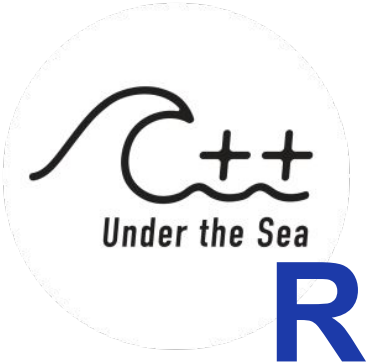


Refactoring Towards Structured Concurrency

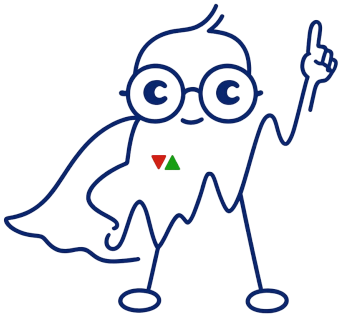
C++ Under the Sea 2026

Roi Barkan

Slides

Hi, I'm Roi

- Roi Barkan (he/him) - רועי ברקן
- I live in Tel Aviv
- C++ developer since 2000
- SVP Development & Technologies @ Istra Research
  - Finance, Low Latency, in Israel
  - careers@istraresearch.com
- Always happy to learn and explore
  - Please - ask questions, make comments



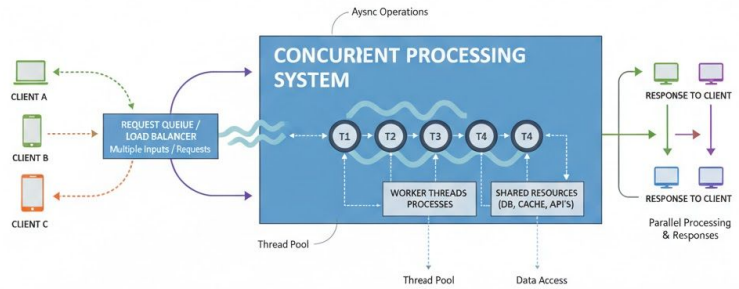
Slides

Outline

- Concurrency in High Performing Systems
  - Definitions
  - Motivations
  - Techniques
- Structured Concurrency
  - What
  - Why
  - How
- Migration Paths

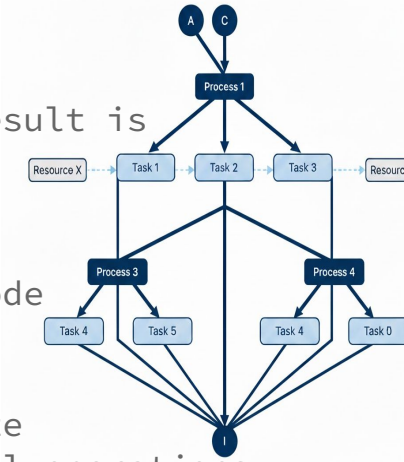
Concurrency in Software Systems

- Computers and algorithms were first sequential
  - Every instruction followed its predecessor and could depend on its result
- The environment is not so simple:
  - Compute - multiple processors/computations can run at the same time
  - Inputs / events - online systems receive many inputs / requests as they run
    - Requests can be unrelated or linked



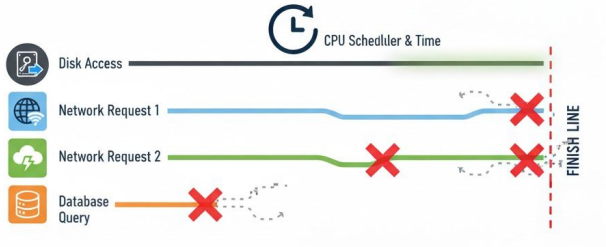
What is Concurrency in Code

- Definition: ways to describe dependencies between instructions/components
- Possible building blocks:
  - Non-blocking API - a function returns before its effect / result is complete
    - Completion callback - ‘continuation passing style’
    - Returns an identifier not the result (“future”).
  - “Fork” - non-blocking API to perform arbitrary function / code
    - (not only O/S level process creation)
    - In C++: **std::jthread** and **std::async.**
  - “Join” - do not continue before another operation is complete
  - Mutex / Semaphore / Barrier - Coordinate potentially parallel operations
  - Cancellation - signal some result isn’t needed
    - In C++: **std::stop_source** / **std::stop_token**



Motivation - What's the Point

- Exploit parallelism - processors, peripherals, network
- Separation of concerns
  - Independently develop components that share a system
  - Manage/configure priorities, fairness and scheduling as a separate challenge.
- Tail latency hiding - 'race' several components, let the winner win.
- Efficient batching / chunking - deal with multiple requests together, especially when overall load is high.



The Challenge: Correctness in Every Scenario

- Extraordinary number possible situations to handle.
  - Full test coverage seemingly impossible.
- Systematic synchronization (mutexes, barriers, …) might defeat the performance gain.
  - And also easy to get wrong
  - “No Raw Synchronization Primitives” Sean Parent.
- Goal: correct by construction.
  - Simple rules and design → easy code-reviews (of human or agent code).
  - Analogous to the way value-semantics and RAII solve object lifetime.
- Approach: composability, local reasoning.

No Raw Synchronization Primitives

YouTube

Techniques for Concurrency - Broad Categories

- Operating system threads (or processes).
  - O/S performs scheduling, preemption, context-switches and hardware interaction.
  - Pros: Simple. Supported since C++11.
  - Cons: High overhead. Little control. Not scalable.
- Virtual threads (green/user threads, stackful coroutines, fibers)
  - Lightweight ‘cooperative’ scheduling and switching between multiple stacks.
  - Pros: Similar coding to O/S threads with lower overhead
  - Cons: No standard support in C++, not zero overhead.
- Reactor/Proactor thread (or thread-pool)
  - Work is split to non-blocking tasks/events, generic thread(s) handles tasks as they arrive
  - Pros: Low overhead, no O/S required. Frequently, don’t parallelize related tasks, so they don’t need mutexes.
  - Cons: Potentially complicated - how does one split the work to tasks ?

Splitting Work into Tasks

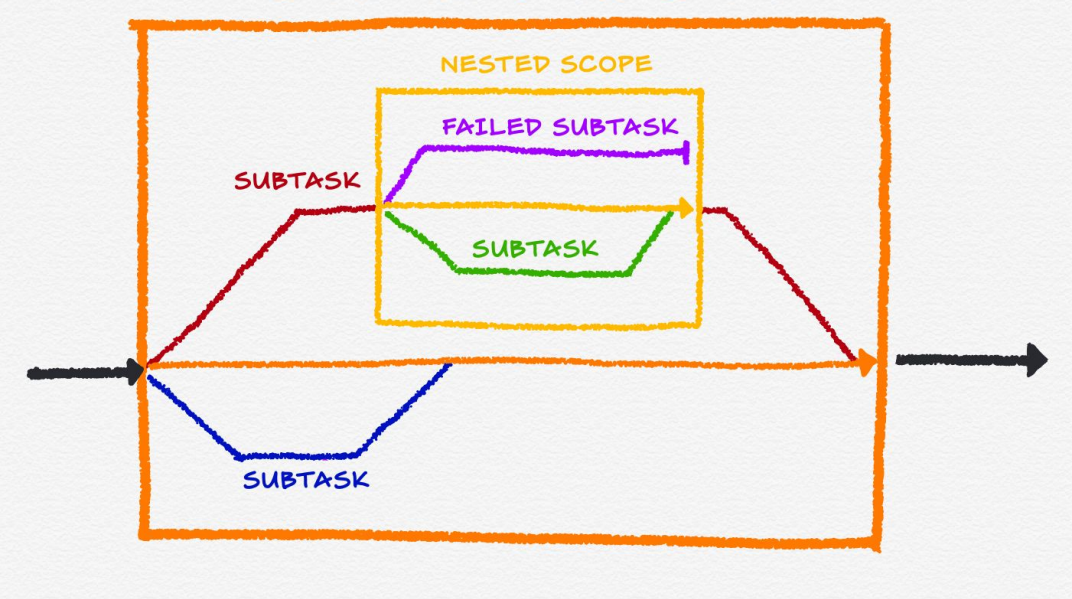
- State machine - implement each independent flow as a state-machine object, reacting to its own event
  - Highly suitable for multi-stage protocol implementations (handshake, session control, etc.)
  - Becomes tricky if state-chart is complex or when there's cross-flow dependence
  - Typical code: reactor does map-lookup of the state, and passes the event to the state.
- Callbacks / Continuation Passing Style (CPS) - async operations, with callbacks for completion
  - Old-school boost.asio library.
  - Fits well with lambda / **std::function** - squint correctly and see sequential code…
  - hard to manage object lifetimes.
- Structured concurrency is meant to improve on the above.

Continuation Passing Style

```
struct session : public enable_shared_from_this<session> {
    session(tcp::socket socket) : socket_(std::move(socket)) {}
    void start() { do_read(); }
  private:
    void do_read() {
        auto self(shared_from_this());
        socket_.async_read_some(
            asio::buffer(data_, max_length),
            [this, self](std::error_code ec,
                         std::size_t length) {
                if (!ec) { do_write(length); }
            });
        /* code here is probably a bug */
    }
    void do_write(std::size_t length) {
        auto self(shared_from_this());
        asio::async_write(
            socket_, asio::buffer(data_,
                                  length),
            [this, self](std::error_code ec,
                         std::size_t /*length*/) {
                if (!ec) { do_read(); }
            });
    }

    tcp::socket socket_;
    enum { max_length = 1024 };
    char data_[max_length];
};
```

Structured Concurrency



Ben Weidig

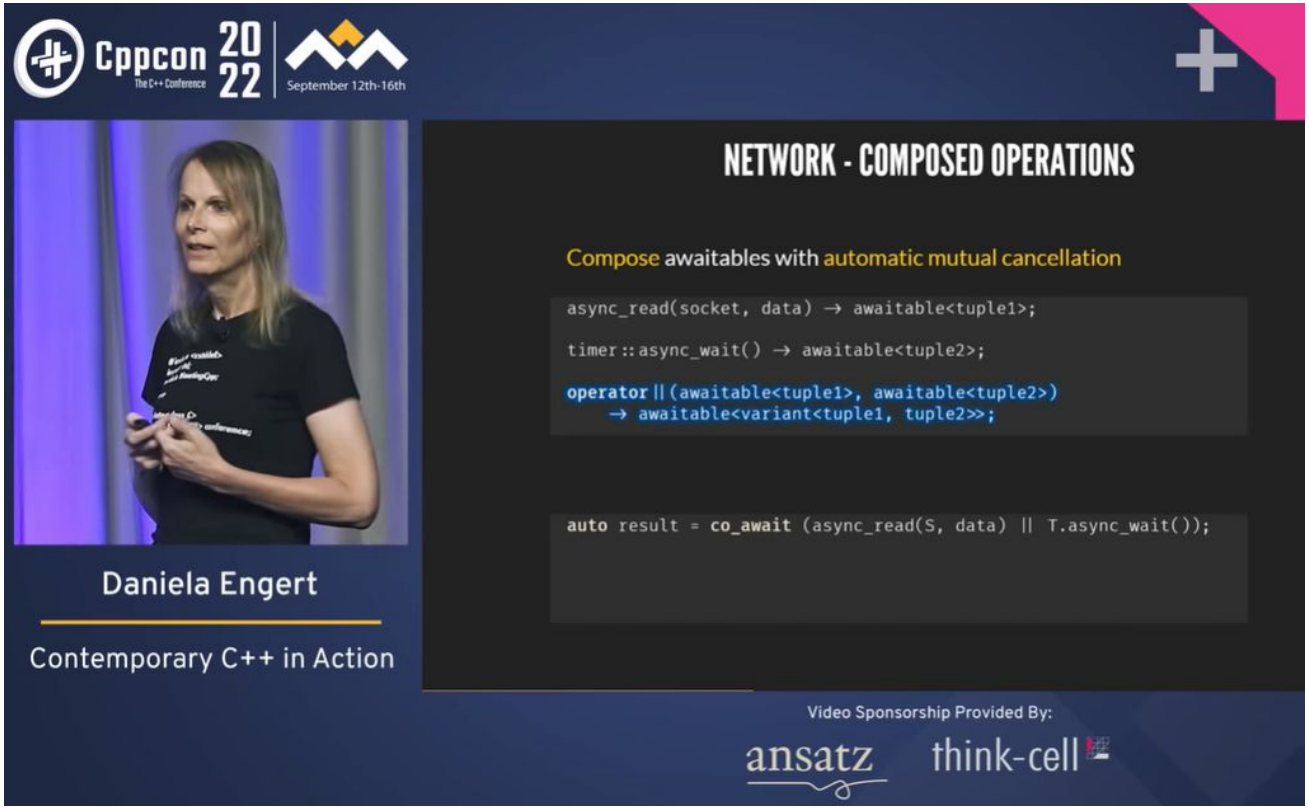
Structured Concurrency Philosophy

- Tasks are resources with lifetimes - use RAII to control them
  - Every task ‘belongs’ to a scope.
  - Sub-tasks typically belong to a sub-scope.
  - Scope’s outlive their tasks
  - Analogy: **std::jthread** vs. **std::thread**.
- Main advantages:
  - Like RAII, exiting a scope (‘}’ character) typically means all tasks defined within are either done (joined) or returned (**std::move**’d up)
  - Hierarchical, composable - tree-like structure means less need for shared_ptr and lifetime management
  - Intuitive propagation of cancellation, error handling and cleanup.

Coroutines are Structured Concurrency

- Coroutine mechanics:
  - Invocation creates a “coroutine frame” (and a return object)
  - **co_await** can suspend the call, keeping the entire state (variables) and the *awaiter* (if any)
  - Once suspended, a coroutine can be resumed or destroyed - both involving the *awaiter*.
- The *awaiter* objects can point to child coroutine(s) (sub-tasks), facilitating the structured invariants
  - Destructor can make sure children are canceled/completed
  - **await_resume()** can handle error scenarios as well as successful completions.
- one-to-many task spawns is also possible by composing awaitables
  - asio has experimental **operator&&** and **operator||** for composition.

Daniela Engert on Coroutine Composition

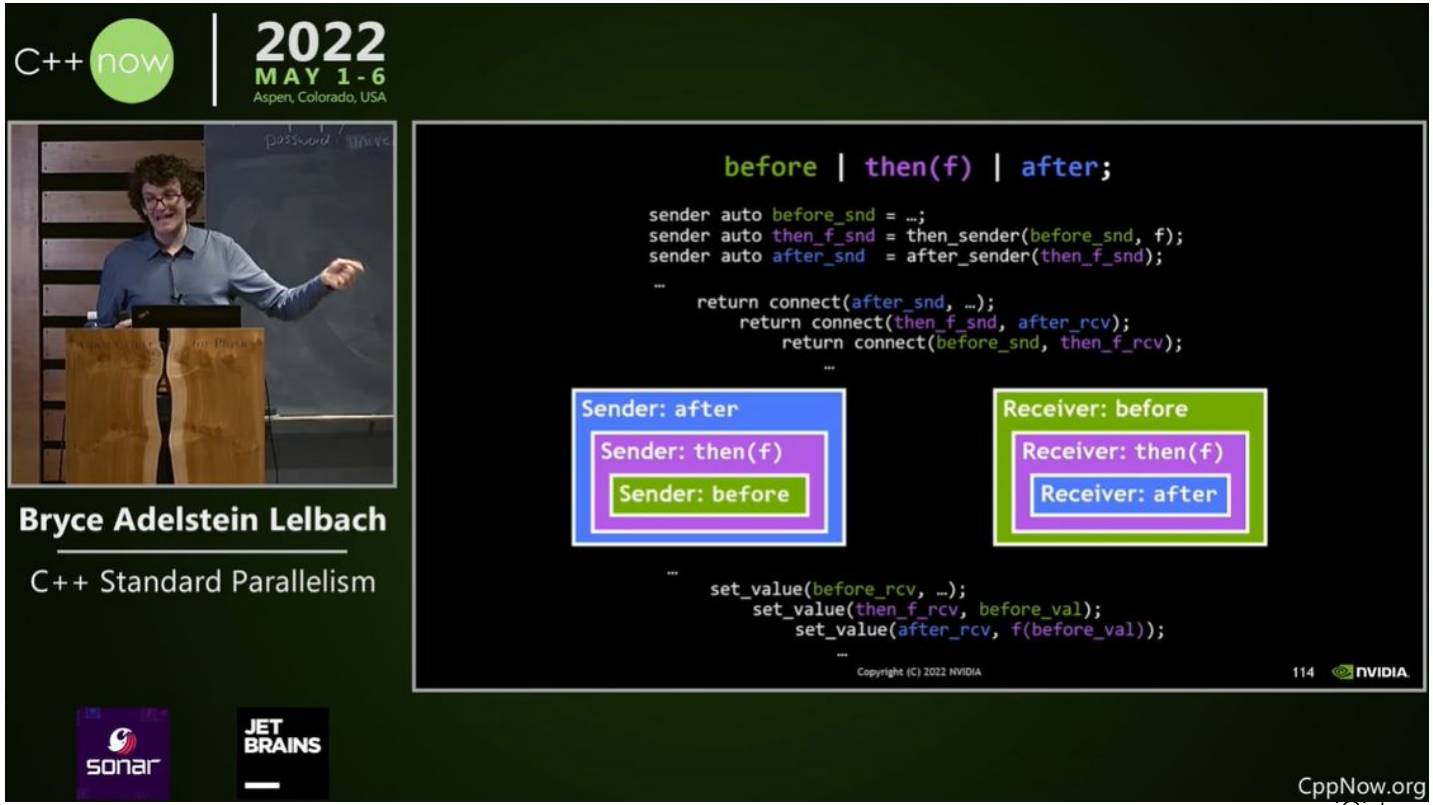


YouTube

Senders are Structured Concurrency

- C++26 senders represent an asynchronous computation, with multiple potential results
- The core operation on senders is composition, via various sender-adaptors
  - Sender adaptors guarantee structured concurrency
  - Similar to range-adaptors - composition creates expression-templates: an “onion” object describing the layers of computation.
- Senders are more “bare-bones” than coroutines and are more capable:
  - More direct interaction with execution resources - less need for awaitables, frames, etc.
  - More control of memory layout, thus no need for heap allocations.
- Key insight - senders are awaitables and awaitables are senders !!!

Bryce Adelstein Lelbach - The Sender/Receiver Onion



YouTube

Coroutines and Sender-Chains Show the Flow

```
asio::awaitable<void> read(tcp::socket socket,
                           std::span<std::byte> span) {
  while (!span.empty()) {
    std::size_t bytes = co_await
                        socket.async_read_some(
                          asio::buffer(span),
                          asio::use_awaitable);
    if (bytes == 0) throw eof_error{};
    span = span.subspan(bytes);
  }
}
```

Asio

```
template<typename Readable>
auto read(Readable& readable, const std::span<std::byte> span) {
  return
    std::execution::just(span) |
    std::execution::let_value([&readable](
      std::span<std::byte>& span)
    {
      return ::exec::repeat_effect_until(
        std::exeutiion::just() |
        std::execution::let_value([&readable, &span]() {
          return readable.read(span);
        }) |
        std::execution::then([&span](std::size_t bytes) {
          if (!bytes) {
            throw eof_error{};
          }
          span = span.subspan(bytes);
          return span.empty();
        }));
    });
}
```

Robert Leahy

Summary So Far - Questions are Welcome

- Concurrent systems can handle complex scenarios of incoming events
- Achieving high performance can mean eliminating helper facilities like threads, stacks and even heap allocations
- Older approaches are error prone and require attention to details
  - Managing object lifetime
  - Handling complex scenarios
  - Errors and cancelation.
- Structured concurrency is a composable low-overhead approach
  - Both coroutines and the senders (std::execution) frameworks provide structured concurrency
  - Both frameworks interoperate - we don’t have to choose one.

Migrating to Structured Concurrency

No Silver Bullet - Just Tips

- Import an existing library, don't start from scratch
  - ASIO - inside or out of boost
  - cppcoro - various coroutine building blocks
  - cobalt - boost coroutine library, with asio compatibility
  - Nvidia stdexec - battle-tested senders library
  - beman.execution - senders aiming for standard conformance
  - beman.net - senders for networking proposed for standartization
- Keep control of your main-loop if possible
  - **asio::io_context::poll** - non-blocking task drainer

Where to Start

- Try coroutines
  - Already in the standard
  - Quite a few libraries exist.
- If current code is non-blocking: consider starting from the root, not leaf
  - integrate “service”, change a single **return** to **co_return**, add **co_await**.
- If current code is blocking (threads): go from leafs towards to root
  - function-coloring is an issue - be creative.
  - Careful with blocking APIs, **mutex**’s.
- Don’t fear cancellation
  - If some old code needs to interact with a suspended coroutine - consider cancellation
  - Alternatively be creative with the **co_await** return type (**std::expected** ?)

Summary

- Structured concurrency is a paradigm shift
- Used correctly - it claims readability, safety and low overhead
- As of C++26 we are will have two ways for Structured Concurrency
  - Coroutines - still relying on third party helper libraries
  - Senders - fresh out of the box
- Now we can start our preparation and migration

Slides

- Thank you !!
  - Questions and comments are welcome

Audience Question - Migrating a State Machine System

- Simple way to convert each state machine to:

```
task<void> runMachine(StateMachine m, EventSource& source) {
  while (not m.done())
    m.handleEvent(co_await source);
}
```

- If machine handles multiple event types, **source** can emit a **std::variant**, which we can **visit()**
- Coroutine to sender:
  - The **while**-loop can become the **repeat_effect_until()** sender adaptor
  - **visit()** can become **when_any()** sender adaptor

THANK YOU

More Coroutines vs. Sender-Chains

```
asio::awaitable<void> echo(tcp::socket socket) {
  try {
    char data[1024];
    for (;;) {
      std::size_t n = co_await
                      socket.async_read_some(
                          asio::buffer(data),
                          asio::use_awaitable);
      co_await async_write(socket,
                           asio::buffer(data, n),
                           asio::use_awaitable);
    }
  } catch (std::exception& e) {
    std::printf("echo Exception: %s\n", e.what());
  }
}
```

Asio

```
template<typename Readable>
auto read(Readable& readable, const std::span<std::byte> span) {
  return
    std::execution::just(span) |
    std::execution::let_value([&readable](
      std::span<std::byte>& span)
    {
      return ::exec::repeat_effect_until(
        std::exeutiion::just() |
        std::execution::let_value([&readable, &span]() {
          return readable.read(span);
        }) |
        std::execution::then([&span](std::size_t bytes) {
          if (!bytes) {
            throw eof_error{};
          }
          span = span.subspan(bytes);
          return span.empty();
        }));
    });
}
```

Robert Leahy

Sender Samples

Timeout

```
1 auto s =
2     | thing
3     | async::timeout_after(5ms, error{42})
4     | async::then( [](auto v) { /* ... */ } )
5     | async::upon_error( [](auto e) { /* ... */ } )
6     ;
```

Michael Caisse

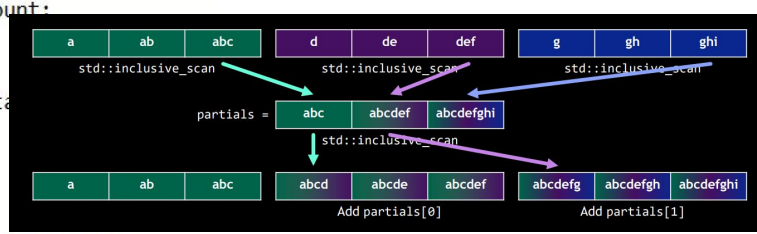
Sender Samples

```
1  auto delay = [](auto v) {
2    return async::continue_on(async::time_scheduler{v});
3  };
4
5  auto led_on = async::then([](){ set_led(true); });
6  auto led_off = async::then([](){ set_led(false); });
7  auto on_cycle  = led_on  | delay(1s);
8  auto off_cycle = led_off | delay(300ms);
9
10 async::sender auto blinky =
11     async::just()
12   | on_cycle
13   | off_cycle
14   ;
15
16 auto s = blinky | async::repeat() | async::start_detached();
```

Michael Caisse

Samples

```
1  inline constexpr sender_adaptor inclusive_scan_async =
2    [](ex::sender auto last, auto init,
3         std::size_t tile_count) -> ex::sender auto {
4     return last
5       | ex::then([=] (stdr::random_access_range auto input) {
6        std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
7        partials[0] = init;
8        return send_values(input, std::move(partials));
9       })
10      | ex::bulk(tile_count,
11       [=] (std::size_t i, auto input, auto partials) {
12       auto tile_size = (input.size() + tile_count - 1) / tile_count;
13       auto start = i * tile_size;
14       auto end = std::min(input.size(), (i + 1) * tile_size);
15       partials[i + 1] = *--std::inclusive_scan(begin(input) + sta
16       begin(input) + end,
17       begin(input) + start);
18     })
19      | ex::then([] (auto input, auto partials) {
20       std::inclusive_scan(begin(partials), end(partials), begin(partials));
21       return send_values(input, std::move(partials));
22     })
23      | ex::bulk(tile_count,
24       [=] (std::size_t i, auto input, auto partials) {
25       auto tile_size = (input.size() + tile_count - 1) / tile_count;
26       auto start = i * tile_size;
27       auto end = std::min(input.size(), (i + 1) * tile_size);
28       std::for_each(begin(input) + start, begin(input) + end,
29         [&] (auto& e) { e = partials[i] + e; });
30      })
31      | ex::then([=] (auto input, auto partials) { return input; });
32   };
```



Bryce Adelstein Lelbach

roi@istraresearch.com

Adaptors

Sender Adaptor	Semantics Of Returned Sender
then(sender auto last, invocable auto f)	Call f with the value sent by last.
bulk(sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.
transfer(sender auto last, scheduler auto sch)	Transition to sch for the next sender.
split(sender auto last)	Can be connected multiple times.
when_all(sender auto... inputs)	Combines multiple senders into an aggregate.
ensure_started(sender auto last)	Connects and starts last.

Sender Factories	Semantics Of Returned Sender
schedule(scheduler auto sch)	Completes on sch.
just(T&&... ts)	Send the values ts.

Sender Consumers	Semantics
sync_wait(sender auto snd) -> *values-sent-by-sender*	Block until snd completes and return or throw whatever it sent.

Copyright (C) 2022 NVIDIA

NVIDIA

Bryce Adelstein Lelbach

broi@istraresearch.com