P2300R0

STD::EXECUTION

Michael Garland, June 2021

AUTHORS

Michał Dominiak,

Lewis Baker,

Lee Howes,

Michael Garland,

Eric Niebler &

Bryce Adelstein Lelbach

GOALS OF THIS PAPER

Produce a single, usable design for controlling execution in C++

Refine design of core features in P0443 based on feedback from LEWG & SG1

Consolidate selected features needed to write effective code, especially from:

- P2181r1 *Correcting the design of bulk execution*
- P1897r3 *Towards C++23 executors: A proposal for an initial set of algorithms*
- P2175r0 *Composable cancellation for sender-based async operations*

Produce a self-contained, well-documented design suitable for inclusion in C++23

HELLO, P2300

Preparing work to be executed on a thread pool & waiting for its completion

```
using namespace std::execution;

scheduler auto sch = my_thread_pool().scheduler();

sender auto begin = schedule(sch);

sender auto hi_again = then(begin, []{
    std::cout << "Hello world! Have an int.";
    return 13;
});

sender auto add_42 = then(hi_again, [](int arg) { return arg + 42; });

auto [i] = std::this_thread::sync_wait(add_42).value();
```

USER-FACING DESIGN

USER NEEDS

Our design provides mechanisms to manage three key concerns

Control *where* work executes

Manage *composition* of asynchronous operations

Control *submission* of work for execution

CONCEPT: SCHEDULERS

Schedulers are handles representing the context in which work will be executed

```
// Schedulers are obtained from context-specific interfaces.
scheduler auto sch = my_thread_pool().scheduler();
```

A single context object, such as a thread pool, may provide multiple different schedulers encapsulating different execution strategies.

CONCEPT: SENDERS

Senders describe work to be executed

An object that represents a computation to be performed and which, upon completion, sends zero or more values.

Senders may alternatively send an *error* or *done* signal in lieu of values.

(More on this later.)

ASSEMBLING DESCRIPTIONS OF WORK

Sender factories initiate the description of a computation

Sender Factories in P2300

`schedule(scheduler auto sch) -> sender`	Returned sender completes on the given scheduler
`just(auto ...values) -> sender`	Returned sender completes and sends the given values
`transfer_just(scheduler auto, auto ...) -> sender`	Returned sender completes on the given scheduler and sends the given values

ASSEMBLING DESCRIPTIONS OF WORK

Sender factories initiate the description of a computation

```
// Produces a sender that completes immediately
// and sends the value 1001.
sender auto a = just(1001);

// Senders may send more than one value ...
sender auto b = just(1001, 1002, 1003);

// ... or send none at all.
sender auto c = just();
```

ASSEMBLING DESCRIPTIONS OF WORK

Sender consumers are terminal points in the description of a computation

Sender Consumers in P2300

`std::this_thread::sync_wait(sender auto snd) -> std::optional<std::tuple<`*`value-types`*`...>>`	Submit computation represented by `snd` for execution. Block the current thread waiting for completion. Return value(s) sent by `snd`. If an *error* is sent, throw exception. If *done* is sent, return an empty optional.
`std::execution::start_detached(sender auto snd) -> void`	Submit computation represented by `snd` for execution. Any values it sends will be discarded. If an error is sent, `std::terminate` is called.

ASSEMBLING DESCRIPTIONS OF WORK

Sender consumers are terminal points in the description of a computation

```
// Produces a sender that completes immediately and sends the value 1.
sender auto one = just(1);

// Wait for completion and obtain the result.
auto [x] = std::this_thread::sync_wait(one).value();

// x == 1 at this point.
```

ASSEMBLING DESCRIPTIONS OF WORK

Sender adaptors enable composing computations from individual pieces

Selected Sender Adaptors in P2300
`then`
`let_value`
`bulk`
`when_all`
`split`

Adaptors accept senders as parameters and return a sender as output.

Sender adaptors are analogous to *range* adaptors

Including support for pipe operator syntax

```
// These are equivalent:
then(snd, ...);
then(...)(snd);
snd | then(...);
```

THEN

Continuing a computation with a single invocable object

Sender Adapters

`then(sender auto snd, invocable f) -> sender`	Returned sender sends the result of `f` invoked with the values sent by `snd`. If `snd` does not send values, sends what `snd` sends.

```
sender auto a = just(100);
sender auto b = then(a, [] (int x) { return 2 * x; });

auto [x] = std::this_thread::sync_wait(b).value();  // x == 200
```

LET_VALUE

Extending a computation with a sender factory

Sender Adapters

`let_value(sender auto snd, invocable f) -> sender`	Will invoke `f` with values sent by `snd`, resulting in a sender. If `snd` doesn't send values, `f` is not invoked. Returned sender will send what that resulting sender does.

```
sender auto sends_A = ...
sender auto snd = sends_A
                | let_value([] (auto& A) {
                    return schedule(my_thread_pool().scheduler())
                        | then([&A] { })
                        | then([&A] { })
                        ...
                  });
```

`let_value` ensures that lifetime of delivered objects outlasts computation described by `f`.

BULK

Extending a computation with a bulk section

Sender Adaptors

`bulk(sender auto snd, std::integral auto size, invocable f) -> sender`	Returned sender completes after invoking `f` with `(i, xs...)` for all `i` in `[0, size)` where `xs...` are the values sent by `snd`, and delivers those same values. If `snd` does not send values, sends what `snd` sends.

```
std::vector in = {2, 3, 0, 0};
int n = in.size();

sender auto update = just(std::move(in))
                   | bulk(n, [](int i, std::vector<int>& x) { x[i] += 1; })
                   | bulk(n, [](int i, std::vector<int>& x) { x[i] += 1; });

auto [out] = std::this_thread::sync_wait(update).value(); // out == {4, 5, 2, 2}
```

WHEN_ALL

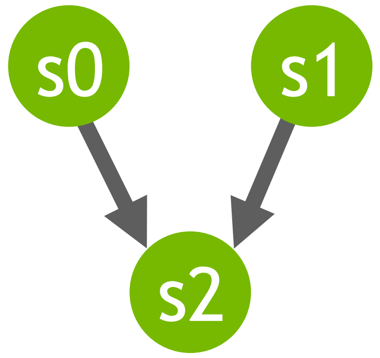
Joining results of multiple computations together

Sender Adapters

`when_all(sender auto ...senders) -> sender`	Returned sender completes once all given senders have completed and sends the values sent by all of them, in the same order they are listed.
`transfer_when_all(scheduler auto sch, sender auto ...senders) -> sender`	Behaves like `when_all`, and returned sender completes on `sch`.

```
sender auto s0 = just(1000);
sender auto s1 = just("hello"sv);
sender auto s2 = when_all(s1, s2);

auto [x, y] = std::this_thread::sync_wait(s2).value();
```



SPLIT

Used when multiple computations have a common predecessor

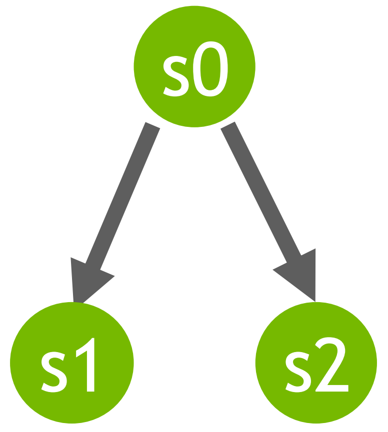
Sender Adaptors

`split(sender auto snd) -> sender`	Returned sender delivers values equivalent to those sent by `snd`. May be `snd` itself or a new sender, as appropriate.

```
sender auto s0 = just(2021);

sender auto fork = split(std::move(s0));

sender auto s1 = fork | then([] (int y) { printf("Left %d\n", y); });
sender auto s2 = fork | then([] (int y) { printf("Right %d\n", y); });
```



WHERE DO SENDERS COMPLETE?

Senders may advertise a completion scheduler

`template <typename signal_cpo>` `get_completion_scheduler(sender auto snd) -> scheduler`	If well-formed, the sender `snd` ensures that it sends the indicated signal on an execution agent belonging to the context represented by the returned scheduler.

Controlling where senders complete with values

`schedule(scheduler auto sch) -> sender`	Produces a scheduler whose completion scheduler is `sch`.
`transfer(sender auto snd, scheduler auto sch)` `    -> sender`	Return a sender that sends the values sent by `snd`, but whose completion scheduler is `sch`. Does not change scheduler that `snd` starts on.
`on(scheduler auto sch, sender auto snd)` `    -> sender`	Starts `snd` on `sch` and returns a sender that sends the values sent by `snd`, but which has no completion scheduler.

USING TRANSFER

Transfer changes completion scheduler when describing work

```
sender auto initiate = schedule(sch1);

                                                 // Assuming no errors:
sender auto work = initiate                      // .. completes on sch1
    | then([]{ printf("On sch1\n"); })           // .. completes on sch1
    | transfer(sch2)                             // .. completes on sch2
    | then([]{ printf("On sch2\n"); });          // .. completes on sch2

std::this_thread::sync_wait(work);
```

USING ON

On specifies scheduler on which work already described will start

```
sender auto work = just()
                 | then([]{ printf("Running\n"); });

// Work will start on sch1.
// (and also complete on sch1, in this example)
sender auto initiate = on(sch1, work);


std::this_thread::sync_wait(initiate);
```

SENDING MORE THAN VALUES

Analogues of `then` and `let_value` for error and done signals

Sender Adapters

`upon_error(sender auto snd, invocable f) -> sender`	Returned sender sends the result of `f` invoked with an error delivered by `snd`. Sends what `snd` sends, otherwise.
`upon_done(sender auto snd, invocable f) -> sender`	Returned sender sends the result of invoking `f` invoked when `snd` completes with done. Sends what `snd` sends, otherwise.
`let_error(sender auto snd, invocable f) -> sender`	Will invoke `f` with the error sent by `snd`, resulting in a sender. If `snd` doesn't send an error, `f` is not invoked. Returned sender will send what that resulting sender does.
`let_done(sender auto snd, invocable f) -> sender`	Will invoke `f` if `snd` sends done, resulting in a sender. If `snd` doesn't send done, `f` is not invoked. Returned sender will send what that resulting sender does.

EXECUTE

One-way execution of invocable objects on schedulers

```
scheduler auto sch = my_thread_pool().scheduler();

// Submitting a function with no arguments that returns no results
// to be run in the execution context represented by the scheduler.
execute(sch, [] { printf("Running on thread pool\n"); });
```

This operation fills the role of one-way executors from P0443.

WHEN IS WORK SUBMITTED?

Default interface supports lazy submission but permits potentially eager submission

Customizations of algorithms are permitted to be *potentially eager*, as in P0443 and other papers

Lazy implementations of potentially eager algorithms are *always* valid

Strictly lazy semantics (i.e., work is never submitted until explicitly started) are chosen by name

Regardless of submission point, required ordering of sender completions *must never* be altered

WHEN IS WORK SUBMITTED?

Default interface supports lazy submission but permits potentially eager submission

```
sender auto a = ...some sender...;

sender auto b = then(a, []{ printf("b\n"); });
// a may be submitted for execution already,
// but must never execute before b completes.

sender auto c = lazy_then(a, []{ printf("c\n"); });
// c must not be submitted for execution yet.



// b may have been submitted before this point.
std::this_thread::sync_wait(b);

// c must not be submitted before this point.
std::this_thread::sync_wait(c);
```

FUTURE WORK

We envision composing with parallel algorithms via operators not yet defined

An existing parallel algorithm of the form:

```
algorithm(ExecutionPolicy&& policy, …) -> T
```

could be executed on a specific scheduler by combining a scheduler with a policy:

```
algorithm(executing_on(scheduler, policy), ...) -> T
```

and a new asynchronous form of the algorithm could be achieved by combining with senders:

```
async_algorithm(executing_async(sender, policy), ...) -> sender
```

```
sender auto async_read(sender auto buffer, auto handle);
```

Separately defined asynchronous algorithm returns a sender.

```
struct dynamic_buffer { std::unique_ptr<std::byte[]> data; std::size_t size; };


sender auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([] (dynamic_buffer& buf) {
           return just(std::as_writeable_bytes(std::span(&buf.size, 1))
                | async_read(handle)
                | then(
                    [&] (std::size_t bytes_read) {
                      assert(bytes_read == sizeof(buf.size));
                      buf.data = std::make_unique(new std::byte[buf.size]);
                      return std::span(buf.data.get(), buf.size);
                    })
                | async_read(handle)
                | then(
                    [&] (std::size_t bytes_read) {
                      assert(bytes_read == buf.size);
                      return std::move(buf);
                    })
       });
}
```

let_value() responsible for managing lifetime of dynamic_buffer

Composition with external async. algorithm

Returned sender delivers a filled dynamic_buffer upon completion.

IMPLEMENTER'S INTERFACE

CONCEPT: RECEIVERS

Receivers are the “glue” between senders

Senders represent continuable computations.

Receivers are the continuations to which they send values.

Receivers provide three channels for receiving completion signals from a sender:

- `set_value(receiver auto recv, auto Ts...) -> void`
- `set_error(receiver auto recv, auto err) -> void`
- `set_done(receiver auto recv) -> void`

CONCEPT: RECEIVERS

Receivers are the “glue” between senders

Senders represent continuable computations.

Receivers are the continuations to which they send values.

Receiver contract:

- Exactly one of these must be successfully invoked on a receiver before it is destroyed.
- If a call to `set_value` fails with an exception, either `set_error` or `set_done` must be invoked on the same receiver.

RECEIVERS ARE FOR IMPLEMENTERS

Receivers are the “glue” between senders

Senders represent continuable computations.

Receivers are the continuations to which they send values.

The `connect` algorithm binds them together.

Design principle: Neither receivers nor `connect` should appear in typical user code; they exist for implementers of senders and low-level operations on senders

SENDERS & RECEIVERS

Connecting a sender to a receiver produces an operation state, which can be started

```
sender auto snd = ...some sender...;
receiver auto rcv = ...some receiver...;


// Connecting a receiver tells a sender where to send its completion signal.
operation_state auto state = connect(snd, rcv);

// The defining interface of an operation state is that it can be started:
start(state);


// NOTE: Operation state objects are not movable; therefore, an operation state
// object must be kept alive until its corresponding operation finishes.
```

CUSTOMIZATION

Sender algorithms are customizable

All of the sender algorithms defined in P2300 are customization points

We rely on the tag_invoke mechanism for defining customization points

A sender algorithm expression *algorithm*(snd, args…) is equivalent to:

1. tag_invoke(<*algorithm*>, get_completion_scheduler<cpo>(snd), snd, args...), if that expression is well-formed; otherwise
2. tag_invoke(<*algorithm*>, snd, args...), if that expression is well-formed; otherwise
3. a default implementation, if there exists a default implementation of the given algorithm.

CANCELLATION

Mechanisms to request cancellation of work that has already started

Fundamental support for certain algorithm and concurrency patterns, including examples such as:

- try multiple network servers; use whichever responds first; cancel the rest
- when one leg of `when_all(ops...)` fails, try to cancel incomplete operations
- apply a generic `timeout()` algorithm to a sender to have it cancelled after given time period

Builds upon `std::stop_token` mechanism in C++20, with `get_stop_token()` receiver query

Design details to follow in P2300r1

COMPARISON WITH P0443

KEY CHANGES IN P2300

Thorough revision of material from prior papers in response to feedback

Provides both a detailed design explanation and a complete specification

Consolidates necessary functionality from companion papers into a self-contained design

Elides certain functionality of P0443 as requested in previous design reviews

FOCUS ON CORE CONCEPTS

Removed functionality from this paper in response to design review of P0443

Removed polymorphic executor wrappers *(P0443r14, Section 2.4)*

Removed thread pool type *(P0443r14, Section 2.5)*

Removed generic property mechanism, and replaced with named query customization points

Removed executor as a distinct concept and defined `execute` to operate on schedulers

CONSOLIDATE FUNCTIONALITY

Functionality needed to write sender-based code was spread across multiple papers

Core specification of concepts and implementer interface in P0443r14

Sender adaptor for bulk execution from P2181r1

Fundamental algorithms for using senders in P1897r3

Approach to managing cancellation from P2175r0

CLARIFIED SEMANTICS

P0443 was unclear on several important semantic questions

Senders now advertise what scheduler, if any, their evaluation will complete on

Places of execution of user code are precisely defined

Semantics of variously qualified connect overloads are specified

Distinction between multi-shot and single-shot senders is made clear

NEW CAPABILITIES

P2300 also adds capabilities not present in P0443 and companion papers

A new `split` algorithm allows generic code to chain a sender with multiple successors

A `transfer` algorithm to explicitly control where senders complete

Fused algorithms (e.g., `transfer_just`, `transfer_when_all`) permit more efficient customizations

Implementors can now customize sender adaptors via completion scheduler of provided sender

Users now have a choice between strictly lazy & possibly eager versions of most sender algorithms

Further Questions?