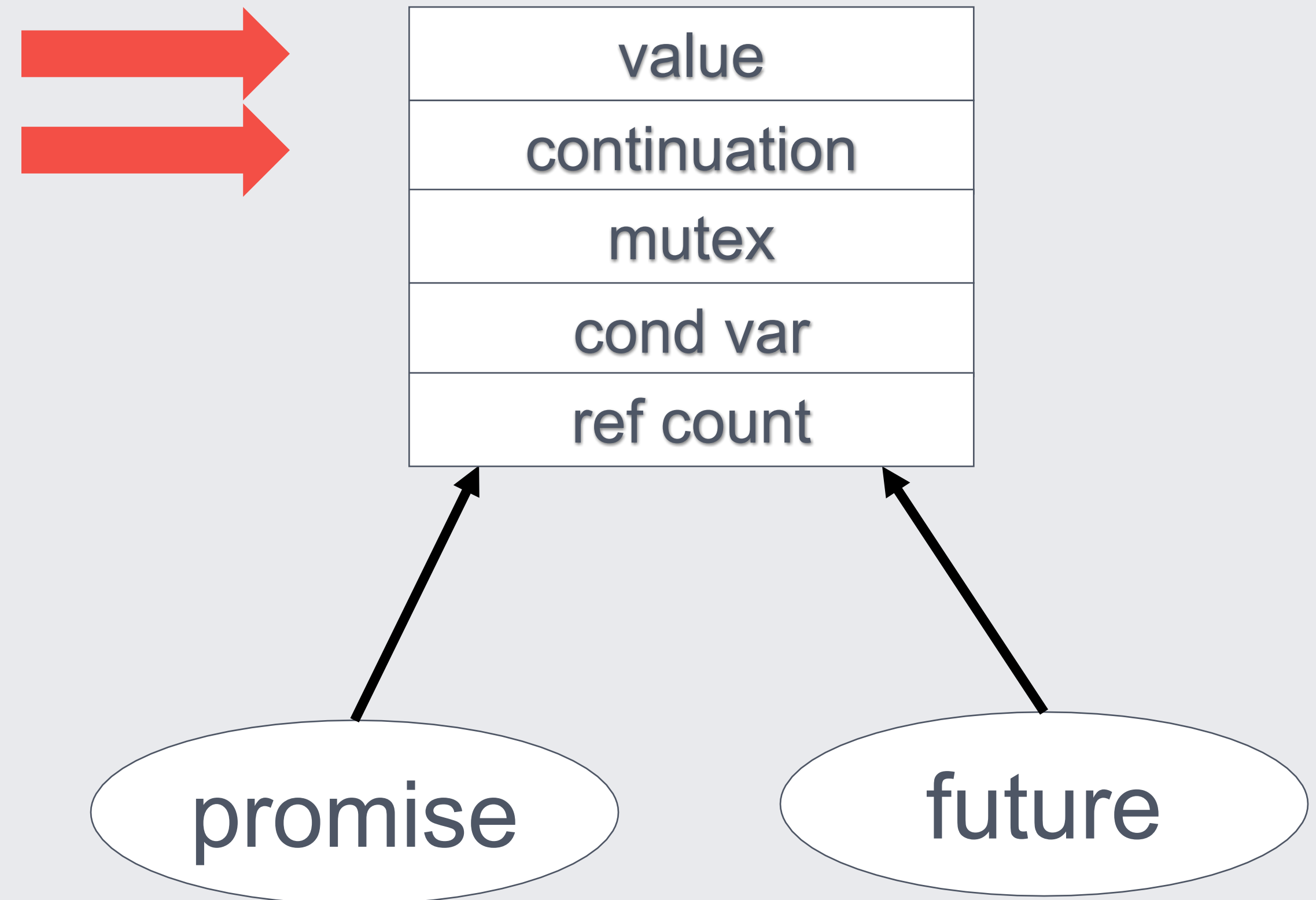
```
int main() {  
    auto f = async_algo();  
    auto f2 = f.then([](int i) {  
        return i + rand();  
    });  
    printf("%d\n", f2.get());  
}
```



# Why are Futures slow?

```
future<int> async_algo() {  
    promise<int> p;  
    auto f = p.get_future();  
    thread t { [p = move(p)]() mutable {  
        int answer = // compute!  
        p.set_value(answer);  
    }};  
    t.detach();  
    return f;  
}
```

```
int main() {  
    auto f = async_algo();  
    auto f2 = f.then([](int i) {  
        return i + rand();  
    });  
    printf("%d\n", f2.get());  
}
```



How successful would the  
STL be if iterators all did  
**allocation, synchronization,**  
**and**  
**type-erasure?**



# A simple observation...

```
template <class Cont>
future<int> async_algo(Cont c) {
    promise<int> p;
    auto f = p.get_future();
    thread t { [p = move(p), c]() mutable {
        int answer = // compute!
        p.set_value(c(answer));
    }};
    t.detach();
    return f;
}
```

This calling code knows  
the continuation at the  
point it calls the  
algorithm!

```
    algo(
        i) {
        return i + rand();
    });
    printf("%d\n", f.get());
}
```

# A less simple observation...

- Passing in a continuation avoids (some) synchronization overhead because it removes the race on reading and writing the continuation.
- We can achieve the same result by starting async work suspended and letting the caller add the continuation *before* launching the work.

# A less simple observation...

```
auto async_algo() {  
    return [](auto p) {  
        thread t { [p = move(p)]() mutable {  
            int answer = // compute!  
            p.set_value(answer);  
        };  
        t.detach();  
    };  
}
```

Defer the thread launch -- return a *function* that takes a promise instead.

The function returned from `async_algo` is like a *lazy future*..

```
int main() {  
    auto f = async_algo();  
    auto f2 = then(f, [](int i) {  
        return i + rand();  
    });  
    // ...  
}
```



# then() is just an algorithm

```
auto then(auto task, auto fun) {  
    return [=](auto p) {  
        struct _promise {  
            decltype(p) p_;  
            decltype(fun) fun_;  
            void set_value(auto ...vs) { p_.set_value(fun_(vs...)); }  
            void set_exception(auto e) { p_.set_exception(e); }  
        };  
        task(_promise{p, fun});  
    };  
}
```

then() returns a lazy future that applies a function to the value produced by another lazy future.

Lazy futures expect promise-like things.

```
int main() {  
    auto f = async_algo();  
    auto f2 = then(f, [](int i) {  
        return i + rand();  
    });  
    // ...  
}
```



# Use then() to compose lazy futures

```
auto async_algo() {  
    return [](auto p) {  
        thread t { [p = move(p)]() mutable {  
            int answer = // compute!  
            p.set_value(answer);  
        };  
        t.detach();  
    };  
}
```

```
struct sink {  
    void set_value(auto... vs) {}  
    void set_exception(auto e) {  
        std::terminate();  
    }  
};
```

Oops, printf in wrong thread!

Oops, main no longer  
blocks!

```
int main() {  
    auto f = async_algo();  
    auto f2 = then(f, [](int i) {  
        return i + rand();  
    });  
    auto f3 = then(f2, [](int j) { // ???  
        printf("%d\n", j);  
    });  
    f3( sink{} ); // Launch  
}
```

# Blocking is just an algorithm, too

```
template< class T, class Task >
T sync_wait(Task task) {
    // Define some state:
    _state<T> state;

    // launch the operation:
    task(_promise<T>{&state});

    // wait for it to finish:
    {
        auto lk = unique_lock{state.mtx};
        state.cv.wait(lk, [&state]{
            return state.data.index() != 0; });
    }
    // throw or return the result:
    if (state.data.index() == 1)
        rethrow_exception( get<1>(state.data) );

    return move(get<2>(state.data));
}
```

```
template< class T >
struct _state {
    mutex mtx;
    condition_variable cv;
    variant<monostate, exception_ptr, T> data;
};
```

```
template< class T >
struct _promise {
    _state<T>* pst;
    template <int I>
    void _set(auto... xs) {
        auto lk = unique_lock{pst->mtx};
        pst->data.template emplace<I>(xs...);
        pst->cv.notify_one();
    }
    void set_value(auto... vs) { _set<2>(vs...); }
    void set_exception(auto e) { _set<1>(e); }
};
```

# Use `sync_wait()` algo to block

```
auto async_algo() {  
    return [](auto p) {  
        thread t { [p = move(p)]() mutable {  
            int answer = // compute!  
            p.set_value(answer);  
        };  
        t.detach();  
    };  
}
```

```
int main() {  
    auto f = async_algo();  
    auto f2 = then(f, [](int i) {  
        return i + rand();  
    });  
    printf("%d\n", sync_wait<int>(f2));  
}  
  
});  
f3( sink{} ); // Launch  
}
```

# Separation of concerns

Why is thread creation the responsibility of `async_algo()`?

`new_thread()` is an “executor.”

```
auto new_thread() {  
    return [](auto p) {  
        thread t { [p = move(p)]() mutable {  
            p.set_value();  
        }};  
        t.detach();  
    };  
}
```

```
auto async_algo(auto task) {  
    return then(task, [] {  
        int answer = // compute!  
        return answer;  
    });  
}
```

```
int main() {  
    auto f = async_algo(new_thread());  
    auto f2 = then(f, [](int i) {  
        return i + rand();  
    });  
    printf("%d\n", sync_wait<int>(f2));  
}
```

# Lazy future advantages

- Async tasks can be composed...
  - ... without allocation
  - ... without synchronization
  - ... without type-erasure
- Composition is a generic algorithm
- Blocking is a generic algorithm

# Generic is as Generic does

```
template <class P, class E = exception_ptr>
concept HasReceiverSignals =
    requires (P& p, E&& e) {
        p.set_error( (E&&) e );
        p.set_done();
    };

```

```
template <class P, class... Vs>
concept Receiver =
    HasReceiverSignals<P> &&
    Invocable<P, Vs...>;

```

Called by a lazy future in response to a request for cancellation.



# Generic is as Generic does

```
template <class F>
concept Sender =
    is_sender_v<decay_t<F>>;

template <class F, class R>
concept SenderTo =
    Sender<F> &&
    HasReceiverSignals<R> &&
    requires (F&& f, R&& r) {
        submit( forward<F>(f), forward<R>(r) );
    };
};
```

# III. Sender/Receiver and Coroutines

# Coroutines and callbacks

```
task<int> async_helper();

task<void> async_algorithm() {
    // ...
    int result = co_await async_helper();
    // use result
    printf("%d\n", result);
}
```

Everything after a  
co\_await or a  
co\_yield  
expression is  
implicitly a  
callback.

If suspended coroutines are callbacks,  
and if callbacks are Receivers, then...

Coroutines are Receivers  
and Awaitables are Senders

# (Some) Senders are Awaitable

```
// In a future version of C++, perhaps?  
namespace std { inline namespace awaitable_senders {  
    template <Sender S>  
    auto operator co_await(S&& s) {  
        return _awaiter_sndr{(S&&)s};  
    };  
}
```



Must be find-able by  
ADL

# (Some) Senders are Awaitable

```
struct DuMb_SeNdEr : std::sender_of<int> {  
    void submit( Receiver<int> auto r ) {  
        r(42);  
    }  
};
```

```
coro_task<int> async_algo( Sender auto s ) {  
    int the_answer = co_await s;  
    assert( the_answer == 42 );  
    co_return the_answer;  
}
```

```
int main() {  
    int res = sync_wait<int>( async_algo( DuMb_SeNdEr{} ) );  
}
```

Senders can be  
co\_awaited in a  
coroutine.

Awaitables (coro\_task)  
can be treated as  
Senders!

# (All) Awaitables are Senders

All awaitable types satisfy the requirements of the Sender concept

```
// A simple co-awaitable type:
struct _awaitable {
    friend auto operator co_await(_awaitable) {
        return ...;
    }
};

// A simple receiver:
struct _receiver {
    void operator()(auto...);
    void set_error(exception_ptr);
    void set_done();
};

int main() {
    // OK, can use awaitables as senders:
    submit( _awaitable{}, _receiver{} );
}
```

# (All) Awaitables are Senders

// Make all awaitables senders:

```
template <Awaitable A, Receiver<await_result_t<A>> R>
void submit(A awaitable, R to) noexcept {
    try {
        invoke([](A a, R&& r) -> oneway_task {
            R rCopy((R&&) r);
            try {
                rCopy(co_await (A&&) a);
            }
            catch (...) {
                rCopy.set_error(current_exception());
            }
        }, (A&&) awaitable, (R&&) to);
    }
    catch (...) {
        to.set_error(current_exception());
    }
}
```

```
struct [[maybe_unused]] oneway_task {
    struct promise_type {
        oneway_task get_return_object() noexcept { return {}; }
        suspend_never initial_suspend() noexcept { return {}; }
        suspend_never final_suspend() noexcept { return {}; }
        void return_void() noexcept {}
        void unhandled_exception() noexcept { std::terminate(); }
    };
};
```



## IV. Building on Sender/Receiver

# Building on Sender/Receiver

- Higher-level functionality can be built efficiently on top of Sender/Receiver:
  - Generic algorithms: `sync_wait`, `wait_all`, `wait_any`, etc.
  - Promises and Futures
  - Channels
  - Async Ranges and Reactive streams

# Example: Futures

We can build eager futures on top of lazy Senders with no overhead beyond that which is inherent in eager execution; *i.e.*, allocation and synchronization.

# Futures

```
template <class T>
struct MyFuture {
private:
    shared_ptr<_my_state<T>> _st =
        make_shared<_my_state<T>>();
public:
    template <SenderOf<T> S>
    explicit MyFuture(S&& src) {
        ((S&&)src).submit(
            _st->make_receiver());
    }
    T get() && {
        return move(*_st).get();
    }
};
```

# Futures

```
template <class T>
struct MyFuture {
    private:
        mutable shared_ptr<_my_state<T>> _st =
            make_shared<_my_state<T>>();
    public:
        template <SenderOf<T> S>
        explicit MyFuture(S&& src) {
            T get((S&&)src).submit(
                _st->make_receiver());
            _cv }
        if (!T get() && {
            return move(*_st).get();
        }
    };
};

_my_state<T>> {
```

# Futures

```
template <class T>
```

```
S template <class T>
```

```
struct _my_recvr {
```

```
    shared_ptr<_my_state<T>> _st;
```

```
    template <int I, class U>
```

```
    void _set(U&& u) {
```

```
        lock_guard lk(_st->_m);
```

```
        _st->_v.template emplace<I>((U&&) u);
```

```
        _st->_cv.notify_one();
```

```
    }
```

```
    void operator()(T t) { _set<1>(move(t)); }
```

```
    void set_error(exception_ptr e) { _set<2>(e); }
```

```
    void set_done() {
```

```
        _set<2>(make_exception_ptr(cancelled{}));
```

```
    }
```

```
};
```

```
template <class T>
```

```
struct MyFuture {
```

```
private:
```

```
    shared_ptr<_my_state<T>> _st =
```

```
        make_shared<_my_state<T>>();
```

```
public:
```

```
    template <SenderOf<T> S>
```

```
    explicit MyFuture(S&& src) {
```

```
        ((S&&)src).submit(
```

```
            _st->make_receiver());
```

```
    }
```

```
    T get() && {
```

```
        return move(*_st).get();
```

```
    }
```

```
};
```

# Futures: Summary

- Eager interfaces can be layered on top of lazy without additional overhead.
- *The converse is not true: we cannot “lazy-ify” an eager async operation while also removing its inherent overhead.*
- *Therefore, lazy operations are more fundamental.*
- The optimal way to “eager-ify” a lazy operation depends on many things: there should be many

But remember:

***Concurrency*** is only half of the story

If we can't also express ***parallel*** use cases,  
are we really being generic?



# FLASHBACK:

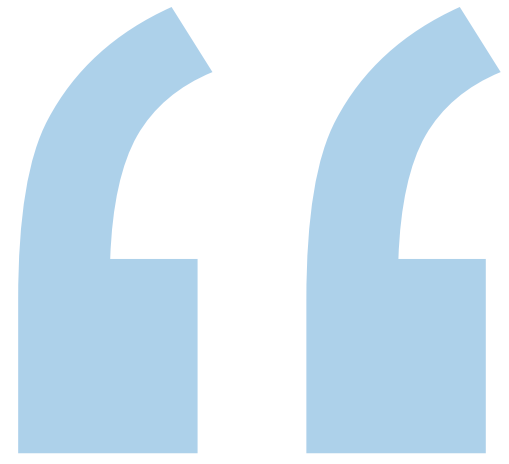
## Why are the parallel algorithms fast?

- Because they let the user communicate to the scheduler critical information about (the lack of) cross-task dependencies.
- In other words, it communicates the full structure of the algorithm's *task graph* to the scheduler.

# Sender/Receiver and Parallelism

- A non-intrusive parallel\_fork algorithm, like then, creates a node in a task graph of lazy Senders.
- By composing lazy Senders, we build a representation of the data flow graph *independent of its execution*.
- *How* that graph gets executed can then be left up to the scheduler.

# VI. Summary



It is very important that we design a system that does not only satisfy Facebook's needs, or Nvidia's, or that satisfies special case argument combinations for individual use cases, but one that cleanly generalizes for interoperation between different libraries, from different vendors with different goals.

Lee Howes, Facebook on the need to formalize callbacks

# Async Abstraction

- Sender/Receiver is a generalization of Future/Promise that:
  - Accommodates both eager and lazy async
  - Supports cancellation and error propagation
  - Composes with low overhead
  - Permits generic algorithms with efficient default implementations
  - Naturally accommodates “executors” as a special case of a Sender.

# Additional Resources

- The Ongoing Saga of Executors by David Hollman  
[https://www.youtube.com/watch?v=iYMfYdO0\\_OU](https://www.youtube.com/watch?v=iYMfYdO0_OU)
- A Compromise Executors Design Sketch by <lots>  
[P1660](#)
- Callbacks and Composition by Kirk Shoop [P1678](#)
- Cancellation is not an Error by Kirk Shoop [P1677](#)