



CHANNELS: AN ALTERNATIVE TO CALLBACKS AND FUTURES

JOHN R. BANDELA, MD

ASYNCHRONOUS PROGRAMMING



call Me!

C++ KNOWS ABOUT 2 MODELS

- Callbacks
- Futures

CALLBACKS

- Oldest model
- When something needs to happen, calls a user supplied function
- Synchronous example
 - qsort
- Asynchronous example
 - signal

CALLBACK

```
socket.async_receive(buffer,  
    [](error_code ec, size_t n) {  
        ...  
    });
```

FUTURE

- Anthony Williams – C++ Concurrency in Action
- The C++ Standard Library models this sort of one-off event with something called a *future*. If a thread needs to wait for a specific one-off event, it somehow obtains a future representing this event. The thread can then periodically wait on the future for short periods of time to see if the event has occurred (check the departures board) while performing some other task (eating in the overpriced café) in between checks. Alternatively, it can do another task until it needs the event to have happened before it can proceed and then just wait for the future to become *ready*. A future may have data associated with it (such as which gate your flight is boarding at), or it may not. Once an event has happened (and the future has become *ready*), the future can't be reset.

FUTURE

```
future<string> f = read_string_from_file();
```

```
// Blocking call:
```

```
string s = f.get();
```

```
use(s);
```

A PROBLEM IN THE FUTURE

- Early recognition that blocking in `future.get` is less than ideal for asynchronous code
- Concurrency TS (ISO/IEC TS 19571:2016) published in January of 2016
 - `future.then`
 - `when_any`
 - `when_all`

FUTURE

```
future<string> f = read_string_from_file();
```

```
// Non-blocking call:
```

```
f.then([](future<string> result) {  
    string s = result.get();  
    use(s);  
});
```

WHEN_ANY

```
future<int> fi = ...; future<char> fc = ...;
auto ready = when_any(fi, fc);
ready.then([](auto result) {
    auto wa = result.get();
    if (wa.index == 0) {
        future<int> fi = get<0>(wa.futures); assert(fi.is_ready());
    } else {
        future<char> fc = get<1>(wa.futures); assert(fc.is_ready());
    }
});
```

CALLBACK VS FUTURE

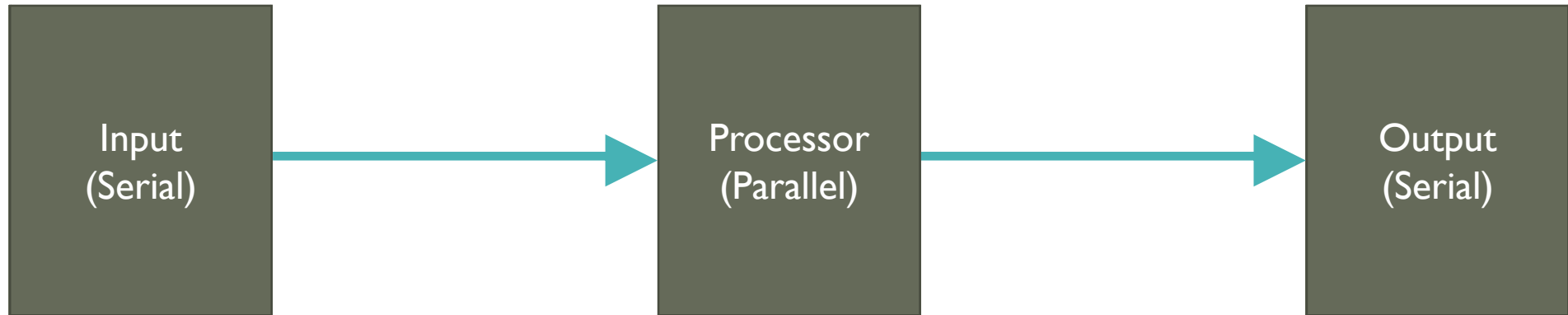
- Callback
 - Conceptually simple
 - Efficient
 - Difficult to compose
- Future
 - More complicated
 - Less efficient
 - Easy to compose ie when_any
 - Concurrency TS futures are not widely implemented

WHY DO WE NEED ANYTHING ELSE BESIDES FUTURES AND CALLBACKS

- Futures are designed for one-off events
- There are many situations that produce a stream of related events
- We want a way to deal with a stream of asynchronous events efficiently while still being able to compose them



THINK ABOUT HOW TO DESIGN THIS WITH FUTURES



PERFORMANCE



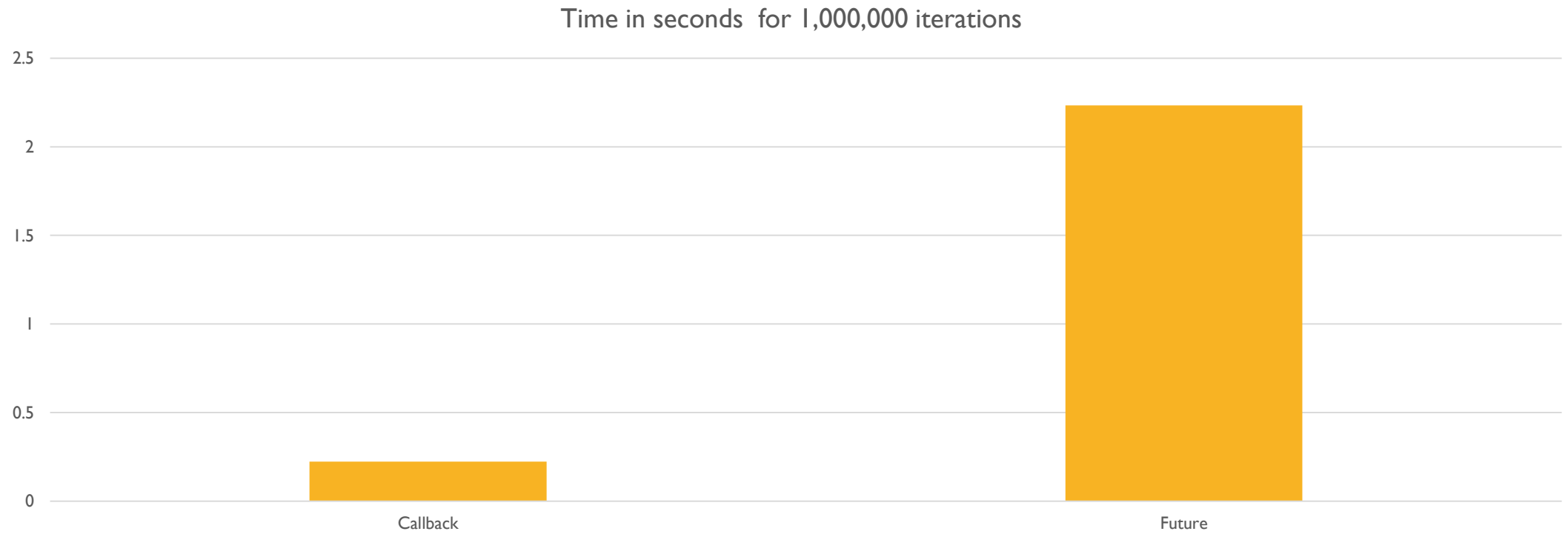
CALLBACK

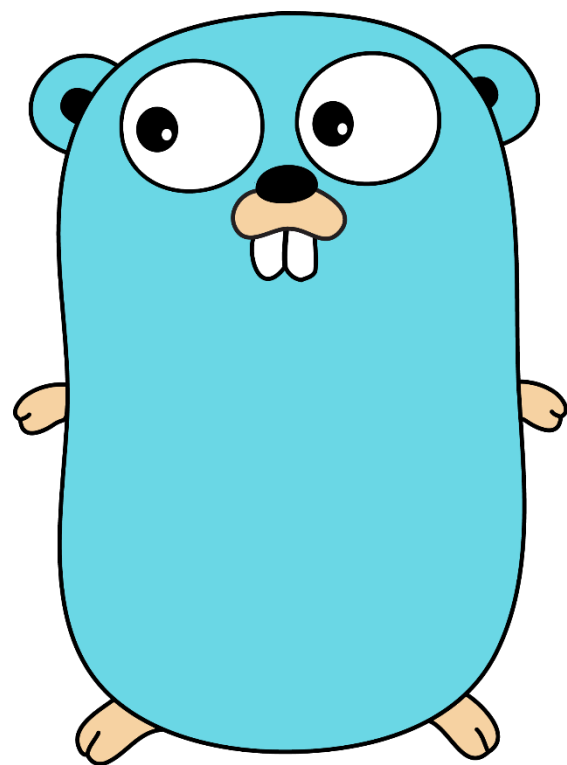
```
auto test_callback(int count) {  
    boost::executors::loop_executor loop;  
    for (int i = 0; i < count; ++i) {  
        loop.submit([]() {});  
        loop.run_queued_closures();  
    }  
}
```


FUTURE

```
auto test_future(int count) {  
    boost::executors::loop_executor loop;  
    for (int i = 0; i < count; ++i) {  
        boost::promise<int> p;  
        auto fut = p.get_future();  
        bool done = false;  
        fut.then(loop, [&](auto &&) { done = true; });  
        p.set_value(i);  
        while (!done) {loop.run_queued_closures();}  
    }  
}
```

CALLBACK VS FUTURE





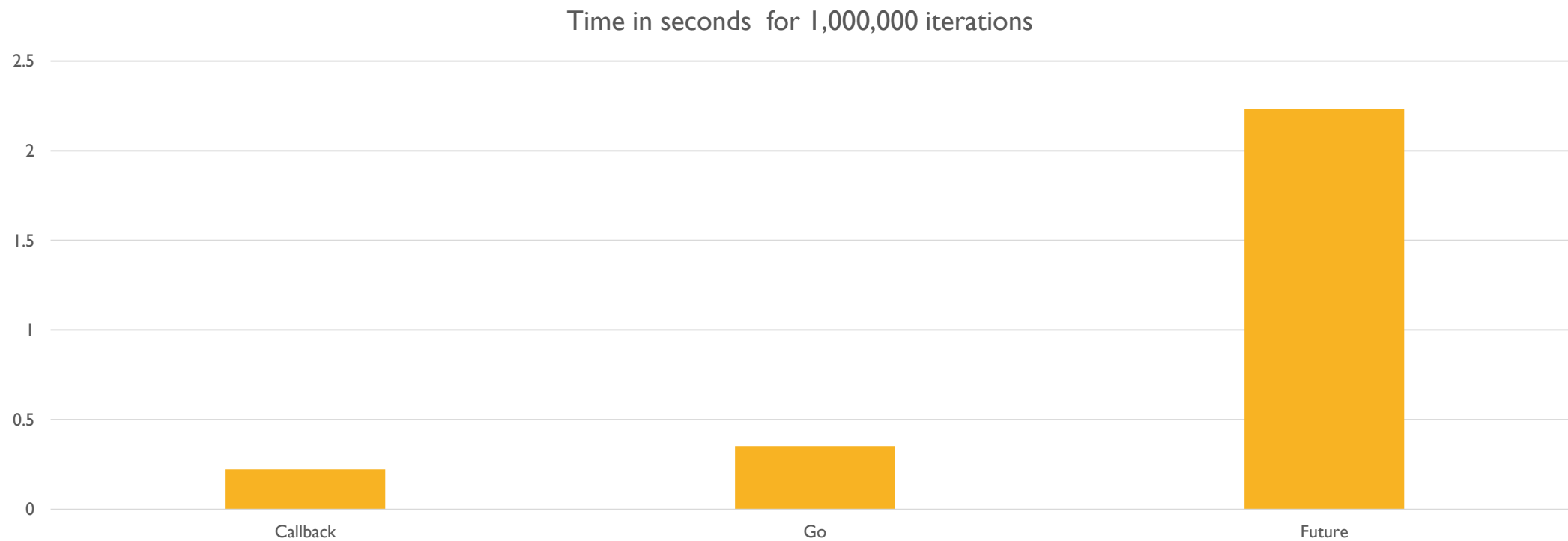
GO WRITER

```
func writer(c chan int, count int) {  
    for x := 0; x < count; x++ {  
        c <- x  
    }  
    close(c)  
}
```

GO READER

```
func reader(c chan int, done chan struct{}) {  
    for {  
        _, ok := <-c  
        if ok == false {  
            done <- struct{}{}  
            break  
        }  
    }  
}
```

CALLBACK VS FUTURE



BOOST 1.62



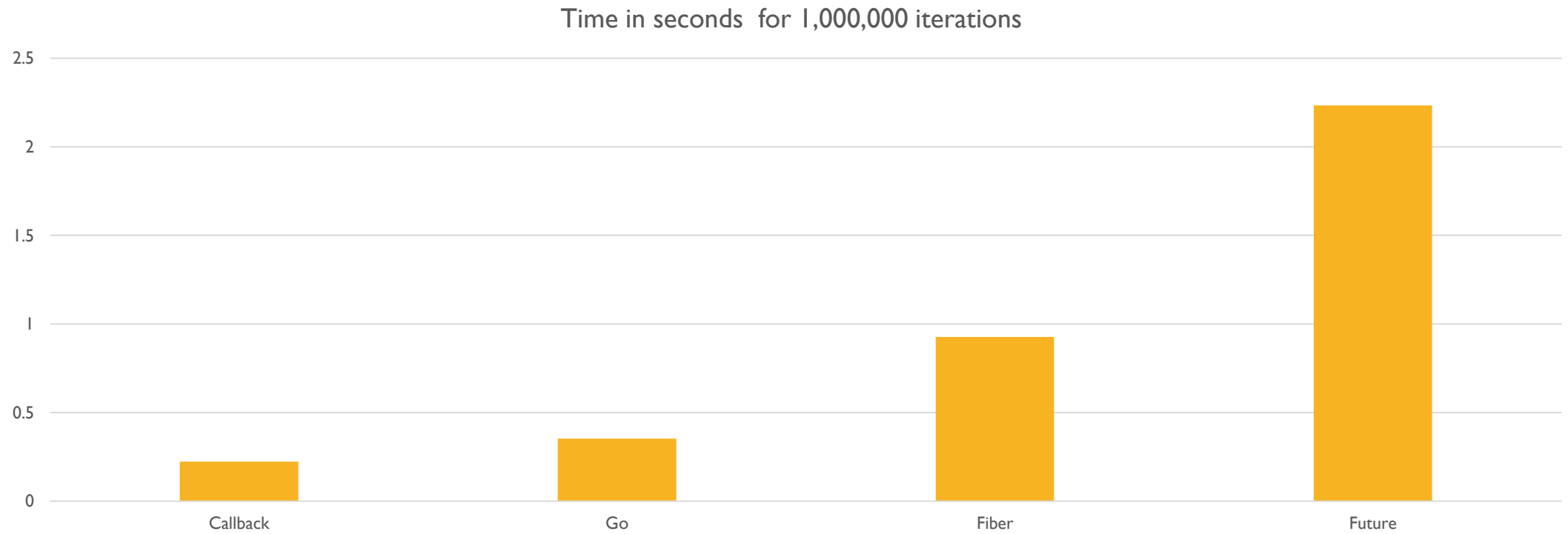
BOOST FIBER CHANNELS

```
void fiber_writer(boost::fibers::bounded_channel<int>& chan, int count) {  
    for (int i = 0; i < count; ++i) {  
        chan.push(i);  
    }  
    chan.close();  
}
```


BOOST FIBER CHANNELS

```
void fiber_reader(boost::fibers::bounded_channel<int> &chan) {  
    for (;;) {  
        int i;  
        auto v = chan.pop(i);  
        if (v == boost::fibers::channel_op_status::closed) {  
            return;  
        }  
    }  
}
```

CALLBACK VS FUTURE VS FIBER CHANNEL



FIBER CHANNEL

- Slower than Go Channels
- Does not have select
- Only works with Fibers

TDD

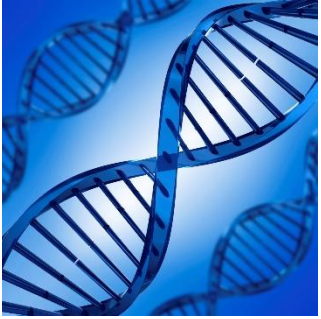
Thought Driven Development

WHAT C++ FEATURE IS THIS?

DUH!



WHAT C++ FEATURE IS THIS?



DUH!

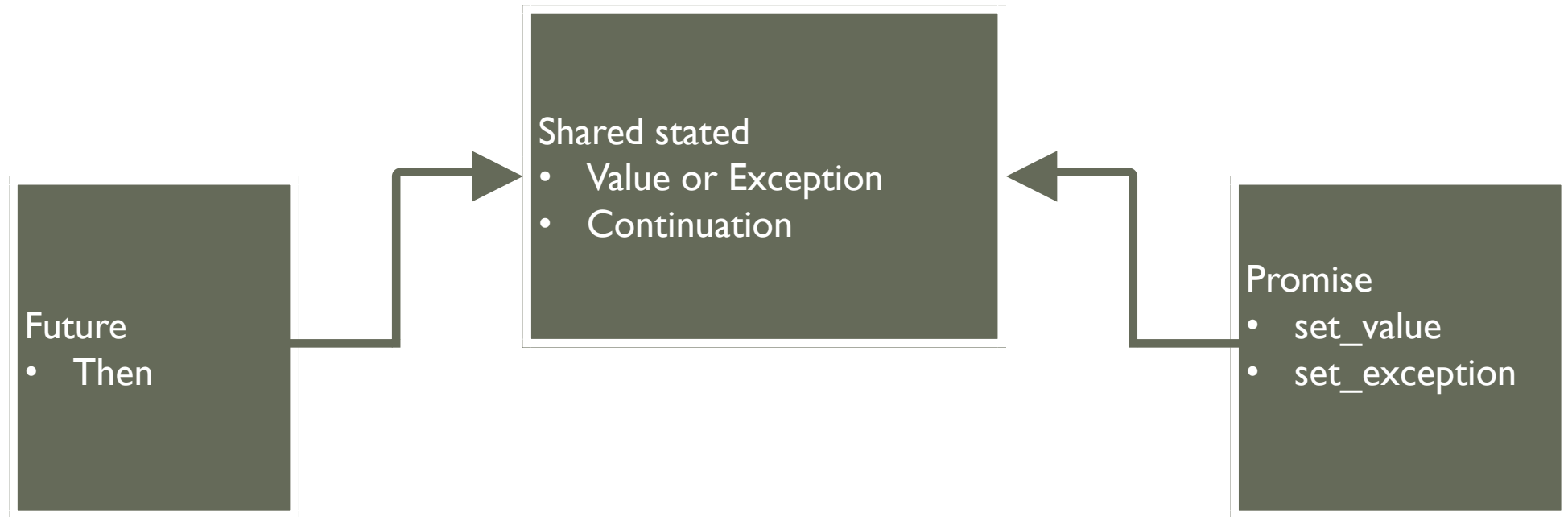
GOALS

- C++ channel with zero dynamic memory allocations once channels set up other than those required by moving
- Channel select – `when_any` for channels
- Ergonomic
- Adaptable for use with other types of coroutines

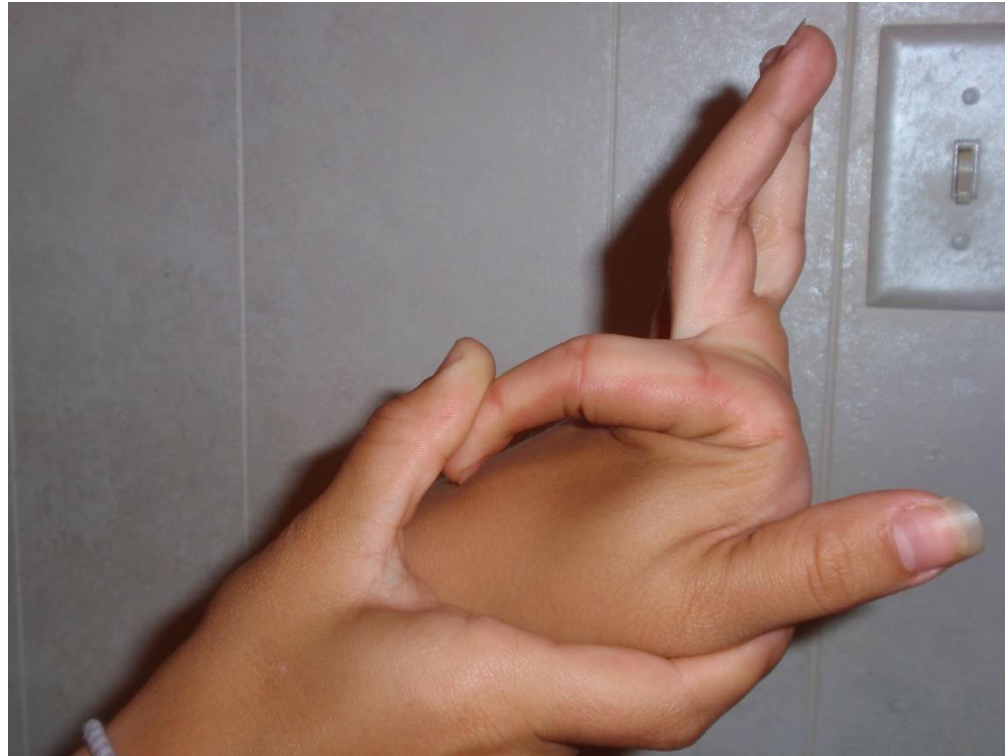
ADAPTABILITY

- The channels developed in this presentation support
 - Coroutine TS (co_await)
 - Library stackless coroutines - https://github.com/jbandela/stackless_coroutine
 - Header-only, no macro, C++14 stackless coroutine implementation
- This presentation will use Coroutine TS as that is standard and less verbose

HOW IS A FUTURE IMPLEMENTED



IT HURTS WHEN I DO THIS



Trivial if synchronous

```
int tcp_reader(int total)
{
    char buf[4 * 1024];
    auto conn = Tcp::Connect("127.0.0.1", 1337);
    for (;;)
    {
        auto bytesRead = conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```

.then

```
future<int> tcp_reader(int64_t total) {  
    struct State {  
        char buf[4 * 1024];  
        int64_t total;  
        Tcp::Connection conn;  
        explicit State(int64_t total) : total(total) {}  
    };  
    auto state = make_shared<State>(total);  
    return Tcp::Connect("127.0.0.1", 1337).then(  
        [state](future<Tcp::Connection> conn) {  
            state->conn = std::move(conn.get());  
            return do_while([state]()->future<bool> {  
                if (state->total <= 0) return make_ready_future(false);  
                return state->conn.read(state->buf, sizeof(state->buf)).then(  
                    [state](future<int> nBytesFut) {  
                        auto nBytes = nBytesFut.get();  
                        if (nBytes == 0) return make_ready_future(false);  
                        state->total -= nBytes;  
                        return make_ready_future(true);  
                    });  
            });  
        });  
});  
}
```

```
future<void> do_while(function<future<bool>()> body) {  
    return body().then([=](future<bool> notDone) {  
        return notDone.get() ? do_while(body) : make_ready_future(); });  
}
```

COROUTINES TO THE RESCUE

Document Number: P0057R5
Date: 2016-07-10
Revises: P0057R4
Audience: CWG / LWG
Authors: Gor Nishanov <gorn@microsoft.com>
Jens Maurer <Jens.Maurer@gmx.net>
Richard Smith <richard@metafoo.co.uk>
Daveed Vandevoorde <daveed@edg.com>

Wording for Coroutines

Trivial

```
auto tcp_reader(int total) -> future<int>
{
    char buf[4 * 1024];
    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    for (;;)
    {
        auto bytesRead = await conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```

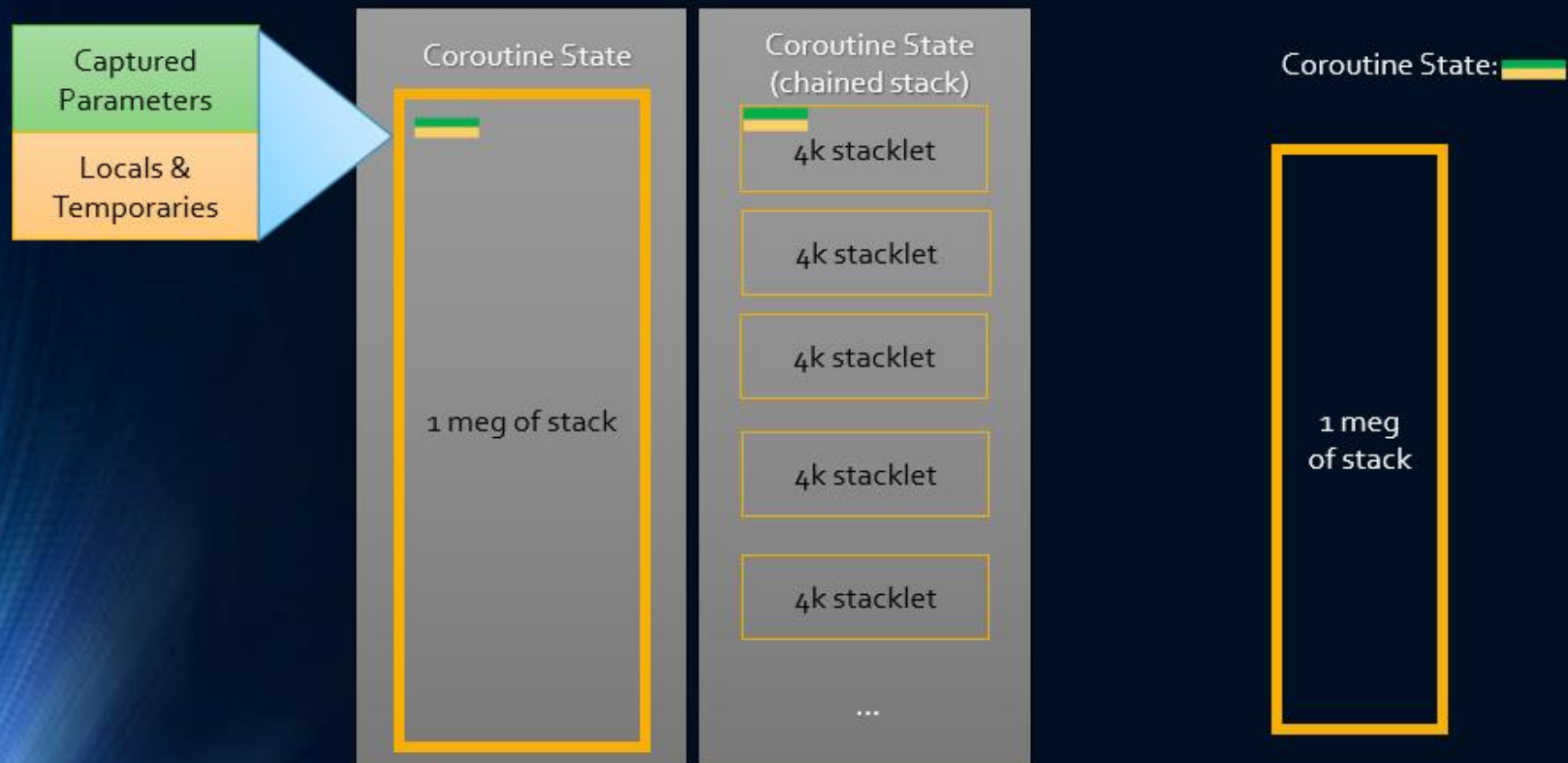
HOW DO COROUTINES WORK?

- Place to store variables across coroutine invocations
- Ability to suspend and resume

Stackful

vs.

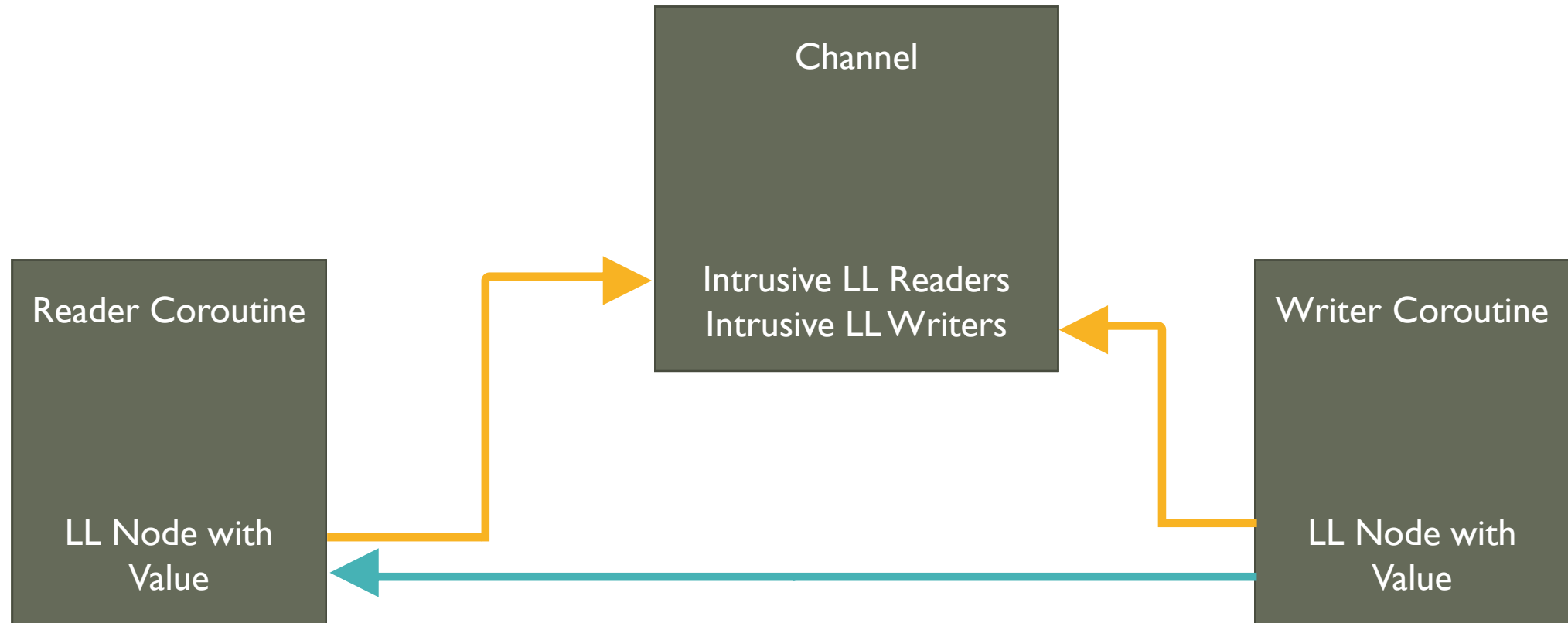
Stackless



GOALS

- C++ channel with zero dynamic memory allocations once channels set up other than those required by moving

TDD



SHOW ME THE CODE



NODE BASE

```
struct node_base {  
    using function_t = void (*)(node_base *node);  
    node_base *next = nullptr;  
    node_base *previous = nullptr;  
    void *data = nullptr;  
    function_t func = nullptr;  
    bool closed = false;  
};
```

NODE

```
template <class T>  
struct node_t : node_base { T value; };
```

CHANNEL

```
template <class T> struct channel {  
    using node_t = detail::node_t<T>;  
    node_base *read_head = nullptr;  
    node_base *read_tail = nullptr;  
    node_base *write_head = nullptr;  
    node_base *write_tail = nullptr;  
    std::atomic<bool> closed{false};  
    std::mutex mut;  
    using lock_t = std::unique_lock<std::mutex>
```

CHANNEL WRITE - DEFINITION

```
void write(node_t *writer) {
```

CHANNEL WRITE – CHECK IF CLOSED

```
if (closed) {  
    writer->closed = true;  
    writer->func(writer);  
    return;  
}
```


CHANNEL WRITE – GET READER

```
lock_t lock{mut};  
auto reader = static_cast<node_t *>(  
    detail::remove(read_head, read_tail, read_head));
```

CHANNEL WRITE – IF NO READER, ADD TO WRITE LIST

```
if (!reader) {  
    detail::add(write_head, write_tail, writer);  
}
```

CHANNEL WRITE - SUCCESS

```
else {  
    lock.unlock();  
    reader->value = std::move(writer->value);  
    reader->closed = false;  
    writer->closed = false;  
    writer->func(writer);  
    reader->func(reader);  
}
```

CHANNEL WRITER

```
template <class T, class PtrType = std::shared_ptr<channel<T>>>
struct await_channel_writer
    using node_t = typename channel<T>::node_t;
    node_t node{};
    PtrType ptr;
```

CHANNEL WRITER WRITE

```
auto write(T t) {  
    node.value = std::move(t);  
    struct awaiter {  
        await_channel_writer *pthis;  
        bool await_ready() { return false; }  
        void await_suspend(coroutine_handle<> rh);  
        auto await_resume() {return !pthis->node.closed;}  
    };  
    return awaiter{this};  
}
```

CHANNEL WRITER – AWAIT SUSPEND

```
void await_suspend(std::experimental::coroutine_handle<> rh) {  
    pthis->node.func = [](detail::node_base *n) {  
        auto node = static_cast<node_t *>(n);  
        auto rh = coroutine_handle<>::from_address(node->data);  
        rh();  
    };  
    pthis->node.data = rh.to_address();  
    pthis->ptr->write(&pthis->node);  
}
```

CHANNEL WRITER

```
goroutine writer(std::shared_ptr<channel<int>> chan, int count) {  
    await_channel_writer<int> writer{chan};  
    for (int i = 0; i < count; ++i) {  
        co_await writer.write(i);  
    }  
    chan->close();  
}
```

CHANNEL READER

```
goroutine reader(std::shared_ptr<channel<int>> chan) {  
    await_channel_reader<int> reader{chan};  
    for (;;) {  
        auto p = co_await reader.read();  
        if (p.first == false)  
            break;  
    }  
}
```




CHANNEL WRITE - SUCCESS

```
else {  
    lock.unlock();  
    reader->value = std::move(writer->value);  
    reader->closed = false;  
    writer->closed = false;  
    writer->func(writer);  
    reader->func(reader);  
}
```



HOW DO WE MAKE AN EXECUTOR WITHOUT ALLOCATION?

CHANNEL EXECUTOR

```
struct channel_executor {  
    using node_base = detail::node_base;  
    node_base *head = nullptr;  
    node_base *tail = nullptr;  
    std::mutex mut;  
    using lock_t = std::unique_lock<std::mutex>;  
    std::condition_variable cvar;
```

CHANNEL EXECUTOR - ADD

```
bool add(node_base *node) {  
    lock_t lock{mut};  
    detail::add(head, tail, node);  
    lock.unlock();  
    cvar.notify_all();  
    return true;  
}
```

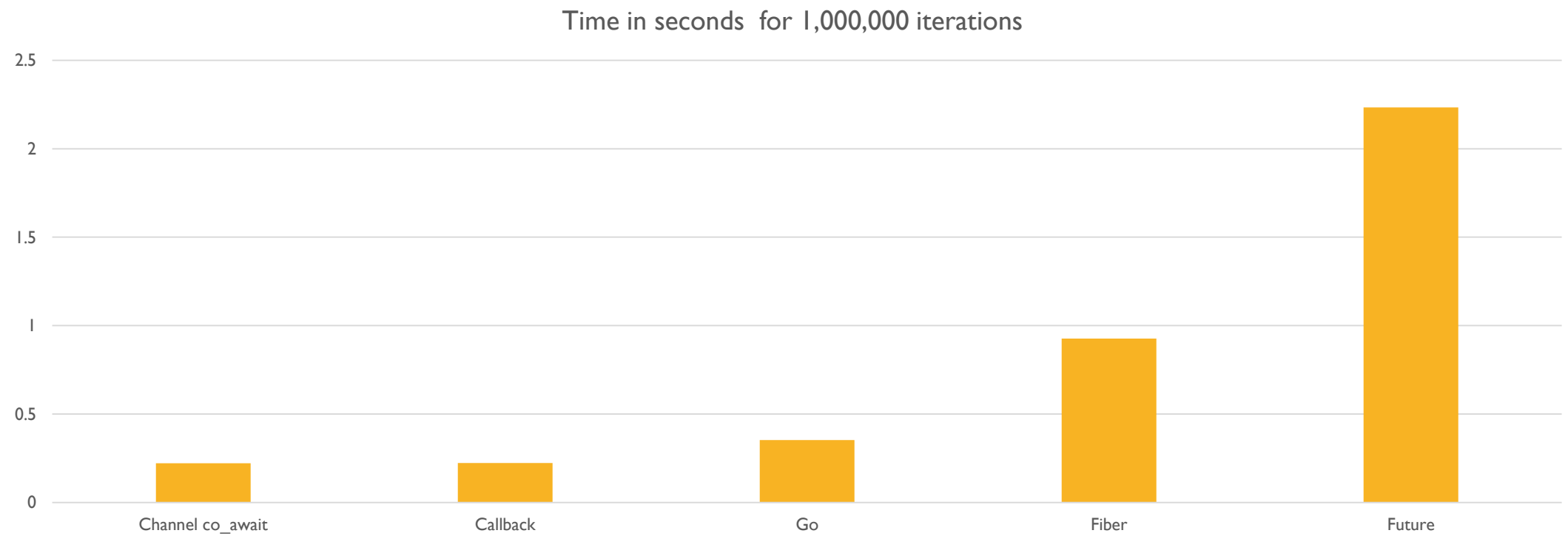
CHANNEL EXECUTOR - RUN

```
void run() {  
    while (true) {  
        lock_t lock{mut};  
        auto node = detail::remove(head, tail, head);  
        while (!node) {  
            cvar.wait(lock);  
            node = detail::remove(head, tail, head);  
        }  
        lock.unlock();  
        node->func(node);  
    }  
}
```

CHANNEL WRITE - SUCCESS

```
else {  
    lock.unlock();  
    reader->value = std::move(writer->value);  
    reader->closed = false;  
    writer->closed = false;  
    executor->add(reader);  
    executor->add(writer);  
}
```


PERFORMANCE



SELECTING CHANNELS



GOALS

- C++ channel with zero dynamic memory allocations once channels set up other than those required by moving
- Channel select – `when_any` for channels

GO SELECT WRITER

```
func writer_select(c1 chan int, c2 chan int, count int) {  
    for x := 0; x < count; x++ {  
        if x % 2 == 0 { c1 <- x }  
        else { c2 <- x }  
    }  
    close(c1)  
    close(c2)  
}
```

GO SELECT READER

```
func reader_select(c1 chan int, c2 chan int) {  
    for {  
        select{  
            case _, ok := <-c1:  
                if ok == false {return}  
            case _, ok := <-c2:  
                if ok == false {return}  
        }  
    }  
}
```

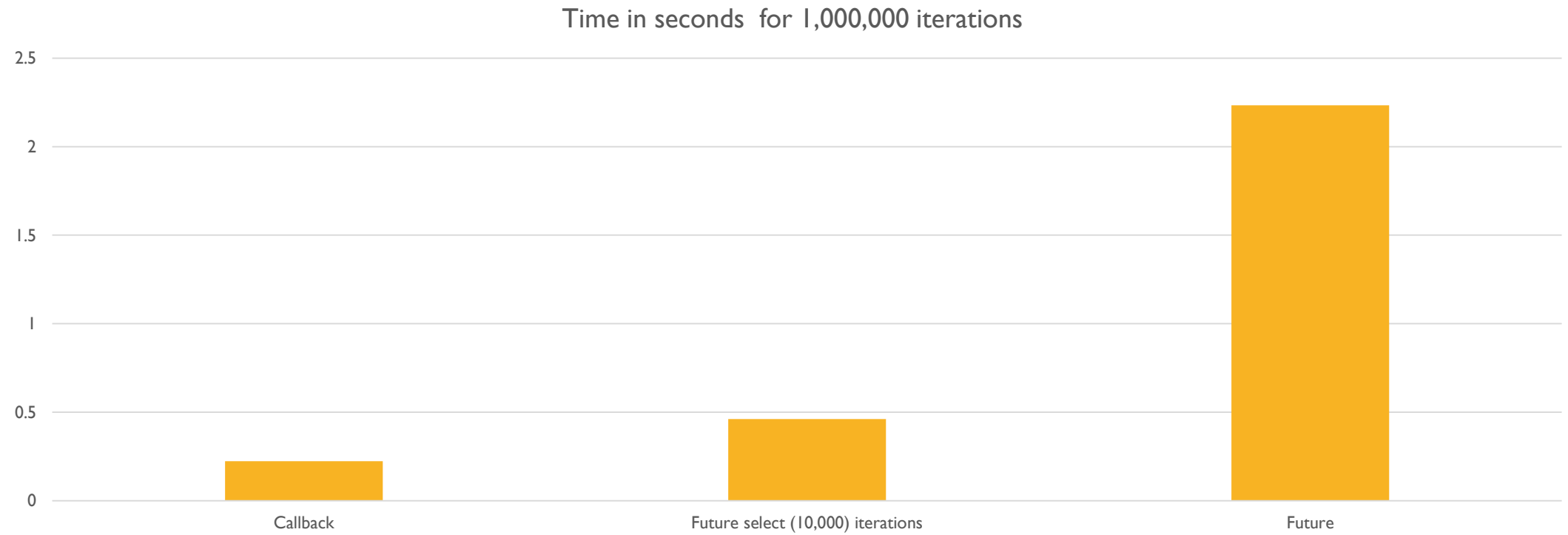
WHEN ANY

```
auto test_future_select(int count) {  
    boost::executors::loop_executor loop;  
    boost::promise<int> p1, p2;  
    auto fut1 = p1.get_future();    auto fut2 = p2.get_future();  
    for (int i = 0; i < count; ++i) {  
        auto selected = boost::when_any(std::move(fut1), std::move(fut2));  
        bool done = false;  
        selected.then(loop, [&](auto &&f) {...});  
        if (i % 2) {p1.set_value(i);} else {p2.set_value(i); }  
        while (!done) { loop.run_queued_closures();}  
    }  
}
```

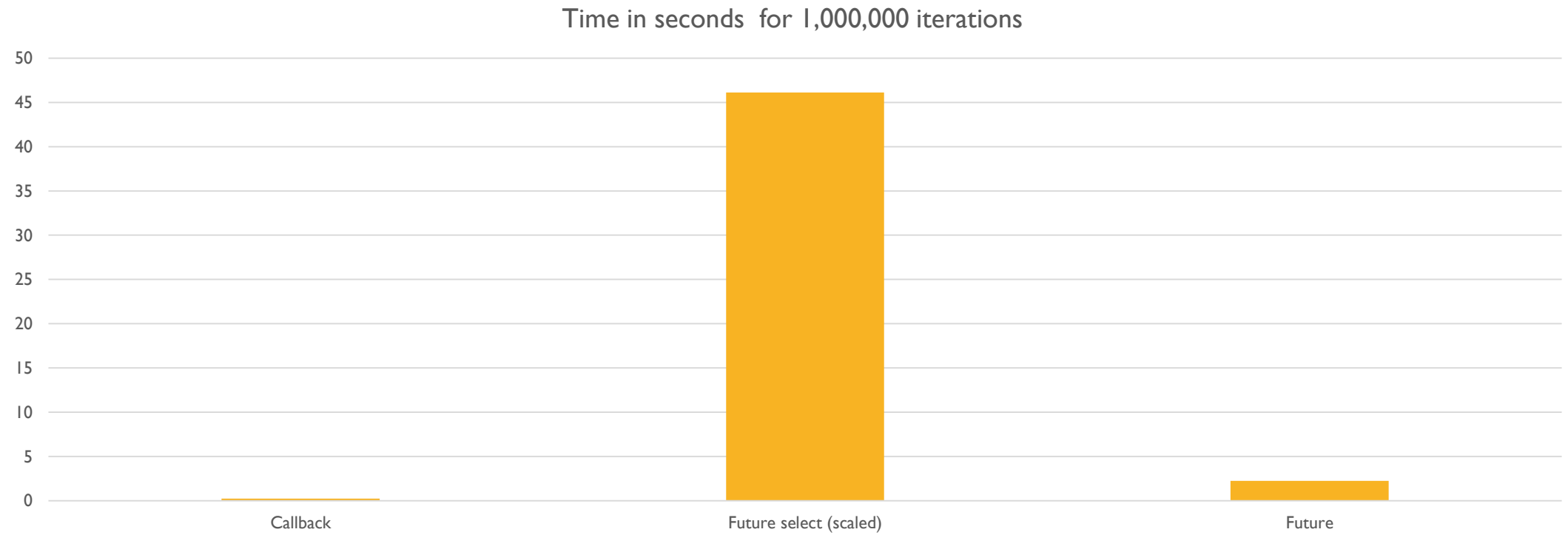
WHEN ANY

```
selected.then(loop, [&](auto &&f) {  
    auto futures = f.get();  
    if (std::get<0>(futures).is_ready()) {  
        p1 = boost::promise<int>{};  
        fut1 = p1.get_future();  
        fut2 = std::move(std::get<1>(futures));  
    } else {  
        p2 = boost::promise<int>{};  
        fut2 = p2.get_future();  
        fut1 = std::move(std::get<0>(futures));  
    }  
    done = true;  
});
```

PERFORMANCE



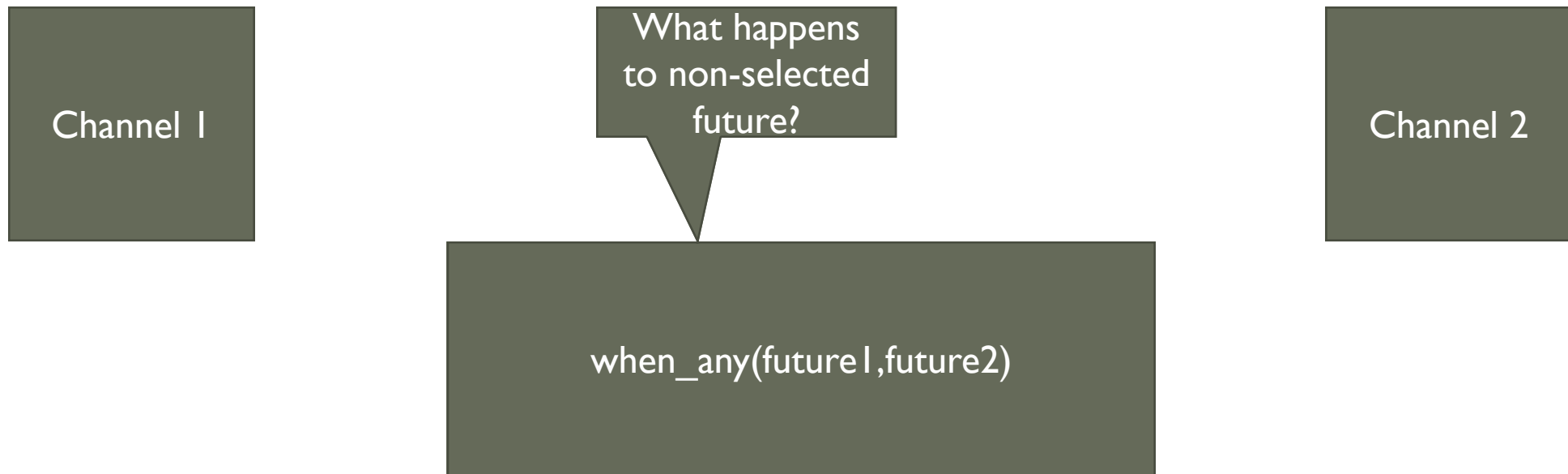
PERFORMANCE



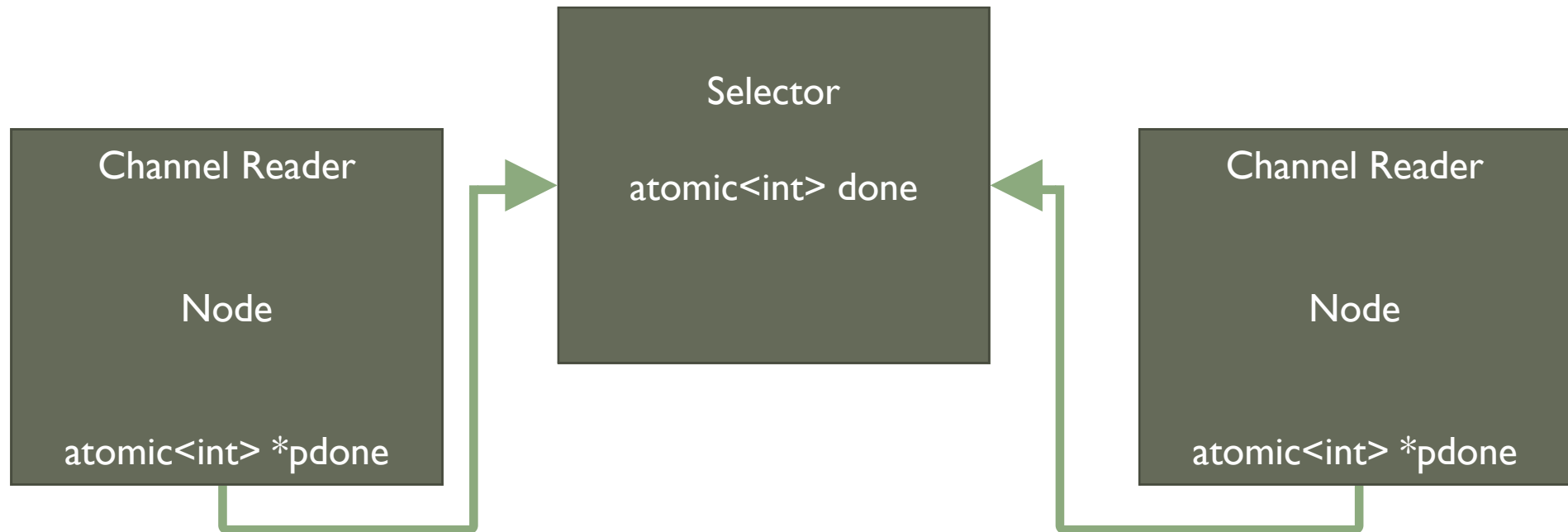
PERFORMANCE



ASIDE: WHY CHANNEL ISN'T JUST A STREAM OF FUTURES



TDD



SHOW ME THE CODE



NODE BASE

```
struct node_base {  
    using function_t = void (*)(node_base *node);  
    node_base *next = nullptr;  
    node_base *previous = nullptr;  
    void *data = nullptr;  
    function_t func = nullptr;  
    bool closed = false;  
  
    std::atomic<int> *pdone = nullptr;  
}
```

HANDLING PDONE

```
// 0 = not done, 1 = done, -1 = indeterminate
static bool set_done(std::atomic<int> *pdone, int expected, int value) {
    if (!pdone) return true;
    while (true) {
        int exp = expected;
        if (pdone->compare_exchange_strong(exp, value)) {return true; }
        if (exp == 1) {
            // somebody else set it to done, we will never succeed
            return false;
        }
        // it is indeterminate - keep looping;
    }
}
```


HANDLING PDONE

```
static bool set_intermediate(std::atomic<int> *pdone) {  
    return set_done(pdone, 0, -1);  
}  
static bool clear_intermediate_success(std::atomic<int> *pdone) {  
    return set_done(pdone, -1, 1);  
}  
static bool clear_intermediate_failure(std::atomic<int> *pdone) {  
    return set_done(pdone, -1, 0);  
}  
static bool set_success(std::atomic<int> *pdone) {  
    return set_done(pdone, 0, 1);  
}
```


CHANNEL SELECTOR

```
struct channel_selector {  
    std::atomic<int> done{0};  
};
```

CHANNEL READ— CHECK IF CLOSED




```
void read(node_t *reader) {  
    lock_t lock{mut};  
    if (closed && write_head == nullptr) {  
        lock.unlock();  
        if (set_success(reader->pdone)) {  
            reader->closed = true;  
            executor->add(reader);  
        }  
        return;  
    }  
}
```

CHANNEL READ – GET WRITER


```
auto writer = static_cast<node_t *>(
    detail::remove(write_head, write_tail, write_head));
while (writer && !set_intermediate(writer->pdone)) {
    writer = static_cast<node_t *>(
        detail::remove(write_head, write_tail, write_head));
}
if (!writer) {
    detail::add(read_head, read_tail, reader);
}
```

CHANNEL READ— WE HAVE WRITER



```
else {  
    if (!set_success(reader->pdone)) {  
        detail::add_head(write_head, write_tail, writer);  
        clear_intermediate_failure(writer->pdone);  
        return;  
    }  
}
```

CHANNEL READ - SUCCESS



```
clear_intermediate_success(writer->pdone);  
lock.unlock();  
reader->value = std::move(writer->value);  
writer->closed = false;  
reader->closed = false;  
executor->add(reader);  
executor->add(writer);  
}
```

```
}
```

SELECT RANGE

```
template <class Range, class Func>
auto select_range(Range &r, Func f) {
    struct awaiter {...};
    return awaiter{&r, std::move(f)};
}
```

AWAITER – MEMBER VARIABLES

```
struct awaiter {  
    Range *rng;  
    Func f;  
    channel_selector s;  
    void *prh = nullptr;  
    detail::node_base *selected_node = nullptr;  
    std::mutex mut;  
    ...  
};
```

AWAITER – AWAIT_SUSPEND

```
bool await_ready() { return false; }  
void await_suspend(coroutine_handle<> rh) {  
    std::unique_lock<std::mutex> lock{mut};  
    for (auto &r : *rng) {  
        r.read(s, rh, *this);  
    }  
}
```


CHANNEL READER - READ

```
template <class Select, class Awaiter>
void read(Select &s, coroutine_handle<> rh, Awaiter &a) {
    a.prh = rh.to_address();
    node.data = &a;
    node.pdone = &s.done;
```

CHANNEL READER - READ

```
node.func = [](detail::node_base *n) {  
    auto node = static_cast<node_t *>(n);  
    auto pa = static_cast<Awaiter *>(node->data);  
    std::unique_lock<std::mutex> lock{pa->mut};  
    lock.unlock();  
    auto rh = coroutine_handle<>::from_address(pa->prh);  
    pa->selected_node = node;  
    rh();  
};  
ptr->read(&node);  
}
```

AWAITER – AWAIT_RESUME

```
auto await_resume() {  
    auto iter = begin(*rng);  
    auto selected_iter = end(*rng);  
    for (; iter != end(*rng); ++iter) {  
        iter->remove();  
        if (iter->get_node() == selected_node) {  
            f(iter->get_node()->value);  
            selected_iter = iter;  
        }  
    }  
    return std::make_pair(!selected_node->closed, selected_iter);  
}
```

CHANNEL READER - REMOVE

```
void remove() { ptr->remove_reader(&node); }
```

CHANNEL REMOVE READER

```
void remove_reader(node_t *reader) {  
    lock_t lock{mut};  
    detail::remove(read_head, read_tail, reader);  
}
```

GO SELECT WRITER

```
func writer_select(c1 chan int, c2 chan int, count int) {  
    for x := 0; x < count; x++ {  
        if x % 2 == 0 { c1 <- x }  
        else { c2 <- x }  
    }  
    close(c1)  
    close(c2)  
}
```

GO SELECT READER

```
func reader_select(c1 chan int, c2 chan int) {  
    for {  
        select{  
            case _, ok := <-c1:  
                if ok == false {return}  
            case _, ok := <-c2:  
                if ok == false {    return}  
        }  
    }  
}
```

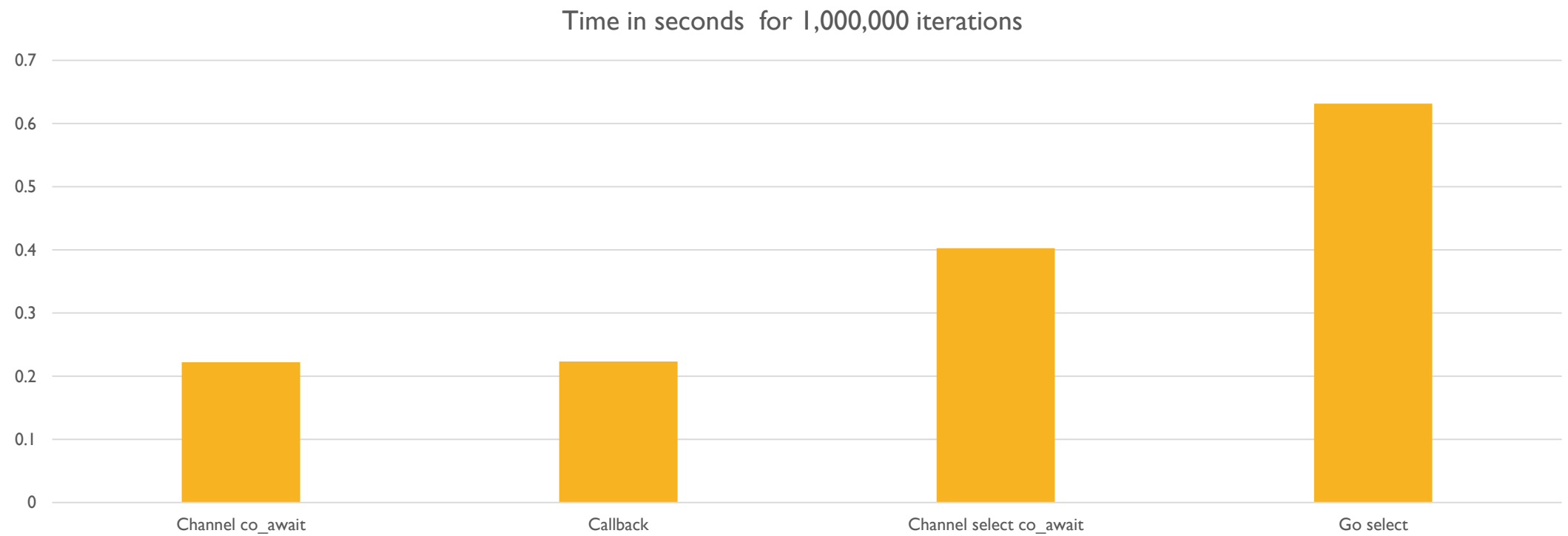
CHANNEL SELECT WRITER

```
goroutine writer_select(shared_ptr<channel<int>> chan1,  
                        shared_ptr<channel<int>> chan2, int count) {  
    await_channel_writer<int> writer1{chan1};  
    await_channel_writer<int> writer2{chan2};  
    for (int i = 0; i < count; ++i) {  
        if (i % 2) co_await writer1.write(i);  
        else      co_await writer2.write(i);  
    }  
    chan1->close();  
    chan2->close();  
}
```


CHANNEL SELECT READER

```
goroutine reader_select(shared_ptr<channel<int>> chan1,  
                        shared_ptr<channel<int>> chan2) {  
    await_channel_reader<int> reader1{chan1};  
    await_channel_reader<int> reader2{chan2};  
    for (;;) {  
        auto p = co_await select(  
            reader1, [](auto) {},  
            reader2, [](auto) {}  
        );  
        if (p.first == false) break;  
    }  
}
```

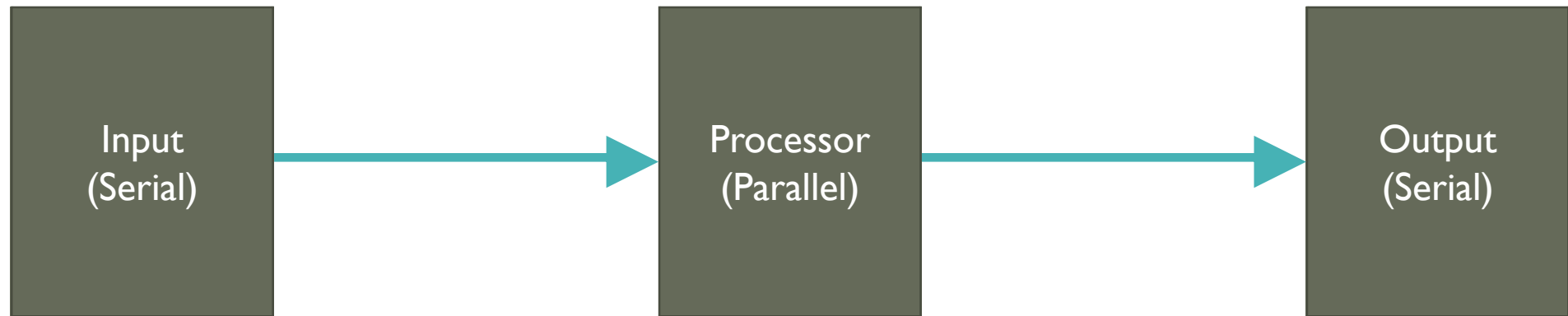
PERFORMANCE



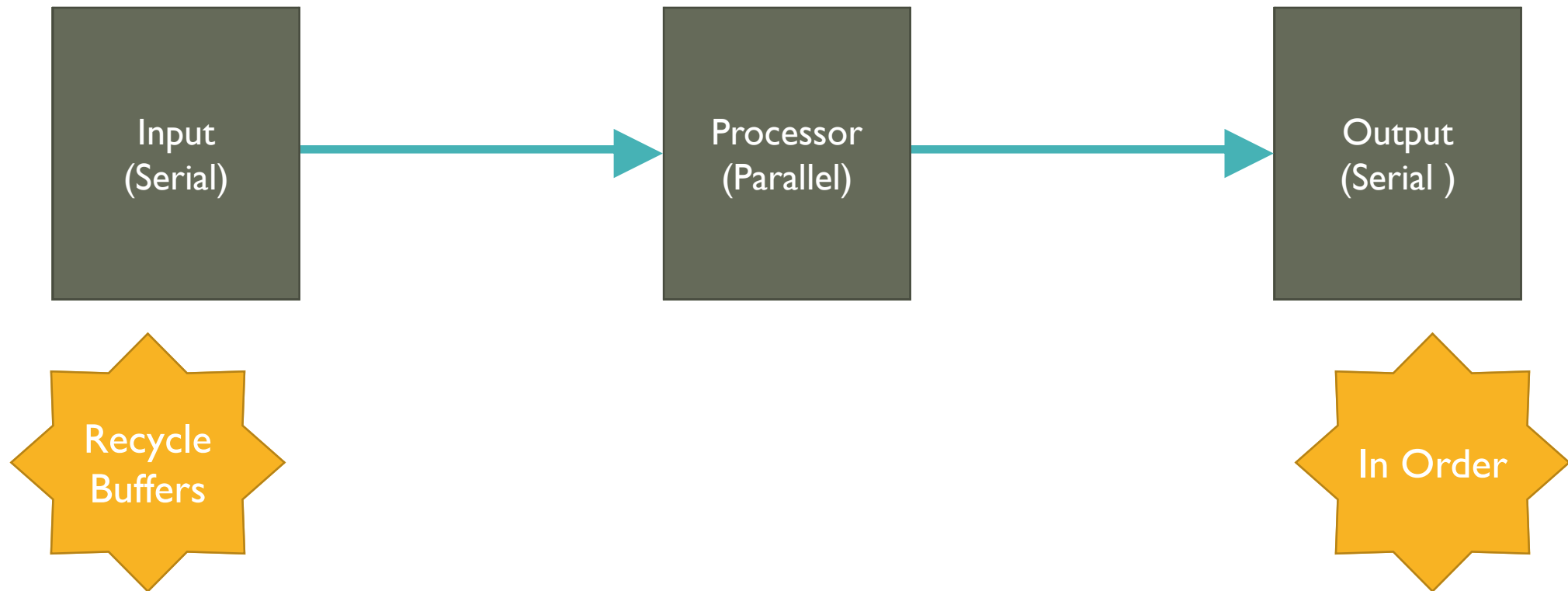
OTHER FEATURES

- Currently Implemented
 - Buffered channels
 - Allow select with read and write of same channel
 - Channel readers and writers for threads
- TBD
 - Further optimization
 - Lock-free buffers
 - Documentation
 - Better packaging
 - Improved promise type for the coroutine

CHANNELS CAN BE A USEFUL WAY TO THINK ABOUT CONCURRENCY



LET'S MAKE IT HARDER





TYPE ALIASES

- `using array_type = std::array<char, 32 * 1024 * 1024>;`
- `using payload_type = std::unique_ptr<array_type>;`
- `using ordered_payload_type = std::pair<int, payload_type>;`

READER

```
goroutine reader(shared_ptr<channel<ordered_payload_type>> chan,  
                 shared_ptr<channel<payload_type>> pool, int count) {  
    await_channel_writer<ordered_payload_type> writer{chan};  
    await_channel_reader<payload_type> pool_reader{pool};  
    for (int i = 0; i < count; ++i) {  
        auto res = co_await pool_reader.read();  
        // Read data into res.second  
        co_await writer.write({i, std::move(res.second)});  
    }  
    chan->close();  
}
```


MAKE PROCESSOR

```
auto make_processor(s
inchan) {
    auto outchan = std:
processor(inchan, c
return outchan;
}
```

PROCESSOR

```
goroutine processor (  
    shared_ptr<channel<ordered_payload_type>> inchan,  
    shared_ptr<channel<ordered_payload_type>> outchan) {
```

PROCESSOR

```
await_channel_reader<ordered_payload_type> reader{inchan};
await_channel_writer<ordered_payload_type> writer{outchan};
auto out_array = std::make_unique<array_type>();
for (;;) {
    auto res = co_await reader.read();
    if (res.first == false) { outchan->close(); return;}
    // Do something with res.second.second and out_array
    std::this_thread::sleep_for(std::chrono::seconds{1});
    co_await writer.write({res.second.first, std::move(out_array)});
    out_array = std::move(res.second.second);
}
}
```

WRITER - SIGNATURE

```
goroutine writer(  
    vector<shared_ptr<channel<ordered_payload_type>>> inchans,  
    shared_ptr<channel<payload_type>> pool,  
    shared_ptr<channel<bool>> done_chan  
) {
```

WRITER - INITIALIZATION

```
await_channel_writer<bool> done_writer{done_chan};  
vector<await_channel_reader<ordered_payload_type>>  
    readers{inchans.begin(),inchans.end()};  
await_channel_writer<payload_type> pool_writer{pool};  
std::vector<ordered_payload_type> reorder_buffer;  
reorder_buffer.reserve(readers.size());  
int current = 0;
```

WRITER – LOOP – EXIT CHECK

```
for (;;) {  
    if (readers.empty() && reorder_buffer.empty()) {  
        co_await done_writer.write(true);  
        return;  
    }  
}
```

WRITER – LOOP – CHANNEL SELECT

```
auto res = co_await select_range(readers,  
    [&](auto &ordered_payload) {  
        reorder_buffer.push_back(std::move(ordered_payload));  
        std::push_heap(reorder_buffer.begin(), reorder_buffer.end(),  
            [](auto &a, auto &b) { return a.first > b.first; });  
    });  
if (res.first == false) {  
    readers.erase(res.second);  
}
```

WRITER – LOOP – WRITE IN ORDER, MOVE PAYLOAD BACK TO

```
while (!reorder_buffer.empty() && reorder_buffer.front().first == current){  
    // Write data in reorder_buffer.front().second to output  
    std::cout << reorder_buffer.front().first << "\n";  
    ++current;  
    std::pop_heap(reorder_buffer.begin(), reorder_buffer.end(),  
                  [](auto &a, auto &b) { return a.first > b.first; });  
    co_await pool_writer.write(std::move(reorder_buffer.back().second));  
    reorder_buffer.pop_back();  
}  
  
}  
  
}
```


MAIN - SETUP

```
std::vector<channel_runner> runners(threads);
```

```
auto inchan = std::make_shared<channel<ordered_payload_type>>();
```

```
auto pool = std::make_shared<channel<payload_type>>(threads);
```

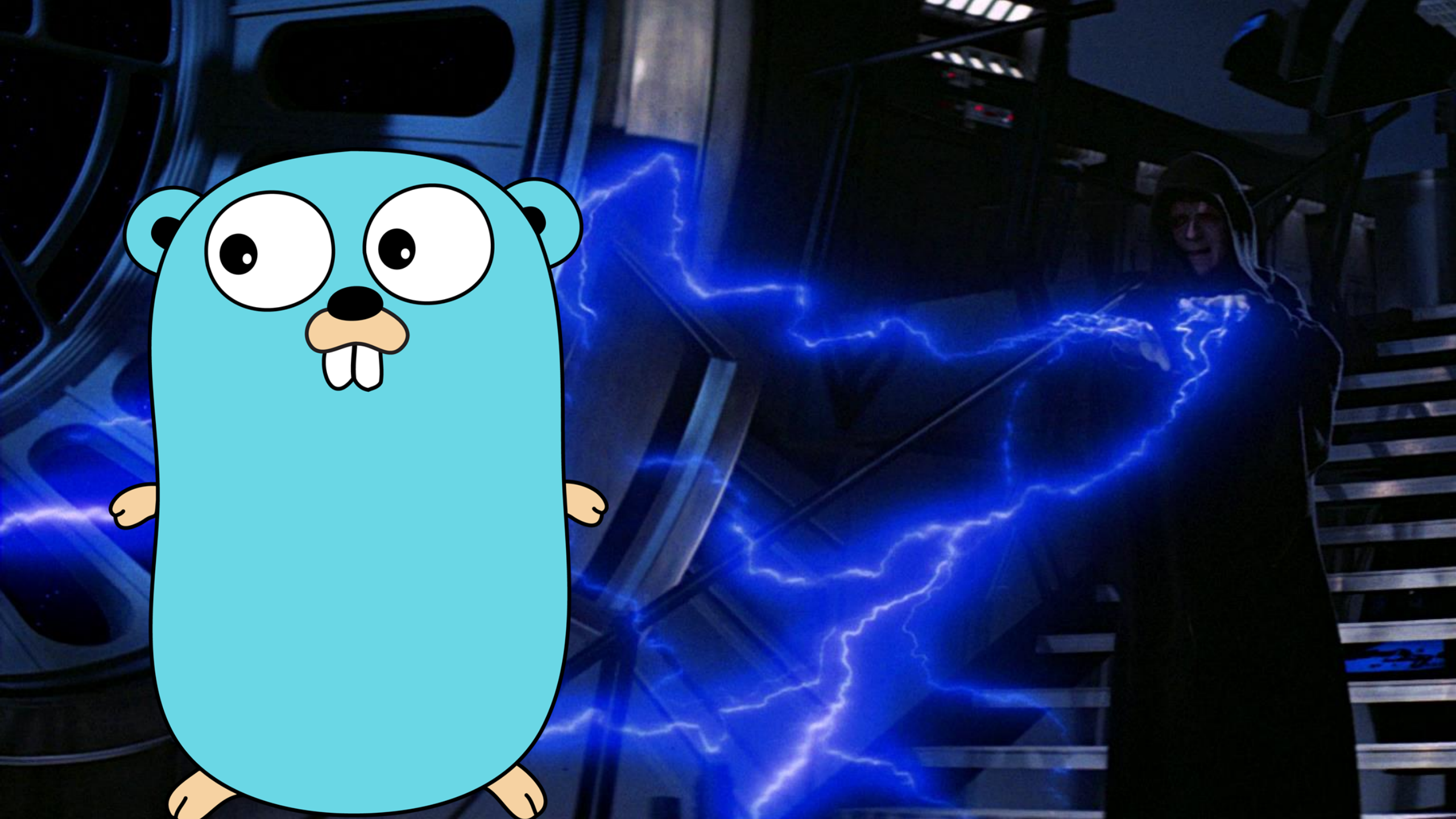
```
auto done_chan = std::make_shared<channel<bool>>();
```

MAIN – CALL READER, PROCESSORS, WRITER

```
reader(inchan, pool, count);  
vector<shared_ptr<channel<ordered_payload_type>>>  
    ordered_payload_chans;  
for (int i = 0; i < threads; ++i) {  
    ordered_payload_chans.push_back(make_processor(inchan));  
}  
writer(std::move(ordered_payload_chans), pool, done_chan);
```

MAIN – SET UP POOL AND WAIT FOR COMPLETION

```
thread_suspender sus;  
sync_channel_reader<bool, thread_suspender> done_reader{done_chan, sus};  
sync_channel_writer<payload_type, thread_suspender> pool_writer{pool, sus};  
  
for (int i = 0; i < threads; ++i) {  
    payload_type payload = std::make_unique<array_type>();  
    pool_writer.write(std::move(payload));  
}  
  
done_reader.read();
```



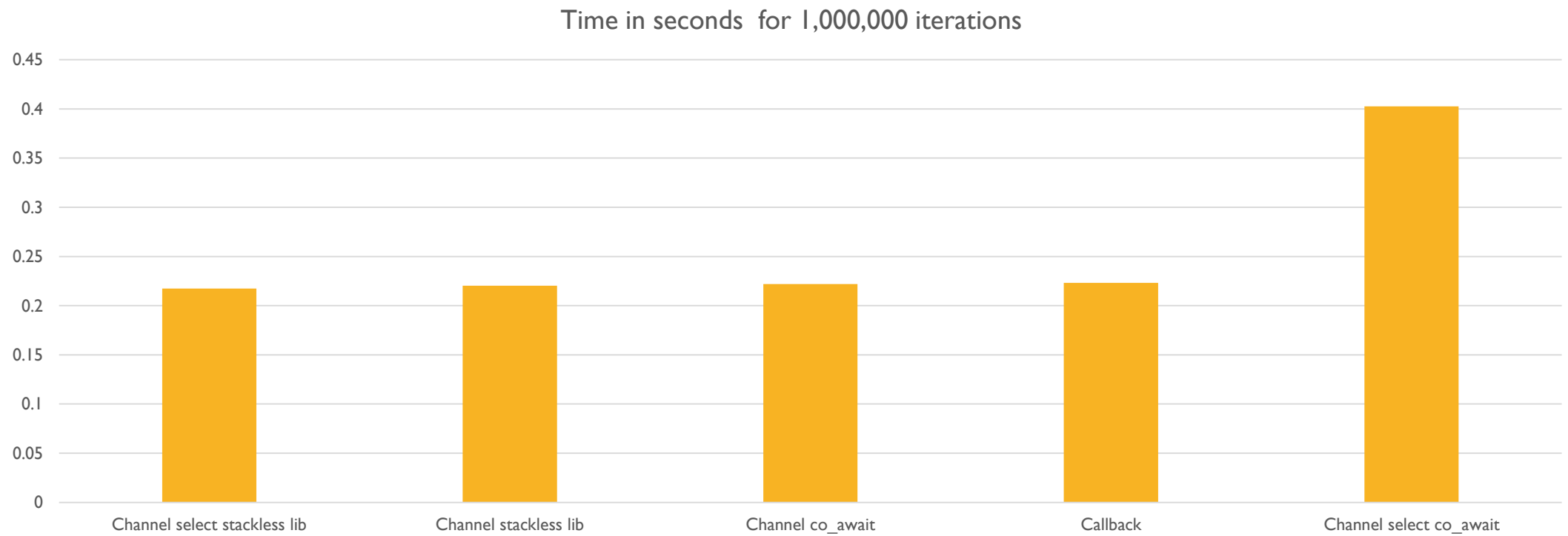
MORE INFORMATION

- Concurrency TS
 - CppCon 2015 – Artur Laksberg – Concurrency TS: Editors Report <https://www.youtube.com/watch?v=mPxlegd9J3w>
- Coroutines
 - Multiple sessions by Gor Nishanov and team this week
 - CppCon 2015 – C++ Coroutines: A negative overhead abstraction https://www.youtube.com/watch?v=_fu0gx-xseY
- Go Channels
 - Go Channels on Steroids Dmitry Vyukov
<https://docs.google.com/document/d/1yIAYmbvL3JxOKOjuCyon7JhW4cSvIwy5hC0ApeGMV9s/pub>

CODE

- https://github.com/jbandela/stackless_coroutine/blob/channel_dev/channel.hpp
- https://github.com/jbandela/stackless_coroutine/blob/channel_dev/perf_future_vs_channel.cpp
- https://github.com/jbandela/stackless_coroutine/blob/channel_dev/channel_example.cpp

PERFORMANCE



CHANNEL WRITER – STACKLESS LIBRARY

```
template <class Context>
void write(T t, Context context) {
    node.value = std::move(t);
    node.func = [](detail::node_base *n) {
        auto node = static_cast<node_t *>(n);
        auto context = Context::get_context(node->data);
        context(node, node->closed);
    };
    node.data = context.to_address();
    ptr->write(&node);
}
```


STACKLESS COROUTINE - READER SELECT

```
auto reader_select(shared_ptr<channel<int>> chan1, shared_ptr<channel<int>> chan2) {  
    struct values {  
        channel_reader<int> reader1; channel_reader<int> reader2;  
        channel_selector selector;    std::mutex mut;  
        values(shared_ptr<channel<int>> c1, shared_ptr<channel<int>> c2)  
            : reader1{c1}, reader2{c2} {}  
    };  
    auto co = stackless_coroutine::make_coroutine<values>(  
        stackless_coroutine::make_block(...),  
        [](auto &&...) {}, chan1, chan2);  
    co();  
}
```

MAKE_BLOCK

```
stackless_coroutine::make_while_true(  
    [](auto &context, values &variables) {  
        std::unique_lock<std::mutex> lock{variables.mut};  
        variables.reader1.read(variables.selector, context);  
        variables.reader2.read(variables.selector, context);  
        return context.do_async();  
    },  
    [](auto &context, values &variables, auto... v) {  
        {std::unique_lock<std::mutex> lock{variables.mut};}  
        auto sel = get_selector(v...);  
        sel.select(variables.reader1, [](auto&&){}).select(variables.reader2, [](auto&&){});  
        if (!sel) return context.do_break(); else return context.do_continue();  
    })
```